Recent Advances on Lightweight Cryptography Designs

Thomas Peyrin

Nanyang Technological University - Singapore

ICISC 2011

Seoul

NANYANG TECHNOLOGICAL UNIVERSITY

Outline

Outline

Lightweight crypto ?

We expect RFID tags to be deployed widely (supply chain management, e-passports, contactless applications, etc.)

- we need to ensure authentication and/or confidentiality
- a basic RFID tag may have a total gate count of anywhere from 1000-10000 gates, with **only 200-2000 gates** budgeted for security
- hardware throughput and software performances are not the most important criterias, but they must be acceptable
- in general aim for smallest possible area, good FOM (throughput/area^2), acceptable speed (hardware and software)
- block ciphers and hash functions are used as basic blocks for RFID device authentication and privacy-preserving protocols.

Lightweight hash functions ?

Standardized or SHA−3 hash functions are too big:

- MD5 (8001 GE), SHA−1 (6122 GE), SHA−2 (10868 GE)
- BLAKE (9890 GE), GRøSTL (14622 GE), JH (?), KECCAK (20790 GE), SKEIN (12890 GE)

Recently, new lightweight hash functions have been proposed (much lower than 10000 GE):

- MAME [Yoshida et al. 2007]
- DM−PRESENT and H−PRESENT [Bogdanov et al. 2008]
- ARMADILLO [Badel et al. 2010]
- QUARK [Aumasson et al. 2010]
- PHOTON [Guo et al. 2011]
- SPONGENT [Bogdanov et al. 2011]

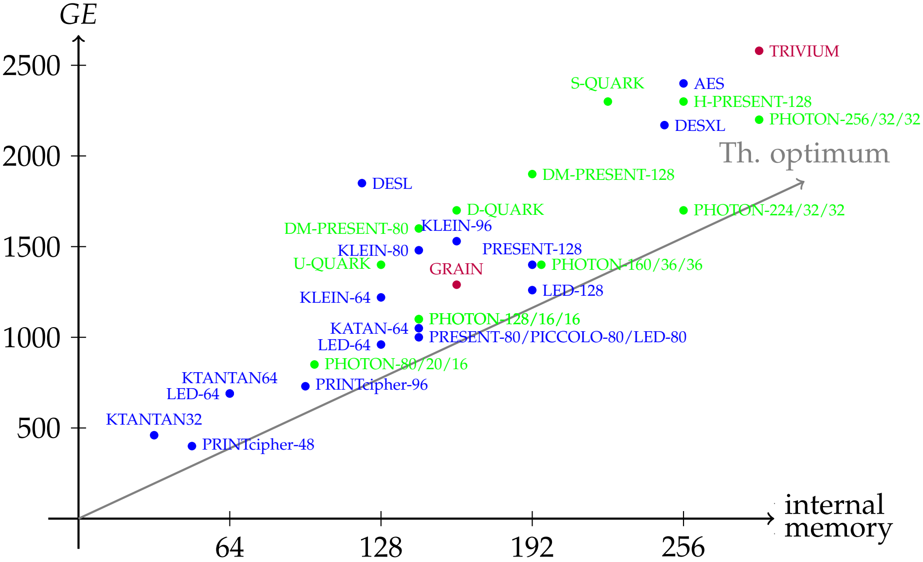
Lightweight block ciphers ?

More mature than hash functions, but are lightweight block ciphers too provocative ?

