Efficient FPGA implementations of high-dimensional cube testers on the stream cipher Grain-128

Jean-Philippe Aumasson  Itai Dinur  Luca Henzen
Willi Meier  Adi Shamir

SHARCS ’09
Agenda

Grain-128

Cube testers

Software precomputations

FPGA implementation

Results and extrapolation

Conclusions
Grain-128

State-of-the-art stream cipher developed within eSTREAM Project (04-08)

- designed by Hell, Johansson, Maximov, Meier (2007)
- 128-bit version of the eSTREAM cowinner Grain-v1 (2005)
- 128-bit key, 96-bit IV, 256-bit state
- previous DPA and related-key attacks
- standard-model attack on round-reduced version (192/256)
Grain-128

\[ \text{deg } f = 1, \text{deg } g = 2, \text{deg } h = 3 \]

Initialization: key in NFSR, IV in LFSR, clock 256 times

Then 1 keystream bit per clock
Cube testers (simple version)

1. pick a random key and fix \((96 - n)\) IV bits
2. vary \(n\) IV bits to obtain the evaluation of \textbf{order-}\(n\) derivative

\[
\bigoplus_{(x_0, \ldots, x_{n-1}) \in \{0,1\}^n} f(x) = \frac{\partial^n f}{\partial x_0 \ldots \partial x_{n-1}}
\]

for \textbf{well-chosen cube} (=variables), statistical bias detectable

ex: \(f\) of degree \(n \Rightarrow\) constant derivative
Comparison...

Cube attacks...
1. Find 128 cubes whose order-$n$ derivative has degree 1
2. Reconstruct their derivatives via black-box linearity tests
3. Evaluate derivatives and solve linear system to recover the key

Cube testers...
• Distinguishers rather than key-recovery
• Need less precomputation than cube attacks
• Don’t require derivatives of degree-1, but with any unexpected and testable property
How to determine variable bits?

Complexity bottleneck, and main distinction with previous high-order differential attacks

**Analytically**: find “weak” variables by analyzing the algorithm

\[
\begin{align*}
t_1 & \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171} \\
t_2 & \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264} \\
t_3 & \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69} \\
(s_1, s_2, \ldots, s_{93}) & \leftarrow (t_3, s_1, \ldots, s_{92}) \\
(s_{94}, s_{95}, \ldots, s_{177}) & \leftarrow (t_1, s_{94}, \ldots, s_{176}) \\
(s_{178}, s_{279}, \ldots, s_{288}) & \leftarrow (t_2, s_{178}, \ldots, s_{287})
\end{align*}
\]

Ex: Trivium

**Empirically**: explore the search space to find good sets of variables with discrete optimization tools
D. J. Bernstein
Hash functions and ciphers

Why haven't cube attacks broken anything?

The talk and the paper

Hundreds of cryptographers were sitting in a dark lecture room at the University of California at Santa Barbara.

"How to solve it: new techniques in algebraic cryptanalysis."

Shamir had already advertised his talk as introducing "cube attacks," a powerful new attack technique that he described as describing a stream cipher with an extremely large key, many S-boxes, etc. David Wagner later wrote that he laughed -- since it seemed ridiculous to imagine an attack on the design, yet I knew if he was describing this...
Going against the Grain

Method:

1. select $n$ variable IV bits
2. set the remaining IV bits to zero
3. set the key bits randomly
4. run Grain-128 for all the $2^n$ values and collect results
5. repeat steps 3-4 $N$ times and make statistics

we try to detect for imbalance in the distribution of the results

e.g., if derivatives look like $x_0x_1x_2 + x_1x_2x_3x_4x_5$
Going against the Grain

Method:

1. select $n$ variable IV bits
2. set the remaining IV bits to zero
3. set the key bits randomly
4. run Grain-128 for all the $2^n$ values and collect results
5. repeat steps 3-4 $N$ times and make statistics

we try to detect for imbalance in the distribution of the results
e.g., if derivatives look like $x_0x_1x_2 + x_1x_2x_3x_4x_5$

Problem 1: finding good cubes/variables (SW: C code + gcc *.c)
Going against the Grain

**Method:**

1. select $n$ variable IV bits
2. set the remaining IV bits to zero
3. set the key bits randomly
4. run Grain-128 for all the $2^n$ values and collect results
5. repeat steps 3-4 $N$ times and make statistics

We try to detect for *imbalance* in the distribution of the results, e.g., if derivatives look like $x_0 x_1 x_2 + x_1 x_2 x_3 x_4 x_5$

Problem 1: finding good cubes/variables (SW: C code + gcc *.c)
Problem 2: implementing the attack (HW: VHDL + FPGA)
Software precomputation

**Bitsliced implementation**

- 64 instances in parallel with different keys and IVs
- tester using order-30 derivatives in $\approx 45$ min

**Evolutionary algorithm**

- generic discrete optimization tool
- search variables that maximize the number of rounds attackable
- huge search space, e.g., $\binom{96}{32} \geq 2^{84}$
- quickly converges into local optima
Software precomputation

**Bitsliced implementation**
- 64 instances in parallel with different keys and IVs
- tester using order-30 derivatives in $\approx 45\text{min}$

**Evolutionary algorithm**
- generic discrete optimization tool
- search variables that maximize the number of rounds attackable
- huge search space, e.g. $\binom{96}{32} \geq 2^{84}$
- quickly converges into local optima

<table>
<thead>
<tr>
<th>Cube dimension</th>
<th>6</th>
<th>10</th>
<th>14</th>
<th>18</th>
<th>22</th>
<th>26</th>
<th>30</th>
<th>. . .</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rounds</td>
<td>180</td>
<td>195</td>
<td>203</td>
<td>208</td>
<td>215</td>
<td>222</td>
<td>227</td>
<td>. . .</td>
<td>256</td>
</tr>
</tbody>
</table>

