Synthesis Tools for White-box Implementations

Aleksei Udovenko

SnT, University of Luxembourg

WhibOx Workshop
May 19, 2019

UNIVERSITÉ DU LUXEMBOURG

securityandtrust.lu

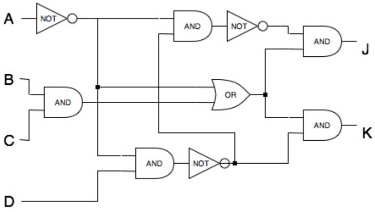
Plan

This talk:

- Python-based framework for practical **white-box** implementations
- Easy to use
- For research purposes
- ... and the WhibOx contest

Circuit Implementations

+ simple framework, both for synthesis and analysis
+ existing literature on hardware design
+ easy to simulate everywhere



– slow (1 bit / register, unless batch execution)
– large code size (storing circuit)
– the power of *Random Access Machine* is not fully utilised (though simulation can be obfuscated on top)

Framework for Circuit WB Synthesis

- easy implementations
  (bitwise are simple, for S-boxes a circuit is needed)
- easy masking (linear + nonlinear)
- starting point for further obfuscation
- included:
  - batch circuit tracing
  - basic DCA-like analysis
    (correlation, exact matches, linear algebra attack)

Framework for Circuit WB Synthesis

- easy implementations
  (bitwise are simple, for S-boxes a circuit is needed)
- easy masking (linear + nonlinear)
- starting point for further obfuscation
- included:
  - batch circuit tracing
  - basic DCA-like analysis
    (correlation, exact matches, linear algebra attack)
- convenient C code generation for the WhibOx contest contest

8	47	■	peaceful_williams	11.77 🍓 / 5.88 🥕	You!	Standing

A Quick Teaser

```
NR = 10
KEY = "MySecretKey!2019"

pt = Bit.inputs("pt", 128)
ct, k10 = BitAES(pt, Bit.consts(str2bin(KEY)), nr=NR)

prng = LFSR(taps=[0, 2, 5, 18, 39, 100, 127],
            state=BitAES(pt, pt, nr=2)[0])
rand = Pool(n=128, prng=prng).step

ct = mask_circuit(ct, MINQ(rand=rand))
ct = mask_circuit(ct, DOM(rand=rand, nshares=2))

whibox_generate(ct, "build/submit.c", "Hello, world!")
```

AES circuit with *configurable* masking
(quadratic MINQ + linear DOM-indep)

A Quick Teaser

```
NR = 10
KEY = "MySecretKey!2019"

pt = Bit.inputs("pt", 128)
ct, k10 = BitAES(pt, Bit.consts(str2bin(KEY)), nr=NR)

prng = LFSR(taps=[0, 2, 5, 18, 39, 100, 127],
            state=BitAES(pt, pt, nr=2)[0])
rand = Pool(n=128, prng=prng).step

ct = mask_circuit(ct, MINQ(rand=rand))
ct = mask_circuit(ct, DOM(rand=rand, nshares=2))

whibox_generate(ct, "build/submit.c", "Hello, world!")
```

AES circuit with *configurable* masking (quadratic MINQ + linear DOM-indep)

Whib0x CTF - ready :)

A Quick Teaser

```
NR = 10
KEY = "MySecretKey!2019"

pt = Bit.inputs("pt", 128)
ct, k10 = BitAES(pt, Bit.consts(str2bin(KEY)), nr=NR)

prng = LFSR(taps=[0, 2, 5, 18, 39, 100, 127],
            state=BitAES(pt, pt, nr=2)[0])
rand = Pool(n=128, prng=prng).step

ct = mask_circuit(ct, MINQ(rand=rand))
ct = mask_circuit(ct, DOM(rand=rand, nshares=2))

whibox_generate(ct, "build/submit.c", "Hello, world!")
```

AES circuit with *configurable* masking
(quadratic MINQ + linear DOM-indep)

Whib0x CTF - ready :)
(ouch, no fault protection...)

Plan

Circuit Construction

- Bit: a circuit node, operations are overloaded:

```
x = Bit.input("x")
y = Bit.input("y")
print ~(x & y) ^ y
Output: (~(x & y) ^ y)
```

Circuit Construction

- Bit: a circuit node, operations are overloaded:

```
x = Bit.input("x")
y = Bit.input("y")
print ~(x & y) ^ y
Output: (~(x & y) ^ y)
```

- Vector: a list that propagates operations to its elements.
- (Keyless) Simon:

```
pt = Vector(Bit.inputs("pt", 32))
l, r = pt.split()
for round in xrange(32):
    r ^= (l.rol(1) & l.rol(8)) ^ l.rol(2)
    l, r = r, l
ct = l.concat(r)
```

AES Circuit (1/2)

