The PHOTON Family of Lightweight Hash Functions

Jian Guo[1], Thomas Peyrin[⋆2], and Axel Poschmann[⋆2]

[1] Institute for Infocomm Research, Singapore
[2] Nanyang Technological University, Singapore
{ntu.guo,thomas.peyrin}@gmail.com, aposchmann@ntu.edu.sg

Abstract. RFID security is currently one of the major challenges cryptography has to face, often solved by protocols assuming that an on-tag hash function is available. In this article we present the PHOTON lightweight hash-function family, available in many different flavors and suitable for extremely constrained devices such as passive RFID tags. Our proposal uses a sponge-like construction as domain extension algorithm and an AES-like primitive as internal unkeyed permutation. This allows us to obtain the most compact hash function known so far (about 1120 GE for 64-bit collision resistance security), reaching areas very close to the theoretical optimum (derived from the minimal internal state memory size). Moreover, the speed achieved by PHOTON also compares quite favorably to its competitors. This is mostly due to the fact that unlike for previously proposed schemes, our proposal is very simple to analyze and one can derive tight AES-like bounds on the number of active Sboxes. This kind of AES-like primitive is usually not well suited for ultra constrained environments, but we describe in this paper a new method for generating the column mixing layer in a serial way, lowering drastically the area required. Finally, we slightly extend the sponge framework in order to offer interesting trade-offs between speed and preimage security for small messages, the classical use-case in hardware.

Key words: lightweight, hash function, sponge function, AES.

1 Introduction

RFID tags are likely to be deployed widely in many different situations of everyday life and they represent a great business opportunity for various markets. However, this rising technology also provides new security challenges that the cryptography community has to handle. RFID tags can be used to fight product counterfeiting by authenticating them and on the other hand, we would also like to guarantee the privacy of the users.

⋆ The authors were supported in part by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03.

These two security aspects have already been studied considerably and, interestingly, in most of the privacy-preserving RFID protocols proposed [3, 20, 23] a hash function is required. Informally, such a primitive is a function that takes an arbitrary length input and outputs a fixed-size value. While no secret is involved in the computation, one would like that finding collisions (two distinct messages hashing to the same value) or (second)-preimages (a message input that hashes to a given challenge output value) is computationally intractable for an attacker. More precisely, for an n-bit ideal hash function we expect to perform $2^{n/2}$ and 2^n computations in order to find a collision and a (second)-preimage respectively. While not as mature as block-ciphers, the research on hash functions saw a rapid development lately, mainly due to the groundbreaking attacks on standardized primitives [37, 35, 36]. At the present time, most of the attention of the symmetric key cryptography academic community is focused on the `SHA-3` competition organized by NIST [28], which should provide a potential replacement of the `MD-SHA` family.

In parallel, nice advances have also been made in the domain of lightweight symmetric key primitives in the last years. Protocol designers now have at disposal `PRESENT`[3], a 64-bit block-cipher with 80-bit key whose security has already been analyzed intensively and that can be as compact as 1075 GE [32]. Stream-ciphers are not outcast with implementations [19] with 80-bit security requiring about 1300 GE and 2600 GE reported for `GRAIN` and `TRIVIUM` respectively, two candidates selected in the final eSTREAM hardware portfolio. However, the situation is not as bright in the case of hash functions.

As already pointed out in [18] and echoed in, the community lacks very compact hash functions. Standardized primitives such as `SHA-1` [26] or `SHA-2` [27] are much too large to fit in very constrained hardware (5527 GE [29] and 10868 GE [18] for 80 and 128-bit aimed security respectively) and even compact-oriented proposals such as `MAME` require 8100 GE for 128-bit security. While hardware is an important criteria in the selection process, one can not expect the `SHA-3` finalists to be much more compact. At the present time, all `SHA-3` finalists require more than 12000 GE for 128-bit security (smaller versions of `KECCAK` that have not been submitted to the competition provide for example 64-bit security with 5090 GE). Note that a basic RFID tag may have a total gate count of anywhere from 1000-10000 gates, with only 200-2000 gates budgeted for security [22].

This compactness problem in hash algorithms is partly due to the fact that it widely depends on the memory registers required for the computation. Most hash functions proposed so far are software-oriented and output at least 256 bits in order to be out of reach of any generic collision search in practice. While such an output size makes sense where high level and long-term security are needed, RFID use-cases could bear much smaller security parameters. This is for example the path taken in, where the authors instantiate lightweight hash functions using literature-based constructions [21, 31] with the compact block-

[3] Due to space limit, we omitted the references for many designs, interested readers are referred to [1] for an extended version of this article.

cipher PRESENT. With SQUASH, Shamir proposed a compact keyed hash function inspired by the Rabin encryption scheme that processes short messages (at most 64-bit inputs) and that provides 64 bits of preimage security, without being collision resistant. At CHES 2010, the lightweight hash-function family ARMADILLO was proposed, but has recently been shown to present serious security weaknesses [11]. At the same conference, Aumasson *et al.* published the hash function QUARK, using sponge functions [4] as domain extension algorithm, and an internal permutation inspired from the stream-cipher GRAIN and the block-cipher KATAN [14]. Using sponge functions as operating mode is another step towards compactness. Indeed, classical n-bit hash function constructions like the MD-SHA family utilize a Merkle-Damgård [24, 17] domain extension algorithm with a compression function h built upon an n-bit block-cipher E in Davies-Meyer mode ($h(CV, M) = E_M(CV) \oplus CV$), where CV stands for the chaining variable and M for the current message block. Avoiding any feed-forward like for sponge constructions saves a lot of memory registers at the cost of an invertible iterative process which induces a lower (second)-preimage security for the same internal state size. All in all, designers have to deal with a trade-off between security and memory requirements.

In this article, we describe a new hardware-oriented hash-function family: PHOTON. We chose to use the sponge functions framework in order to keep the internal memory size as low as possible. However, we extend this framework so as to provide very interesting trade-offs in hardware between preimage security and small messages hashing speed (small message scenario is a classical use-case and can be problematic for sponge functions because of their squeezing process that can be very slow in practice). The internal permutations of PHOTON can be seen as AES-like primitives especially derived for hardware: our columns mixing layer can be computed in a serial way while maintaining optimal diffusion properties. Overall, as shown in Table 2 in Section 4.3, not only PHOTON is easily the smallest hash function known so far, but it also achieves excellent area/throughput trade-offs.

