PHOTON-Beetle Authenticated Encryption and Hash Family

Designers/Submitters:

Zhenzhen Bao - Nanyang Technological University, Singapore
Avik Chakraborti - University of Exeter, UK
Nilanjan Datta - Indian Statistical Institute, Kolkata, India
Jian Guo - Nanyang Technological University, Singapore
Mridul Nandi - Indian Statistical Institute, Kolkata, India
Thomas Peyrin - Nanyang Technological University, Singapore
Kan Yasuda - NTT Secure Platform Laboratories, Japan

zzbao@ntu.edu.sg, avikchkrbrti@gmail.com, nilanjan_isi_jrf@yahoo.com,
guojian@ntu.edu.sg, mridul.nandi@gmail.com, thomas.peyrin@ntu.edu.sg,
yasuda.kan@lab.ntt.co.jp

May 17, 2021

Chapter 1

Introduction

In this document, we propose PHOTON-Beetle, an authenticated encryption and hash family, that uses a sponge-based mode Beetle with the P_{256} (used for the PHOTON hash [6]) being the underlying permutation. We denote this permutation by PHOTON_{256}. Based on the functionalities, PHOTON-Beetle can be classified into two categories: a family of authenticated encryptions, dubbed as PHOTON-Beetle-AEAD and a family of hash functions, dubbed as PHOTON-Beetle-Hash. Both these families are parameterized by r, the rate of message absorption.

1.1 Notations

Here we introduce all the required notations. By $\{0,1\}^*$ we denote the set of all strings, and by $\{0,1\}^n$ the set of strings of length n. $|A|$ denotes the number of the bits in the string A. We use the notation $\oplus$ and $\odot$ to refer the binary addition and matrix multiplication respectively. For $A, B \in \{0,1\}^*$, $A\|B$ to denotes the concatenation of A and B. We use the notation $V_1\|\cdots\|V_v \xleftarrow{(a_1,\ldots,a_v)} V$ to denote parsing of the string V into v vectors of size $a_1, \ldots, a_v$ respectively. When $a_1 = \cdots = a_{v-1} = a$ and $a_v \leq a$, we simply use $V_1\|\cdots\|V_v \xleftarrow{a} V$. $B \ggg k$ denotes k bit right-rotation of the bit string B. The expression $\mathcal{E}?\ a : b$ evaluates to a if $\mathcal{E}$ holds and b otherwise. Similarly, $(\mathcal{E}_1 \texttt{ and } \mathcal{E}_2)?\ a : b : c : d$ evaluates to a if both $\mathcal{E}_1$ and $\mathcal{E}_2$ holds, b if only $\mathcal{E}_1$ holds, c if only $\mathcal{E}_2$ holds and d otherwise. $\mathsf{Trunc}(V, i)$ is a function that returns the most significant i bits of the V and Ozs is the function that applies $10^\star$ padding on r bits, i.e. $\mathsf{Ozs}_r(V) = V\|1\|0^{r-|V|-1}$ when $|V| < r$. For any two integers m and n, we use $m|\ n$ to denote that m divides n. and For any matrix X, we use the notation $X[i,j]$ to denote the element at i-th row and j-th column of X. We represent a serial matrix $\mathsf{Serial}[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7]$ by

$$\mathsf{Serial}[a_0, a_1, a_2, a_3, a_4, a_5, a_6, a_7] := \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ a_0 & a_1 & a_2 & a_3 & a_4 & a_5 & a_6 & a_7 \end{pmatrix}.$$

1.2 Organization

Here we provide a brief organization of this write-up. We revisit and provide a brief description of PHOTON_{256} that we will use in our mode as the underlying permutation in Chapter 2. We provide the complete formal specification of PHOTON-Beetle family of authenticated encryption and hash family and the recommended versions in Chapter 3. In Chapter 4, we provide the security claims of our proposals with proper justification. Finally, we detail our design decisions in Chapter 5.

Chapter 2

$\texttt{PHOTON}_{256}$ Permutation

We use $\texttt{PHOTON}_{256}$ [6] as the underlying 256-bit permutation in our mode. It is applied on a state of 64 elements of 4 bits each, which is represented as a (8×8) matrix X. $\texttt{PHOTON}_{256}$ is composed of 12 rounds, each containing four layers AddConstant, SubCells, ShiftRows and MixColumnSerial. Informally, AddConstant adds fixed constants to the cells of the internal state. SubCells applies an 4-bit S-Box (see Table. 2.1) to each of the 64 4-bit cells. ShiftRows rotates the position of the cells in each of the rows and MixColumnSerial linearly mixes all the columns independently using a serial matrix multiplication. The multiplication with the coefficients in the matrix is in $GF(2^4)$ with $x^4 + x + 1$ being the irreducible polynomial.

Table 2.1: The PHOTON S-box

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S-box	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Formal description of all these operations are given in Fig. 2.1.

$\texttt{PHOTON}_{256}(X)$

for $i = 0$ to 11 :
    $X \leftarrow \mathsf{AddConstant}(X, i)$;
    $X \leftarrow \mathsf{SubCells}(X)$;
    $X \leftarrow \mathsf{ShiftRows}(X)$;
    $X \leftarrow \mathsf{MixColumnSerial}(X)$;
return X;

$\mathsf{SubCells}(X)$

for $i = 0$ to $7, j = 0$ to 7 :
    $X[i, j] \leftarrow \mathsf{S\text{-}Box}(X[i, j])$;
return X;

$\mathsf{ShiftRows}(X)$

for $i = 0$ to $7, j = 0$ to 7 :
    $X'[i, j] \leftarrow X[i, (j + i)\%8]$;
return X';

$\mathsf{AddConstant}(X, k)$

$RC[12] \leftarrow \{1, 3, 7, 14, 13, 11, 6, 12, 9, 2, 5, 10\}$;
$IC[8] \leftarrow \{0, 1, 3, 7, 15, 14, 12, 8\}$;
for $i = 0$ to 7 :
    $X[i, 0] \leftarrow X[i, 0] \oplus RC[k] \oplus IC[i]$;
return X;

$\mathsf{MixColumnSerial}(X)$

$M \leftarrow \mathsf{Serial}[2, 4, 2, 11, 2, 8, 5, 6]$;
$X \leftarrow M^8 \odot X$;
return X;

Figure 2.1: $\texttt{PHOTON}_{256}$ Permutation.