- AES-128 circuit included ($\approx$ 31000 gates); based on Canright's S-Box.

```
key = Bit.consts(str2bin("MySecreyKey!2019"))
pt = Bit.inputs("pt", 128)
ct, k10 = BitAES(pt, key, nr=10)
# k10 is the last subkey
```

AES Circuit (2/2)

- Clean and modular internal structure, easy to modify.
- Rect: representation of rectangular (AES-like) states.

```
def BitAES(plaintext, key, rounds=10):
    bx = Vector(plaintext).split(16)
    bk = Vector(key).split(16)
    state = Rect(bx, w=4, h=4).transpose()
    kstate = Rect(bk, w=4, h=4).transpose()
    for rno in xrange(rounds):
        state = AK(state, kstate)
        state = SB(state)
        state = SR(state)
        if rno < rounds-1:
            state = MC(state)
        kstate = KS(kstate, rno)
    state = AK(state, kstate)
    bits = sum( map(list, state.transpose().flatten()), [])
    kbits = sum( map(list, kstate.transpose().flatten()), [])
    return bits, kbits
```

Masking (1/3)

```
class DOM(MaskingScheme):
    def encode(self, s):
        x = [self.rand() for _ in xrange(self.nshares-1)]
        x.append(reduce(xor, x) ^ s)
        return tuple(x)
    def decode(self, x):
        return reduce(xor, x)
    def XOR(self, x, y):
        return tuple(xx ^ yy for xx, yy in zip(x, y))
    def AND(self, x, y):
        matrix = [[xx & yy for yy in y] for xx in x]
        for i in xrange(1, self.nshares):
            for j in xrange(i + 1, self.nshares):
                r = self.rand()
                matrix[i][j] ^= r
                matrix[j][i] ^= r
        return tuple(reduce(xor, row) for row in matrix)
    def NOT(self, x):
        return (~x[0],) + tuple(x[1:])
```

Masking (2/3)

Linear Masking:

$$s = x_0 \oplus x_1 \oplus \ldots \oplus x_{r-1}$$

Minimalist Quadratic Masking:

$$s = x_0 x_1 \oplus x_2$$

Masking (3/3)

```python
def mask_circuit(circuit, scheme, encode=True, decode=True):
    """ Mask a given @circuit with a given masking @scheme.
    Arguments @encode and @decode specify
    whether encoding and decoding steps should be added. """
    ...
pt = Bit.inputs("pt", 128)
ct, _ = BitAES(pt, ..., rounds=NR)

# define a PRNG initialized with plaintext, also a circuit!
# here we use 2-round AES for initialization
# and LFSR for further generation
prng = LFSR(taps=[0, 2, 5, 18, 39, 100, 127],
            state=BitAES(pt, pt, rounds=2)[0])
rand = Pool(n=128, prng=prng).step

# nested masking
ct = mask_circuit(ct, MINQ(rand=rand))
ct = mask_circuit(ct, DOM(rand=rand, nshares=2))
```

Plan

```
typedef uint16_t A;
switch (op) {
case XOR:
    a = *((A *)p); pop();
    b = *((A *)p); pop();
    ram[dst] = ram[a] ^ ram[b];
    break;
case AND:
    a = *((A *)p); pop();
    b = *((A *)p); pop();
    ram[dst] = ram[a] & ram[b];
    break;
case OR:
    a = *((A *)p); pop();
    b = *((A *)p); pop();
    ram[dst] = ram[a] | ram[b];
    break;
case NOT:
    a = *((A *)p); pop();
    ram[dst] = ~ram[a];
    break;
case RANDOM:
    ram[dst] = rand();
    break;
default: return;
}
```

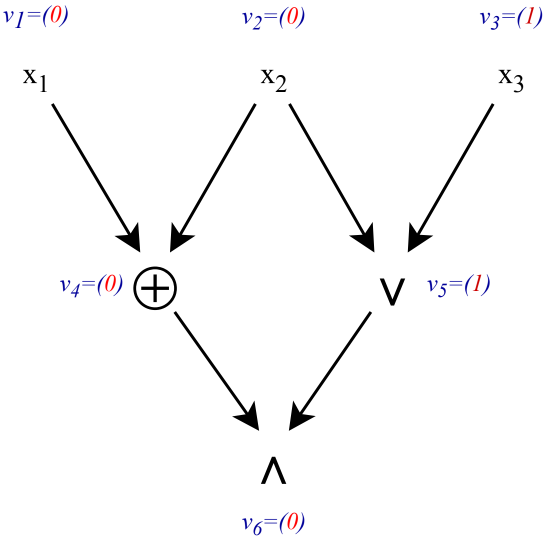
- C code for simulation
- requires some encoding of the circuit
- easier to encode more compact than by a compiler
- *usecase 1:* local tracing/analysis
- *usecase 2:* PoC generation