In terms of security, it is particularly interesting to use AES-like permutations as we can fully leverage all the previous cryptanalysis performed on AES and on AES-based hash functions (again due to space limit we refer the reader to [1] for a detailed security analysis). Moreover, we can directly derive very simple bounds on the number of active Sboxes for 4 rounds of the permutation. These bounds being tight, we can confidently set an appropriate number of rounds that ensures a comfortable security margin.

2 Design Choices

In tag-based applications, one typically does not require high security primitives, such as a 512-bit output hash function. In contrary, 64 or 80-bit security is often appropriate considering the value of objects an RFID tag is protecting and the use cases. Moreover, a designer should use exactly the level that he expects from his primitive, so as to avoid any waste of area or computing power. This is the reason why we chose to precisely instantiate several security levels for `PHOTON`, ranging from 64-bit preimage resistance security to 128-bit collision resistance security.

2.1 Extended Sponge functions

Sponge functions have been introduced by Bertoni *et al.* [4] as a new way of building hash functions from a fixed permutation (later more applications were proposed [7]). The internal state S, composed of the c-bit capacity and the r-bit bitrate, is first initialized with some fixed value. Then, after having appropriately padded and split the message into r-bit chunks, one simply and iteratively processes all r-bit message chunks by xoring them to the bitrate part of the internal state and then applying the $(c+r)$-bit permutation P. Once all message chunks have been handled by this absorbing phase, one successively outputs r bits of the final hash value by extracting r bits from the bitrate part of the internal state and then applying the permutation P on it (squeezing process).

When the internal permutation P is modeled as a randomly chosen permutation, a sponge function has been proven to be indifferentiable from a random oracle [5] up to $2^{c/2}$ calls to P. More precisely, for an n-bit sponge hash function with capacity c and bitrate r, when the internal primitive is modeled as a random permutation, one obtains $\min\{2^{n/2}, 2^{c/2}\}$ as collision resistance bound and $\min\{2^n, 2^{c/2}\}$ as (second)-preimage bound. However, in the case of preimage, there exists a gap between this bound and the best known generic attack[4]. Therefore, we expect the following complexities in the generic case:

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

Moreover, sponge functions can be used as a Message Authentication Code with $MAC_K(M) = H(K||M)$, where $K \in \{0,1\}^k$ stands for the key and M for the message. It has been shown [8] that as long as the amount of message queries is limited to 2^a with $a \ll c/2$, then no attack better than exhaustive key search exists if $c \geq k + a + 1$.

Sponge functions seem a natural choice in order to minimize the amount of memory registers in hardware since they can offer speed/area/security tradeoffs. Indeed, the only memory required for the internal state is $c + r$ bits, while for a classical Davies-Meyer construction using an m-bit block cipher with a k-bit key input one needs to store $2m + k$ bits, out of which m bits are required for the feed-forward. For an equivalent ideal collision security level (thus setting $m = c = n$) and by minimizing the area (r and k are very small), the sponge function requires only about half of the memory. Note that if one looks for a

[4] The 2^{n-r} term for preimage comes from the fact that in order to invert the hash function, the attacker will have to invert the squeezing process. The best known generic attack to solve this “multiblock constrained-input constrained-output problem” [6] requires 2^{n-r} computations.

perfectly (second)-preimage resistant hash function (up to the 2^n ideal bound), then it is required that $c \geq 2 \cdot n$ (which implies that the n-bit hash function built is indifferentiable from an n-bit random oracle anyway). In that particular case the sponge functions are not better than the Davies-Meyer construction in terms of area requirements and therefore in this work we will not focus on this scenario. Instead, we will build hash functions that may have ideal resistance to collision, but not for (second)-preimage. The typical shape will be a capacity c equal to the hash output n and a very small bitrate r. This security/area trade-off, already utilized by the `QUARK` designers, will allow us to aim at extremly low area requirements, while maintaining security expectations very close to ideal.

In, the authors identify that in most RFID applications the user will not hash a large amount of data, *i.e.* in general less than 256 bits. Consider for example the *electronic product code (EPC)* number, which is a 96-bit string that is meant to identify globally any tag/product. In this particular case of small messages, sponge functions with a small bitrate r seem to be slow since one needs to call $(\lceil n/r \rceil - 1)$ times the internal permutation to complete the final squeezing process. This is for example the case with `U-QUARK`, that has a throughput of 1.47 kbps for very long messages which drops to 0.63 kbps for 96-bit inputs. On the other side, this "small messages" effect is reduced by the fact that having a small bitrate will reduce the amount of padding actually hashed (the padding simply consists in adding a "1" and as many "0" required to fill the last message block). Note that lightweight proposals based on classical Davies-Meyer construction that include the message length as suffix padding are also slow for small messages: `DM-PRESENT`-80 has a throughput of 14.63 kbps for very long messages which drops to 5.85 kbps for 96-bit inputs, because in the latter case many of the compression function calls are spent in order to handle padding blocks.

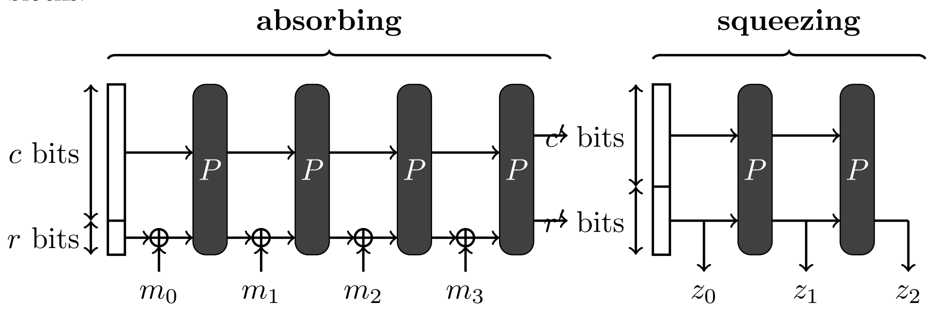

Fig. 1. The extended sponge framework, the domain extension algorithm used by the `PHOTON` hash-function family.

In order to allow more flexibility about this issue, we propose to slightly extend the sponge framework by allowing the number r' of bits extracted during each iteration of the squeezing process to be different from the bitrate r[5] (see

[5] A recent work from Andreeva *et al.* [2] also independently proposed such an extension of the sponge model.

