Stream ciphers Grain and Trivium

Santanu Sarkar

IIT Madras, Chennai

Outline of the Talk

First Part:

1. Designs of Grain family
2. Designs of Trivium

Second Part:

2. Analysis of Grain v1
2. Analysis of Trivium

eStrem portfolio

Software

1. HC-128
2. Rabbit
3. Salsa 20
4. SOSEMANUK

Hardware

1. Grain v1
2. MICKEY 2.0
3. Trivium

Grain Family

- Proposed by Hell, Johansson and Meier in 2005
- Part of eStream portfolio
- Grain v1, Grain 128 and Grain 128a

Grain v1

Consists of an 80 bit LFSR and an 80 bit NFSR.

The LFSR update function is

$$y_{t+80} = y_{t+62} + y_{t+51} + y_{t+38} + y_{t+23} + y_{t+13} + y_t.$$

NFSR update

The NFSR state is updated as follows

$$x_{t+80} = y_t + g(x_{t+63}, x_{t+62}, x_{t+60}, x_{t+52}, x_{t+45}, x_{t+37}, x_{t+33}, x_{t+28}, x_{t+21}, x_{t+15}, x_{t+14}, x_{t+9}, x_t) \text{ where}$$

$$g(x_{t+63}, x_{t+62}, x_{t+60}, x_{t+52}, x_{t+45}, x_{t+37}, x_{t+33}, x_{t+28}, x_{t+21}, x_{t+15}, x_{t+14}, x_{t+9}, x_t)$$

$$\begin{aligned} &= x_{t+62} + x_{t+60} + x_{t+52} + x_{t+45} + x_{t+37} + x_{t+33} + x_{t+28} + x_{t+21} + \\ &\quad x_{t+14} + x_{t+9} + x_t + x_{t+63}x_{t+60} + x_{t+37}x_{t+33} + x_{t+15}x_{t+9} + \\ &\quad x_{t+60}x_{t+52}x_{t+45} + x_{t+33}x_{t+28}x_{t+21} + x_{t+63}x_{t+45}x_{t+28}x_{t+9} + \\ &\quad x_{t+60}x_{t+52}x_{t+37}x_{t+33} + x_{t+63}x_{t+60}x_{t+21}x_{t+15} + \\ &\quad x_{t+63}x_{t+60}x_{t+52}x_{t+45}x_{t+37} + x_{t+33}x_{t+28}x_{t+21}x_{t+15}x_{t+9} + \\ &\quad x_{t+52}x_{t+45}x_{t+37}x_{t+33}x_{t+28}x_{t+21} \end{aligned}$$

Output Keystream

$$z_t = \bigoplus_{a \in A} x_{t+a} + h(y_{t+3}, y_{t+25}, y_{t+46}, y_{t+64}, x_{t+63})$$

where $A = \{1, 2, 4, 10, 31, 43, 56\}$ and

$$h(s_0, s_1, s_2, s_3, s_4) = s_1 + s_4 + s_0 s_3 + s_2 s_3 + s_3 s_4 + s_0 s_1 s_2 + s_0 s_2 s_3 + s_0 s_2 s_4 + s_1 s_2 s_4 + s_2 s_3 s_4$$

Key Scheduling Algorithm (KSA)

- Grain v1 uses 80-bit key K, and 64-bit initialization vector IV.
- The key is loaded in the NFSR
- The IV is loaded in the 0^{th} to the 63^{th} bits of the LFSR.
- The remaining 64^{th} to 79^{th} bits of the LFSR are loaded with 1.
- Then, for the first 160 clocks, the key-stream bit z_t is XOR-ed to both the LFSR and NFSR update functions.

Pseudo-Random key-stream Generation Algorithm (PRGA)

- After the KSA, z_t is no longer XOR-ed to the LFSR and the NFSR.
- Thus, the LFSR and NFSR are updated as $y_{t+n} = f(Y_t), x_{t+n} = y_t + g(X_t)$.

