Comparing Lightweight Hash Functions – PHOTON & Quark

Tobias Meuser[1], Larissa Schmidt[1], and Alex Wiesmaier[1,2]

[1] Technische Universität Darmstadt, Germany
[2] AGT International

Abstract. In the last few years, much progress has been made in the research field of lightweight hash-functions. Due to the fact, that most conventional hash-functions are too expensive in terms of energy consumption and bad performance, hash-functions have been developed, which are meant to be used on resource-constrained devices. In this paper, we present and compare two promising hash-functions, PHOTON and Quark, in terms of design, security and performance.

Keywords: Internet of things (IoT); lightweight hash functions; PHOTON; QUARK

1 Introduction

The Internet of Things (IoT) is one of the most promising research fields. In Germany for example, the government supports the research with the initiative IKT2020. The idea of the IoT is the overall presence of the internet in things of the everyday life. Today, internet connected devices like smart phones, tablets, smart tv's and multi media applications in cars are quite normal. But the vision of the IoT is the distribution of so called smart objects in every sector of the every day life. An common example is a fridge noticing the missing milk and ordering it via internet. The applications that manage these communication flows are often called IoT applications. In these applications, resource constrained devices ("things" or smart objects) with internet access are used to communicate with each other and servers. The most common examples in the literature are RFID tags and systems-on-chip. The RFID technology was the basis for the identification of objects in processes and therefore has a leading role in the development of the vision IoT. But other technologies could be used as well. For all of them, the devices should encrypt their messages and ensure their integrity. If two flash lights would be communicating and an attacker could change a message, a traffic desaster could be the result. Another example could be a fridge buying too much milk. Every use case has other security requirements. A common requirement to every use case is based on the resource restriction of the IoT devices. Therefore, securing communication between the devices needs new cryptographic primitives that satisfy the following explicit requirements: The algorithm and implementation should ...

1. be very efficient due to the resource restriction of the devices.
2. achieve an adequate security level for different use cases.
3. not be vulnerable to known attacks.

The machine-to-machine communication comprises server-to-server, server-to-device and device-to-device communication. For the latter, the range of cryptographic primitives is enhanced with so-called lightweight cryptography. These resource constrained devices only allow a small number of gates, a limited energy consumption and a decreased number of cycles for computing values with the primitives. Therefore, the existing primitives are not usable. For encryption, lightweight stream and block ciphers were introduced in the last years and for message authentication codes, pseudorandom generators and key derivation functions, lightweight hash functions have been developed. In this paper, we present the hash functions PHOTON [18] and Quark [3]. Those two both are based on some kind of Sponge function [9] and are quite performant. In comparison to many other hash algorithms this two are not based on a standard (non lightweight) hash algorithm. As the highest security level is not always applicable and sometimes necessary for IoT applications, several instances of the two hash functions exist to meet different use cases.

1.1 Related work

Our work focuses on two lightweight hash functions that are compared to each other and set in context to related ciphers and hashes. Papers and works in the same area are those of Poschmann [23] and Balasch et al. [7].
Poschmann [23] did his research in lightweight cryptography in 2009. His focus was on the PRESENT [12] cipher. PRESENT is a block cipher and therefore out of our scope. But it influenced the development of lightweight ciphers and hashes. Balasch et al. [7] did a quantitative comparison of several lightweight hash functions in their paper. We refer to their work in section 6 where we compare the two ciphers to others. Instead of a quantitative comparison like Balasch et al. [7] did or a complete overview over the area of lightweight cryptography comparable to the dissertation of Poschmann [23], our work summarizes other papers and gives an overview of the state of the art in lightweight hash functions with a special focus on the hash functions Photon and Quark. These two functions are rather new, not completely analysed and a very interesting approach in this area. Additionally, they are very similar in use cases, construction and general tendencies. Therefore it is straightforward to compare them and find out the differences. Our goal was not to compare two completely different hash functions. We also distinguish them from others in section 6 with help of statistics and other papers.
The work of Anandakumar et al. [1] was not directly taken into account in this work but belongs to interesting works for further reading.

