Sage & Algebraic Techniques for the Lazy Symmetric Cryptographer

Martin R. Albrecht
@martinralbrecht

Crypto Group, DTU Compute, Denmark

IceBreak, Reykjavik, Iceland
#icebreak

Outline

Outline

Blurb

Sage open-source mathematical software system

"Creating a viable free open source alternative to Magma, Maple, Mathematica and Matlab."

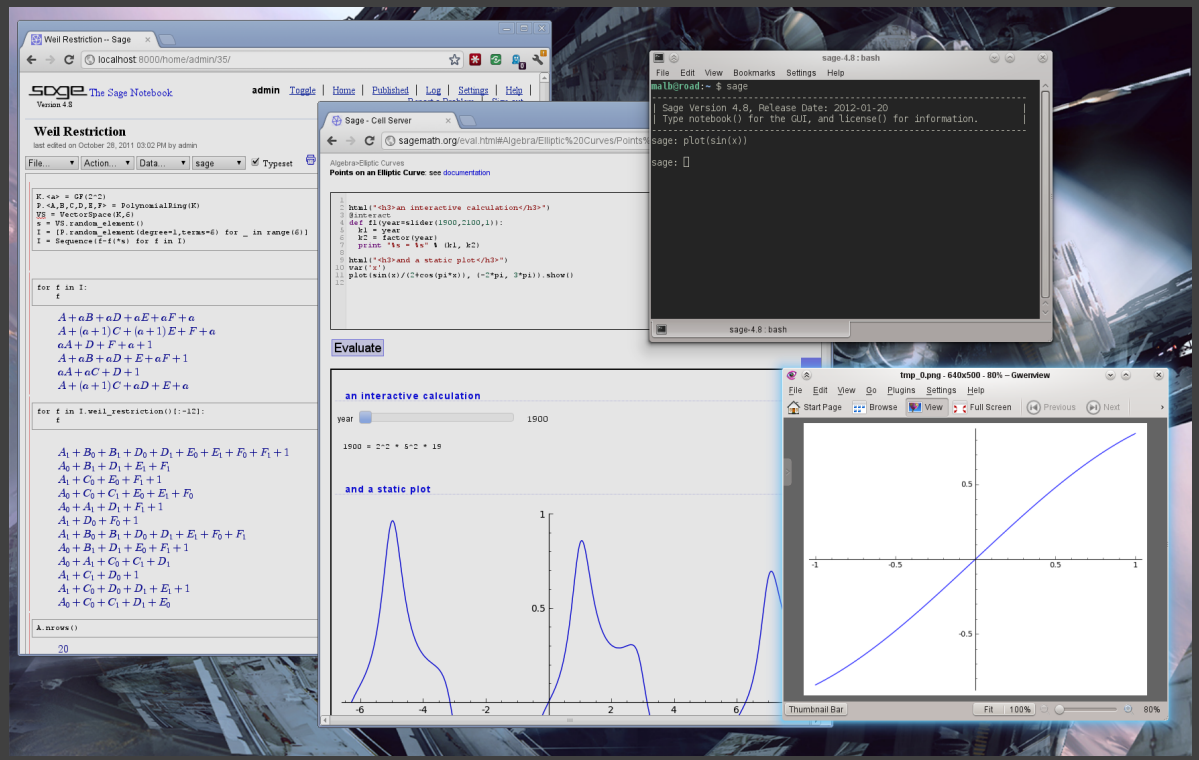
Sage is a free open-source mathematics software system licensed under the GPL. It combines the power of many existing open-source packages into a common Python-based interface.

- First release 2005
- > 300 releases
- > 180 developers
- > 200 papers cite Sage
- > 100,000 web visitors/month

- Latest version 5.9 released 2013-05-03
- Shell, webbrowser (GUI), library
- ~ 100 components
- > 2100 subscribers [sage-support]
- > 6, 500 downloads/month

How to use it

Sage can be used via the command line, as a webapp hosted on your local computer and via the Internet, or embedded on any website.



Check out: `http://aleph.sagemath.org` and `https://cloud.sagemath.com/`

“How do I do …in Sage?”

…It’s easy: implement it and send us a patch.

Sage is a largely volunteer-driven effort, this means that

- developers work on whatever suits their needs best;
- the quality of code in Sage varies:
  - is a generic or a specialised, optimised implementation used,
  - how much attention is paid to details,
  - is your application an untested “corner case”,
  - how extensive are the tests, the documentation, or
  - is the version of a particular package up to date.
- you cannot expect people to fix your favourite bug quickly (although we do try!),
- you can get involved and make Sage better for your needs!

send us a patch

I will highlight relevant issues to encourage you to get involved.

Outline

Python & Cython

Sage does **not** come with yet-another ad-hoc mathematical programming language, it uses **Python** instead.

- one of the most widely used programming languages (Google, IML, YouTube, NASA),
- easy for you to define your own data types and methods on it (bitstreams, ciphers, rings, whatever),
- very clean language that results in easy to read code,
- a **huge number of libraries**: statistics, networking, databases, bioinformatic, physics, video games, 3d graphics, numerical computation (scipy), and serious "pure" mathematics (via Sage)
- easy to use existing C/C++ libraries from Python (via **Cython**)

Python Example: Networking

Scapy is a powerful interactive packet manipulation program written in Python. It is able to forge or decode packets of a wide number of protocols, send them on the wire, capture them, match requests and replies, and much more. It can easily handle most classical tasks like scanning, tracerouting, probing, unit tests, attacks or network discovery.

```
from scapy.all import *

class Test(Packet):
    name = "Test packet"
    fields_desc = [ ShortField("test1", 1),
                    ShortField("test2", 2) ]

print Ether()/IP()/Test(test1=x,test2=y)

p=sr1(IP(dst="127.0.0.1")/ICMP())
if p:
    p.show()
```

Cython: Your Own Code

```
sage: cython("""
def foo(unsigned long a, unsigned long b):
  cdef int i
  for i in range(64):
    a ^= a*(b<<i)
  return a
""")
sage: foo(a,b)
```

This generates C code like this:

```
for (__pyx_t_1 = 0; __pyx_t_1 < 64; __pyx_t_1+=1) {
    __pyx_v_i = __pyx_t_1;
    __pyx_v_a = (__pyx_v_a ^ _pyx_v_a * (__pyx_v_b << __pyx_v_i));
}
```

Cython: External Code I

```
#cargs -std=c99 -ggdb
cdef extern from "katan.c":
    ctypedef unsigned long uint64_t
    void katan32_encrypt(uint64_t *p, uint64_t *c, uint64_t *k, int nr)
    void katan32_keyschedule(uint64_t *k, uint64_t *key, int br)
    uint64_t ONES

def k32_encrypt(plain, key):
    cdef int i
    cdef uint64_t _plain[32], _cipher[32], kk[2*254], _key[80]

    for i in range(80):
        _key[i] = ONES if key[i] else 0
    for i in range(32):
        _plain[i] = ONES if plain[i] else 0

    katan32_keyschedule(kk, _key, 254)
    katan32_encrypt(_plain, _cipher, _key, 254)

    return [int(_cipher[i]%2) for i in range(32)]

sage: load("sage-katan.spyx")
sage: k32_encrypt(random_vector(GF(2),32),random_vector(GF(2),80))
[1, 0, 0, 1, 0, 1, 0, 0, 0, 1, ... 0, 1, 0, 0]
```

Cython: External Code II

```
sage: rv = lambda : random_vector(GF(2),32)
sage: E = lambda : k32_encrypt(rv(),rv())

sage: l = [E() for _ in range(1024)]
sage: l = [sum(e) for e in l]
sage: r.summary(l) # We are using R!
Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
8.00   14.00   16.00   16.03   18.00   27.00

sage: c = E()
sage: K = GF(next_prime(2^32))
sage: g = K(sum(2^i*c[i] for i in range(32))); g
2859908881
sage: g.multiplicative_order() # We are using Pari/GP
858993462

sage: A = matrix(GF(2),32,32,[E() for _ in range(32)])
sage: A.rank() # We are using M4RI
30
```

Symmetric Multiprocessing

Embarrassingly parallel computations on multicore machines are easy in Sage:

```
sage: @parallel(2)
....: def f(n):
....:     return factor(n)
....:

sage: %time _ = [f(2^217-1), f(2^217-1)]
CPU times: user 1.07 s, sys: 0.02 s, total: 1.09 s
Wall time: 1.10 s

sage: %time _ = list( f([2^217-1, 2^217-1]) )
CPU times: user 0.00 s, sys: 0.02 s, total: 0.02 s
Wall time: 0.62 s

sage: 1.08/0.62
1.74193548387097
```

Outline

Dense Linear Algebra

```
sage: for p in (2,3,4,5,7,8,9,11):
....:     K = GF(p,'a')
....:     A = random_matrix(K,2000,2000)
....:     B = random_matrix(K,2000,2000)
....:     t = cputime()
....:     C = A*B
....:     print "%32s %7.3f"%(K,cputime(t))
....:
Finite Field of size 2           0.008 # M4RI
Finite Field of size 3           0.972 # LinBox
Finite Field in a of size 2^2    0.048 # M4RIE
Finite Field of size 5           0.996 # LinBox
Finite Field of size 7           0.968 # LinBox
Finite Field in a of size 2^3    0.072 # M4RIE
Finite Field in a of size 3^2  695.863 # generic
Finite Field of size 11          1.020 # LinBox
```

send us a patch

We know how to make $\mathbb{F}_{p^k}$ really fast, but someone needs to step up.