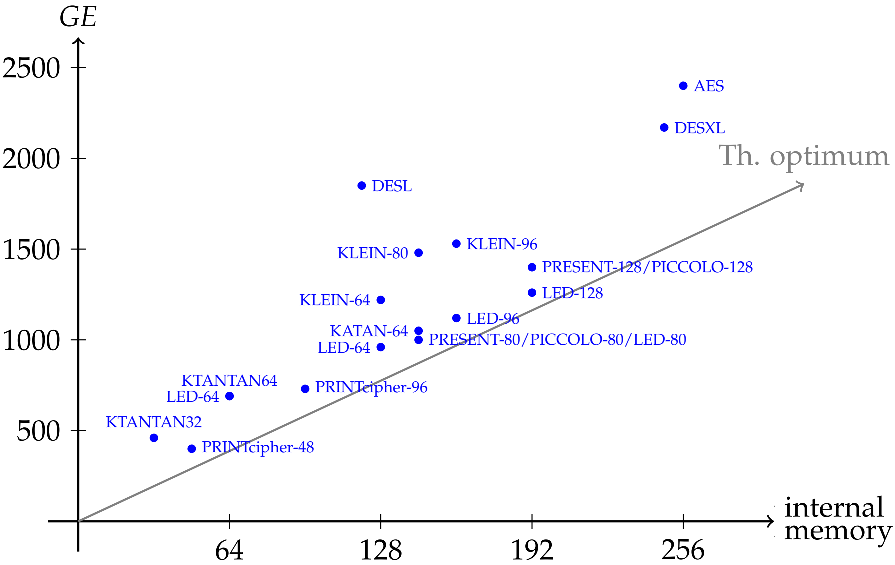
- ARMADILLO: key-recovery attacks [A+-2011]
- HIGHT: related-key attacks [K+-2010]
- Hummingbird-1: practical related-IV attacks [S-2011]
- KTANTAN: practical related-key attacks [Å-2011]
- PRINTcipher: large weak-keys classes [ÅJ-2011]

PRESENT is still unbroken.

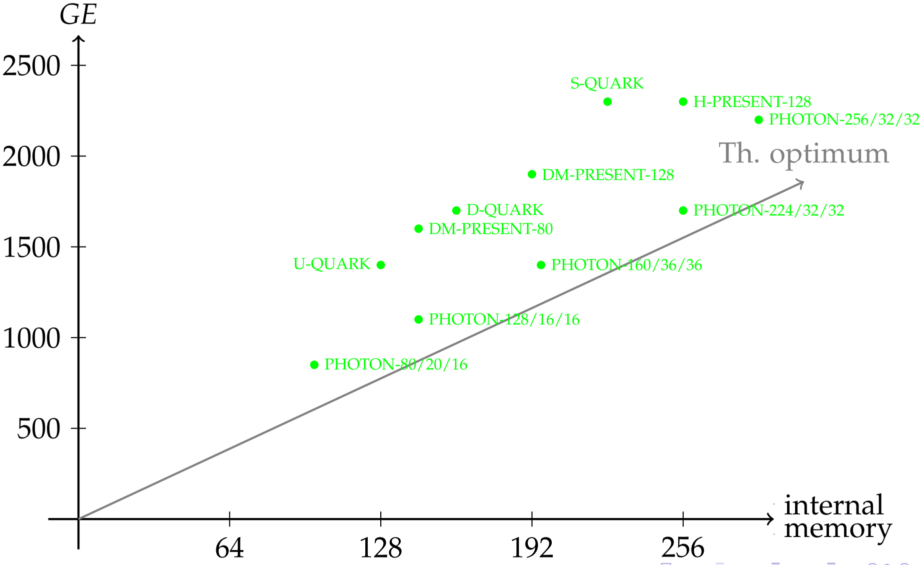
Current picture of lightweight primitives - graphically



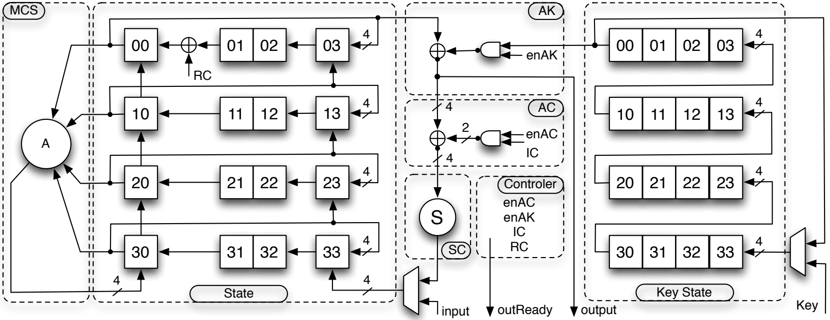
Current picture of lightweight block ciphers - graphically