Grain overview

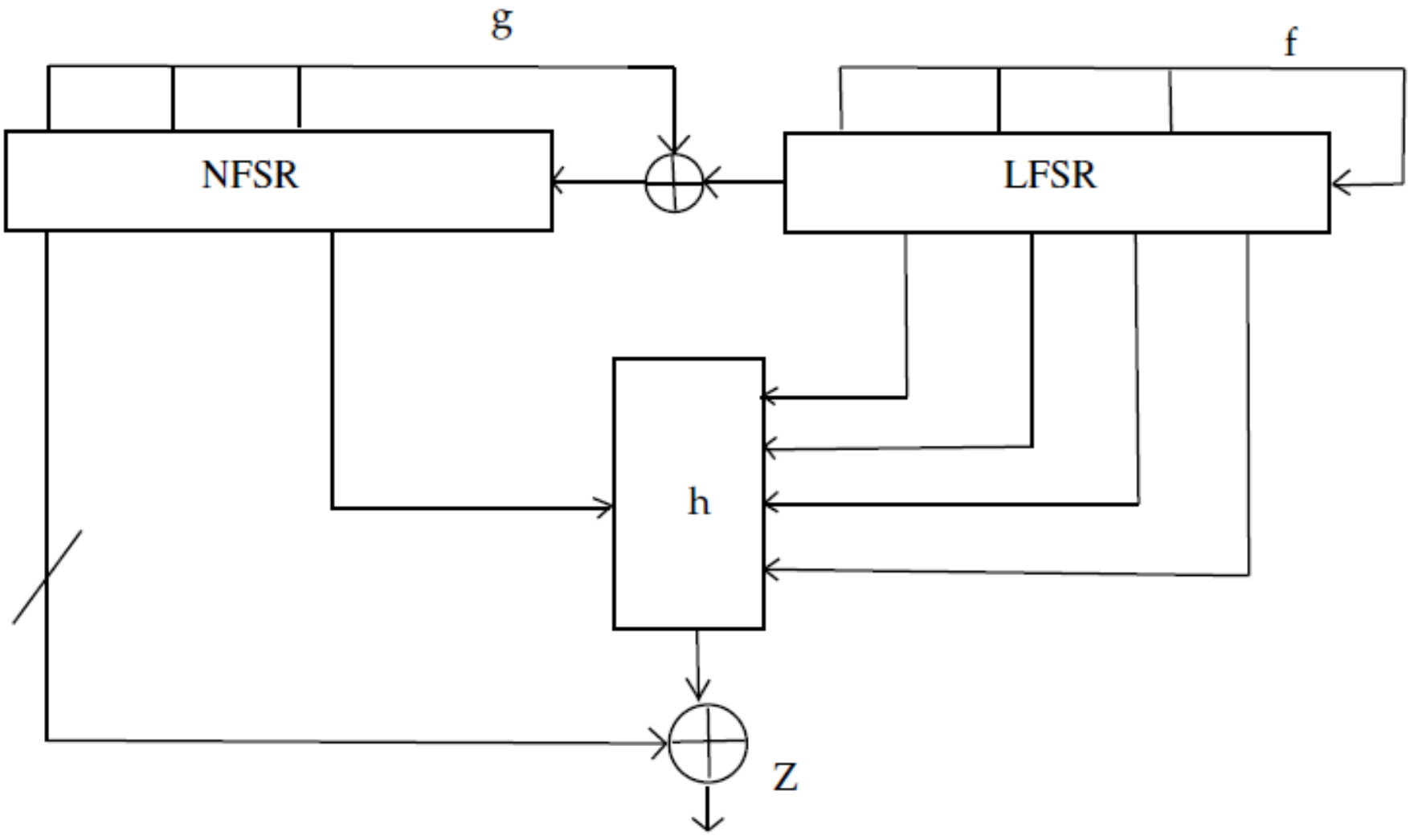


Figure: Structure of Grain v1

```
 // Grain v1

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

int s[80], b[80];

int myrand(){
    if (drand48() < 0.5)
return 0;
else
return 1;
}


// s LFSR: IV, b NFSR: key
```

```
int h(int x0, int x1, int x2, int x3, int x4){
 return(x1 ^ x4 ^ (x0 & x3) ^ (x2 & x3 )^ (x3 & x4) ^
 (x0 & x1 & x2) ^(x0 & x2 & x3) ^(x0 & x2 & x4) ^
 (x1 & x2 & x4) ^(x2 & x3 & x4));


}

void keyload(){

int i;



for(i = 0; i < 80; i++) b[i] = myrand();
for(i = 0; i < 64; i++) s[i] = myrand();
for(i = 64; i < 80; i++) s[i] = 1;
```

```
}

void ksa(){

int i;
int t1, t2, z;

t1 = s[62] ^ s[51] ^ s[38] ^ s[23] ^ s[13] ^ s[0];

t2 = s[0] ^ b[62] ^ b[60] ^ b[52] ^ b[45] ^ b[37] ^ b[33]

  ^ b[28] ^ b[21] ^ b[14] ^ b[9] ^ b[0] ^

 (b[63] & b[60]) ^ (b[37] & b[33]) ^ (b[15] & b[9]) ^

 (b[60] & b[52] & b[45]) ^ (b[33] & b[28] & b[21])

 ^ (b[63] & b[45] & b[28] & b[9]) ^
```

```
 (b[60] & b[52] & b[37] & b[33])

  ^ (b[63] & b[60] & b[21] & b[15])

 ^ (b[63] & b[60] & b[52] & b[45] & b[37])

 ^ (b[33] & b[28] & b[21] & b[15] & b[9]) ^

 (b[52] & b[45] & b[37] & b[33] & b[28] & b[21]);

z = b[1] ^ b[2] ^ b[4] ^ b[10] ^ b[31] ^ b[43] ^ b[56]

 ^   h(s[3], s[25], s[46], s[64], b[63]);
for (i = 0; i < 79; i++) s[i] = s[i+1]; s[79] = t1^z;
for (i = 0; i < 79; i++) b[i] = b[i+1]; b[79] = t2^z;
}

int prga(){
```

```
int i;
int t1, t2, z;

t1 = s[62] ^ s[51] ^ s[38] ^ s[23] ^ s[13] ^ s[0];

t2 = s[0] ^ b[62] ^ b[60] ^ b[52] ^ b[45] ^ b[37] ^ b[33]

  ^ b[28] ^ b[21] ^ b[14] ^ b[9] ^ b[0] ^

 (b[63] & b[60]) ^ (b[37] & b[33]) ^ (b[15] & b[9]) ^

 (b[60] & b[52] & b[45]) ^ (b[33] & b[28] & b[21])

 ^ (b[63] & b[45] & b[28] & b[9]) ^

 (b[60] & b[52] & b[37] & b[33])

  ^ (b[63] & b[60] & b[21] & b[15])
```

```
 ^ (b[63] & b[60] & b[52] & b[45] & b[37])

 ^ (b[33] & b[28] & b[21] & b[15] & b[9]) ^

 (b[52] & b[45] & b[37] & b[33] & b[28] & b[21]);

z = b[1] ^ b[2] ^ b[4] ^ b[10] ^ b[31] ^ b[43] ^ b[56]

 ^   h(s[3],s[25],s[46],s[64],b[63]);
for (i = 0; i < 79; i++) s[i] = s[i+1]; s[79] = t1;
for (i = 0; i < 79; i++) b[i] = b[i+1]; b[79] = t2;

return z;
}

int main(){

int i;
```

```
    srand48(time(NULL));


    keyload();
    for (i = 0; i < 160; i++) ksa();
    for (i = 0; i < 100; i++)
       printf("%d  ", prga());


}
```

Observation

$$h = x_1 + x_4 + x_0x_3 + x_2x_3 + x_3x_4x_0x_1x_2 + x_0x_2x_3 + x_0x_2x_4 + x_1x_2x_4 + x_2x_3x_4$$

$$h(x_0, 0, x_2, 0, 0) = ?$$

Observation

$$h = x_1 + x_4 + x_0x_3 + x_2x_3 + x_3x_4x_0x_1x_2 + x_0x_2x_3 + x_0x_2x_4 + x_1x_2x_4 + x_2x_3x_4$$

$h(x_0, 0, x_2, 0, 0) = ?$

ZERO

Observation

$$h = x_1+x_4+x_0x_3+x_2x_3+x_3x_4x_0x_1x_2+x_0x_2x_3+x_0x_2x_4+x_1x_2x_4+x_2x_3x_4$$

$h(x_0, 0, x_2, 0, 0) = ?$

ZERO

State Recovery on Grain v1: Mihaljevic et al.