Chapter 3

Specification of PHOTON-Beetle Family

In this chapter, we provide a formal specification of PHOTON-Beetle that includes a family of authenticated encryption PHOTON-Beetle-AEAD and a family of hash functions PHOTON-Beetle-Hash. Before going into the details, we first introduce a mathematical component that we will use.

3.1 Mathematical Component ρ and ρ^{-1}

ρ is a linear function that receives as input a state $S \in \{0,1\}^r$ and an input data $U \in \{0,1\}^{\leq r}$. It produces an output data $V \in \{0,1\}^{|U|}$ using the simple xor operation of the shuffled state and the input data (padded with zeros to an r bit block), and then updates the state S by xoring it with the input data. By shuffled state, we mean shuffling of the state bits in some order such that both $S \to \mathsf{Shuffle}(S)$ and $S \to \mathsf{Shuffle}(S) \oplus S$ are linear functions with rank r and $r-1$ respectively. ρ^{-1} is the inverse function of ρ, which takes the state S and the output data V to reproduce the input data U and update the state. Formal description of ρ and ρ^{-1} can be found in Fig. 2.1.

$\rho(S, U)$

$V \leftarrow \mathsf{Trunc}(\mathsf{Shuffle}(S), |U|) \oplus U;$
$S \leftarrow S \oplus \mathsf{Ozs}_r(U);$
return $(S, V);$

$\rho^{-1}(S, V)$

$U \leftarrow \mathsf{Trunc}(\mathsf{Shuffle}(S), |V|) \oplus V;$
$S \leftarrow S \oplus \mathsf{Ozs}_r(U);$
return $(S, U);$

$\mathsf{Shuffle}(S)$

$S_1 \| S_2 \xleftarrow{r/2} S;$
return $S_2 \| (S_1 \ggg 1);$

Figure 3.1: Mathematical Component: ρ and ρ^{-1}.

3.2 PHOTON-Beetle-AEAD Authenticated Encryption

PHOTON-Beetle-AEAD.ENC$[r]$ authenticated encryption takes an encryption key $K \in \{0,1\}^{128}$, a nonce $N \in \{0,1\}^{128}$, an associated data $A \in \{0,1\}^*$ and a message $M \in \{0,1\}^*$ as inputs and returns a ciphertext $C \in \{0,1\}^{|M|}$ and a tag $T \in \{0,1\}^{128}$. Corresponding decryption algorithm PHOTON-Beetle-AEAD.DEC$[r]$ takes a key $K \in \{0,1\}^{128}$, a nonce $N \in \{0,1\}^{128}$, an associated data $A \in \{0,1\}^*$, a ciphertext $C \in \{0,1\}^*$ and a tag $T \in \{0,1\}^{128}$ as inputs and returns the plaintext $M \in \{0,1\}^{|C|}$ corresponding to C if the tag T is verified. The parameter r signifies the rate of message absorption.

In PHOTON-Beetle-AEAD.ENC$[r]$, first an initial state is generated by simple concatenation of the nonce N and the key K. Next we process the associated data A identically to the original sponge mode i.e. at each step the state is updated using $\mathtt{PHOTON}_{256}$ and the first r bits (i.e. the rate part) of the permutation output is xored with the next associated data block to define the rate part of the next input for the next permutation call.

After A is processed, we process M in a similar way. To generate the ciphertext block, we shuffle the rate part of the permutation output and then xor it with the corresponding message block. This step differentiates

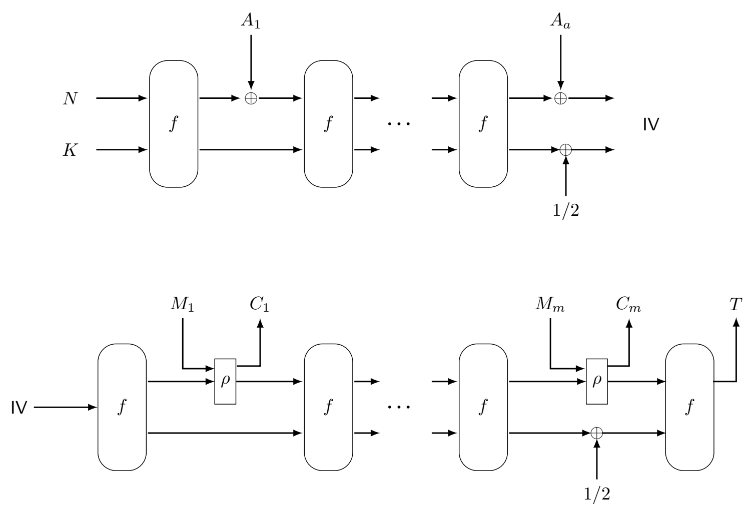

Figure 3.2: PHOTON-Beetle-AEAD.ENC with a AD blocks and m message blocks.

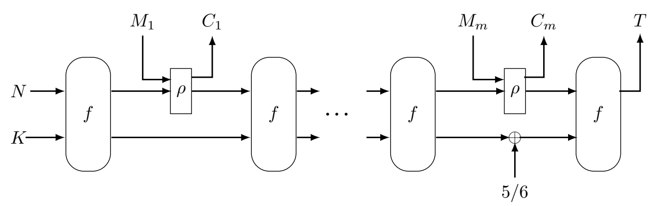

Figure 3.3: PHOTON-Beetle-AEAD.ENC with empty AD and m message blocks.

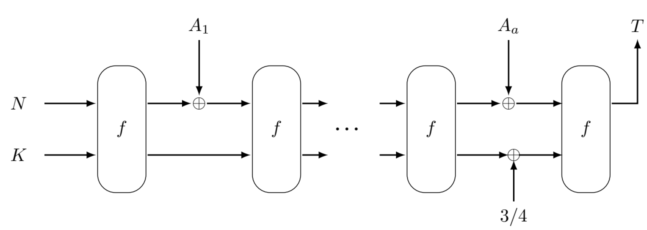

Figure 3.4: PHOTON-Beetle-AEAD.ENC Construction with a AD blocks and empty message.

