Reverse a cellular automata

Google CTF 2019 Quals

Challenge

We have built a cellular automata with 64 bit steps and obeys Wolfram rule 126, it's boundary condition wraps around so that the last bit is a neighbor of the first bit. Below you can find a special step we chose from the automata.

The flag is encrypted with AES256-CBC, the encryption key is the previous step that generates the given step. Your task is to reverse the given step and find the encryption key.

Example decryption with 32 bit steps:

```
echo "404c368b" > /tmp/plain.key; xxd -r -p /tmp/plain.key > /tmp/enc.key
```

```
echo "U2FsdGVkX18+Wl0awCH/gWgLGZC4NiCkrlpesuuX8E70tX8t/TAarSEHTnpY/C1D" | openssl enc -d -aes-256-cbc -pbkdf2 -md sha1 -base64 --pass file:/tmp/enc.key
```

Examples of 32 bit steps, reverse_rule126 in the example yields only one of the multiple values.

rule126('deadbeef') = 73ffe3b8 | reverse_rule126('73ffe3b8') = deadbeef

rule126('73ffe3b8') = de0036ec | reverse_rule126('de0036ec') = 73ffe3b8

rule126('de0036ec') = f3007fbf | reverse_rule126('f3007fbf') = de0036ec

Flag (base64)

U2FsdGVkX1/andRK+WVfKqJILMVdx/69xjAzW4KUqsjr98GqzFR793lfNHrw1Blc8UZHWOBrRhtLx3SM38R1MpRegLTHgHzf0EAa3oUeWcQ=

Obtained step (in hex)

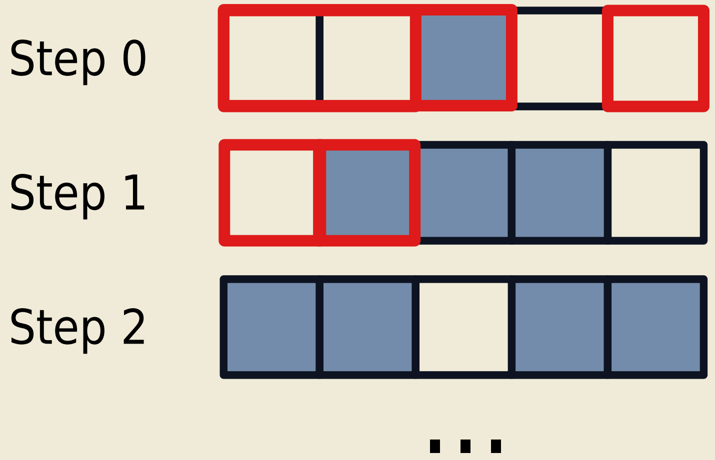
66de3c1bf87fdfcf

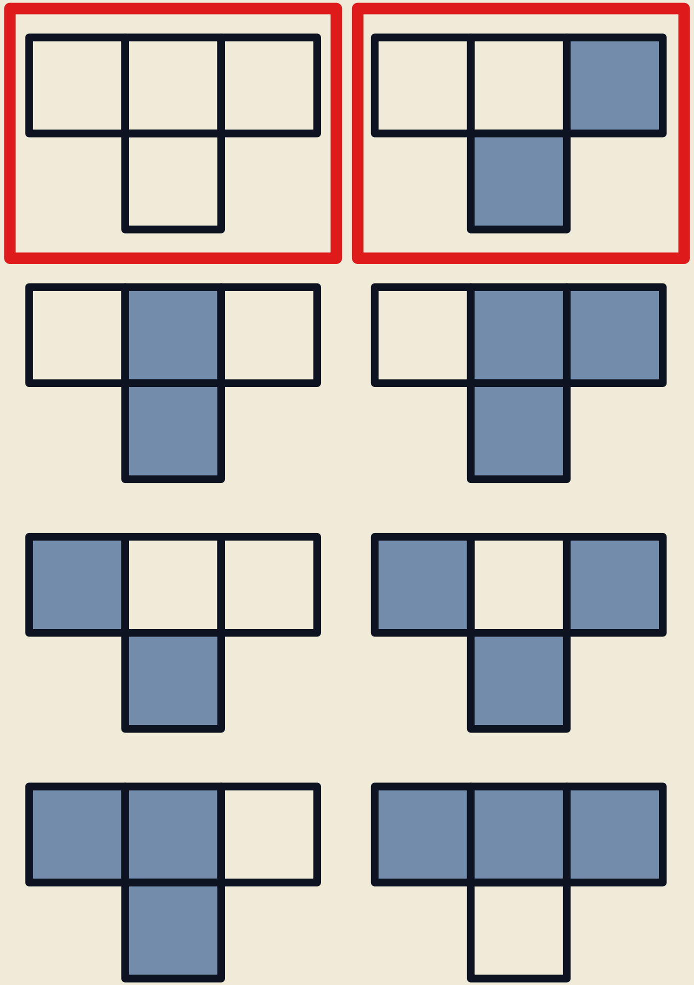
https://cellularautomata.web.ctfcompetition.com/