Grain Family

	Grain v1	Grain-128	Grain-128a
n	80	128	128
m	64	96	96
Pad	FFFF	FFFFFFFF	FFFFFFFE
$f(\cdot)$	$y_{t+62} \oplus y_{t+51} \oplus y_{t+38} \oplus y_{t+23} \oplus y_{t+13} \oplus y_t$	$y_{t+96} \oplus y_{t+81} \oplus y_{t+70} \oplus y_{t+38} \oplus y_{t+7} \oplus y_t$	$y_{t+96} \oplus y_{t+81} \oplus y_{t+70} \oplus y_{t+38} \oplus y_{t+7} \oplus y_t$
$g(\cdot)$	$x_{t+62} \oplus x_{t+60} \oplus x_{t+52} \oplus x_{t+45} \oplus x_{t+37} \oplus x_{t+33} \oplus x_{t+28} \oplus x_{t+21} \oplus x_{t+14} \oplus x_{t+9} \oplus x_t \oplus x_{t+63}x_{t+60} \oplus x_{t+37}x_{t+33} \oplus x_{t+15}x_{t+9} \oplus x_{t+60}x_{t+52}x_{t+45} \oplus x_{t+33} \oplus x_{t+28}x_{t+21} \oplus x_{t+63}x_{t+60} \oplus x_{t+21}x_{t+15} \oplus x_{t+63}x_{t+60} \oplus x_{t+52}x_{t+45}x_{t+37} \oplus x_{t+33} \oplus x_{t+28}x_{t+21}x_{t+15}x_{t+9} \oplus x_{t+52}x_{t+45}x_{t+37}x_{t+33} \oplus x_{t+28}x_{t+21}$	$y_t \oplus x_t \oplus x_{t+26} \oplus x_{t+56} \oplus x_{t+91} \oplus x_{t+96} \oplus x_{t+3}x_{t+67} \oplus x_{t+11}x_{t+13} \oplus x_{t+17}x_{t+18} \oplus x_{t+27}x_{t+59} \oplus x_{t+40}x_{t+48} \oplus x_{t+61} \oplus x_{t+65} \oplus x_{t+68}x_{t+84}$	$y_t \oplus x_t \oplus x_{t+26} \oplus x_{t+56} \oplus x_{t+91} \oplus x_{t+96} \oplus x_{t+3}x_{t+67} \oplus x_{t+11}x_{t+13} \oplus x_{t+17}x_{t+18} \oplus x_{t+27}x_{t+59} \oplus x_{t+40}x_{t+48} \oplus x_{t+61} \oplus x_{t+65} \oplus x_{t+68}x_{t+84} \oplus x_{t+88}x_{t+92}x_{t+93}x_{t+95} \oplus x_{t+22}x_{t+24}x_{t+25} \oplus x_{t+70}x_{t+78}x_{t+82}$
$h(\cdot)$	$y_{t+3}y_{t+25}y_{t+46} \oplus y_{t+3} y_{t+46}y_{t+64} \oplus y_{t+3}y_{t+46} x_{t+63} \oplus y_{t+25}y_{t+46}x_{t+63} \oplus y_{t+46}y_{t+64}x_{t+63} \oplus y_{t+3} y_{t+64} \oplus y_{t+46}y_{t+64} \oplus y_{t+64} x_{t+63} \oplus y_{t+25} \oplus x_{t+63}$	$x_{t+12}x_{t+95}y_{t+95} \oplus x_{t+12} y_{t+8} \oplus y_{t+13}y_{t+20} \oplus x_{t+95} y_{t+42} \oplus y_{t+60}y_{t+79}$	$x_{t+12}x_{t+95}y_{t+94} \oplus x_{t+12} y_{t+8} \oplus y_{t+13}y_{t+20} \oplus x_{t+95} y_{t+42} \oplus y_{t+60}y_{t+79}$
z_t	$x_{t+1} \oplus x_{t+2} \oplus x_{t+4} \oplus x_{t+10} \oplus x_{t+31} \oplus x_{t+43} x_{t+56} \oplus h$	$x_{t+2} \oplus x_{t+15} \oplus x_{t+36} \oplus x_{t+45} \oplus x_{t+64} \oplus x_{t+73} \oplus x_{t+89} \oplus y_{t+93} \oplus h$	$x_{t+2} \oplus x_{t+15} \oplus x_{t+36} \oplus x_{t+45} \oplus x_{t+64} \oplus x_{t+73} \oplus x_{t+89} \oplus y_{t+93} \oplus h$

MAC Generation Algorithm (MGA) in Grain-128a

- Cipher picks every second keystream bit as output of the cipher after skipping the first 64 pre-output bits of PRGA rounds
- Message of length L defined by the bits $m_0, \ldots, m_{L-1}$. Set $m_L = 1$ as padding.
- To provide authentication, two registers, called accumulator and shift register of size 32 bits each, are used.

MAC Generation Algorithm (MGA) in Grain-128a

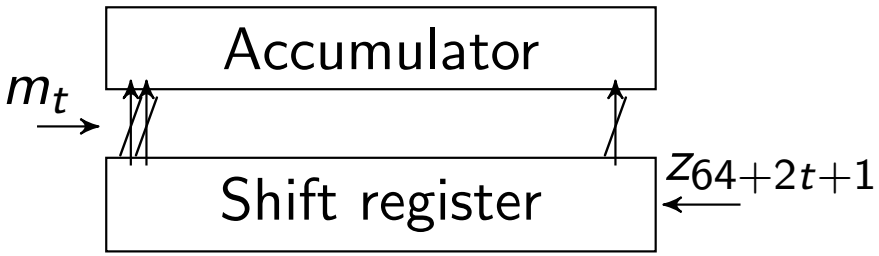


Figure: Authentication

- The content of accumulator and shift register at time t is denoted by $a_t^0, \ldots, a_t^{31}$ and $r_t, \ldots, r_{t+31}$ respectively.

MAC Generation Algorithm (MGA) in Grain-128a

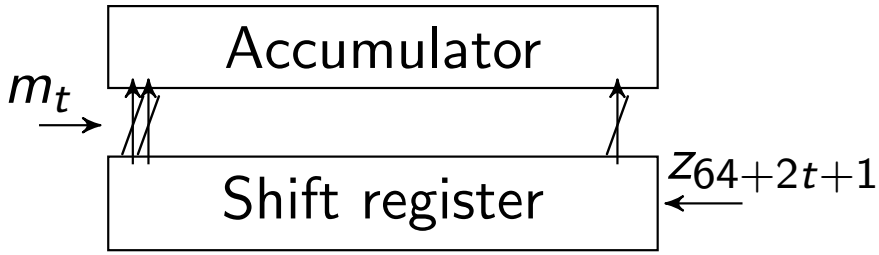


Figure: Authentication

- The accumulator is initialized through $a_0^j = z_j, -64 \leq j \leq -33$ and the shift register is initialized through $r_j = z_j, -32 \leq j \leq -1$.
- The shift register is updated as $r_{t+32} = z_{2t+1}$.
- The accumulator is updated as $a_{t+1}^j = a_t^j + m_t r_{t+j}$ for $0 \leq j \leq 31$ and $0 \leq t \leq L$.
- The final content of accumulator, $a_{L+1}^0, \ldots, a_{L+1}^{31}$ is used for authentication.

Trivium

Trivium Parameters

Key Size	80 bits
IV Size	80 bits
Internal State	288 bits

Key Loading Algorithm (KLA).

The initialization routine works as following:

$$(s_1, s_2, \ldots, s_{93}) \leftarrow (K_1, \ldots, K_{80}, 0, \ldots, 0)$$
$$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (IV_1, \ldots, IV_{80}, 0, \ldots, 0)$$
$$(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$$

We denote the state after KLA as $S^{(0)}$.

$S^{(0)} = [K_1, \ldots, K_{80}, 0, \ldots, 0, IV_1, \ldots, IV_{80},$
$0, \ldots, 0, 0, \ldots, 0, 1, 1, 1]$.

Key Scheduling Algorithm (KSA).

After the KLA, the cipher is clocked 4×288 times without producing any key-stream bits. The Key Scheduling Algorithm (KSA) is as follows:

for $i = 1$ to 4×288 do

$t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$

$t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$

$t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$

$(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1 \ldots, s_{92})$

$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94} \ldots, s_{176})$

$(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_2, s_{178} \ldots, s_{287})$

Key-stream Generation.

for $i = 1$ to N do

$t_1 \leftarrow s_{66} + s_{93}$

$t_2 \leftarrow s_{162} + s_{177}$

$t_3 \leftarrow s_{243} + s_{288}$

$z_i \leftarrow t_1 + t_2 + t_3$;

$t_1 \leftarrow t_1 + s_{91} \cdot s_{92} + s_{171}$

$t_2 \leftarrow t_2 + s_{175} \cdot s_{176} + s_{264}$

$t_3 \leftarrow t_3 + s_{286} \cdot s_{287} + s_{69}$;

$(s_1, s_2, \ldots, s_{93}) \leftarrow (t_3, s_1 \ldots, s_{92})$;

$(s_{94}, s_{95}, \ldots, s_{177}) \leftarrow (t_1, s_{94} \ldots, s_{176})$;

$(s_{178}, s_{179}, \ldots, s_{288}) \leftarrow (t_2, s_{178} \ldots, s_{287})$;

end for

CAESAR ciphers

- CAESAR - the Competition for Authenticated Encryption: Security, Applicability, and Robustness
- Trivia-SC
- Acorn