Figure 1). Increasing r' will directly reduce the time spent in the squeezing process, but might also reduce the preimage security. On the contrary, decreasing r' might improve the preimage bound at the cost of a speed drop for small messages. As long as the preimage security remains in an acceptable bound, this configuration can be interesting in many scenarios where only tiny inputs are to be hashed. More precisely, in this new model, the best known generic attacks require the following amount of computations:

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

Finally, in most tag-based applications the collision resistance is not a requirement, while only the one-wayness of the function must be ensured. However, as we previously explained, for lightweight scenarios the sponge construction does not maintain the (second)-preimage security at the full level of its capacity c. This is due to the output process of the sponge operating mode. Of course, performing a Davies-Meyer like feed-forward just after the final truncation would do the job, but that would also double the memory area required (which is precisely what we are trying to avoid). The nice trick of squeezing in the sponge functions framework permits to avoid any feed-forward while somehow rendering the process non-invertible, up to some extend (see multiblock constrained-input constrained-output problem in [6]). One solution to reach the full capacity preimage security would be to add one more squeezing iteration, thus increasing the output size of the hash by r' bits.[6] Then, the best known generic preimage attack for this $(n + r')$-bit hash function will run in $\min\{2^{n+r'}, 2^{c}, \max\{2^{n}, 2^{c/2}\}\} \geq 2^{n}$ when $c \geq n$ and one has to note that this hash output extension has no influence on the second-preimage resistance.

In this article, we will provide five sizes of internal permutations and one PHOTON flavor for each of them. The four biggest versions fit the classical sponge model and will ensure $2^{n/2}$ collision and second preimage resistance and 2^{n-r} concerning preimage. However, in order to illustrate the powerful trade-offs allowed by our extended model, the smaller PHOTON variant will have different input/output bitrates and an extended hash size. Using the five permutations defined in the next Section, one can derive its own PHOTON flavor depending on the collision / (second)-preimage / MAC security required, the maximal area and the maximal hash output size allowed. Note that the area required will only depend on the internal permutation chosen.

2.2 An AES-like internal permutation

We define an AES-like function to be a fixed key permutation P applied on an internal state of d^2 elements of s bits each, which can be represented as a $(d \times d)$

[6] This generalization has been independently utilized by the QUARK designers in a revised version of their original article.

matrix. P is composed of N_r rounds, each containing four layers : AddConstants (AC), SubCells (SC), ShiftRows (ShR), and MixColumnsSerial (MCS). Informally, AddConstants simply consists in adding fixed values to the cells of the internal state, while SubCells applies an s-bit Sbox to each of them. ShiftRows rotates the position of the cells in each of the rows and MixColumnsSerial linearly mixes all the columns independently.

We chose to use AES-like permutations because they offer much confidence in the design strategy as one can leverage previous cryptanalysis works done on AES and on AES-like hash functions. Moreover, AES-like permutations allow to derive very simple proofs on the number of active Sboxes over four rounds of the primitive. More precisely, if the matrix underlying the MixColumnsSerial layer is Maximum Distance Separable (MDS), then one can immediately show that at least $(d+1)^2$ Sboxes will be active for any 4-round differential path [16]. This bound is tight, and we already know differential paths with only $(d+1)^2$ active Sboxes for four rounds (we will use them later for security analysis purposes). Moreover, note that the permutations we will design are fixed-key, so we naturally get rid of related-key attacks or any issue that might arise from the construction of a key-schedule [9, 10].

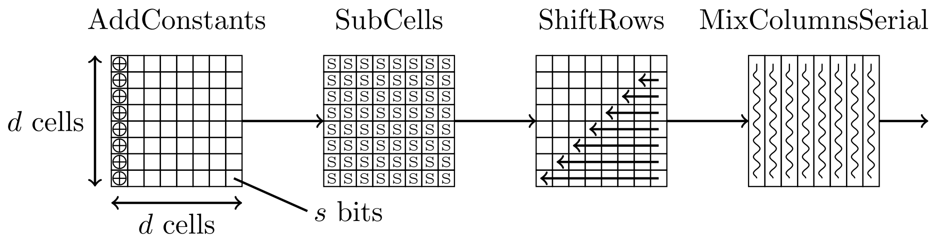

Fig. 2. One round of a PHOTON permutation.

AddConstants. The constants have been chosen such that each of the N_r round computations are different, and such that the classical symmetry between columns in AES-like designs are destroyed (without the AddConstants layer, an input with all columns equal would maintain this property through any number of rounds). Also, the round constants can be generated by a combination of very compact Linear Feedback Shift Registers. For performance reasons, only the first column of the internal state is involved.

SubCells. Our choice of the Sboxes was mostly motivated by their hardware quality. 4-bit Sboxes can be very compact in hardware while the acceptable upper limit on the cell size is $s = 8$. We avoided to use an Sbox size s which is odd, because this leads to odd message block size or capacity when d is also odd. This leaves us with $s = 4, 6, 8$, but we also believe that reusing some already trusted and well analyzed components increases the confidence in the security of the scheme and saves a lot of time for cryptanalysts. Finally, we will use two types of Sboxes: the 4-bit PRESENT Sbox $\mathtt{SBOX}_{\mathtt{PRE}}$ and the 8-bit AES Sbox $\mathtt{SBOX}_{\mathtt{AES}}$ the latter being only utilized for high security levels (at least 128 bits of collision resistance). Note also that $s = 4, 8$ allows simpler and faster software implementations.

ShiftRows. The choice of the ShiftRows constants is very simple for PHOTON since our internal state is always a square of cells. Therefore, row i will classically be rotated by i positions to the left, i counts from 0.

MixColumnsSerial. The matrix underlying the AES MixColumns function is a circulant matrix with low hamming weight coefficients. Even if those coefficients and the irreducible polynomial used to create the Galois field for the AES MixColumns function have been chosen so as to improve the hardware footprint of the cipher, it can not be implemented in an extremely compact way. One of the main reason is that the byte-serial implementation of this function is not compact. Said in other words, if we write the AES MixColumns matrix as the composition of d operations each updating a single byte at a time in a serial way, then the coefficients of these d matrices will be very bad for small area implementations.