FLINT 2.3 (in Sage 5.10) improves $\mathbb{F}_p$ for $2^{23} < p < 2^{64}$, but an interface is missing.

Sparse Linear Algebra

to construct and compute with sparse matrices by using the `sparse=True` keyword.

```
sage: A = random_matrix(GF(32003),2000,2000,density=~200,sparse=True)
sage: %time copy(A).rank() # LinBox
CPU times: user 3.26 s, sys: 0.05 s, total: 3.31 s
Wall time: 3.33 s
2000
sage: %time copy(A).echelonize() # custom code
CPU times: user 9.51 s, sys: 0.02 s, total: 9.52 s
Wall time: 9.56 s
sage: v = random_vector(GF(32003),2000)
sage: %time _ = copy(A).solve_right(v) # LinBox + custom code
CPU times: user 3.74 s, sys: 0.00 s, total: 3.74 s
Wall time: 3.76 s
```

send us a patch

LinBox's claim to fame is good support for **black box** algorithms for sparse and structured matrices. Help us to expose more of this functionality.

Lattices

Sage includes both NTL and fpLLL:

```
sage: from sage.libs.fplll.fplll import gen_intrel  # Knapsack-style
sage: A = gen_intrel(50,50); A
50 x 51 dense matrix over Integer Ring ...
sage: min(v.norm().n() for v in A.rows())
2.178593181109 50e13

sage: L = A.LLL() # using fpLLL, NTL optional
sage: L[0].norm().n()
5.47722557505166

sage: L = A.BKZ() # using NTL
sage: L[0].norm().n()
3.60555127546399
```

send us a patch

Our version of fpLLL is old (to be updated in 5.11, but an interface to its BKZ is missing).

Symbolics

Sage uses Pynac (GiNaC fork) and Maxima for most of its symbolic manipulation. SymPy is included in Sage as well.

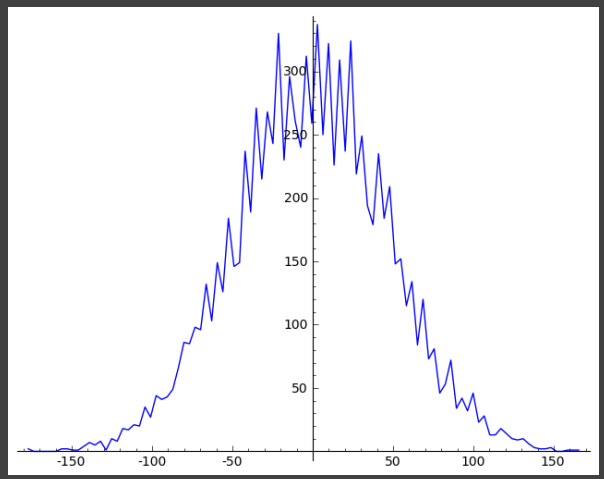
```
sage: q = var('q')
sage: expr = (1-1/q)/(q-1)
sage: f = expr.function(q); f
q |--> -(1/q - 1)/(q - 1)
sage: f(10)
1/10
sage: f(q^2)
-(1/q^2 - 1)/(q^2 - 1)
sage: f(0.1)
10.0000000000000
sage: g = P.random_element(); g
4*x^2 + 3/4*x
sage: f(g)
-4*(4/((16*x + 3)*x) - 1)/((16*x + 3)*x - 4)

sage: expr.simplify_full()
1/q
sage: expr.integrate(q)
log(q)
```

Statistics

Sage ships R which is a very powerful package for doing statistics, Sage also uses SciPy for stats related tasks.

```
sage: O() # some oracle
sage: l = [O() for _ in range(10000)] # we sample it
sage: r.summary(l) # and ask R about it
   Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
-154.000  -31.000     2.000     0.298    33.000   140.000
sage: import pylab # use pylab to compute a histogram
sage: a,b,_ = pylab.hist(l,100)
sage: line(zip(b,a)) # and Sage's code to plot it
```



send us a patch

Our interface to R could be greatly improved

Coding Theory

Computations in coding theory are mainly realised by GAP. For example, we can ask about the minimum distance of a code.

```
sage: A = random_matrix(GF(2), 8, 8)
sage: while A.rank() != 8: A = random_matrix(GF(2), 8, 8)
sage: A
[0 0 0 0 1 0 1 0]
[1 1 0 1 1 1 1 0]
[0 0 1 1 0 1 1 0]
[0 0 0 1 0 1 1 0]
[0 0 0 0 0 0 1 0]
[0 1 0 1 0 0 1 0]
[1 0 1 0 1 0 0 1]
[1 0 0 1 1 1 0 0]
sage: G = matrix(GF(2), 8, 8, 1).augment(A.T)
sage: LinearCodeFromCheckMatrix(G).minimum_distance()
```

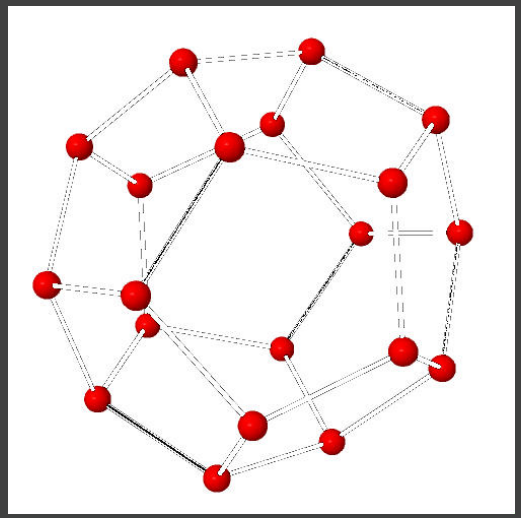
send us a patch

Our interface to GAP sucks, try doing the same over $\mathbb{F}_{2^e}$ to see how much.
Our coding theory is rather basic: things tend to get pushed down to linear codes, i.e., disregarding their structure.

Graph Theory

- builds on `NetworkX` (Los Alamos's Python graph library)
- graph isomorphism testing – Robert Miller's new implementation
- graph databases
- 2d and 3d visualization

```
sage: D = graphs.DodecahedralGraph()
sage: D.show3d()
```



```
sage: E = D.copy()
sage: gamma = SymmetricGroup(20).random_element()
sage: E.relabel(gamma)
sage: D.is_isomorphic(E)
True
sage: D.radius()
5
```

S-Boxes I

- We create the PRESENT S-box

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2); S
(12, 5, 6, 11, 9, 0, 10, 13, 3, 14, 15, 8, 4, 7, 1, 2)
sage: type(S)
<class 'sage.crypto.mq.sbox.SBox'>
```

- evaluate it

```
sage: S(0)
12
sage: S([0,0,0,1])
[0, 1, 0, 1]
```

- compute the interpolation polynomial

```
sage: f = S.interpolation_polynomial()
sage: f(0), S(0)
(a^3 + a^2, 12)
```

$$f = a^{13}(x^{14}+x^{13}) + a^{6}(x^{12}+x^{6}+1) + a^{11}(x^{11}+x^{4}) + a^{14}(x^{10}+x^{9}) + a^{10}(x^{8}+x^{3}+x^{2}) + a^{2}x^{7} + a^{9}x^{5}$$

S-Boxes II

- compute the linear approximation matrix

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: S.linear_approximation_matrix()
[ 8  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0 -4  0 -4  0  0  0  0  0 -4  0  4]
[ 0  0  2  2 -2 -2  0  0  2 -2  0  4  0  4 -2  2]
[ 0  0  2  2  2 -2 -4  0 -2  2 -4  0  0  0 -2 -2]
[ 0  0 -2  2 -2 -2  0  4 -2 -2  0 -4  0  0 -2  2]
[ 0  0 -2  2 -2  2  0  0  2  2 -4  0  4  0  2  2]
[ 0  0  0 -4  0  0 -4  0  0 -4  0  0  4  0  0  0]
[ 0  0  0  4  4  0  0  0  0 -4  0  0  0  0  4  0]
[ 0  0  2 -2  0  0 -2  2 -2  2  0  0 -2  2  4  4]
[ 0  4 -2 -2  0  0  2 -2 -2 -2 -4  0 -2  2  0  0]
[ 0  0  4  0  2  2  2 -2  0  0  0 -4  2  2 -2  2]
[ 0 -4  0  0 -2 -2  2 -2 -4  0  0  0  2  2  2 -2]
[ 0  0  0  0 -2 -2 -2 -2  4  0  0 -4 -2  2  2 -2]
[ 0  4  4  0 -2 -2  2  2  0  0  0  0  2 -2  2 -2]
[ 0  0  2  2 -4  4 -2 -2 -2 -2  0  0 -2 -2  0  0]
[ 0  4 -2  2  0  0 -2 -2 -2  2  4  0  2  2  0  0]
```

