RSA CONFERENCE 2014
FEBRUARY 24 – 28 | MOSCONE CENTER | SAN FRANCISCO

Share.
Learn.
Secure.
Capitalizing on
Collective Intelligence

Attacking PUF-based Pattern Matching Key Generators via Helper Data Manipulation

SESSION ID: CRYP-W01

Jeroen Delvaux

Doctoral Researcher
University of Leuven (KU Leuven) & iMinds, Belgium
Department of Electrical Engineering ESAT/COSIC

Contents

- Physically Unclonable Functions (PUF): preliminaries
- Pattern Matching Key Generator (PMKG)
  - Architecture
  - Failure behavior
  - Attacks
  - Countermeasures
- Conclusion & Further Work

Physically Unclonable Functions (PUFs)

- Emerging hardware primitive
- IC-unique function (manufacturing variations)
- Analogous: human fingerprint (biometrics)
- Main usage: secret key generation

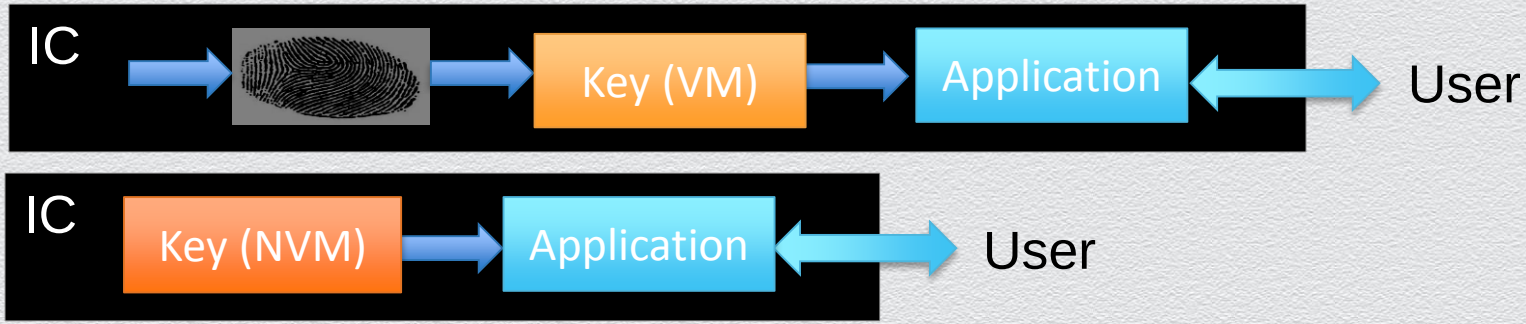
Secret Keys: PUFs versus NVM

Traditionally: NVM (e.g. Flash)

- Costly: no minimal CMOS (additional processing steps)
- Physical attacks: vulnerable

PUF

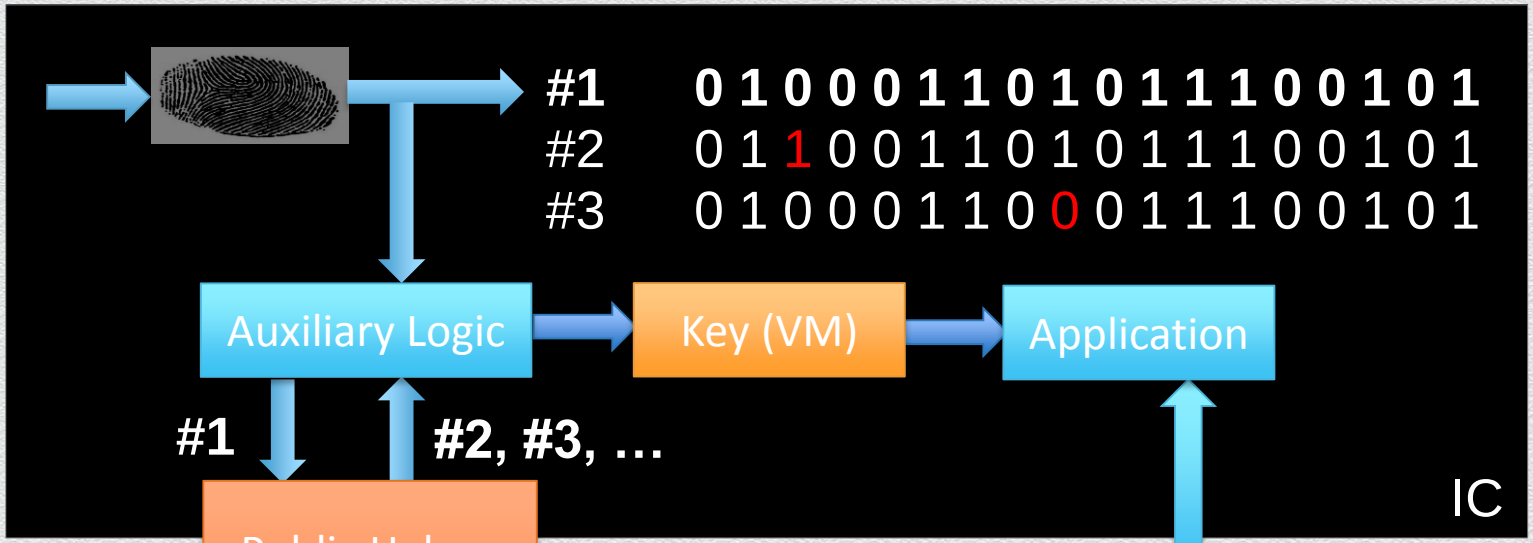
- Minimal CMOS
- Invasion damages PUF; no permanent electrical storage



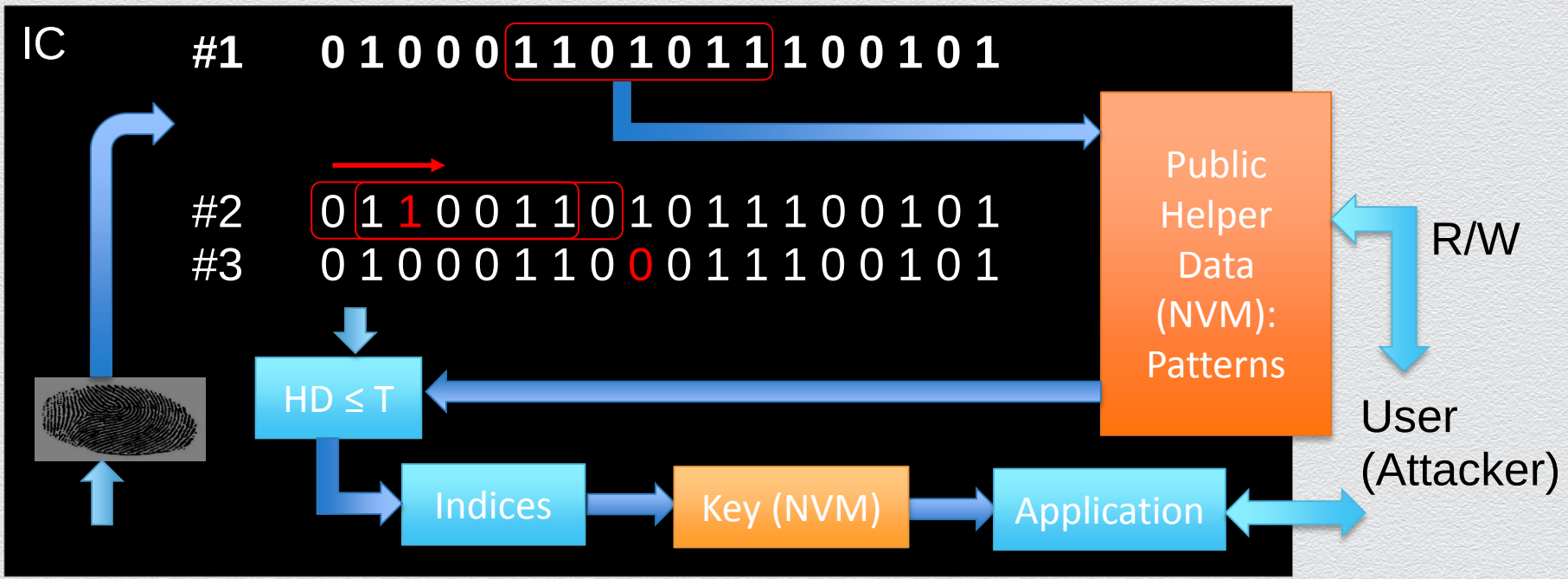
Auxiliary Logic Resolves PUF Issues

Key reproducibility: noise, ΔV, ΔT

Key entropy: non-uniformity



Pattern Matching Key Generator (HOST 2011 & patent)

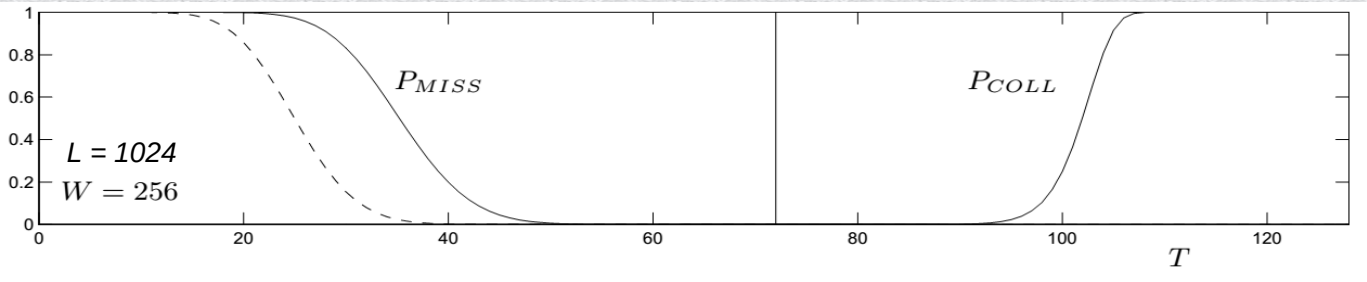


secret index = sub-key

(basic principle only)