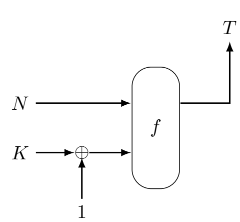

Figure 3.5: PHOTON-Beetle-AEAD.ENC Construction with empty AD and empty message.

our mode from the Sponge Duplex where the rate part of the next input to the permutation itself is released as the ciphertext block. This state update and the ciphertext generation during the message processing is captured by the function ρ. During decryption, the state update and the message block computation

$\mathsf{PHOTON\text{-}Beetle\text{-}AEAD.ENC}[r](K, N, A, M)$

$\mathsf{IV} \leftarrow N \| K;\ C \leftarrow \lambda;$
$\texttt{if}\ A = \lambda, M = \lambda:$
$\quad T \leftarrow \mathsf{TAG}_{128}(\mathsf{IV} \oplus 1);\ \texttt{return}\ (\lambda, T);$
$c_0 \leftarrow (M \neq \lambda\ \texttt{and}\ r|\ |A|)?\ 1:2:3:4$
$c_1 \leftarrow (A \neq \lambda\ \texttt{and}\ r|\ |M|)?\ 1:2:5:6$
$\texttt{if}\ A \neq \lambda:$
$\quad \mathsf{IV} \leftarrow \mathsf{HASH}_r(\mathsf{IV}, A, c_0);$
$\texttt{if}\ M \neq \lambda:$
$\quad M_1 \| \cdots \| M_m \xleftarrow{r} M;$
$\quad \texttt{for}\ i = 1\ \texttt{to}\ m:$
$\quad\quad Y \| Z \xleftarrow{(r, 256-r)} \mathtt{PHOTON}_{256}(\mathsf{IV});$
$\quad\quad (W, C_i) \leftarrow \rho(Y, M_i);$
$\quad\quad \mathsf{IV} \leftarrow W \| Z;$
$\quad \mathsf{IV} \leftarrow \mathsf{IV} \oplus c_1;$
$\quad C \leftarrow C_1 \| \cdots \| C_m;$
$T \leftarrow \mathsf{TAG}_{128}(\mathsf{IV});$
$\textbf{return}\ (C, T);$

$\mathsf{PHOTON\text{-}Beetle\text{-}AEAD.DEC}[r](K, N, A, C, T)$

$\mathsf{IV} \leftarrow N \| K;\ M \leftarrow \lambda;$
$\texttt{if}\ A = \lambda, C = \lambda:$
$\quad T^\star \leftarrow \mathsf{TAG}_{128}(\mathsf{IV} \oplus 1);$
$\quad \texttt{return}\ (T = T^\star)?\ \lambda : \perp;$
$c_0 \leftarrow (C \neq \lambda\ \texttt{and}\ r|\ |A|)?\ 1:2:3:4$
$c_1 \leftarrow (A \neq \lambda\ \texttt{and}\ r|\ |C|)?\ 1:2:5:6$
$\texttt{if}\ A \neq \lambda:$
$\quad \mathsf{IV} \leftarrow \mathsf{HASH}_r(\mathsf{IV}, A, c_0);$
$\texttt{if}\ C \neq \lambda:$
$\quad C_1 \| \cdots \| C_m \xleftarrow{r} C;$
$\quad \texttt{for}\ i = 1\ \texttt{to}\ m:$
$\quad\quad Y \| Z \xleftarrow{(r, 256-r)} \mathtt{PHOTON}_{256}(\mathsf{IV});$
$\quad\quad (W, M_i) \leftarrow \rho^{-1}(Y, C_i);$
$\quad\quad \mathsf{IV} \leftarrow W \| Z;$
$\quad \mathsf{IV} \leftarrow \mathsf{IV} \oplus c_1;$
$\quad M \leftarrow M_1 \| \cdots \| M_m;$
$T^\star \leftarrow \mathsf{TAG}_{128}(\mathsf{IV});$
$\textbf{return}\ (T = T^\star)?\ M : \perp;$

$\mathsf{PHOTON\text{-}Beetle\text{-}Hash}[r](M)$

$\texttt{if}\ M = \lambda:$
$\quad \mathsf{IV} \leftarrow 0 \| 0;$
$\quad T \leftarrow \mathsf{TAG}_{256}(\mathsf{IV} \oplus 1);\ \texttt{return}\ T;$
$\texttt{if}\ |M| \leq 128:$
$\quad c_0 \leftarrow (|M| < 128)?\ 1:2$
$\quad \mathsf{IV} \leftarrow \mathsf{Ozs}_{128}(M) \| 0;$
$\quad T \leftarrow \mathsf{TAG}_{256}(\mathsf{IV} \oplus c_0);\ \texttt{return}\ T;$
$M_1 \| M' \xleftarrow{(128, |M|-128)} M;$
$c_0 \leftarrow (r|\ |M'|)?\ 1:2$
$\mathsf{IV} \leftarrow M_1 \| 0$
$\mathsf{IV} \leftarrow \mathsf{HASH}_r(\mathsf{IV}, M', c_0);$
$T \leftarrow \mathsf{TAG}_{256}(\mathsf{IV});$
$\textbf{return}\ T;$

$\mathsf{HASH}_r(\mathsf{IV}, D, c_0)$

$D_1 \| \cdots \| D_d \xleftarrow{r} \mathsf{Ozs}_r(D);$
$\texttt{for}\ i = 1\ \texttt{to}\ d:$
$\quad Y \| Z \xleftarrow{(r, 256-r)} \mathtt{PHOTON}_{256}(\mathsf{IV});$
$\quad W \leftarrow Y \oplus D_i;$
$\quad \mathsf{IV} \leftarrow W \| Z;$
$\mathsf{IV} \leftarrow \mathsf{IV} \oplus c_0;$
$\textbf{return}\ \mathsf{IV};$

$\mathsf{TAG}_\tau(T_0)$

$\texttt{for}\ i = 1\ \texttt{to}\ \lceil \tau/128 \rceil:$
$\quad T_i \leftarrow \mathtt{PHOTON}_{256}(T_{i-1});$
$T \leftarrow \mathsf{Trunc}(T_1, 128) \| \cdots \| \mathsf{Trunc}(T_{\tau/128}, 128);$
$\textbf{return}\ T;$

Figure 3.6: Formal Specification of PHOTON-Beetle-AEAD [r] := (PHOTON-Beetle-AEAD.ENC [r], PHOTON-Beetle-AEAD.DEC [r]) authenticated encryption and PHOTON-Beetle-Hash$[r]$ hash mode.

using the ciphertext blocks is captured by ρ^{-1}. 3-bit constants are added in the capacity part after the associated data and message processing for domain separation. A proper usage of these constants ensure that the algorithm allows empty associated data and/or empty message processing without any additional permutation calls. Formal specification PHOTON-Beetle-AEAD.ENC is given in Fig. 3.6. Corresponding figures can be found in Fig. 3.2 – 3.5. In the figure f denotes the permutation $\mathtt{PHOTON}_{256}$.