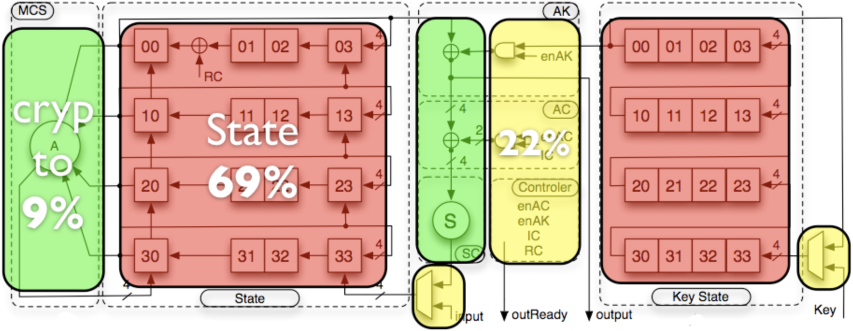
Current picture of lightweight hash functions - graphically



Lightweight ≃ low memory



Lightweight $\simeq$ low memory



Lightweight $\simeq$ low memory

The storage of one bit depends the technology, but for UMC 180nm it costs about:

- 4.67 GE for one input flip-flop
- 6 GE for two inputs flip-flop

Of course, all the security parameters will be small in order to avoid any waste of memory because of unwanted extra security:

- **block ciphers:** 64-bit block, 64 to 128-bit key
- **hash functions:** depends on security property. Can go from 64-bit hash output for preimage up to 256-bit output for collision resistance

"Security made to measure" (M. Robshaw)

Outline

Outline

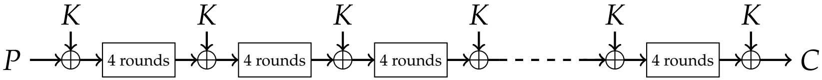
Minimizing the memory for block ciphers

Minimizing the memory for block ciphers:

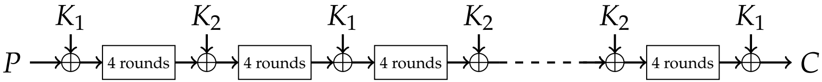
- **Key schedule:**
  - avoid complex key expansion or non-invertible key schedules !
  - use simple invertible key register update (AES, PRESENT, KATAN)
  - or subkeys simply selected from master key bits (IDEA, PICCOLO, KTANTAN)
  - or no key schedule: subkeys = master key (LED)
  - for the two last, one can hardwire the key and further save memory in some scenarios
- **Internal permutation:**
  - use general construction that allows maximal serialisation
  - avoid classical Feistel, better to use Feistel with many branches (for a light internal function F, one can use the PICCOLO trick)
  - for SPN, use small MixColumns (or use PHOTON/LED trick)

Example: LED key schedule

For **64-bit key**, the key is xored to the internal state **every four rounds**. In related-key setting, one gets at least half of the boxes active:

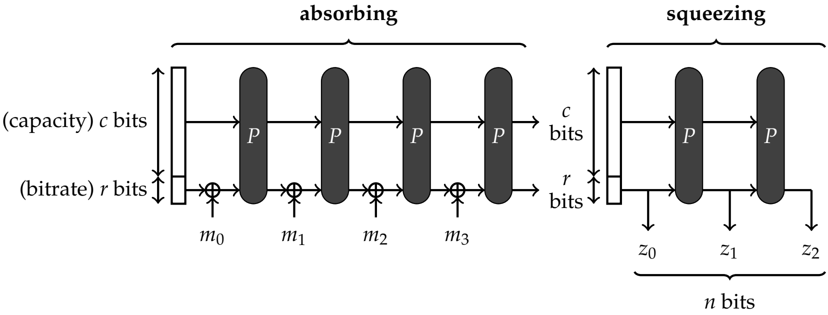


For **up to 128-bit key**, the key is divided into **two equal chunks** K_1 and K_2 that are alternatively xored to the internal state **every four rounds**. In related-key setting, one gets at least half of the boxes active:



Outline

Orginial sponge functions [Bertoni et al. 2007]



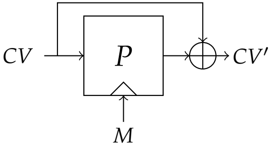
A sponge function has been proven to be indifferentiable from a random oracle up to $2^{c/2}$ calls to the internal permutation P. However, **the best known generic attacks have the following complexity:**

- **Collision:** $\min\{2^{n/2}, 2^{c/2}\}$
- **Second-preimage:** $\min\{2^{n}, 2^{c/2}\}$
- **Preimage:** $\min\{2^{\min\{n,c+r\}}, \max\{2^{\min\{n-r,c\}}, 2^{c/2}\}\}$

Sponges vs Davies-Meyer

We would like to build the smallest possible hash function with no better collision attack than generic ($2^{n/2}$ operations). Thus **we try to minimize the internal state size**:

- **in a classical Davies-Meyer compression function** using a m-bit block cipher with k-bit key, one needs to store $2m + k$ bits. We minimize the internal state size with $m \simeq n$ and k as small as possible.



- **in sponge functions**, one needs to store $c + r$ bits. We minimize the internal state size by using $c \simeq n$ and a bitrate r as small as possible.

Sponge function will require about twice less memory bits for lightweight scenarios.

Outline

Basic lightweight design tricks

- **constants:** use no constants, or at least some that are easy to generate with a LFSR (avoid pure counter)
- **non-linearity:**
  - use NLFSR (KATAN)
  - use NAND gates (KECCAK)
  - use small Sboxes (PRESENT, LED, PICCOLO...). 4-bit Sboxes seem a good compromise between size (PRESENT Sbox is about 20GE) and cryptographic quality, since a 8-bit Sbox is quite big (AES Sbox is about 230 GE)
- **diffusion:**
  - use bit position permutation branching (PRESENT): almost no diffusion (the diffusion is provided by the Sboxes), but fast and lightweight ... be carefull with hull effect
  - serially computable MDS (LED): very good diffusion, lightweight, but slow

MDS Matrix

What is an **MDS Matrix** ("Maximum Distance Separable") ?

- it is used as **diffusion layer** in many block ciphers and in particular AES
- it has excellent diffusion properties. In short, **for a d-cell vector, we are ensured that at least $d + 1$ input / output cells will be active** ...
- ... which is very good for linear / differential cryptanalysis resistance

The AES diffusion matrix can be implemented fast in software (using tables), but **the situation is not so great in hardware**. Indeed, even if the coefficients of the matrix minimize the hardware footprint, $d - 1$ **cells of temporary memory are needed for the computation**.

$$A = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

Efficient Serially Computable MDS Matrices

Idea: use a MDS matrix that can be efficiently computed in a serial way.

How to find it: build a very light matrix A and check if A^d is MDS.

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix}$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** `AES` **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Efficient Serially Computable MDS Matrices

Idea: use a MDS matrix that can be efficiently computed in a serial way.

How to find it: build a very light matrix A and check if A^d is MDS.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-4} \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \end{pmatrix} =$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** `AES` **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Efficient Serially Computable MDS Matrices

Idea: use a MDS matrix that can be efficiently computed in a serial way.

How to find it: build a very light matrix A and check if A^d is MDS.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-4} \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \end{pmatrix} = \begin{pmatrix} v_1 \\ \\ \vdots \\ \\ \\ \\ \\ \end{pmatrix}$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** `AES` **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Efficient Serially Computable MDS Matrices

Idea: use a MDS matrix that can be efficiently computed in a serial way.

