The PHOTON Family of Lightweight Hash Functions

Jian Guo, Thomas Peyrin and Axel Poschmann

I2R and NTU

Coding, Cryptology and Combinatorial Designs

Singapore

Institute for Infocomm Research
A*STAR

NANYANG TECHNOLOGICAL UNIVERSITY

Outline

Outline

Lightweight hash functions

Why do we need lightweight hash functions ?

- RFID device authentication and privacy
- **in most of the privacy-preserving RFID protocols proposed, a hash function is required**
- 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

Current picture

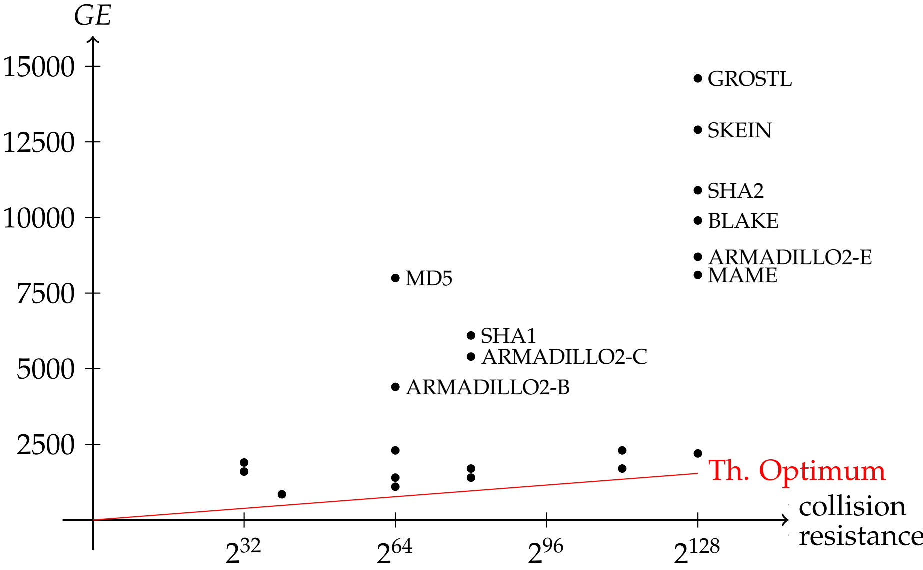
Standardized or SHA−3 hash functions are too big:

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

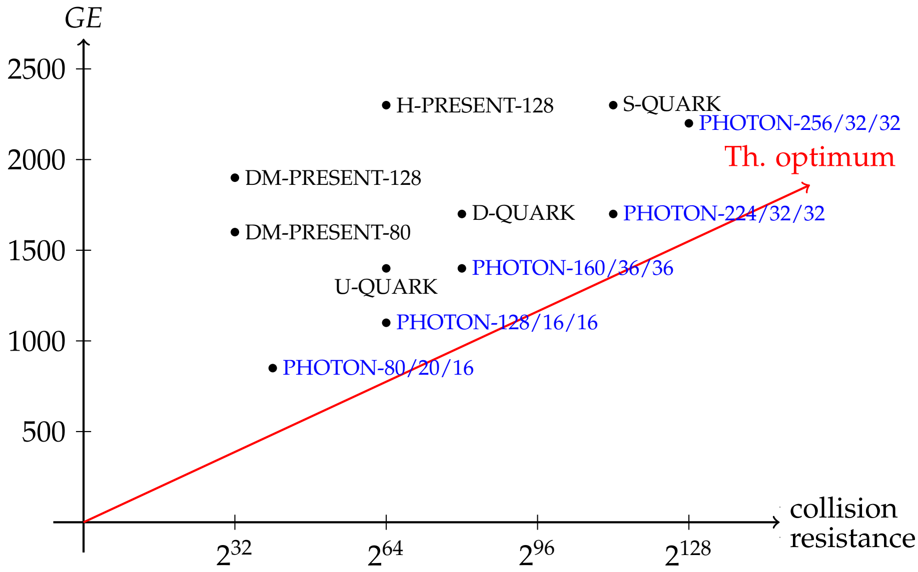
Recently, new lightweight hash functions have been proposed:

- SQUASH (2646 GE) [Shamir 2005]
- MAME (8100 GE) [Yoshida et al. 2007]
- DM−PRESENT (1600 GE) and H−PRESENT (2330 GE) [Bogdanov et al. 2008]
- ARMADILLO (4353 GE) [Badel et al. 2010]
- QUARK (1379 GE) [Aumasson et al. 2010]

Current picture - graphically

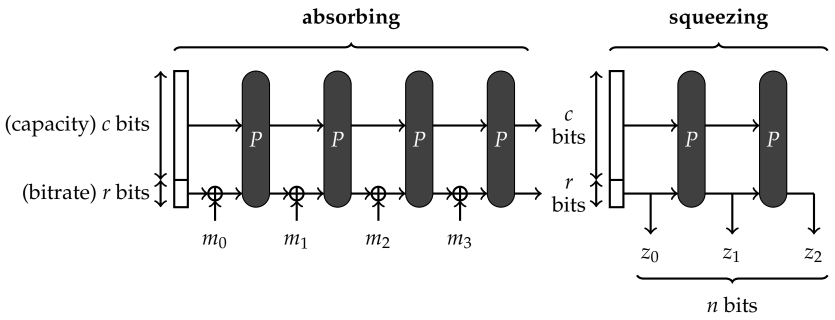


Current picture - graphically



Outline

Orginial sponge functions [Bertoni et al. 2007]



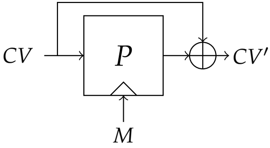
A sponge function has been proven to be indifferentiable from a random oracle up to $2^{c/2}$ calls to the internal permutation P. However, **the best known generic attacks have the following complexity:**

- **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**:

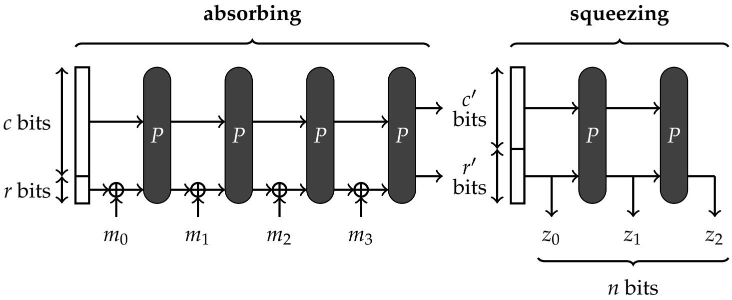
- **in a classical Davies-Meyer compression function** using a m-bit block cipher with k-bit key, one needs to store $2m + k$ bits. We minimize the internal state size with $m \simeq n$ and k as small as possible.



- **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.

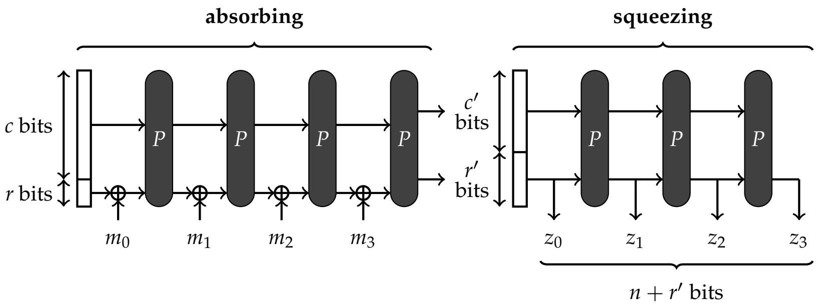
Generalization 1



Sponges with small r are slow for small messages (which is a typical usecase for lightweight applications, as an example EPC is 96 bit long). Thus **we can allow the output bitrate r' to be different from the input bitrate r** and obtain a preimage security / small message speed tradeoff:

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

Generalization 2



Sponges with $c \simeq n$ are not n-bit preimage resistant (often only preimage resistance is needed for lightweight applications). Thus **we can allow for bigger outputs by adding an extra squeezing step** and increase the preimage security:

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

Outline

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

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.

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

- 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.

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

- 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.

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

- 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).

