Quick introduction into SAT/SMT solvers and symbolic execution

Dennis Yurichev <dennis(a)yurichev.com>

December 2015 – August 2016

Contents

1 Draft! **2**

2 Introduction **2**

3 SAT[1]-solvers **2**

3.1 CNF form . . . 2

3.2 Example: 2-bit adder . . . 2

3.2.1 MiniSat . . . 4

3.2.2 CryptoMiniSat . . . 5

4 SMT[2]-solvers **6**

4.1 School-level system of equations . . . 6

4.2 Another school-level system of equations . . . 8

4.3 Finding modular inverse . . . 8

4.4 Connection between SAT and SMT solvers . . . 9

4.5 Zebra puzzle (AKA Einstein puzzle) . . . 9

4.6 Sudoku puzzle . . . 12

4.6.1 The first idea . . . 13

4.6.2 The second idea . . . 17

4.6.3 Conclusion . . . 19

4.6.4 Homework . . . 19

4.6.5 Further reading . . . 19

4.6.6 Sudoku as a SAT problem . . . 19

4.7 Solving Problem Euler 31 - “Coin sums” . . . 19

4.8 Using Z3 theorem prover to prove equivalence of some weird alternative to XOR operation . . . 20

4.8.1 In SMT-LIB form . . . 22

4.8.2 Using universal quantifier . . . 22

4.8.3 How the equation works . . . 23

4.9 Dietz’s formula . . . 23

4.10 Cracking LCG[3] with Z3 SMT solver . . . 24

4.11 Simple hash function . . . 26

4.11.1 Manual decompiling . . . 26

4.11.2 Now let’s use the Z3 SMT solver . . . 30

4.12 Rockey dongle: finding unknown algorithm using only input/output pairs . . . 34

4.12.1 Conclusion . . . 39

4.12.2 Future work . . . 39

4.13 Solving pipe puzzle using Z3 SMT-solver . . . 39

4.13.1 Generation . . . 40

4.13.2 Solving . . . 42

5 Symbolic execution **45**

5.1 Symbolic computation . . . 45

5.1.1 Rational data type . . . 46

[1]Boolean satisfiability problem
[2]Satisfiability modulo theories
[3]Linear congruential generator

6 KLEE 46
6.1 School-level equation 46
6.2 Zebra puzzle 47
6.3 Sudoku 53
6.4 Unit test: HTML color 57
6.5 Unit test: strcmp() function 60
6.6 UNIX date/time 62
6.7 Inverse function for base64 decoder 67
6.8 CRC 70
6.8.1 Buffer alteration case #1 70
6.8.2 Buffer alteration case #2 71
6.8.3 Recovering input data for given CRC32 value of it 72
6.8.4 In comparison with other hashing algorithms 74
6.9 LZSS decompressor 74
6.10 Unit testing: simple expression evaluator (calculator) 77
6.11 Regular expressions 83
6.12 Exercise 84
7 Acronyms used 84

1 Draft!

This is very early draft, but still can be interesting for someone.

For news about updates, you may subscribe my twitter[4], facebook[5], or github repo[6].

2 Introduction

SAT/SMT solvers can be viewed as solvers of huge systems of equations. The difference is that SMT solvers takes systems in arbitrary format, while SAT solvers are limited to boolean equations in CNF form.

A lot of real world problems can be represented as problems of solving system of equations.

3 SAT-solvers

3.1 CNF form

CNF[7][8] is a *normal form*.

Any boolean expression can be converted to *normal form* and CNF is one of them. The CNF expression is bunch of clauses (sub-expressions) constisting of terms (variables), ORs and NOTs, all of which are then glueled together with AND into an expression. There is a way to memorize it: CNF is "AND of ORs" (or "product of sums") and DNF[9] is "OR of ANDs" (or "sum of products").

Example is: $(\neg A \lor B) \land (C \lor \neg D)$.

$\lor$ stands for OR (logical disjunction[10]), "+" sign is also sometimes used for OR.

$\land$ stands for AND (logical conjunction[11]). It is easy to memorize: $\land$ looks like "A" letter. "·" is also sometimes used for AND.

$\neg$ is negation (NOT).

3.2 Example: 2-bit adder

SAT-solver is merely a solver of huge boolean equations in CNF form. It just gives the answer, if there is a set of input variables which can satisfy CNF expression, and what input values must be.

Here is a 2-bit adder for example:

[4] https://twitter.com/yurichev
[5] https://www.facebook.com/dennis.yurichev.5
[6] https://github.com/dennis714/SAT_SMT_article
[7] Conjunctive normal form
[8] https://en.wikipedia.org/wiki/Conjunctive_normal_form
[9] Disjunctive normal form
[10] https://en.wikipedia.org/wiki/Logical_disjunction
[11] https://en.wikipedia.org/wiki/Logical_conjunction

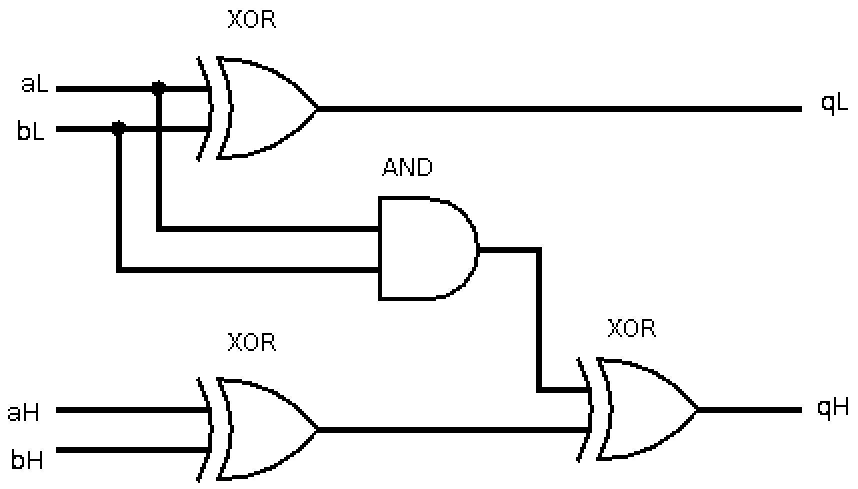


Figure 1: 2-bit adder schematics

The adder in its simplest form: it has no carry-in and carry-out, and it has 3 XOR gates and one AND gate. Let's try to figure out, which sets of inputs will force adder to set both two output bits? By doing quick memory calculation, we can see that there are 4 ways to do so: $0+3=3, 1+2=3, 2+1=3, 3+0=3$. Here is also truth table, with these rows highlighted:

	aH	aL	bH	bL	qH	qL
3+3 = 6 $\equiv$ 2 (mod 4)	1	1	1	1	1	0
3+2 = 5 $\equiv$ 1 (mod 4)	1	1	1	0	0	1
3+1 = 4 $\equiv$ 0 (mod 4)	1	1	0	1	0	0
3+0 = 3 $\equiv$ 3 (mod 4)	1	1	0	0	1	1
2+3 = 5 $\equiv$ 1 (mod 4)	1	0	1	1	0	1
2+2 = 4 $\equiv$ 0 (mod 4)	1	0	1	0	0	0
2+1 = 3 $\equiv$ 3 (mod 4)	1	0	0	1	1	1
2+0 = 2 $\equiv$ 2 (mod 4)	1	0	0	0	1	0
1+3 = 4 $\equiv$ 0 (mod 4)	0	1	1	1	0	0
1+2 = 3 $\equiv$ 3 (mod 4)	0	1	1	0	1	1
1+1 = 2 $\equiv$ 2 (mod 4)	0	1	0	1	1	0
1+0 = 1 $\equiv$ 1 (mod 4)	0	1	0	0	0	1
0+3 = 3 $\equiv$ 3 (mod 4)	0	0	1	1	1	1
0+2 = 2 $\equiv$ 2 (mod 4)	0	0	1	0	1	0
0+1 = 1 $\equiv$ 1 (mod 4)	0	0	0	1	0	1
0+0 = 0 $\equiv$ 0 (mod 4)	0	0	0	0	0	0

Let's find, what SAT-solver can say about it?
First, we should represent our 2-bit adder as CNF expression.
Using Wolfram Mathematica, I'll express 1-bit expressions for both adder outputs:

```
In[]:=AdderQ0[aL_,bL_]=Xor[aL,bL]
Out[]:=aL ⊻ bL
```

```
In[]:=AdderQ1[aL_,aH_,bL_,bH_]=Xor[And[aL,bL],Xor[aH,bH]]
Out[]:=aH ⊻ bH ⊻ (aL && bL)
```

We need such expression, where both parts will generate 1's. Let's use Wolfram Mathematica find all instances of such expression (I glueled both parts with And):

```
In[]:=Boole[SatisfiabilityInstances[And[AdderQ0[aL,bL],AdderQ1[aL,aH,bL,bH]],{aL,aH,bL,bH},4]]
Out[]:={1,1,0,0},{1,0,0,1},{0,1,1,0},{0,0,1,1}
```

Yes, indeed, Mathematica says, there are 4 inputs which will lead to the result we need. So, Mathematica can also be used as SAT solver.

Nevertheless, let's proceed to CNF form. Using Mathematica again, let's convert our expression to CNF form:

```
In[]:=cnf=BooleanConvert[And[AdderQ0[aL,bL],AdderQ1[aL,aH,bL,bH]],``CNF'']
Out[]:=(!aH || !bH) && (aH || bH) && (!aL || !bL) && (aL || bL)
```

Looks more complex. The reason of such verbosity is that CNF form doesn't allow XOR operations.

3.2.1 MiniSat

For the starters, we can try MiniSat[12]. The standard way to encode CNF expression for MiniSat is to enumerate all OR parts at each line. Also, MiniSat doesn't support variable names, just numbers. Let's enumerate our variables: 1 will be aH, 2 – aL, 3 – bH, 4 – bL.

Here is what I've got when I converted Mathematica expression to the MiniSat input file:

```
p cnf 4 4
-1 -3 0
1 3 0
-2 -4 0
2 4 0
```

Two 4's at the first lines are number of variables and number of clauses respectively. There are 4 lines then, each for each OR clause. Minus before variable number meaning that the variable is negated. Absence of minus – not negated. Zero at the end is just terminating zero, meaning end of the clause.

In other words, each line is OR-clause with optional negation, and the task of MiniSat is to find such set of input, which can satisfy all lines in the input file.

The following file I named as *adder.cnf* and now let's try MiniSat:

```
\$ minisat -verb=0 adder.cnf results.txt
SATISFIABLE
```

The results are in *results.txt* file:

```
SAT
-1 -2 3 4 0
```

This means, if the first two variables (aH and aL) will be *false*, and the last two variables (bH and bL) will be set to *true*, the whole CNF expression is satisfiable. Seems to be true: if bH and bL are the only inputs set to *true*, both resulting bits are also has *true* states.

Now how to get other instances? SAT-solvers, like SMT solvers, produce only one solution (or *instance*).

MiniSat uses PRNG and its initial seed can be set explicitely. I tried different values, but result is still the same. Nevertheless, CryptoMiniSat in this case was able to show all possible 4 instances, in chaotic order, though. So this is not very robust way.

Perhaps, the only known way is to negate solution clause and add it to the input expression. We've got `-1 -2 3 4`, now we can negate all variables in it (just toggle minuses: `1 2 -3 -4` and add it to the end of the input file:

```
p cnf 4 5
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
```

[12] http://minisat.se/MiniSat.html

Now we've got another result:

```
SAT
1 2 -3 -4 0
```

This means, aH and aL must be both *true* and bH and bL must be *false*, to satisfy the input expression. Let's negate this clause and add it again:

```
p cnf 4 6
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
```

The result is

```
SAT
-1 2 3 -4 0
```

aH=false, aL=true, bH=true, bL=false. This is also correct, according to our truth table.
Let's add it again:

```
p cnf 4 7
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
1 -2 -3 4 0
```

```
SAT
1 -2 -3 4 0
```

aH=true, aL=false, bH=false, bL=true. This is also correct.
This is fourth result. There are shouldn't be more. What if to add it?

```
p cnf 4 8
-1 -3 0
1 3 0
-2 -4 0
2 4 0
1 2 -3 -4
-1 -2 3 4 0
1 -2 -3 4 0
-1 2 3 -4 0
```

Now MiniSat just says "UNSATISFIABLE" without any additional information in the resulting file.
Our example is tiny, but MiniSat can work with huge CNF expressions.

3.2.2 CryptoMiniSat

XOR operation is absent in CNF form, but crucial in cryptographical algorithms. Simplest possible way to represent single XOR operation in CNF form is: $(\neg x \lor \neg y) \land (x \lor y)$ – not that small expression, though, many XOR operations in single expression can be optimized better.

One significant difference between MiniSat and CryptoMiniSat is that the latter supports CNF clauses with XOR operations instead of ORs, because CryptoMiniSat has aim to analyze some of crypto algorithms[13]. XOR clauses are handled by CryptoMiniSat in a special way without translating to OR clauses.

You need just to prepend a clause with "x" in .cnf file and OR clause is then treated as XOR clause by CryptoMiniSat. As of 2-bit adder, this smallest possible XOR-CNF expression can be used to find all inputs where both output adder bits are set:

[13] http://www.msoos.org/xor-clauses/

$(aH \oplus bH) \wedge (aL \oplus bL)$
This is 2-bit adder `.cnf` file for CryptoMiniSat:

```
p cnf 4 2
x1 3 0
x2 4 0
```

Now I run CryptoMiniSat with various random seeds...

```
\$ cryptominisat4 --verb 0 --random 0 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
\$ cryptominisat4 --verb 0 --random 1 XOR_adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
\$ cryptominisat4 --verb 0 --random 2 XOR_adder.cnf
s SATISFIABLE
v 1 -2 -3 4 0
\$ cryptominisat4 --verb 0 --random 3 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
\$ cryptominisat4 --verb 0 --random 4 XOR_adder.cnf
s SATISFIABLE
v -1 2 3 -4 0
\$ cryptominisat4 --verb 0 --random 5 XOR_adder.cnf
s SATISFIABLE
v -1 2 3 -4 0
\$ cryptominisat4 --verb 0 --random 6 XOR_adder.cnf
s SATISFIABLE
v -1 -2 3 4 0
\$ cryptominisat4 --verb 0 --random 7 XOR_adder.cnf
s SATISFIABLE
v 1 -2 -3 4 0
\$ cryptominisat4 --verb 0 --random 8 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
\$ cryptominisat4 --verb 0 --random 9 XOR_adder.cnf
s SATISFIABLE
v 1 2 -3 -4 0
```

Nevertheless, all 4 possible solutions are:

```
v -1 -2 3 4 0
v -1 2 3 -4 0
v 1 -2 -3 4 0
v 1 2 -3 -4 0
```

The same as reported by MiniSat.

4 SMT-solvers

4.1 School-level system of equations

I've got this school-level system of equations copypasted from Wikipedia[14]:

$$\begin{aligned} 3x + 2y - z &= 1 \\ 2x - 2y + 4z &= -2 \\ -x + \tfrac{1}{2}y - z &= 0 \end{aligned}$$

Will it be possible to solve it using Z3? Here it is:

[14] https://en.wikipedia.org/wiki/System_of_linear_equations

```
#!/usr/bin/python
from z3 import *

x = Real('x')
y = Real('y')
z = Real('z')
s = Solver()
s.add(3*x + 2*y - z == 1)
s.add(2*x - 2*y + 4*z == -2)
s.add(-x + 0.5*y - z == 0)
print s.check()
print s.model()
```

We see this after run:

```
sat
[z = -2, y = -2, x = 1]
```

If we change equation in some way so it will have no solution, s.check() will return "unsat".

I've used "Real" *sort* (some kind of data type in SMT-solvers) because the last expression has $\frac{1}{2}$, which is, of course, a real number. For the integer system of equations, "Int" *sort* would work fine.

Python (and other high-level PLs like C#) interface is highly popular, because it's practical, but in fact, there is a standard language for SMT-solvers named SMT-LIB[15].

Our example rewritten to it looks like this:

```
(declare-const x Real)
(declare-const y Real)
(declare-const z Real)
(assert (=(-(+(* 3 x) (* 2 y)) z) 1))
(assert (=(+(-(* 2 x) (* 2 y)) (* 4 z)) -2))
(assert (=(-(+ (- 0 x) (* 0.5 y)) z) 0))
(check-sat)
(get-model)
```

This language is very close to LISP, but hard to read for untrained eyes.

Now we run it:

```
\$ z3 -smt2 example.smt
sat
(model
  (define-fun z () Real
    (- 2.0))
  (define-fun y () Real
    (- 2.0))
  (define-fun x () Real
    1.0)
)
```

So when you look back to my Python code, you may feel that these 3 expressions could be executed. This is not true: Z3Py API offers overloaded operators, so expressions are constructed and passed into the guts of Z3 without any execution [16]. I would call it "embedded DSL[17]".

Same thing for Z3 C++ API, you may find there "operator+" declarations and many more [18].

Z3 APIs for Java, ML and .NET are also exist[19].

Z3Py tutorial: https://github.com/ericpony/z3py-tutorial.

Z3 tutorial which uses SMT-LIB language: http://rise4fun.com/Z3/tutorial/guide.

[15] http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.5-r2015-06-28.pdf
[16] https://github.com/Z3Prover/z3/blob/6e852762baf568af2aad1e35019fdf41189e4e12/src/api/python/z3.py
[17] Domain-specific language
[18] https://github.com/Z3Prover/z3/blob/6e852762baf568af2aad1e35019fdf41189e4e12/src/api/c%2B%2B/z3%2B%2B.h
[19] https://github.com/Z3Prover/z3/tree/6e852762baf568af2aad1e35019fdf41189e4e12/src/api

4.2 Another school-level system of equations

I've found this somewhere at Facebook:

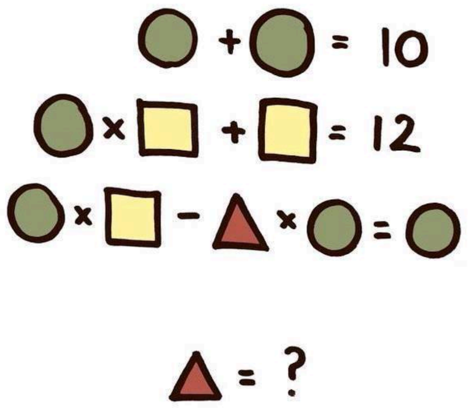


Figure 2: System of equations

It's that easy to solve it in Z3 SMT-solver:

```
#!/usr/bin/python
from z3 import *

circle, square, triangle = Ints('circle square triangle')
s = Solver()
s.add(circle+circle==10)
s.add(circle*square+square==12)
s.add(circle*square-triangle*circle==circle)
print s.check()
print s.model()
```

```
sat
[triangle = 1, square = 2, circle = 5]
```

4.3 Finding modular inverse

And now one useful equation.

It's possible to replace division operation (slow) by multiplication (faster). Read here or here about it.

In shortest possible terms, here is how it works.

If you need to divide integer by 13, this is the same as multiplication it by $\frac{1}{13}$. But you operate on integer numbers (**CPU!**[20] registers) and can't use ratios.

Given the fact that the division by 2^n on **CPU!** occurring at lightning speed, we can reformulate the problem as:

$$\frac{x}{13} = x \cdot \frac{1}{13} = x \cdot \frac{M}{2^n} = \frac{x \cdot M}{2^n}$$

... where M is *magic coefficient* and n is usually started at register width in bits (32 or 64).

But how to find *magic coefficient*? This equation is to be solved:

$$17 \cdot x = 1 + k \cdot 2^{64}$$

This is a little overkill to use SMT-solver to solve this equation [21], but still possible:

```
(declare-const x Int)
(declare-const k Int)
(assert (> x 0))
```

[20] **CPU!**

[21] https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm is usually used: http://rosettacode.org/wiki/Modular_inverse

```
(assert (> k 0))
(assert (=
                (* 17 x)
                (+ 1 (* k (^ 2 64)))
        )
)
(check-sat)
(get-model)
```

Important note: we use *Int* sort here, because this is Diophantine equation (i.e., only integer solutions are allowed). Resulting code will run on integer **CPU!** registers, after all.

Z3 gave this:

```
\$ z3 -smt2 1.smt
sat
(model
  (define-fun k () Int
    16)
  (define-fun x () Int
    17361641481138401521)
)
```

Let's check:

```
#include <stdint.h>
#include <stdio.h>

uint64_t div_by_17 (uint64_t x)
{
        __int128 result = (__int128)x * (__int128)17361641481138401521UL;
        return result >> 64 >> 4;
};

int main()
{
        printf ("%lld\n", 123456789/17);
        printf ("%lld\n", div_by_17(123456789));
};
```

... and it works.

GCC doesn't use 128-bit registers, it just exploit the fact that x86 multiplication instruction produces 128-bit result, lower 64-bit part in RAX register, and higher 64-bit part in RDX register. In standard C/C++ it's not possible to access higher part, so this hack is helpful.

This is what optimizing GCC 4.8.4 generates:

```
div_by_17:
        movabs  rax, -1085102592571150095
        mul     rdi
        mov     rax, rdx
        sar     rax, 4
        ret
```

Z3 may give "unsat" (for XXX) if there is no integer solution for the equation (but real solution may exist).

4.4 Connection between SAT and SMT solvers

Early SMT-solvers were frontends to SAT solvers, i.e., they translating input SMT expressions into CNF and feed SAT-solver with it. Conversion process is sometimes called "bit blasting". Some SMT-solvers still works in that way: STP uses MiniSAT or CryptoMiniSAT as backend SAT-solver. Some other SMT-solvers are more advanced (like Z3), so they use something even more complex.

4.5 Zebra puzzle (AKA Einstein puzzle)

Zebra puzzle is a popular puzzle, defined as follows:

1.There are five houses.
2.The Englishman lives in the red house.
3.The Spaniard owns the dog.
4.Coffee is drunk in the green house.
5.The Ukrainian drinks tea.
6.The green house is immediately to the right of the ivory house.
7.The Old Gold smoker owns snails.
8.Kools are smoked in the yellow house.
9.Milk is drunk in the middle house.
10.The Norwegian lives in the first house.
11.The man who smokes Chesterfields lives in the house next to the man with the fox.
12.Kools are smoked in the house next to the house where the horse is kept.
13.The Lucky Strike smoker drinks orange juice.
14.The Japanese smokes Parliaments.
15.The Norwegian lives next to the blue house.