In order to solve this issue, we took the problem the other way round. Let A be the matrix that updates the last cell of the column vector with a linear combination of all of the vector cells and then rotates the vector by one position towards the top. Our new MixColumnsSerial layer will be composed of d applications of this matrix to the input column vector. More formally, let $X = (x_0, \ldots, x_{d-1})^T$ be an input column vector of MixColumnsSerial and $Y = (y_0, \ldots, y_{d-1})^T$ be the corresponding output. Then, we have $Y = A^d \times X$, where A is a $(d \times d)$ matrix of the form:

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

where coefficients $(Z_0, \ldots, Z_{d-1})$ can be chosen freely. We denote by $Serial(Z_0, \ldots, Z_{d-1})$ such a matrix. Of course, we would like the final matrix A^d to be MDS, so as to maintain as much diffusion as for the AES initial design strategy. For each square size d we picked during the design of PHOTON, we used MAGMA [12] to test all the possible values of $Z_0, \ldots, Z_{d-1}$ and picked the most compact candidate making A^d an MDS matrix. We also chose the irreducible polynomial with compactness as main criterion.

For design strategy comparison purposes, we can take as an example the AES case. By using our new mixing layer design method, we were able to find the matrix $A = Serial(1, 2, 1, 4)$ which gives the following MDS final matrix:

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

The smallest AES hardware implementation requires 2400 GE [25], for which 263 GE are dedicated to MixColumns. It is possible to implement MixColumns of AES in a byte-by-byte fashion, which requires only 81 GE to calculate one byte of the output column. However, since AES uses a circulant matrix, at least three additional 8-bit registers (144 GE), are required to hold the output, plus additional control logic, which increases the area requirements significantly. That is why [25] does not use a serial MixColumns, but rather processes one column at a time.

Please note that in general the choice of non-zero constants for any $d \times d$ MDS matrix on s-bit cells has only a minor impact of the area consumption, since a multiplication by x consists of w XOR gates, where w denotes the Hamming weight of the irreducible polynomial used. At the same time, $(d-1) \cdot s$ XOR gates are required to sum up the d individual terms of s bits each. It is no surprise, that multiplying with the constants above accounts for only 21.3 GE out of the 74 GE required. In fact, the efficiency of our approach lies in the shifting property of A, since this allows to re-use the existing memory with neither temporary storage nor additional control logic required.

All in all, using our approach would provide a tweaked AES cipher with the very same diffusion properties as the original one (the matrix being MDS), but that can fit in only 2210 GE, a total saving of around 8%. Moreover, for the deciphering process, a slightly modified hardware can be used in order to unroll the MixColumnsSerial, further reducing the area footprint of such a PHOTON-based cipher. One might think that the software implementations will suffer from this new layer. While our goal is to make a hardware-oriented primitive, we would like to remark that most AES software implementations are precomputed tables-based (applying both the Sbox and the MixColumns coefficients at the same time) and the very same method can be applied to PHOTON. This is confirmed by our first software implementations, whose benchmarks are given in Section 4.4.

3 The PHOTON Hash-Function Family

We describe in this section the PHOTON family of hash functions.[7] Each variant will be fully defined by its hash output size $64 \leq n \leq 256$, its input and its output bitrate r and r' respectively. Therefore we denote each function PHOTON-$n/r/r'$. The internal state size $t = (c + r)$ depends on the hash output size and can take only 5 distinct values: 100, 144, 196, 256 and 288 bits. As a consequence, we only have to define 5 internal permutations P_t, one for each internal state size.

[7] An extended version of this paper including a more detailed description of PHOTON and test vectors can be found at the PHOTON website [1].

In order to cover a wide spectrum of applications, we propose five different flavors of PHOTON, one for each internal state size: PHOTON-80/20/16, PHOTON-128/16/16, PHOTON-160/36/36, PHOTON-224/32/32 and PHOTON-256/32/32 will use internal permutations P_{100}, P_{144}, P_{196}, P_{256} and P_{288} respectively. Note that the first proposal is special in the sense that it is designed for the specific cases where 64-bit preimage security and 64-bit key MAC are considered to be sufficient.[8] In contrary, the last proposal provides a high security level of 128-bit collision resistance, thus making it suitable for generic applications.

3.1 The domain extension algorithm

The message M to hash is first padded by appending a "1" bit and as many zeros (possibly none) such that the total length is a multiple of the bitrate r and we can finally obtain l message blocks $m_0, \ldots, m_{l-1}$ of r bits each. The t-bit internal state S is initialized by setting it to the value $S_0 = IV = \{0\}^{t-24}||n/4||r||r'$, where $||$ denotes the concatenation and each value is coded on 8 bits. For implementation purposes, note that each byte is interpreted in big-endian form.

Then, as for the classical sponge strategy, at iteration i we absorb the message block m_i on leftmost part of the internal state S_i and then apply the permutation P_t, *i.e.* $S_{i+1} = P_t(S_i \oplus (m_i||\{0\}^c))$. Once all l message blocks have been absorbed, we build the hash value by concatenating the successive r'-bit output blocks z_i until we reach the appropriate output size n: $hash = z_0|| \ldots ||z_{l'-1}$, where l' denotes the number of squeezing iterations, that is $l' = \lceil n/r' \rceil - 1$. More precisely, z_i is the r' leftmost bits of the internal state S_{l+i} and we have $S_{l+i+1} = P_t(S_{l+i})$ for $0 \leq i < l'$. If the hash output size is not a multiple of r', one just truncates $z_{l'-1}$ to $n \bmod r'$ bits.

3.2 The internal permutations

We define here the internal permutations P_t, where $t \in \{100, 144, 196, 256, 288\}$. The internal state of the N_r-round permutation is viewed as a $(d \times d)$ matrix of s-bit cells and the corresponding values depending of t are given in Table 1. Note that we will always use a cell size of 4 bits, except for the largest version for which we use 8-bit cells, and that the number of rounds is always $N_r = 12$, whatever the value of t is. The internal state cell located at row i and column j is denoted $S[i,j]$ with $0 \leq i,j < d$.

One round is composed of four layers (see Figure 2): AddConstant (AC), SubCell (SC), ShiftRows (ShR) and MixColumnsSerial (MCS).

[8] By sponge keying and using the security bound from [8], PHOTON-80/20/16 provides a secure 64-bit key MAC as long as the number of messages to be computed is lower than 2^{15}. For a secure 64-bit key MAC handling more messages (up to 2^{27}), one can for example go for a very similar PHOTON-80/8/8 version that also uses P_{100}. This version with capacity $c = 92$ would require the same area as PHOTON-80/20/16 but would be slower.