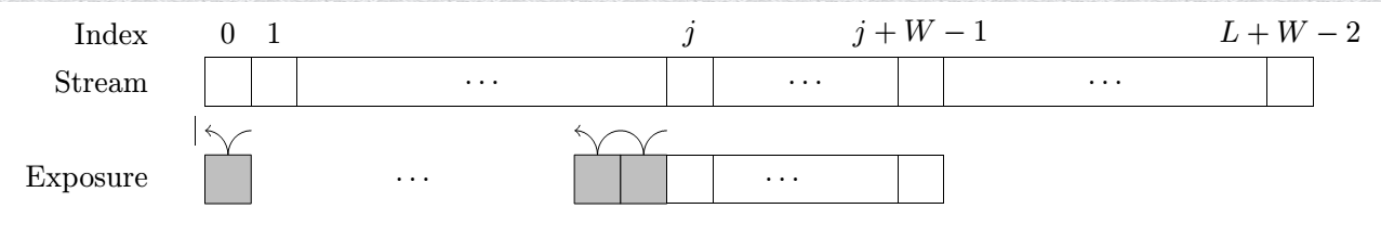
PMKG Failure Conditions

- Pattern Miss: HD > T @ sub-key index
- Pattern Collision: HD ≤ T @ non sub-key index
- We construct approximating formulas for failure probabilities
  - System parameters: W = pattern width, L = #indices, T = threshold
  - Incorporate PUF statistics (65nm CMOS measurements)



Attacks: Common Framework

- Expose more stream bits (e.g. left to right), one-by-one
- Two hypotheses for unknown bit: ‘0’ or ‘1’
  - Corresponding helper data (manipulation)
  - Observe (small) statistical difference in PMKG failure rate
- Exceeding index 0: abrupt change in failure rate

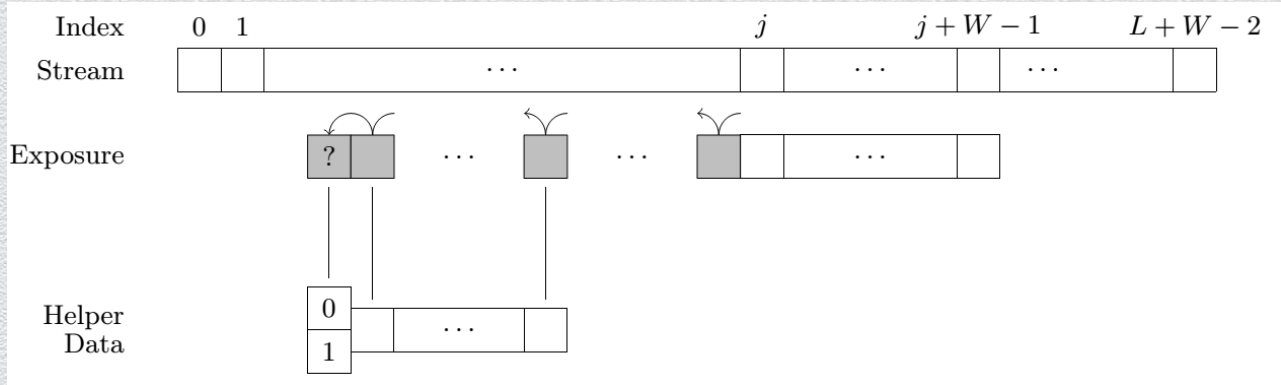


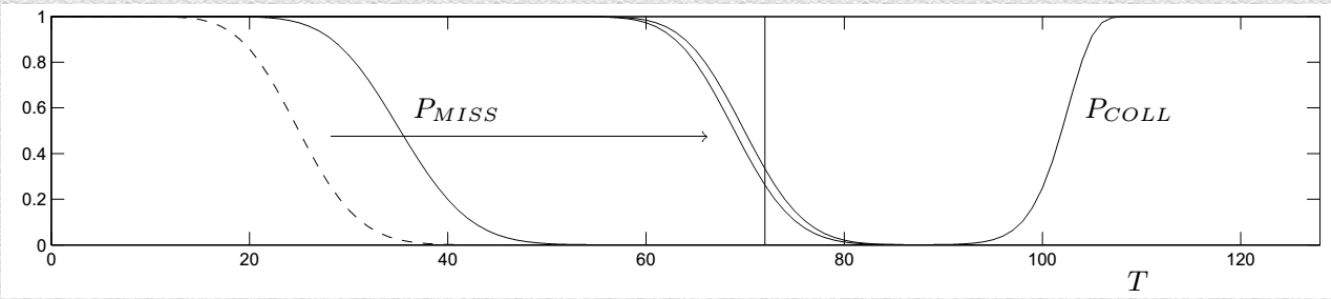
j = secret index,
W = pattern witdh,
L = #indices

W, L, T known (Kerckhoff’s principle)

Snake I: Exploit Pattern Misses

- Shift sub-key
- Correct guess: less misses
- Exceed index 0: miss always

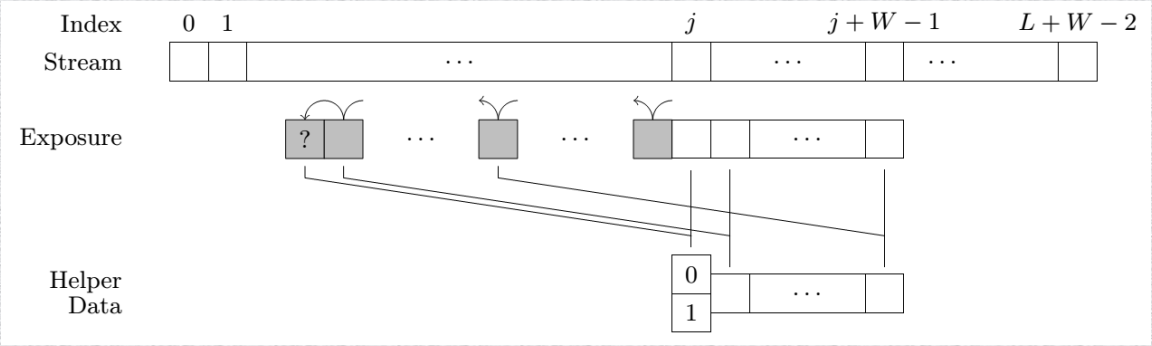




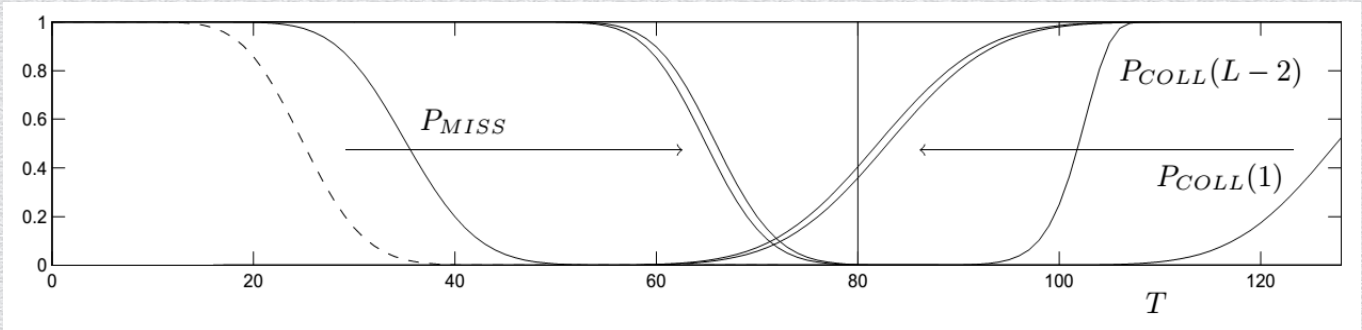
j = secret index,
W = pattern widh,
L = #indices
T = threshold

Snake II: Exploit Pattern Collisions

- Correct guess: more collisions
- Exceed index 0: collide never



- Pattern misses side-effect complicates the attack



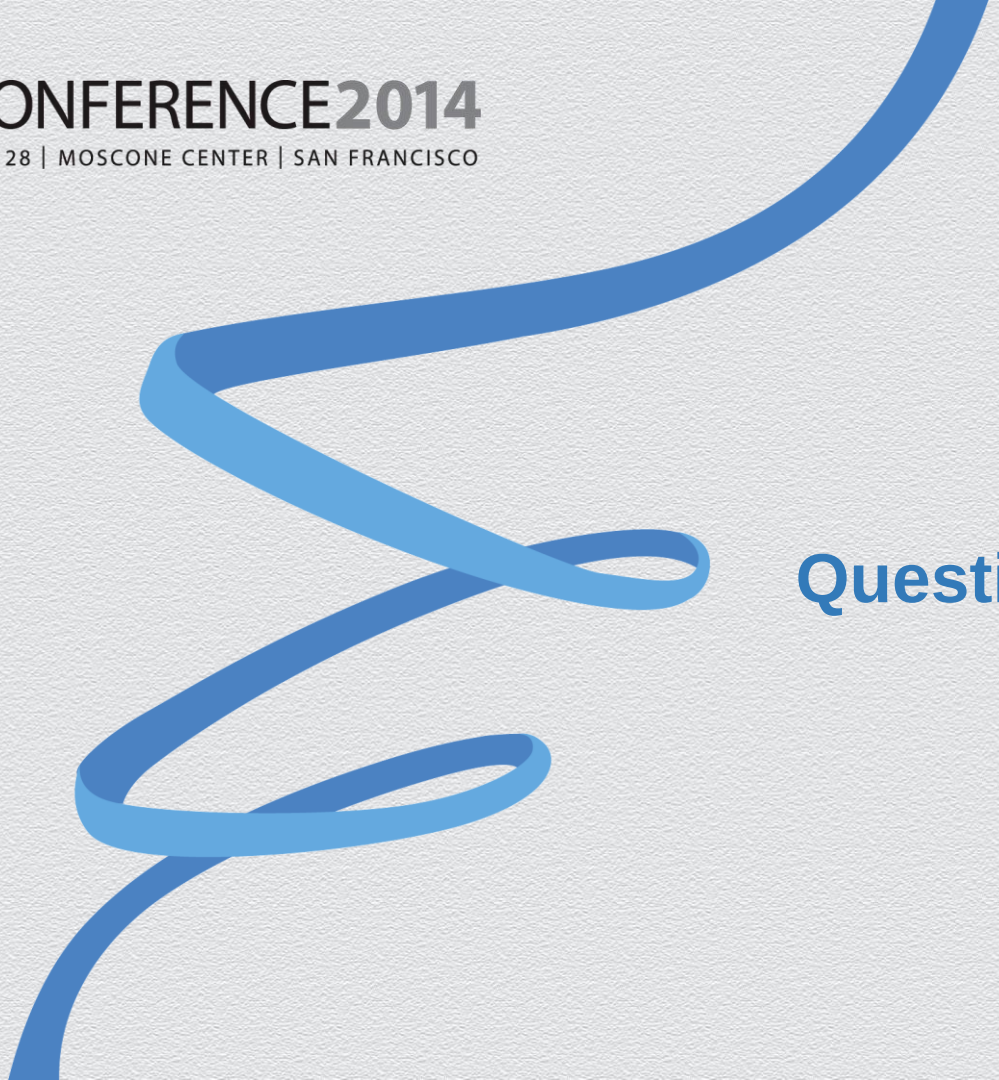
j = secret index,
W = pattern widh,
L = #indices
T = threshold

Effective countermeasures

- Avoid abrupt change in failure rate @ index 0
- Non-overlapping patterns
  (patent only / not HOST; not proposed with a security objective)
  - #stream bits = L*W (before L + W – 1)
  - Inefficient
- Newly proposed: circularity of the response bits
  - #stream bits = L

Conclusion & Further Work

- PMKG vulnerable to helper data manipulation
- Statistical observation of the failure rate
- Countermeasure: circularity of PUF bits
- PMKG basic principle: HD measurements
  - Simple operations, compared to traditional method (ECC, Hash function)
  - Lightweight for some use cases?
  - Secure variants?