S-Boxes III

- and the difference distribution matrix:

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: S.difference_distribution_matrix()
[16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  4  0  0  0  4  0  4  0  0  0  4  0  0]
[ 0  0  0  2  0  4  2  0  0  0  2  0  2  2  2  0]
[ 0  2  0  2  2  0  4  2  0  0  2  2  0  0  0  0]
[ 0  0  0  0  0  4  2  2  0  2  2  0  2  0  2  0]
[ 0  2  0  0  2  0  0  0  0  2  2  2  4  2  0  0]
[ 0  0  2  0  0  0  2  0  2  0  0  4  2  0  0  4]
[ 0  4  2  0  0  0  2  0  2  0  0  0  2  0  0  4]
[ 0  0  0  2  0  0  0  2  0  2  0  4  0  2  0  4]
[ 0  0  2  0  4  0  2  0  2  0  0  0  2  0  4  0]
[ 0  0  2  2  0  4  0  0  2  0  2  0  0  2  2  0]
[ 0  2  0  0  2  0  0  0  4  2  2  2  0  2  0  0]
[ 0  0  2  0  0  4  0  2  2  2  2  0  0  0  2  0]
[ 0  2  4  2  2  0  0  2  0  0  2  2  0  0  0  0]
[ 0  0  2  2  0  0  2  2  2  2  0  0  2  2  0  0]
[ 0  4  0  0  4  0  0  0  0  0  0  0  0  0  4  4]
```

S-Boxes IV

- recover a bunch of boolean polynomials that satisfy the relations of the S-box:

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: S.polynomials() #default: degree=2
[x1*x2 + x0 + x1 + x3 + y3,
 x0*x1 + x0*x2 + x0 + x1 + y0 + y2 + y3 + 1,
 x0*x3 + x1*x3 + x1*y0 + x0*y1 + x0*y2 + x1 + x2 + y2,
 x0*x3 + x0*y0 + x1*y1 + x0 + x2 + y2,
 x0*x2 + x0*y0 + x0*y1 + x1*y2 + x1 + x2 + x3 + y2 + y3, ...]
```

- write the S-box in algebraic normal form

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: P.<y0,y1,y2,y3,x0,x1,x2,x3> = PolynomialRing(GF(2),order='lex')
sage: X = [x0,x1,x2,x3]
sage: Y = [y0,y1,y2,y3]
sage: S.polynomials(X=X,Y=Y,degree=3,groebner=True)
[y0 + x0*x1*x3 + x0*x2*x3 + x0 + x1*x2*x3 + x1*x2 + x2 + x3 + 1,
 y1 + x0*x1*x3 + x0*x2*x3 + x0*x2 + x0*x3 + x0 + x1 + x2*x3 + 1,
 y2 + x0*x1*x3 + x0*x1 + x0*x2*x3 + x0*x2 + x0 + x1*x2*x3 + x2,
 y3 + x0 + x1*x2 + x1 + x3]
```

Boolean Functions I

```
sage: from sage.crypto.boolean_function import *
sage: P.<x0,x1,x2,x3> = BooleanPolynomialRing()
sage: b = x0*x1 + x2*x3
sage: f = BooleanFunction(b)
sage: [b(x[0],x[1],x[2],x[3]) for x in GF(2)^4]
[0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0]
sage: f.truth_table()
(False, False, False, True, False, False, False, True, False, False,
 False, True, True, True, True, False)
```

Boolean Functions II

```
sage: WT = f.walsh_hadamard_transform(); WT
(-4, -4, -4, 4, -4, -4, -4, 4, -4, -4, -4, 4, 4, 4, 4, -4)
sage: f.absolute_walsh_spectrum()
{4: 16}
sage: f.nonlinearity()
6
sage: 2^(4-1) - (1/2)*max([abs(x) for x in WT])
6
```

Boolean Functions III

```
sage: f.autocorrelation()
(16, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
sage: f.absolute_autocorrelation()
{16: 1, 0: 15}
sage: f.absolut_indicator()
0
sage: f.is_bent()
True
sage: f.is_balanced()
False
sage: f.is_symmetric()
False
sage: f.sum_of_square_indicator()
256
sage: f.correlation_immunity()
0
```

```
sage: R.<x> = GF(2^8,'a')[]
sage: B = BooleanFunction(x^31)
sage: B.algebraic_immunity()
4
```

Outline

Here's a picture of a kitten

What are Algebraic Attacks?

1. Algebraic attacks model a cryptographic primitive (such as a block cipher) as a system of equations.
2. Then, by applying (algebraic) transformations to these equations they (attempt to) recover information about the secret of the primitive (the key).

Hence, they are quite different in spirit from statistical techniques such as linear and differential cryptanalysis.

A Polemic History of Algebraic Attacks I

1959 – the "prophecy"[1]

> *"Thus, if we could show that solving a certain system requires at least as much work as solving a system of simultaneous equations in a large number of unknowns, of a complex type, then we would have a lower bound of sorts for the work characteristic."*
>
> – Claude Shannon

[1] This quote is often given to back up the claim that Claude Shannon predicted algebraic attacks. I don't think this quote delivers this. It merely states a design goal not that different from what is now known as provable security.

A Polemic History of Algebraic Attacks II

2002 – the breakthrough

> *Crucial Cipher Flawed, Cryptographers Claim – Two cryptographers say that the new Advanced Encryption Standard, […] has a hole in it. Although some of their colleagues doubt the validity of their analysis, the cryptographic community is on edge, wondering whether the new cipher can withstand a future assault.*
>
> – Science Magazine

A Polemic History of Algebraic Attacks III

2011 – the disillusion

No proper **block cipher** has been broken using algebraic techniques faster than with other techniques

So, why bother?

Algebraic techniques

1. are one of the few choices if very few plaintext-ciphertext pairs are available,
2. become more relevant as focus shifts toward (very) lightweight constructions,
3. can be combined with other techniques such as side-channel attacks,
4. offer a trade-off for researcher time vs. CPU times
5. are fun …well, to some anyway!

Outline

SP-Networks I

We construct an equation system for the block cipher PRESENT, which

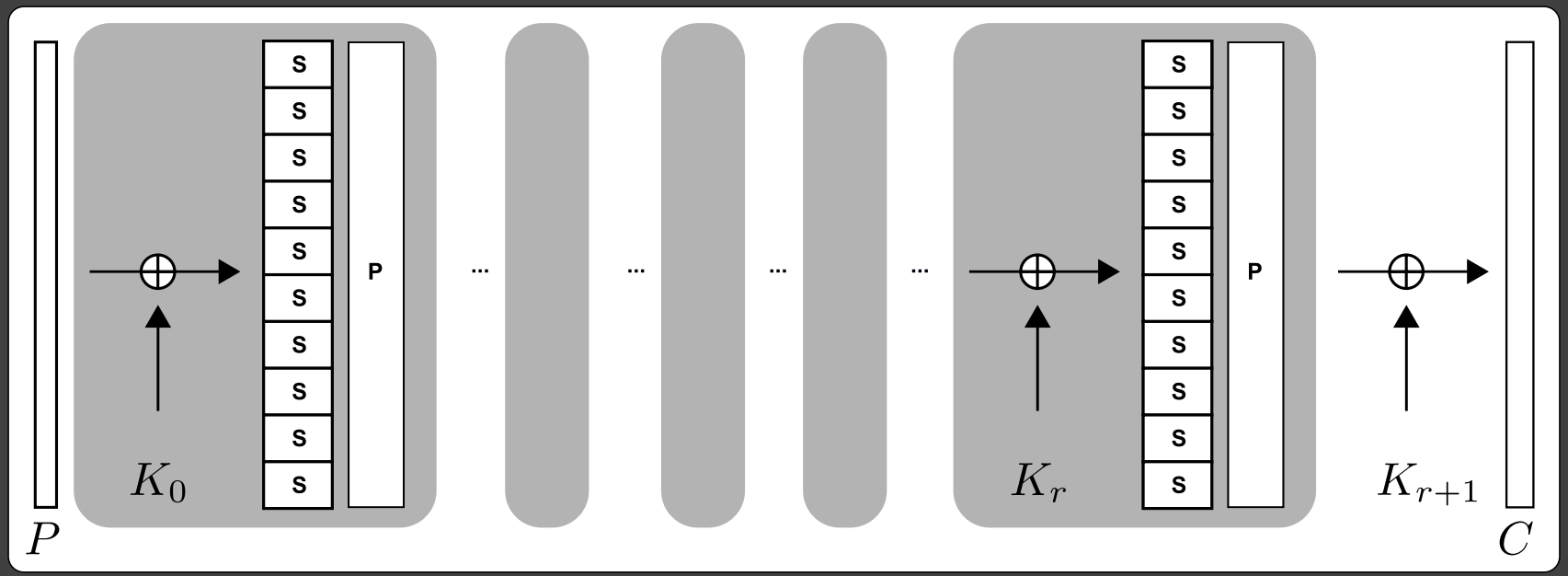
- is a substituion-permutation network,
- has a block size of 64 bits,
- either takes 80-bit or 128-bit keys (PRESENT-80 and -128 resp.)
- has 31 rounds (shorter variants are denoted by PRESENT-{80,128}-*Nr*),
- is conceptually simple, and
- has been extensively studied.

A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, and C. Vikkelsoe.

PRESENT: An ultra-lightweight block cipher.

In *Cryptographic Hardware and Embedded Systems - CHES 2007*, volume 7427 of *Lecture Notes in Computer Science*, pages 450–466, Berlin, Heidelberg, New York, 2007. Springer Verlag.

SP-Networks II



Key Addition and the Permutation Layer

- **Key addition** is easy, if X_i is a bit before key addition and Y_i is a bit after key addition, we write:

$$Y_i + X_i + K_i (= 0).$$

- the **Permutation layer** is just a permutation of wires given by the rule

$$s \cdot j + i \Rightarrow B \cdot i + j \text{ for } 0 \leq j < 16 \text{ and } 0 \leq i < 4,$$

hence we simply rename variables.

S-Box I

The S-box is a non-linear operation. However, finding equations is still easy.

As an example consider the 3-bit (since it fits on the slides) S-box

$$[7, 6, 0, 4, 2, 5, 1, 3].$$

Construct the matrix on the right and perform Gaussian elimination on it.

$$\begin{array}{cccccccc|c}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \\
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & x_0 \\
0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & x_1 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & x_2 \\
1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & y_0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & y_1 \\
1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & y_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & x_0x_1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & x_0x_2 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & x_0y_0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & x_0y_1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & x_0y_2 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & x_1x_2 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & x_1y_0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x_1y_1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & x_1y_2 \\
0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & x_2y_0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & x_2y_1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & x_2y_2 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & y_0y_1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & y_0y_2 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & y_1y_2
\end{array}$$

S-Box II

$$\left(\begin{array}{cccccccc|r}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0y_0 + x_1 + x_2 + y_0 + y_1 + 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & x_0y_0 + x_0 + x_1 + y_2 + 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & x_0y_0 + x_0 + y_0 + 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & x_0y_0 + x_0 + x_2 + y_1 + y_2 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & x_0y_0 + x_0 + x_1 + x_2 + y_0 + y_1 + y_2 + 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & x_0y_0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & x_0y_0 + x_2 + y_0 + y_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & x_0y_0 + x_1 + y_1 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0x_2 + x_1 + y_1 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0x_1 + x_1 + x_2 + y_0 + y_1 + y_2 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0y_1 + x_0 + x_2 + y_0 + y_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0y_0 + x_0y_2 + x_1 + x_2 + y_0 + y_1 + y_2 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_1x_2 + x_0 + x_1 + x_2 + y_2 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0y_0 + x_1y_0 + x_0 + x_2 + y_1 + y_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0y_0 + x_1y_1 + x_1 + y_1 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_1y_2 + x_1 + x_2 + y_0 + y_1 + y_2 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_0y_0 + x_2y_0 + x_1 + x_2 + y_1 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_2y_1 + x_0 + y_1 + y_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x_2y_2 + x_1 + y_1 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & y_0y_1 + x_0 + x_2 + y_0 + y_1 + y_2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & y_0y_2 + x_1 + x_2 + y_0 + y_1 + 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & y_1y_2 + x_2 + y_0
\end{array}\right)$$

S-Box III

If you cannot be bothered to do that yourself, use Sage:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3) # 3-bit S-box
sage: S.polynomials()
[x0*x2 + x1 + y1 + 1,
 x0*x1 + x1 + x2 + y0 + y1 + y2 + 1,
 x0*y1 + x0 + x2 + y0 + y2,
 x0*y0 + x0*y2 + x1 + x2 + y0 + y1 + y2 + 1,
 x1*x2 + x0 + x1 + x2 + y2 + 1,
 x0*y0 + x1*y0 + x0 + x2 + y1 + y2,
 x0*y0 + x1*y1 + x1 + y1 + 1,
 x1*y2 + x1 + x2 + y0 + y1 + y2 + 1,
 x0*y0 + x2*y0 + x1 + x2 + y1 + 1,
 x2*y1 + x0 + y1 + y2,
 x2*y2 + x1 + y1 + 1,
 y0*y1 + x0 + x2 + y0 + y1 + y2,
 y0*y2 + x1 + x2 + y0 + y1 + 1,
 y1*y2 + x2 + y0]
```