Table 1. The parameters of the internal permutations P_t, together with the internal constants IC_d, the irreducible polynomials and the Z_i coefficients for the MixColumnsSerial computation.

	t	d	s	N_r	$IC_d(\cdot)$	irr. polynomial	Z_i coefficients
P_{100}	100	5	4	12	$[0, 1, 3, 6, 4]$	$x^4 + x + 1$	$(1, 2, 9, 9, 2)$
P_{144}	144	6	4	12	$[0, 1, 3, 7, 6, 4]$	$x^4 + x + 1$	$(1, 2, 8, 5, 8, 2)$
P_{196}	196	7	4	12	$[0, 1, 2, 5, 3, 6, 4]$	$x^4 + x + 1$	$(1, 4, 6, 1, 1, 6, 4)$
P_{256}	256	8	4	12	$[0, 1, 3, 7, 15, 14, 12, 8]$	$x^4 + x + 1$	$(2, 4, 2, 11, 2, 8, 5, 6)$
P_{288}	288	6	8	12	$[0, 1, 3, 7, 6, 4]$	$x^8 + x^4 + x^3 + x + 1$	$(2, 3, 1, 2, 1, 4)$

AddConstant. At round number v (starting the counting from 1), we first XOR a round constant $RC(v)$ to each cell $S[i, 0]$ of the first column of the internal state. Then, we XOR distinct internal constants $IC_d(i)$ to each cell $S[i, 0]$ of the same first column. Overall, for round v we have $S'[i, 0] = S[i, 0] \oplus RC(v) \oplus IC_d(i)$ for all $0 \leq i < d$. The round constants are

$$RC(v) = [1, 3, 7, 14, 13, 11, 6, 12, 9, 2, 5, 10].$$

The internal constants depend on the square size d and on the row position i. They are given in Table 1.

SubCells. This layer simply applies an s-bit Sbox to each of the cells of the internal state, *i.e.* $S'[i, j] = \mathtt{SBOX}(S[i, j])$ for all $0 \leq i, j < d$. In the case of 4-bit cells, we use the `PRESENT` Sbox $\mathtt{SBOX_{PRE}}$ while for the 8-bit cells case we use the `AES` Sbox $\mathtt{SBOX_{AES}}$ [16].

ShiftRows. As for the `AES`, for each row i this layer rotates all cells to the left by i column positions. Namely, $S'[i, j] = S[i, (j + i) \bmod d]$ for all $0 \leq i, j < d$.

MixColumnsSerial. The final mixing layer is applied to each of the columns of the internal state independently. For each column j input vector $(S[0, j], \dots, S[d-1, j])^T$, we apply d times the matrix $A_t = Serial(Z_0, \dots, Z_{d-1})$. That is, for all $0 \leq j < d$: $(S'[0, j], \dots, S'[d-1, j])^T = A_t^d \times (S[0, j], \dots, S[d-1, j])^T$ where the coefficients $Z_0, \dots, Z_{d-1}$ are given in Table 1. In the case of 4-bit cells, the irreducible polynomial we chose is $x^4 + x + 1$, while for the 8-bit case we chose the `AES` one, *i.e.* $x^8 + x^4 + x^3 + x + 1$. Note that all A_t^d matrices are Maximum Distance Separable.[9]

4 Performances and Comparison

Before we detail the hardware architectures and the optimizations done, we first describe the tools used. Finally we compare our results to previous work.

[9] One could wonder why we did not propose a version with $d = 9$ and $s = 4$. The reason is that there is no matrix fulfilling the desired "serial MDS" properties for those parameters, whatever the irreducible polynomial chosen.

4.1 Design flow

We used *Mentor Graphics ModelSimXE 6.4b* and *Synopsys DesignCompiler A-2007.12-SP1* for functional simulation and synthesis of the designs to the *Virtual Silicon* (VST) standard cell library *UMCL18G212T3* [34], which is based on the *UMC L180 0.18μm 1P6M* logic process with a typical voltage of 1.8 V. We used *Synopsys Power Compiler* version *A-2007.12-SP1* to estimate the power consumption of our ASIC implementations. For synthesis and for power estimation we advised the compiler to keep the hierarchy and use a clock frequency of 100 KHz. Note that the wire-load model used, though it is the smallest available for this library, still simulates the typical wire-load of a circuit with a size of around 10 000 GE.

4.2 Hardware architectures

To substantiate our claims on the hardware efficiency of our `PHOTON` family, we have implemented the flavors specified in Section 3 in `VHDL` and simulated their post-synthesis performance. We designed two architectures: one is fully serialized, *i.e.* performing operations on one cell per clock cycle, and aims for the smallest area possible; the second one is a d times parallelization of the first architecture, thus performing operations on one row in one clock cycle, resulting in a significant speed-up. As can be seen in Figure 3, our serialized design consists of six modules: `MCS`, `State`, `IO`, `AC`, `SC`, and `Controller`.

`IO` allows to 1) initialize our implementation with an all '0' vector, 2) input the IV, 3) absorb message chunks, and 4) forward the output of the `State` module to the `AC` module without further modification. Instead of using two Multiplexer and an XOR gate, we used two NAND and one XOR gate thereby reducing the gate count required from $s \cdot 7.33$ to $s \cdot 4.67$ GE.

`State` comprises a $d \cdot d$ array of flip-flop cells storing s bits each. Every row constitutes a shift-register using the output of the last stage, *i.e.* column 0, as the input to the first stage (column $d-1$) of the same row and the next row. Using this feedback functionality ShiftRows can be performed in $d-1$ clock cycles with no additional hardware costs. Further, since MixColumnsSerial is performed on column 0, also a vertical shifting direction is required for this column. Consequently, columns 0 and $d-1$ consist of flip-flop cells with two inputs (6 GE), while columns 1 to $d-2$ consist of flip-flop cells with only one input (4.67 GE). The overall gate count for this module is $s \cdot d \cdot ((d-2) \cdot 4.67 + 2 \cdot 6)$ GE and for all flavors it occupies the majority of the area required (between 65 and 77.5%).

`MCS` calculates the last row of A_t in one clock cycle. The result is stored in the `State` module, that is in the last row of column 0, which has been shifted upwards at the same time. Consequently, after d clock cycles the MixColumnsSerial operation is applied to an entire column. Then the whole state array is rotated by one position to the left and the next column is processed. In total $d \cdot (d+1)$ clock cycles are required to perform MCS. As an example of the hardware efficiency of MCS we depict A_{100} in the upper and its sub-components in the lower right part of Figure 3. Using our library, for a multiplication by 2, 4 and 8, we need 2.67 GE, 4.67 GE, and 7 GE when using the irreducible polynomial $x^4 + x + 1$, respectively. Therefore the choice of the coefficients has only a minor impact on the overall gate count, as the majority is required to sum up the intermediate results. For example, in the case of A_{100}, 56 out of 75.33 GE are required for the XOR sum. The gate counts for the other matrices are: 80 GE, 99 GE, 145 GE, and 144 GE for A_{144}, A_{196}, A_{256}, and A_{288}, respectively.