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

Our 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

Introduction and Motivation

Generalized Sponge Construction

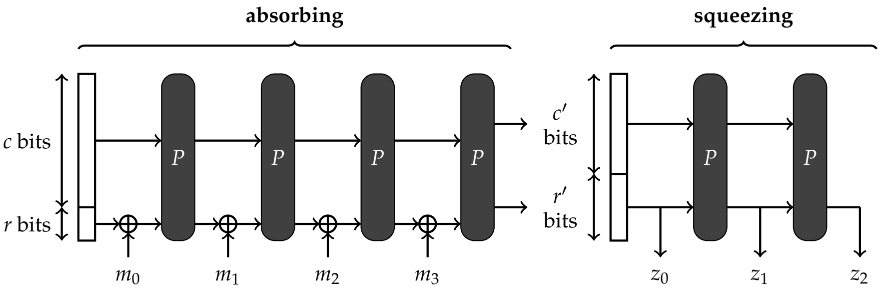
Efficient Serially Computable MDS Matrices

The PHOTON Family of Lightweight Hash Functions

The Security of PHOTON

Conclusion and Future Works

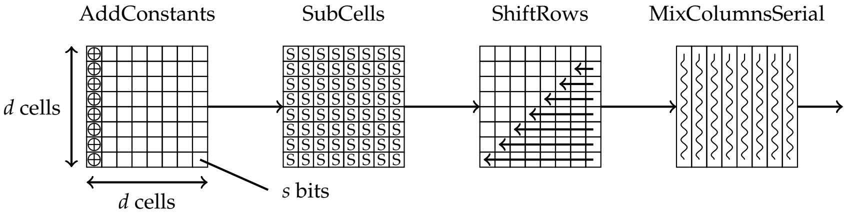
Domain extension algorithm



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

Outline

Extended sponge claims

Our security claims (a little bit more than flat sponge claims):

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

For the security proofs, the internal permutation is modeled as a random permutation:

- the problem is reduced to studying the quality of the PHOTON internal permutations
- hermetic sponge-like strategy: it is assumed that the internal permutations have no structural flaw, up to $2^{c/2}$ operations
- even if one finds a structural flaw for the internal permutations, it is unlikely to turn it into an attack ...
- ... **this is particularily true for** PHOTON **which has a very small bitrate** (i.e. the attacker has in practice a very small amount of freedom degrees in order to use the distinguisher).

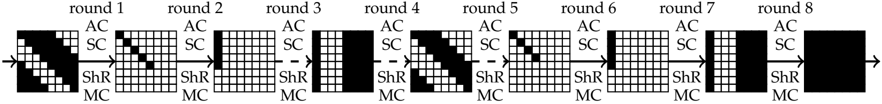
AES-like fixed-key permutation security

- 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 $2^{s(2d-1)}$.