If we post-process these polynomials (`groebner=True`), we get 21 quadratic equations and one cubic equation for the S-Box which have a nice algebraic structure.

Putting it all together

- We have equations for the S layer, P layer and the key addtion.
- The key schedule is similar and has one/two S-boxes.
- For each round we introduce $2 \cdot 64$ new state variables for the S layer.
- Adding key schedule and key variables we get $132 \cdot Nr + 80$ variables
- On ther other hand, we get $(22 \cdot 16 + 22 + 64)Nr + 64$ equations

```
sage: s='https://bitbucket.org/malb/research-snippets/raw/tip/present.py'
sage: load(s)
sage: p = PRESENT(Nr=31)
sage: F,s = p.polynomial_system(); F
Polynomial System with 13642 Polynomials in 4172 Variables
```

Solving this system means recovering the key …

Outline

Solver Families

Three families of algorithms are popular in cryptography:

1. SAT solvers: MiniSat2, CryptoMiniSat …
2. Gröbner basis methods: Buchberger's algorithm, F_4, F_5, …
3. Mixed Integer (Linear) Solvers: SCIP, CPLEX, Gurobi, …

It is very useful to understand a bit how these solvers work.

not a valid analysis:

"We put our equations into Magma and it ran out of memory."

Gröbner Bases I

for Cryptographers

- $\mathbb{F}_q$ is a finite field of order q.
- $P = \mathbb{F}_q[x_1, \ldots, x_n]$.
- $\mathcal{I}$ is an ideal $\subset P$. That is, $f, g \in \mathcal{I} \to f + g \in \mathcal{I}$ and $f \in P, g \in \mathcal{I} \to f \cdot g \in \mathcal{I}$.
- $\langle f_1, \ldots, f_m \rangle$ is the ideal spanned by $f_1, \ldots, f_m$.

```
sage: P.<x,y,z> = PolynomialRing(GF(127),order='deglex')
sage: I = ideal(x*y + z, y^3 + 1, z^2 - x*5 - 1)
sage: (x*y + z) + (y^3 + 1) in I
True
sage: x*z*(z^2 - x*5 - 1) in I
True
```

A familiar example:

```
sage: I = ideal([5]); I
Principal ideal (5) of Integer Ring
sage: 5 + 10 in I
True
sage: 3*5 in I
True
```

Gröbner Bases II

for Cryptographers

- A term order decides how we compare monomials, e.g., is xy or y^3 bigger (e.g. degree or variable first)?

```
sage: P.<x,y,z> = PolynomialRing(GF(127),order='lex')
sage: x*y > y^3 # variable then degree
True
sage: P.<x,y,z> = PolynomialRing(GF(127),order='deglex')
sage: x*y > y^3 # degree then variable
False
```

- $\mathrm{M}(f)$ is the set of all monomials in f.
- $\mathrm{LM}(f)$ is the leading or largest monomial in f.

```
sage: P.<x,y,z> = PolynomialRing(GF(127),order='deglex')
sage: f = x*y + x + 3
sage: f.lm()
x*y
sage: f.monomials()
[x*y, x, 1]
```

Gröbner Bases III

for Cryptographers

Definition (Gröbner Basis)

Let $\mathcal{I}$ be an ideal in $\mathbb{F}[x_1, \ldots, x_n]$ and fix a monomial ordering. A finite subset

$$G = \{g_1, \ldots, g_m\} \subset \mathcal{I}$$

is said to be a **Gröbner basis** of $\mathcal{I}$ if for any $f \in \mathcal{I}$ there exists $g_i \in G$ such that

$$\mathrm{LM}(g_i) \mid \mathrm{LM}(f).$$

Gröbner Bases IV

for Cryptographers

Gröbner bases generalise greatest common divisors over $\mathbb{F}[x]$ and row echelon forms over $\mathbb{F}^n$.

```
sage: P.<x,y,z> = PolynomialRing(GF(7), order='deglex')
sage: F = Sequence([ -x*y - x*z - 2*z^2 - 2*y,
                      x*y + 2*y*z - 2*z^2 - x - 2*y,
                      z^2 + 2*x + 3*y - 3*z - 3])
sage: map(lambda f: f.lm(), F.groebner_basis())
[y^3, y^2*z, x^2, x*y, x*z, z^2]

# GCD
sage: R.<x> = PolynomialRing(GF(7))
sage: f = x^2 + 6
sage: I = Ideal(map(P, [R.random_element() * f for _ in range(5)]))
sage: I.groebner_basis()
[x^2 - 1]

# Echelon Form
sage: F = Sequence([-3*y, -2*x - y - 3*z + 2, x + y + 2*z - 1])
sage: F.groebner_basis()
[x - 1, y, z]
sage: A,v = F.coefficient_matrix()
sage: A.echelonize()
sage: (A*v).T
[x - 1     y     z]
```

Gröbner Bases V

for Cryptographers

As a warm-up, consider a linear system of equations over $\mathbb{F}_{127}[x, y, z]$.

$$f = 26y + 52z + 62 = 0$$
$$g = 54y + 119z + 55 = 0$$
$$h = 41x + 91z + 13 = 0$$

$$\begin{pmatrix} 0 & 26 & 52 & 62 \\ 0 & 54 & 119 & 55 \\ 41 & 0 & 91 & 13 \end{pmatrix}$$

After Gaussian elimination:

$$f' = x + 29 = 0$$
$$g' = y + 38 = 0$$
$$h' = z + 75 = 0$$

$$\begin{pmatrix} 1 & 0 & 0 & 29 \\ 0 & 1 & 0 & 38 \\ 0 & 0 & 1 & 75 \end{pmatrix}$$

Thus, $x = -29$, $y = -38$ and $z = -75$ is a solution. We know this because Gaussian elimination produced small enough elements ($z + 75$) such that we can simply read of the solution.

