The `PHOTON` Family of Lightweight Hash Functions

Jian Guo, Thomas Peyrin, Axel Poschmann

CRYPTO 2011, 15 August 2011

Institute for Infocomm Research
A*STAR

NANYANG TECHNOLOGICAL UNIVERSITY

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 Following Work

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 Following Work

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

Main goal of `PHOTON`**:**

- minimize the hardware footprint
- hardware throughput and software performances are not the most important criteria, but they must be acceptable

Current picture

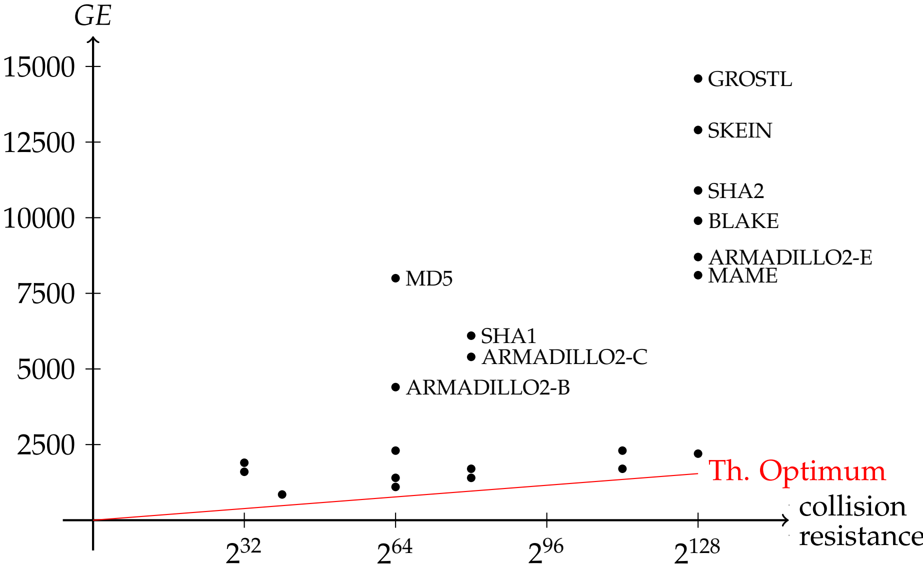
Standardized or `SHA-3` hash functions are too big:

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

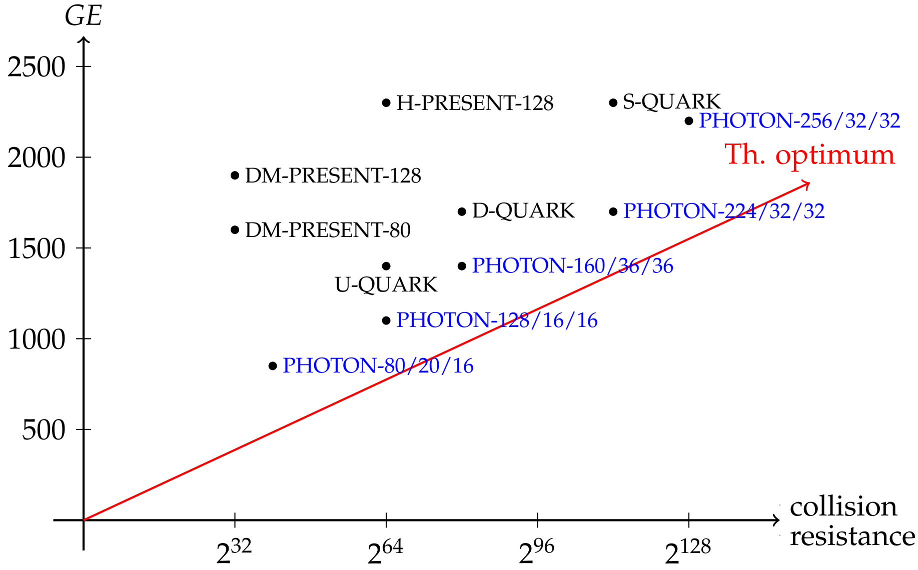
Recently, new lightweight hash functions have been proposed:

- `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

Introduction and Motivation

Generalized Sponge Construction

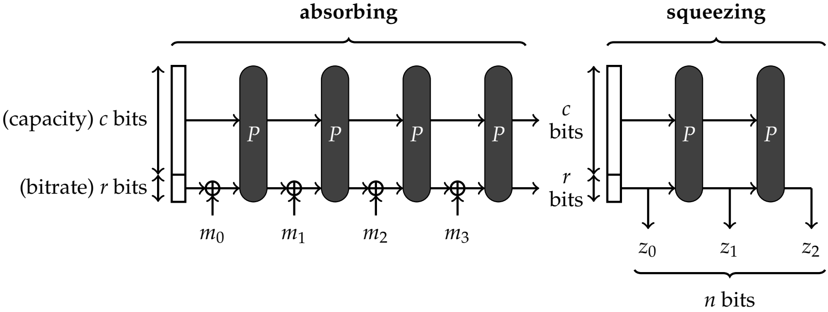
Efficient Serially Computable MDS Matrices

The `PHOTON` Family of Lightweight Hash Functions

The Security of `PHOTON`

Conclusion and Following Work

Original sponge functions [Bertoni et al. 2007]



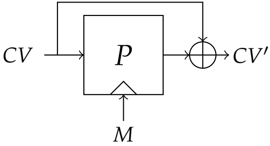
A sponge function has been proven to be indifferentiable from a random oracle up to $2^{c/2}$ calls to the internal permutation P. However, **the best known generic attacks have the following complexity (fix $c = n$):**

- **Collision:** $2^{n/2}$
- **Second-preimage:** $2^{n/2}$
- **Preimage:** 2^{n-r}