How to find it: build a very light matrix A and check if A^d is MDS.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-4} \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ \\ \\ \\ \\ \end{pmatrix}$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** `AES` **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Efficient Serially Computable MDS Matrices

<u>Idea:</u> use a MDS matrix that can be efficiently computed in a serial way.

<u>How to find it:</u> build a very light matrix A and check if A^d is MDS.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-4} \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{d-3} \\ \\ \\ \\ \end{pmatrix}$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** `AES` **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Efficient Serially Computable MDS Matrices

Idea: use a MDS matrix that can be efficiently computed in a serial way.

How to find it: build a very light matrix A and check if A^d is MDS.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-4} \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{d-3} \\ v_{d-2} \\ \\ \end{pmatrix}$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** AES **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Efficient Serially Computable MDS Matrices

<u>Idea:</u> use a MDS matrix that can be efficiently computed in a serial way.

<u>How to find it:</u> build a very light matrix A and check if A^d is MDS.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-4} \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \\ \end{pmatrix}$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** `AES` **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Efficient Serially Computable MDS Matrices

Idea: use a MDS matrix that can be efficiently computed in a serial way.

How to find it: build a very light matrix A and check if A^d is MDS.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & \cdots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 & 0 & 0 & 0 \\ & \vdots & & & & & & \vdots & \\ 0 & 0 & 0 & 0 & \cdots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \cdots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{d-4} \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_{d-3} \\ v_{d-2} \\ v_{d-1} \\ v'_0 \end{pmatrix}$$

- we keep the same good diffusion properties since A^d is MDS
- **excellent in hardware (no additional memory cell needed)**
- **as good as** `AES` **in software**, we can use d lookup tables
- same coefficients for deciphering, so **the invert of the matrix is also excellent in hardware**

Tweaking AES for hardware: AES-HW

The smallest AES implementation requires 2400 GE with 263 GE dedicated to the MixColumns layer (the matrix A is MDS).

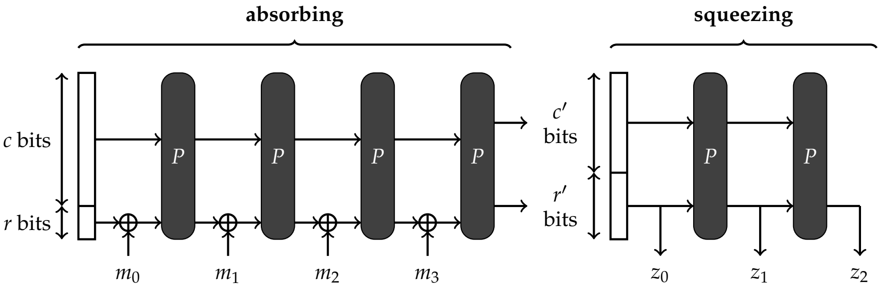
$$A = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \qquad A^{-1} = \begin{pmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{pmatrix}$$

A tweaked AES-HW **implementation** requires 2210 GE with 74 GE dedicated to the MixColumnsSerial layer (the matrix $(B)^4$ is MDS):

$$(B)^4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 2 & 1 & 4 \end{pmatrix}^4 = \begin{pmatrix} 1 & 2 & 1 & 4 \\ 4 & 9 & 6 & 17 \\ 17 & 38 & 24 & 66 \\ 66 & 149 & 100 & 11 \end{pmatrix} \qquad B^{-1} = \begin{pmatrix} 2 & 1 & 4 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Outline

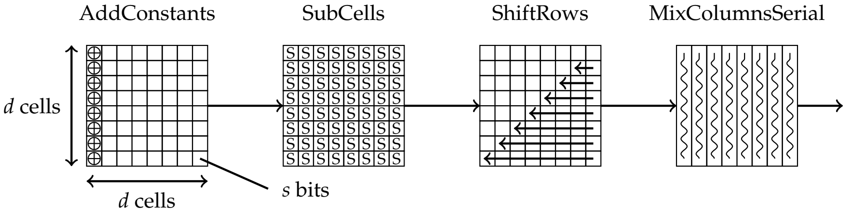
Domain extension algorithm



The $(c + r)$-bit internal state is viewed as a $d \times d$ matrix of s-bit cells.