Questions?

RSA CONFERENCE 2014
FEBRUARY 24 – 28 | MOSCONE CENTER | SAN FRANCISCO

Share.
Learn.
Secure.
Capitalizing on
Collective Intelligence

On Increasing the Throughput of Stream Ciphers

SESSION ID: CRYP-W01

Frederik Armknecht, Vasily Mikhalev

University Mannheim
Theoretical Computer Science and IT Security Group

Contents

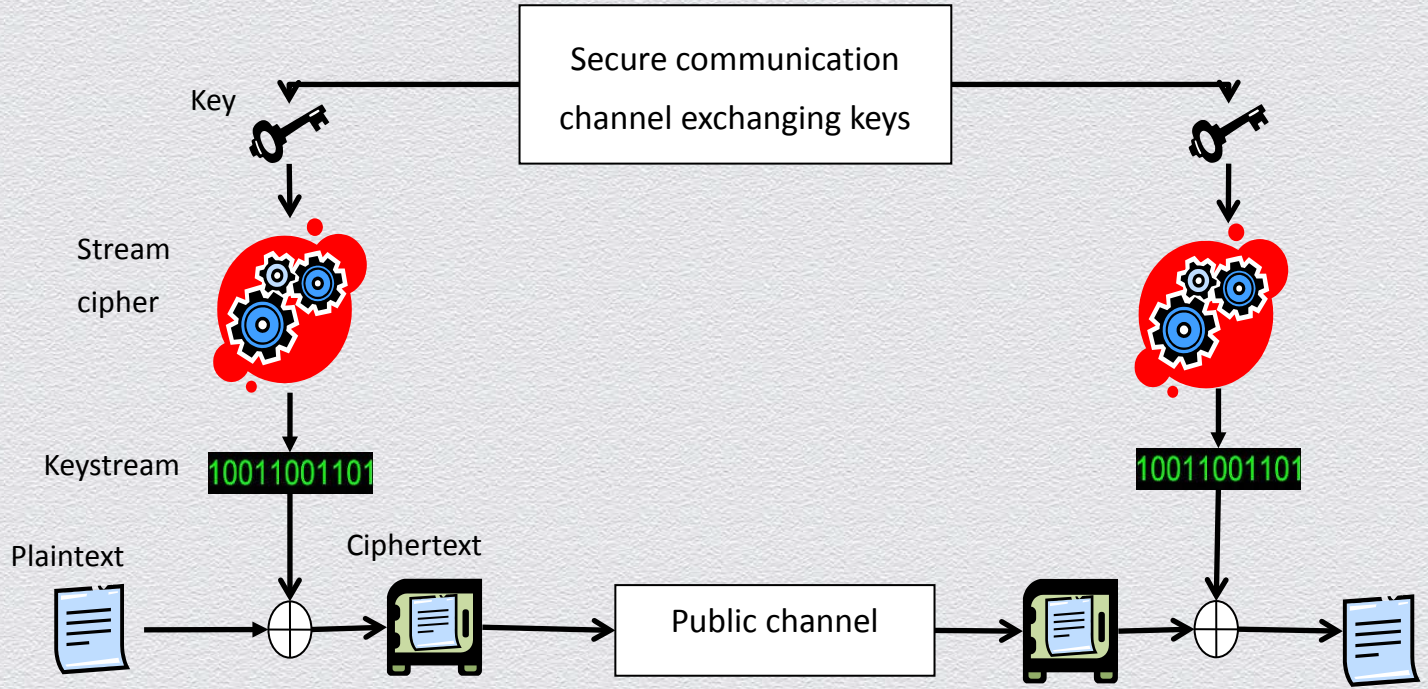
- Introduction
- New Transformation
- Conclusion

Introduction

Stream Ciphers

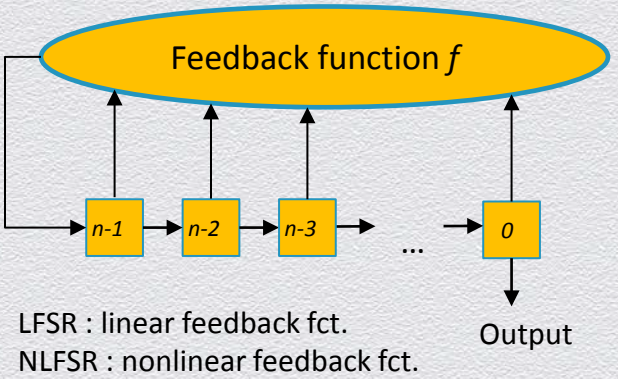
- Designed for efficiently encrypting data streams of arbitrary length
- Preferable choice if real-time encryption is required
- Most popular application scenario: Mobile communication
- Examples:
  - GSM: A5-ciphers
  - Bluetooth: E0 ciphers
  - SSL: RC4 cipher

Principle of Stream Ciphers



Produce long bitstreams
from short seed?

Feedback Shift Registers

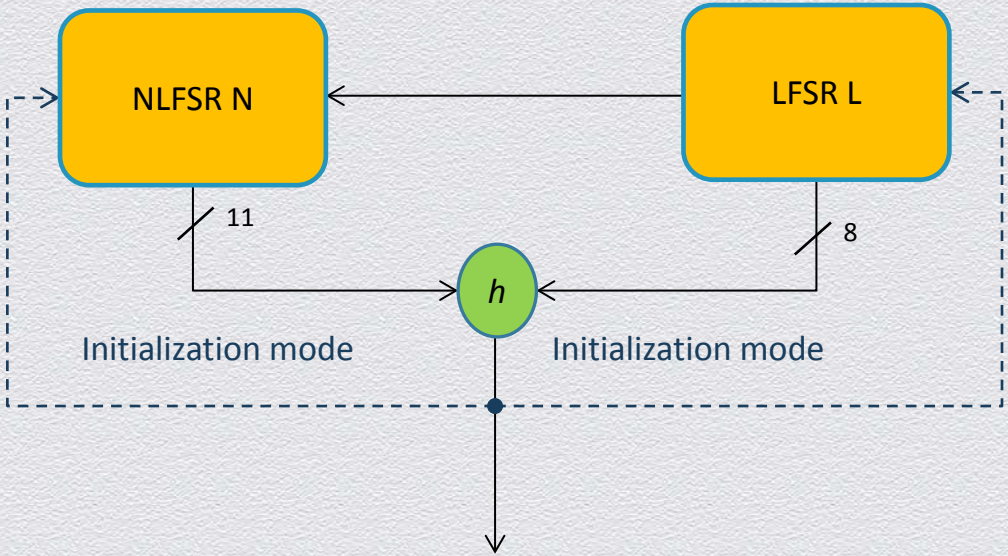


LFSR : linear feedback fct.
NLFSR : nonlinear feedback fct.

At each clock-cycle:
1) The value of stage 0 is output
2) The new value of stage n-1 is computed
3) All other values are shifted

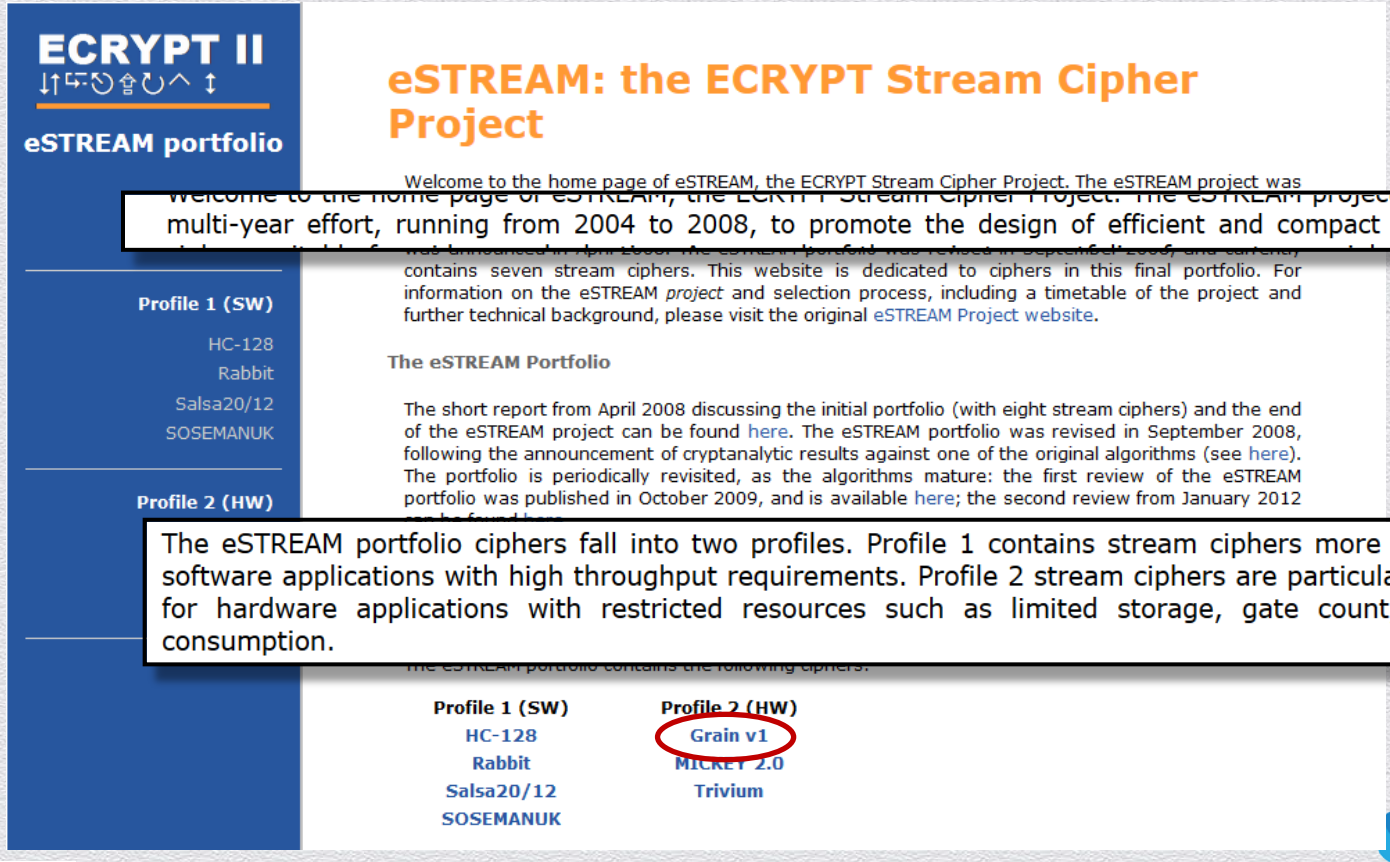
- Hardware efficient
- Important building blocks of stream ciphers
- Can produce bit streams with high period
- Often combined with other components