3.3 PHOTON-Beetle-Hash Hash function

PHOTON-Beetle-Hash takes a message $M \in \{0,1\}^*$ and generates a tag $T \in \{0,1\}^{256}$. We first parse the message into 128-bit block (the first block) followed by r-bit blocks. In this algorithm, the output of each permutation is xored with the next r-bit message block concatenated with zeros to compute the input to the next permutation call. We initialize the state by the first 128 bit block of the message concatenated with the required number of zeros and this initial state is the input to the first permutation call. When the final message block is processed, we xor a small constant in the capacity part depending on whether the final block is full or partial. This is done for domain separation. The 256-bit tag is squeezed into 2 parts of 128 bits each. The description of PHOTON-Beetle-Hash is given in Fig. 3.6.

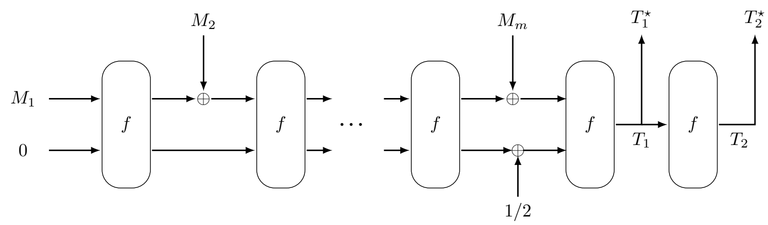

Figure 3.7: PHOTON-Beetle-Hash with m message blocks. Here $|M_1| = 128$, $|M_i| = r$, for $i = 2, \ldots, m-1$ and $|M_m| \leq r$. The tag T is computed as $T_1^\star \| T_2^\star$, where $T_i^\star = \mathsf{Trunc}(T_i, 128)$.

3.4 Recommended Versions

3.4.1 Authenticated Encryption Family

Our recommended versions for authenticated encryption with associated data are:

1. PHOTON-Beetle-AEAD[128]. This is our primary AEAD member. This design aims to be implemented with low hardware footprint yet with high throughput. Here we keep the rate of absorption of this cipher to be $r = 128$.

2. PHOTON-Beetle-AEAD[32]. This is another AEAD member that aims to be implemented with extremely low hardware footprint without giving much importance to the throughput. Hence, we keep the rate of absorption of this cipher to only $r = 32$.

3.4.2 Hash Function Family

Our recommended version for hash function is:

1. PHOTON-Beetle-Hash[32]. This is our only recommended Hash. The hash function absorbs the first 128 bits of plaintext as the initial vector and successive rate of absorption is kept to $r = 32$ bits. This design also aims to be implemented with extremely low hardware footprint and it is in particular has excellent throughput and energy efficiency for smaller messages. Note that, for any plaintext of size less than or equal to 128 bits, the hash function requires only 1 primitive call to process the message along with the two additional calls require to generate the hash value.

3.4.3 Combined AEAD and Hash Function Family

Based on our recommendations, we pair the following that provide both AEAD and hashing functionality.

1. PHOTON-Beetle-AEAD[32] + PHOTON-Beetle-Hash[32]. Both these AEAD and Hash operate on a 256-bit state, follow the sponge mode and use PHOTON_{256} as the underlying permutation with the same rate of data absorption (i.e. $r = 32$). The associated data process phase in PHOTON-Beetle-AEAD[32] is exactly the same as the message process phase of PHOTON-Beetle-Hash[32]. PHOTON-Beetle-Hash[32] with input

$X := X_1 \| X'$, where $X_1 \in \{0,1\}^{128}$, functions exactly in the similar way as PHOTON-Beetle-AEAD[32] with $N = X_1$, $K = 0^{128}$, $A = X'$ and $M = \lambda$ except the fact that PHOTON-Beetle-Hash[32] makes an additional call to PHOTON to generate 256 bit tag (in contrast with 128 bit tags in PHOTON-Beetle-AEAD[32]). Hence, in a combined PHOTON-Beetle-AEAD[32], PHOTON-Beetle-Hash[32] implementation, the implementation of PHOTON-Beetle-Hash[32] comes at a free of cost.

2. PHOTON-Beetle-AEAD[128] + PHOTON-Beetle-Hash[32]. In this version, the state size, mode and the underlying permutation remain same. However, the rate of absorption is different for the AEAD and the hash. From the functional point of view, the main design components remain same.

Chapter 4

Security

The security claims for PHOTON-Beetle-AEAD and PHOTON-Beetle-Hash is given in Table 4.1 and 4.2 respectively. To achieve these bounds, we assume all the nonces used in the encryption are distinct. A sketch of the security proof for the Beetle mode is given in Sect. 4.1 and 4.2.

Table 4.1: Security of Authenticated Encryption Family.

Mode	Security Model	Data complexity security (in bits)	Time complexity security (in bits)
PHOTON-Beetle-AEAD[128]	IND-CPA	121	121
PHOTON-Beetle-AEAD[128]	INT-CTXT	121	121
PHOTON-Beetle-AEAD[32]	IND-CPA	128	128
PHOTON-Beetle-AEAD[32]	INT-CTXT	128	128

Table 4.2: Security of Hash Function Family.

Mode	Security	Time complexity security (in bits)
PHOTON-Beetle-Hash[32]	Collision	112 (Query Complexity: $2^{111.5}$)
PHOTON-Beetle-Hash[32]	Pre-image	128

Third Party Analyses:

We would like to mention two third party analyses on PHOTON-Beetle-AEAD. The first one has been done on PHOTON-Beetle-AEAD[128] by Dobraunig and Mennink [1]. The paper remarked that when one performs a generic key recovery attack, the constant factors lead to a key recovery attack in encryption query complexity (data) $2^{122.8}$ and offline query complexity (time) of 2^{124}. Thus, as advised by the authors of [1], we have updated our security claims accordingly in Table 4.1. We can also insist on the fact that the authors actually said in their message that "We point out that our observation does not seriously threaten PHOTON-Beetle and that the problem is easily resolved by updating the security claims."