What we have

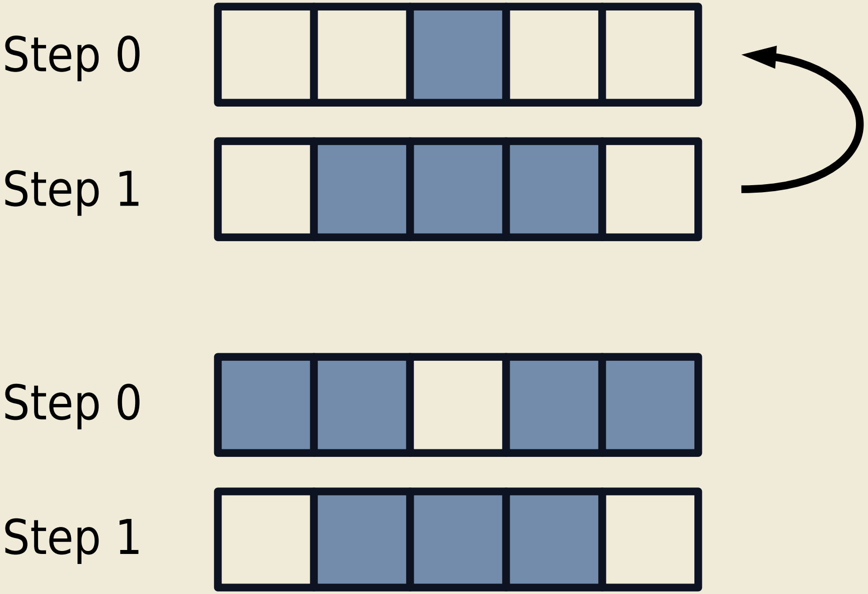
- AES-encrypted flag (base64)
  - Decryption command is provided
- Rule 126 cellular automata step (hexadecimal)
  - Previous step is the encryption key

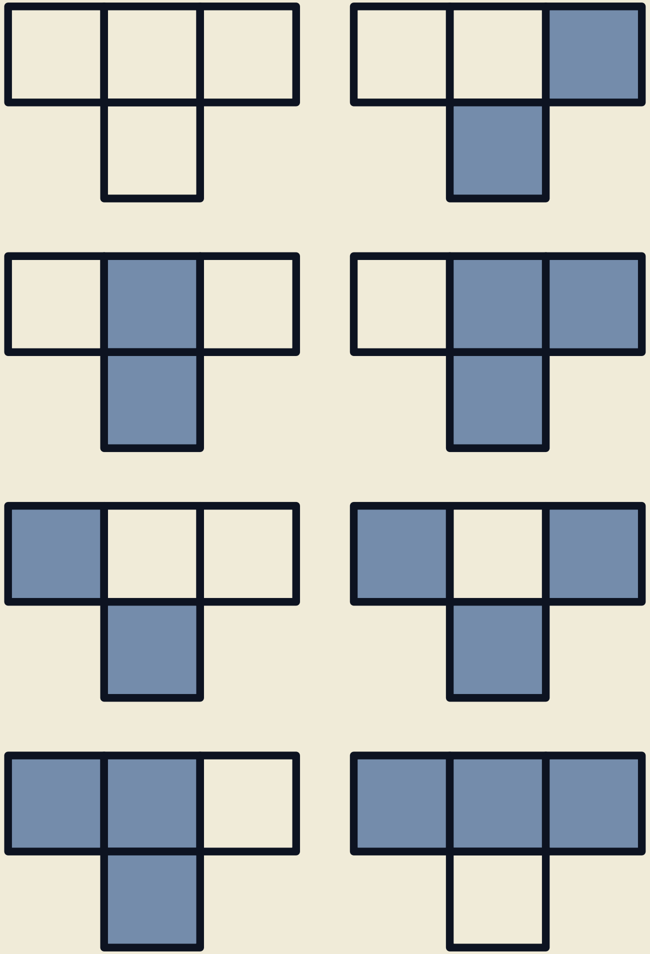
Wolfram rule 126





What’s the catch?