Example: Grain-128



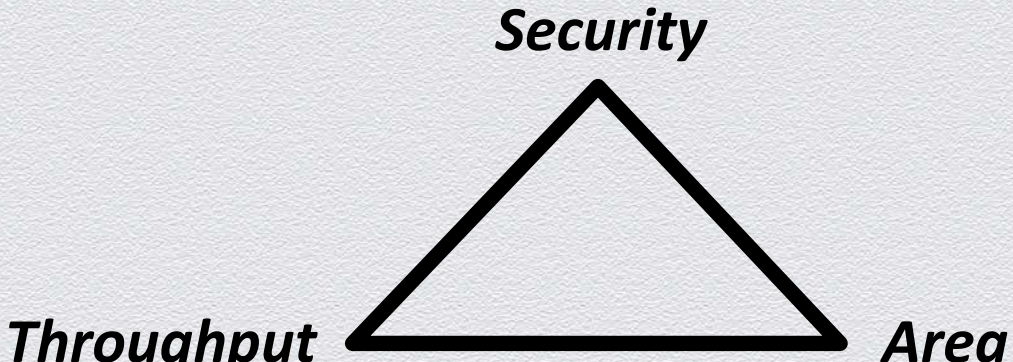
Grain 128 e-STREAM finalist in the second profile portfolio (restricted hardware recourses)

eStream


ECRYPT II
eSTREAM portfolio
eSTREAM: the ECRYPT Stream Cipher Project
Welcome to the home page of eSTREAM, the ECRYPT Stream Cipher Project. The eSTREAM project was
multi-year effort, running from 2004 to 2008, to promote the design of efficient and compact stream
contains seven stream ciphers. This website is dedicated to ciphers in this final portfolio. For information on the eSTREAM project and selection process, including a timetable of the project and further technical background, please visit the original eSTREAM Project website.
The eSTREAM Portfolio
The short report from April 2008 discussing the initial portfolio (with eight stream ciphers) and the end of the eSTREAM project can be found here. The eSTREAM portfolio was revised in September 2008, following the announcement of cryptanalytic results against one of the original algorithms (see here). The portfolio is periodically revisited, as the algorithms mature: the first review of the eSTREAM portfolio was published in October 2009, and is available here; the second review from January 2012
Profile 1 (SW)
HC-128
Rabbit
Salsa20/12
SOSEMANUK
Profile 2 (HW)
The eSTREAM portfolio ciphers fall into two profiles. Profile 1 contains stream ciphers more suitable for software applications with high throughput requirements. Profile 2 stream ciphers are particularly suitable for hardware applications with restricted resources such as limited storage, gate count, or power consumption.
Profile 1 (SW)
HC-128
Rabbit
Salsa20/12
SOSEMANUK
Profile 2 (HW)
Grain v1
MICKEY 2.0
Trivium

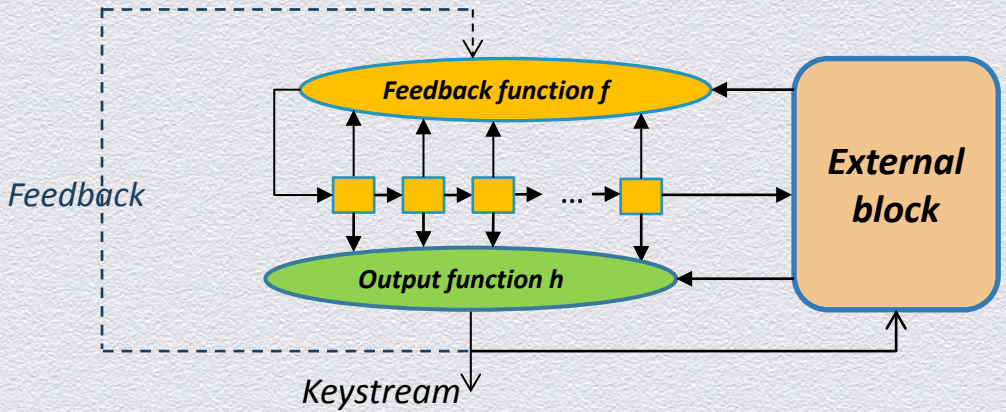
Requirements

Design of a cryptographic primitives is a trade-off between:



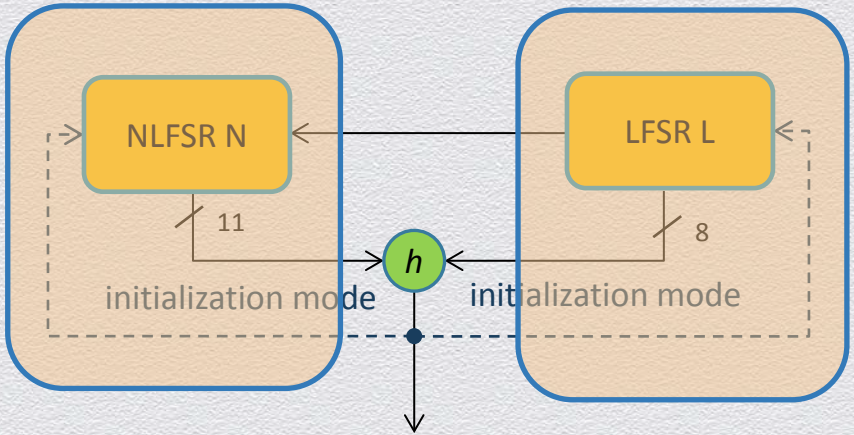
Can we increase
the **throughput**
without (or only little)
increase of the **area**?

Generic Stream Cipher



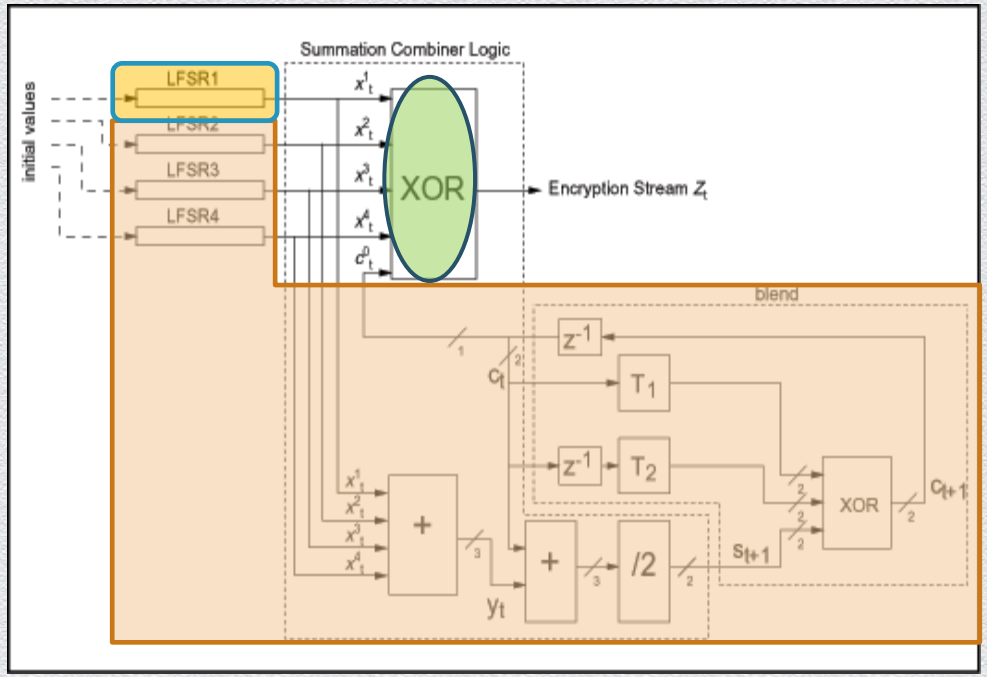
- We consider a generic stream cipher composed of three building blocks:
  - A feedback shift register FSR
  - An external block
  - An output function
- Majority of stream ciphers comes within this structure

Example: Grain-128

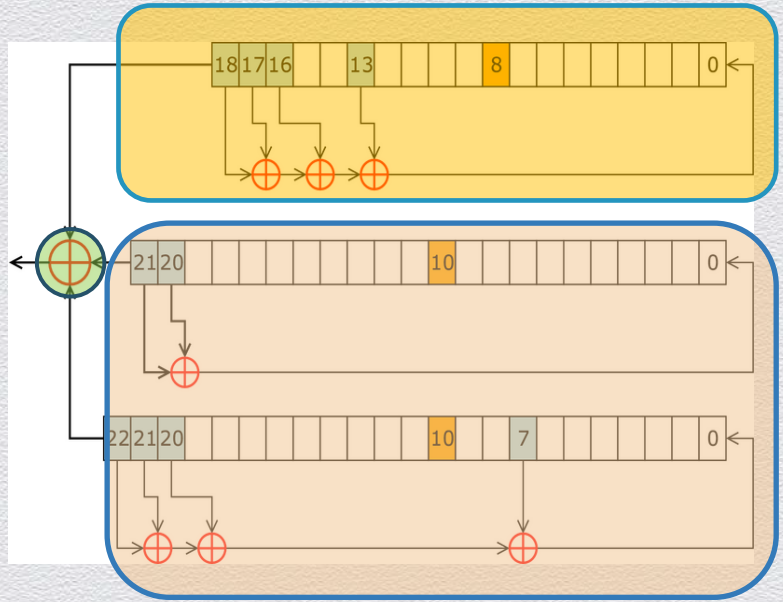


Example: E0 (Bluetooth Standard)