The second analysis by Mege [1], depicts that a collision can be obtained for PHOTON-Beetle-Hash[32] with $2^{111.5}$ query complexity due to an 1-bit constant addition (for domain separation) in the capacity part.

4.1 IND-CPA Security of PHOTON-Beetle-AEAD[r]

To attack against the privacy of PHOTON-Beetle-AEAD, we assume that an adversary makes at most q encryption queries (also known as on-line queries) $(N_i, A_i, M_i)_{i=1..q}$ to PHOTON-Beetle-AEAD[r] with an aggregate of total σ many blocks and q_p many off-line or direct permutation queries $(Q_i)_{i=1..q_p}$ to $\mathtt{PHOTON}_{256}$ or $\mathtt{PHOTON}_{256}^{-1}$. The adversary can distinguish the construction from a random function with the same domain and range if it finds a state collision (i) among the internal states of two on-line queries or (ii) among one online query internal state and an offline query output. As the adversary uses distinct nonces for each encryption, this is a possible way to mount a distinguishing attack. It is easy to see that the probability of a collision for case (i) can be bounded by $\frac{\sigma^2}{2^{256}}$. For case (ii), there are two sub-cases: (a) the initial state of an on-line query collides with the input of an offline query and (b) an intermediate state for an on-line query collides with the output of an off-line query. The first sub-case can occur with the probability at most $\frac{\sigma}{2^{256-r}}$ as having such a collision implies guessing the key. As different nonces are used for the on-line queries, an adversary can not control the rate part of any intermediate states of the on-line queries one can bound the probability of a collision between the internal state of one encryption query and input (or output) of an off-line query to $\frac{q_p\sigma}{2^{256}} + \frac{rq_p}{2^{128}}$. The term $\frac{rq_p}{2^{128}}$ appears when there is no r-multicollision in the 128-bit rate part of the internal states in the encryption queries. Now, one can easily bound the probability of r-multicollision in the upper 128-bit of the internal states by $\frac{\sigma_e^r}{2^{128(r-1)}} = \frac{\binom{\sigma_e}{r}}{2^{128.(r-1)}}$ (σ_e is the total number of blocks in th encryption queries). Hence, the privacy or IND-CPA advantage of PHOTON-Beetle-AEAD [r] can be bounded by $O(\frac{\sigma^2}{2^{256}} + \frac{q_p}{2^{256-r}} + \frac{q.q_p}{2^{256}} + \frac{rq_p}{2^{128}} + \frac{\sigma_e^r}{2^{128(r-1)}})$.

4.2 INT-CTXT Security of PHOTON-Beetle-AEAD$[r]$

On the other hand, to attack against the integrity of PHOTON-Beetle-AEAD, assume that an adversary makes at most q encryption queries (also known as on-line queries) $(N_i, A_i, M_i)_{i=1..q}$ to PHOTON-Beetle-AEAD$[r]$ with an aggregate of total σ many blocks and q_p many off-line queries $(Q_i)_{i=1..q_f}$ to PHOTON_{256} or $\mathsf{PHOTON}_{256}^{-1}$ and attempt to forge with $(N_i', A_i', C_i', T_i')_{i..q'}$ with an aggregate of σ' blocks. The trivial solution for forging is to guess the key or the tag which can be bounded by $\frac{q+q'}{2^{128}}$. Also, if an adversary can obtain a state collision among the input/output of a permutation query with the state of an encryption query or decryption query, it can use the fact to mount an forgery attack. The probability of having such a collision can be bounded by $(\frac{q_p(q+q')}{2^{256}} + \frac{rq_p}{2^{128}})$. Another possible (non-trivial) direction for the adversary is to construct an off-line query chain $(X_1, C_2, \ldots, C_k, T)$ such that $\exists Z_1, \ldots, Z_k$ and $c \in \{1, \ldots, 6\}$ with

$$\begin{aligned}
f(X_1\|Z_1) &= Y_1\|Z_2,\ \mathsf{Shuffle}(Y_1) \oplus C_2 = X_2,\\
f(X_2\|Z_2) &= Y_2\|Z_3,\ \mathsf{Shuffle}(Y_2) \oplus C_3 = X_3,\\
&\ \ \vdots\\
f(X_{k-1}\|Z_{k-1}) &= Y_{k-1}\|Z_k,\ \mathsf{Shuffle}(Y_{k-1}) \oplus C_k = X_k,\\
f(X_k\|(Z_k \oplus c)) &= T\|\star
\end{aligned}$$

and use this chain for forging. Here we claim that, if no r-multicollision occurs in the upper 128-bit outputs of the off-line queries, then the number of Z_1 for which this offline chain occurs can be at most $(\ell+1).r$ and the probability of forging in this case can be bounded by $\frac{r\sigma'}{2^c}$. This is due to the properties of the ρ function. Now, one can easily bound the probability of r-multicollision in the upper 128-bit outputs by $\frac{\binom{q_p}{r}}{2^{128.(r-1)}}$. Combining everything together, we claim that the INT-CTXT advantage of PHOTON-Beetle-AEAD [r] can be bounded by $O(\frac{q_p(q+q')}{2^{256}} + \frac{rq_p}{2^{128}} + \frac{q_p^r}{2^{128.(r-1)}} + \frac{r\sigma'}{2^{256-r}})$. Details of the security claim can be found in [3].

4.3 Collision Security of PHOTON-Beetle-Hash$[r]$

To mount a collision attack on PHOTON-Beetle-Hash [r], suppose an adversary can make q many permutation calls. Suppose all the states reachable from the initial state (we define the initial state as 0^{256}) using the permutation calls are called *reachable states*. The adversary can set up the queries in an adaptive way to make all the query inputs (and hence query outputs) *reachable states*. Now, if there is a collision in the capacity part of the output of two permutation calls, it can adjust the message in the rate part to force a state collision, which in turn can be used to make a collision in the hash. The probability of this event can be bounded by $\frac{q^2}{2^{256-r-1}}$ (1-bit extra due to the constant addition in the capacity part).