PHOTON-$n/r/r'$		n	c	r	r'	d	s
PHOTON-80/20/16	P_{100}	80	80	20	16	5	4
PHOTON-128/16/16	P_{144}	128	128	16	16	6	4
PHOTON-160/36/36	P_{196}	160	160	36	36	7	4
PHOTON-224/32/32	P_{256}	224	224	32	32	8	4
PHOTON-256/32/32	P_{288}	256	256	32	32	6	8

Internal permutations



The internal permutations apply **12 rounds** of an AES-like fixed-key permutation:

- **AddConstants:** xor round-dependant constants to the first column
- **SubCells:** apply the PRESENT (when $s = 4$) or AES Sbox (when $s = 8$) to each cell
- **ShiftRows:** rotate the i-th line by i positions to the left
- **MixColumnsSerial:** apply the special MDS matrix to each columns

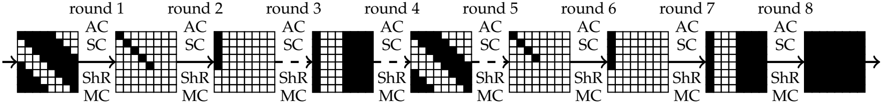
AES-like fixed-key permutation security

- AES-like permutations are simple to understand, well studied, provide very good security
- one can easily derive clear and powerful proofs on the minimal number of active Sboxes for 4 rounds of the permutation: $(d+1)^2$ **active Sboxes for 4 rounds of** PHOTON
- **we avoid any key schedule issue** since the permutations are fixed-key

	P_{100}	P_{144}	P_{196}	P_{256}	P_{288}
differential path probability	2^{-72}	2^{-98}	2^{-128}	2^{-162}	2^{-294}
differential probability	2^{-50}	2^{-72}	2^{-98}	2^{-128}	2^{-246}
linear approximation probability	2^{-72}	2^{-98}	2^{-128}	2^{-162}	2^{-294}
linear hull probability	2^{-50}	2^{-72}	2^{-98}	2^{-128}	2^{-246}

Table: Upper bounds for 4 rounds of the five PHOTON internal permutations.

Rebound attack and improvements



The currently best known technique achieves **8 rounds** for an AES-like permutation, with quite low complexity.

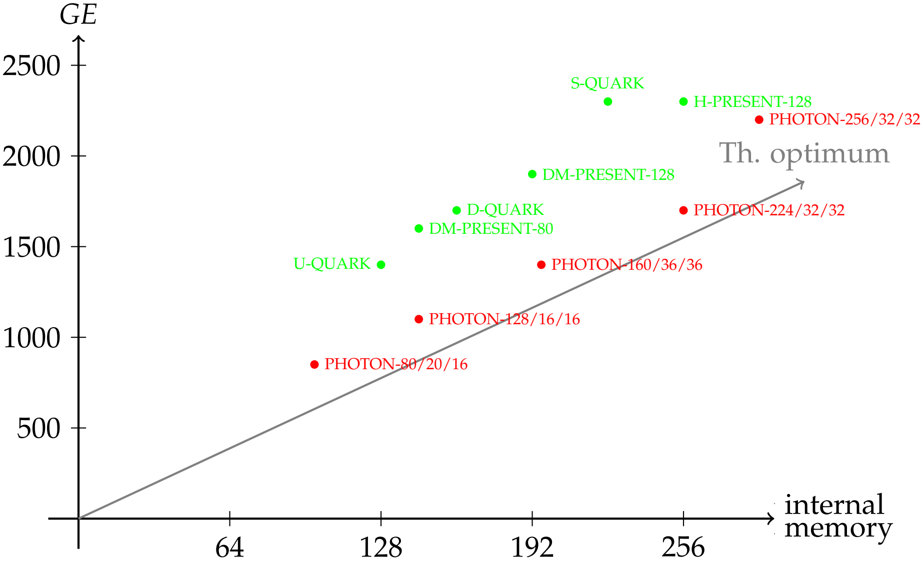
	P_{100}	P_{144}	P_{196}	P_{256}	P_{288}
computations	2^8	2^8	2^8	2^8	2^{16}
memory	2^4	2^4	2^4	2^4	2^8
generic	2^{10}	2^{12}	2^{14}	2^{16}	2^{24}

Improvements are unlikely since no key is used in the permutation, so **the amount of freedom degrees given to the attacker is limited to the minimum**.