Gröbner Bases VI

for Cryptographers

Now consider two polynomials in $\mathbb{F}_{127}[x, y, z]$ with term ordering **deglex**.

$$f = x^2 + 2xy - 2y^2 + 14z^2 + 22z$$
$$g = x^2 + xy + y^2 + z^2 + x + 2z$$

$$\begin{pmatrix} 1 & 2 & -2 & 14 & 0 & 22 \\ 1 & 1 & 1 & 1 & 1 & 2 \end{pmatrix}$$

$$f = x^2 + 4y^2 - 12z^2 + 2x - 18z$$
$$g' = xy + -3y^2 + 13z^2 - x + 20z$$

$$\begin{pmatrix} 1 & 0 & 4 & -12 & 2 & -18 \\ 0 & 1 & -3 & 13 & -1 & 20 \end{pmatrix}$$

Gaussian elimination still “reduces” the system.

Gröbner Bases VII

for Cryptographers

This approach fails for

$$f = x^2 - 2xy - 2y^2 + 14z^2,$$
$$g = x + y + 2z.$$

since x is not a monomial of f.

However, x divides two monomials of f: x^2 and xy.

To account for those include multiples $m \cdot g$ of g such that

$$\mathrm{LM}(m \cdot g) = m \cdot \mathrm{LM}(g) \in \mathrm{M}(f).$$

Gröbner Bases VIII

for Cryptographers

$$
\begin{aligned}
f &= x^2 - 2xy - 2y^2 + \dots \\
x \cdot g &= x^2 + xy \dots \\
y \cdot g &= xy + y^2 + \dots \\
g &= x + y + 2z
\end{aligned}
$$

$$
\begin{pmatrix}
1 & -2 & -2 & 0 & 0 & 14 & 0 & \dots \\
1 & 1 & 0 & 2 & 0 & 0 & 0 & \dots \\
0 & 1 & 1 & 0 & 2 & 0 & 0 & \dots \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & \dots
\end{pmatrix}
$$

$$
\begin{aligned}
f' &= x^2 + 4yz + 14z^2, \\
h_1 &= xy + 2xz + -4yz - \dots, \\
h_2 &= y^2 - 2xz + 6yz + \dots, \\
g &= x + y + 2z
\end{aligned}
$$

$$
\begin{pmatrix}
1 & 0 & 0 & 0 & \dots & 0 & \dots \\
0 & 1 & 0 & 2 & \dots & 0 & \dots \\
0 & 0 & 1 & -2 & \dots & 0 & \dots \\
0 & 0 & 0 & 0 & \dots & 1 & \dots
\end{pmatrix}
$$

Let's call the preprocessing we performed "symbolic preprocessing" …but that alone is still not enough to solve the system.

Gröbner Bases IX

for Cryptographers

Finally, consider

$$\begin{aligned} f &= yx + 1, \\ g &= zx + 2. \end{aligned}$$

Neither $\mathrm{LM}(f)$ nor $\mathrm{LM}(g)$ divides any monomial in the other polynomial. However, we have

$$\begin{aligned} zf - yg &= z(yx + 1) - y(zx + 2), \\ &= xyz + z - xyz - 2y, \\ &= z - 2y. \end{aligned}$$

We constructed multiples of f and g such that when we subtract them their leading terms cancel out and something smaller is produced: we constructed an **S-polynomial**.

Definition (S-Polynomial)

Let $f, g \in \mathbb{F}[x_1, \ldots, x_n]$ be non-zero polynomials.

Let x^γ be the least common multiple of $\mathrm{LM}(f)$ and $\mathrm{LM}(g)$, written as

$$x^\gamma = \mathrm{LCM}(\mathrm{LM}(f), \mathrm{LM}(g)).$$

The S-polynomial of f and g is defined as

$$S(f,g) = \frac{x^\gamma}{\mathrm{LT}(f)} \cdot f - \frac{x^\gamma}{\mathrm{LT}(g)} \cdot g.$$

Gröbner Bases XI

for Cryptographers

It is **sufficient** to consider **only** S-polynomials since **any** reduction of leading terms can be attributed to S-polynomials.

[Buc65] Bruno Buchberger
Ein Algorithmus zum Auffinden der Basiselemente des Restklassenrings nach einem nulldimensionalen Polynomideal,
Phd Thesis at Universität Innsbruck, 1965.

[Buc06] Bruno Buchberger
Bruno Buchberger's PhD thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal
Journal of Symbolic Computation, 41:3-4, p. 475-511, 2006.

Gröbner Bases XII

for Cryptographers

Input: $F = [f_1, \ldots, f_m]$ – list of polynomials
Output: a Gröbner basis for $\langle f_1, \ldots, f_{m-1} \rangle$

```
1 begin
2     while True do
3         F̄ ← multiply all pairs f_i, f_j ∈ F by m_i, m_j such that LM(m_i f_i) = LM(m_j f_j);
4         F̄ ← perform "symbolic preprocessing" on F̄ ∪ F;
5         F̃ ← peform Gaussian elimination on F̄ ;
6         F ← F ∪ {f ∈ F̃ with ∀g ∈ F we have LM(g) ∤ LM(f)};
7         if F didn't change in the last iteration then
8             return F;
```

Algorithm 1: simplified F_4

Gröbner Bases XIII

for Cryptographers

1 **begin**
2 **while** True **do**
3 $\overline{F} \leftarrow$ multiply all pairs $f_i, f_j \in F$ by m_i, m_j such that $\mathrm{LM}(m_i f_i) = \mathrm{LM}(m_j f_j)$;
4 $\overline{F} \leftarrow$ perform "symbolic preprocessing" on $\overline{F} \cup F$;
5 $\tilde{F} \leftarrow$ peform Gaussian elimination on $\overline{F}$;
6 $F \leftarrow F \cup \{f \in \tilde{F}$ with $\forall g \in F$ we have $\mathrm{LM}(g) \nmid \mathrm{LM}(f)\}$;
7 **if** *F* **didn't change in the last iteration then**
8 **return** *F*;

Buchberger select one pair in line 3 and use polynomial division instead of Gaussian elimination in line 5; implemented everywhere

F_4 use Buchberger's criteria in line 3 to avoid useless pairs (= zero rows in the matrix); implemented in Magma, PolyBoRi, FGB

F_5 use criteria in lines 3 and 4 such that all matrices have full rank under some assumption; implementation worked on in Singular

Gröbner Bases XIV

for Cryptographers

1 **begin**
2 **while** True **do**
3 $\overline{F} \leftarrow$ multiply all pairs $f_i, f_j \in F$ by m_i, m_j such that $\mathrm{LM}(m_i f_i) = \mathrm{LM}(m_j f_j)$;
4 $\overline{F} \leftarrow$ perform "symbolic preprocessing" on $\overline{F} \cup F$;
5 $\tilde{F} \leftarrow$ peform Gaussian elimination on $\overline{F}$;
6 $F \leftarrow F \cup \{f \in \tilde{F}$ with $\forall g \in F$ we have $\mathrm{LM}(g) \nmid \mathrm{LM}(f)\}$;
7 **if** F **didn't change in the last iteration then**
8 **return** F;

(Mutant)XL multiply by everything up to some degree in line 3 and skip line 4 (worse than Algorithm 1 because of redundancies)

XSL make some (very bad!) choice in line 3 and line 4 (worse than Algorithm 1 because of bad choices)

ElimLin always stay at degree 2 → line 4 + line 5 (less powerful than Algorithm 1)