4.4 Preimage Security of PHOTON-**Beetle-Hash**$[r]$

In PHOTON-Beetle-Hash$[r]$ we set the tag size as 256 bits and the tag squeeze rate as 128 bits. Now, to find a pre-image of a hash value say $T_1\|T_2$, an adversary needs to find a Z such that $\texttt{PHOTON}_{256}(T_1\|Z_1) = T_2\|\star$ or $\texttt{PHOTON}_{256}^{-1}(T_2\|Z) = T_1$. It is easy to see that the probability of this event can be bounded by $\frac{q}{2^{128}}$.

4.5 Security of $\texttt{PHOTON}_{256}$ and Existing Analysis

The basic security analysis for $\texttt{PHOTON}_{256}$ has been provided explicitly in the original paper [6]. It has been there for several years now (ISO standard as well) and still remains with a comfortable security margin. Here we briefly discuss all the existing analysis on $\texttt{PHOTON}_{256}$. In [6], the authors mentioned a rebound-like attack that allows one to distinguish 8 rounds of $\texttt{PHOTON}_{256}$ from an ideal permutation of the same size with time complexity 2^{16} and memory complexity of 2^8. Later, [8] extended the previous result to further decrease the time complexity from 2^{16} to $2^{10.8}$. In [7] Jean et al. presented a distinguisher for 9 round $\texttt{PHOTON}_{256}$ with time complexity of 2^{184} and memory complexity of 2^{32}. In 2017, [5] presented a statistical Integral distinguisher that mounts an attack on 10 round $\texttt{PHOTON}_{256}$ with time complexity of $2^{96.59}$ and data complexity of $2^{70.46}$. Recently, Wang et al. [10] presented the first full round distinguishers on $\texttt{PHOTON}_{256}$ based on zero-sum partitions of size 2^{184}. We believe these distinguishers have no impact on the security of PHOTON-Beetle as these attacks are much more costlier than the security target we are aiming, and these attacks are basically unusable in the mode.

Chapter 5

Design Rationale

5.1 Choice of Beetle

Sponge is a well-known mode of operations typically used for light-weight applications. The main novelty behind Beetle sponge mode (the generic mode) is the *combined feedback* of the permutation output and the ciphertext block to generate the next permutation input. Recall that, in the simple Duplex Sponge [2], the ciphertext block itself is used as the rate part of the next permutation input. This technique actually resists the attacker to control the input block and the next blockcipher input simultaneously. This in turn uplifts the security level and helps us to reduce the state size and eventually come up with a low state implementation. In fact, this security upgrade ensures that we meet the security requirements of NIST even with a state size of 256 bits only.

5.2 Choice of ρ

Recall the definition of $\rho(S, U) := (S \leftarrow S \oplus U,\ Y \leftarrow \mathsf{Shuffle}(S) \oplus U)$. We need the ρ function such that, $S \rightarrow \mathsf{Shuffle}(S)$ should have full rank. Moreover, the rank of $S \rightarrow \mathsf{Shuffle}(S) \oplus S$ must be almost full. The ρ function ensures rank r and $(r-1)$ for the above two cases respectively. It is easy to see that our choice of Shuffle function only requires 1-bit right rotation of a string of $r/2$ bits, which is even cheaper than an xor operation of $r/2$ bits (as was used in the original Beetle). Moreover, the choice of ρ ensures uniform state update for associated data and message and identical to the state update of the duplex sponge.

5.3 Choice of PHOTON

Given that we have a good light-weight AEAD and hash mode based on public permutation, we now need a light-weight permutation with 256-bit state. Among the existing 256-bit permutations, $\texttt{PHOTON}_{256}$ [6] is considered as one of the lightest design in the literature. It can be implemented with a very low number of GE because all its components have been chosen with low-area in mind. In particular, the diffusion matrix is very lightweight in the sense that it can be serialised very easily and efficiently. Additionally, the constants are also chosen in such a manner that they can be generated on the fly with a very lightweight LFSR, without killing the performance. PHOTON promises much increased efficiency (both lighter and faster) over most of the existing designs and it has been well studied and well analysized. PHOTON is also a part of ISO-IEC: 29192-5 standard, which deal specifically with light-weight cryptography. Finally, PHOTON is not only of the smallest hash function (mainly due to the underlying permutation), it also achieves excellent area/throughput trade-offs and it even achieves very acceptable performances with simple software implementations.

Overall, a combination of PHOTON and Beetle can be considered as one of the best AEAD design in terms of state size and hardware area. We would like to point out that this design also deals with empty associated data and/or empty messages, which was missing in the original paper [4]. We employ the constant addition strategy for the domain separation. Also, we increase the size of the tag and the number of the tag bits squeezed per permutation call. This is to reduce the number of permutation invocations to make it more enegy efficient.

Chapter 6

Performance and Implementation Costs

6.1 Hardware Implementations

An advantage of PHOTON-Beetle is that the area of the hardware implementations of its members can be very small. The mode Beetle costs little on top of the costs of the underlying permutation $\mathtt{PHOTON}_{256}$. The underlying permutation $\mathtt{PHOTON}_{256}$ is one of the most compact among primitives with the same dimension. It can be implemented with a very low number of GE because all its components have been chosen with low-area in mind. In particular, the diffusion matrix is lightweight in the sense that it can be serialized very easily and efficiently.

Concretely, the area of the hardware implementations of all members in PHOTON-Beetle can be estimated using that of the hash function PHOTON-224/32/32, which also uses $\mathtt{PHOTON}_{256}$ as its underlying permutation. PHOTON-224/32/32 adopts Sponge construction with in-/output bit-rate 32/32. Considering that the Sponge construction also costs little on top of the costs of the underlying permutation, it is reasonable to use the area of the hardware implementation of PHOTON-224/32/32 to estimate that of PHOTON-Beetle. According to [6], as for serial ASIC implementation of PHOTON-224/32/32 using the standard cell library UMCL18G212T3 (with data path $s = 4$, which is the size of cells in the state), when target at minimizing area, it costs 1736 GEs and the latency of the underlying permutation is 1716 clock cycles; when target at minimizing latency, it costs 2786 GEs and the latency of the underlying permutation is 204 clock cycles.