Now, who drinks water? Who owns the zebra?

In the interest of clarity, it must be added that each of the five houses is painted a different color, and their inhabitants are of different national extractions, own different pets, drink different beverages and smoke different brands of American cigarets [sic]. One other thing: in statement 6, right means your right.

(https://en.wikipedia.org/wiki/Zebra_Puzzle)

It's a very good example of constraint satisfaction problem (CSP).

We would encode each entity as integer variable, representing number of house.

Then, to define that Englishman lives in red house, we will define this constraint: `Englishman == Red`, meaning that number of a house where Englishmen resides and where tea is drunk is the same.

To define that Norwegian lives next to the blue house, we don't realy know, if it is at left side of blue house or at right side, but we know that house numbers are different by just 1. So we will define this constraint: `Norwegian==Blue-1 OR Norwegian==Blue+1`.

We will also need to limit all house numbers, so they will be in range of 1..5.

We will also use `Distinct` to show that all various entities of the same type are all has different house numbers.

```
#!/usr/bin/env python
from z3 import *

Yellow, Blue, Red, Ivory, Green=Ints('Yellow Blue Red Ivory Green')
Norwegian, Ukrainian, Englishman, Spaniard, Japanese=Ints('Norwegian Ukrainian Englishman Spaniard
    Japanese')
Water, Tea, Milk, OrangeJuice, Coffee=Ints('Water Tea Milk OrangeJuice Coffee')
Kools, Chesterfield, OldGold, LuckyStrike, Parliament=Ints('Kools Chesterfield OldGold LuckyStrike
    Parliament')
Fox, Horse, Snails, Dog, Zebra=Ints('Fox Horse Snails Dog Zebra')

s = Solver()

# colors are distinct for all 5 houses:
s.add(Distinct(Yellow, Blue, Red, Ivory, Green))

# all nationalities are living in different houses:
s.add(Distinct(Norwegian, Ukrainian, Englishman, Spaniard, Japanese))

# so are beverages:
s.add(Distinct(Water, Tea, Milk, OrangeJuice, Coffee))

# so are cigarettes:
s.add(Distinct(Kools, Chesterfield, OldGold, LuckyStrike, Parliament))
```

```
# so are pets:
s.add(Distinct(Fox, Horse, Snails, Dog, Zebra))

# limits.
# adding two constraints at once (separated by comma) is the same
# as adding one And() constraint with two subconstraints
s.add(Yellow>=1, Yellow<=5)
s.add(Blue>=1, Blue<=5)
s.add(Red>=1, Red<=5)
s.add(Ivory>=1, Ivory<=5)
s.add(Green>=1, Green<=5)

s.add(Norwegian>=1, Norwegian<=5)
s.add(Ukrainian>=1, Ukrainian<=5)
s.add(Englishman>=1, Englishman<=5)
s.add(Spaniard>=1, Spaniard<=5)
s.add(Japanese>=1, Japanese<=5)

s.add(Water>=1, Water<=5)
s.add(Tea>=1, Tea<=5)
s.add(Milk>=1, Milk<=5)
s.add(OrangeJuice>=1, OrangeJuice<=5)
s.add(Coffee>=1, Coffee<=5)

s.add(Kools>=1, Kools<=5)
s.add(Chesterfield>=1, Chesterfield<=5)
s.add(OldGold>=1, OldGold<=5)
s.add(LuckyStrike>=1, LuckyStrike<=5)
s.add(Parliament>=1, Parliament<=5)

s.add(Fox>=1, Fox<=5)
s.add(Horse>=1, Horse<=5)
s.add(Snails>=1, Snails<=5)
s.add(Dog>=1, Dog<=5)
s.add(Zebra>=1, Zebra<=5)

# main constraints of the puzzle:

# 2.The Englishman lives in the red house.
s.add(Englishman==Red)

# 3.The Spaniard owns the dog.
s.add(Spaniard==Dog)

# 4.Coffee is drunk in the green house.
s.add(Coffee==Green)

# 5.The Ukrainian drinks tea.
s.add(Ukrainian==Tea)

# 6.The green house is immediately to the right of the ivory house.
s.add(Green==Ivory+1)

# 7.The Old Gold smoker owns snails.
s.add(OldGold==Snails)

# 8.Kools are smoked in the yellow house.
s.add(Kools==Yellow)

# 9.Milk is drunk in the middle house.
```

```
s.add(Milk==3) # i.e., 3rd house

# 10.The Norwegian lives in the first house.
s.add(Norwegian==1)

# 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
s.add(Or(Chesterfield==Fox+1, Chesterfield==Fox-1)) # left or right

# 12.Kools are smoked in the house next to the house where the horse is kept.
s.add(Or(Kools==Horse+1, Kools==Horse-1)) # left or right

# 13.The Lucky Strike smoker drinks orange juice.
s.add(LuckyStrike==OrangeJuice)

# 14.The Japanese smokes Parliaments.
s.add(Japanese==Parliament)

# 15.The Norwegian lives next to the blue house.
s.add(Or(Norwegian==Blue+1, Norwegian==Blue-1)) # left or right

r=s.check()
print r
if r==unsat:
    exit(0)
m=s.model()
print(m)
```

When we run it, we got correct result:

```
sat
[Snails = 3,
 Blue = 2,
 Ivory = 4,
 OrangeJuice = 4,
 Parliament = 5,
 Yellow = 1,
 Fox = 1,
 Zebra = 5,
 Horse = 2,
 Dog = 4,
 Tea = 2,
 Water = 1,
 Chesterfield = 2,
 Red = 3,
 Japanese = 5,
 LuckyStrike = 4,
 Norwegian = 1,
 Milk = 3,
 Kools = 1,
 OldGold = 3,
 Ukrainian = 2,
 Coffee = 5,
 Green = 5,
 Spaniard = 4,
 Englishman = 3]
```

4.6 Sudoku puzzle

Sudoku puzzle is a 9*9 grid with some cells filled, some are left to be found:

		5	3					
8							2	
	7			1		5		
4					5	3		
	1			7				6
		3	2				8	
	6		5					9
		4					3	
					9	7		

Unsolved Sudoku

Numbers of each row must be unique, i.e., must contain all 9 numbers in range of 1..9 without repetition. Same story for each column and also for each 3*3 square.

This puzzle is good candidate to try SMT solver on, because it's essentially an unsolved system of equations.

4.6.1 The first idea

The only thing we must solve is that how to determine in one expression, if the input 9 variables has all 9 unique numbers? They are not ordered or sorted, after all.

From the school-level mathematics, we can devise this idea:

$$\underbrace{10^{i_1} + 10^{i_2} + \cdots + 10^{i_9}}_{9} = 1111111110 \quad (1)$$

Take each input variable, calculate 10^i and sum them all. If all input values are unique, each will be settled at its own place. Even more than that: there will be no holes, i.e., no skipped values. So, in case of Sudoku, 1111111110 number will be final result, indicating that all 9 input variables are unique, in range of 1..9.

Exponentiation is heavy operation, can we use binary operations? Yes, just replace 10 with 2:

$$\underbrace{2^{i_1} + 2^{i_2} + \cdots + 2^{i_9}}_{9} = 1111111110_2 \quad (2)$$

The effect is just the same, but the final value is in base 2 instead of 10.

Now a working example:

```
import sys
from z3 import *

"""
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""
```

```
s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[BitVec('cell%d%d' % (r, c), 16) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==BitVecVal(int(i),16))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

one=BitVecVal(1,16)
mask=BitVecVal(0b1111111110,16)

# for all 9 rows
for r in range(9):
        s.add(((one<<cells[r][0]) +
                (one<<cells[r][1]) +
                (one<<cells[r][2]) +
                (one<<cells[r][3]) +
                (one<<cells[r][4]) +
                (one<<cells[r][5]) +
                (one<<cells[r][6]) +
                (one<<cells[r][7]) +
                (one<<cells[r][8]))==mask)

# for all 9 columns
for c in range(9):
        s.add(((one<<cells[0][c]) +
                (one<<cells[1][c]) +
                (one<<cells[2][c]) +
                (one<<cells[3][c]) +
                (one<<cells[4][c]) +
                (one<<cells[5][c]) +
                (one<<cells[6][c]) +
                (one<<cells[7][c]) +
                (one<<cells[8][c]))==mask)

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add((one<<cells[r+0][c+0]) +
                                (one<<cells[r+0][c+1]) +
                                (one<<cells[r+0][c+2]) +
                                (one<<cells[r+1][c+0]) +
                                (one<<cells[r+1][c+1]) +
                                (one<<cells[r+1][c+2]) +
                                (one<<cells[r+2][c+0]) +
                                (one<<cells[r+2][c+1]) +
                                (one<<cells[r+2][c+2])==mask)
```

```
#print s.check()
s.check()
#print s.model()
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ""
```

(https://github.com/dennis714/SAT_SMT_article/blob/master/SMT/sudoku_plus.py)

```
\$ time python sudoku_plus.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m11.717s
user    0m10.896s
sys     0m0.068s
```

Even more, we can replace summing operation to logical OR:

$$\underbrace{2^{i_1} \vee 2^{i_2} \vee \cdots \vee 2^{i_9}}_{9} = 1111111110_2 \tag{3}$$

```
import sys
from z3 import *

"""
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[BitVec('cell%d%d' % (r, c), 16) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."
```

```
# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==BitVecVal(int(i),16))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

one=BitVecVal(1,16)
mask=BitVecVal(0b1111111110,16)

# for all 9 rows
for r in range(9):
        s.add(((one<<cells[r][0]) |
                (one<<cells[r][1]) |
                (one<<cells[r][2]) |
                (one<<cells[r][3]) |
                (one<<cells[r][4]) |
                (one<<cells[r][5]) |
                (one<<cells[r][6]) |
                (one<<cells[r][7]) |
                (one<<cells[r][8]))==mask)

# for all 9 columns
for c in range(9):
        s.add(((one<<cells[0][c]) |
                (one<<cells[1][c]) |
                (one<<cells[2][c]) |
                (one<<cells[3][c]) |
                (one<<cells[4][c]) |
                (one<<cells[5][c]) |
                (one<<cells[6][c]) |
                (one<<cells[7][c]) |
                (one<<cells[8][c]))==mask)

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add(one<<cells[r+0][c+0] |
                                one<<cells[r+0][c+1] |
                                one<<cells[r+0][c+2] |
                                one<<cells[r+1][c+0] |
                                one<<cells[r+1][c+1] |
                                one<<cells[r+1][c+2] |
                                one<<cells[r+2][c+0] |
                                one<<cells[r+2][c+1] |
                                one<<cells[r+2][c+2]==mask)

#print s.check()
s.check()
#print s.model()
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ""
```

(https://github.com/dennis714/SAT_SMT_article/blob/master/SMT/sudoku_or.py)

Now it works much faster. Z3 handles OR operation over bit vectors better than addition?

```
\$ time python sudoku_or.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m1.429s
user    0m1.393s
sys     0m0.036s
```

The puzzle I used as example is dubbed as one of the hardest known [22] (well, for humans). It took 1̃.4 seconds on my Intel Core i3-3110M 2.4GHz notebook to solve it.

4.6.2 The second idea

My first approach is far from effective, I did what first came to my mind and worked. Another approach is to use `distinct` command from SMTLIB, which tells Z3 that some variables (of unspecific number) must be distinct (or unique). This command is also available in Z3 Python interface.

I've rewritten my first Sudoku solver, now it operates over *Int sort*, it has `distinct` commands instead of bit operations, and now also other constaint added: each cell value must be in 1..9 range, because, otherwise, Z3 will offer (although correct) solution with too big and/or negative numbers.

```
import sys
from z3 import *

"""
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
"""

s=Solver()

# using Python list comprehension, construct array of arrays of BitVec instances:
cells=[[Int('cell%d%d' % (r, c)) for c in range(9)] for r in range(9)]

# http://www.norvig.com/sudoku.html
# http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
puzzle="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97.."
```

[22] http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294

```

# process text line:
current_column=0
current_row=0
for i in puzzle:
        if i!='.':
                s.add(cells[current_row][current_column]==int(i))
        current_column=current_column+1
        if current_column==9:
                current_column=0
                current_row=current_row+1

# this is important, because otherwise, Z3 will report correct solutions with too big and/or negative
    numbers in cells
for r in range(9):
        for c in range(9):
                s.add(cells[r][c]>=1)
                s.add(cells[r][c]<=9)

# for all 9 rows
for r in range(9):
        s.add(Distinct(cells[r][0],
                cells[r][1],
                cells[r][2],
                cells[r][3],
                cells[r][4],
                cells[r][5],
                cells[r][6],
                cells[r][7],
                cells[r][8]))

# for all 9 columns
for c in range(9):
        s.add(Distinct(cells[0][c],
                cells[1][c],
                cells[2][c],
                cells[3][c],
                cells[4][c],
                cells[5][c],
                cells[6][c],
                cells[7][c],
                cells[8][c]))

# enumerate all 9 squares
for r in range(0, 9, 3):
        for c in range(0, 9, 3):
                # add constraints for each 3*3 square:
                        s.add(Distinct(cells[r+0][c+0],
                                cells[r+0][c+1],
                                cells[r+0][c+2],
                                cells[r+1][c+0],
                                cells[r+1][c+1],
                                cells[r+1][c+2],
                                cells[r+2][c+0],
                                cells[r+2][c+1],
                                cells[r+2][c+2]))

#print s.check()
s.check()
#print s.model()
```

```
m=s.model()

for r in range(9):
        for c in range(9):
                sys.stdout.write (str(m[cells[r][c]])+" ")
        print ""
```

(https://github.com/dennis714/SAT_SMT_article/blob/master/SMT/sudoku2.py)

```
\$ time python sudoku2.py
1 4 5 3 2 7 6 9 8
8 3 9 6 5 4 1 2 7
6 7 2 9 1 8 5 4 3
4 9 6 1 8 5 3 7 2
2 1 8 4 7 3 9 5 6
7 5 3 2 9 6 4 8 1
3 6 7 5 4 2 8 1 9
9 8 4 7 6 1 2 3 5
5 2 1 8 3 9 7 6 4

real    0m0.382s
user    0m0.346s
sys     0m0.036s
```

That's much faster.

4.6.3 Conclusion

The awesomeness of SMT-solvers is that our Sudoku solver has nothing else, we have just defined relationships between variables (cells).

4.6.4 Homework

As it seems, true Sudoku puzzle is the one which has only one solution. The piece of code I've included in this article shows only the first one. Using the method described earlier (4.7, also called "model counting"), try to find more solutions, or prove that the solution you have just found is the only one possible.

4.6.5 Further reading

http://www.norvig.com/sudoku.html

4.6.6 Sudoku as a SAT problem

It's also possible to represend Sudoku puzzle as a huge CNF equation and use SAT-solver to find solution, but it's just trickier.

Some articles about it: *Building a Sudoku Solver with SAT*[23], Tjark Weber, *A SAT-based Sudoku Solver*[24], Ines Lynce, Joel Ouaknine, *Sudoku as a SAT Problem*[25], Gihwon Kwon, Himanshu Jain, *Optimized CNF Encoding for Sudoku Puzzles*[26].

SMT-solver can also use SAT-solver in its core, so it does all mundane translating work. As a "compiler", it may not do this in the most efficient way, though.

4.7 Solving Problem Euler 31 - "Coin sums"

(This post was first published in my blog[27] at 10-May-2013).

[23] http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-005-elements-of-software-construction-fall-2011/assignments/MIT6_005F11_ps4.pdf
[24] https://www.lri.fr/~conchon/mpri/weber.pdf
[25] http://sat.inesc-id.pt/~ines/publications/aimath06.pdf
[26] http://www.cs.cmu.edu/~hjain/papers/sudoku-as-SAT.pdf
[27] http://dennisyurichev.blogspot.de/2013/05/in-england-currency-is-made-up-of-pound.html

In England the currency is made up of pound, £, and pence, p, and there are eight coins in general circulation:
1p, 2p, 5p, 10p, 20p, 50p, £1 (100p) and £2 (200p). It is possible to make £2 in the following way:
1£1 + 150p + 220p + 15p + 12p + 31p How many different ways can £2 be made using any number of coins?

(Problem Euler 31 - Coin sums)

Using Z3 SMT solver for solving this is overkill, and also slow, but nevertheless, it works, showing all possible solutions as well. The piece of code for blocking already found solution and search for next, and thus, counting all solutions, was taken from Stack Overflow answer [28]. This is also called "model counting". Constraints like "a>=0" must be present, because Z3 solver will search for solutions with negative numbers.

```
#!/usr/bin/python

from z3 import *

a,b,c,d,e,f,g,h = Ints('a b c d e f g h')
s = Solver()
s.add(1*a + 2*b + 5*c + 10*d + 20*e + 50*f + 100*g + 200*h == 200,
   a>=0, b>=0, c>=0, d>=0, e>=0, f>=0, g>=0, h>=0)
result=[]

while True:
    if s.check() == sat:
        m = s.model()
        print m
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
        for d in m:
            # d is a declaration
            if d.arity() > 0:
                raise Z3Exception("uninterpreted functions are not suppported")
            # create a constant from declaration
            c=d()
            #print c, m[d]
            if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
                raise Z3Exception("arrays and uninterpreted sorts are not supported")
            block.append(c != m[d])
        #print "new constraint:",block
        s.add(Or(block))
    else:
        print len(result)
        break
```

Works very slow, and this is what it produces:

```
[h = 0, g = 0, f = 0, e = 0, d = 0, c = 0, b = 0, a = 200]
[f = 1, b = 5, a = 0, d = 1, g = 1, h = 0, c = 2, e = 1]
[f = 0, b = 1, a = 153, d = 0, g = 0, h = 0, c = 1, e = 2]
...
[f = 0, b = 31, a = 33, d = 2, g = 0, h = 0, c = 17, e = 0]
[f = 0, b = 30, a = 35, d = 2, g = 0, h = 0, c = 17, e = 0]
[f = 0, b = 5, a = 50, d = 2, g = 0, h = 0, c = 24, e = 0]
```

73682 results in total.

4.8 Using Z3 theorem prover to prove equivalence of some weird alternative to XOR operation

(The article was first published in my blog at April 2015: http://blog.yurichev.com/node/86).

[28]http://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation, another question: http://stackoverflow.com/questions/13395391/z3-finding-all-satisfying-models

There is a ”A Hacker’s Assistant” program[29] (*Aha!*) written by Henry Warren, who is also the author of the great ”Hacker’s Delight” book.

The *Aha!* program is essentially *superoptimizer*[30], which blindly brute-force a list of some generic RISC CPU instructions to achieve shortest possible (and jumpless or branch-free) CPU code sequence for desired operation. For example, *Aha!* can find jumpless version of abs() function easily.

Compiler developers use superoptimization to find shortest possible (and/or jumpless) code, but I tried to do otherwise – to find longest code for some basic operation. I tried *Aha!* to find equivalent of basic XOR operation without usage of the actual XOR instruction, and the most bizarre example *Aha!* gave is:

```
Found a 4-operation program:
   add   r1,ry,rx
   and   r2,ry,rx
   mul   r3,r2,-2
   add   r4,r3,r1
   Expr: (((y & x)*-2) + (y + x))
```

And it’s hard to say, why/where we can use it, maybe for obfuscation, I’m not sure. I would call this *suboptimization* (as opposed to *superoptimization*). Or maybe *superdeoptimization*.

But my another question was also, is it possible to prove that this is correct formula at all? The *Aha!* checking some intput/output values against XOR operation, but of course, not all the possible values. It is 32-bit code, so it may take very long time to try all possible 32-bit inputs to test it.

We can try Z3 theorem prover for the job. It’s called “prover”, after all.

So I wrote this:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 32)
y = BitVec('y', 32)
output = BitVec('output', 32)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFE) + (y + x)!=output)
print s.check()
```

In plain English language, this means ”are there any case for x and y where $x \oplus y$ doesn’t equals to $((y\&x) * -2) + (y + x)$?” ... and Z3 prints ”unsat”, meaning, it can’t find any counterexample to the equation. So this *Aha!* output is proved to be working just like XOR operation.

Oh, I also tried to extend the formula to 64-bit:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFE) + (y + x)!=output)
print s.check()
```

Nope, now it says ”sat”, meaning, Z3 found at least one counterexample. Oops, it’s because I forgot to extend -2 number to 64-bit value:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
```

[29] http://www.hackersdelight.org/
[30] http://en.wikipedia.org/wiki/Superoptimization

```
s = Solver()
s.add(x^y==output)
s.add(((y & x)*0xFFFFFFFFFFFFFFFE) + (y + x)!=output)
print s.check()
```

Now it says ”unsat”, so the formula given by *Aha!* works for 64-bit code as well.

4.8.1 In SMT-LIB form

Now we can rephrase our equation to more suitable form: $(x + y - ((x\&y) << 1))$. It also works well is Z3:

```
#!/usr/bin/python
from z3 import *

x = BitVec('x', 64)
y = BitVec('y', 64)
output = BitVec('output', 64)
s = Solver()
s.add(x^y==output)
s.add((x + y - ((x & y)<<1)) != output)
print s.check()
```

Here is how to define it in SMT-LIB way:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (not
                (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64)))
                        (bvxor x y)
                )
        )
)
(check-sat)
```

4.8.2 Using universal quantifier

Z3 has universal quantifier `exists`, which defines a constraint, which is true if at least one set of variables satistfied underlying condition:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (exists ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (not (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64))
                        )
                        (bvxor x y)
                ))
        )
)
(check-sat)
```

It returns “unsat”, meaning, Z3 couldn’t find any counterexample of the equation, i.e., it’s not exist.

This is also known as $\exists$ in mathematical logic lingo.

Z3 has also universal quantifier `forall`, which defines a constraint, where all possible values for the equation must be true. So we can rewrite our SMT example as:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (forall ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (=
                        (bvsub
                                (bvadd x y)
                                (bvshl (bvand x y) (_ bv1 64))
                        )
                        (bvxor x y)
                )
        )
)
(check-sat)
```

It returns `sat`, meaning, the equation is correct for all possible 64-bit `x` and `y` values, like them all were checked. Mathematically speaking: $\forall n \in \mathbb{N}\ (x \oplus y = (x + y - ((x \& y) << 1)))$ [31]

4.8.3 How the equation works

First of all, binary addition can be viewed as binary XORing with carrying (3.2). Here is an example: let's add 2 (10b) and 2 (10b). XORing these two values resulting 0, but there is a carry generated during addition of two second bits. That carry bit is propagated further and settles at the place of the 3rd bit: 100b. 4 (100b) is hence a final result of addition.