Cryptanalysis of Grain v1

Distinguisher on Grain v1

- Knellwolf et al. in Asiacrypt 2010: New distinguisher on Grain v1.
- Variant of cube tester

Cube tester

- $f(x_1, \ldots, x_n) = t_I p(\ldots) + q(x_1, \ldots, x_n)$, where p is polynomial no variable common with t_I and no monomial of q contains t_I.
- p: superpoly
- $f(k_1, k_2, v_1, v_2) = k_1 + k_1 k_2 v_1 + k_1 k_2 v_2 + v_1$

Cube tester

- $f(x_1, \ldots, x_n) = t_I p(\ldots) + q(x_1, \ldots, x_n)$, where p is polynomial no variable common with t_I and no monomial of q contains t_I.
- p: superpoly
- $f(k_1, k_2, v_1, v_2) = k_1 + k_1 k_2 v_1 + k_1 k_2 v_2 + v_1$
- $f(k_1, k_2, 0, 0) + f(k_1, k_2, 1, 0)$
  $+ f(k_1, k_2, 0, 1) + f(k_1, k_2, 1, 1) = 0$

- 80 bit key $k_0, \ldots, k_{79}$ and 64 bit IV $v_0, \ldots, v_{63}$.
- Grain v1 is first intialised with $X_0 = [k_0, \ldots, k_{79}]$ and $Y_0 = [v_0, \ldots, v_{63}, \overbrace{1, \ldots, 1}^{16}]$.
- Here X_0 corresponds to NFSR and Y_0 corresponds to LFSR.

The idea

- Next start with NFSR $X_0' = [k_0, \ldots, k_{79}]$ but different LFSR $Y_0' = [v_0, \ldots, 1 \oplus v37, v_{63}, \overbrace{1, \ldots, 1}^{16}]$.
- Thus two states S_0 and S_0' initialized by (X_0, Y_0) and (X_0', Y_0') different only at one position.
- But when more and more KSA rounds are completed, more and more positions of the states will be differ.
- Conditions of z_{12}, z_{34} and z_{40} of KSA

The idea

- The idea is to delay the diffusion of the differential.
- The conditions may be classified in to two types:
  - **Type 1:** Conditions only on IV
  - **Type 2:** Conditions on both Key and IV.

Attack Idea

z_t and z'_t: Output bit produced in the t-th KSA round when states are loaded by (X_0, Y_0) and (X'_0, Y'_0).

The attack idea is as follows:

1. For $i = 0, \ldots, 11$, it is not difficult to show that $z_i = z'_i$.
2. When $i = 12$, $z_i \oplus z'_i = v_{15} v_{58} \oplus v_{58} k_{75} \oplus 1$.

Attack Idea

z_t and z'_t: Output bit produced in the t-th KSA round when states are loaded by (X_0, Y_0) and (X'_0, Y'_0).

The attack idea is as follows:

1. For $i = 0, \ldots, 11$, it is not difficult to show that $z_i = z'_i$.
2. When $i = 12$, $z_i \oplus z'_i = v_{15} v_{58} \oplus v_{58} k_{75} \oplus 1$.
3. To make $v_{15} v_{58} \oplus v_{58} k_{75} \oplus 1 = 0$, set $v_{58} = 1$ and $v_{15} = 1 \oplus k_{75}$.
4. Thus we have one Type 1 condition $v_{58} = 1$ and one Type 2 condition $C_1 : v_{15} = 1 \oplus k_{75}$.
5. For $i = 13, \ldots, 29$, z_i will be always equal to z'_i.

6. When $i = 30$, z_{30} will be always different from z'_{30}.
7. z_i will be always equal to z'_i for $i = 31$ and 32.
8. When $i = 34$, $z_{34} \oplus z'_{34}$ will be an algebraic expression on Key and IV.
9. If attacker sets 13 Type 1 conditions
   $v_0 = 0, v_1 = 0, v_3 = 0, v_4 = 0, v_5 = 0, v_{21} = 0, v_{25} = 0, v_{26} = 0, v_{27} = 0, v_{43} = 0, v_{46} = 0, v_{47} = 0, v_{48} = 0$ and two Type 2 conditions

$$\begin{aligned} C_2 : v_{13} &= v_{23} \oplus v_{38} \oplus v_{51} \oplus v_{62} \oplus k_1 \oplus k_2 \oplus k_4 \oplus k_{10} \\ &\quad \oplus k_{31} \oplus k_{43} \oplus k_{56}, \\ C_3 : v_2 &= v_{18} \oplus v_{31} \oplus v_{40} \oplus v_{41} \oplus v_{53} \oplus v_{56} \oplus f_1(K), \end{aligned}$$

   where $f_1(K)$ is a polynomial over Key of degree 7 and 39 monomials, $z_{34} = z'_{34}$.

Attack idea

10. $z_i = z'_i$ for $35 \leq i \leq 39$.
11. When $i = 40$, again $z_{40} \oplus z'_{40}$ will be an algebraic expression on Key and IV.
12. However if attacker sets 13 Type 1 conditions $v_8 = 0, v_9 = 0, v_{10} = 0, v_{19} = 0, v_{28} = 0, v_{29} = 0, v_{31} = 0, v_{44} = 0, v_{49} = 0, v_{51} = 0, v_{52} = 0, v_{53} = 0, v_{57} = 0$ and two Type 2 conditions

$$C_4 : v_6 = k_7 \oplus k_8 \oplus k_{10} \oplus k_{16} \oplus k_{37} \oplus k_{49} \oplus k_{62} \oplus 1,$$
$$C_5 : v_7 = v_{20} \oplus v_{23} \oplus v_{32} \oplus v_{45} \oplus f_2(K),$$

where $f_2(K)$ is a polynomial over Key of degree 15 and 2365 monomials, $z_{40} = z'_{40}$.

Attack Idea

- Total of 27 Type 1 conditions and 5 Type 2 conditions $C_1, \ldots, C_5$. Hence IV space is reduced to $\{0,1\}^{64-27} = \{0,1\}^{37}$.
- Corresponding to 5 Type 2 conditions, attacker divides this space into $2^5 = 32$ partitions.
- That is since there are 5 expressions on unknown Key, attacker chooses all 32 options. Among these 32 options, one must be correct.

Attack idea

- Knellwolf et al. observed experimentally for the correct guess on 5 key expressions, $z_{97} \oplus z'_{97}$ is more likely to be zero.
- This gives a distinguisher on Grain v1 for reduced round.
- Five Type 2 conditions are crucial for Key recovery.

Sage: www.sagemath.org

Online Sage Notebook | Screen Shots | FAQ | Wiki | Blog | Publications | News | Why | Thanks | JSAGE | For Devs | Mirrors | Search

SAGE: Open Source Mathematics Software

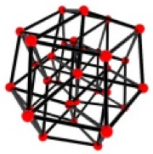
Creating a viable free open source alternative to Magma, Maple, Mathematica, and Matlab

Download Documentation
Tutorial Support

SAGE 2.10.1 has been released (February 02, 2008)!

SAGE Days 7 at IPAM (UCLA) was a huge success!

General and Advanced Pure and Applied Mathematics
Use SAGE for studying a huge range of mathematics, including algebra, calculus, elementary to very advanced number theory, cryptography, numerical computation, commutative algebra, group theory, combinatorics, graph theory, and exact linear algebra.



