Symbolic Computation assists Algebraic Cryptanalysis: SCrypt

Burçin Eröcal

Research Institute for Symbolic Computation
Johannes Kepler University, Linz

12 October 2008
Algebraic cryptanalysis

- represent cryptographic problems as systems of polynomial equations
- solve these systems

Cryptosystems $f_k: \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n$

- variables: key, plaintext, ciphertext, internal state
- solutions $X_L, X_{SL}$
- $f_k(p) = c$
- $F_4, F_5$

Questions

- "suitable" representation
- solution methods
Algebraic cryptanalysis

- represent cryptographic problems as systems of polynomial equations
- solve these systems

Cryptosystems → systems of equations → solutions

\[ f_k : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n \]
\[ f_k(p) = c \]
variables: key, plaintext, ciphertext, internal state

XL, XSL
F_4, F_5

Questions

- "suitable" representation
- solution methods
Algebraic cryptanalysis

- represent cryptographic problems as systems of polynomial equations
- solve these systems

\[ f_k : \mathbb{F}_2^n \rightarrow \mathbb{F}_2^n \]
\[ f_k(p) = c \]

Cryptosystems \(\rightarrow\) systems of equations \(\rightarrow\) solutions

variables: key, plaintext, ciphertext, internal state

XL, XSL \(\rightarrow\) F₄, F₅

Questions

- "suitable" representation
- solution methods
Problem

Only a few examples available.

- Write down equations for specific components by hand, use a computer algebra system to put them together.
  - Tedious.
  - Resulting systems are big, hard to manipulate and work with.
- This process can be automated!

Goal

Given the specification of a cryptosystem, allow plugging in different representations of components to generate systems of polynomial equations.
Algebraic cryptanalysis & examples

Problem

Only a few examples available.

- Write down equations for specific components by hand, use a computer algebra system to put them together.
  - Tedious.
  - Resulting systems are big, hard to manipulate and work with.
- This process can be automated!

Goal

Given the specification of a cryptosystem, allow plugging in different representations of components to generate systems of polynomial equations.
Problem

Only a few examples available.

- Write down equations for specific components by hand, use a computer algebra system to put them together.
  - Tedious.
  - Resulting systems are big, hard to manipulate and work with.
- This process can be automated!

Goal

Given the specification of a cryptosystem, allow plugging in different representations of components to generate systems of polynomial equations.
Problem

Only a few examples available.

- Write down equations for specific components by hand, use a computer algebra system to put them together.
  - Tedium.
  - Resulting systems are big, hard to manipulate and work with.
- This process can be automated!

Goal

Given the specification of a cryptosystem, allow plugging in different representations of components to generate systems of polynomial equations.
SCrypt: Symbolic computation & cryptography

- Easy structural description of cryptosystems.
- Compute intermediate states as symbolic expressions.
- Provide different ways of converting symbolic expressions to algebraic models.

```python
self.SBox0 = function('SBox0', 4, 2)
self.SBox1 = function('SBox1', 4, 2)

Emat = matrix(ZZ, 8, 4, [0, 0, 0, 1,
                          1, 0, 0, 0,
                          0, 1, 0, 0,
                          0, 0, 1, 0,
                          0, 1, 0, 0,
                          0, 0, 1, 0,
                          0, 0, 0, 1,
                          1, 0, 0, 0])

ExpPermuation = MatrixOp('ExpPerm', Emat)

lpmat = matrix(ZZ, 4, 4, [0, 1, 0, 0,
                           0, 0, 0, 1,
                           0, 0, 1, 0,
                           1, 0, 0, 0])

LastPerm = MatrixOp('LastPerm', lpmat)

F = OpChain(4)
F.chain_op(ExpPermuation)
F.chain_op(KeyAddition)
F.chain_op(ParallelOp('S', [self.SBox0, self.SBox1]))
F.chain_op(LastPerm)

FeistelCipher.__init__(self, F, 8, 10, 1, 0)
```
SCrypt: Symbolic computation & cryptography