If the carry bits are not generated during addition, the addition operation is merely XORing. For example, let's add 1 (1b) and 2 (10b). $1 + 2$ equals to 3, but $1 \oplus 2$ is also 3.

If the addition is XORing plus carry generation and application, we should eliminate effect of carrying somehow here. The first part of the equation $(x+y)$ is addition, the second $((x\&y) << 1)$ is just calculation of every carry bit which was used during addition. Subtracting removes carry bits from the result of addition, so the only XOR effect is left then.

It's hard to say how Z3 proves this: maybe it just simplifies the equation down to single XOR using simple boolean algebra rewriting rules?

4.9 Dietz's formula

Now something practical and useful.

One of the impressive examples of *Aha!* work is finding of Dietz's formula[32], which is the code of computing average number of two numbers without overflow (which is important if you want to find average number of numbers like 0xFFFFFF00 and so on, using 32-bit registers). Taking this in input:

```
int userfun(int x, int y) {     // To find Dietz's formula for
                                // the floor-average of two
                                // unsigned integers.
   return ((unsigned long long)x + (unsigned long long)y) >> 1;
}
```

... the *Aha!* gives this:

```
Found a 4-operation program:
   and   r1,ry,rx
   xor   r2,ry,rx
   shrs  r3,r2,1
   add   r4,r3,r1
   Expr: (((y ^ x) >>s 1) + (y & x))
```

And it works correctly[33]. But how to prove it?

We will place Dietz's formula on the left side of equation and $x + y/2$ (or $x + y >> 1$) on the right side:

[31] $\forall$ means *equation must be true for all possible values*, which are choosen from natural numbers ($\mathbb{N}$).

[32] http://aggregate.org/MAGIC/#Average%20of%20Integers

[33] For those who interesting how it works, its mechanics is closely related to the weird XOR alternative we just saw. That's why I placed these two pieces of text one after another.

$$\forall n \in 0..2^{64}-1\ ((x\&y) + (x \oplus y) >> 1 = x + y >> 1)$$

One important thing is that we can't operate on 64-bit values on right side, because result will overflow. So we will zero extend inputs on right side by 1 bit (in other words, we will just 1 zero bit before each variable). The result of Dietz's formula will also be extended by 1 bit. Hence, both sides of the equation will have a width of 64 bits:

```
(declare-const x (_ BitVec 64))
(declare-const y (_ BitVec 64))
(assert
        (forall ((x (_ BitVec 64)) (y (_ BitVec 64)))
                (=
                        ((_ zero_extend 1)
                                (bvadd
                                        (bvand x y)
                                        (bvlshr (bvxor x y) (_ bv1 64))
                                )
                        )
                        (bvlshr
                                (bvadd ((_ zero_extend 1) x) ((_ zero_extend 1) y))
                                (_ bv1 65)
                        )
                )
        )
)
(check-sat)
```

Z3 says "sat".

65 bits are enough, because the result of addition of two biggest 64-bit values has width of 65 bits: `0xFF...FF + 0xFF...FF = 0x1FF...FE`.

As in previous example about XOR equivalent, `(not (= ... ))` and `exists` can also be used here instead of `forall`.

4.10 Cracking LCG with Z3 SMT solver

This part is first appeared in my blog in June 2015 at http://yurichev.com/blog/modulo/.

There are well-known weaknesses of LCG (1, 2, 3), but let's see, if it would be possible to crack it straightforwardly, without any special knowledge. We would define all relations between LCG states in term of Z3 SMT solver. (I first made attempt to do it using FindInstance in Wolfram Mathematica, but failed, perhaps, made a mistake somewhere). Here is a test progam:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

int main()
{
        int i;

        srand(time(NULL));

        for (i=0; i<10; i++)
                printf ("%d\n", rand()%100);
};
```

It is intended to print 10 pseudorandom numbers in 0..99 range. So it does:

```
37
29
74
95
98
40
```

```
23
58
61
17
```

Let's say we are observing only 8 of these numbers (from 29 to 61) and we need to predict next one (17) and/or previous one (37). The program is compiled using MSVC 2013 (I choose it because its LCG is simpler than that in Glib):

```
.text:0040112E rand             proc near
.text:0040112E                  call    __getptd
.text:00401133                  imul    ecx, [eax+0x14], 214013
.text:0040113A                  add     ecx, 2531011
.text:00401140                  mov     [eax+14h], ecx
.text:00401143                  shr     ecx, 16
.text:00401146                  and     ecx, 7FFFh
.text:0040114C                  mov     eax, ecx
.text:0040114E                  retn
.text:0040114E rand             endp
```

This is very simple LCG, but the result is not clipped state, but it's rather shifted by 16 bits. Let's define LCG in Z3:

```
#!/usr/bin/python
from z3 import *

output_prev = BitVec('output_prev', 32)
state1 = BitVec('state1', 32)
state2 = BitVec('state2', 32)
state3 = BitVec('state3', 32)
state4 = BitVec('state4', 32)
state5 = BitVec('state5', 32)
state6 = BitVec('state6', 32)
state7 = BitVec('state7', 32)
state8 = BitVec('state8', 32)
state9 = BitVec('state9', 32)
state10 = BitVec('state10', 32)
output_next = BitVec('output_next', 32)

s = Solver()

s.add(state2 == state1*214013+2531011)
s.add(state3 == state2*214013+2531011)
s.add(state4 == state3*214013+2531011)
s.add(state5 == state4*214013+2531011)
s.add(state6 == state5*214013+2531011)
s.add(state7 == state6*214013+2531011)
s.add(state8 == state7*214013+2531011)
s.add(state9 == state8*214013+2531011)
s.add(state10 == state9*214013+2531011)

s.add(output_prev==URem((state1>>16)&0x7FFF,100))
s.add(URem((state2>>16)&0x7FFF,100)==29)
s.add(URem((state3>>16)&0x7FFF,100)==74)
s.add(URem((state4>>16)&0x7FFF,100)==95)
s.add(URem((state5>>16)&0x7FFF,100)==98)
s.add(URem((state6>>16)&0x7FFF,100)==40)
s.add(URem((state7>>16)&0x7FFF,100)==23)
s.add(URem((state8>>16)&0x7FFF,100)==58)
s.add(URem((state9>>16)&0x7FFF,100)==61)
s.add(output_next==URem((state10>>16)&0x7FFF,100))

print(s.check())
print(s.model())
```

URem states for *unsigned remainder*. It works for some time and gave us correct result!

```
sat
[state3 = 2276903645,
 state4 = 1467740716,
 state5 = 3163191359,
 state7 = 4108542129,
 state8 = 2839445680,
 state2 = 998088354,
 state6 = 4214551046,
 state1 = 1791599627,
 state9 = 548002995,
 output_next = 17,
 output_prev = 37,
 state10 = 1390515370]
```

I added 10 states to be sure result will be correct. It may be not if you supply lesser amount of PRNG numbers.

That is the reason why LCG is not suitable for any security-related task. This is why cryptographically secure pseudorandom number generators exist: they are designed to be protected against such simple attack. Well, at least if NSA is not involved.

As far, as I can understand, security tokens like RSA SecurID can be viewed just as CPRNG[34] with a secret seed. It shows new pseudorandom number each minute, and the server can predict it, because it knows the seed. Imagine if such token would implement LCG – it would be much easier to break!

4.11 Simple hash function

(This piece of text was initially added to my "Reverse Engineering for Beginners" book (`beginners.re`) at March 2014) [35].

Amateur cryptography is usually (unintentionally) very weak and can be broken easily—for cryptographers, of course.

But let's pretend we are not among these crypto-professionals.

Here is one-way hash function, that converted a 64-bit value to another and we need to try to reverse its flow back.

4.11.1 Manual decompiling

Here its assembly language listing in IDA:

```
sub_401510      proc near
                ; ECX = input
                mov     rdx, 5D7E0D1F2E0F1F84h
                mov     rax, rcx        ; input
                imul    rax, rdx
                mov     rdx, 388D76AEE8CB1500h
                mov     ecx, eax
                and     ecx, 0Fh
                ror     rax, cl
                xor     rax, rdx
                mov     rdx, 0D2E9EE7E83C4285Bh
                mov     ecx, eax
                and     ecx, 0Fh
                rol     rax, cl
                lea     r8, [rax+rdx]
                mov     rdx, 8888888888888889h
                mov     rax, r8
                mul     rdx
                shr     rdx, 5
                mov     rax, rdx
                lea     rcx, [r8+rdx*4]
                shl     rax, 6
                sub     rcx, rax
```

[34] Cryptographically Secure Pseudorandom Number Generator

[35] This example was also used by Murphy Berzish in his lecture about SAT and SMT: `http://mirror.csclub.uwaterloo.ca/csclub/mtrberzi-sat-smt-slides.pdf`, `http://mirror.csclub.uwaterloo.ca/csclub/mtrberzi-sat-smt.mp4`

```
                mov     rax, r8
                rol     rax, cl
                ; EAX = output
                retn
sub_401510      endp
```

The example was compiled by GCC, so the first argument is passed in ECX.

If you don't have Hex-Rays, or if you distrust to it, you can try to reverse this code manually. One method is to represent the CPU registers as local C variables and replace each instruction by a one-line equivalent expression, like:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;
        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        rdx=rdx>>5;
        rax=rdx;
        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

If you are careful enough, this code can be compiled and will even work in the same way as the original.

Then, we are going to rewrite it gradually, keeping in mind all registers usage. Attention and focus is very important here—any tiny typo may ruin all your work!

Here is the first step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=r8+rdx*4;
        rax=rax<<6;
        rcx=rcx-rax;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

Next step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rdx=0x8888888888888889;
        rax=r8;
        rax*=rdx;
        // RDX here is a high part of multiplication result
        rdx=rdx>>5;
        // RDX here is division result!
        rax=rdx;

        rcx=(r8+rdx*4)-(rax<<6);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

We can spot the division using multiplication. Indeed, let’s calculate the divider in Wolfram Mathematica:

Listing 1: Wolfram Mathematica

```
In[1]:=N[2^(64 + 5)/16^^8888888888888889]
Out[1]:=60.
```

We get this:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        ecx=input;

        rdx=0x5D7E0D1F2E0F1F84;
        rax=rcx;
        rax*=rdx;
        rdx=0x388D76AEE8CB1500;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=rdx;
        rdx=0xD2E9EE7E83C4285B;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+rdx;

        rax=rdx=r8/60;

        rcx=(r8+rax*4)-(rax*64);
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

One more step:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        rcx=r8-(r8/60)*60;
        rax=r8
        rax=_lrotl (rax, rcx&0xFF); // rotate left
        return rax;
};
```

By simple reducing, we finally see that it's calculating the remainder, not the quotient:

```
uint64_t f(uint64_t input)
{
        uint64_t rax, rbx, rcx, rdx, r8;

        rax=input;
        rax*=0x5D7E0D1F2E0F1F84;
        rax=_lrotr(rax, rax&0xF); // rotate right
        rax^=0x388D76AEE8CB1500;
        rax=_lrotl(rax, rax&0xF); // rotate left
        r8=rax+0xD2E9EE7E83C4285B;

        return _lrotl (r8, r8 % 60); // rotate left
};
```

We end up with this fancy formatted source-code:

```
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <string.h>
#include <intrin.h>

#define C1 0x5D7E0D1F2E0F1F84
#define C2 0x388D76AEE8CB1500
#define C3 0xD2E9EE7E83C4285B
```

```
uint64_t hash(uint64_t v)
{
        v*=C1;
        v=_lrotr(v, v&0xF); // rotate right
        v^=C2;
        v=_lrotl(v, v&0xF); // rotate left
        v+=C3;
        v=_lrotl(v, v % 60); // rotate left
        return v;
};

int main()
{
        printf ("%llu\n", hash(...));
};
```

Since we are not cryptoanalysts we can't find an easy way to generate the input value for some specific output value. The rotate instruction's coefficients look frightening—it's a warranty that the function is not bijective, it has collisions, or, speaking more simply, many inputs may be possible for one output.

Brute-force is not solution because values are 64-bit ones, that's beyond reality.

4.11.2 Now let's use the Z3 SMT solver

Still, without any special cryptographic knowledge, we may try to break this algorithm using the excellent SMT solver from Microsoft Research named Z3[36]. It is in fact theorem prover, but we are going to use it as SMT solver. Simply said, we can think about it as a system capable of solving huge equation systems.

Here is the Python source code:

```
 1  from z3 import *
 2
 3  C1=0x5D7E0D1F2E0F1F84
 4  C2=0x388D76AEE8CB1500
 5  C3=0xD2E9EE7E83C4285B
 6
 7  inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
 8
 9  s = Solver()
10  s.add(i1==inp*C1)
11  s.add(i2==RotateRight (i1, i1 & 0xF))
12  s.add(i3==i2 ^ C2)
13  s.add(i4==RotateLeft(i3, i3 & 0xF))
14  s.add(i5==i4 + C3)
15  s.add(outp==RotateLeft (i5, URem(i5, 60)))
16
17  s.add(outp==10816636949158156260)
18
19  print s.check()
20  m=s.model()
21  print m
22  print (" inp=0x%X" % m[inp].as_long())
23  print ("outp=0x%X" % m[outp].as_long())
```

This is going to be our first solver.

We see the variable definitions on line 7. These are just 64-bit variables. `i1..i6` are intermediate variables, representing the values in the registers between instruction executions.

Then we add the so-called constraints on lines 10..15. The last constraint at 17 is the most important one: we are going to try to find an input value for which our algorithm will produce 10816636949158156260.

Essentially, the SMT-solver searches for (any) values that satisfies all constraints.

RotateRight, RotateLeft, URem—are functions from the Z3 Python API, not related to Python language.

Then we run it:

[36] http://go.yurichev.com/17314

```
...>python.exe 1.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 1364123924608584563,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x12EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

“sat” mean “satisfiable”, i.e., the solver was able to find at least one solution. The solution is printed in the square brackets. The last two lines are the input/output pair in hexadecimal form. Yes, indeed, if we run our function with `0x12EE577B63E80B73` as input, the algorithm will produce the value we were looking for.

But, as we noticed before, the function we work with is not bijective, so there may be other correct input values. The Z3 SMT solver is not capable of producing more than one result, but let’s hack our example slightly, by adding line 19, which implies “look for any other results than this”:

```
 1 from z3 import *
 2 
 3 C1=0x5D7E0D1F2E0F1F84
 4 C2=0x388D76AEE8CB1500
 5 C3=0xD2E9EE7E83C4285B
 6 
 7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
 8 
 9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16 
17 s.add(outp==10816636949158156260)
18 
19 s.add(inp!=0x12EE577B63E80B73)
20 
21 print s.check()
22 m=s.model()
23 print m
24 print (" inp=0x%X" % m[inp].as_long())
25 print ("outp=0x%X" % m[outp].as_long())
```

Indeed, it finds another correct result:

```
...>python.exe 2.py
sat
[i1 = 3959740824832824396,
 i3 = 8957124831728646493,
 i5 = 10816636949158156260,
 inp = 10587495961463360371,
 outp = 10816636949158156260,
 i4 = 14065440378185297801,
 i2 = 4954926323707358301]
 inp=0x92EE577B63E80B73
outp=0x961C69FF0AEFD7E4
```

This can be automated. Each found result can be added as a constraint and then the next result will be searched for. Here is a slightly more sophisticated example:

```
from z3 import *

C1=0x5D7E0D1F2E0F1F84
C2=0x388D76AEE8CB1500
C3=0xD2E9EE7E83C4285B

inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)

s = Solver()
s.add(i1==inp*C1)
s.add(i2==RotateRight (i1, i1 & 0xF))
s.add(i3==i2 ^ C2)
s.add(i4==RotateLeft(i3, i3 & 0xF))
s.add(i5==i4 + C3)
s.add(outp==RotateLeft (i5, URem(i5, 60)))

s.add(outp==10816636949158156260)

# copypasted from http://stackoverflow.com/questions/11867611/z3py-checking-all-solutions-for-equation
result=[]
while True:
    if s.check() == sat:
        m = s.model()
        print m[inp]
        result.append(m)
        # Create a new constraint the blocks the current model
        block = []
        for d in m:
            # d is a declaration
            if d.arity() > 0:
                raise Z3Exception("uninterpreted functions are not supported")
            # create a constant from declaration
            c=d()
            if is_array(c) or c.sort().kind() == Z3_UNINTERPRETED_SORT:
                raise Z3Exception("arrays and uninterpreted sorts are not supported")
            block.append(c != m[d])
        s.add(Or(block))
    else:
            print "results total=",len(result)
            break
```

We got:

```
13641239246085845637
1234567890
9223372038089343698
4611686019661955794
13835058056516731602
3096040143925676201
12319412180780452009
7707726162353064105
16931098199207839913
1906652839273745429
11130024876128521237
15741710894555909141
6518338857701133333
5975809943035972467
15199181979890748275
10587495961463360371
results total= 16
```

So there are 16 correct input values for `0x92EE577B63E80B73` as a result.

The second is 1234567890—it is indeed the value which was used by me originally while preparing this example.

Let's also try to research our algorithm a bit more. Acting on a sadistic whim, let's find if there are any possible input/output pairs in which the lower 32-bit parts are equal to each other?

Let's remove the *outp* constraint and add another, at line 17:

```
 1 from z3 import *
 2 
 3 C1=0x5D7E0D1F2E0F1F84
 4 C2=0x388D76AEE8CB1500
 5 C3=0xD2E9EE7E83C4285B
 6 
 7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
 8 
 9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16 
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
18 
19 print s.check()
20 m=s.model()
21 print m
22 print (" inp=0x%X" % m[inp].as_long())
23 print ("outp=0x%X" % m[outp].as_long())
```

It is indeed so:

```
sat
[i1 = 14869545517796235860,
 i3 = 8388171335828825253,
 i5 = 6918262285561543945,
 inp = 1370377541658871093,
 outp = 14543180351754208565,
 i4 = 10167065714588685486,
 i2 = 5541032613289652645]
 inp=0x13048F1D12C00535
outp=0xC9D3C17A12C00535
```

Let's be more sadistic and add another constraint: last 16 bits must be `0x1234`:

```
 1 from z3 import *
 2 
 3 C1=0x5D7E0D1F2E0F1F84
 4 C2=0x388D76AEE8CB1500
 5 C3=0xD2E9EE7E83C4285B
 6 
 7 inp, i1, i2, i3, i4, i5, i6, outp = BitVecs('inp i1 i2 i3 i4 i5 i6 outp', 64)
 8 
 9 s = Solver()
10 s.add(i1==inp*C1)
11 s.add(i2==RotateRight (i1, i1 & 0xF))
12 s.add(i3==i2 ^ C2)
13 s.add(i4==RotateLeft(i3, i3 & 0xF))
14 s.add(i5==i4 + C3)
15 s.add(outp==RotateLeft (i5, URem(i5, 60)))
16 
17 s.add(outp & 0xFFFFFFFF == inp & 0xFFFFFFFF)
```

```
s.add(outp & 0xFFFF == 0x1234)

print s.check()
m=s.model()
print m
print (" inp=0x%X" % m[inp].as_long())
print ("outp=0x%X" % m[outp].as_long())
```

Oh yes, this possible as well:

```
sat
[i1 = 2834222860503985872,
 i3 = 2294680776671411152,
 i5 = 17492621421353821227,
 inp = 461881484695179828,
 outp = 419247225543463476,
 i4 = 2294680776671411152,
 i2 = 2834222860503985872]
 inp=0x668EEC35F961234
outp=0x5D177215F961234
```

Z3 works very fast and it implies that the algorithm is weak, it is not cryptographic at all (like the most of the amateur cryptography).

Is it possible to tackle real cryptography by these methods? Real algorithms like AES, RSA, etc, can also be represented as huge system of equations, but these are so huge that they are impossible to work with on computers, now or in the near future. Of course, cryptographers are fully aware of this.

Summarizing, when dealing with amateur crypto, it's a very good idea to try a SMT/SAT solver (like Z3).

4.12 Rockey dongle: finding unknown algorithm using only input/output pairs

(This article was first published in August 2012 in my blog: http://blog.yurichev.com/node/71).

Some smartcards can execute Java or .NET code - that's the way to hide your sensitive algorithm into chip that is very hard to break (decapsulate). For example, one may encrypt/decrypt data files by hidden crypto algorithm rendering software piracy of such software nearly impossible.

That's what called Black box in mathematics[37].

Some software protection dongles offers this functionality too.

One example is Rockey 4 (www.rockey.nl).

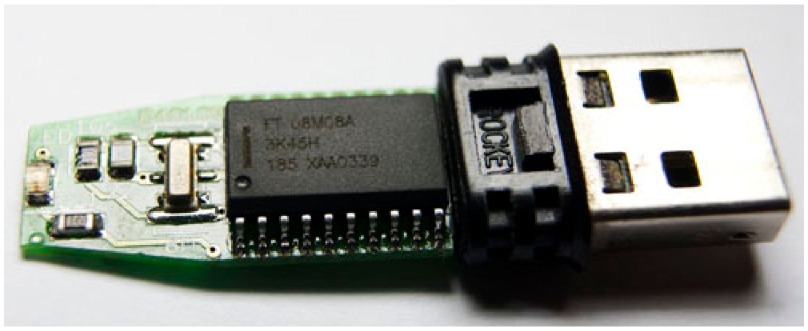


Figure 3: Rockey 4 dongle

This is small dongle connected via USB. Is contain some user-defined memory but also memory for user algorithms.

The virtual (toy) CPU for these algorithms is very simple: it offer only 8 16-bit registers (however, only 4 can be set and read) and 8 operations (add, subtract, cyclic left shift, multiplication, or, xor, and, negate).

Second instruction argument can be a constant (from 0 to 63) instead of register.

Each algorithm is described by string like `A=A+B, B=C*13, D=D^A, C=B*55, C=C&A, D=D|A, A=A*9, A=A&B`.

There are no stack, conditional/unconditional jumps, etc.

Each algorithm, obviously, can't have side effects, so they are actually pure functions [38] and their results can be memoized[39].

[37] http://en.wikipedia.org/wiki/Black_box
[38] http://en.wikipedia.org/wiki/Pure_function
[39] http://en.wikipedia.org/wiki/Memoization

By the way, as it was mentioned in Rockey 4 manual, first and last instruction cannot have constants. Maybe that's because these fields used for some internal data: each algorithm start and end should be marked somehow internally anyway.

Would it be possible to reveal hidden impossible-to-read algorithm only by recording input/output dongle traffic?

Common sense tell us "no". But we can try anyway.

Since, my goal wasn't to break into some Rockey-protected software, I was interesting only in limits (which algorithms could we find), so I make some things simpler: we will work with only 4 16-bit registers, and there will be only 6 operations (add, subtract, multiplication, or, xor, and).

Let's first calculate, how much information will be used in brute-force case.

There are 384 of all possible instructions in format reg=reg,op,reg for 4 registers and 6 operations, and also 6144 instructions in format reg=reg,op,constant. Remember that constant limited to 63 as maximal value? That help us for a little.

So, here 6528 all possible instructions. This mean, there are about 11854977354713530368 5-instruction algorithms. Wow! That's too much. I don't even know a precise name for such numbers. That's about 11 quintillions (thanks to Wikipedia).

Now let's try to use real heavy machinery:

> Constraint satisfaction problems (CSPs) are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time. The boolean satisfiability problem (SAT), the Satisfiability Modulo Theories (SMT) and answer set programming (ASP) can be roughly thought of as certain forms of the constraint satisfaction problem.

[40]

... and:

> In computer science and mathematical logic, the Satisfiability Modulo Theories (SMT) problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. Examples of theories typically used in computer science are the theory of real numbers, the theory of integers, and the theories of various data structures such as lists, arrays, bit vectors and so on. SMT can be thought of as a form of the constraint satisfaction problem and thus a certain formalized approach to constraint programming.

[41]

I'll use Z3 Theorem Prover [42] from Microsoft Research with excellent Python bindings, working like SMT solver.

It can be said, SMT solver is just solver of very big system of equations. By the way, its sibling SAT-solver is intended for solving very big boolean system of equations.

Needless to say, a lot of tasks can be expressed as system of equations. One very simple example is Sudoku: https://sites.google.com/site/modante/sudokusolver

So let's back to our toy CPU inside of Rockey 4 dongle.

How can we express each instruction as system of equations? While remembering some school math, I wrote this:

Function one_step()=

```
Each Bx is integer, but may be only 0 or 1.