- 4 LFSRs
- Additional 4-bit memory
- Memory ≈ Addition with carry



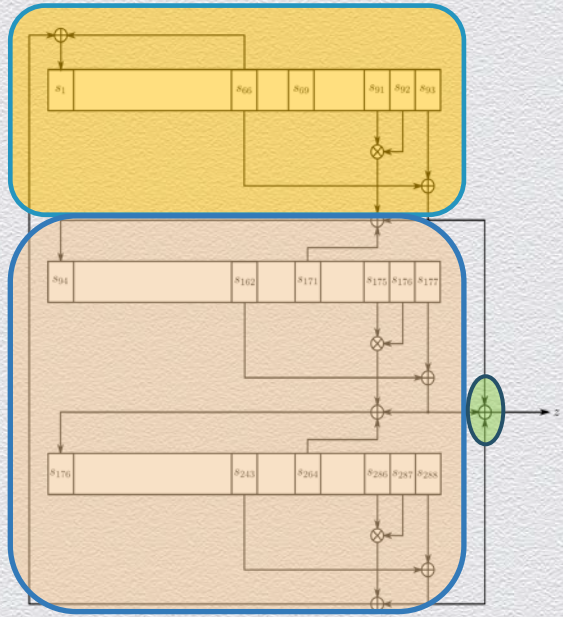
A5/1 (GSM Standard)

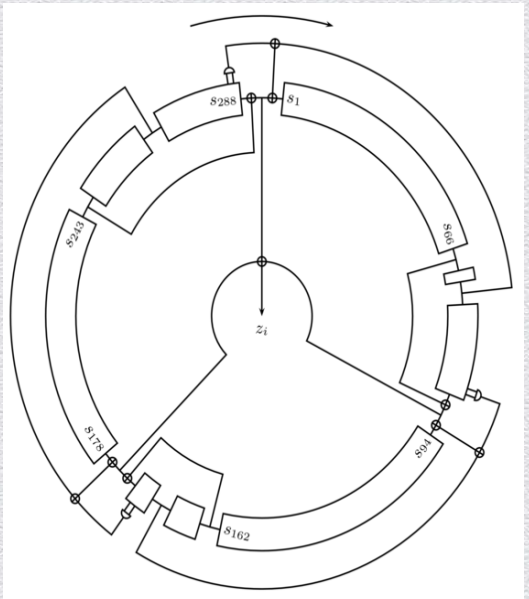


- 3 LFSRs
- Irregular clocking based on majority decision

Trivium

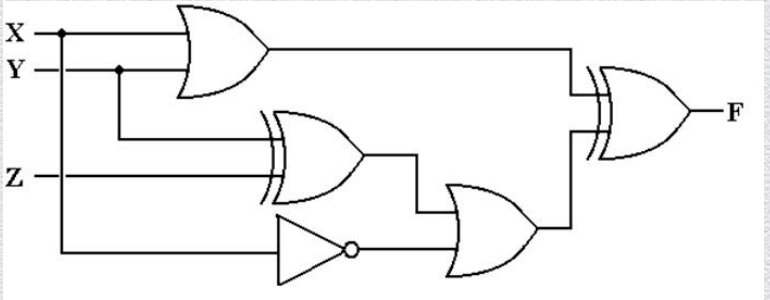
- 3 non-linear feedback shift registers
- Another eStream finalist (low area hardware)





Throughput

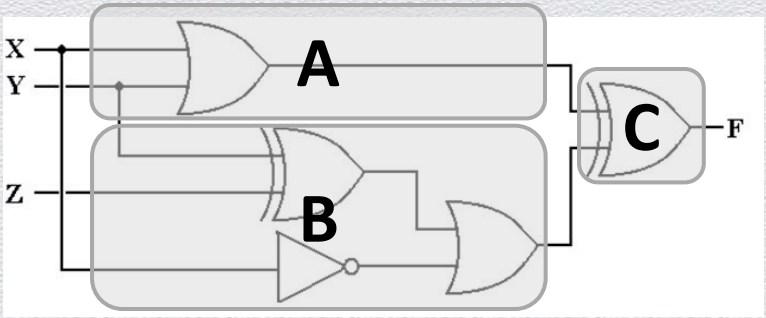
- *Throughput* = rate at which new ouput is produced with respect to time
- Hardware implementation = circuit



- Each component has a certain *delay*
- Impacts *total delay* of circuit
- Short total delay = higher frequency possible
  = increase in maximum throughput

Total Delay

- Example:



- Total Delay Formula:

$$\text{Delay(Circuit)} = \max\{\text{Delay(A)},\text{Delay(B)}\} + \text{Delay(C)}$$

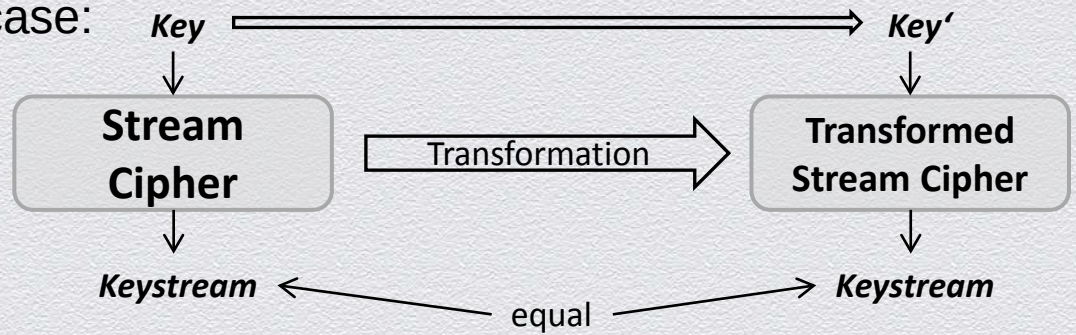
- Here:

$$\text{Delay(Circuit)} = \text{Delay(B)} + \text{Delay(C)}$$

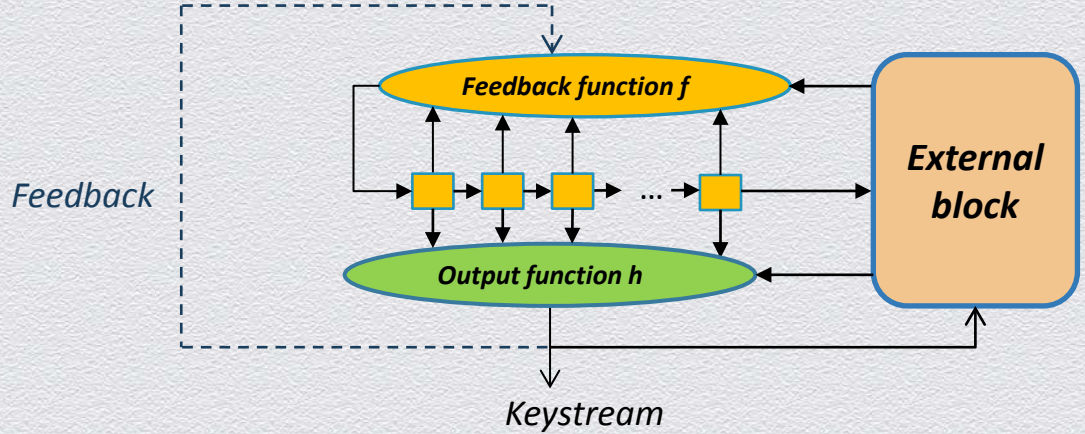
- Reducing Delay(B) decreases Delay(Circuit) !!

Preserving Transformations

- Modification does **not** change the functionality
- Preserving transformation T on circuit C:
  - For any input (initial state) of C,
  - there exists a corresponding input (initial state) for T(C)
  - such that both produce the same output (bitstream)
- In our case:



Improving the Throughput



Three possible approaches

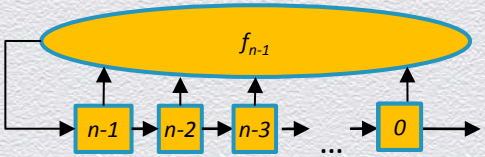
a) Decrease Delay(Feedback Shift Register)

b) Decrease Delay(Output Function h)

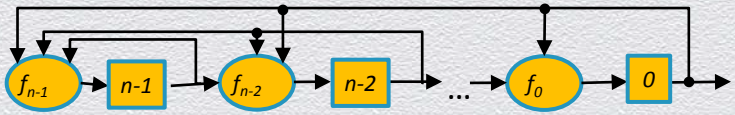
c) Decrease Delay(External Block) (*out of scope*)

a) Modifying Feedback Shift Register

- Motivation: Parallelize feedback function
- Realization: Each state entry has its „own“ feedback function



Fibonacci configuration



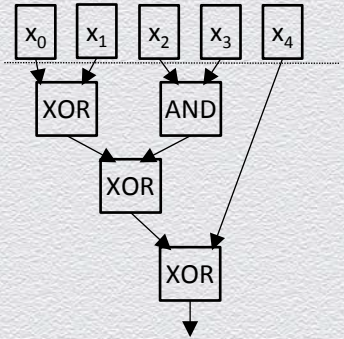
Galois configuration

$$f_5 = x_0 + x_1x_2 + x_3x_4 + x_4$$
$$f_4 = x_5$$
$$f_3 = x_4$$
$$f_2 = x_3$$
$$f_1 = x_2$$
$$f_0 = x_1$$

$$f_5 = x_0 + x_1x_2$$
$$f_4 = x_5 + x_2x_3$$
$$f_3 = x_4 + x_2$$
$$f_2 = x_3$$
$$f_1 = x_2$$
$$f_0 = x_1$$

b) Modifying Output Function

- Main idea also here: parallelization
- Example: implementing function $f=x_0+x_1+x_2x_3+x_4$ using 2-input gates
- Straightforward:

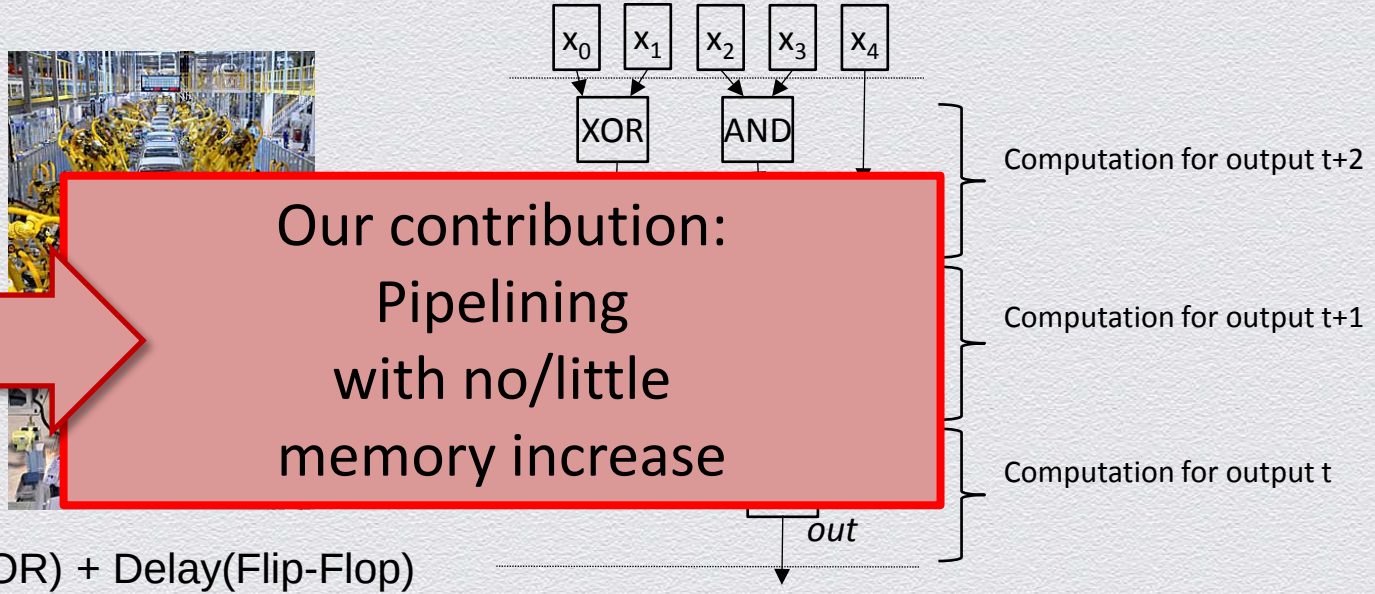


$$\text{Delay}(f) = \max\{\text{Delay}(\text{XOR}),\text{Delay}(\text{AND})\} + \text{Delay}(\text{XOR}) + \text{Delay}(\text{XOR})$$
$$= 3 \times \text{Delay}(\text{XOR})$$

- Computing n outputs takes time: $n \times$ Delay(f)
- We can do better by using pipelining

Pipelining

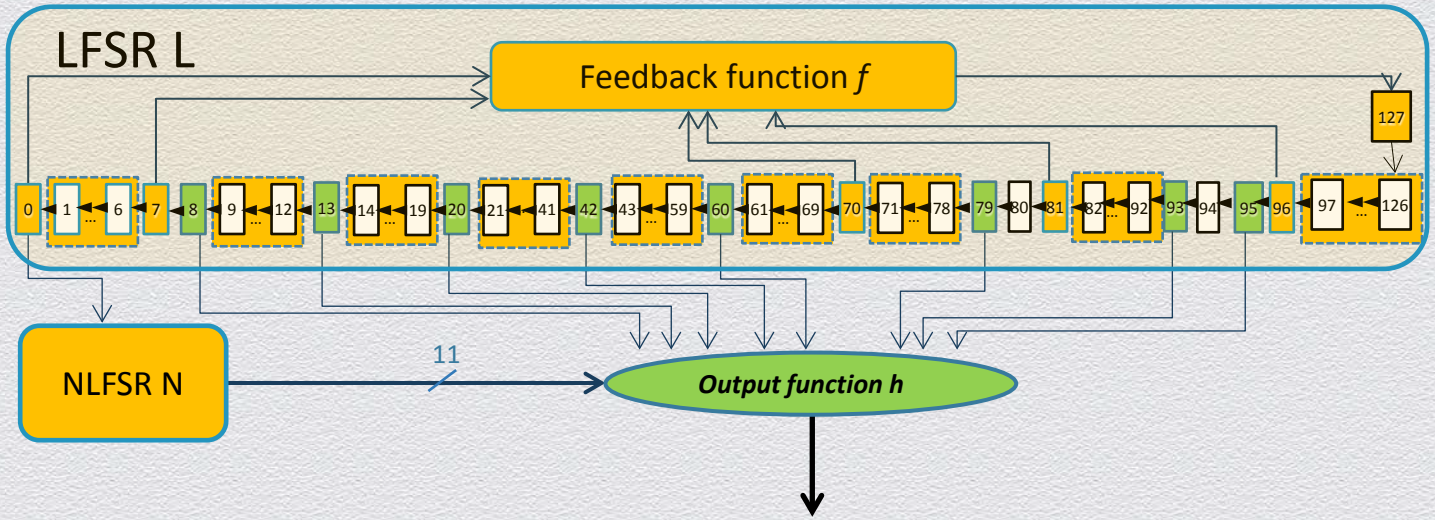
- Idea: Once one layer of computation is finished for one ouput, it starts with computatin for next output



- Delay: Delay(XOR) + Delay(Flip-Flop)
- Problems: Intermediate values need to be stored → memory increase

New Transformation

LFSR of Grain-128



Schematic View of LFSR L

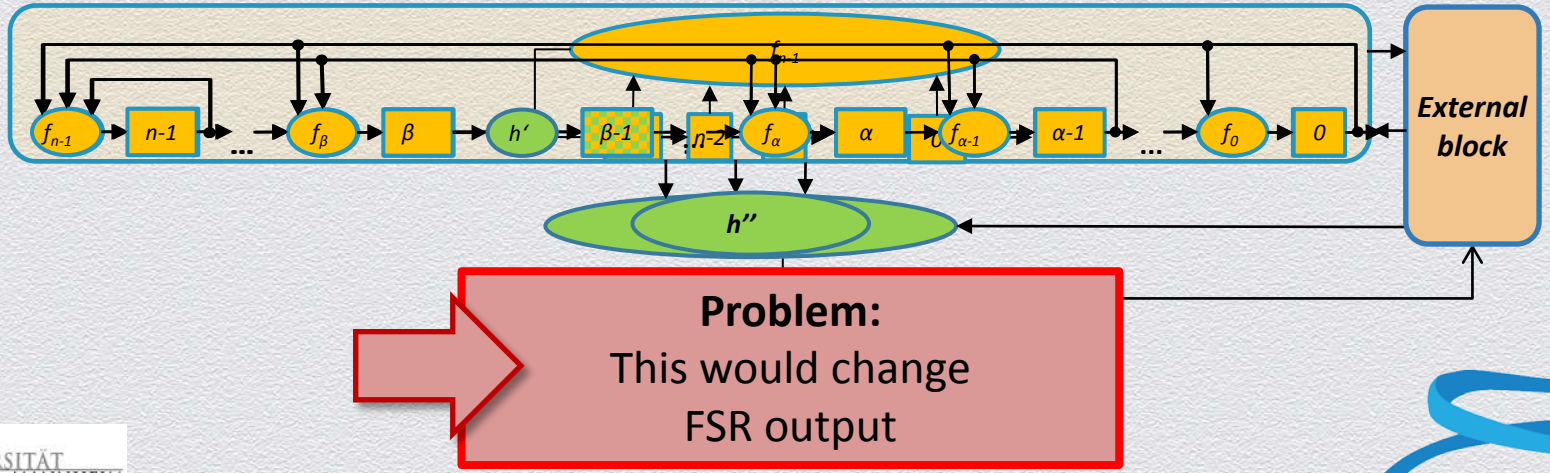
Idea

- Schematic View:

- Observation:
  - **114 of 128 stages** are not used by feedback function or other components
  - **Only task: store and shift values**
- Idea:
  - **Use these regions for storing intermediate values**

Idea (cont.)

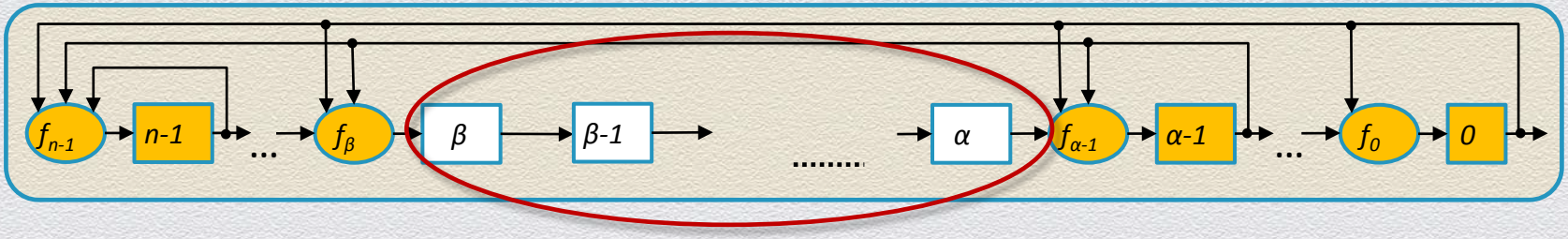
- Use feedback shift register in Galois configuration
- Move part of the output function h into the feedback function of the FSR



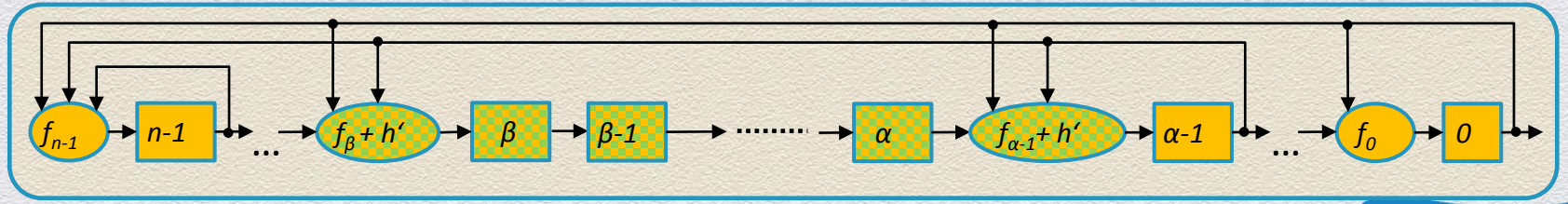
Idea (cont.)