Comparing implementations of the members of PHOTON-Beetle with that of PHOTON-224/32/32, additional costs of area may comes from the storage for key, nonce and larger message block (and the XOR gates for larger bit-rate). However, since key bits and nonce bits are used to initialize the state without schedule and will not be used after the initialization, such local storage can be reused and thus costs no additional area on top of the underlying permutation. In serial implementations with data path $s = 4$, larger bit-rate do not cause additional XOR gates because the XOR-ings are serialized. Hence, we estimate that for all members of PHOTON-Beetle, the area cost will be close to that cost by PHOTON-224/32/32.

6.2 Software Implementations

PHOTON-Beetle is primarily targeted for the constrained devices, and we mainly focus on the software implementation and performance of PHOTON-Beetle on micro-controllers.

6.2.1 Software Implementations on 8-bit AVR

Members of PHOTON-Beetle have small code size (ROM) and low RAM requirement when being implemented in bit-sliced way on 8-bit AVR microcontrollers. To show the speed and the possible trade-off between memory and speed, we present performance of two sets of our implementations in Table 6.1, one is targeted at optimizing ROM (avr8_lowrom), and the other is targeted at improving speed (avr8_speed). The cores of the implementations are all written in assembly; the main authenticated encryption, decryption, and hash functions have C APIs (we extended our previous pure assembly implementations to be compliant with the

Table 6.1: Performances of implementations on 8-bit AVR MCU

PHOTON-Beetle Pairs	Functionality	avr8_lowrom			avr8_speed		
		RAM	ROM	Speed	RAM	ROM	Speed
PHOTON-Beetle-AEAD[128] + PHOTON-Beetle-Hash[32]	AEAD	86	2136	8128.03	86	4084	4835.35
	Hash	54	1034	6566.27	54	2982	3860.66
	AEAD+Hash	86	2416	-	86	4364	-
PHOTON-Beetle-AEAD[32] + PHOTON-Beetle-Hash[32]	AEAD	74	2134	19789.79	74	4082	11596.39
	Hash	54	1034	6566.27	54	2982	3860.66
	AEAD+Hash	86	2414	-	86	4362	-

- RAM is in bytes, and is measured excluding those used for storing test vectors (including plaintexts, associated data, master key, ciphertexts, tags, nonce, etc.). ROM is in bytes, and is measured excluding the codes for generating test vectors and looping of calling the functions.

- Speed in cycles per byte, and is measured by using the total cycles divided by the total bytes of data (length of associated data is **from 0 to 32 bytes**, length of plaintexts is **from 0 to 32 bytes**.) So, for AEAD, the total data length is 34848 bytes; For Hash, the total data length is 528. For AEAD, the total cycles includes that takes both by 'crypto_aead_encrypt' and 'crypto_aead_decrypt'. Thus, for AEAD, the speed is cycles per 'encrypting' and 'decrypting' one byte. This measurement is in line with that of `https://lwc.las3.de/`.

- We extended our previous pure assembly implementations to be compliant with the SUPERCOP API of AEAD and hash. Due to this change, the updated ROM and RAM requirements are larger than that reported in our previous submitted document.

SUPERCOP API of AEAD and hash). The implementations were compiled using AVR8/GNU C Compiler 5.4.0 in Atmel Studio 7.0. The specific targeted device is AVR ATmega328P. The code sizes, RAM usage, and cycles are also measured using components of Atmel Studio 7.0.

From Table 6.1, when targeting at optimizing ROM (avr8_lowrom), PHOTON-Beetle-AEAD can be implemented with code size less than 2200 bytes, and PHOTON-Beetle-Hash can be implemented with code size less than 1100 bytes. Supporting hashing on top of AEAD costs very limited additional resources (less than 300 bytes of ROM); Supporting full functionality (authenticated encryption, authenticated decryption, and hashing), all PHOTON-Beetle-Pairs requires less than 2500 bytes of ROM, less than 100 bytes of RAM. For the primary pair, PHOTON-Beetle-AEAD[128] runs (executing both authenticated encryption and decryption) at an average speed faster than 8200 cycles per byte; PHOTON-Beetle-Hash[32] runs at an average speed faster than 6600 cycles per byte.

When targeting at improving speed (avr8_speed), PHOTON-Beetle-AEAD can be implemented with code size less than 4100 bytes, and PHOTON-Beetle-Hash can be implemented with code size less than 3000 bytes. Supporting hashing on top of AEAD costs very limited additional resources (less than 300 bytes of ROM); Supporting full functionality (authenticated encryption, authenticated decryption, and hashing), all PHOTON-Beetle-Pairs requires less than 4100 bytes of ROM, less than 100 bytes of RAM. Specifically, for the primary pair, PHOTON-Beetle-AEAD[128] runs (executing both authenticated encryption and decryption) at an average speed faster than 4900 cycles per byte; PHOTON-Beetle-Hash[32] runs at an average speed faster than 3900 cycles per byte.

To see how the speeds vary with length of short messages, we present detailed speed for the primary pair in Table 6.2. Compared with the performance of implementations of AES-GCM in [9], the speed is slower but acceptable, the ROM and RAM requirements are much less.

The implementations on 8-bit AVR are available via `https://github.com/PHOTON-Beetle/Software`.

Performance on Benchmarking Project. The platform established by Sebastian Renner, Enrico Pozzobon, and Jürgen Mottok (introduced in `https://lwc.las3.de/`), provides benchmarks of software implementations of AEAD of the second-round candidates. This platform also provided benchmarks of our submitted two sets of AVR implementations of PHOTON-Beetle. From the result about time and ROM on Arduino Uno R3 (MCU board based on the 8 bit ATmega328P MCU) presented in `https://lwc.las3.de/table.php`, the primary member PHOTON-Beetle-AEAD[128] have remarkable low ROM requirement[1]. Within a reasonable increase on the ROM (but is still relatively small), it can achieve moderate speed.

[1] In the presented result in `https://lwc.las3.de/table.php`, the ROM requirement includes that used to generate and check the test vectors. Thus, there is an obvious deviation between the ROM requirement presented in Table 6.1 and that presented in `https://lwc.las3.de/table.php`.