Use an Open Source Alternative
By using SAGE you help to support a viable open source alternative to Magma, Maple, Mathematica, and MATLAB. SAGE includes many high-quality open source math packages.

Use Most Mathematics Software from Within SAGE
SAGE makes it easy for you to use most mathematics software together. SAGE includes interfaces to Magma, Maple, Mathematica, MATLAB, and MuPAD, and the free programs Axiom, GAP, GP/PARI, Macaulay2, Maxima, Octave, and Singular.

```
%magma
E := EllipticCurve([1,2,3,4,5]);
Rank(E);
   1
octave('rand(3)')
      0.861146 0.323013 0.0147563
      0.482702 0.888791 0.55302
      0.0686355 0.202225 0.22652
```

```
V=BooleanPolynomialRing(144,['k%d'%(i) for i in range(80)]
+['v%d'%(i) for i in range(64)])
V.inject_variables()
Z=[]

SS=[k0, k1, k2, k3, k4, k5, k6, k7, k8, k9, k10, k11, k12,
k13,k14, k15, k16, k17, k18, k19, k20, k21, k22, k23, k24,
k25, k26, k27,k28, k29, k30, k31, k32, k33, k34, k35, k36,
k37, k38, k39, k40, k41,k42, k43, k44, k45, k46, k47, k48,
k49, k50, k51, k52, k53, k54, k55,k56, k57, k58, k59, k60,
k61, k62, k63, k64, k65, k66, k67, k68, k69,k70, k71, k72,
k73, k74, k75, k76, k77, k78, k79]

TT=[v0, v1, v2, v3, v4,v5, v6, v7, v8, v9, v10, v11, v12,
v13, v14, v15, v16, v17, v18, v19,v20, v21, v22, v23, v24,
v25, v26, v27, v28, v29, v30, v31, v32, v33,v34, v35, v36,
v37, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47,v48,
v49, v50, v51, v52, v53, v54, v55, v56, v57, v58, v59, v60,
v61,v62, v63, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
```

```
1, 1, 1, 1]



for i in range(35):

        tl=TT[0]+TT[13]+TT[23]+TT[38]+TT[51]+TT[62]

        tn=  TT[0]+SS[62]+SS[60]+SS[52]+SS[45]+SS[37]
        +SS[33]+SS[28]+SS[21]+SS[14]+SS[9]+SS[0]+SS[63]
        *SS[60]+SS[37]*SS[33]+SS[15]*SS[9]+SS[60]*SS[52]
        *SS[45]+SS[33]*SS[28]*SS[21]+SS[63]*SS[45]*SS[28]
        *SS[9]+SS[60]*SS[52]*SS[37]*SS[33]+SS[63]*SS[60]
        *SS[21]*SS[15]+SS[63]*SS[60]*SS[52]*SS[45]*SS[37]
        +SS[33]*SS[28]*SS[21]*SS[15]*SS[9]+SS[52]
        *SS[45]*SS[37]*SS[33]*SS[28]*SS[21]
```

```
op=SS[1]+SS[2]+SS[4]+SS[10]+SS[31]+SS[43]+SS[56]+
TT[25]+SS[63]+TT[3]*TT[64]+TT[46]*TT[64]+TT[64]*
SS[63]+TT[3]*TT[25]*TT[46]+ TT[3]*TT[46]*TT[64]+
TT[3]*TT[46]*SS[63]+ TT[25]*TT[46]*SS[63]+TT[46]
*TT[64]*SS[63]



Z.append(op)

tl=tl+op
tn=tn+op

for j in range(79):
   SS[j]=SS[j+1]
for j in range(79):
      TT[j]=TT[j+1]
SS[79]=tn
TT[79]=tl
```

```
        #print tl



 Z1=[]

SS=[k0, k1, k2, k3, k4, k5, k6, k7, k8, k9, k10, k11, k12,
k13,k14, k15, k16, k17, k18, k19, k20, k21, k22, k23, k24,
k25, k26, k27,k28, k29, k30, k31, k32, k33, k34, k35, k36,
k37, k38, k39, k40, k41,k42, k43, k44, k45, k46, k47, k48,
k49, k50, k51, k52, k53, k54, k55,k56, k57, k58, k59, k60,
k61, k62, k63, k64, k65, k66, k67, k68, k69,k70, k71, k72,
k73, k74, k75, k76, k77, k78, k79]

TT=[v0, v1, v2, v3, v4,v5, v6, v7, v8, v9, v10, v11, v12,
v13, v14, v15, v16, v17, v18, v19,v20, v21, v22, v23, v24,
v25, v26, v27, v28, v29, v30, v31, v32, v33,v34, v35, v36,
v37+1, v38, v39, v40, v41, v42, v43, v44, v45, v46, v47,v48
```

```
v49, v50, v51, v52, v53, v54, v55, v56, v57, v58, v59, v60,
v61,v62, v63, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1]
```

```
for i in range(35):

        tl=TT[0]+TT[13]+TT[23]+TT[38]+TT[51]+TT[62]

        tn=  TT[0]+SS[62]+SS[60]+SS[52]+SS[45]+SS[37]
        +SS[33]+SS[28]+SS[21]+SS[14]+SS[9]+SS[0]+SS[63]
        *SS[60]+SS[37]*SS[33]+SS[15]*SS[9]+SS[60]*SS[52]
        *SS[45]+SS[33]*SS[28]*SS[21]+SS[63]*SS[45]*SS[28]
        *SS[9]+SS[60]*SS[52]*SS[37]*SS[33]+SS[63]*SS[60]
        *SS[21]*SS[15]+SS[63]*SS[60]*SS[52]*SS[45]*SS[37]
        +SS[33]*SS[28]*SS[21]*SS[15]*SS[9]+SS[52]
```

```
*SS[45]*SS[37]*SS[33]*SS[28]*SS[21]

op=SS[1]+SS[2]+SS[4]+SS[10]+SS[31]+SS[43]+SS[56]+
TT[25]+SS[63]+TT[3]*TT[64]+TT[46]*TT[64]+TT[64]*
SS[63]+TT[3]*TT[25]*TT[46]+ TT[3]*TT[46]*TT[64]+
TT[3]*TT[46]*SS[63]+ TT[25]*TT[46]*SS[63]+TT[46]
*TT[64]*SS[63]



Z1.append(op)

tl=tl+op
tn=tn+op

for j in range(79):
SS[j]=SS[j+1]
for j in range(79):
   TT[j]=TT[j+1]
```

```
        SS[79]=tn
        TT[79]=tl
        #print tl



 for i in range(32):
    print Z[i]+Z1[i],i
```

```
 0 0
0 1
0 2
0 3
0 4
0 5
0 6
0 7
0 8
```

```
0 9
0 10
0 11
k75*v58 + v15*v58 + 1 12
0 13
0 14
0 15
0 16
0 17
0 18
0 19
0 20
0 21
0 22
0 23
0 24
0 25
0 26
0 27
```

```
k75*v58 + v15*v58 + 1 28
k75*v32*v58 + k75*v54*v58 + k75*v58 +
v15*v32*v58 + v15*v54*v58 +
v15*v58 + v32 + v54 + 1 29
1 30
```