- To „correct“ the modification, we cancel out the difference at some later stage
- Identify *isolated interval:*

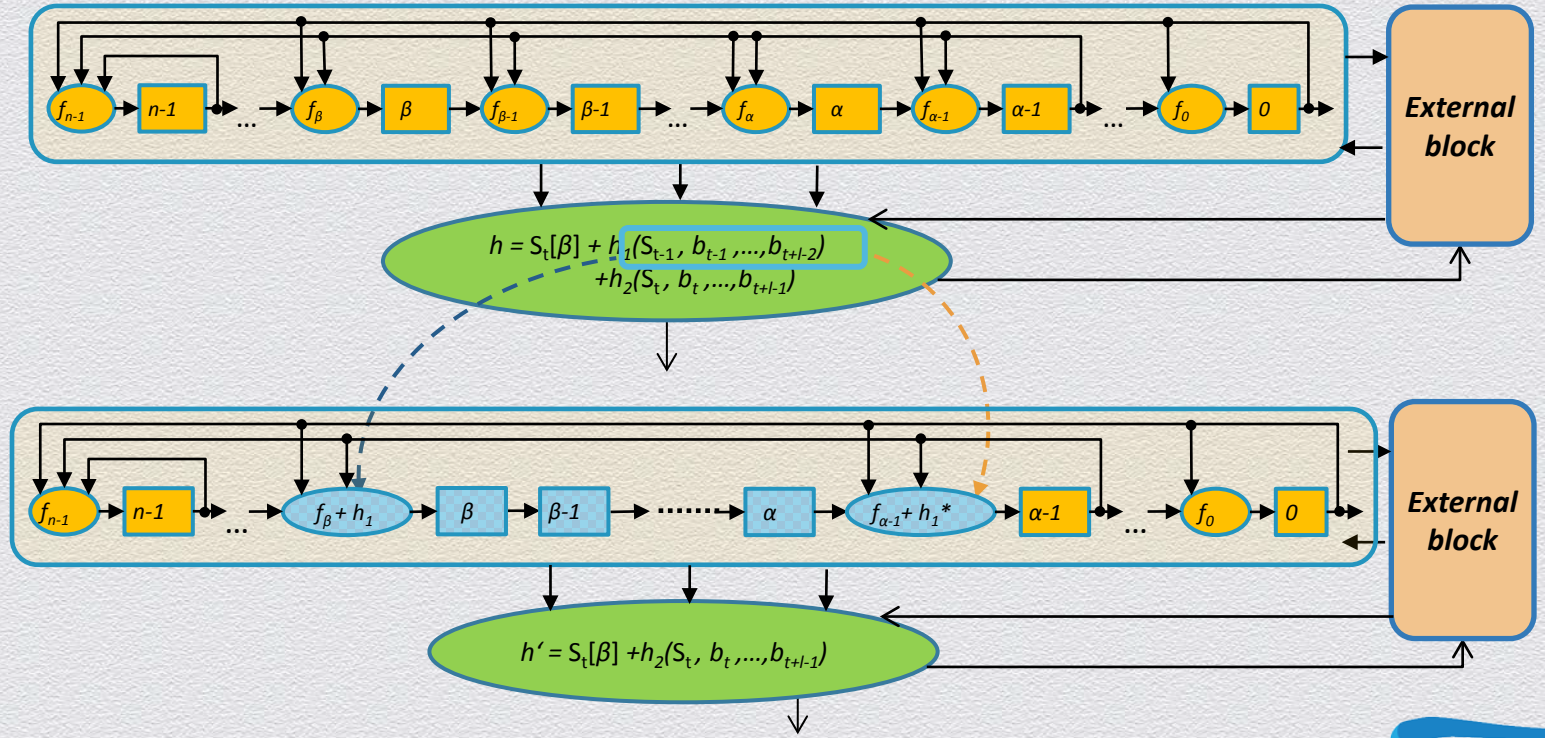
a) State entries are only shifted
b) Updates outside the interval are independent of values inside the interval



- Insert change at beginning of interval and cancel it out at the end



Cipher Transformation

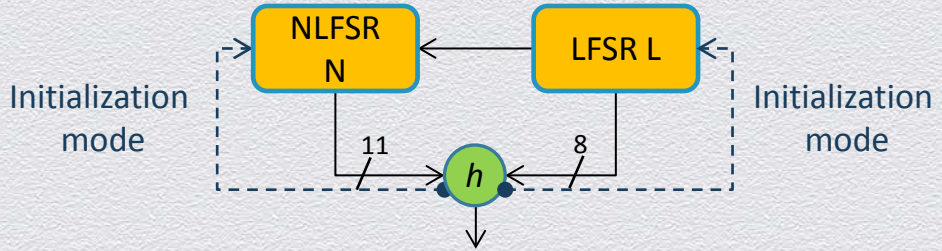


$h_1(S_t, b_t,\ldots)=h_1^*(S_{t+r}, b_{t+r},\ldots)$

Requirement: Inputs to h_1 are still available r clocks later (*r-sustainable values*)

Application

- We applied the transformations to Grain-128
- Two modes:
  - Initialization mode (feedback from *h* to FSRs)
  - Keygeneration mode (no feedback from *h* to FSRs)



- Specifications

NLFSR N feedback functions :

$$g_{127} = L_t[0] + N_t[0] + N_t[26] + N_t[56] + N_t[91] + N_t[96] + N_t[3]N_t[67] + N_t[11]N_t[13] + N_t[17]N_t[18] + N_t[27]N_t[59] + N_t[40]N_t[48] + N_t[61]N_t[65] + N_t[68]N_t[84]$$

$$g_i = N_t[i+1], 0 \le i \le 127$$

LFSR L feedback functions:

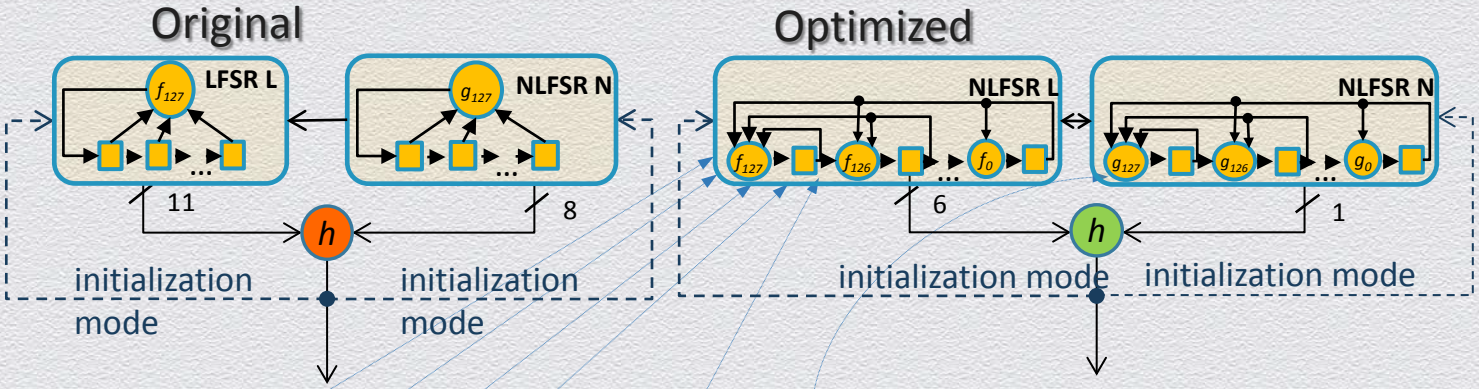
$$f_{127} = L_t[0] + L_t[7] + L_t[38] + L_t[70] + L_t[81] + L_t[96]$$

$$f_i = L_t[i+1], 0 \le i \le 127$$

Output function *h*:

$$h = N_t[2] + N_t[15] + N_t[36] + N_t[45] + N_t[64] + N_t[73] + N_t[89] + L_t[93] + L_t[13]L_t[20] + N_t[95]L_t[42] + L_t[60]L_t[79] + N_t[12]L_t[8] + N_t[12]N_t[95]L_t[95]$$

Optimization of Grain-128



$$h = N_2 + N_{15} + N_{36} + N_{45} + N_{64} + N_{73} + L_{93} + N_{89} + N_{12}L_8 + L_{13}L_{20} + N_{95}L_{42} + N_{60}L_{79} + N_{12}N_{95}L_{95}$$

$$h' = N'_{15} + N'_{36} + N'_{45} + N'_{64} + N'_{73} + L'_{93} + N'_{89}$$

Exact Specifications of Grain-128

Original

LFSR L feedback functions:

$f_{127} = L_t[0] + L_t[7] + L_t[38] + L_t[70] + L_t[81] + L_t[96]$

$f_i = L_t[i+1], 0 \le i \le 127$

NLFSR N feedback functions :

$g_{127} = L_t[0] + N_t[0] + N_t[26] + N_t[56] + N_t[91] + N_t[96] + N_t[3]N_t[67] + N_t[11]N_t[13] + N_t[17]N_t[18] + N_t[27]N_t[59] + N_t[40]N_t[48] + N_t[61]N_t[65] + N_t[68]N_t[84]$

$g_i = N_t[i+1], 0 \le i \le 127$

Output function: $h = N_t[2] + N_t[15] + N_t[36] + N_t[45] + N_t[64] + N_t[73] + N_t[89] + L_t[93] + L_t[13]L_t[20] + N_t[95]L_t[42] + L_t[60]L_t[79] + N_t[12]L_t[8] + N_t[12]N_t[95]L_t[95]$

Optimized:

NLFSR L feedback functions:

$f_{127} = L_t[0]$
$f_{123} = L_t[124] + L_t[3]$
$f_{119} = L_t[120] + L_t[30]$
$f_{111} = L_t[112] + L_t[80]$
$f_{103} = L_t[104] + L_t[46]$
$f_{97} = L_t[98] + L_t[51]$
$f_{93} = L_t[94] + L_t[61]L_t[80]$
$f_{91} = L_t[92] + L_t[59]L_t[78]$

NLFSR N feedback functions :

$g_{127} = L_t[0] + N_t[0]$
$g_{125} = N_t[126] + N_t[1]N_t[65]$
$g_{123} = N_t[124] + N_t[7]N_t[9]$
$g_{121} = N_t[122] + N_t[20]$
$g_{119} = N_t[120] + N_t[9]N_t[10]$
$g_{117} = N_t[118] + N_t[17]N_t[49]$
$g_{113} = N_t[114] + N_t[77]$
$g_{111} = N_t[112] + N_t[80]$
$g_{100} = N_t[101] + N_t[34]N_t[38]$
$g_{99} = N_t[100] + N_t[40]N_t[56]$
$g_{98} = N_t[99] + N_t[11]N_t[19]$
$g_{97} = N_t[98] + N_t[26]$
$g_{89} = N_t[90] + N_t[3]$
$g_{87} = N_t[88] + N_t[1]$
$g_{73} = N_t[74] + N_t[13]L_t[9]$
$g_{71} = N_t[72] + N_t[11]L_t[7]$
$g_{64} = N_t[65] + N_t[14]N_t[21]$
$g_{62} = N_t[63] + N_t[12]N_t[19]$
$g_{36} = N_t[37] + N_t[96]N_t[43]]$
$g_{34} = N_t[35] + N_t[94]N_t[41]]$
$g_{15} = N_t[16] + N_t[13]N_t[96]N_t[96]]$
$g_{13} = N_t[14] + N_t[11]N_t[94]N_t[94]]$

All update functions are computed in parallel. The one with the biggest delay is $g_{15} = N_t[16] + N_t[13]N_t[96]N_t[96]]$

Output function: $h = N_t[15] + N_t[36] + N_t[45] + N_t[64] + N_t[73] + N_t[89] + L_t[93]$

Implementation Results

- Implementation details
  - Hardware description language: Verilog
  - Integrated circuit: ASIC
  - Cell library: Faraday Design Kit for UMC L180 GII technology library
  - Synthesis and simulation using: Cadence RTL Compiler*
- Results:

Compiler settings	Change throughput		Change Area
	Initialization	Keystream generation	
Optimize throughput	Increase by 18% (from 1.11 GHz to 1.31 GHz)	Increase by 24% (from 1.29 GHz to 1.45 GHz)	Decrease by 46 GE (from 1794 GE to 1748 GE)
Optimize area	Increase by 20% (from 0.60 GHz to 0.72 GHz)	Increase by 20% (from 0.90 GHz to 1.08 GHz)	Increase by 29 GE (from 1627 GE to 1656 GE)

***The compiler offers a set of algorithms that perform synthesis based on concurrent optimization of timing/area/power consumption. It is commonly used in research.**

Conclusion

Contribution

- A generic transformation for increasing throughput without memory increase
- Applicable to a broad class of stream ciphers
- Idea: follow pipelining approach but re-use existing structures for saving memory
- Theory:
  - Formal description and proof of correctness
- Practice:
  - Applied to Grain-128 stream cipher
  - Increase of throughput by 18% (24%) and no area increase

Future Work

- Developing an algorithm which automatically finds a (nearly) optimal solution
- Application to other stream ciphers
- Design of a new stream cipher with decreased hardware-size
- Other applications?