- Easy structural description of cryptosystems.
- Compute intermediate states as symbolic expressions.
- Provide different ways of converting symbolic expressions to algebraic models.

```sage
sd.states[2]
regs: [p4 + rr_0_5, p5 + rr_0_7, p6 + rr_0_6, p7 + rr_0_4, p0 + rr_0_1, p1 + rr_0_3, p2 + rr_0_2, p3 + rr_0_0]
varpref: rr_0
pref_inds: {'p': 8, 't': 0, 'rr_0_': 8, 'k': 10}
rels: [(SBox0(k0 + p7, k6 + p4, k8 + p5, k3 + p6), [rr_0_0, rr_0_1]), (SBox1(k7 + p5, k2 + p6, k9 + p7, k5 + p4), [rr_0_2, rr_0_3]), (SBox0(p3 + rr_0_0, p0 + rr_0_1, p1 + rr_0_3, p2 + rr_0_2), [rr_0_4, rr_0_5]), (SBox1(p1 + rr_0_3, p2 + rr_0_2, p3 + rr_0_0, p0 + rr_0_1), [rr_0_6, rr_0_7])]
```
Easy structural description of cryptosystems.
Compute intermediate states as symbolic expressions.
Provide different ways of converting symbolic expressions to algebraic models.

```python
sage: from scrypt.chain import State
sage: from scrypt.eqgen import EqGen
sage: st = State([])
sage: t0 = st.new_var()
sage: eq = EqGen(st, 4, base_field = GF(2))
sage: eq.sringel_to_polys(t0)
[t0_0, t0_1, t0_2, t0_3]
sage: from scrypt.op import Not
sage: eq.sringel_to_polys(Not(t0))
[t0_0 + 1, t0_1 + 1, t0_2 + 1, t0_3 + 1]
```

```python
sage: eq = EqGen(st, 4, base_field = GF(4, 'a'))
sage: eq.sringel_to_polys(t0)
[t0_0, t0_1]
sage: eq.sringel_to_polys(Not(t0))
[t0_0 + (a + 1), t0_1 + (a + 1)]
```
SCrypt provides

- Constructs for common cipher components
  (linear diffusion, rotation, modulo sums, field multiplication, etc.)
- Cryptosystem design patterns
  (block ciphers, Feistel networks, stream ciphers, etc.)
- Framework to connect components together similar to circuit diagram patterns
  commonly used in design specifications

Example

```
ApplyF = BinaryOp('ApplyF', 2, 0, 1, 0, func2=F)
HalfRound = OpChain(2, adjust_blocks=True)
HalfRound.chain_op(ApplyF)
HalfRound.chain_op(Swap)
```
SHA1 compression function

#(b and c) or ((not b) and d)
F1 = function ('F1', 3, 1)
#b xor c xor d
F2 = function ('F2', 3, 1)
#(b and c) or (b and d) or (c and d)
F3 = function ('F3', 3, 1)

RotL5 = RotL(5)
F = ParamFunc('F')

@fn2Op(cache_result=False)
def AFunc(state, args, kwds):
    return ModSum(F(*(state[1:4]), **kwds),
                  state[4], RotL5(state[0]),
                  args[0], args[1])

RotR2 = RotR(2)
CFunc = OpChain(5)
CFunc.chain_op(Proj('P2', 2))
CFunc.chain_op(RotR2)

SHA1C = ParallelOp('SHA1C',
       [AFunc, Proj('P1', 1), CFunc, Proj('P3', 3), Proj('P4', 4)],
       input_all=True, args_all=True, cache_result=False)
SHA1 compression function

sage: from scrypt.symb import var
sage: from scrypt.chain import State
sage: s = [var('x' + str(i))
       for i in range(5)]

sage: w0, w1 = var('w0'), var('w1')

sage: res = SHA1C(w0, w1,
               state=State(s), F=F1)

sage: res
regs: [ModSum(x4, w0, w1, RotL5(x0),
             F1(x1, x2, x3)), x1, RotR2(x2), x3, x4]

varpref: t
pref_inds: {'t': 0}
rels: []

sage: w2, w3 = var('w2'), var('w3')