Outline

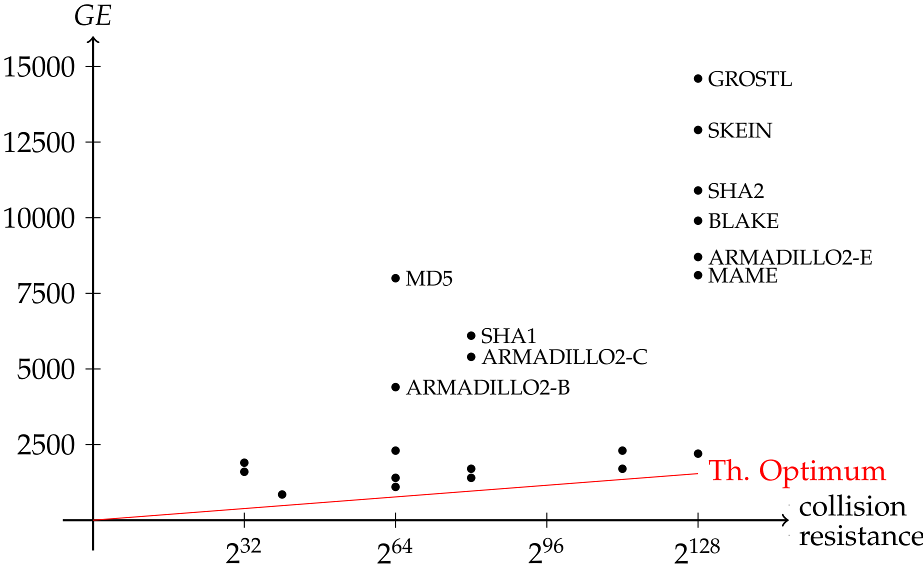
Hardware implementations

Name	Security		Performance						
	Pre	Col	**Area [GE]**	Latency [clk]		Throughput [kbps]		FOM [nb/clk/GE^2]	
				P/E	H	long	96-bit	long	96-bit
64-bit security (preimage only)									
SQUASH	64	0	**2646**	31800	31800	0.2	0.15	0.29	0.14
DM-PRESENT-80	64	32	**1600**	547	547	14.63	8.78	57.13	28.56
DM-PRESENT-80	64	32	**2213**	33	33	242.42	145.45	495.01	247.50
DM-PRESENT-128	64	32	**1886**	559	559	22.90	22.90	64.37	64.37
DM-PRESENT-128	64	32	**2530**	33	33	387.88	387.88	605.98	605.98
PHOTON-80/20/16	64	40	**865**	708	3540	2.82	1.51	37.73	20.12
PHOTON-80/20/16	64	40	**1168**	132	660	15.15	8.08	111.13	59.27
64-bit security									
U-QUARK	120	64	**1379**	544	8704	1.47	0.63	7.73	3.31
U-QUARK	120	64	**2392**	68	1088	11.76	5.04	20.56	8.81
H-PRESENT-128	128	64	**2330**	559	559	11.45	8.59	21.09	15.82
H-PRESENT-128	128	64	**4256**	32	32	200.00	150.00	110.41	82.81
ARMADILLO2-B	128	64	**4353**	256	256	25.00	18.75	13.19	9.90
ARMADILLO2-B	128	64	**6025**	64	64	100.00	75.00	27.55	20.66
PHOTON-128/16/16	112	64	**1122**	996	7968	1.61	0.69	12.78	5.48
PHOTON-128/16/16	112	64	**1708**	156	1248	10.26	4.4	35.15	15.06