Thank you very much!

RSA CONFERENCE 2014
FEBRUARY 24 – 28 | MOSCONE CENTER | SAN FRANCISCO

Share.
Learn.
Secure.
Capitalizing on
Collective Intelligence

On Double Exponentiation for Securing RSA against Fault Analysis

SESSION ID: CRYP-W01

Lê Đức Phong

Temasek Laboratories
National University of Singapore

Hardware Implementations

Outline

- RSA and Fault Analysis (FA)
  - Fault attack against CT-RSA implementation
  - Usual countermeasures
- Using Double exponentiation against FA
  - Principle
  - Improvements: Binary, sliding-window methods
  - Right-to-Left Double exponentiation
  - Performance Comparison
- Conclusion

Fault Analysis Attack

- In traditional security model (provable security):
  - one should believe that cryptographic primitives are secure whenever *computational complexity assumptions* (Factorization, RSA, DL, …) are difficult to solve
- But, in practice:
  - Fault analysis attack [BDL97] recovers the secret key from the RSA-CRT signature scheme due to only one faulty signature
  - Fault Injection:
    - Various mechanisms
    - To misinterpret, skip instructions, random modify data, …

Fault attacks against RSA-CRT

- RSA-CRT (RSA with CRT mode) uses to Chinese Reminder Theorem
  - $N=p \cdot q$: RSA modulus, p and q: large primes
  - $e \cdot d = 1 \bmod (p-1)(q-1)$
  - $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$
  - I_q: inverse of q modulo p
- Signature S *of a message* m
  - $S_p = m^{dp} \bmod p$ *and* $S_q = m^{dq} \bmod q$
  - $S = CRT(S_p, S_q) = S_q + q \cdot ((S_p - S_q) \cdot I_q \bmod p)$
- 4x faster than RSA straightforward implementation
- Reduce the size of data

Fault attacks against RSA-CRT

- Signature S *of a message* m
  1. $S_p = m^{dp} \bmod p$ ← Fault attack
     $S_q = m^{dq} \bmod q$
  2. $S = CRT(S_p, S_q) = S_q + q \cdot ((S_p - S_q) \cdot I_q \bmod p)$
- Find primes p and q
  - $q = GCD((S - \underline{S}) \bmod N, N)$ *or* $q = GCD((\underline{S}^e - m) \bmod N, N)$
- Basic countermeasures
  - compute a signature twice and compare the two results
  - verify the signature before returning

Usual countermeasures

- Modulus extension based countermeasures (Sharmir's trick)
  - Concept: compute $S^* = m^d \bmod rN$ and $Z = m^d \bmod r$
  - Check whether $S^* \equiv Z \bmod r$
- Self-secure exponentiation
  - Montgomery ladder, R2L multiply always exponentiation
  - Check the coherence of variables, *e.g.* in Montgomery ladder
    - Check whether:

$$R_0 \cdot m = R_1$$

Double addition chain [Riv09]

- In CT-RSA 2009, Rivain presented an alterative solution, called *double addition chain*
  - Basic concept: compute $A = m^d$ and $B = m^{\varphi(N) - d}$
  - Check whether the relation $A \times B \equiv 1 \bmod N$
- Need to find a short addition chain to raise m to both powers d and $\varphi(N) - d$. Rivain introduced an efficient heuristics
  - Basically, it is a binary method, compute *on-the-fly*
  - *e.g.,* need to compute (m^7, m^{35}) → 10 multiplications

$$(0,1) \xrightarrow{+} (1,1) \xrightarrow{\times 2\ +1} (1,3) \xrightarrow{\times 2\ +1} (1,7) \xrightarrow{+} (7,8) \xrightarrow{\times 2\ +1} (7,17) \xrightarrow{\times 2\ +1} (7,35)$$

Double exponentiation

Improvement to binary method

- Formally, he defined

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\alpha_i, \beta_i/2) & \text{if } \alpha_i \leq \beta_i/2 \text{ and } \beta_i \text{ is even} \\ (\alpha_i, (\beta_i - 1)/2) & \text{if } \alpha_i \leq \beta_i/2 \text{ and } \beta_i \text{ is odd} \\ (\beta_i - \alpha_i, \alpha_i) & \text{if } \alpha_i > \beta_i/2. \end{cases}$$

- The following chain requires only 8 multiplications:

$$(0,1) \xrightarrow{\times 2\ +1} (0,3) \xrightarrow{\times 2\ +1} (0,7) \xrightarrow{+} (7,7) \xrightarrow{\times 2} (7,14) \xrightarrow{\times 2} (7,28) \xrightarrow{+} (7,35)$$

- We define:

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\beta_i - \alpha_i, \alpha_i) & \text{if } \beta_i < 2\alpha_i \\ (\alpha_i, \beta_i - \alpha_i) & \text{if } (\beta_i \in [2\alpha_i; 3\alpha_i] \text{ and } \beta_i \text{ is odd}) \text{ or } (\beta_i > 2\alpha_i \\ & \text{and } \exists\ k \text{ s.t. } \alpha_i \pmod{2^k} = \beta_i \pmod{2^k} \neq 0) \\ (\alpha_i, (\beta_i - \gamma)/2) & \text{otherwise,} \end{cases}$$

Double exponentiation

Sliding window method

- Shorter addition chains can be obtained by using window methods
  - Require pre-computations, more memory
  - It is natural extension for a single exponentiation
  - For double exponentiation: need to find an appropriate encoding
- We define:

$$(\alpha_{i+1}, \beta_{i+1}) = \begin{cases} (\alpha_i, \beta_i/2) & \text{if } \alpha_i \leq \beta_i/2 \text{ and } \beta_i \text{ is even,} \\ (\alpha_i, (\beta_i - r_i)/2^{k_i}) & \text{if } \alpha_i \leq \beta_i/2,\ \beta_i \text{ is odd,} \\ (\beta_i - \alpha_i, \alpha_i) & \text{if } \alpha_i > \beta_i/2, \end{cases}$$

- For example:, the following chain saves one multiplication

$$(0,1) \xrightarrow{\times 2\ +1} (0,3) \xrightarrow{\times 2} (0,6) \xrightarrow{+} (6,6) \xrightarrow{\times 2\ \times 2\ +3} (6,27)$$

Right-to-Left Sliding-window Double exponentiation

- In principle, it works as two parallel executions of Yao's algorithm
  - Don't require pre-computation of the chain encoding, values: m^3, m^5, …
- For example, compute (29, 50)

$$\begin{pmatrix} (a,b) \\ M \\ (A_1, A_3) \\ (B_1, B_3) \end{pmatrix} : \begin{pmatrix} (29,50) \\ m \\ (1,1) \\ (1,1) \end{pmatrix} \rightarrow \begin{pmatrix} (28,50) \\ m^2 \\ (m,1) \\ (1,1) \end{pmatrix} \rightarrow \begin{pmatrix} (28,48) \\ m^4 \\ (m,1) \\ (m^2,1) \end{pmatrix}$$

$$\rightarrow \begin{pmatrix} (16,48) \\ m^8 \\ (m,m^4) \\ (m^2,1) \end{pmatrix} \rightarrow \begin{pmatrix} (16,48) \\ m^{16} \\ (m,m^4) \\ (m^2,1) \end{pmatrix} \rightarrow \begin{pmatrix} (0,0) \\ m^{32} \\ (m^{17},m^4) \\ (m^2,m^{16}) \end{pmatrix}$$

- Then, one compute:

$m^{29} = m^{17} (m^4)^3$; and $m^{50} = m^2 (m^{16})^3$

Algorithm 3. Sliding-Window Double Exponentiation

Input: m, a, b
Output: (m^a, m^b)

1. $M \leftarrow m$
2. **for** $d \in \{1, 3, \ldots, 2^w - 1\}$ **do**
3. $\quad A_d \leftarrow 1 \;;\; B_d \leftarrow 1$
4. **end for**
5. **for** $i = 0$ **to** $\ell - 1$ **do**
6. $\quad$ **if** $(a_i = 1)$ **then**
7. $\quad\quad d \leftarrow (a_{i+w-1}, \ldots, a_{i+1}, a_i)_2$
8. $\quad\quad A_d \leftarrow A_d \cdot M$
9. $\quad\quad a \leftarrow a - 2^i d$
10. $\quad$ **endif**
11. $\quad$ **if** $(b_i = 1)$ **then**
12. $\quad\quad d \leftarrow (b_{i+w-1}, \ldots, b_{i+1}, b_i)_2$
13. $\quad\quad B_d \leftarrow B_d \cdot M$
14. $\quad\quad b \leftarrow b - 2^i d$
15. $\quad$ **endif**
16. $\quad M \leftarrow M^2$
17. **end for**
18. $A_1 \leftarrow \prod_d A_d^d$
19. $B_1 \leftarrow \prod_d B_d^d$
20. **return** (A_1, B_1)

Double exponentiation

Performance

- We make simulations for various exponent bit-length: 512, 1024, 2048, and for various window sizes
- For binary method, we improve 7%
- L2R window-based method improves 4%-8%, depending the window size
- Combined method improves 7%-9%
- R2L sliding window method improves up to 19%

Conclusion

- We revisited double exponentiation algorithms for fault analysis resistant RSA
- We introduced new variants of Rivain's heuristics for double addition chains
  - improved up to 9%
- We introduced a generalization of Yao's right to left exponentiation to perform a double exponentiation
  - improved up to 19%
  - can be extended to compute monomials: m^{d1}, m^{d2}, …, m^{dk}

Thank you !

Q & A