The plan

- Generate all possible previous steps (i.e. keys)
- Try every candidate key
- `grep` the flag
- ????
- PROFIT!!!!

But how…?

- Design some algorithm?
- Convert it to CNF and let a SAT solver do it?
- All very cumbersome…

Z3

What is Z3?

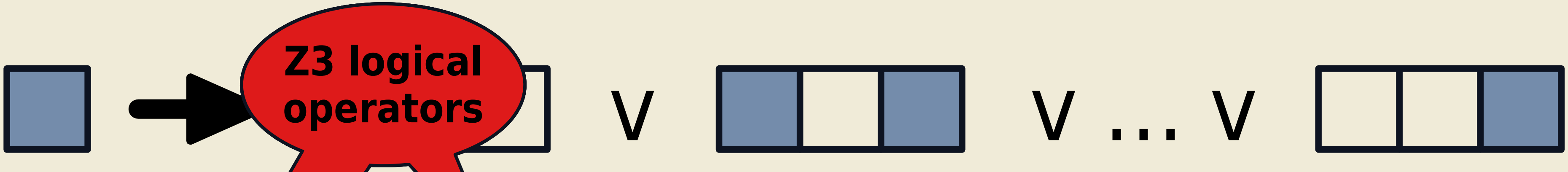
- *"SAT solver that is extremely useful for crypto/rev challenges (and for life in general)"* – lavish
- SMT solver – Satifiablity modulo theories
- Built-in support for
  - Arithmetic
  - Bitvectors
  - ...
- Python bindings!

Modelling the input

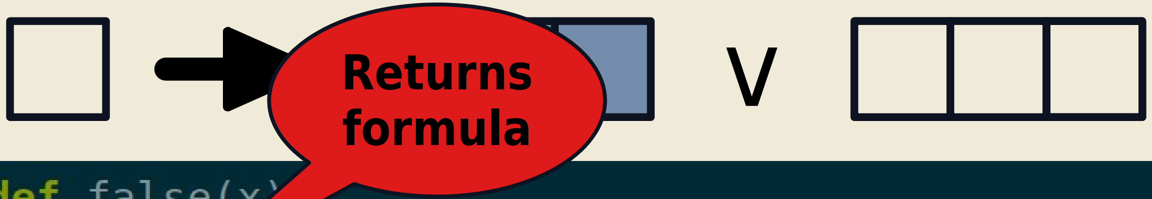
```
#!/usr/bin/    thon
from z3

BINARY = "0    110111100011110000011011111110000111111110111111001111"

length = len(BINARY)
bits = [Bool(i) for i in range(length)]

def neighbors(x):
    return bits[(x - 1) % length], bits[x], bits[(x + 1) % length]
```

Z3 named boolean

Modelling the rules




```
def true(x):
    a, b, c = neighbors(x)
    return Or(And(a, b, Not(c)), And(a, Not(b), c), ..., And(Not(a), Not(b), c))
```




```
def false(x):
    a, b, c = neighbors(x)
    return Or(And(a, b, c), And(Not(a), Not(b), Not(c)))
```

Solving

```
s = Solver()
for i in range(0, length):
    s.add(true(i) if BINARY[i] == "1" else false(i))
```

Add constraint

Negated model

```
while(s.check() == sat):
    m = s.model()
    res = "".join(map(lambda x: "1" if m[x] else "0", bits))
    s.add(Not(And([x == m[x] for x in bits])))
    print("%x" % int(res, 2))
```

Quick example

- Flag input generates ~10.000 keys
- `73ffe3b8 → deadbeef`

Some links

- https://riseforfun.com/Z3
- https://yurichev.com/writings/SAT_SMT_by_example.pdf
- https://ericpony.github.io/z3py-tutorial/guide-examples.htm
- https://z3prover.github.io/api/html/namespacez3py.html