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

int s[80], b[80], s1[80], b1[80];

int myrand(){
    if (drand48() < 0.5)
return 0;
else
return 1;
}


// s LFSR: IV, b NFSR: key
```

```
int h(int x0, int x1, int x2, int x3, int x4){
 return(x1 ^ x4 ^ (x0 & x3) ^ (x2 & x3 )^
 (x3 & x4) ^ (x0 & x1 & x2) ^(x0 & x2 & x3)
 ^(x0 & x2 & x4) ^(x1 & x2 & x4) ^(x2 & x3 & x4));


}



void keyload(){

int i;



for(i = 0; i < 80; i++) b[i] = myrand();
```

```
}

void ivload(){
int i;



for(i = 0; i < 64; i++) s[i] = myrand();
for(i = 64; i < 80; i++) s[i] = 1;


}

void conditions(){

s[58]=1;
s[0]=s[1]=s[3]=s[4]=s[5]=s[21]=s[25]=s[26]=
s[27]=s[43]=s[46]=s[47]=s[48]=0;
```

```
s[8]=s[9]=s[10]=s[19]=s[28]= s[29]=
s[31]= s[44]= s[49]= s[51]=s[52]= s[53]=
s[57]=0;

}

void statesave(){

          int i;



for(i = 0; i < 80; i++) b1[i] = b[i];
for(i = 0; i < 64; i++) s1[i] = s[i];
for(i = 64; i < 80; i++) s[i] = 1;

}

void stateback(){
```

```
 int i;



for(i = 0; i < 80; i++) b[i] = b1[i];
for(i = 0; i < 64; i++) s[i] = s1[i];
for(i = 64; i < 80; i++) s[i] = 1;

s[37]=s[37]^1;

}

int ksa(){

int i;
int t1, t2, z;

t1 = s[62] ^ s[51] ^ s[38] ^ s[23] ^ s[13] ^ s[0];
```

```
t2 = s[0] ^ b[62] ^ b[60] ^ b[52] ^ b[45] ^ b[37]
^ b[33] ^ b[28] ^ b[21] ^ b[14] ^ b[9] ^ b[0] ^
(b[63] & b[60]) ^ (b[37] & b[33]) ^ (b[15] & b[9])
^ (b[60] & b[52] & b[45]) ^ (b[33] & b[28] & b[21])
^ (b[63] & b[45] & b[28] & b[9]) ^ (b[60] & b[52]
& b[37] & b[33]) ^ (b[63] & b[60] & b[21] & b[15])
^ (b[63] & b[60] & b[52] & b[45] & b[37]) ^ (b[33]
& b[28] & b[21] & b[15] & b[9]) ^ (b[52] & b[45]
& b[37] & b[33] & b[28] & b[21]);

z = b[1] ^ b[2] ^ b[4] ^ b[10] ^ b[31] ^ b[43]
^ b[56] ^ h(s[3],s[25],s[46],s[64],b[63]);

for (i = 0; i < 79; i++) s[i] = s[i+1]; s[79] = t1^z;
for (i = 0; i < 79; i++) b[i] = b[i+1]; b[79] = t2^z;

return z;
}
```

```
int prga(){

int i;
int t1, t2, z;

t1 = s[62] ^ s[51] ^ s[38] ^ s[23] ^ s[13] ^ s[0];

t2 = s[0] ^ b[62] ^ b[60] ^ b[52] ^ b[45] ^ b[37]
^ b[33] ^ b[28] ^ b[21] ^ b[14] ^ b[9] ^ b[0] ^
(b[63] & b[60]) ^ (b[37] & b[33]) ^ (b[15] & b[9])
 ^ (b[60] & b[52] & b[45]) ^ (b[33] & b[28] & b[21])
 ^ (b[63] & b[45] & b[28] & b[9]) ^ (b[60] & b[52]
 & b[37] & b[33]) ^ (b[63] & b[60] & b[21] & b[15])
 ^ (b[63] & b[60] & b[52] & b[45] & b[37]) ^ (b[33]
 & b[28] & b[21] & b[15] & b[9]) ^ (b[52] & b[45]
 & b[37] & b[33] & b[28] & b[21]);

z = b[1] ^ b[2] ^ b[4] ^ b[10] ^ b[31] ^ b[43] ^
```

```
b[56] ^  h(s[3],s[25],s[46],s[64],b[63]);

for (i = 0; i < 79; i++) s[i] = s[i+1]; s[79] = t1;
for (i = 0; i < 79; i++) b[i] = b[i+1]; b[79] = t2;

return z;
}

int main(){

int i, z[100], z1[100], A[100], j;
int ksaz[160], ksaz1[160],c=0;

for(i=0;i<100;i++)
  A[i]=0;

    srand48(time(NULL));
```

```
keyload();

for(j=0;j<100000000;j++){

 ivload();
 conditions();
 statesave();

 for (i = 0; i < 97; i++)
       ksaz[i] = ksa();

 for (i = 0; i < 10; i++)
   z[i]=  prga();
 stateback();

for (i = 0; i < 97; i++)
     ksaz1[i]=ksa();
 for (i = 0; i < 10; i++)
   z1[i]=  prga();
```

```
    if((ksaz[12]==ksaz1[12]) && (ksaz[34]==ksaz1[34])
    && (ksaz[40]==ksaz1[40])){
        c++;

        for(i=0;i<10;i++)
         if(z[i]==z1[i])
            A[i]++;
    }
    if(c>0 && c%10000==0){
         for(i=0;i<10;i++)
             printf("%d  %lf\n", 97+i, (double)A[i]/c);
         printf("--------------------------\n");
             }
    }