`AC` performs the AddConstant operation by XORing the sum of the round constant RC with the current internal constant IC. Furthermore, since AC is only applied to the first column, the input to the XNOR gate is gated with a NAND gate. Instead of using an AND gate in combination with an XOR gate, our approach allows to reduce the area required from $s \cdot 6.67$ to $s \cdot 6$ GE.

`SC` performs the SubCells operation and consists of a single instantiation of the corresponding Sbox. For $s = 4$ we used an optimized Boolean representation of the `PRESENT` Sbox, which only requires 22.33 GE and for $s = 8$ we used Canright's representation of the AES Sbox [15] which requires 233 GE. It takes $d \cdot d$ clock cycles to perform AddConstant and SubCells on the whole state.

`Controller` uses a Finite State Machine (FSM) to generate all control signals required. Furthermore, also the round constants and the internal constants are generated within this module, as their values are used for the transition conditions of the FSM. The FSM consists of one idle state, one state for the combined execution of AC and SC, $d - 1$ states for ShR and two states for MCS (one for processing one column and another one to rotate the whole state to the left). Naturally, its gate count varies depending on d: 197 GE, 210 GE, 235 GE, and 254 GE for $d = 5, 6, 7, 8$, respectively.

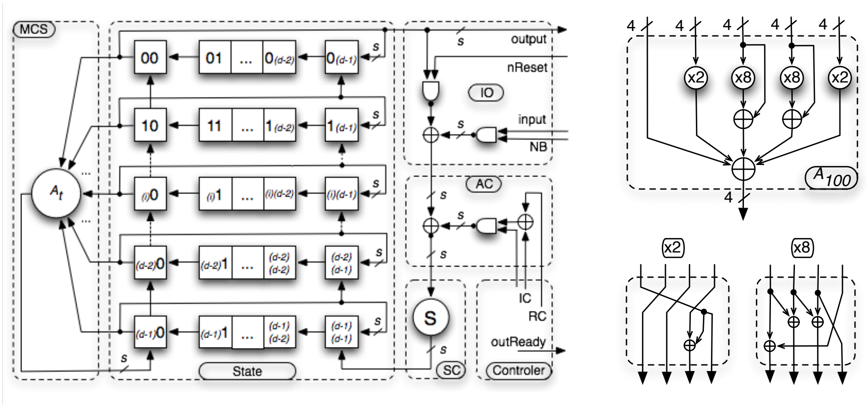

Fig. 3. Serial hardware architecture of `PHOTON` (left). As an example for its component A_t we also depict A_{100} with its sub-components (right).

4.3 Hardware results and comparison

We assume the message to be padded correctly and the IVs to be loaded at the beginning of the operation. Then it requires $d \cdot d + (d-1) + d \cdot (d+1)$ clock cycles to perform one round of the permutation P, resulting in a total latency of $12 \cdot (2 \cdot d \cdot (d+1) - 1)$ clock cycles. Table 2 compares our results to previous works, sorted after preimage and collision resistance levels. Area requirements are provided in GE, while the latency is given in clock cycles for only the internal permutation P (or the internal block-cipher E), and the whole hash function H. Further metrics are Throughput in kbps and a Figure of Merit (FOM) proposed by. In order to have a comparison for a best case scenario and a real-world application, we provide the latter two metrics for 'long' messages (omitting any padding influences) and for 96-bit messages, where we do take padding into account. In particular this means that a 96-bit message is padded with "1" and as many "0"s as required. Furthermore Merkle-Damgård constructions need additional 64 bits to encode the message length. The parameters n, c, r and r' stand for the hash output size, the capacity, the input bitrate and the output bitrate respectively. Finally, the column "Pre" gives the claimed preimage resistance security and "Col" the claimed collision resistance security.

As can be seen, our proposals compete well in terms of area requirements, since they are 18% to 75% smaller compared to previous proposals with a similar preimage/collision resistance level. For a smaller area, the throughput of PHOTON variants is comparable to the QUARK proposals[10]. Alternatively, for a similar area, PHOTON variants are much faster than the QUARK proposals. This can be observed in the Figure of Merit column of the results Table. One could argue that the throughput of two proposals can not be compared because the security margin is not taken in account. However, we would like to emphasize that the security margin is very hard to measure as it greatly depends on the simplicity of the scheme, the amount of work spent by the cryptanalysts, etc. Unlike most of the lightweight hash functions proposed, in the case of PHOTON, we chose very simple to analyse internal permutations, thus directly leveraging the extensive analysis work already known for AES-like permutations. While 8 rounds over 12 of the internal permutations of PHOTON can be distinguished from a random permutation, we provide strong arguments that this is very unlikely to be much improved.

We did not include power figures in Table 2 for several reasons. First, the power consumption strongly depends on the technology used and cannot be compared between different technologies in a fair manner. Furthermore, simulated power figures strongly depend on the simulation method used, and the effort spent. Instead, we just briefly list the simulated power figures for our proposals here: 1.59, 2.29, 2.74, 4.01, and 4.55μW for serialized implementation of PHOTON-80/20/16, PHOTON-128/16/16, PHOTON-160/36/36, PHOTON-224/32/32, and PHOTON-256/32/32, respectively. The d-parallel implementations require 2.7,

[10] We synthesized the publicly available VHDL source code of U-QUARK using the same tool chain and ASIC library as for our proposals. The post-synthesis figures for U-QUARK are slightly higher than the previously published ones, *i.e.* 1400 GE instead of 1379 GE, which indicates that PHOTONs smaller footprint is not caused by a different tool chain. However, for comparison we took the previously available figures, which is in favour of QUARK.

Table 2. Overview of parameters, security level, and performance of several lightweight hash functions. Throughput and FOM figures have been derived at a clock frequency of 100 KHz. We marked by a * the preimage resistances of PHOTON-128/16/16, PHOTON-160/36/36, PHOTON-224/32/32 and PHOTON-256/32/32 in order to indicate that these PHOTON variants can achieve equal preimage resistance compared to its competitors by simply adding one more squeezing round. This will increase the hash output size n by r' bits and slightly reduce the throughput for small messages, while the area and the long message performances will remain the same.