Compile and Run

```
KEY = "MySecretKey!2019"

pt = Bit.inputs("pt", 128)
ct, k10 = BitAES(pt, Bit.consts(str2bin(KEY)), rounds=10)

# Encode circuit to file
RawSerializer().serialize_to_file(ct, "circuits/aes10.bin")

# Python API for C simulator
C = FastCircuit("circuits/aes10.bin")

ciphertext = C.compute_one("my_plaintext_abc")

# Verify correctness
from Crypto.Cipher import AES
ciphertext2 = AES.new(KEY).encrypt(plaintext)
assert ciphertext == ciphertext2
```

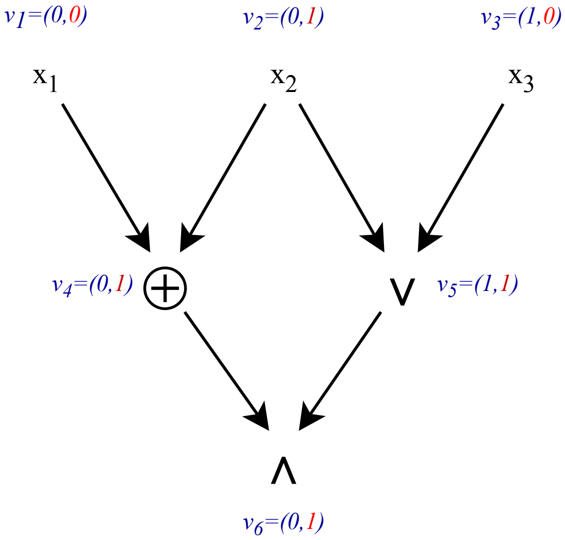
Plan

General DCA Framework



$x^{(1)} = (0,0,1)$

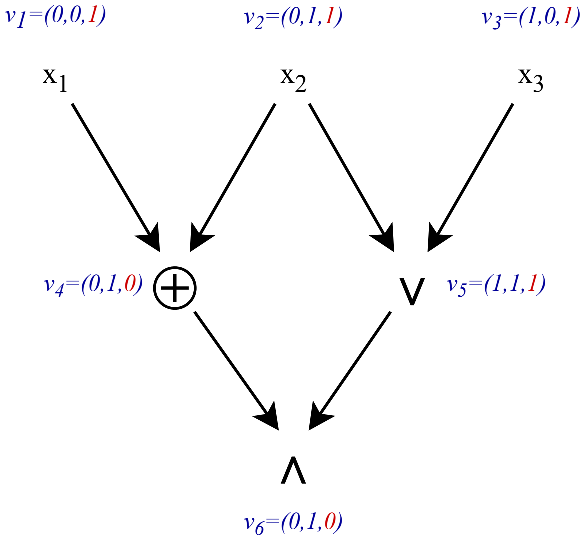
General DCA Framework



$x^{(1)} = (0,0,1)$

$x^{(2)} = (0,1,0)$

General DCA Framework



$x^{(1)} = (0,0,1)$

$x^{(2)} = (0,1,0)$

$x^{(3)} = (1,1,1)$

DCA Attacks

1. Correlation-based attacks
   (up to 2nd order)
   using github.com/SideChannelMarvels/Daredevil

2. Exact Match / Time-Memory Trade-off
   (up to 2nd order, more efficient but fragile)
   using github.com/cryptolu/whitebox

3. Linear Algebraic Attack
   using github.com/cryptolu/whitebox

Batch Tracing

- Compute 64 traces in parallel, by using full registers in bit-slice fashion.
- C-code with Python interface: very efficient.

```
from whitebox.fastcircuit import FastCircuit, chunks

plaintexts = os.urandom(64 * 16)
plaintexts = chunks(plaintexts, 16)
FC = FastCircuit("./circuits/aes10.bin")
FC.compute_batch(
    inputs=plaintexts,
    trace_filename="./traces/aes10.bin"
)

trace_split_batch(
    filename="./traces/aes10.bin",
    ntraces=64,
    packed=True
)
```

D E M O

Configurations and Attacks Summary

Masking		Attacks			
MINQ	DOM-r shares	Exact-1	Exact-2	Daredevil-1	Lin.Alg.
-	-	🔥	🔥	🔥	🔥
-	2 shares	-	🔥	-	🔥
-	3+ shares	-	-	-	🔥
+	-	-	-	🔥	-
+	2 shares	-	-	-	-

Plan

Conclusions

»» github.com/cryptolu/whitebox/synthesis/ ««

- Towards easier *synthesis* and *analysis* of white-box implementations
- For research & proof-of-concept implementations
- /!\ early version, may contain bugs
- Contributions are welcome!