# only one of B1..B4 and B5..B9 can be set
reg1=B1*A + B2*B + B3*C + B4*D
reg_or_constant2=B5*A + B6*B + B7*C + B8*D + B9*constant
reg1 should not be equal to reg_or_constant2

# Only one of B10..B15, can be set
result=result+B10*(reg1*reg2)
```

[40] http://en.wikipedia.org/wiki/Constraint_satisfaction_problem
[41] http://en.wikipedia.org/wiki/Satisfiability_Modulo_Theories
[42] http://research.microsoft.com/en-us/um/redmond/projects/z3/

```
result=result+B11*(reg1^reg2)
result=result+B12*(reg1+reg2)
result=result+B13*(reg1-reg2)
result=result+B14*(reg1|reg2)
result=result+B15*(reg1&reg2)

B16 - true if register isn't updated in this part
B17 - true if register is updated in this part
(B16 cannot be equal to B17)
A=B16*A + B17*result
B=B18*A + B19*result
C=B20*A + B21*result
D=B22*A + B23*result
```

That's how we can express each instruction in algorithm.
5-instructions algorithm can be expressed like this: one_step (one_step (one_step (one_step (one_step (input_registers)))))
Let's also add five known input/output pairs and we'll get system of equations like this:

```
one_step (one_step (one_step (one_step (one_step (input_1)))))==output_1
one_step (one_step (one_step (one_step (one_step (input_2)))))==output_2
one_step (one_step (one_step (one_step (one_step (input_3)))))==output_3
one_step (one_step (one_step (one_step (one_step (input_4)))))==output_4
.. etc
```

So the question now is to find about 5*23 boolean values satisfying known input/output pairs.
I wrote small utility to probe Rockey 4 algorithm with random numbers, it produce results in form:

```
RY_CALCULATE1: (input) p1=30760 p2=18484 p3=41200 p4=61741 (output) p1=49244 p2=11312 p3=27587 p4=12657
RY_CALCULATE1: (input) p1=51139 p2=7852 p3=53038 p4=49378 (output) p1=58991 p2=34134 p3=40662 p4=9869
RY_CALCULATE1: (input) p1=60086 p2=52001 p3=13352 p4=45313 (output) p1=46551 p2=42504 p3=61472 p4=1238
RY_CALCULATE1: (input) p1=48318 p2=6531 p3=51997 p4=30907 (output) p1=54849 p2=20601 p3=31271 p4=44794
```

p1/p2/p3/p4 are just another names for A/B/C/D registers.
Now let's start with Z3. We will need to express Rockey 4 toy CPU in Z3Py (Z3 for Python) terms.
It can be said, my Python script is divided into two parts:

- constraint definitions (like, "output_1 should be n for input_1=m", "constant cannot be greater than 64", etc);
- functions constructing system of equations.

This piece of code define some kind of "structure" consisting of 4 named 16-bit variables, each represent register in our toy CPU.

```
Registers_State=Datatype ('Registers_State')
Registers_State.declare('cons', ('A', BitVecSort(16)), ('B', BitVecSort(16)), ('C', BitVecSort(16)), ('
    D', BitVecSort(16)))
Registers_State=Registers_State.create()
```

These enumarations define two new types (or "sorts" in Z3's terminology):

```
Operation, (OP_MULT, OP_MINUS, OP_PLUS, OP_XOR, OP_OR, OP_AND) = EnumSort('Operation', ('OP_MULT', '
    OP_MINUS', 'OP_PLUS', 'OP_XOR', 'OP_OR', 'OP_AND'))

Register, (A, B, C, D) = EnumSort('Register', ('A', 'B', 'C', 'D'))
```

This part is very important, it define all variables in our system of equations. op_step is type of operation in instruction. reg_or_constant is selector between register or constant in second argument — False if register and True if constant. reg_step is register assigned in this instruction. reg1_step and reg2_step are just registers at arg1 and arg2. constant_step is constant (in case it's used in instruction instead of arg2).

```
op_step=[Const('op_step%s' % i, Operation) for i in range(STEPS)]
reg_or_constant_step=[Bool('reg_or_constant_step%s' % i) for i in range(STEPS)]
reg_step=[Const('reg_step%s' % i, Register) for i in range(STEPS)]
reg1_step=[Const('reg1_step%s' % i, Register) for i in range(STEPS)]
reg2_step=[Const('reg2_step%s' % i, Register) for i in range(STEPS)]
constant_step = [BitVec('constant_step%s' % i, 16) for i in range(STEPS)]
```

Adding constraints is very simple. Remember, I wrote that each constant cannot be larger than 63?

```
# according to Rockey 4 dongle manual, arg2 in first and last instructions cannot be a constant
s.add (reg_or_constant_step[0]==False)
s.add (reg_or_constant_step[STEPS-1]==False)

...

for x in range(STEPS):
   s.add (constant_step[x]>=0, constant_step[x]<=63)
```

Input/output values are added as constraints too.
Now let's see how to construct our system of equations:

```
# Register, Registers_State -> int
def register_selector (register, input_registers):
    return If(register==A, Registers_State.A(input_registers),
           If(register==B, Registers_State.B(input_registers),
           If(register==C, Registers_State.C(input_registers),
           If(register==D, Registers_State.D(input_registers),
                           0)))) # default
```

This function returning corresponding register value from "structure". Needless to say, the code above is not executed. If() is Z3Py function. The code only declares the function, which will be used in another. By the way, expression declaration resembling LISP language in some way.
Here is another function where register_selector() used:

```
# Bool, Register, Registers_State, int -> int
def register_or_constant_selector (register_or_constant, register, input_registers, constant):
    return If(register_or_constant==False, register_selector(register, input_registers), constant)
```

The code here is never executed too. It only construct one small piece of very big expression. But for the sake of simplicity, one can think all these functions will be called during bruteforce search.

```
# Operation, Bool, Register, Register, Int, Registers_State -> int
def one_op (op, register_or_constant, reg1, reg2, constant, input_registers):
    arg1=register_selector(reg1, input_registers)
    arg2=register_or_constant_selector (register_or_constant, reg2, input_registers, constant)
    return If(op==OP_MULT,   arg1*arg2,
           If(op==OP_MINUS,  arg1-arg2,
           If(op==OP_PLUS,   arg1+arg2,
           If(op==OP_XOR,    arg1^arg2,
           If(op==OP_OR,     arg1|arg2,
           If(op==OP_AND,    arg1&arg2,
                          0)))))) # default
```

Here is expression describing each instruction. Register assigned is instruction is substitued with new_val, while all other registers are copied from input register's state:

```
# Bool, Register, Operation, Register, Register, Int, Registers_State -> Registers_State
def one_step (register_or_constant, register_assigned_in_this_step, op, reg1, reg2, constant,
    input_registers):
    new_val=one_op(op, register_or_constant, reg1, reg2, constant, input_registers)
    return If (register_assigned_in_this_step==A, Registers_State.cons (new_val,
                                                                        Registers_State.B(
    input_registers),
                                                                        Registers_State.C(
    input_registers),
                                                                        Registers_State.D(
    input_registers)),
           If (register_assigned_in_this_step==B, Registers_State.cons (Registers_State.A(
    input_registers),
                                                                        new_val,
                                                                    Registers_State.C(
    input_registers),
                                                                    Registers_State.D(
    input_registers)),
           If (register_assigned_in_this_step==C, Registers_State.cons (Registers_State.A(
    input_registers),
                                                                    Registers_State.B(
    input_registers),
                                                                    new_val,
                                                                    Registers_State.D(
    input_registers)),
           If (register_assigned_in_this_step==D, Registers_State.cons (Registers_State.A(
    input_registers),
                                                                    Registers_State.B(
    input_registers),
                                                                    Registers_State.C(
    input_registers),
                                                                    new_val),
                                                      Registers_State.cons(0,0,0,0))))) # default
```

This is the last function describing whole n-step program:

```
def program(input_registers, STEPS):
    cur_input=input_registers
    for x in range(STEPS):
        cur_input=one_step (reg_or_constant_step[x], reg_step[x], op_step[x], reg1_step[x], reg2_step[x
    ], constant_step[x], cur_input)
    return cur_input
```

Again, for the sake of simplicity, it can be said, now Z3 will try each possible registers/operations/constants against this expression to find such combination which satisfy input/output pairs. But it's not true. As far as I right, Z3 use DPLL algorithm[43].

Now let's start with very simple 3-step algorithm: "B=A^D, C=D*D, D=A*C". Please note: register A left unchanged. I programmed Rockey 4 dongle with it and recorded algorithm outputs:

```
RY_CALCULATE1: (input) p1=8803 p2=59946 p3=36002 p4=44743 (output) p1=8803 p2=36004 p3=7857 p4=24691
RY_CALCULATE1: (input) p1=5814 p2=55512 p3=52155 p4=55813 (output) p1=5814 p2=52403 p3=33817 p4=4038
RY_CALCULATE1: (input) p1=25206 p2=2097 p3=55906 p4=22705 (output) p1=25206 p2=15047 p3=10849 p4=43702
RY_CALCULATE1: (input) p1=10044 p2=14647 p3=27923 p4=7325 (output) p1=10044 p2=15265 p3=47177 p4=20508
RY_CALCULATE1: (input) p1=15267 p2=2690 p3=47355 p4=56073 (output) p1=15267 p2=57514 p3=26193 p4=53395
```

It took about one second and only 5 pairs above to find algorithm (on my quad-core Xeon E3-1220 (clocked at 3.1GHz), however, Z3 solver working in single-thread mode):

```
B = A ^ D
C = D * D
D = C * A
```

Notice last instruction: C and A registers are swapped comparing to version I wrote by hand. But of course, this instruction is working in the same way.

Now if I try to find all 4-step programs satisfying to these values, my script will offer this:

```
B = A ^ D
C = D * D
D = A * C
A = A | A
```

... and that's really fun, because last instruction do nothing with value in register A, it's like "no operation" — but still, algorithm is correct for values given!

Here is another 5-step algorithm: "B=B^D, C=A*22, A=B*19, A=A&42, D=B&C" and values:

```
RY_CALCULATE1: (input) p1=61876 p2=28737 p3=28636 p4=50362 (output) p1=32 p2=46331 p3=50552 p4=33912
RY_CALCULATE1: (input) p1=46843 p2=43355 p3=39078 p4=24552 (output) p1=8 p2=63155 p3=47506 p4=45202
```

[43] http://en.wikipedia.org/wiki/DPLL_algorithm

```
RY_CALCULATE1: (input) p1=22425 p2=51432 p3=40836 p4=14260 (output) p1=0 p2=65372 p3=34598 p4=34564
RY_CALCULATE1: (input) p1=44214 p2=45766 p3=19778 p4=59924 (output) p1=2 p2=22738 p3=55204 p4=20608
RY_CALCULATE1: (input) p1=27348 p2=49060 p3=31736 p4=59576 (output) p1=0 p2=22300 p3=11832 p4=1560
```

It took 37 seconds and we've got:

```
B = D ^ B
C = A * 22
A = B * 19
A = A & 42
D = C & B
```

A=A&42 was correctly deduced (look at these five p1's at output (assigned to output A register): 32,8,0,2,0)
6-step algorithm "A=A+B, B=C*13, D=D^A, C=C&A, D=D|B, A=A&B" and values:

```
RY_CALCULATE1: (input) p1=4110 p2=35411 p3=54308 p4=47077 (output) p1=32832 p2=50644 p3=36896 p4=60884
RY_CALCULATE1: (input) p1=12038 p2=7312 p3=39626 p4=47017 (output) p1=18434 p2=56386 p3=2690 p4=64639
RY_CALCULATE1: (input) p1=48763 p2=27663 p3=12485 p4=20563 (output) p1=10752 p2=31233 p3=8320 p4=31449
RY_CALCULATE1: (input) p1=33174 p2=38937 p3=54005 p4=38871 (output) p1=4129 p2=46705 p3=4261 p4=48761
RY_CALCULATE1: (input) p1=46587 p2=36275 p3=6090 p4=63976 (output) p1=258 p2=13634 p3=906 p4=48966
```

90 seconds and we've got:

```
A = A + B
B = C * 13
D = D ^ A
D = B | D
C = C & A
A = B & A
```

But that was simple, however. Some tasks are not possible even for 6-step algorithms, for example: "A=A^B, A=A*9, A=A^C, A=A*19, A=A^D, A=A&B". Solver was working too long (up to several hours), so I didn't even know is it possible to find it anyway.

4.12.1 Conclusion

This is in fact an exercise in program synthesis.
Some short algorithms for tiny CPU's are really possible to find using so minimum data about it! Of course it's still not possible to reveal some harder algorithm, but this method definitely should not be ignored!
Now, files: Rockey 4 dongle programmer and reader, Rockey 4 manual, Z3Py script for finding algorithms, input/output pairs, and also fixed z3.py file from Z3 (my script may fail to work with unfixed z3.py coming with Z3 4.0 installation, so I got another, you may try to use it too):
http://yurichev.com/non-wiki-files/rockey4_algo_search.zip

4.12.2 Future work

Perhaps, constructing LISP-like S-expression can be better than a program for toy-level CPU.
It's also possible to start with smaller constants and then proceed to bigger. This is somewhat similar to increasing password length in password brute-force cracking.

4.13 Solving pipe puzzle using Z3 SMT-solver

"Pipe puzzle" is a popular puzzle (just google "pipe puzzle" and look at images).
This is how shuffled puzzle looks like:

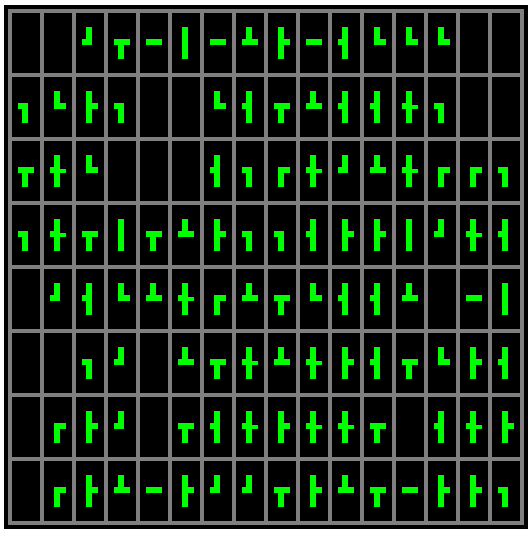


Figure 4: Shuffled puzzle

…and solved:

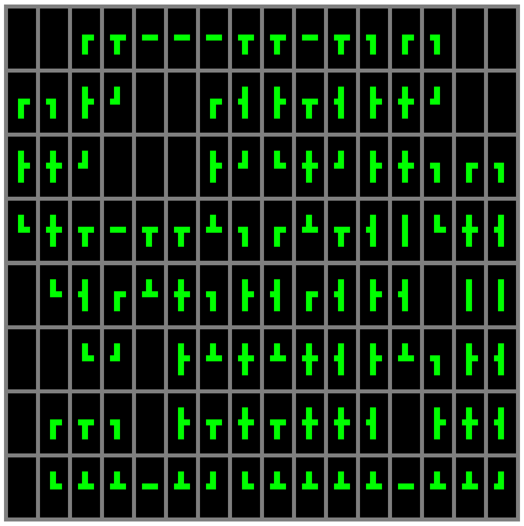
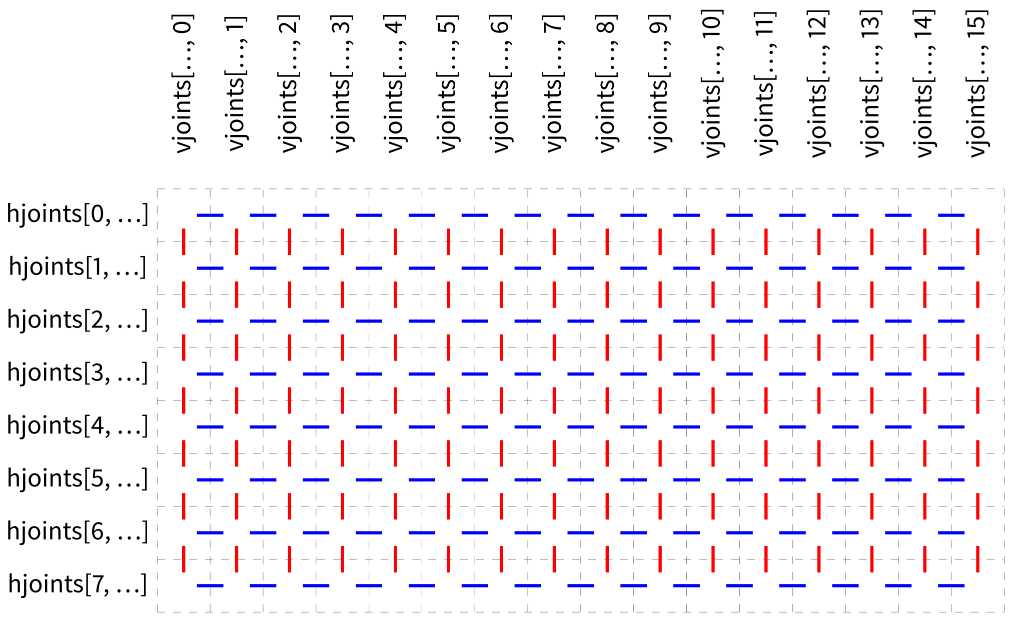


Figure 5: Solved puzzle

Let's try to find a way to solve it.

4.13.1 Generation

First, we need to generate it. Here is my quick idea on it. Take 8*16 array of cells. Each cell may contain some kind of block. There are joints between cells:



Blue lines are horizontal joints, red lines are vertical joints. We just set each joint to 0/false (absent) or 1/true (present), randomly. Once set, it's now easy to find type for each cell. There are:

joints	our internal name	angle	symbol
0	type 0	0°	(space)
2	type 2a	0°	┃
2	type 2a	90°	━
2	type 2b	0°	┏
2	type 2b	90°	┓
2	type 2b	180°	┛
2	type 2b	270°	┗
3	type 3	0°	┣
3	type 3	90°	┳
3	type 3	180°	┫
3	type 3	270°	┻
4	type 4	0°	╋

Dangling joints are possible at the first stage (i.e., cell with only one joint), but they are removed recursively, these cells are transforming into empty cells. Hence, at the end, all cells has at least two joints, and the whole plumbing system has no connections with outer world—I hope this would make things clearer.

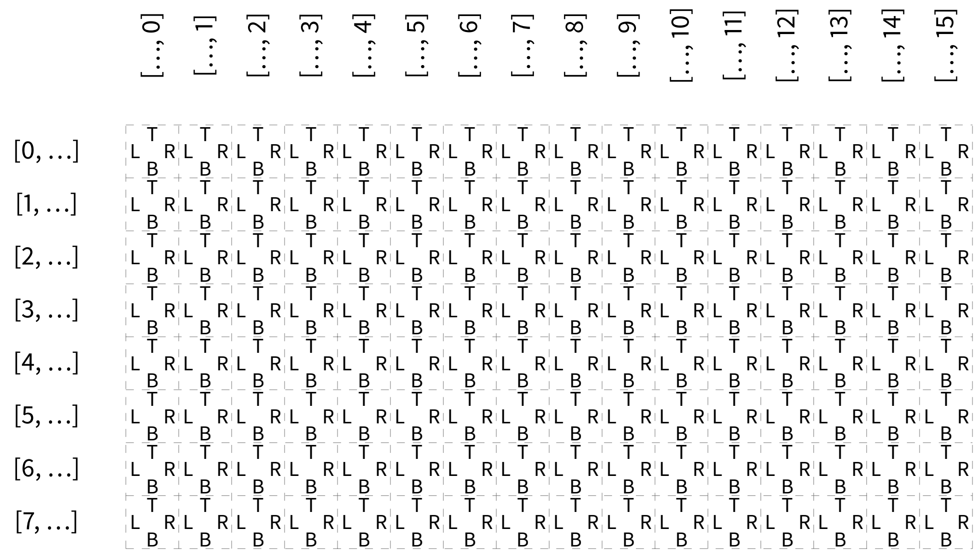
The C source code of generator is here: `https://github.com/dennis714/SAT_SMT_article/tree/master/SMT/pipe/generator`. All horizontal joints are in global array *hjoints[]* and vertical in *vjoints[]*.

The C program generates ANSI-colored output like it has been showed above (4.13, 4.13) plus an array of types, with no angle information about each cell:

```
[
["0", "0", "2b", "3", "2a", "2a", "2a", "3", "3", "2a", "3", "2b", "2b", "2b", "0", "0"],
["2b", "2b", "3", "2b", "0", "0", "2b", "3", "3", "3", "3", "3", "4", "2b", "0", "0"],
["3", "4", "2b", "0", "0", "0", "3", "2b", "2b", "4", "2b", "3", "4", "2b", "2b", "2b"],
["2b", "4", "3", "2a", "3", "3", "3", "2b", "2b", "3", "3", "3", "2a", "2b", "4", "3"],
["0", "2b", "3", "2b", "3", "4", "2b", "3", "3", "2b", "3", "3", "3", "0", "2a", "2a"],
["0", "0", "2b", "2b", "0", "3", "3", "4", "3", "4", "3", "3", "3", "2b", "3", "3"],
["0", "2b", "3", "2b", "0", "3", "3", "4", "3", "4", "4", "3", "0", "3", "4", "3"],
["0", "2b", "3", "3", "2a", "3", "2b", "2b", "3", "3", "3", "3", "2a", "3", "3", "2b"],
]
```

4.13.2 Solving

First of all, we would think about 8*16 array of cells, where each has four bits: "T" (top), "B" (bottom), "L" (left), "R" (right). Each bit represents half of joint.



Now we define arrays of each of four half-joints + angle information:

```
HEIGHT=8
WIDTH=16

# if T/B/R/L is Bool instead of Int, Z3 solver will work faster
T=[[Bool('cell_%d_%d_top' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
B=[[Bool('cell_%d_%d_bottom' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
R=[[Bool('cell_%d_%d_right' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
L=[[Bool('cell_%d_%d_left' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
A=[[Int('cell_%d_%d_angle' % (r, c)) for c in range(WIDTH)] for r in range(HEIGHT)]
```

We know that if each half-joint is present, corresponding half-joint must be also present, and vice versa. We define this using these constraints:

```
# shorthand variables for True and False:
t=True
f=False

# "top" of each cell must be equal to "bottom" of the cell above
# "bottom" of each cell must be equal to "top" of the cell below
# "left" of each cell must be equal to "right" of the cell at left
# "right" of each cell must be equal to "left" of the cell at right
for r in range(HEIGHT):
    for c in range(WIDTH):
        if r!=0:
            s.add(T[r][c]==B[r-1][c])
        if r!=HEIGHT-1:
            s.add(B[r][c]==T[r+1][c])
        if c!=0:
            s.add(L[r][c]==R[r][c-1])
        if c!=WIDTH-1:
            s.add(R[r][c]==L[r][c+1])

# "left" of each cell of first column shouldn't have any connection
# so is "right" of each cell of the last column
for r in range(HEIGHT):
    s.add(L[r][0]==f)
    s.add(R[r][WIDTH-1]==f)

# "top" of each cell of the first row shouldn't have any connection
# so is "bottom" of each cell of the last row
for c in range(WIDTH):
    s.add(T[0][c]==f)
    s.add(B[HEIGHT-1][c]==f)
```

Now we'll enumerate all cells in the initial array (4.13.1). First two cells are empty there. And the third one has type "2b". This is "┏" and it can be oriented in 4 possible ways. And if it has angle 0°, bottom half-joint and right one are present, others are absent. If it has angle 90°, it looks like "┓", and bottom and left half-joints are present, others are absent.

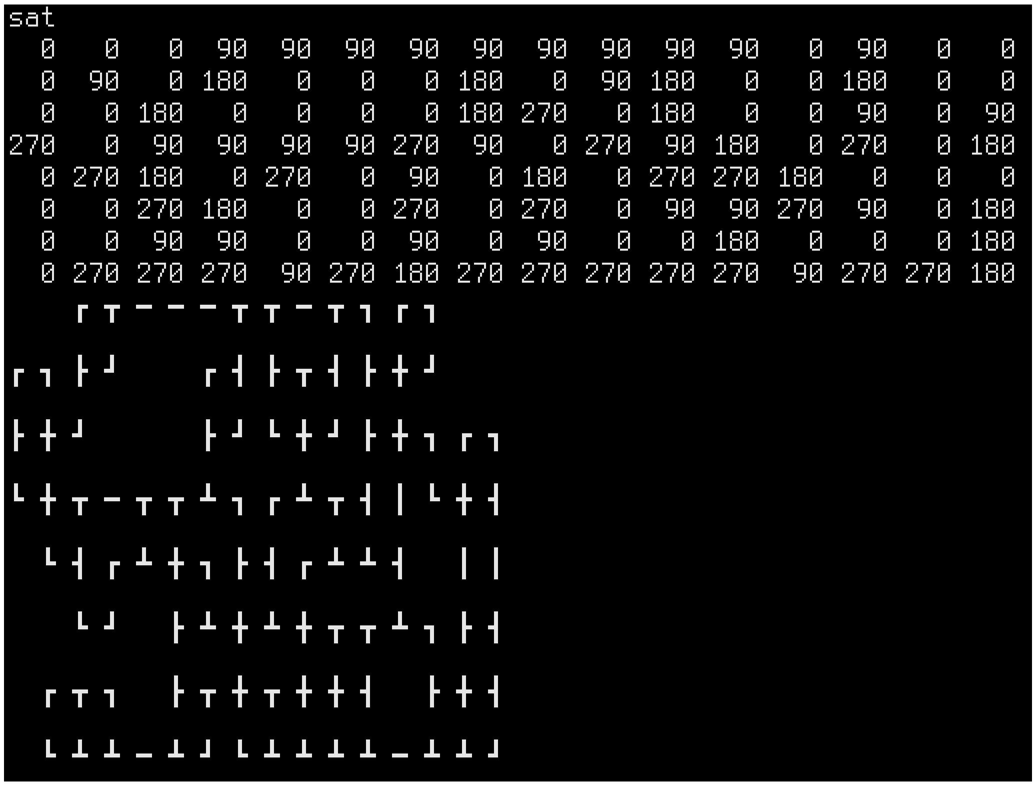
In plain English: "if cell of this type has angle 0°, these half-joints must be present **OR** if it has angle 90°, these half-joints must be present, **OR**, etc, etc."

Likewise, we define all these rules for all types and all possible angles:

```
for r in range(HEIGHT):
    for c in range(WIDTH):
        ty=cells_type[r][c]

        if ty=="0":
            s.add(A[r][c]==f)
            s.add(T[r][c]==f, B[r][c]==f, L[r][c]==f, R[r][c]==f)

        if ty=="2a":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==f, T[r][c]==t, B[r][c]==t),   # ┃
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==t, T[r][c]==f, B[r][c]==f)))  # ━

        if ty=="2b":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==t, T[r][c]==f, B[r][c]==t),   # ┏
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==f, T[r][c]==f, B[r][c]==t),   # ┓
                    And(A[r][c]==180, L[r][c]==t, R[r][c]==f, T[r][c]==t, B[r][c]==f),  # ┛
                    And(A[r][c]==270, L[r][c]==f, R[r][c]==t, T[r][c]==t, B[r][c]==f))) # ┗

        if ty=="3":
            s.add(Or(And(A[r][c]==0, L[r][c]==f, R[r][c]==t, T[r][c]==t, B[r][c]==t),   # ┣
                    And(A[r][c]==90, L[r][c]==t, R[r][c]==t, T[r][c]==f, B[r][c]==t),   # ┳
                    And(A[r][c]==180, L[r][c]==t, R[r][c]==f, T[r][c]==t, B[r][c]==t),  # ┫
                    And(A[r][c]==270, L[r][c]==t, R[r][c]==t, T[r][c]==t, B[r][c]==f))) # ┻

        if ty=="4":
            s.add(A[r][c]==0)
            s.add(T[r][c]==t, B[r][c]==t, L[r][c]==t, R[r][c]==t) # ╋
```

Full listing is here: https://github.com/dennis714/SAT_SMT_article/blob/master/SMT/pipe/solver/solve_pipe_puzzle1.py.

It produces this result (prints angle for each cell and (pseudo)graphical representation):



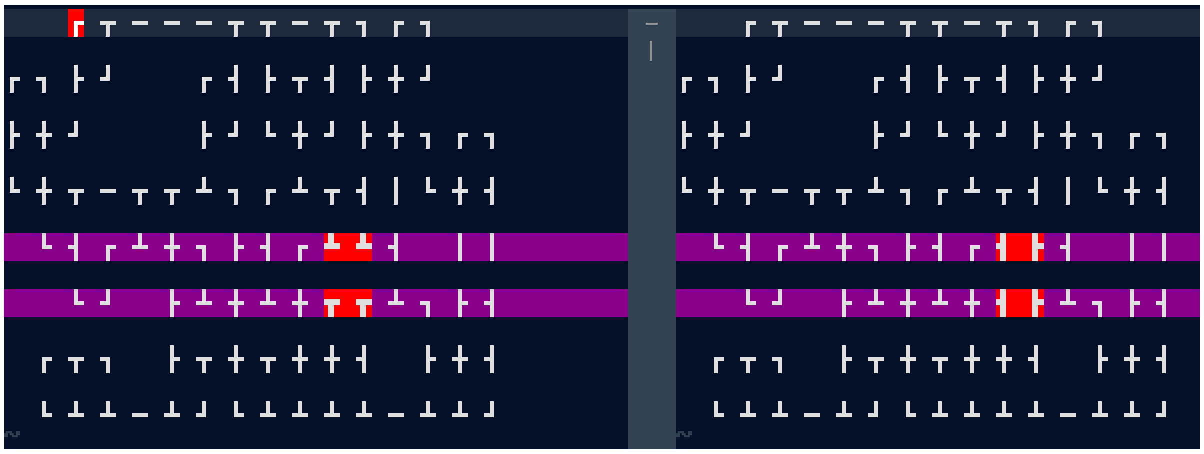
```
sat
   0   0   0  90  90  90  90  90  90  90  90  90   0  90   0   0
   0  90   0 180   0   0   0 180   0  90 180   0   0 180   0   0
   0   0 180   0   0   0   0 180 270   0 180   0   0  90   0  90
 270   0  90  90  90  90 270  90   0 270  90 180   0 270   0 180
   0 270 180   0 270   0  90   0 180   0 270 270 180   0   0   0
   0   0 270 180   0   0 270   0 270   0  90  90 270  90   0 180
   0   0  90  90   0   0  90   0  90   0   0 180   0   0   0 180
   0 270 270 270  90 270 180 270 270 270 270 270  90 270 270 180
    ┌ ┬ ─ ─ ─ ┬ ┬ ─ ┬ ┐ ┌ ┐
┌ ┐ ├ ┘     ┌ ┤ ├ ┬ ┤ ├ ┼ ┘
├ ┼ ┘       ├ ┘ └ ┼ ┘ ├ ┼ ┐ ┌ ┐
└ ┼ ┬ ─ ┬ ┬ ┴ ┐ ┌ ┴ ┬ ┤ │ └ ┼ ┤
  └ ┤ ┌ ┴ ┼ ┐ ├ ┤ ┌ ┴ ┴ ┤   │ │
    └ ┘   ├ ┴ ┼ ┴ ┼ ┬ ┬ ┴ ┐ ├ ┤
  ┌ ┬ ┐   ├ ┬ ┼ ┬ ┼ ┼ ┤   ├ ┼ ┤
  └ ┴ ┴ ─ ┴ ┘ └ ┴ ┴ ┴ ┴ ─ ┴ ┴ ┘
```

Figure 6: Solver script output

It worked 4 seconds on my old and slow Intel Atom N455 1.66GHz. Is it fast? I don't know, but again, what is really cool, we do not know about any mathematical background of all this, we just defined cells, (half)-joints and defined relations between them.

Now the next question is, how many solutions are possible? Using method described earlier (4.7), I've altered solver script [44] and solver said two solutions are possible.

Let's compare these two solutions using gvimdiff:



```
    ┌ ┬ ─ ─ ─ ┬ ┬ ─ ┬ ┐ ┌ ┐            ┌ ┬ ─ ─ ─ ┬ ┬ ─ ┬ ┐ ┌ ┐
┌ ┐ ├ ┘     ┌ ┤ ├ ┬ ┤ ├ ┼ ┘        ┌ ┐ ├ ┘     ┌ ┤ ├ ┬ ┤ ├ ┼ ┘
├ ┼ ┘       ├ ┘ └ ┼ ┘ ├ ┼ ┐ ┌ ┐    ├ ┼ ┘       ├ ┘ └ ┼ ┘ ├ ┼ ┐ ┌ ┐
└ ┼ ┬ ─ ┬ ┬ ┴ ┐ ┌ ┴ ┬ ┤ │ └ ┼ ┤    └ ┼ ┬ ─ ┬ ┬ ┴ ┐ ┌ ┴ ┬ ┤ │ └ ┼ ┤
  └ ┤ ┌ ┴ ┼ ┐ ├ ┤ ┌ ┴ ┴ ┤   │ │      └ ┤ ┌ ┴ ┼ ┐ ├ ┤ ┌ ┤ ├ ┤   │ │
    └ ┘   ├ ┴ ┼ ┴ ┼ ┬ ┬ ┴ ┐ ├ ┤        └ ┘   ├ ┴ ┼ ┴ ┼ ┤ ├ ┴ ┐ ├ ┤
  ┌ ┬ ┐   ├ ┬ ┼ ┬ ┼ ┼ ┤   ├ ┼ ┤      ┌ ┬ ┐   ├ ┬ ┼ ┬ ┼ ┼ ┤   ├ ┼ ┤
  └ ┴ ┴ ─ ┴ ┘ └ ┴ ┴ ┴ ┴ ─ ┴ ┴ ┘      └ ┴ ┴ ─ ┴ ┘ └ ┴ ┴ ┴ ┴ ─ ┴ ┴ ┘
```

Figure 7: gvimdiff output (pardon my red cursor at left pane at left-bottom corner)

4 cells in the middle orientated differently. Perhaps, other puzzles may produce different results.

P.S. *Half-joints* are defined using boolean type. But in fact, the first version of the solver has been written using integer type for half-joints, and 0 was used for False and 1 for True. I did it so because I wanted to make source code tidier and narrower without using long words like "False" and "True". And it worked, but slower. Perhaps, Z3 handles boolean data types faster? Better? Anyway, I writing this to note that integer type can also be used instead of boolean, if needed.

[44] https://github.com/dennis714/SAT_SMT_article/blob/master/SMT/pipe/solver/solve_pipe_puzzle2.py

5 Symbolic execution

5.1 Symbolic computation

Let's first start with symbolic computation[45].

Some numbers can only be represented in binary system approximately, like $\frac{1}{3}$ and π. If we calculate $\frac{1}{3} \cdot 3$ step-by-step, we may have some undesirable noise. We also know that $sin(\frac{\pi}{2}) = 1$, but calculating this expression in usual way, we can also have some noise in result. Arbitrary-precision arithmetic[46] is not a solution, because these numbers cannot be stored as a finite binary number in memory.

How we could tackle this problem? Humans reduce such expressions using paper and pencil without any calculations. We can mimic human behaviour programmatically if we will store expression as tree and numbers like π will be converted into number at the very last step(s).

This is what Wolfram Mathematica[47] does. Let's start it and try this:

```
In[]:= x + 2*8
Out[]= 16 + x
```

Since Mathematica has no clue what x is, it's left *as is*, but $2 \cdot 8$ can be reduced easily, both by Mathematica and by humans, so that is what has done. In some time in near future, Mathematica's user may assign some number to x and then, Mathematica will reduce the expression even further.

Mathematica does this because it parses the expression and finds some known patterns. This is also called *term rewriting*[48]. In plain English language it may sounds like this: "if there is a $+$ operator between two known numbers, replace this subexpression by a computed number which is sum of these two numbers, if possible". Just like humans do.

Mathematica also has rules like "replace $sin(\pi)$ by 0" and "replace $sin(\frac{\pi}{2})$ by 1", but as you can see, π must be preserved as some kind of symbol instead of a number.

So Mathematica left x as unknown value. This is, in fact, common mistake by Mathematica's users: a small typo in an input expression may lead to a huge irreducible expression with this typo included.

Another example: Mathematica left this deliberately while computing binary logarithm:

```
In[]:= Log[2, 36]
Out[]= Log[36]/Log[2]
```

Because it has a hope that at some point in future, this expression will become a subexpression in another expression and it will be reduced nicely at the very end. But if we really need an answer, we can force Mathematica to calculate it:

```
In[]:= Log[2, 36] // N
Out[]= 5.16993
```

Sometimes unresolved values are desirable:

```
In[]:= Union[{a, b, a, c}, {d, a, e, b}, {c, a}]
Out[]= {a, b, c, d, e}
```

Characters in the expression are just unresolved symbols[49] with no connections to numbers or other expressions, so Mathematica left them *as is*.

Another real world example is symbolic integration[50], i.e., finding formula for integral by rewriting initial expression using some predefined rules. Mathematica also does it:

```
In[]:= Integrate[1/(x^5), x]
Out[]= -(1/(4 x^4))
```

Benefits of symbolic computation are obvious: it is not prone to loss of significance[51] and round-off errors[52], but drawbacks are also obvious: you need to store expression in (possible huge) tree and process it many times. Term rewriting is also slow. All these things are extremely clumsy in comparison to a fast **FPU!**[53].

"Symbolic computation" is opposed to "numerical computation", the last one is just processing numbers step-by-step, using calculator, **CPU!** or **FPU!**.

[45] https://en.wikipedia.org/wiki/Symbolic_computation
[46] https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic
[47] Another well-known symbolic computation system is Maxima and SymPy
[48] https://en.wikipedia.org/wiki/Rewriting
[49] *Symbol* like in LISP
[50] https://en.wikipedia.org/wiki/Symbolic_integration
[51] https://en.wikipedia.org/wiki/Loss_of_significance
[52] https://en.wikipedia.org/wiki/Round-off_error
[53] **FPU!**

Some problems can be solved better by the first method, some others – by the second one.

5.1.1 Rational data type

Some LISP implementations can store a number as a ratio/fraction [54], i.e., placing two numbers in a cell (which, in this case, is called *atom* in LISP lingo). For example, you divide 1 by 3, and the interpreter, by understanding that $\frac{1}{3}$ is an irreducible fraction[55], creates a cell with 1 and 3 numbers. Some time after, you may multiply this cell by 6, and the multiplication function inside LISP interpreter may return much better result (2 without *noise*).

Printing function in interpreter can also print something like `1 / 2` instead of floating point number.

This is sometimes called "fractional arithmetic", [see Donald E. Knuth, *The Art of Computing Programming*, 3rd ed., (1997), 4.5.1, p330].

This is not symbolic computation in any way, but this is slightly better than storing ratios/fractions as just floating point numbers.

Drawbacks are clearly visible: you need more memory to store ratio instead of a number; and all arithmetic functions are more complex and slower, because they must handle both numbers and ratios.

Perhaps, because of drawbacks, some programming languages offers separate (*rational*) data type, as language feature, or supported by a library [56]: Haskell, OCaml, Perl, Ruby, Python, Smalltalk, Java, Clojure, C/C++[57].

6 KLEE

6.1 School-level equation

Let's revisit school-level system of equations from (XXX).

We will force KLEE to find a path, where all the constraints are satisfied:

```
int main()
{
        int circle, square, triangle;

        klee_make_symbolic(&circle, sizeof circle, "circle");
        klee_make_symbolic(&square, sizeof square, "square");
        klee_make_symbolic(&triangle, sizeof triangle, "triangle");

        if (circle+circle!=10) return 0;
        if (circle*square+square!=12) return 0;
        if (circle*square-triangle*circle!=circle) return 0;

        // all constraints should be satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);
};
```

```
\$ clang -emit-llvm -c -g klee_eq.c
...