}
```

Banik's result

- Banik shows a distinguishing attack for 105 round.
- Instead of 37-th bit of IV, he chooses 61-bit of IV.
- So initial states are $S_0 = (X_0, Y_0)$ and $S_0' = (X_0', Y_0')$ where

$$X_0 = X_0' = [k_0, \ldots, k_{79}],\ Y_0 = [v_0, \ldots, v_{63}, \overbrace{1, \ldots, 1}^{16}],$$

$$Y_0' = [v_0, \ldots, 1 \oplus v61, v_{62}, v_{63}, \overbrace{1, \ldots, 1}^{16}].$$

Banik's idea: 105 round

Differential on v_{61}.

25 Type 1 conditions

$v_{18} = 1, v_2 = v_5 = v_8 = v_{11} = v_{22} = v_{25} = v_{27} = v_{28} = v_{30} = v_{33} = v_{39} = v_{40} = v_{42} = v_{44} = v_{45} = v_{48} = v_{49} = v_{50} = v_{51} = v_{52} = v_{53} = v_{54} = v_{55} = 0$. He also sets 6 Type 2 Conditions:

$$
\begin{aligned}
C_1 : v_{15} &= f_1(K), \\
C_2 : v_3 &= v_1 \oplus v_4 \oplus v_6 \oplus v_{16} \oplus v_{19} \oplus v_{41} \oplus f_2(K), \\
C_3 : v_{43} &= v_{56} \oplus f_3(K), \\
C_4 : v_{57} &= v_4 \oplus v_6 \oplus v_7 \oplus v_9 \oplus v_{19} \oplus v_{31} \oplus f_4(K), \\
C_5 : v_{46} &= v_{21} \oplus v_{31} \oplus v_{59} \oplus f_5(K), \\
C_6 : v_{60} &= v_1 \oplus v_4 \oplus v_7 \oplus v_9 \oplus v_{10} \oplus v_{12} \oplus v_{26} \oplus v_{34} \oplus f_6(K),
\end{aligned}
$$

where $f_i(K)$ are polynomials over the Key for $1 \leq i \leq 6$.

$$P(z_{105} \oplus z'_{105} = 0) \approx \frac{1}{2} + 0.0002.$$

Attack up to 106 rounds: Sarkar 2015

Differential on v_{62}

1. For $i = 0, \ldots, 15$, $z_i = z'_i$.
2. When $i = 16$, set $v_{19} = v_{41} = 1, v_{46} = 0$ and $v_0 = k_1 \oplus k_2 \oplus k_4 \oplus k_{10} \oplus k_{31} \oplus k_{43} \oplus k_{56} \oplus v_3 \oplus v_{13} \oplus v_{23} \oplus v_{25} \oplus v_{38} \oplus v_{51}$.
3. For $i = 17, \ldots, 26$, z_i will be always equal to z'_i.
4. When $i = 27$, z_{27} will be always different from z'_{27}.

5. z_i will be always equal to z_i' for $i = 28, \ldots, 33$.
6. When $i = 34$, $z_{34} \oplus z_{34}'$ will be an algebraic expression on Key and IV.
   17 Type 1 conditions
   $v_2 = v_{15} \oplus v_{18} \oplus v_{25} \oplus v_{31} \oplus v_{40} \oplus v_{53} \oplus v_{56} \oplus v_{59}, v_{63} = 0, v_{14} = v_{24} \oplus v_{39} \oplus v_{52}, v_{13} = v_{23} \oplus v_{38} \oplus v_{51}, v_{17} = v_{42}, v_{43} = 0, v_{47} = 0, v_{38} = 0, v_4 = 0, v_1 = 0, v_5 = 0, v_{20} = 0, v_{21} = 0, v_{26} = 0, v_{27} = 0, v_{37} = 0, v_{48} = 0$ and one Type 2 condition

$$C_2 : v_{59} \quad = \quad f_1(K),$$

   where $f_1(K)$ is a polynomial over Key of degree 16 and 9108 monomials, $z_{34} = z_{34}'$.

7. $z_i = z'_i$ for $i = 35, 36$.
8. When $i = 37$, again $z_{37} \oplus z'_{37}$ will be an algebraic expression on Key and IV. However if attacker sets 7 Type 1 conditions $v_{15} = v_{18} \oplus v_{25} \oplus v_{31} \oplus v_{53} \oplus v_{55} \oplus v_{56} \oplus v_{59}, v_{16} = v_{54}, v_{49} = 1, v_{28} = 0, v_6 = 0, v_{50} = 0, v_{23} = v_{45}$
and two Type 2 conditions

$$\begin{aligned} C_3 : v_3 &= k_4 \oplus k_5 \oplus k_7 \oplus k_{13} \oplus k_{34} \oplus k_{46} \oplus k_{59} \oplus k_{66} \\ C_4 : v_7 &= v_{29} \oplus f_2(K), \end{aligned}$$

where $f_2(K)$ is a polynomial over Key of degree 15 and 1535 monomials, $z_{37} = z'_{37}$.
9. $z_i = z'_i$ for $i = 38, 39$.
10. If we set 7 Type 1 conditions $v_{58} = v_7, v_{57} = v_{44} \oplus v_{29}, v_{51} = 0, v_{52} = 0, v_{10} = 0, v_{32} = 0, v_{53} = 0$ and 2 Type 2 conditions

$$\begin{aligned} C_5 : v_9 &= k_7 \oplus k_8 \oplus k_{10} \oplus k_{16} \oplus k_{37} \oplus k_{49} \oplus k_{62} \oplus v_{31} \\ C_6 : v_8 &= f_3(K), \end{aligned}$$

where $f_3(K)$ is a polynomial over Key of degree 15 and 1572 monomials, $z_{40} = z'_{40}$.

Possible attack up to 106 rounds

- Type 1: 34
- Type 2: 6
- IV space is reduced to $\{0,1\}^{64-34} = \{0,1\}^{30}$

Fault attack

DFA: Differential Fault Attack

Attack model:

- Load same key and IV many times
- Disturb only one bit in PRGA
- Timing of fault is known
- Location of fault is unknown

How to find the location?

Faults in LFSR: Bits 0 to 10