Other cryptanalysis techniques

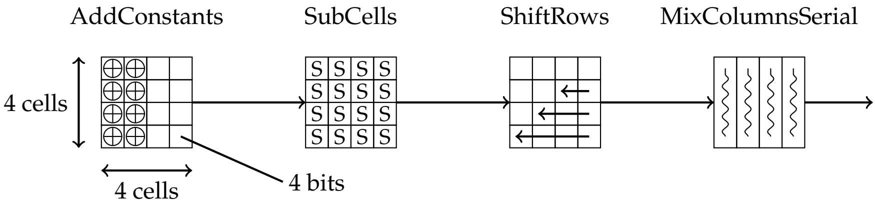
- **cube testers:** the best we could find within practical time complexity is at most 3 rounds for all PHOTON variants.
- **zero-sum partitions:** distinguishers for at most 8 rounds (for complexity $< 2^{c/2}$).
- **algebraic attacks:** the entire system for the internal permutations of PHOTON consists of $d^2 \cdot N_r \cdot \{21, 40\}$ quadratic equations in $d^2 \cdot N_r \cdot \{8, 16\}$ variables.
- **slide attacks on permutation level:** all rounds of the internal permutation are made different thanks to the round-dependent constants addition.
- **slide attacks on operating mode level:** the sponge padding rule from PHOTON forces the last message block to be different from zero.
- **rotational cryptanalysis:** any rotation property in a cell will be directly removed by the application of the Sbox layer.
- **integral attacks:** can reach 7 rounds with complexity $2^{s(2d-1)}$.

Hardware implementation results of PHOTON



Outline

A single round of LED



The 64-bit round function is an SP-network (we apply **32** to **48 rounds**):

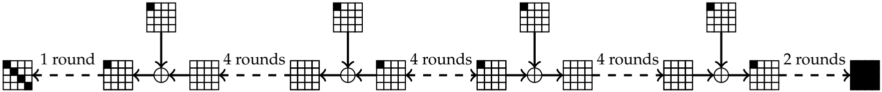
- **AddConstants:** xor round-dependent constants to the two first columns
- **SubCells:** apply the PRESENT 4-bit Sbox to each cell
- **ShiftRows:** rotate the i-th line by i positions to the left
- **MixColumnsSerial:** apply the special MDS matrix to each columns independently

Differential/linear attacks

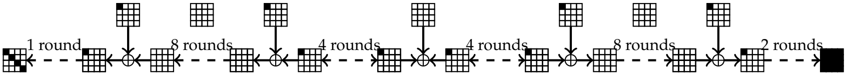
- AES**-like permutations** are simple to understand, well studied, provide very good security
- **In single-key model:** one can easily derive proofs on the minimal number of active Sboxes for 4 rounds of the permutation:
  $(d+1)^2 = 25$ **active Sboxes for 4 rounds of** LED
- **In related-key model:** we have at least half of the 4-round steps active, using the same reasoning we obtain:
  $(d+1)^2 = 25$ **active Sboxes for 8 rounds of** LED

	LED-64 SK	LED-64 RK	LED-128 SK	LED-128 RK
minimal no. of active Sboxes	200	100	300	150
differential path probability	2^{-400}	2^{-200}	2^{-600}	2^{-300}
linear approx. probability	2^{-400}	2^{-200}	2^{-600}	2^{-300}

Rebound attack and improvements



In the **chosen-related-key model**, one can distinguish **15 rounds** (over 32) of `LED`-64 with complexity 2^{16}