\$ klee klee_eq.bc
KLEE: output directory is "/home/klee/klee-out-93"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_eq.c:18: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 32
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1
```

[54] https://en.wikipedia.org/wiki/Rational_data_type
[55] https://en.wikipedia.org/wiki/Irreducible_fraction
[56] More detailed list: https://en.wikipedia.org/wiki/Rational_data_type
[57] By GNU Multiple Precision Arithmetic Library

Let's find out, where `klee_assert()` has been triggered:

```
\$ ls klee-last | grep err
test000001.external.err

\$ ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['klee_eq.bc']
num objects: 3
object    0: name: b'circle'
object    0: size: 4
object    0: data: 5
object    1: name: b'square'
object    1: size: 4
object    1: data: 2
object    2: name: b'triangle'
object    2: size: 4
object    2: data: 1
```

This is indeed correct solution to the system of equations.

KLEE has intrinsic `klee_assume()` which tells KLEE to cut path if some constraint is not true. So we can rewrite our example in such cleaner way:

```
int main()
{
        int circle, square, triangle;

        klee_make_symbolic(&circle, sizeof circle, "circle");
        klee_make_symbolic(&square, sizeof square, "square");
        klee_make_symbolic(&triangle, sizeof triangle, "triangle");

        klee_assume (circle+circle==10);
        klee_assume (circle*square+square==12);
        klee_assume (circle*square-triangle*circle==circle);

        // all constraints should be satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);
};
```

6.2 Zebra puzzle

Let's revisit zebra puzzle from (XXX).

We just define all variables and add constraints:

```
int main()
{
        int Yellow, Blue, Red, Ivory, Green;
        int Norwegian, Ukrainian, Englishman, Spaniard, Japanese;
        int Water, Tea, Milk, OrangeJuice, Coffee;
        int Kools, Chesterfield, OldGold, LuckyStrike, Parliament;
        int Fox, Horse, Snails, Dog, Zebra;

        klee_make_symbolic(&Yellow, sizeof(int), "Yellow");
        klee_make_symbolic(&Blue, sizeof(int), "Blue");
        klee_make_symbolic(&Red, sizeof(int), "Red");
        klee_make_symbolic(&Ivory, sizeof(int), "Ivory");
        klee_make_symbolic(&Green, sizeof(int), "Green");

        klee_make_symbolic(&Norwegian, sizeof(int), "Norwegian");
        klee_make_symbolic(&Ukrainian, sizeof(int), "Ukrainian");
```

```
    klee_make_symbolic(&Englishman, sizeof(int), "Englishman");
    klee_make_symbolic(&Spaniard, sizeof(int), "Spaniard");
    klee_make_symbolic(&Japanese, sizeof(int), "Japanese");

    klee_make_symbolic(&Water, sizeof(int), "Water");
    klee_make_symbolic(&Tea, sizeof(int), "Tea");
    klee_make_symbolic(&Milk, sizeof(int), "Milk");
    klee_make_symbolic(&OrangeJuice, sizeof(int), "OrangeJuice");
    klee_make_symbolic(&Coffee, sizeof(int), "Coffee");

    klee_make_symbolic(&Kools, sizeof(int), "Kools");
    klee_make_symbolic(&Chesterfield, sizeof(int), "Chesterfield");
    klee_make_symbolic(&OldGold, sizeof(int), "OldGold");
    klee_make_symbolic(&LuckyStrike, sizeof(int), "LuckyStrike");
    klee_make_symbolic(&Parliament, sizeof(int), "Parliament");

    klee_make_symbolic(&Fox, sizeof(int), "Fox");
    klee_make_symbolic(&Horse, sizeof(int), "Horse");
    klee_make_symbolic(&Snails, sizeof(int), "Snails");
    klee_make_symbolic(&Dog, sizeof(int), "Dog");
    klee_make_symbolic(&Zebra, sizeof(int), "Zebra");

    // limits.
    if (Yellow<1 || Yellow>5) return 0;
    if (Blue<1 || Blue>5) return 0;
    if (Red<1 || Red>5) return 0;
    if (Ivory<1 || Ivory>5) return 0;
    if (Green<1 || Green>5) return 0;

    if (Norwegian<1 || Norwegian>5) return 0;
    if (Ukrainian<1 || Ukrainian>5) return 0;
    if (Englishman<1 || Englishman>5) return 0;
    if (Spaniard<1 || Spaniard>5) return 0;
    if (Japanese<1 || Japanese>5) return 0;

    if (Water<1 || Water>5) return 0;
    if (Tea<1 || Tea>5) return 0;
    if (Milk<1 || Milk>5) return 0;
    if (OrangeJuice<1 || OrangeJuice>5) return 0;
    if (Coffee<1 || Coffee>5) return 0;

    if (Kools<1 || Kools>5) return 0;
    if (Chesterfield<1 || Chesterfield>5) return 0;
    if (OldGold<1 || OldGold>5) return 0;
    if (LuckyStrike<1 || LuckyStrike>5) return 0;
    if (Parliament<1 || Parliament>5) return 0;

    if (Fox<1 || Fox>5) return 0;
    if (Horse<1 || Horse>5) return 0;
    if (Snails<1 || Snails>5) return 0;
    if (Dog<1 || Dog>5) return 0;
    if (Zebra<1 || Zebra>5) return 0;

    // colors are distinct for all 5 houses:
    if (((1<<Yellow) | (1<<Blue) | (1<<Red) | (1<<Ivory) | (1<<Green))!=0x3E) return 0; // 111110

    // all nationalities are living in different houses:
    if (((1<<Norwegian) | (1<<Ukrainian) | (1<<Englishman) | (1<<Spaniard) | (1<<Japanese))!=0x3E)
return 0; // 111110
```

```
    // so are beverages:
    if (((1<<Water) | (1<<Tea) | (1<<Milk) | (1<<OrangeJuice) | (1<<Coffee))!=0x3E) return 0; //
111110

    // so are cigarettes:
    if (((1<<Kools) | (1<<Chesterfield) | (1<<OldGold) | (1<<LuckyStrike) | (1<<Parliament))!=0x3E)
 return 0; // 111110

    // so are pets:
    if (((1<<Fox) | (1<<Horse) | (1<<Snails) | (1<<Dog) | (1<<Zebra))!=0x3E) return 0; // 111110

    // main constraints of the puzzle:

    // 2.The Englishman lives in the red house.
    if (Englishman!=Red) return 0;

    // 3.The Spaniard owns the dog.
    if (Spaniard!=Dog) return 0;

    // 4.Coffee is drunk in the green house.
    if (Coffee!=Green) return 0;

    // 5.The Ukrainian drinks tea.
    if (Ukrainian!=Tea) return 0;

    // 6.The green house is immediately to the right of the ivory house.
    if (Green!=Ivory+1) return 0;

    // 7.The Old Gold smoker owns snails.
    if (OldGold!=Snails) return 0;

    // 8.Kools are smoked in the yellow house.
    if (Kools!=Yellow) return 0;

    // 9.Milk is drunk in the middle house.
    if (Milk!=3) return 0; // i.e., 3rd house

    // 10.The Norwegian lives in the first house.
    if (Norwegian!=1) return 0;

    // 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
    if (Chesterfield!=Fox+1 && Chesterfield!=Fox-1) return 0; // left or right

    // 12.Kools are smoked in the house next to the house where the horse is kept.
    if (Kools!=Horse+1 && Kools!=Horse-1) return 0; // left or right

    // 13.The Lucky Strike smoker drinks orange juice.
    if (LuckyStrike!=OrangeJuice) return 0;

    // 14.The Japanese smokes Parliaments.
    if (Japanese!=Parliament) return 0;

    // 15.The Norwegian lives next to the blue house.
    if (Norwegian!=Blue+1 && Norwegian!=Blue-1) return 0; // left or right

    // all constraints are satisfied at this point
    // force KLEE to produce .err file:
    klee_assert(0);

    return 0;
```

```
};
```

I force KLEE to find distinct values for colors, nationalities, cigarettes, etc, in the same way as I did for Sudoku earlier (XXX).
Let's run it:

```
\$ clang -emit-llvm -c -g klee_zebra1.c
...

\$ klee klee_zebra1.bc
KLEE: output directory is "/home/klee/klee-out-97"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_zebra1.c:130: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 761
KLEE: done: completed paths = 55
KLEE: done: generated tests = 55
```

It works for 7 seconds on my Intel Core i3-3110M 2.4GHz notebook. Let's find out path, where `klee_assert()` has been executed:

```
\$ ls klee-last | grep err
test000051.external.err

\$ ktest-tool --write-ints klee-last/test000051.ktest | less

ktest file : 'klee-last/test000051.ktest'
args       : ['klee_zebra1.bc']
num objects: 25
object    0: name: b'Yellow'
object    0: size: 4
object    0: data: 1
object    1: name: b'Blue'
object    1: size: 4
object    1: data: 2
object    2: name: b'Red'
object    2: size: 4
object    2: data: 3
object    3: name: b'Ivory'
object    3: size: 4
object    3: data: 4

...

object   21: name: b'Horse'
object   21: size: 4
object   21: data: 2
object   22: name: b'Snails'
object   22: size: 4
object   22: data: 3
object   23: name: b'Dog'
object   23: size: 4
object   23: data: 4
object   24: name: b'Zebra'
object   24: size: 4
object   24: data: 5
```

This is indeed correct solution.
`klee_assume()` also can be used this time:

```
int main()
{
        int Yellow, Blue, Red, Ivory, Green;
        int Norwegian, Ukrainian, Englishman, Spaniard, Japanese;
        int Water, Tea, Milk, OrangeJuice, Coffee;
        int Kools, Chesterfield, OldGold, LuckyStrike, Parliament;
        int Fox, Horse, Snails, Dog, Zebra;

        klee_make_symbolic(&Yellow, sizeof(int), "Yellow");
        klee_make_symbolic(&Blue, sizeof(int), "Blue");
        klee_make_symbolic(&Red, sizeof(int), "Red");
        klee_make_symbolic(&Ivory, sizeof(int), "Ivory");
        klee_make_symbolic(&Green, sizeof(int), "Green");

        klee_make_symbolic(&Norwegian, sizeof(int), "Norwegian");
        klee_make_symbolic(&Ukrainian, sizeof(int), "Ukrainian");
        klee_make_symbolic(&Englishman, sizeof(int), "Englishman");
        klee_make_symbolic(&Spaniard, sizeof(int), "Spaniard");
        klee_make_symbolic(&Japanese, sizeof(int), "Japanese");

        klee_make_symbolic(&Water, sizeof(int), "Water");
        klee_make_symbolic(&Tea, sizeof(int), "Tea");
        klee_make_symbolic(&Milk, sizeof(int), "Milk");
        klee_make_symbolic(&OrangeJuice, sizeof(int), "OrangeJuice");
        klee_make_symbolic(&Coffee, sizeof(int), "Coffee");

        klee_make_symbolic(&Kools, sizeof(int), "Kools");
        klee_make_symbolic(&Chesterfield, sizeof(int), "Chesterfield");
        klee_make_symbolic(&OldGold, sizeof(int), "OldGold");
        klee_make_symbolic(&LuckyStrike, sizeof(int), "LuckyStrike");
        klee_make_symbolic(&Parliament, sizeof(int), "Parliament");

        klee_make_symbolic(&Fox, sizeof(int), "Fox");
        klee_make_symbolic(&Horse, sizeof(int), "Horse");
        klee_make_symbolic(&Snails, sizeof(int), "Snails");
        klee_make_symbolic(&Dog, sizeof(int), "Dog");
        klee_make_symbolic(&Zebra, sizeof(int), "Zebra");

        // limits.
        klee_assume (Yellow>=1 && Yellow<=5);
        klee_assume (Blue>=1 && Blue<=5);
        klee_assume (Red>=1 && Red<=5);
        klee_assume (Ivory>=1 && Ivory<=5);
        klee_assume (Green>=1 && Green<=5);

        klee_assume (Norwegian>=1 && Norwegian<=5);
        klee_assume (Ukrainian>=1 && Ukrainian<=5);
        klee_assume (Englishman>=1 && Englishman<=5);
        klee_assume (Spaniard>=1 && Spaniard<=5);
        klee_assume (Japanese>=1 && Japanese<=5);

        klee_assume (Water>=1 && Water<=5);
        klee_assume (Tea>=1 && Tea<=5);
        klee_assume (Milk>=1 && Milk<=5);
        klee_assume (OrangeJuice>=1 && OrangeJuice<=5);
        klee_assume (Coffee>=1 && Coffee<=5);

        klee_assume (Kools>=1 && Kools<=5);
        klee_assume (Chesterfield>=1 && Chesterfield<=5);
        klee_assume (OldGold>=1 && OldGold<=5);
        klee_assume (LuckyStrike>=1 && LuckyStrike<=5);
        klee_assume (Parliament>=1 && Parliament<=5);
```

```
    klee_assume (Fox>=1 && Fox<=5);
    klee_assume (Horse>=1 && Horse<=5);
    klee_assume (Snails>=1 && Snails<=5);
    klee_assume (Dog>=1 && Dog<=5);
    klee_assume (Zebra>=1 && Zebra<=5);

    // colors are distinct for all 5 houses:
    klee_assume (((1<<Yellow) | (1<<Blue) | (1<<Red) | (1<<Ivory) | (1<<Green))==0x3E); // 111110

    // all nationalities are living in different houses:
    klee_assume (((1<<Norwegian) | (1<<Ukrainian) | (1<<Englishman) | (1<<Spaniard) | (1<<Japanese)
)==0x3E); // 111110

    // so are beverages:
    klee_assume (((1<<Water) | (1<<Tea) | (1<<Milk) | (1<<OrangeJuice) | (1<<Coffee))==0x3E); //
111110

    // so are cigarettes:
    klee_assume (((1<<Kools) | (1<<Chesterfield) | (1<<OldGold) | (1<<LuckyStrike) | (1<<Parliament
))==0x3E); // 111110

    // so are pets:
    klee_assume (((1<<Fox) | (1<<Horse) | (1<<Snails) | (1<<Dog) | (1<<Zebra))==0x3E); // 111110

    // main constraints of the puzzle:

    // 2.The Englishman lives in the red house.
    klee_assume (Englishman==Red);

    // 3.The Spaniard owns the dog.
    klee_assume (Spaniard==Dog);

    // 4.Coffee is drunk in the green house.
    klee_assume (Coffee==Green);

    // 5.The Ukrainian drinks tea.
    klee_assume (Ukrainian==Tea);

    // 6.The green house is immediately to the right of the ivory house.
    klee_assume (Green==Ivory+1);

    // 7.The Old Gold smoker owns snails.
    klee_assume (OldGold==Snails);

    // 8.Kools are smoked in the yellow house.
    klee_assume (Kools==Yellow);

    // 9.Milk is drunk in the middle house.
    klee_assume (Milk==3); // i.e., 3rd house

    // 10.The Norwegian lives in the first house.
    klee_assume (Norwegian==1);

    // 11.The man who smokes Chesterfields lives in the house next to the man with the fox.
    klee_assume (Chesterfield==Fox+1 || Chesterfield==Fox-1); // left or right

    // 12.Kools are smoked in the house next to the house where the horse is kept.
    klee_assume (Kools==Horse+1 || Kools==Horse-1); // left or right
```

```
        // 13.The Lucky Strike smoker drinks orange juice.
        klee_assume (LuckyStrike==OrangeJuice);

        // 14.The Japanese smokes Parliaments.
        klee_assume (Japanese==Parliament);

        // 15.The Norwegian lives next to the blue house.
        klee_assume (Norwegian==Blue+1 || Norwegian==Blue-1); // left or right

        // all constraints are satisfied at this point
        // force KLEE to produce .err file:
        klee_assert(0);
};
```

... and this version works slightly faster (5 seconds), maybe because KLEE is aware of this intrinsic and handles it in a special way?

6.3 Sudoku

I've also rewritten Sudoku example (XXX) for KLEE:

```
#include <stdint.h>

/*
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
*/

uint8_t cells[9][9];

// http://www.norvig.com/sudoku.html
// http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
char *puzzle="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..";

int main()
{
        klee_make_symbolic(cells, sizeof cells, "cells");

        // process text line:
        for (int row=0; row<9; row++)
                for (int column=0; column<9; column++)
                {
                        char c=puzzle[row*9 + column];
                        if (c!='.')
                        {
                                if (cells[row][column]!=c-'0') return 0;
                        }
                        else
                        {
                                // limit cells values to 1..9:
                                if (cells[row][column]<1) return 0;
                                if (cells[row][column]>9) return 0;
                        };
                };

        // for all 9 rows
        for (int row=0; row<9; row++)
        {

                if (((1<<cells[row][0]) |
                        (1<<cells[row][1]) |
                        (1<<cells[row][2]) |
                        (1<<cells[row][3]) |
                        (1<<cells[row][4]) |
                        (1<<cells[row][5]) |
                        (1<<cells[row][6]) |
                        (1<<cells[row][7]) |
                        (1<<cells[row][8]))!=0x3FE ) return 0; // 11 1111 1110
        };

        // for all 9 columns
        for (int c=0; c<9; c++)
        {
                if (((1<<cells[0][c]) |
                        (1<<cells[1][c]) |
                        (1<<cells[2][c]) |
                        (1<<cells[3][c]) |
                        (1<<cells[4][c]) |
                        (1<<cells[5][c]) |
                        (1<<cells[6][c]) |
                        (1<<cells[7][c]) |
                        (1<<cells[8][c]))!=0x3FE ) return 0; // 11 1111 1110
        };

        // enumerate all 9 squares
        for (int r=0; r<9; r+=3)
                for (int c=0; c<9; c+=3)
                {
                        // add constraints for each 3*3 square:
                        if ((1<<cells[r+0][c+0] |
                                1<<cells[r+0][c+1] |
                                1<<cells[r+0][c+2] |
                                1<<cells[r+1][c+0] |
                                1<<cells[r+1][c+1] |
                                1<<cells[r+1][c+2] |
                                1<<cells[r+2][c+0] |
                                1<<cells[r+2][c+1] |
                                1<<cells[r+2][c+2])!=0x3FE ) return 0; // 11 1111 1110
                };

        // at this point, all constraints should be satisfied
        klee_assert(0);
};
```

Let’s run it:

```
\$ clang -emit-llvm -c -g klee_sudoku_or1.c
...

\$ time klee klee_sudoku_or1.bc
```

```
KLEE: output directory is "/home/klee/klee-out-98"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:93: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 7512
KLEE: done: completed paths = 161
KLEE: done: generated tests = 161

real    3m44.111s
user    3m43.319s
sys     0m0.951s
```

Now this is really slower (on my Intel Core i3-3110M 2.4GHz notebook) in comparison to Z3Py solution (XXX).
But the answer is correct:

```
\$ ls klee-last | grep err
test000161.external.err

\$ ktest-tool --write-ints klee-last/test000161.ktest
ktest file : 'klee-last/test000161.ktest'
args       : ['klee_sudoku_or1.bc']
num objects: 1
object    0: name: b'cells'
object    0: size: 81
object    0: data: b'\x01\x04\x05\x03\x02\x07\x06\t\x08\x08\x03\t\x06\x05\x04\x01\x02\x07\x06\x07\x02\t
    \x01\x08\x05\x04\x03\x04\t\x06\x01\x08\x05\x03\x07\x02\x02\x01\x08\x04\x07\x03\t\x05\x06\x07\x05\
    x03\x02\t\x06\x04\x08\x01\x03\x06\x07\x05\x04\x02\x08\x01\t\t\x08\x04\x07\x06\x01\x02\x03\x05\x05\
    x02\x01\x08\x03\t\x07\x06\x04'
```

`t` is character with ASCII code of 9 in C/C++, and KLEE attempts to treat byte array as C/C++ string, so it shows some values in such way. We can just remember that there is 9 at the each place where we see
`t`. The solution, while not properly formatted, correct indeed (XXX).

By the way, at lines 42, 43 you may see how we tell to KLEE that all array elements must be within some limits. If we comment these lines out, we've got this:

```
\$ time klee klee_sudoku_or1.bc
KLEE: output directory is "/home/klee/klee-out-100"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:51: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:51: overshift error
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_sudoku_or1.c:51: overshift error
KLEE: NOTE: now ignoring this error at this location
...
```

KLEE warns us that shift value at line 51 is too big. Indeed, KLEE may try all byte values up to 255 (0xFF), which are pointless to use there, and may indicate error or bug, so KLEE warns about it.

Now let's use `klee_assume()` again:

```
#include <stdint.h>

/*
coordinates:
------------------------------
00 01 02 | 03 04 05 | 06 07 08
10 11 12 | 13 14 15 | 16 17 18
```

```
20 21 22 | 23 24 25 | 26 27 28
------------------------------
30 31 32 | 33 34 35 | 36 37 38
40 41 42 | 43 44 45 | 46 47 48
50 51 52 | 53 54 55 | 56 57 58
------------------------------
60 61 62 | 63 64 65 | 66 67 68
70 71 72 | 73 74 75 | 76 77 78
80 81 82 | 83 84 85 | 86 87 88
------------------------------
*/

uint8_t cells[9][9];

// http://www.norvig.com/sudoku.html
// http://www.mirror.co.uk/news/weird-news/worlds-hardest-sudoku-can-you-242294
char *puzzle="..53.....8......2..7..1.5..4....53...1..7...6..32...8..6.5....9..4....3......97..";

int main()
{
        klee_make_symbolic(cells, sizeof cells, "cells");

        // process text line:
        for (int row=0; row<9; row++)
                for (int column=0; column<9; column++)
                {
                        char c=puzzle[row*9 + column];
                        if (c!='.')
                                klee_assume (cells[row][column]==c-'0');
                        else
                        {
                                klee_assume (cells[row][column]>=1);
                                klee_assume (cells[row][column]<=9);
                        };
                };


        // for all 9 rows
        for (int row=0; row<9; row++)
        {

                klee_assume (((1<<cells[row][0]) |
                                (1<<cells[row][1]) |
                                (1<<cells[row][2]) |
                                (1<<cells[row][3]) |
                                (1<<cells[row][4]) |
                                (1<<cells[row][5]) |
                                (1<<cells[row][6]) |
                                (1<<cells[row][7]) |
                                (1<<cells[row][8]))==0x3FE ); // 11 1111 1110

        };

        // for all 9 columns
        for (int c=0; c<9; c++)
        {
                klee_assume (((1<<cells[0][c]) |
                                (1<<cells[1][c]) |
                                (1<<cells[2][c]) |
                                (1<<cells[3][c]) |
                                (1<<cells[4][c]) |
                                (1<<cells[5][c]) |
                                (1<<cells[6][c]) |
                                (1<<cells[7][c]) |
                                (1<<cells[8][c]))==0x3FE ); // 11 1111 1110
        };


        // enumerate all 9 squares
        for (int r=0; r<9; r+=3)
                for (int c=0; c<9; c+=3)
                {
                        // add constraints for each 3*3 square:
                        klee_assume ((1<<cells[r+0][c+0] |
                                        1<<cells[r+0][c+1] |
                                        1<<cells[r+0][c+2] |
                                        1<<cells[r+1][c+0] |
                                        1<<cells[r+1][c+1] |
                                        1<<cells[r+1][c+2] |
                                        1<<cells[r+2][c+0] |
                                        1<<cells[r+2][c+1] |
                                        1<<cells[r+2][c+2])==0x3FE ); // 11 1111 1110
                };

        // at this point, all constraints should be satisfied
        klee_assert(0);
};
```

```
\$ time klee klee_sudoku_or2.bc
KLEE: output directory is "/home/klee/klee-out-99"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_sudoku_or2.c:93: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 7119
KLEE: done: completed paths = 1
KLEE: done: generated tests = 1

real    0m35.312s
user    0m34.945s
sys     0m0.318s
```

That works much faster: perhaps KLEE indeed handle this intrinsic in a special way. And, as we see, the only one path is generated (one we actually interesting in it) instead of 161.

It's still much slower than Z3Py solution, though.

6.4 Unit test: HTML color

The most popular ways to represent HTML/CSS color is by English name (e.g., "red") and by 6-digit hexadecimal number (e.g., "#0077CC"). There is third, less popular way: if each byte in hexadecimal number has two doubling digits, it can be *abbreviated*, thus, "#0077CC" can be written just as "#07C".

Let's write a function to convert 3 color components into name (if possible, first priority), 3-digit hexadecimal form (if possible, second priority), or as 6-digit hexadecimal form (as a last resort).