Gröbner Bases in Sage I

Let's play a bit with Gröbner bases in Sage:

```
sage: K = GF(32003)
sage: T = TermOrder("deglex",2) + TermOrder("deglex",2)
sage: P.<w,x,y,z> = PolynomialRing(K,order=T)
sage: I = sage.rings.ideal.Katsura(P)
sage: [g.lm() for g in I.groebner_basis()] # Singular
[w, x, y*z^3, z^4, y^3, y^2*z]
sage: I.dimension() # finite number of solutions
0
sage: V = I.variety(); V # solutions
[{y: 0, z: 0, w: 1, x: 0}, {y: 0, z: 10668, w: 10668, x: 0}]
sage: J = I.change_ring(P.change_ring(QQ))  # over the rationals
sage: J.variety()
[{y: 0, z: -32002/3, w: -32002/3, x: 0}, {y: 0, z: 0, w: -32002, x: 0}]
sage: len(J.variety(CC)) # over the complex numbers
8
```

Gröbner Bases in Sage II

We estimate the complexity before solving:

```
sage: n = 10; P = PolynomialRing(GF(32003), n, 'x')
sage: F = [P.random_element() for _ in range(P.ngens()+2)]
sage: s = random_vector(GF(32003),n)
sage: I = Ideal(f-f(*s) for f in F)
sage: D = I.degree_of_semi_regularity()
sage: D, log(binomial(n + D, n)^3, 2).n() # prediction: degree six
6, 34.65...
sage: I.groebner_basis('magma',prot='sage')
...
Leading term degree:  4. Critical pairs: 178.
Leading term degree:  5. Critical pairs: 515.
Leading term degree:  5. Critical pairs: 1155.
Leading term degree:  3. Critical pairs: 2845.
Leading term degree:  2. Critical pairs: 2850.
Leading term degree:  3. Critical pairs: 2795.
Leading term degree:  4. Critical pairs: 2605.
Leading term degree:  5. Critical pairs: 1609.
Leading term degree:  6. Critical pairs: 864 (all pairs ...
Leading term degree:  7. Critical pairs: 4 (all pairs ...
Leading term degree:  8. Critical pairs: 1 (all pairs ...

Highest degree reached during computation:  5.
[x0 + 7334, x1 - 12304, x2 - 7977, x3 - 8365, x4 - 7982, x5 + 676, ...]
```

Gröbner Bases in Sage III

Sage has a very good implementation of Gröbner basis computations over $\mathbb{F}_2[x_1,\ldots,x_n]/\langle x_1^2+x_1,\ldots,x_n^2+x_n\rangle$ thanks to `PolyBoRi`.

```
sage: B = BooleanPolynomialRing(50,'x',order='deglex')
sage: s = random_vector(GF(2),50)
sage: F = [B.random_element() for _ in range(500)]
sage: I = Ideal(f-f(*s) for f in F)
sage: G = I.groebner_basis(); G # PolyBoRi
Polynomial Sequence with 50 Polynomials in 50 Variables
sage: sorted(G)[0], s[0]
(x0 + 1, 1)
sage: I.variety()
[{x40: 0, x42: 1, x44: 0, ..., x23: 0, x25: 1}]
```

SAT Solvers I

- The SAT problem is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses.
- The question is: given the expression, is there some assignment of TRUE and FALSE values to the variables that will make the entire expression true?
- A **literal** is either a variable or the negation of a variable
- A **clause** is a disjunction of literals ($\vee$ = OR).
- A formula is in **Conjunctive Normal Form** (CNF) if is a conjunction ($\wedge$ = AND) of clauses.

$x_1 \vee \neg x_2 \vee x_3$

$\neg x_1 \vee \neg x_3$

```
sage: from sage.sat.solvers import CryptoMiniSat
sage: cms = CryptoMiniSat()
sage: cms.add_clause( (1, -2, 3) )
sage: cms.add_clause( (-1, -3) )
```

SAT Solvers II

We can solve (Boolean) polynomial systems using SAT solvers using ANF to CNF conversion:

```
sage: import sage.sat.converters as satconv
sage: from satconv.polybori import CNFEncoder
sage: from sage.sat.solvers import CryptoMiniSat
sage: B.<a,b> = BooleanPolynomialRing()
sage: cms = CryptoMiniSat()
sage: ce = CNFEncoder(cms, B)
sage: ce([a*b + b +1])
[None, a, b]
sage: cms.clauses()
[((2,), False, None), ((-1,), False, None)]
```

Gregory V. Bard, Nicolas T. Courtois, and Chris Jefferson.

Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers.

Cryptology ePrint Archive, Report 2007/024, 2007.

SAT Solvers III

```
1 begin
2     while True do
3         simplify clauses;
4         if contradiction then
5             backtrack;
6         if solution then
7             return ;
8         guess something;
```

- SAT solvers decide satisfiability, hence they will terminate once **one** solution is found.
- …in contrast to Gröbner basis methods.
- SAT solvers are randomised: one success does not constitute an average running time.
- Run hundreds/thousands of experiments with lots of re-randomisation.
- The conversion from ANF to CNF can make a huge difference, Sage supports two strategies at the moment.
- Different solvers behave differently.

```
sage: from sage.sat.boolean_polynomials import solve as solve_sat
sage: sr = mq.SR(1,2,2,4,gf2=True,polybori=True)
sage: set_random_seed(1337^3)
sage: F,s = sr.polynomial_system()
sage: len(solve_sat(F))
1
sage: len(solve_sat(F, n=infinity))
3
```

SAT Solvers in Sage I

Sage has an interface for SAT solving via **CryptoMiniSat** which supports XOR clauses

```
sage: from sage.sat.solvers import CryptoMiniSat
sage: cms = CryptoMiniSat(verbosity=3)
sage: cms.add_clause((1,2,-3)) # x1 OR x2 OR -x3
sage: cms.add_clause((1,-2,3)) # x1 OR -x2 OR x3
sage: cms.add_xor_clause((1,2,3), 0) # (x1 XOR x2 XOR x3 = 1)
sage: cms.add_xor_clause((1,3), 1) # (x1 XOR x3 = 0)
sage: cms()
...
(None, True, True, True)
```

or via any solver supporting the **DIMACS** format

```
sage: from sage.sat.boolean_polynomials import solve as solve_sat
sage: F,s = mq.SR(1,1,1,4,gf2=True,polybori=True).polynomial_system()
sage: solve_sat(F, solver=sage.sat.solvers.RSat)
[{k003: 0,  k002: 0,  k001: 1,  k000: 1,  ...,
  k103: 0,  k102: 1,  k101: 0,  k100: 0}]
```

SAT Solvers in Sage II

Sage can take care of the ANF to CNF conversion details:

```
sage: load('https://bitbucket.org/malb/research-snippets/raw/tip/present.py
sage: from sage.sat.boolean_polynomials import solve as solve_sat
sage: F,s = PRESENT(Nr=2).polynomial_system()
sage: sprime = solve_sat(F,s_verbosity=3)
```

…and also learn new equations using a SAT solver with early abort:

```
sage: set_random_seed(2300)
sage: sr = mq.SR(1,4,4,4,gf2=True,polybori=True)
sage: F,s = sr.polynomial_system()
sage: from sage.sat.boolean_polynomials import learn as learn_sat
sage: H = learn_sat(F, s_maxrestarts=20, interreduction=True)
sage: H[-1]
k001503*s021*x011502 + s021*x011502 + k001503*x011502 + x011502
```

Mixed Integer Programming I

- MIP minimises (or maximises) a linear function $c^T x$ subject to linear equality and inequality constraints given by linear inequalities

$$Ax \leq b.$$

- We restrict some variables to integer values while others may take any real values.
- The main advantage of MIP solvers compared to other branch-and-cut solvers (SAT solvers etc.) is that they can relax the problem to an (easy) floating point problem.
- This allows to obtain lower and upper bounds for $c^T x$ which can be used to cut search branches.
- The non-linear generalisation is called Constraint Integer Programming (CIP).
- Solvers: `CPLEX` & `Gurobi` (accademic licenses available), `SCIP` (≈ open-source)

Mixed Integer Programming II

We can convert a polynomial $f \in \mathbb{F}_2[x_1, \ldots, x_n]$ to MIP and then use an off-the-shelf MIP solver (in this example SCIP).

```
sage: from sage.libs.scip.scip import SCIP # trac 10879
sage: B.<a,b,c> = BooleanPolynomialRing()
sage: f = a*c + a + b + c + 1
sage: s = SCIP(maximization=False,name="icebreak")
sage: boolean_polynomials(s, Sequence([f]))
down: {b: 1, c: 0, a: 2}
  up: {0: c, 1: b, 2: a}
sage: _ = s.solve()
sage: s.get_all_solutions()
[({0: 0.0, 1: 1.0, 2: 0.0, 3: 0.0}, 1.0)]
```

Julia Borghoff, Lars R. Knudsen, and Mathias Stolpe.

Bivium as a Mixed-Integer Linear programming problem.

In Matthew G. Parker, editor, *Cryptography and Coding – 12th IMA International Conference*, volume 5921 of *Lecture Notes in Computer Science*, pages 133–152, Berlin, Heidelberg, New York, 2009. Springer Verlag.