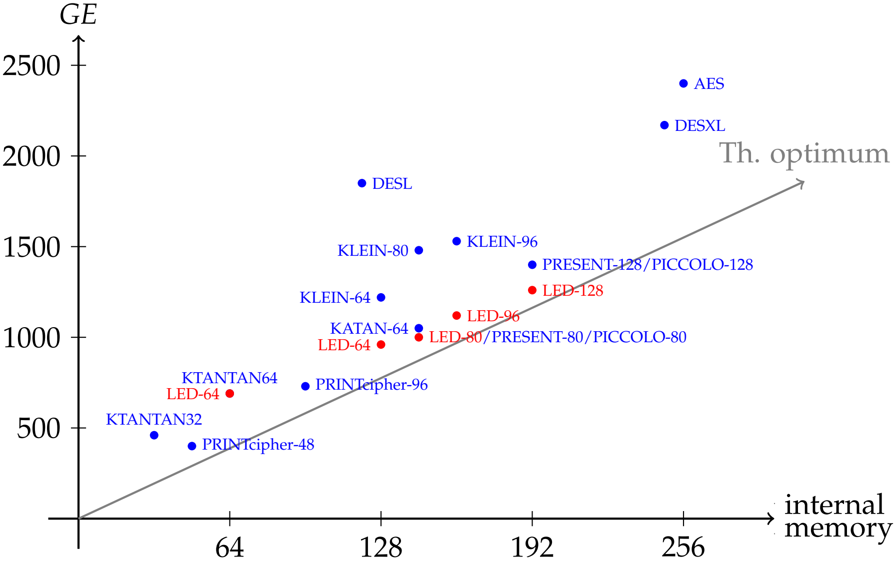
In the **chosen-related-key model**, one can distinguish **27 rounds** (over 48) of `LED`-128 with complexity 2^{16}

Improvements are unlikely since no key is used during four rounds of the permutation, so **the amount of freedom degrees given to the attacker is limited to the minimum**.

Other cryptanalysis techniques

- **cube testers:** the best we could find within practical time complexity is at most 3 rounds
- **zero-sum partitions:** distinguishers for at most 12 rounds with 2^{64} complexity in the known-key model
- **algebraic attacks:** the entire system for a 64-bit fixed-key `LED` permutation consists of 10752 quadratic equations in 4096 variables
- **slide attacks:** all rounds are made different thanks to the round-dependent constants addition
- **rotational cryptanalysis:** any rotation property in a cell will be directly removed by the application of the Sbox layer
- **integral attacks:** currently can't even break 2 steps

Hardware implementation results of LED



Software implementation results

Table: Software implementation results of LED.

	table-based implementation
LED-64	57 cycles/byte
LED-128	86 cycles/byte

One can use **"Super-Sbox" implementations** (ongoing work).

Outline

Conclusion

PHOTON **and** LED**:**

- are very **simple**, **clean** and based on the AES design strategy
- are one of the **smallest hash functions/block ciphers** (both use serially computable MDS)
- have acceptable software performances
- provide **provable security** against classical linear/differential cryptanalysis **both in the single-key and related-key models for** LED
- have a **large security margin**:
  - PHOTON: very small amount of freedom degrees given to the attacker per iteration
  - LED: security analysis done in the very optimistic known/chosen-keys model, Margin especially large in the single-key model.

PHOTON **latest results on https://sites.google.com/site/photonhashfunction/**

LED **latest results on https://sites.google.com/site/ledblockcipher/**

Future Works

- **cryptanalysis !**
- **other aims** than minimal area are possible: high throughput, energy consumption, a little bit everything, ...
- better **key schedule**: can we find key schedules that provably closes the gap between single-key and related-key models ?
- better **MDS matrices**: can we find matrices that offer good diffusion (maybe not MDS), with hardware-friendly serial decomposition (maybe not fully serial), and with less clock cycles ... find the best tradeoff.

Thank you for your attention !