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**:

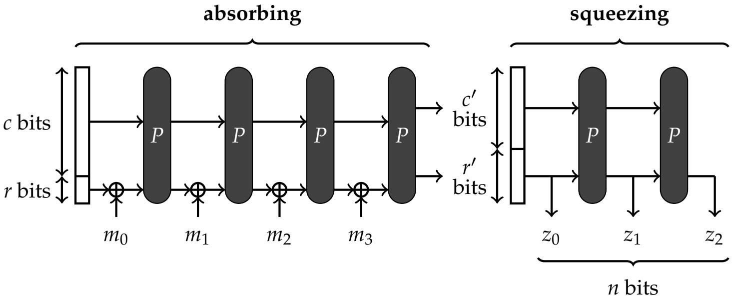
- **in a classical Davies-Meyer compression function** using a n-bit block cipher with k-bit key, one needs to store $2n + k$ bits.



- **in sponge functions**, one needs to store $n + r$ bits.

Sponge function will require about half memory bits for lightweight scenarios.

Generalization



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:** $\min\{2^{\min\{n,b\}}, \max\{2^{\min\{n,b\}-r'}, 2^{c/2}\}\}$ with $b = r + c$.

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 Following Work

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**.

$$v' = A \cdot v = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{pmatrix}$$

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

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

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

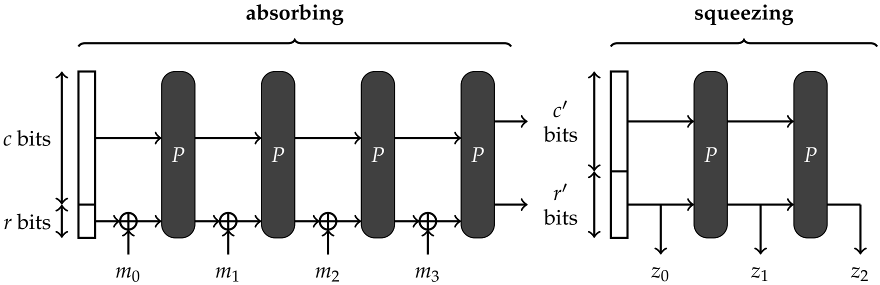
Efficient Serially Computable MDS Matrices

The `PHOTON` Family of Lightweight Hash Functions

The Security of `PHOTON`

Conclusion and Following Work

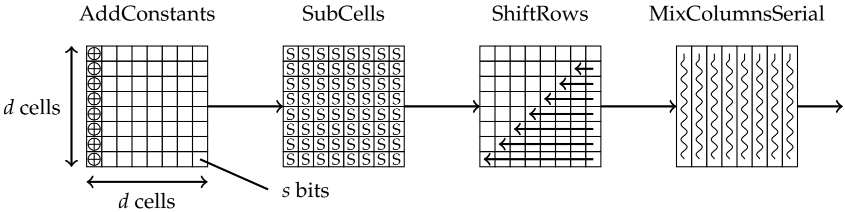
Domain extension algorithm



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

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

Introduction and Motivation

Generalized Sponge Construction

Efficient Serially Computable MDS Matrices

The PHOTON Family of Lightweight Hash Functions

The Security of PHOTON

Conclusion and Following Work

Extended sponge claims

Our security claims:

- **Collision:** $2^{n/2}$
- **Second-preimage:** $2^{n/2}$
- **Preimage:** $2^{n-r'}$

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}

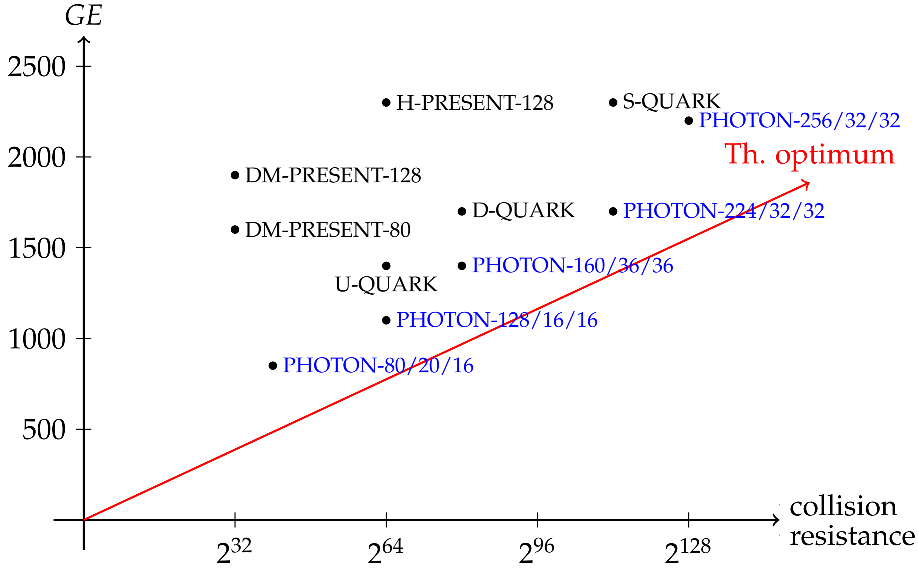
Table: Upper bounds for 4 rounds of the five `PHOTON` internal permutations.

Other cryptanalysis techniques & results

- **rebound attack:** distinguishers for at most 8 rounds with complexity 2^8 or 2^{16}.
- **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 $d^2 \cdot 12 \cdot \{21, 40\}$ quadratic equations in $d^2 \cdot 12 \cdot \{8, 16\}$ 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

Current picture - graphically



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	135

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** published 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/

Following Work

`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**

To appear in CHES 2011
Latest results on https://sites.google.com/site/ledblockcipher/

Thank you!

Questions?