```
00:  0000000000000000..0000001000000000..0.000.10..00010.....0.........
01:  0000000000000000..0000001000000000..0.000.10..00010.....0.........
02:  0000000000000000..0000001000000000..0.000.10..00010.....0........
03:  .0000000000000000..0000001000000000..0.000.10..00010.....0.......
04:  0.0000000000000000..0000001000000000..0.000.10..00010.....0......
05:  00.0000000000000000..0000001000000000..0.000.10..00010.....0.....
06:  000.0000000000000000..0000001000000000..0.000.10..00010.....0....
07:  0000.0000000000000000..0000001000000000..0.000.10..00010.....0...
08:  00000.0000000000000000..0000001000000000..0.000.10..00010.....0..
09:  000000.0000000000000000..0000001000000000..0.000.10..00010.....0.
10:  0000000.0000000000000000..0000001000000000..0.000.10..00010.....0
```

Recover state

- Fault-free key-stream $z_0, \ldots, z_{\ell-1}$
- LFSR $Y_0 = [y_0, y_1, \ldots, y_{n-1}]$ and NFSR $X_0 = [x_0, x_1, \ldots, x_{n-1}]$.
- Not feasible to compute the Algebraic Normal Form (ANF) of z_{159}
- At the beginning of the t-th ($t \geq 0$) PRGA round as

$$Y_t = [y_t, y_{t+1}, \ldots, y_{t+79}], \quad X_t = [x_t, x_{t+1}, \ldots, x_{t+79}].$$

Generate equations

1. LFSR equation: $y_{t+80} = f(Y_t)$.
2. NFSR equation: $x_{t+80} = y_t \oplus g(X_t)$.
3. Key-stream equation: $z_t = \bigoplus_{i=0}^{79} b_i y_{t+i}$
   $\oplus \bigoplus_{i=0}^{79} a_i x_{t+i} \oplus h(y_t, \ldots, y_{t+79}, x_t, \ldots, x_{t+79})$.

Sarkar, Banik and Maitra: IEEE Trans. on Computers 2014

Faults in LFSR only					
Cipher	Number of faults	Amount of key-stream	Time (in sec.)		
			Minimum	Maximum	Average
Grain v1	10	160	16.48	49.23	27.40
	9	160	22.10	32.71	40.50
	8	160	18.62	92.34	48.40
Grain-128	5	256	5.21	9.43	7.10
	4	256	9.03	96.68	34.40
	3	256	24.52	361.53	163.70
Grain-128a	11	175	14.47	37.85	23.60
	10	175	26.82	253.15	52.74

Faults in NFSR only					
Cipher	Number of faults	Amount of key-stream	Time (in sec.)		
			Minimum	Maximum	Average
Grain v1	11	160	27.93	105.44	55.35
	10	160	21.14	89.50	43.64
	9	160	29.64	123.98	56.35
Grain-128	6	256	16.64	196.32	93.45
	5	256	22.87	380.01	147.70
Grain-128a	11	175	179.62	8453.14	1542.27
	10	175	175.07	8387.21	1495.54

Faults in both LFSR and NFSR					
Cipher	Number of faults	Amount of key-stream	Time (in sec.)		
			Minimum	Maximum	Average
Grain v1	11	160	54.96	1420.71	220.90
	10	160	19.17	452.30	352.20
Grain-128	6	256	6.48	14.32	10.41
	5	256	12.18	37.56	22.15
	4	256	27.63	4876.53	581.80
Grain-128a	11	175	46.45	259.34	101.10
	10	175	69.63	5144.56	1472.35

Slid pair

Slid pair

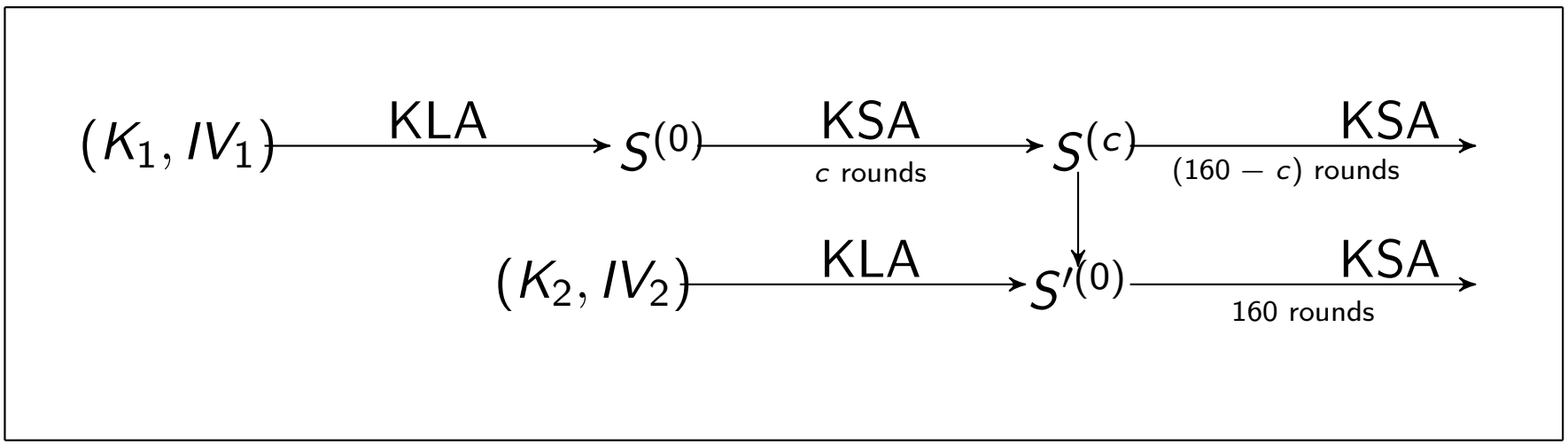


Figure: Slid attack on a cipher

Slid pairs

- Key-IV pairs in Grain v1 and Grain-128, that produce ϵ-bit shifted keystream with complexity $2^{2\epsilon}$: Cannière, Küçük and Preneel (Africacrypt 2008)
- Complexity 2^{ϵ}: Banik, Maitra and Sarkar (Space 2012)
- Both these techniques utilized the fact that the padding P used in Grain v1 and Grain-128 was symmetric
- Padding of Grain-128a: $P =$`0x ffff fffe` a set of 31 ones followed by a single zero.

Banik, Maitra, Sarkar and Turan (ACISP 2013)

32-bit shifted keystreams for Grain-128a

Pair	Key	IV	Output bits
1	9bbe 7e2b b99d 1477 0317 9f3b a1aa 8c70	5a7c 21e9 3a77 52ce ffff fffe	41d5c1f0387c 3bf64e031725
2	f32a 7bd3 9bbe 7e2b b99d 1477 0317 9f3b	032d 0fee 5a7c 21e9 3a77 52ce	0000000041d5c1f0387c 3bf64e031725

Sat solver

```
V=BooleanPolynomialRing(4,['x%d'%(i) for i in range(4)]  )
V.inject_variables()

A=[x0*x1+1, x1+x2+1, x3*x2]
tt=cputime()
I = Ideal(A)

import sage.sat.boolean_polynomials
E= sage.sat.boolean_polynomials.solve(I.gens(),n=10)
print cputime(tt)
print E

Defining x0, x1, x2, x3
0.001268
[{x3: 0, x1: 1, x2: 0, x0: 1},
{x3: 1, x1: 1, x2: 0, x0: 1}]
```

Analysis of Trivium

Table: Overview of cube attacks and cube testers on Trivium

Author(s): Type of the attack	Cube size : Round(s)
Dinur, Shamir (Eprint'08, Eurocrypt'09) : Cube attack	$12 : 672 - 685$; $23 : 735 - 747$; $29 : 767 - 774$
Aumasson, Dinur, Meier, Shamir (FSE'09): Cube tester	$24 : 772$; $30 : 790$ (Distinguisher) $24 : 842$; $27 : 885$ (Non-randomness)
Stankovski (Indocrypt'10): Cube tester	$44 : 806$ (Distinguisher) $45 : 1026$; $54 : 1078$ (Non-randomness)
Knellwolf, Meier, Naya-Plasencia (SAC'11): Cube tester	$25 : 772, 782, 789, 798$ (Distinguisher) $25 : 868$ (2^{31} key space); $25 : 953, 961$ (2^{26} key space)(Non-randomness)
Fouque, Vannet (FSE'13): Cube attack	$30 : 784$; $37 : 799$
Liu, Lin, Wang (ISIT'15): Distinguisher	$31 : 812$; $34 : 817$; $34 : 824$; $37 : 839$
Sarkar, Maitra, Baksi (DCC'17): Cube tester (Distinguisher)	$13 : 710$; $20 : 766$; $21 : 777$; $22 : 804$; $27 : 823$

Table: Cubes obtained for Trivium giving bias in key-stream

r	$\mathcal{P}^{(r)}$	Cube indices	(#)
710	0.462977	1, 3, 10, 12, 14, 23, 38, 45, 48, 50, 69, 75, 79	(13)
766	0.496747	2, 4, 6, 8, 11, 13, 15, 17, 19, 26, 28, 30, 32, 37, 55, 58, 67, 73, 76, 79	(20)
777	0.491104	2, 4, 6, 8, 11, 13, 15, 17, 19, 26, 28, 30, 32, 34, 37, 55, 58, 67, 73, 76, 79	(21)
804	0.497907	2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 56, 68, 74, 77, 80	(22)
823	0.495747	2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59, 62, 65, 68, 71, 74, 77, 80	(27)

Recap

- Grain family
- Trivium
- Attacks on Grain
- Attacks on Trivium

References

- Differential Fault Attack against Grain family with very few faults and minimal assumptions.
  Santanu Sarkar, Subhadeep Banik and Subhamoy Maitra.
  IEEE Transactions on Computers 2015.

- Conditional Differential Cryptanalysis of NLFSR-based Cryptosystems.
  Simon Knellwolf, Willi Meier and Maria Naya-Plasencia.
  Asiacrypt 2010.

- Observing Biases in the State: Case Studies with Trivium and Trivia-SC.
  Santanu Sarkar, Subhamoy Maitra and Anubhab Baksi.
  Designs, Codes and Cryptography 2017.

IIT Madras


Thank You!