```
#include <string.h>
#include <stdio.h>
#include <stdint.h>

void HTML_color(uint8_t R, uint8_t G, uint8_t B, char* out)
```

```
{
        if (R==0xFF && G==0 && B==0)
        {
                strcpy (out, "red");
                return;
        };

        if (R==0x0 && G==0xFF && B==0)
        {
                strcpy (out, "green");
                return;
        };

        if (R==0 && G==0 && B==0xFF)
        {
                strcpy (out, "blue");
                return;
        };

        // abbreviated hexadecimal
        if (R>>4==(R&0xF) && G>>4==(G&0xF) && B>>4==(B&0xF))
        {
                sprintf (out, "#%X%X%X", R&0xF, G&0xF, B&0xF);
                return;
        };

        // last resort
        sprintf (out, "#%02X%02X%02X", R, G, B);
};

int main()
{
        uint8_t R, G, B;
        klee_make_symbolic (&R, sizeof R, "R");
        klee_make_symbolic (&G, sizeof R, "G");
        klee_make_symbolic (&B, sizeof R, "B");

        char tmp[16];

        HTML_color(R, G, B, tmp);
};
```

There are 5 paths in function, and let's see, if KLEE could find them all? It's indeed so:

```
\$ clang -emit-llvm -c -g color.c

\$ klee color.bc
KLEE: output directory is "/home/klee/klee-out-134"
KLEE: WARNING: undefined reference to function: sprintf
KLEE: WARNING: undefined reference to function: strcpy
KLEE: WARNING ONCE: calling external: strcpy(51867584, 51598960)
KLEE: ERROR: /home/klee/color.c:33: external call with symbolic argument: sprintf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/color.c:28: external call with symbolic argument: sprintf
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 479
KLEE: done: completed paths = 19
KLEE: done: generated tests = 5
```

We can ignore calls to strcpy() and sprintf(), because we are not really interesting in state of `out` variable.

So there are exactly 5 paths:

```
\$ ls klee-last
assembly.ll   run.stats             test000003.ktest     test000005.ktest
info          test000001.ktest      test000003.pc        test000005.pc
messages.txt  test000002.ktest      test000004.ktest     warnings.txt
run.istats    test000003.exec.err   test000005.exec.err
```

1st set of input variables will result in “red” string:

```
\$ ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\xff'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x00'
```

2nd set of input variables will result in “green” string:

```
\$ ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\x00'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\xff'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x00'
```

3rd set of input variables will result in “#010000” string:

```
\$ ktest-tool --write-ints klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\x01'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x00'
```

4th set of input variables will result in “blue” string:

```
\$ ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\x00'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\xff'
```

5th set of input variables will result in “#F01” string:

```
\$ ktest-tool --write-ints klee-last/test000005.ktest
ktest file : 'klee-last/test000005.ktest'
args       : ['color.bc']
num objects: 3
object    0: name: b'R'
object    0: size: 1
object    0: data: b'\xff'
object    1: name: b'G'
object    1: size: 1
object    1: data: b'\x00'
object    2: name: b'B'
object    2: size: 1
object    2: data: b'\x11'
```

These 5 sets of input variables can form a unit test for our function.

6.5 Unit test: strcmp() function

The standard `strcmp()` function from C library can return 0, -1 and 1, depending of comparison result.

Here is my own implementation of strcmp():

```
int my_strcmp(const char *s1, const char *s2)
{
        int ret = 0;

        while (1)
        {
                ret = *(unsigned char *) s1 - *(unsigned char *) s2;
                if (ret!=0)
                        break;
                if ((*s1==0) || (*s2)==0)
                        break;
                s1++;
                s2++;
        };

        if (ret < 0)
        {
                return -1;
        } else if (ret > 0)
        {
                return 1;
        }

        return 0;
}

int main()
{
        char input1[2];
        char input2[2];

        klee_make_symbolic(input1, sizeof input1, "input1");
        klee_make_symbolic(input2, sizeof input2, "input2");

        klee_assume((input1[0]>='a') && (input1[0]<='z'));
        klee_assume((input2[0]>='a') && (input2[0]<='z'));

        klee_assume(input1[1]==0);
        klee_assume(input2[1]==0);

        my_strcmp (input1, input2);
};
```

Let's find out, if KLEE is capable of finding all three paths? I intentionaly made things simpler for KLEE by limiting input arrays to two 2 bytes or to 1 character + terminal zero byte.

```
$ clang -emit-llvm -c -g strcmp.c

$ klee strcmp.bc
KLEE: output directory is "/home/klee/klee-out-131"
KLEE: ERROR: /home/klee/strcmp.c:35: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/strcmp.c:36: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 137
KLEE: done: completed paths = 5
KLEE: done: generated tests = 5

$ ls klee-last
assembly.ll   run.stats            test000002.ktest     test000004.ktest
info          test000001.ktest     test000002.pc        test000005.ktest
messages.txt  test000001.pc        test000002.user.err  warnings.txt
run.istats    test000001.user.err  test000003.ktest
```

The first two errors are about klee_assume(). These are input values on which klee_assume() calls are stuck. We can ignore them, or take a peek out of curiosity:

```
\$ ktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'\x00\x00'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'\x00\x00'

\$ ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'a\xff'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'\x00\x00'
```

Three rest files are the input values for each path inside of my implementation of strcmp():

```
\$ ktest-tool --write-ints klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'b\x00'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'c\x00'

\$ ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'c\x00'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'a\x00'

\$ ktest-tool --write-ints klee-last/test000005.ktest
ktest file : 'klee-last/test000005.ktest'
args       : ['strcmp.bc']
num objects: 2
object    0: name: b'input1'
object    0: size: 2
object    0: data: b'a\x00'
object    1: name: b'input2'
object    1: size: 2
object    1: data: b'a\x00'
```

3rd is about first argument ("b") is lesser than the second ("c"). 4th is opposite ("c" and "a"). 5th is when they are equal ("a" and "a").

Using these 3 test cases, we've got full coverage of our implementation of strcmp().

6.6 UNIX date/time

UNIX date/time[58] is a number of seconds that have elapsed since 1-Jan-1970 00:00 UTC. C/C++ gmtime() function is used to decode this value into human-readable date/time.

Here is a piece of code I've copypasted from some ancient version of Minix OS (http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/gmtime.c) and reworked slightly:

```
#include <stdint.h>
#include <time.h>
#include <assert.h>

/*
 * copypasted and reworked from
 * http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/loc_time.h
 * http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/misc.c
 * http://www.cise.ufl.edu/~cop4600/cgi-bin/lxr/http/source.cgi/lib/ansi/gmtime.c
 */

#define YEAR0           1900
#define EPOCH_YR        1970
```

[58]https://en.wikipedia.org/wiki/Unix_time

```
#define SECS_DAY        (24L * 60L * 60L)
#define YEARSIZE(year)  (LEAPYEAR(year) ? 366 : 365)

const int _ytab[2][12] =
{
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
        { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};

const char *_days[] =
{
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
};

const char *_months[] =
{
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
};

#define LEAPYEAR(year)  (!((year) % 4) && (((year) % 100) || !((year) % 400)))

void decode_UNIX_time(const time_t time)
{
        unsigned int dayclock, dayno;
        int year = EPOCH_YR;

        dayclock = (unsigned long)time % SECS_DAY;
        dayno = (unsigned long)time / SECS_DAY;

        int seconds = dayclock % 60;
        int minutes = (dayclock % 3600) / 60;
        int hour = dayclock / 3600;
        int wday = (dayno + 4) % 7;
        while (dayno >= YEARSIZE(year))
        {
                dayno -= YEARSIZE(year);
                year++;
        }

        year = year - YEAR0;

        int month = 0;

        while (dayno >= _ytab[LEAPYEAR(year)][month])
        {
                dayno -= _ytab[LEAPYEAR(year)][month];
                month++;
        }

        char *s;
        switch (month)
        {
                case 0: s="January"; break;
                case 1: s="February"; break;
                case 2: s="March"; break;
                case 3: s="April"; break;
74                  case 4: s="May"; break;
75                  case 5: s="June"; break;
76                  case 6: s="July"; break;
77                  case 7: s="August"; break;
78                  case 8: s="September"; break;
79                  case 9: s="October"; break;
80                  case 10: s="November"; break;
81                  case 11: s="December"; break;
82                  default:
83                           assert(0);
84          };
85
86          printf ("%04d-%s-%02d %02d:%02d:%02d\n", YEAR0+year, s, dayno+1, hour, minutes, seconds);
87          printf ("week day: %s\n", _days[wday]);
88  }
89
90  int main()
91  {
92          uint32_t time;
93
94          klee_make_symbolic(&time, sizeof time, "time");
95
96          decode_UNIX_time(time);
97
98          return 0;
99  }
```

Let's try it:

```
\$ clang -emit-llvm -c -g klee_time1.c
...

\$ klee klee_time1.bc
KLEE: output directory is "/home/klee/klee-out-107"
KLEE: WARNING: undefined reference to function: printf
KLEE: ERROR: /home/klee/klee_time1.c:86: external call with symbolic argument: printf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_time1.c:83: ASSERTION FAIL: 0
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 101579
KLEE: done: completed paths = 1635
KLEE: done: generated tests = 2
```

Wow, assert() at line 83 has been triggered, why? Let's see a value of UNIX time which triggers it:

```
\$ ls klee-last | grep err
test000001.exec.err
test000002.assert.err

\$ ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_time1.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 978278400
```

Let's decode this value using UNIX date utility:

```
\$ date -u --date='@978278400'
Sun Dec 31 16:00:00 UTC 2000
```

After may investigation, I've found that `month` variable can hold incorrect value of 12 (while 11 is maximal, for December), because LEAPYEAR() macro should receive year number as 2000, not as 100. So I've introduced a bug during rewritting this function, and KLEE found it!

Just interesting, what would be if I'll replace switch() to array of strings, like it usually happens in concise C/C++ code?

```
        ...

const char *_months[] =
{
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", "December"
};

        ...

        while (dayno >= _ytab[LEAPYEAR(year)][month])
        {
                dayno -= _ytab[LEAPYEAR(year)][month];
                month++;
        }

        char *s=_months[month];

        printf ("%04d-%s-%02d %02d:%02d:%02d\n", YEAR0+year, s, dayno+1, hour, minutes, seconds);
        printf ("week day: %s\n", _days[wday]);

        ...
```

KLEE detects attempt to read beyond array boundaries:

```
\$ klee klee_time2.bc
KLEE: output directory is "/home/klee/klee-out-108"
KLEE: WARNING: undefined reference to function: printf
KLEE: ERROR: /home/klee/klee_time2.c:69: external call with symbolic argument: printf
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_time2.c:67: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 101716
KLEE: done: completed paths = 1635
KLEE: done: generated tests = 2
```

This is the same UNIX time value we've already seen:

```
\$ ls klee-last | grep err
test000001.exec.err
test000002.ptr.err

\$ ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_time2.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 978278400
```

So, if this piece of code can be triggered on remote computer, with this input value (*input of death*), it's possible to crash the process (with some luck, though).

OK, now I'm fixing a bug by moving year subtracting expression to line 43, and let's find, what UNIX time value corresponds to some fancy date like 2022-February-2?

```
#include <stdint.h>
#include <time.h>
#include <assert.h>

#define YEAR0           1900
#define EPOCH_YR        1970
#define SECS_DAY        (24L * 60L * 60L)
#define YEARSIZE(year)  (LEAPYEAR(year) ? 366 : 365)

const int _ytab[2][12] =
{
        { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 },
        { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }
};

#define LEAPYEAR(year)  (!((year) % 4) && (((year) % 100) || !((year) % 400)))

void decode_UNIX_time(const time_t time)
{
        unsigned int dayclock, dayno;
        int year = EPOCH_YR;

        dayclock = (unsigned long)time % SECS_DAY;
        dayno = (unsigned long)time / SECS_DAY;

        int seconds = dayclock % 60;
        int minutes = (dayclock % 3600) / 60;
        int hour = dayclock / 3600;
        int wday = (dayno + 4) % 7;
        while (dayno >= YEARSIZE(year))
        {
                dayno -= YEARSIZE(year);
                year++;
        }

        int month = 0;

        while (dayno >= _ytab[LEAPYEAR(year)][month])
        {
                dayno -= _ytab[LEAPYEAR(year)][month];
                month++;
        }
        year = year - YEAR0;

        if (YEAR0+year==2022 && month==1 && dayno+1==22)
                klee_assert(0);
}
int main()
{
        uint32_t time;

        klee_make_symbolic(&time, sizeof time, "time");

        decode_UNIX_time(time);

        return 0;
}
```

```
\$ clang -emit-llvm -c -g klee_time3.c
```

```
...

\$ klee klee_time3.bc
KLEE: output directory is "/home/klee/klee-out-109"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_time3.c:47: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 101087
KLEE: done: completed paths = 1635
KLEE: done: generated tests = 1635

\$ ls klee-last | grep err
test000587.external.err

\$ ktest-tool --write-ints klee-last/test000587.ktest
ktest file : 'klee-last/test000587.ktest'
args       : ['klee_time3.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 1645488640

\$ date -u --date='@1645488640'
Tue Feb 22 00:10:40 UTC 2022
```

Success, but hours/minutes/seconds are seems random – they are random indeed, because, KLEE satisfied all constraints we've put, nothing else. We didn't ask it to set hours/minutes/seconds to zeroes.

Let's add constraints to hours/minutes/seconds as well:

```
        ...

        if (YEAR0+year==2022 && month==1 && dayno+1==22 && hour==22 && minutes==22 && seconds==22)
                klee_assert(0);

        ...
```

Let's run it and check...

```
\$ ktest-tool --write-ints klee-last/test000597.ktest
ktest file : 'klee-last/test000597.ktest'
args       : ['klee_time3.bc']
num objects: 1
object    0: name: b'time'
object    0: size: 4
object    0: data: 1645568542

\$ date -u --date='@1645568542'
Tue Feb 22 22:22:22 UTC 2022
```

Now that is precise.

Yes, of course, C/C++ libraries has function(s) to encode human-readable date into UNIX time value, but what we've got here is KLEE working *antipode* of decoding function, *inverse function* in a way.

6.7 Inverse function for base64 decoder

It's piece of cake for KLEE to reconstruct input base64 string given just base64 decoder code without corresponding encoder code. I've copypasted this piece of code from `http://www.opensource.apple.com/source/QuickTimeStreamingServer/QuickTimeStreami` `CommonUtilitiesLib/base64.c`.

We add constraints (lines 84, 85) so that output buffer must have byte values from 0 to 15. We also tell to KLEE that the Base64decode() function must return 16 (i.e., size of output buffer in bytes, line 82).

```c
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

// copypasted from http://www.opensource.apple.com/source/QuickTimeStreamingServer/
    QuickTimeStreamingServer-452/CommonUtilitiesLib/base64.c

static const unsigned char pr2six[256] =
{
        /* ASCII table */
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 62, 64, 64, 64, 63,
        52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 64, 64, 64, 64, 64, 64,
        64,  0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
        15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 64, 64, 64, 64, 64,
        64, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
        41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64,
        64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64, 64
};

int Base64decode(char *bufplain, const char *bufcoded)
{
        int nbytesdecoded;
        register const unsigned char *bufin;
        register unsigned char *bufout;
        register int nprbytes;

        bufin = (const unsigned char *) bufcoded;
        while (pr2six[*(bufin++)] <= 63);
        nprbytes = (bufin - (const unsigned char *) bufcoded) - 1;
        nbytesdecoded = ((nprbytes + 3) / 4) * 3;

        bufout = (unsigned char *) bufplain;
        bufin = (const unsigned char *) bufcoded;

        while (nprbytes > 4) {
                *(bufout++) =
                        (unsigned char) (pr2six[*bufin] << 2 | pr2six[bufin[1]] >> 4);
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[1]] << 4 | pr2six[bufin[2]] >> 2);
                *(bufout++) =
                        (unsigned char) (pr2six[bufin[2]] << 6 | pr2six[bufin[3]]);
                bufin += 4;
                nprbytes -= 4;
        }

        /* Note: (nprbytes == 1) would be an error, so just ingore that case */
        if (nprbytes > 1) {
                *(bufout++) =
                        (unsigned char) (pr2six[*bufin] << 2 | pr2six[bufin[1]] >> 4);
        }
59          if (nprbytes > 2) {
60                  *(bufout++) =
61                          (unsigned char) (pr2six[bufin[1]] << 4 | pr2six[bufin[2]] >> 2);
62          }
63          if (nprbytes > 3) {
64                  *(bufout++) =
65                          (unsigned char) (pr2six[bufin[2]] << 6 | pr2six[bufin[3]]);
66          }
67
68          *(bufout++) = '\0';
69          nbytesdecoded -= (4 - nprbytes) & 3;
70          return nbytesdecoded;
71  }
72
73  int main()
74  {
75          char input[32];
76          uint8_t output[16+1];
77
78          klee_make_symbolic(input, sizeof input, "input");
79
80          klee_assume(input[31]==0);
81
82          klee_assume (Base64decode(output, input)==16);
83
84          for (int i=0; i<16; i++)
85                  klee_assume (output[i]==i);
86
87          klee_assert(0);
88
89          return 0;
90  }
```

```
\$ clang -emit-llvm -c -g klee_base64.c
...

\$ klee klee_base64.bc
KLEE: output directory is "/home/klee/klee-out-99"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_base64.c:99: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_base64.c:104: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_base64.c:85: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_base64.c:81: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_base64.c:65: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location

...
```

We're interesting in the second error, where `klee_assert()` has been triggered:

```
\$ ls klee-last | grep err
test000001.user.err
test000002.external.err
test000003.ptr.err
test000004.ptr.err
test000005.ptr.err
```

```
\$ ktest-tool --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['klee_base64.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 32
object    0: data: b'AAECAwQFBgcICQoLDA0OD4\x00\xff\xff\xff\xff\xff\xff\xff\xff\x00'
```

This is indeed a real base64 string, terminated with the zero byte, just as it's requested by C/C++ standards. The final zero byte at 31th byte (starting at zeroth byte) is our deed: so that KLEE would report lesser number of errors.

The base64 string is indeed correct:

```
\$ echo AAECAwQFBgcICQoLDA0OD4 | base64 -d | hexdump -C
base64: invalid input
00000000  00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f  |................|
00000010
```

base64 decoder Linux utility I've just run blaming for "invalid input" – it means input string is not properly padded: Now let's pad it manually, and decoder utility will no complain anymore:

```
\$ echo AAECAwQFBgcICQoLDA0OD4== | base64 -d | hexdump -C
00000000  00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f  |................|
00000010
```

The reason our generated base64 string is not padded is because base64 decoders are usually discards padding symbols ("=") at the end. In other words, they are not require them, so is the case of our decoder. Hence, padding symbols are left unnoticed to KLEE.

So we again made *antipode* or *inverse function* of base64 decoder.

6.8 CRC

6.8.1 Buffer alteration case #1

Sometimes, you need to alter a piece of data which is *protected* by some kind of checksum or **CRC!**[59], and you can't change checksum or CRC value, but can alter piece of data so that checksum will remain the same.

Let's pretend, we've got a piece of data with "Hello, world!" string at the beginning and "and goodbye" string at the end. We can alter 14 characters at the middle, but for some reason, they must be in *a..z* limits, but we can put any characters there. CRC64 of the whole block must be `0x12345678abcdef12`.

Let's see[60]:

```
#include <string.h>
#include <stdint.h>

uint64_t crc64(uint64_t crc, unsigned char *buf, int len)
{
        int k;

        crc = ~crc;
        while (len--)
        {
                crc ^= *buf++;
                for (k = 0; k < 8; k++)
                        crc = crc & 1 ? (crc >> 1) ^ 0x42f0e1eba9ea3693 : crc >> 1;
        }
        return crc;
}

int main()
{
#define HEAD_STR "Hello, world!.. "
```

[59] **CRC!**

[60] There are several slightly different CRC64 implementations, the one I use here can also be different from popular ones.

```
#define HEAD_SIZE strlen(HEAD_STR)
#define TAIL_STR " ... and goodbye"
#define TAIL_SIZE strlen(TAIL_STR)
#define MID_SIZE 14 // work
#define BUF_SIZE HEAD_SIZE+TAIL_SIZE+MID_SIZE

        char buf[BUF_SIZE];

        klee_make_symbolic(buf, sizeof buf, "buf");

        klee_assume (memcmp (buf, HEAD_STR, HEAD_SIZE)==0);

        for (int i=0; i<MID_SIZE; i++)
                klee_assume (buf[HEAD_SIZE+i]>='a' && buf[HEAD_SIZE+i]<='z');

        klee_assume (memcmp (buf+HEAD_SIZE+MID_SIZE, TAIL_STR, TAIL_SIZE)==0);

        klee_assume (crc64 (0, buf, BUF_SIZE)==0x12345678abcdef12);

        klee_assert(0);

        return 0;
}
```

Since our code uses memcmp() standard C/C++ function, we need to add `--libc=uclibc` switch, so KLEE will use its own uClibc implementation.