Related (lightweight) hash functions Related to PHOTON and Quark are other hashes that can be divided into different categories. We can distinguish

between lightweight and standard hash algorithms. Very common and important, but not in the range of lightweight hash functions, is the family of the secure hash algorithms (SHA). Today, SHA-1, SHA-2 and SHA-3 are used. Unmodified, they need far too much resources for devices in the Internet of Things. Therefore, lightweight hash algorithms were introduced. The family of lightweight hash functions comprises:

1. new hash functions that may use existing construction schemes and that are designed especially for IoT use cases (e.g. Spongent [11], Armadillo [6], Spritz [24] and SQUASH [26]),
2. existing hash functions that are modified to become more lightweight (e.g. Reduced Keccak [20] that is the lightweight variant of the Keccak hash algorithm, better known as SHA-3.)
3. and existing (lightweight) ciphers that are transformed to hash functions (Poschmann [23] explains in the sections 6.5 and 6.6 the functionality of DM- and H-PRESENT that are hash algorithms based on the lightweight block cipher PRESENT.).

The referred hash functions are only a part of the recent research and listed as examples for related algorithms.

Related (lightweight) ciphers In general, the range of lightweight ciphers is very broad. The eStream Project [5] found three lightweight stream ciphers: Grain, MICKEY and Trivium. Grain and the block cipher KATAN [17] influenced the hash function Quark [3]. Other block ciphers that belong to related ciphers are PRESENT [12], the Cube [8] block cipher family and SEA [27].

1.2 Security features of hash functions

In the design of cryptographic hash functions it is crucial to avoid so called collisions [25]. This means that two input values are mapped to the same hash value. These collisions are used within attacks to the applications of the hash function. Therefore, a hash function should be collision resistant [13]. Two other requirements to hash functions are preimage and second preimage resistance. If a hash function is a one way function, it should meet the features second preimage and preimage resistance. A collision resistant function owns the three properties.

Collision resistance: In order to be collision resistant, it must be hard to find any two inputs x and x' for a certain hash function such that the function outputs the same value: $h(x) = h(x')$ for $x \neq x'$. Hard means in that case that there is not a more efficient way then just brute forcing. Anyway, because it is not possible to find a message with an equal hash value as a given message, this weakness is not as easy to exploit as weaknesses in the second preimage resistance.

Second preimage resistance: A hash function is 2nd preimage resistant if it is not

efficiently possible[3] to find a collision (x, x') for given input x with $h(x) = h(x')$. This is important because if an hash function is not second preimage resistant, then it is possible to find another message with the same hash value which has been signed using the hash function. Therefore an attacker could attach the signature to the other message and has a valid signature for the other message.

Preimage resistance: It should be infeasible to find an x to a given y, such that $h(x) = y$. The preimage resistance is also called "onewayness". This means an attacker is not able to invert the function. If it is possible to invert a hash function, then the function is neither collision nor second preimage resistant. Forging would be quite easy in this case.

1.3 Attacks against hash functions

In this subsection general attacks against hash functions are explained. The applicability to the hash functions PHOTON and Quark is checked later in Sections 4.3 and 5.3.

Birthday attack The birthday attack [14] is based on the theory of the birthday paradox. The attacker tries to find a collision for a k-bit hash function by generating $2^{k/2}$ messages. The possibility to find a collision is $k \geq \frac{1+\sqrt{1+(8ln(2))*2^n}}{2}$ (see Buchmann [13] at pp. 193). The probability to find in these $2^{k/2}$ messages a collision is more than 50%. This shows that the security level of a hash function depends on its output length.

Cube attacks and testers Cube attacks [2] are applicable to block ciphers, stream ciphers and hash functions. They are a high order differential cryptanalysis techniques. In this attack the secret variables of the underlying algebraic structure are extracted by solving a system of polynomial equations. Cube attacks were successfully applied to polynomials of d^n secret variables.

Side channel attacks Side Channel attacks are directed against an implementation of a cryptographic primitive. Therefore, such attacks cannot be completely avoided in the design of a hash algorithm because even if they were considered in the design process, the implementation can still be vulnerable to these attacks. Therefore, it is necessary to check a hardware implementation to see if this implementation with mechanisms to prevent side channel attacks is still usable for IoT use cases.

Slide (resynchronisation) attacks Slide attacks are directed against the key schedule of the algorithm. The number of cycles is therefore irrelevant and does not provide a higher security against this kind of attack. Related to the slide

[3] All possible methods are not much better than just brute forcing.

attacks are slide resynchronisation attacks. Kuccuk et al. [21] found a slide resynchronisation attack against Grain in 2006.