Mixed Integer Programming in Sage

Sage also has a highlevel interface to Mixed Integer Linear solving

```
sage: g = graphs.PetersenGraph()
sage: p = MixedIntegerLinearProgram(maximization=True)
sage: b = p.new_variable()
sage: p.set_objective(sum([b[v] for v in g]))
sage: for (u,v) in g.edges(labels=None):
...       p.add_constraint(b[u] + b[v], max=1)
sage: p.set_binary(b)
sage: p.solve(objective_only=True)
4.0
```

which supports many backends: `GLPK`, `Coin`, `Gurobi`, `CPLEX`.

Outline

Here's another picture of a kitten

Implications I

. . . or "I cannot be bothered to work this out by hand"

- Consider an arbitrary function $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ and its boolean polynomial representation $f_1, \ldots, f_m$
- Let $x_1, \ldots x_n$ be the input variables and $y_1, \ldots, y_m$ the output variables
- Consider the ideal $\mathcal{I} = \langle f_1, \ldots, f_m \rangle$:
  - Every member g of this ideal is a combination of $f_1, \ldots, f_m$.
  - If $f_1, \ldots, f_m$ vanish, so does g.

$f_1, \ldots, f_m$ implies g

"If $f_1, \ldots, f_m$ hold, so does g".

Implications II

. . . or “I cannot be bothered to work this out by hand”

Example: The PRESENT S-box: [c, 5, 6, b, 9, 0, a, d, 3, e, f, 8, 4, 7, 1, 2]:

$$
\begin{aligned}
y_0 &= x_0x_1x_3 + x_0x_2x_3 + x_0 + x_1x_2x_3 + x_1x_2 + x_2 + x_3 + 1,\\
y_1 &= x_0x_1x_3 + x_0x_2x_3 + x_0x_2 + x_0x_3 + x_0 + x_1 + x_2x_3 + 1,\\
y_2 &= x_0x_1x_3 + x_0x_1 + x_0x_2x_3 + x_0x_2 + x_0 + x_1x_2x_3 + x_2,\\
y_3 &= x_0 + x_1x_2 + x_1 + x_3.
\end{aligned}
$$

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: B.<y0,y1,y2,y3,x0,x1,x2,x3> = BooleanPolynomialRing(order='lex')
sage: X = (x0,x1,x2,x3); Y = (y0,y1,y2,y3)
sage: S.polynomials(X=X, Y=Y, degree=3, groebner=True)
[y0 + x0*x1*x3 + x0*x2*x3 + x0 + x1*x2*x3 + x1*x2 + x2 + x3 + 1,
 y1 + x0*x1*x3 + x0*x2*x3 + x0*x2 + x0*x3 + x0 + x1 + x2*x3 + 1,
 y2 + x0*x1*x3 + x0*x1 + x0*x2*x3 + x0*x2 + x0 + x1*x2*x3 + x2,
 y3 + x0 + x1*x2 + x1 + x3]
```

Implications III

. . . or “I cannot be bothered to work this out by hand”

- Let c be a condition on the input variables (in polynomial form).

  Example: $x_0 = 1$ `sage: c = x0 + 1`

Implications IV

. . . or “I cannot be bothered to work this out by hand”

- Calculate a Gröbner basis for $\langle c, f_1, \ldots, f_m \rangle$ in an elimination ordering which eliminates input variables first.

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: BPRing = BooleanPolynomialRing
sage: T = TermOrder('lex')
sage: B.<x0,x1,x2,x3,y0,y1,y2,y3> = BPRing(order=T) # x > y
sage: X = (x0,x1,x2,x3); Y = (y0,y1,y2,y3)
sage: F = Sequence( S.polynomials(X=X, Y=Y, degree=3) )
sage: c = x0 + 1
sage: F += [c]
sage: G = F.groebner_basis()
```

Implications V

. . . or “I cannot be bothered to work this out by hand”

- The smallest elements of this Gröbner basis will be polynomials with a minimum number of input variables (if possible, none). Call them $g_0, \ldots, g_{r-1}$.
- These polynomials are **implied** by the polynomials $f_1, \ldots, f_m$ and the condition c.

“If $f_1, \ldots, f_m$ and the condition c hold, so do $g_1, \ldots, g_r$”

```
sage: G[-1]
y1*y2*y3 + y1*y3
```

Implications VI

. . . or "I cannot be bothered to work this out by hand"

- Moreover, **all** on the output bits that are implied by f under condition c are **combinations** of $g_1, \ldots, g_r$
- If we pick the term ordering right, $g_1, \ldots, g_r$ have minimal degree.

For a given function f under a precondition c we can calculate **all** conditions on the output bits that **must** hold.