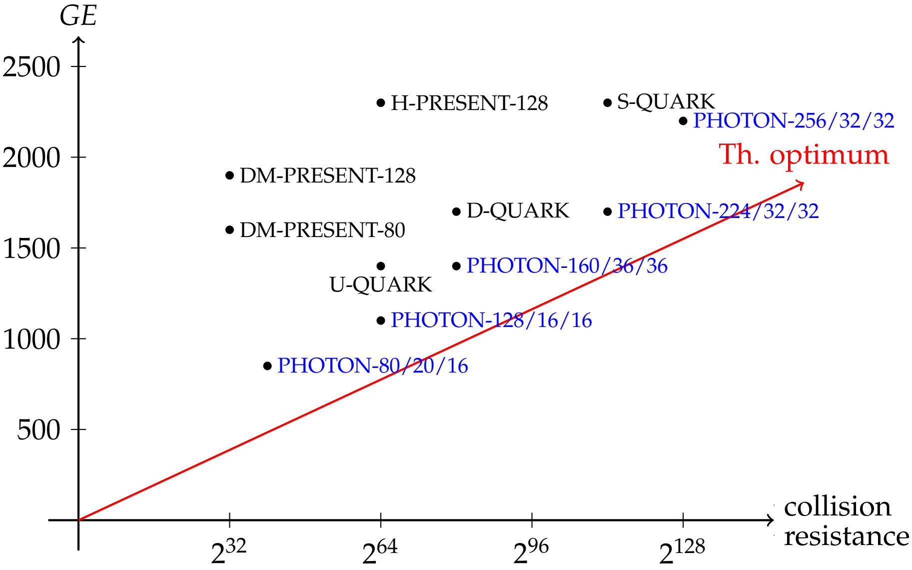
Hardware implementations

Name	Security		Performance						
	Pre	Col	**Area [GE]**	Latency [clk]		Throughput [kbps]		FOM [nb/clk/GE^2]	
				P/E	H	long	96-bit	long	96-bit
80-bit security									
D-QUARK	144	80	**1702**	704	7040	2.27	0.85	7.85	2.94
D-QUARK	144	80	**2819**	88	880	18.18	6.42	22.88	8.58
ARMADILLO2-C	160	80	**5406**	320	320	25.00	15.00	8.55	5.13
ARMADILLO2-C	160	80	**7492**	80	80	100.00	60.00	17.82	10.69
PHOTON-160/36/36	124	80	**1396**	1332	6660	2.70	1.03	13.87	5.28
PHOTON-160/36/36	124	80	**2117**	180	900	20	7.62	44.64	17.01
112-bit security									
S-QUARK	192	112	**2296**	1024	7168	3.13	0.94	5.93	1.78
S-QUARK	192	112	**4640**	64	448	50.00	15.00	23.22	6.97
PHOTON-224/32/32	192	112	**1736**	1716	12012	1.86	0.56	6.19	1.86
PHOTON-224/32/32	192	112	**2786**	204	1428	15.69	4.71	20.21	6.06
128-bit security									
ARMADILLO2-E	256	128	**8653**	512	0	25.00	18.75	3.34	2.50
ARMADILLO2-E	256	128	**11914**	128	0	100.00	75.00	7.05	5.28
PHOTON-256/32/32	224	128	**2177**	996	7968	3.21	0.88	6.78	1.85
PHOTON-256/32/32	224	128	**4362**	156	1248	20.51	5.59	10.78	2.94

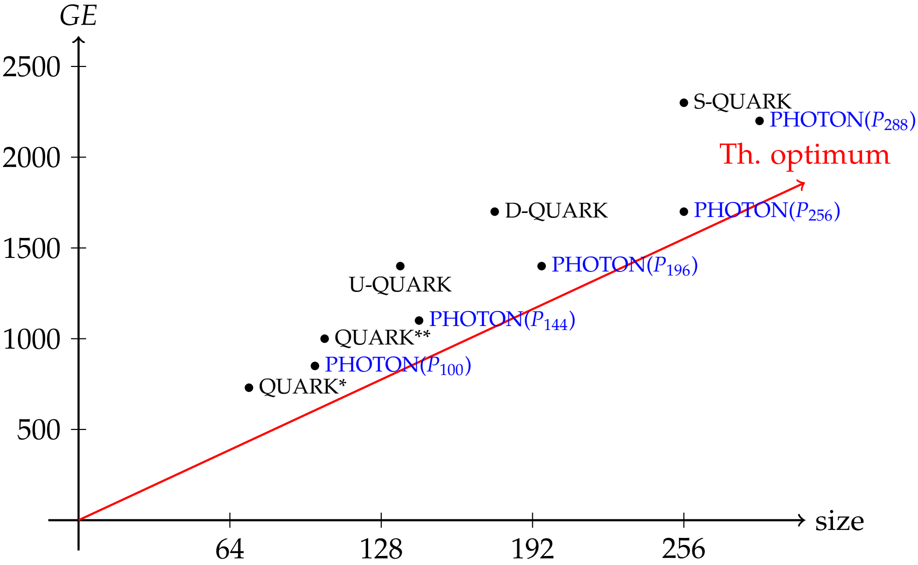
Current picture - graphically



Current picture - graphically



A fair area comparison for sponge-based lightweight hash functions



Software implementations

hash function	software speed (c/B)
PHOTON-80/20/16	95
PHOTON-128/16/16	156
PHOTON-160/36/36	116
PHOTON-224/32/32	227
PHOTON-256/32/32	157

Benchmarks done on an Intel(R) Core(TM) i7 CPU Q 720 cadenced at 1.60GHz

Conclusion

The PHOTON **family of hash functions**

- is very **simple**, clean, based on the AES design strategy
- are the **smallest hash functions** known so far
- provides acceptable software performances
- provides **provable security** against classical linear/differential cryptanalysis, and resists all known and recent attacks against hash functions with a large security margin.

Latest results on https://sites.google.com/site/photonhashfunction/

Future works

`LED` **(Light Encryption Device) is a 64-bit block cipher:**

- can take any key size up to 128 bits
- reuses the serial MDS matrix idea
- is **slightly smaller than** `PRESENT` **in hardware**
- is **"only" about three time slower than** `AES` **in software**
- provides **provable security** against classical linear/differential cryptanalysis ...
- ... **both in single-key and related-key model**