Name	Ref.	Parameters				Security		Performance						
		n	c	r	r'	Pre	Col	Area [GE]	Latency [clk]		Throughput [kbps]		FOM [nb/clk/GE^2]	
									P/E	H	long	96-bit	long	96-bit
64-bit preimage resistance														
SQUASH	[38]	64	x	x	x	64	0	2646	31800	31800	0.2	0.15	0.29	0.14
DM-PRESENT-80		64	64	80	x	64	32	1600	547	547	14.63	5.85	57.13	19.04
DM-PRESENT-80		64	64	80	x	64	32	2213	33	33	242.42	96.67	495.01	165.00
DM-PRESENT-128		64	64	128	x	64	32	1886	559	559	22.90	8.59	64.37	32.19
DM-PRESENT-128		64	64	128	x	64	32	2530	33	33	387.88	145.45	605.98	302.99
KECCAK-f[200]		64	128	72	72	64	32	2520	900	900	8.00	5.33	12.6	8.4
PHOTON-80/20/16		80	80	20	16	64	40	865	708	3540	2.82	1.51	37.73	20.12
PHOTON-80/20/16		80	80	20	16	64	40	1168	132	660	15.15	8.08	111.13	59.27
64-bit collision resistance														
U-QUARK		136	128	8	8	128	64	1379	544	9248	1.47	0.61	7.73	3.20
U-QUARK		136	128	8	8	128	64	2392	68	1156	11.76	4.87	20.56	8.51
H-PRESENT-128		128	128	64	x	128	64	2330	559	559	11.45	5.72	21.09	10.54
H-PRESENT-128		128	128	64	x	128	64	4256	32	32	200.00	100.00	110.41	55.21
ARMADILLO2-B		128	128	64	x	128	64	4353	256	256	25.00	12.50	13.19	6.60
ARMADILLO2-B		128	128	64	x	128	64	6025	64	64	100.00	50.00	27.55	13.77
KECCAK-f[400]		128	256	144	144	128	64	5090	1000	1000	14.40	9.60	5.56	3.71
PHOTON-128/16/16		128	128	16	16	112*	64	1122	996	7968	1.61	0.69	12.78	5.48
PHOTON-128/16/16		128	128	16	16	112*	64	1708	156	1248	10.26	4.4	35.15	15.06
80-bit collision resistance														
D-QUARK		176	160	16	16	160	80	1702	704	7744	2.27	0.80	7.85	2.77
D-QUARK		176	160	16	16	160	80	2819	88	968	18.18	6.42	22.88	8.08
ARMADILLO2-C		160	160	80	x	160	80	5406	320	320	25.00	10.00	8.55	3.42
ARMADILLO2-C		160	160	80	x	160	80	7492	80	80	100.00	40.00	17.82	7.13
SHA-1	[29]	160	160	512	x	160	80	5527	344	344	148.84	27.91	48.72	9.14
PHOTON-160/36/36		160	160	36	36	124*	80	1396	1332	6660	2.70	1.03	13.87	5.28
PHOTON-160/36/36		160	160	36	36	124*	80	2117	180	900	20	7.62	44.64	17.01
112-bit collision resistance														
S-QUARK		256	224	32	32	224	112	2296	1024	8192	3.13	0.85	5.93	1.62
S-QUARK		256	224	32	32	224	112	4640	64	512	50.00	13.64	23.22	6.33
PHOTON-224/32/32		224	224	32	32	192*	112	1736	1716	12012	1.86	0.56	6.19	1.86
PHOTON-224/32/32		224	224	32	32	192*	112	2786	204	1428	15.69	4.71	20.21	6.06
128-bit collision resistance														
ARMADILLO2-E		256	256	128	x	256	128	8653	512	512	25.00	9.38	3.34	1.25
ARMADILLO2-E		256	256	128	x	256	128	11914	128	128	100.00	37.50	7.05	2.64
SHA-2	[18]	256	256	512	x	256	128	10868	1128	1128	45.39	8.51	3.84	0.72
PHOTON-256/32/32		256	256	32	32	224*	128	2177	996	7968	3.21	0.88	6.78	1.85
PHOTON-256/32/32		256	256	32	32	224*	128	4362	156	1248	20.51	5.59	10.78	2.94

3.45, 4.35, 6.5, and 8.38μW, respectively. This let us conclude that all PHOTON flavors seem to be suitable for ultra-constrained devices, such as passive RFID tags, which was one of our initial design goals.

4.4 Software implementation

We give in Table 3 our software implementation performances for the PHOTON variants. The processor used for the benchmarks is an Intel(R) Core(TM) i7 CPU Q 720 clocked at 1.60GHz. For comparison purposes, we also benchmarked the speed of an AES permutation (without the key schedule) and a modified version of it with a serially computable MDS matrix instead (the 4×4 matrix A given in Section 2.2). As expected, the table-based implementations reach the same speed for both versions. We also benchmarked other lightweight hash function designs. QUARK reference code, very likely to be optimizable, runs at 8k, 30k and 22k cycles per byte for U-QUARK, D-QUARK and S-QUARK, respectively. The optimized PRESENT code runs at 90 cycles per byte, hence the estimate speed for DM-PRESENT-80, DM-PRESENT-128 and H-PRESENT-128 are 72, 45 and 90 cycles per byte, respectively.

Table 3. Software performances in cycles per byte of the PHOTON variants for long messages.

PHOTON-80/20/16	PHOTON-128/16/16	PHOTON-160/36/36	PHOTON-224/32/32	PHOTON-256/32/32
95 c/B	156 c/B	116 c/B	227 c/B	157 c/B

5 Conclusion

We proposed PHOTON, the most lightweight hash-function family known so far, very close to the theoretical optimum. Our proposal is based on the well known AES design strategy, but we introduced a new mixing layer building method that perfectly fits small area scenarios. This allows us to directly leverage the extensive work done on AES and AES-like hash functions so as to provide good confidence in the security of our scheme. Due to page restrictions we refer to an extended version of this paper [1] for a detailed security analysis of PHOTON. Finally, PHOTON is not only the smallest hash function, but it also achieves excellent area/throughput trade-offs and we obtained very acceptable performances with simple software implementations.

Acknowledgement

The authors would like to thank the anonymous referees for their helpful comments. Also, we are very grateful to Dag Arne Osvik and AlpCode for providing an optimized Boolean representation of the PRESENT Sbox, to Jean-Philippe Aumasson for providing his cube testers source code and to Christina Boura for her help with zero-sum distinguishers.