sage: SHA1C(w2, w3, state=res, F=F2)
regs: [ModSum(x4, w2, w3, RotL5(ModSum(x4, w0, w1,
                    RotL5(x0), F1(x1, x2, x3))), F2(x1, RotR2(x2)
                    , x3)), x1, RotR2(RotR2(x2)), x3, x4]

varpref: t
pref_inds: {'t': 0}
rels: []
Symbolic expressions

- Symbolic variables in a ring of characteristic 2
  \[( + \rightarrow \text{xor}, \times \rightarrow \text{and})\]
- Symbolic expressions used to denote other constructs
- Implementation based on PolyBoRi in Sage

**Example**

```python
sage: from scrypt.symb import var, function
sage: x, y = var('x'), var('y')
```

```python
sage: sage: from scrypt.op import FMul, ModSum, Not
sage: FMul16 = FMul(GF(16, 'a'))
```

```python
sage: x + ModSum(FMul16(x, y), Not(x))
```

```python
sage: SBox = function('SBox', 2, 2)
```

```python
sage: SBox(x, y)
SBox(x, y)
```
Symbolic relations

Multiple outputs

If a function has multiple outputs (i.e., an SBox),

- new symbolic variables are created to represent the outputs
- a relation is recorded in the relevant data structure.

Example

```sage
import ctc
sage: c = ctc.CTC(1, 1)
sage: c.calc_states()
sage: c.states[1]
regs: [k0 + p0, k1 + p1, k2 + p2]
...
sage: c.states[2]
regs: [rr_0_0 + rr_0_1, rr_0_1 + rr_0_2, rr_0_0]
...
rels: [(SBox(k0 + p0, k1 + p1, k2 + p2),
       [rr_0_0, rr_0_1, rr_0_2])]
```
Symbolic expressions to equations

- Specify how many bits in a block and base field,
- SCrypt processes the symbolic expressions to create systems of polynomial equations

Example

```python
sage: from scrypt.chain import State
sage: from scrypt.eqgen import EqGen
sage: st = State([])
sage: t0 = st.new_var()
sage: eq = EqGen(st, 4, base_field = GF(2))

sage: eq.sringel_to_polys(t0)
[t0_0, t0_1, t0_2, t0_3]

sage: eq.sringel_to_polys(Not(t0))
[t0_0 + 1, t0_1 + 1, t0_2 + 1, t0_3 + 1]

sage: eq = EqGen(st, 4, base_field = GF(4, 'a'))

sage: eq.sringel_to_polys(t0)
[t0_0, t0_1]

sage: eq.sringel_to_polys(Not(t0))
[t0_0 + (a + 1), t0_1 + (a + 1)]
```
Example

```python
sage: F = GF(16, 'a')
sage: a = F.gen()
sage: from scrypt.symb import constant
sage: from scrypt.op import FMul
sage: FMul16 = FMul(F)
sage: FElem3 = constant('FElem3', a+1)
sage: t1 = st.new_var()
sage: eq = EqGen(st, 4, base_field = GF(2))
sage: eq.string_to_polys(FMul16(t0, FElem3))
[t0_0 + t0_3, t0_0 + t0_1 + t0_3, t0_1 + t0_2, t0_2 + t0_3]
sage: eq.string_to_polys(t0 + FMul16(t0, Not(t1)))
[t0_0*t1_0 + t0_3*t1_1 + t0_2*t1_2 + t0_1*t1_3 + t0_1 + t0_2 + t0_3, t0_1*t1_0 + t0_0*t1_1 + t0_3*t1_1 + t0_2*t1_2 + t0_3*t1_2 + t0_1*t1_3 + t0_2*t1_3 + t0_0 + t0_1, t0_2*t1_0 + t0_1*t1_1 + t0_0*t1_2 + t0_3*t1_2 + t0_2*t1_3 + t0_3*t1_3 + t0_0 + t0_1 + t0_2, t0_3*t1_0 + t0_2*t1_1 + t0_1*t1_2 + t0_0*t1_3 + t0_3*t1_3 + t0_0 + t0_1 + t0_2 + t0_3]
```