To evaluate larger cubes we need more computational power
Grain-128 in FPGA

- 32× parallelization (32 cipher clocks/system clock)
- on Xilinx Virtex-5 LX330: 180 slices for 1 instance at 200 MHz
- 256 instances: 46080 slices, of available 51840 slices available
Cube testers in FPGA

- exploit (almost) all the slices available
- 256 Grain-128 modules work on distinct IVs
- additional units to generate inputs and to store results
  - simulation controller
  - input generator
  - output collector
- evaluation of cubes for 32 consecutive rounds
- LSFR to generate keys efficiently
FPGA parallel cube tester core

Key and IV generation

LFSR

offset_0

128

CV router

offset_1

m

CV router

offset_2

m

CV router

offset_{2m-1}

m

Partial IV

n-m

s_inst

e_inst

Incrementer

Simulation controller

s_inst

e_inst

IV_0\

IV_1\

IV_2\

IV_{2m-1}

Key

IV_0 = \text{Key}

IV_1 = \text{IV}_0

IV_2 = \text{IV}_1

\vdots

IV_{2m-1} = \text{IV}_{2m-2}

Grain_1

Grain_2

Grain_3

\vdots

Grain_{2^m}

Output collection

Out_0

Out_1

Out_2

Out_{2^m-1}

u_inst

Simulation controller

ARRAY

13 / 20
Performance and results

- evaluation of \((n + 8)\)-dimensional cubes as fast as for \(n\)-dimensional cubes with a single instance
- approx. 10 seconds for a cube of degree 30 (64 runs)
- approx. 3 hours for a cube of degree 40 (64 runs)

<table>
<thead>
<tr>
<th>Cube dimension</th>
<th>30</th>
<th>35</th>
<th>37</th>
<th>40</th>
<th>44</th>
<th>46</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nb. of queries</td>
<td>(2^{22})</td>
<td>(2^{27})</td>
<td>(2^{29})</td>
<td>(2^{32})</td>
<td>(2^{36})</td>
<td>(2^{38})</td>
<td>(2^{42})</td>
</tr>
<tr>
<td>Time</td>
<td>0.17 sec</td>
<td>5.4 sec</td>
<td>21 sec</td>
<td>3 min</td>
<td>45 min</td>
<td>3 h</td>
<td>2 days</td>
</tr>
</tbody>
</table>
Performance and results

- evaluation of \((n + 8)\)-dimensional cubes as fast as for \(n\)-dimensional cubes with a single instance
- approx. 10 seconds for a cube of degree 30 (64 runs)
- approx. 3 hours for a cube of degree 40 (64 runs)

<table>
<thead>
<tr>
<th>Cube dimension</th>
<th>30</th>
<th>35</th>
<th>37</th>
<th>40</th>
<th>44</th>
<th>46</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nb. of queries</td>
<td>(2^{22})</td>
<td>(2^{27})</td>
<td>(2^{29})</td>
<td>(2^{32})</td>
<td>(2^{36})</td>
<td>(2^{38})</td>
<td>(2^{42})</td>
</tr>
<tr>
<td>Time</td>
<td>0.17 sec</td>
<td>5.4 sec</td>
<td>21 sec</td>
<td>3 min</td>
<td>45 min</td>
<td>3 h</td>
<td>2 days</td>
</tr>
</tbody>
</table>

Found a **distinguisher on 237 rounds in \(2^{54}\) clocks**
- \(#\text{samples} \times \#\text{cipher clocks} \times \#\text{evaluations} = 64 \times 256 \times 2^{40} = 2^{54}\)
Extrapolation

Logarithmic extrapolation with standard linear model

cubes of degree 77 conjectured sufficient for the full Grain-128 ⇒ attack in $2^{83}$ initializations vs. $2^{128}$ ideally
Conclusions

First dedicated hardware for cube attacks/testers

Cube attacks/testers seem to have eventually broken something

High variance of cubes’ efficiency; preliminary discrete optimization step essential

Software experiments on Grain-v1: much more resistant (higher degree $g$)
The end

Thanks for your attention

Questions?
Search for good cubes

Evolutionary algorithm: generic discrete optimization tool

In a nutshell: population = subset of variables

1. initialize population pseudorandomly
2. reproduction (crossover + mutation)
3. selection of best fitting individuals
4. go to 2.

#generations (steps 2-4) before halting = parameter
Key-recovery attacks

- Search for IV terms with linear superpoly in the key bits (or maxterms)
- Search for maxterms is difficult for reduced variants of Grain-128

- Key bits mix non-linearly together before mixing with the IV bits
- Output bits polynomials contain few IV terms whose superpoly is linear in the key bits
- Applying linearization techniques becomes a complicated task
Observations on Grain-v1

Differences:

- The size of the LFSR and the NFSR is 80-bit
- 80-bit keys, 64-bit IVs, and 160 initialization rounds
- Feedback polynomial of NFSR has degree six and is less sparse
- Filter function $h$ is denser
- Algebraic degree and density converge faster towards ideal ones

<table>
<thead>
<tr>
<th>Rounds</th>
<th>64</th>
<th>70</th>
<th>73</th>
<th>79</th>
<th>81</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cube dimension</td>
<td>6</td>
<td>10</td>
<td>14</td>
<td>20</td>
<td>24</td>
</tr>
</tbody>
</table>

Grain-v1 seems to resist cube testers and basic cube attack techniques