References

1. The PHOTON Family of Lightweight Hash Functions. http://sites.google.com/site/photonhashfunction/.

2. E. Andreeva, B. Mennink, and B. Preneel. The Parazoa Family: Generalizing the Sponge Hash Functions. Cryptology ePrint Archive, Report 2011/028, 2011.
3. G. Avoine and P. Oechslin. A Scalable and Provably Secure Hash-Based RFID Protocol. In *PerCom Workshops*, pages 110–114. IEEE Computer Society, 2005.
4. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge functions. Ecrypt Hash Workshop 2007, May 2007.
5. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the Indifferentiability of the Sponge Construction. In Paterson [30], pages 181–197.
6. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Keccak specifications. Submission to NIST (Round 2), 2009.
7. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. Sponge-Based Pseudo-Random Number Generators. In S. Mangard and F.-X. Standaert, editors, *CHES*, volume 6225 of *LNCS*, pages 33–47. Springer, 2010.
8. G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche. On the security of the keyed sponge construction. In G. Leander and S. Thomsen, editors, *SKEW*, 2011.
9. A. Biryukov and D. Khovratovich. Related-Key Cryptanalysis of the Full AES-192 and AES-256. In M. Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 1–18. Springer, 2009.
10. A. Biryukov, D. Khovratovich, and I. Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In S. Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 231–249. Springer, 2009.
11. C. Blondeau, M. Naya-Plasencia, M. Videau, and E. Zenner. Cryptanalysis of ARMADILLO2. Cryptology ePrint Archive, Report 2011/160, 2011.
12. W. Bosma, J. Cannon, and C. Playoust. The Magma algebra system. I. The user language. *J. Symbolic Comput.*, 24(3-4):235–265, 1997. Computational algebra and number theory (London, 1993).
13. G. Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *LNCS*. Springer, 1990.
14. C. D. Cannière, O. Dunkelman, and M. Knezevic. KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In C. Clavier and K. Gaj, editors, *CHES*, volume 5747 of *LNCS*, pages 272–288. Springer, 2009.
15. D. Canright. A Very Compact S-Box for AES. In J. R. Rao and B. Sunar, editors, *CHES*, volume 3659 of *LNCS*, pages 441–455. Springer, 2005. The HDL specification is available at the author's official webpage `http://faculty.nps.edu/drcanrig/pub/index.html`.
16. J. Daemen and V. Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
17. I. Damgård. A Design Principle for Hash Functions. In Brassard [13], pages 416–427.
18. M. Feldhofer and C. Rechberger. A Case Against Currently Used Hash Functions in RFID Protocols. In R. Meersman, Z. Tari, and P. Herrero, editors, *OTM Workshops (1)*, volume 4277 of *LNCS*, pages 372–381. Springer, 2006.
19. T. Good and M. Benaissa. ASIC Hardware Performance. In M. J. B. Robshaw and O. Billet, editors, *The eSTREAM Finalists*, volume 4986 of *LNCS*, pages 267–293. Springer, 2008.
20. D. Henrici, J. Götze, and P. Müller. A Hash-based Pseudonymization Infrastructure for RFID Systems. In *SecPerU*, pages 22–27. IEEE Computer Society, 2006.
21. S. Hirose. Some Plausible Constructions of Double-Block-Length Hash Functions. In M. J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 210–225. Springer, 2006.

22. A. Juels and S. A. Weis. Authenticating Pervasive Devices with Human Protocols. In Shoup [33], pages 293–308.
23. S.-M. Lee, Y. J. Hwang, D. H. Lee, and J. I. Lim. Efficient Authentication for Low-Cost RFID Systems. In O. Gervasi, M. L. Gavrilova, V. Kumar, A. Laganà, H. P. Lee, Y. Mun, D. Taniar, and C. J. K. Tan, editors, *ICCSA (1)*, volume 3480 of *LNCS*, pages 619–627. Springer, 2005.
24. R. C. Merkle. One Way Hash Functions and DES. In Brassard [13], pages 428–446.
25. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of the AES. In Paterson [30].
26. National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard. `http://csrc.nist.gov`, April 1995.
27. National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard. `http://csrc.nist.gov`, August 2002.
28. National Institute of Standards and Technology. Announcing Request for Candidate Algorithm Nominations for a NewCryptographic Hash Algorithm (SHA-3) Family. *Federal Register*, 27(212):62212–62220, November 2007. Available:`http://csrc.nist.gov/groups/ST/hash/documents/FR_Notice_Nov07.pdf`(2008/10/17).
29. M. O'Neill. Low-Cost SHA-1 Hash Function Architecture for RFID Tags. In S. Dominikus and M. Aigner, editors, *RFIDSec*, 2008. Available via `http://events.iaik.tugraz.at/RFIDSec08/Papers/`.
30. K. G. Paterson, editor. *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *LNCS*. Springer, 2011.
31. T. Peyrin, H. Gilbert, F. Muller, and M. J. B. Robshaw. Combining Compression Functions and Block Cipher-Based Hash Functions. In X. Lai and K. Chen, editors, *ASIACRYPT*, volume 4284 of *LNCS*, pages 315–331. Springer, 2006.
32. C. Rolfes, A. Poschmann, G. Leander, and C. Paar. Ultra-Lightweight Implementations for Smart Devices - Security for 1000 Gate Equivalents. In G. Grimaud and F.-X. Standaert, editors, *CARDIS*, volume 5189 of *LNCS*, pages 89–103. Springer, 2008.
33. V. Shoup, editor. *Advances in Cryptology - CRYPTO 2005: 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings*, volume 3621 of *LNCS*. Springer, 2005.
34. Virtual Silicon Inc. 0.18 μm VIP Standard Cell Library Tape Out Ready, Part Number: UMCL18G212T3, Process: UMC Logic 0.18 μm Generic II Technology: 0.18μm, July 2004.
35. X. Wang, Y. L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In Shoup [33], pages 17–36.
36. X. Wang and H. Yu. How to Break MD5 and Other Hash Functions. In *EUROCRYPT*, pages 19–35, 2005.
37. X. Wang, H. Yu, and Y. L. Yin. Efficient Collision Search Attacks on SHA-0. In Shoup [33], pages 1–16.
38. S. Zhilyaev. Evaluating a new MAC for current and next generation RFID. Master's thesis, University of Massachusetts Amherst, 2010, available via `http://scholarworks.umass.edu/cgi/viewcontent.cgi?article=1477&context=theses`.