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2)
sage: BPRing = BooleanPolynomialRing
sage: T = TermOrder('deglex',4) + TermOrder('deglex',4) # order!
sage: B.<x0,x1,x2,x3,y0,y1,y2,y3> = BPRing(order=T) # x > y
sage: X = (x0,x1,x2,x3); Y = (y0,y1,y2,y3)
sage: F = Sequence( S.polynomials(X=X, Y=Y, degree=3) )
sage: c = x0 + 1
sage: F += [c]
sage: G = F.groebner_basis()
sage: G[4:]
[y1*y2*y3 + y1*y3,
 y0*y1 + y1*y2 + y2*y3 + y0 + y1 + y2 + y3 + 1,
 y0*y2 + y1*y2 + y2*y3 + y0 + y1 + y2 + y3 + 1,
 y0*y3 + y1*y3 + y2*y3 + y0 + y1 + y2 + y3 + 1]
```

Implications VII

. . . or “I cannot be bothered to work this out by hand”

Applications:

Differential: algebraic description of all possible output differences under some input difference.

Cond. Diff.: conditional relations on the plaintext and the key bits.

Integral: algebraic descriptions on the output bits after r rounds.

Martin Albrecht, Carlos Cid, Thomas Dullien, Jean-Charles Faugère, and Ludovic Perret.

Algebraic precomputations in Differential and Integral Cryptanalysis.

In *INSCRYPT 2010 – Information Security and Cryptology 6th International Conference*, *Lecture Notes in Computer Science*, 18 pages, October 2010.

Algebraic Techniques and Integral Cryptanalysis I

In integral or **higher-order differential cryptanalysis** the attacker encrypts plaintexts with some structure such that the output (after some rounds) also has some (algebraic) structure.

Algebraic Techniques and Integral Cryptanalysis II

- In [ZRHD08] **bit-pattern based integral attacks** against up to 7 rounds of PRESENT are proposed, based on a 3.5 round distinguisher.
- The attacker prepares 16 chosen plaintexts which agree in all bit values except the bits at the positions 51, 55, 59, 63.
- These four bits take all possible values $(0,0,0,0), (0,0,0,1), \ldots, (1,1,1,1)$.
- Then he input bits to the 4th round are then balanced, i.e., the sum of all bits at the same bit position across all 16 encryptions is zero.
- If $X_{i,j,k}$ denotes the k-th input bit of the j-th round of the i-th encryption, we have that

$$0 = \sum_{i=0}^{15} X_{i,4,k} \text{ for } 0 \leq k < 64.$$

Muhammad Reza Z'Aba, Håvard Raddum, Matt Henricksen, and Ed Dawson. Bit-pattern based integral attacks.
In Kaisa Nyberg, editor, *Fast Software Encryption 2008*, number 5086 in Lecture Notes In Computer Science, pages 363–381, Berlin, Heidelberg, New York, 2008. Springer Verlag.

Algebraic Techniques and Integral Cryptanalysis III

I presume the result in [ZRHD08] was found by carefully tracing relations between bits through the cipher, I am too lazy for that.

- Setup an equation system for the 16 plaintexts as in [ZRHD08],
- run a Gröbner basis computation up to degree 2 under a term ordering where variables from small rounds are eliminated first.
- This produces 500 linear polynomials in $X_{i,4,k}$ and 26 linear polynomials in $X_{i,5,k}$

Martin R. Albrecht
Algorithmic Algebraic Techniques and their Application to Block Cipher Cryptanalysis
Phd Thesis at University of London, 2010

Algebraic Techniques and Integral Cryptanalysis IV

Cipher	Method	#P	Wall time
PRESENT-80-5	HODC	$5 \cdot 2^4$	$\approx 2^{25.7}$ CPU cycles
PRESENT-80-5	AHODC	2^4	$\approx 2^{23.3}$ CPU cycles
PRESENT-80-6	HODC	$2^{22.4}$	$\approx 2^{41.7}$ CPU cycles
PRESENT-80-6	AHODC	2^{20}	$\approx 2^{39.3}$ CPU cycles
PRESENT-80-7	HODC	$2^{24.4}$	$\approx 2^{100.1}$ CPU cycles
PRESENT-80-7	AHODC	$2^{21.9}$	$\approx 2^{97.8}$ CPU cycles
KTANTAN32-65	AHODC	2^5	59004.10 s

Designing Linear Layers I

…or “I cannot be bothered to design an algorithm for that”

- Assume we have a (sparse) $n \times n$ matrix over $\mathbb{F}_2$ with a good differential branch number.
- We can use this matrix to construct an $2n \times 2n$ matrix over $\mathbb{F}_2$ with the same number of ones per row/column and branch number.

$$\begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} \end{pmatrix} \begin{pmatrix} b_{1,1} & \dots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \dots & b_{n,n} \end{pmatrix} \Rightarrow \left(\begin{array}{cc|c|cc} A_{1,1} & 0 & \dots & A_{1,n} & 0 \\ 0 & b_{1,1} & \dots & 0 & b_{1,n} \\ \hline \vdots & & \ddots & & \vdots \\ \hline A_{n,1} & 0 & \dots & A_{n,n} & 0 \\ 0 & b_{n,1} & \dots & 0 & b_{n,n} \end{array}\right)$$

- We want to avoid that both $a_{i,j} = b_{i,j=0}$ to maximise diffusion.

Task

Find $B = P \cdot A \cdot Q$ such that diffusion is maximised where P, Q are permutations.

Designing Linear Layers II

…or “I cannot be bothered to design an algorithm for that”

We would go about this problem something like this:

1. Try a bunch of random permutations and hope to get lucky (incomplete)
2. Try some heuristic, such as local optimisation (incomplete)
3. Run an exhaustive search over all permutation matrices, prunning search trees based on fill-in (complete, but takes a while to implement)

…or we can recognise this as a Constraint Integer Programming problem

Designing Linear Layers III

…or "I cannot be bothered to design an algorithm for that"

We consider the matrices

$$P = \begin{pmatrix} p_{1,1} & \dots & p_{1,n} \\ \vdots & \ddots & \vdots \\ p_{n,1} & \dots & p_{n,n} \end{pmatrix}, Q = \begin{pmatrix} q_{1,1} & \dots & q_{1,n} \\ \vdots & \ddots & \vdots \\ q_{n,1} & \dots & q_{n,n} \end{pmatrix}.$$

where $p_{i,j}, q_{i,j}$ are boolean variables and denote

$$\begin{pmatrix} b_{1,1} & \dots & b_{1,n} \\ \vdots & \ddots & \vdots \\ b_{n,1} & \dots & b_{n,n} \end{pmatrix} = P \cdot A \cdot Q,$$

We add

1. quadratic constraints expressing $b_{i,j}$ in terms of $P \cdot A \cdot Q$.
2. linear constraints that all rows/columns of P and Q have exactly one 1 per row/column.
3. OR constraints $v_{i,j} = a_{i,j} \vee b_{i,j}$.

We then maximize $\sum v_{i,j}$ using an off-the-shelf CIP solver such as SCIP.

Designing Linear Layers IV

. . . or “I cannot be bothered to design an algorithm for that”

- Including documentation, error checking, comments and generalised for arbitrarily many permutations of A this takes about 100 lines of code to implement in Sage.
- It will run (much) slower than a dedicated algorithm or implementation.

Trade Off

At what ratio do you “trade” CPU cycles and your time?

Algebraic Techniques and Side-Channel Cryptanalysis I

. . . or full ciphers are too hard

- Solving full ciphers seems way beyond what algebraic attacks can deliver.
- Side-channel attacks provide information about the internal state of an encryption operation to the attacker.
- This information can then be used to recover key information.

Algebraic Techniques and Side-Channel Cryptanalysis II

…or full ciphers are too hard

This means,

- tracking information from the leak back to the key,
- where often only a limited number of readings are available,

…so we might want to throw algebraic solvers at the problem.

It is always possible to find a dedicated attack that is at least as good as the generic "algebraic side-channel" attack and plausibly one can often find dedicated attacks which are strictly better. Yet, the question is how hard it is to find these attacks.

Algebraic Techniques and Side-Channel Cryptanalysis III

. . . or full ciphers are too hard

Example: Power readings reveal information about S-box bits, combine these with cipher description and solve using a SAT solver.

Mathieu Renauld and Francois-Xavier Standaert.
Algebraic Side-Channel Attacks.
In *INSCRYPT 2009 – Information Security and Cryptology 5th International Conference*, volume 6151 of *Lecture Notes in Computer Science*, pages 393-410, Berlin, Heidelberg, New York, 2009. Springer Verlag.

Algebraic Techniques and Side-Channel Cryptanalysis IV

. . . or full ciphers are too hard

Example: Cold boot attacks recover noisy versions of the key schedule output, recover noise-free version via polynomial system solving with noise via Mixed Integer Programming.

Martin Albrecht and Carlos Cid.
Cold Boot Key Recovery by Solving Polynomial Systems with Noise
In *ACNS 2011 – 9th International Conference on Applied Cryptography and Network Security*, in *Lecture Notes in Computer Science*, Berlin, Heidelberg, New York, 2011. Springer Verlag.

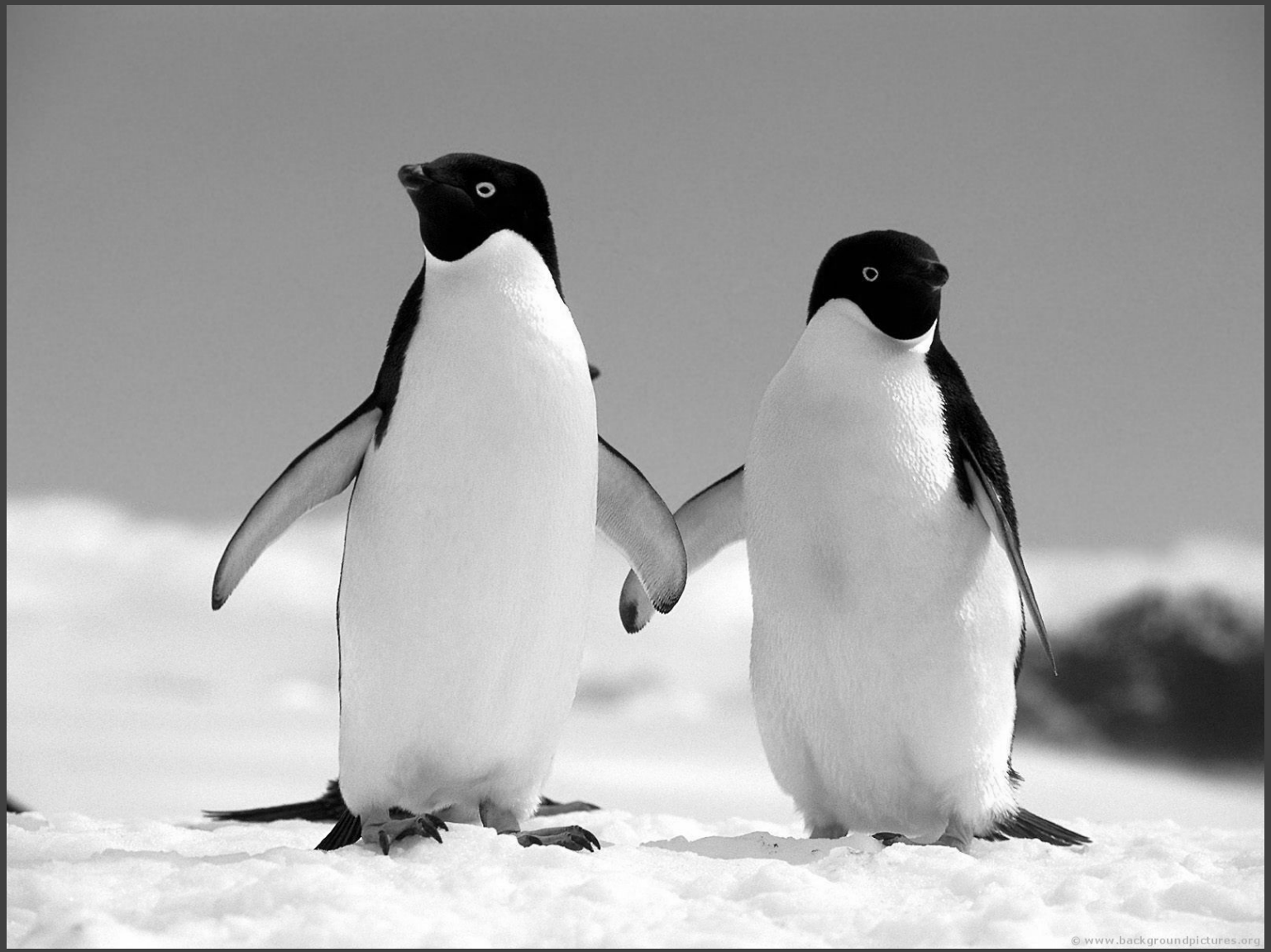
Algebraic Techniques and Side-Channel Cryptanalysis V

. . . or full ciphers are too hard

Example: Fault attacks introduce a fault in the encryption and exploit the result wrong result, the key recovery can be accomplished using a SAT solver.

Philipp Jovanovic and Martin Kreuzer and Ilia Polian
An Algebraic Fault Attack on the LED Block Cipher
Cryptology ePrint Archive, Report 2012/400

Questions?



Thank You!