```
\$ clang -emit-llvm -c -g klee_CRC64.c

\$ time klee --libc=uclibc klee_CRC64.bc
```

It takes about 1 minute (on my Intel Core i3-3110M 2.4GHz notebook) and we getting this:

```
...
real    0m52.643s
user    0m51.232s
sys     0m0.239s
...
\$ ls klee-last | grep err
test000001.user.err
test000002.user.err
test000003.user.err
test000004.external.err

\$ ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['klee_CRC64.bc']
num objects: 1
object    0: name: b'buf'
object    0: size: 46
object    0: data: b'Hello, world!.. qqlicayzceamyw ... and goodbye'
```

Maybe it's slow, but definitely faster than bruteforce. Indeed, $log_2 26^{14} \approx 65.8$ which is close to 64 bits. In other words, one need ≈ 14 latin characters to encode 64 bits. And KLEE + SMT solver needs 64 bits at some place it can alter to make final CRC64 value equal to what we defined.

I tried to reduce length of the *middle block* to 13 characters: no luck for KLEE then, it has no space enough.

6.8.2 Buffer alteration case #2

I went sadistic: what if the buffer must contain the CRC64 value which, after calculation of CRC64, will result in the same value? Fascinately, KLEE can solve this. The buffer will have the following format:

```
Hello, world! <8-bytes (64-bit value)> and goodbye <6 more bytes>
```

```
int main()
{
#define HEAD_STR "Hello, world!.. "
#define HEAD_SIZE strlen(HEAD_STR)
#define TAIL_STR " ... and goodbye"
#define TAIL_SIZE strlen(TAIL_STR)
// 8 bytes for 64-bit value:
#define MID_SIZE 8
#define BUF_SIZE HEAD_SIZE+TAIL_SIZE+MID_SIZE+6

        char buf[BUF_SIZE];

        klee_make_symbolic(buf, sizeof buf, "buf");

        klee_assume (memcmp (buf, HEAD_STR, HEAD_SIZE)==0);

        klee_assume (memcmp (buf+HEAD_SIZE+MID_SIZE, TAIL_STR, TAIL_SIZE)==0);

        uint64_t mid_value=*(uint64_t*)(buf+HEAD_SIZE);
        klee_assume (crc64 (0, buf, BUF_SIZE)==mid_value);

        klee_assert(0);

        return 0;
}
```

It works:

```
\$ time klee --libc=uclibc klee_CRC64.bc
...
real    5m17.081s
user    5m17.014s
sys     0m0.319s

\$ ls klee-last | grep err
test000001.user.err
test000002.user.err
test000003.external.err

\$ ktest-tool --write-ints klee-last/test000003.ktest
ktest file : 'klee-last/test000003.ktest'
args       : ['klee_CRC64.bc']
num objects: 1
object    0: name: b'buf'
object    0: size: 46
object    0: data: b'Hello, world!.. T+]\xb9A\x08\x0fq ... and goodbye\xb6\x8f\x9c\xd8\xc5\x00'
```

8 bytes between two strings is 64-bit value which equals to CRC64 of this whole block. Again, it's faster than brute-force way to find it. If to decrease last spare 6-byte buffer to 4 bytes or less, KLEE works so long so I've stopped it.

6.8.3 Recovering input data for given CRC32 value of it

I've always wanted to do so, but everyone knows this is impossible for input buffers larger than 4 bytes. As my experiments show, it's still possible for tiny input buffers of data, constrained in some way.

The CRC32 value of 6-byte "SILVER" string is known: `0xDFA3DFDD`. KLEE can find this 6-byte string, if it knows that each byte of input buffer is in *A..Z* limits:

```
#include <stdint.h>
#include <stdbool.h>
```

```

uint32_t crc32(uint32_t crc, unsigned char *buf, int len)
{
    int k;

    crc = ~crc;
    while (len--)
    {
        crc ^= *buf++;
        for (k = 0; k < 8; k++)
            crc = crc & 1 ? (crc >> 1) ^ 0xedb88320 : crc >> 1;
    }
    return ~crc;
}

#define SIZE 6

bool find_string(char str[SIZE])
{
        int i=0;
        for (i=0; i<SIZE; i++)
                if (str[i]<'A' || str[i]>'Z')
                        return false;

        if (crc32(0, &str[0], SIZE)!=0xDFA3DFDD)
                return false;

        // OK, input str is valid
        klee_assert(0); // force KLEE to produce .err file
        return true;
};

int main()
{
        uint8_t str[SIZE];

        klee_make_symbolic(str, sizeof str, "str");

        find_string(str);

        return 0;
}
```

```
\$ clang -emit-llvm -c -g klee_SILVER.c
...

\$ klee klee_SILVER.bc
...

\$ ls klee-last | grep err
test000013.external.err

\$ ktest-tool --write-ints klee-last/test000013.ktest
ktest file : 'klee-last/test000013.ktest'
args       : ['klee_SILVER.bc']
num objects: 1
object    0: name: b'str'
object    0: size: 6
object    0: data: b'SILVER'
```

Still, it's no magic: if to remove condition at lines 23..25 (i.e., if to relax constraints), KLEE will produce some other string, which will be still correct for the CRC32 value given.

It works, because 6 Latin characters in *A..Z* limits contain ≈ 28.2 bits: $log_2 26^6 \approx 28.2$, which is even smaller value than 32. In other words, the final CRC32 value holds enough bits to recover ≈ 28.2 bits of input.

The input buffer can be even bigger, if each byte of it will be in even tighter constraints (decimal digits, binary digits, etc).

6.8.4 In comparison with other hashing algorithms

Things are that easy for some other hashing algorithms like *fletcher checksum*, but not for cryptographically secure ones (like MD5, SHA1, etc), they are protected from such simple cryptoanalysis.

6.9 LZSS decompressor

I've googled for a very simple LZSS decompressor and landed at this page: http://www.opensource.apple.com/source/boot/boot-132/i386/boot2/lzss.c.

Let's pretend, we're looking at unknown compressing algorithm with no compressor available. Will it be possible to construct a compressed piece of data so that decompressor would generate data we need?

Here is my first experiment:

```
// copypasted from http://www.opensource.apple.com/source/boot/boot-132/i386/boot2/lzss.c
//
#include <string.h>
#include <stdint.h>
#include <stdbool.h>

//#define N         4096  /* size of ring buffer - must be power of 2 */
#define N         32  /* size of ring buffer - must be power of 2 */
#define F         18    /* upper limit for match_length */
#define THRESHOLD 2     /* encode string into position and length
                           if match_length is greater than this */
#define NIL       N     /* index for root of binary search trees */

int
decompress_lzss(uint8_t *dst, uint8_t *src, uint32_t srclen)
{
    /* ring buffer of size N, with extra F-1 bytes to aid string comparison */
    uint8_t *dststart = dst;
    uint8_t *srcend = src + srclen;
    int  i, j, k, r, c;
    unsigned int flags;
    uint8_t text_buf[N + F - 1];

    dst = dststart;
    srcend = src + srclen;
    for (i = 0; i < N - F; i++)
        text_buf[i] = ' ';
    r = N - F;
    flags = 0;
    for ( ; ; ) {
        if (((flags >>= 1) & 0x100) == 0) {
            if (src < srcend) c = *src++; else break;
            flags = c | 0xFF00;  /* uses higher byte cleverly */
        }   /* to count eight */
        if (flags & 1) {
            if (src < srcend) c = *src++; else break;
            *dst++ = c;
            text_buf[r++] = c;
            r &= (N - 1);
        } else {
            if (src < srcend) i = *src++; else break;
            if (src < srcend) j = *src++; else break;
            i |= ((j & 0xF0) << 4);
            j  =  (j & 0x0F) + THRESHOLD;
            for (k = 0; k <= j; k++) {
                c = text_buf[(i + k) & (N - 1)];
                *dst++ = c;
                text_buf[r++] = c;
                r &= (N - 1);
            }
        }
    }

    return dst - dststart;
}

int main()
{
#define COMPRESSED_LEN 15
        uint8_t input[COMPRESSED_LEN];
        uint8_t plain[24];
        uint32_t size=COMPRESSED_LEN;

        klee_make_symbolic(input, sizeof input, "input");

        decompress_lzss(plain, input, size);

        // https://en.wikipedia.org/wiki/
    Buffalo_buffalo_Buffalo_buffalo_buffalo_buffalo_Buffalo_buffalo
        for (int i=0; i<23; i++)
                klee_assume (plain[i]=="Buffalo buffalo Buffalo"[i]);

        klee_assert(0);

        return 0;
}
```

What I did is changing size of ring buffer from 4096 to 32, because if bigger, KLEE consumes all RAM it can. I found that KLEE can live with that small buffer. I also decreased `COMPRESSED_LEN` gradually to check, whether KLEE would find compressed piece of data, and it did:

```
\$ clang -emit-llvm -c -g klee_lzss.c
...

\$ time klee klee_lzss.bc
KLEE: output directory is "/home/klee/klee-out-7"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_lzss.c:122: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:47: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:37: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_lzss.c:124: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 41417919
KLEE: done: completed paths = 437820
KLEE: done: generated tests = 4

real    13m0.215s
user    11m57.517s
```

```
sys     1m2.187s

\$ ls klee-last | grep err
test000001.user.err
test000002.ptr.err
test000003.ptr.err
test000004.external.err

\$ ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['klee_lzss.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 15
object    0: data: b'\xffBuffalo \x01b\x0f\x03\r\x05'
```

KLEE consumed 1GB of RAM and worked for 15 minutes (on my Intel Core i3-3110M 2.4GHz notebook), but here it is, a 15 bytes which, if decompressed by our copypasted algorithm, will result in desired text!

During my experimentation, I've found that KLEE can do even more cooler thing, to find out size of compressed piece of data:

```
int main()
{
        uint8_t input[24];
        uint8_t plain[24];
        uint32_t size;

        klee_make_symbolic(input, sizeof input, "input");
        klee_make_symbolic(&size, sizeof size, "size");

        decompress_lzss(plain, input, size);

        for (int i=0; i<23; i++)
                klee_assume (plain[i]=="Buffalo buffalo Buffalo"[i]);

        klee_assert(0);

        return 0;
}
```

... but then KLEE works much slower, consumes much more RAM and I had success only with even smaller pieces of desired text.

So how LZSS works? Without peeking into Wikipedia, we can say that: if LZSS compressor observes some data it already had, it replaces the data with a link to some place in past with size. If it observes something yet unseen, it puts data as is. This is theory. This is indeed what we've got. Desired text is three "Buffalo" words, the first and the last are equivalent, but the second is *almost* equivalent, differing with first by one character.

That's what we see:

```
'\xffBuffalo \x01b\x0f\x03\r\x05'
```

Here is some control byte (0xff), "Buffalo" word is placed *as is*, then another control byte (0x01), then we see beginning of the second word ("b") and more control bytes, perhaps, links to the beginning of the buffer. These are command to decompressor, like, in plain English, "copy data from the buffer we've already done, from that place to that place", etc.

Interesting, is it possible to meddle into this piece of compressed data? Out of whim, can we force KLEE to find a compressed data, where not just "b" character has been placed *as is*, but also the second character of the word, i.e., "bu"?

I've modified main() function by adding `klee_assume()`: now the 11th byte of input (compressed) data (right after "b" byte) should have "u". I has no luck with 15 byte of compressed data, so I increased it to 16 bytes:

```
int main()
{
#define COMPRESSED_LEN 16
        uint8_t input[COMPRESSED_LEN];
        uint8_t plain[24];
        uint32_t size=COMPRESSED_LEN;
        klee_make_symbolic(input, sizeof input, "input");

        klee_assume(input[11]=='u');

        decompress_lzss(plain, input, size);

        for (int i=0; i<23; i++)
                klee_assume (plain[i]=="Buffalo buffalo Buffalo"[i]);

        klee_assert(0);

        return 0;
}
```

... and voila: KLEE found a compressed piece of data which satisfied our whimsical constraint:

```
\$ time klee klee_lzss.bc
KLEE: output directory is "/home/klee/klee-out-9"
KLEE: WARNING: undefined reference to function: klee_assert
KLEE: ERROR: /home/klee/klee_lzss.c:97: invalid klee_assume call (provably false)
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:47: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: ERROR: /home/klee/klee_lzss.c:37: memory error: out of bound pointer
KLEE: NOTE: now ignoring this error at this location
KLEE: WARNING ONCE: calling external: klee_assert(0)
KLEE: ERROR: /home/klee/klee_lzss.c:99: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location

KLEE: done: total instructions = 36700587
KLEE: done: completed paths = 369756
KLEE: done: generated tests = 4

real    12m16.983s
user    11m17.492s
sys     0m58.358s

\$ ktest-tool --write-ints klee-last/test000004.ktest
ktest file : 'klee-last/test000004.ktest'
args       : ['klee_lzss.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 16
object    0: data: b'\xffBuffalo \x13bu\x10\x02\r\x05'
```

So now we find a piece of data where two strings are placed *as is*: “Buffalo” and “bu”.

```
'\xffBuffalo \x13bu\x10\x02\r\x05'
```

Both pieces of compressed data, if feeded into our copypasted function, produce “Buffalo buffalo Buffalo” text string.

Please note, I still have no access to LZSS compressor code, and I didn’t get into LZSS decompressor details yet.

Unfortunately, things are not that cool: KLEE is very slow and I had success only with small pieces of text, and also ring buffer size had to be decreased significantly (original LZSS decompressor with ring buffer of 4096 bytes cannot decompress correctly what we found).

Nevertheless, it’s very impressive, taking into account the fact that we’re not getting into internals of this specific LZSS decompressor. Once more time, we’ve created *antipode* of decompressor, or *inverse function*.

6.10 Unit testing: simple expression evaluator (calculator)

I has been looking for simple expression evaluator (calculator in other words) which takes expression like “2+2” on input and gives answer. I’ve found one at http://stackoverflow.com/a/13895198. Unfortunately, it has no bugs, so I’ve introduced one: a token buffer (`buf[]` at line 31) is smaller than input buffer (`input[]` at line 19).

```
 1 // copypasted from http://stackoverflow.com/a/13895198 and reworked
 2 
 3 // Bare bones scanner and parser for the following LL(1) grammar:
 4 // expr -> term { [+-] term }     ; An expression is terms separated by add ops.
 5 // term -> factor { [*/] factor } ; A term is factors separated by mul ops.
 6 // factor -> unsigned_factor      ; A signed factor is a factor,
 7 //         | - unsigned_factor    ;   possibly with leading minus sign
 8 // unsigned_factor -> ( expr )    ; An unsigned factor is a parenthesized expression
 9 //         | NUMBER               ;   or a number
10 //
11 // The parser returns the floating point value of the expression.
12 
13 #include <string.h>
14 #include <stdio.h>
15 #include <stdlib.h>
16 #include <stdint.h>
17 #include <stdbool.h>
18 
19 char input[128];
20 int input_idx=0;
21 
22 char my_getchar()
23 {
24         char rt=input[input_idx];
25         input_idx++;
26         return rt;
27 };
28 
29 // The token buffer. We never check for overflow! Do so in production code.
30 // it's deliberately smaller than input[] so KLEE could find buffer overflow
31 char buf[64];
32 int n = 0;
33 
34 // The current character.
35 int ch;
36 
37 // The look-ahead token.  This is the 1 in LL(1).
38 enum { ADD_OP, MUL_OP, LEFT_PAREN, RIGHT_PAREN, NOT_OP, NUMBER, END_INPUT } look_ahead;
39 
40 // Forward declarations.
41 void init(void);
42 void advance(void);
43 int expr(void);
44 void error(char *msg);
45 
46 // Parse expressions, one per line.
47 int main(void)
48 {
49         // take input expression from input[]
50         //input[0]=0;
51         //strcpy (input, "2+2");
52         klee_make_symbolic(input, sizeof input, "input");
53         input[127]=0;
54 
55         init();
56         while (1)
57         {
58                 int val = expr();
```

```
                printf("%d\n", val);

                if (look_ahead != END_INPUT)
                        error("junk after expression");
                advance();  // past end of input mark
        }
        return 0;
}

// Just die on any error.
void error(char *msg)
{
        fprintf(stderr, "Error: %s. Exiting.\n", msg);
        exit(1);
}

// Buffer the current character and read a new one.
void read()
{
        buf[n++] = ch;
        buf[n] = '\0';  // Terminate the string.
        ch = my_getchar();
}

// Ignore the current character.
void ignore()
{
        ch = my_getchar();
}

// Reset the token buffer.
void reset()
{
        n = 0;
        buf[0] = '\0';
}

// The scanner.  A tiny deterministic finite automaton.
int scan()
{
        reset();
START:
        // first character is digit?
        if (isdigit (ch))
                goto DIGITS;

        switch (ch)
        {
                case ' ': case '\t': case '\r':
                        ignore();
                        goto START;

                case '-': case '+': case '^':
                        read();
                        return ADD_OP;

                case '~':
                        read();
                        return NOT_OP;

                case '*': case '/': case '%':
                        read();
                        return MUL_OP;

                case '(':
                        read();
                        return LEFT_PAREN;

                case ')':
                        read();
                        return RIGHT_PAREN;

                case 0:
                case '\n':
                        ch = ' ';    // delayed ignore()
                        return END_INPUT;

                default:
                        printf ("bad character: 0x%x\n", ch);
                        exit(0);
        }

DIGITS:
        if (isdigit (ch))
        {
                read();
                goto DIGITS;
        }
        else
                return NUMBER;
}

// To advance is just to replace the look-ahead.
void advance()
{
        look_ahead = scan();
}

// Clear the token buffer and read the first look-ahead.
void init()
{
        reset();
        ignore(); // junk current character
        advance();
}

int get_number(char *buf)
{
        char *endptr;

        int rt=strtoul (buf, &endptr, 10);

        // is the whole buffer has been processed?
        if (strlen(buf)!=endptr-buf)
        {
                fprintf (stderr, "invalid number: %s\n", buf);
                exit(0);
        };
        return rt;
};
```

```

int unsigned_factor()
{
        int rtn = 0;
        switch (look_ahead)
        {
                case NUMBER:
                        rtn=get_number(buf);
                        advance();
                        break;

                case LEFT_PAREN:
                        advance();
                        rtn = expr();
                        if (look_ahead != RIGHT_PAREN) error("missing ')'");
                        advance();
                        break;

                default:
                        printf("unexpected token: %d\n", look_ahead);
                        exit(0);
        }
        return rtn;
}

int factor()
{
        int rtn = 0;
        // If there is a leading minus...
        if (look_ahead == ADD_OP && buf[0] == '-')
        {
                advance();
                rtn = -unsigned_factor();
        }
        else
                rtn = unsigned_factor();

        return rtn;
}

int term()
{
        int rtn = factor();
        while (look_ahead == MUL_OP)
        {
                switch(buf[0])
                {
                        case '*':
                                advance();
                                rtn *= factor();
                                break;

                        case '/':
                                advance();
                                rtn /= factor();
                                break;
                        case '%':
                                advance();
                                rtn %= factor();
                                break;
        }
    }
    return rtn;
}

int expr()
{
    int rtn = term();
    while (look_ahead == ADD_OP)
    {
        switch(buf[0])
        {
            case '+':
                advance();
                rtn += term();
                break;

            case '-':
                advance();
                rtn -= term();
                break;
        }
    }
    return rtn;
}
```

KLEE found buffer overflow with little effort (65 zero digits + one tabulation symbol):

```
\$ ktest-tool --write-ints klee-last/test000468.ktest
ktest file : 'klee-last/test000468.ktest'
args       : ['calc.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 128
object    0: data: b'0\t0000000000000000000000000000000000000000000000000000000000000000\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
    \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff\xff\xff'
```

Hard to say, how tabulation symbol (\t) got into input[] array, but KLEE achieved what has been desired: buffer overflown.

KLEE also found two expression strings which leads to division error ("0/0" and "0%0"):

```
\$ ktest-tool --write-ints klee-last/test000326.ktest
ktest file : 'klee-last/test000326.ktest'
args       : ['calc.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 128
object    0: data: b'0/0\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
    \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
    \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff'

\$ ktest-tool --write-ints klee-last/test000557.ktest
ktest file : 'klee-last/test000557.ktest'
args       : ['calc.bc']
num objects: 1
object    0: name: b'input'
object    0: size: 128
object    0: data: b'0%0\x00\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
    \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff
    \xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\xff\
    xff\xff\xff\xff\xff\xff\xff\xff'
```

Maybe this is not impressive result, nevertheless, it's yet another reminder that division and remainder operations must be wrapped somehow in production code to avoid crash.

6.11 Regular expressions

I've always wanted to generate possible strings for given regular expression. This is not so hard if to dive into regular expression matcher theory and details, but can we force RE matcher to do this?

I took lightest RE engine I've found: https://github.com/cesanta/slre, and wrote this:

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5)
                klee_assert(0);
}
```

So I wanted a string consisting of digit, "a" or "b" or "c" (at least one character) and "x" or "y" or "z" (one character). The whole string must have size of 5 characters.

```
\$ klee --libc=uclibc slre.bc
...
KLEE: ERROR: /home/klee/slre.c:445: failed external call: klee_assert
KLEE: NOTE: now ignoring this error at this location
...

\$ ls klee-last | grep err
test000014.external.err

\$ ktest-tool --write-ints klee-last/test000014.ktest
ktest file : 'klee-last/test000014.ktest'
args       : ['slre.bc']
num objects: 1
object    0: name: b's'
object    0: size: 6
object    0: data: b'5aaax\xff'
```

This is indeed correct string. "\xff" is at the place where terminal zero byte should be, but RE engine we use ignores the last zero byte, because it has buffer lenght as a passed parameter. Hence, KLEE doesn't *reconstruct* final byte.

Can we get more? Now we add additional constraint:

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5 &&
                        strcmp(s, "5aaax")!=0)
                klee_assert(0);
}
```

```
\$ ktest-tool --write-ints klee-last/test000014.ktest
ktest file : 'klee-last/test000014.ktest'
args       : ['slre.bc']
```

```
num objects: 1
object    0: name: b's'
object    0: size: 6
object    0: data: b'7aaax\xff'
```

Let's say, out of whim, we don't like "a" at the 2nd position (starting at 0th):

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5 &&
                        strcmp(s, "5aaax")!=0 &&
                        s[2]!='a')
                klee_assert(0);
}
```

KLEE found a way to circumvent our new constraint:

```
\$ ktest-tool --write-ints klee-last/test000014.ktest
ktest file : 'klee-last/test000014.ktest'
args       : ['slre.bc']
num objects: 1
object    0: name: b's'
object    0: size: 6
object    0: data: b'7abax\xff'
```

Let's also define constraint KLEE cannot solve:

```
int main(void)
{
        char s[6];
        klee_make_symbolic(s, sizeof s, "s");
        s[5]=0;
        if (slre_match("^\\d[a-c]+(x|y|z)", s, 5, NULL, 0, 0)==5 &&
                        strcmp(s, "5aaax")!=0 &&
                        s[2]!='a' &&
                        s[2]!='b' &&
                        s[2]!='c')
                klee_assert(0);
}
```

It cannot indeed, and KLEE finished without reporting about `klee_assert()` triggering.

6.12 Exercise

Here is my crackme/keygenme, which may be tricky, but easy to solve using KLEE: http://challenges.re/74/.

7 Acronyms used

CNF Conjunctive normal form 2

DNF Disjunctive normal form 2

DSL Domain-specific language 7

CPRNG Cryptographically Secure Pseudorandom Number Generator 26

SMT	Satisfiability modulo theories	1
SAT	Boolean satisfiability problem	1
LCG	Linear congruential generator	1