Table 6.2: Detailed speed of the primary pair of PHOTON-Beetle on AVR 8-bit MCU (length of AD = 16 bytes)

Algorithms	Func.	Package Length mlen [B]					ROM	RAM
		8	16	32	64	128		
PHOTON-Beetle-AEAD[128]	Enc	2476.38	1858.72	1652.56	1487.50	1377.38	4084	86
	Dec	2483.00	1863.41	1655.48	1489.14	1378.24		
	Enc+Dec	4959.38	3722.13	3308.04	2976.64	2755.63		
PHOTON-Beetle-Hash[32]	Hash	4880.13	2433.06	3568.50	4135.70	4419.27	2982	54

Denote the length of the message package by mlen, and the length of associated data by adlen that equals 16, the speeds of AEAD are measured by using cycles divided by (mlen+adlen).

Table 6.3: Speed of bitslice-based implementations of PHOTON-Beetle on PC

PHOTON-Beetle-AEAD[128] authenticated encryption (cycles/byte)

mlen/adlen	0	16	32	64	128	256	512	1024
0	-	209.40	155.70	129.70	116.40	109.60	104.40	104.00
16	214.20	160.50	141.30	126.40	116.40	111.00	108.20	104.80
32	163.60	143.80	133.70	123.30	115.60	110.10	106.50	104.00
64	135.90	129.30	124.80	117.60	111.40	109.40	106.70	104.70
128	120.40	117.90	118.00	115.60	112.50	108.50	106.80	105.50
256	112.60	114.10	113.50	112.50	111.20	109.00	107.50	105.40
512	111.60	111.40	111.00	110.50	109.70	108.20	106.40	105.60
1024	109.50	108.50	109.30	108.60	108.80	108.20	106.90	105.50

PHOTON-Beetle-AEAD[32] authenticated encryption (cycles/byte)

mlen/adlen	0	16	32	64	128	256	512	1024
0	-	767.40	682.40	460.50	434.80	425.90	421.60	420.10
16	530.50	474.10	463.00	442.10	434.40	426.60	422.20	420.20
32	477.10	464.50	447.60	438.10	434.30	426.70	422.90	421.40
64	466.60	492.30	458.70	453.20	437.30	434.00	425.10	420.20
128	435.30	433.80	432.60	430.50	427.50	424.20	421.10	420.30
256	432.20	427.70	427.90	425.70	422.70	425.40	421.40	419.90
512	423.40	422.90	421.50	422.10	421.50	421.40	421.20	420.20
1024	421.20	424.60	421.00	424.00	420.70	421.60	420.50	418.90

PHOTON-Beetle-Hash[32] hashing (cycles/byte)

mlen	0	16	32	64	128	256	512	1024
	-	201.50	307.80	363.40	387.50	398.90	403.40	406.30

mlen: length of messages in Bytes, adlen: length of AD in Bytes.
The programs are compiled using GNU gcc 7.5.0. The processor is Intel(R) Core(TM) i7-8565U CPU (Whiskey Lake). The CPU frequency scaling were disabled and the system was set to be performance during timing. The timing method used was that in `http://github.com/BrianGladman/AES`.

6.2.2 Software Implementations on General-Purpose Computers

For general-purpose computers, table-based and bit-slicing-based implementations were evaluated.

For table-based way, the performances of different implementations of combining different numbers of 4-bit S-boxes to create lookup tables of different sizes were tested. The implementation of combining two or three S-boxes performs faster than the implementation without combining or combining four S-boxes on a personal computer.

For bit-slicing-based way, the performances of different implementations of performing 8 S-boxes, 32 S-boxes, or 64 S-boxes in a sequence of logical instructions using 8-bit, 32-bit, or 64-bit registers, respectively, were evaluated.

The bit-slicing-based implementations performing 64 S-boxes in a sequence of logical instructions achieve the best performances among all table-based and bit-slicing-based implementations on a personal computer (equipped with Intel(R) Core(TM) i7 CPU), which are summarized in Table 6.3. The other implementations might be of interest to fitting low-end processors.

Implementations of PHOTON-Beetle in various ways and the source codes for timing can be found via `https://github.com/PHOTON-Beetle/Software`.

Bibliography

[1] Round 2 Official Comments - PHOTON-Beetle. `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/official-comments/photon-beetle-round2-official-comment.pdf`.

[2] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. *IACR Cryptology ePrint Archive*, 2011:499, 2011.

[3] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle family of lightweight and secure authenticated encryption ciphers. *IACR Cryptology ePrint Archive*, 2018:805, 2018.

[4] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle family of lightweight and secure authenticated encryption ciphers. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(2):218–241, 2018.

[5] Tingting Cui, Ling Sun, Huaifeng Chen, and Meiqin Wang. Statistical integral distinguisher with multi-structure and its application on AES. In *Information Security and Privacy - 22nd Australasian Conference, ACISP 2017, Auckland, New Zealand, July 3-5, 2017, Proceedings, Part I*, pages 402–420, 2017.

[6] Jian Guo, Thomas Peyrin, and Axel Poschmann. The PHOTON family of lightweight hash functions. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2011.

[7] Jérémy Jean, María Naya-Plasencia, and Thomas Peyrin. Improved rebound attack on the finalist grøstl. In Anne Canteaut, editor, *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, volume 7549 of *LNCS*, pages 110–126. Springer, 2012.

[8] Jérémy Jean, María Naya-Plasencia, and Thomas Peyrin. Multiple limited-birthday distinguishers and applications. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 533–550, 2013.

[9] Yaroslav Sovyn, Volodymyr Khoma, and Michal Podpora. Comparison of three cpu-core families for iot applications in terms of security and performance of AES-GCM. *IEEE Internet Things J.*, 7(1):339–348, 2020.

[10] Qingju Wang, Lorenzo Grassi, and Christian Rechberger. Zero-sum partitions of PHOTON permutations. In *Topics in Cryptology - CT-RSA 2018 - The Cryptographers' Track at the RSA Conference 2018, San Francisco, CA, USA, April 16-20, 2018, Proceedings*, pages 279–299, 2018.

Changelog

- Updated IND-CPA bounds for PHOTON-Beetle-AEAD[128], INT-CTXT bound for PHOTON-Beetle-AEAD[32] in Table 4.1 and added the query complexity for Collision attack against PHOTON-Beetle-Hash[32] in Table 4.2 in Chapter 4.
- Updated software implementations on general-purpose computers in Chapter 6.2.