2 Constraints

In the IoT, different devices communicate with each other. The authors tend to use RFID chips as example for IoT devices. Most of the RFID chips contain less than 10000 gates, of which only 2000 are meant to be used for security purposes. Moreover, the maximum CPU *load* of such a chip is around 100 kHz. Due to that fact, most of the common hash algorithms like SHA-3 are not applicable. Researches have been done to develop hash functions, that are suitable to run on the given system requirements.
The two hash algorithms investigated in this paper both manage to run *with low resources load*, but have some issues in terms of security and throughput. In cipher design, a tradeoff between cost and security is necessary. The authors suggest that due to the non-criticality of the data transmitted by a single RFID chip. The complete mass of all RFID chips's data is critical. Therefore, the data transmitted to other chips or servers should not be transmitted in clear. But a small cipher with a reduced security is still secure enough because attacker must compromise the hashed data of all RFIDs. This is still with the presented hash functions a rather difficult challenge.
Typically in IoT scenarios, the availability might be more important than the confidentiality. But this depends on the use case.

3 Construction scheme of PHOTON and Quark

The central component in the design of a hash function is the construction method. For Quark and PHOTON, sponge construction introduced by Bertoni et al. [9] was preferred to the classical Merkle Damgard construction.

3.1 Merkle-Damgård construction

The Merkle-Damgård Construction Scheme is used to build hash functions [15]. A compression function $f(x)$ is transformed into a secure hash function. In detail, the input message is divided into n Blocks of length l. The last block is then filled with padding bits. The blocks are enumerated with x_1 to x_n. The hash function is now constructed with $h_i = f(h_{i-1}||x_i)$. h_0 is a special case which is filled with zeros. The amount of zeros is defined through the output size of the hash function. h_n can then be used as final hash-value.

3.2 Sponge construction

The sponge construction offers a new way of building a hash function based on a fixed permutation. Usual arguments to prefer sponges include:

1. Sponge relies on a single unkeyed permutation.
2. The sponge construction is in-differentiable from a random oracle (see Section 5.3).
3. Sponge function are able to generate Message Authentication Codes.
4. The method is very flexible because the parameters offer a wide range. Developing a new instance of a hash algorithm can be done by changing the parameters only.

The sponge construction is for example also used in Keccak and Spritz [24].

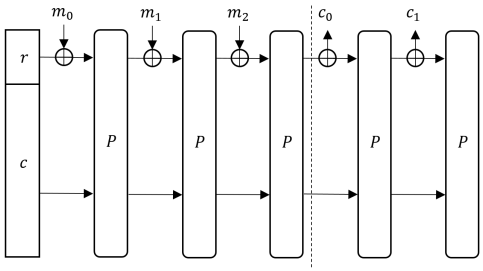

Fig. 1. Sponge Construction, adapted from Fig. 1 of [3]

The Sponge construction has the following parameters:

1. The width b=r+c defines the block size used for the permutation P. This block size is the sum of the rate r and the capacity c:
   (a) The rate r is the block length used in the algorithm. It is the length of one message part m_x ($x \in$ N).
   (b) The capacity c directly influences the security level of the hash function. It is the part of the previous permutation that is used for the next one.
2. The output length n is in other construction methods the main parameter for the security level.

The message $m = \{m_0, m_1, m_2, m_3\}$ is used in the example of figure 1 for the hash creation. In this example a 4-block message is used. The Permutation P is applied to the different parts of m.

4 PHOTON

4.1 General information

PHOTON is a lightweight hash function developed by Guo et al. [18] and is mostly meant to be used in RFID applications. Most of the implementations are hardware based, nevertheless it is possible to implement PHOTON in software. PHOTON is not very performant for big amounts of data because the bitrate is chosen to be very small in order to minimize the memory used.

4.2 Design

Guo et al. assume that due to the fact that RFID tags are not protecting objects of high value, tag-based applications do not require such high security primitives like they are required in other applications. A security level of an 64-/80-Bit output should be appropriate for the use in RFID tags because of the fact that the data to be protected is not that valueable for an attacker. In order to do serious damage, the attacker normally needs to break the cryptosystem within a very short period of time. If the attack takes to much time, most of the information are outdated.

Extended sponge functions The sponge function has been introduced by Bertoni et al. [9] It offers a new way of building hash functions from a fixed permutation, in which the internal state S is composed of a capacity of c bits and a bitrate of r bits. The expected complexities of a hash function generated using a sponge function are:

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

One big advantage of Sponge Functions is the fact that they can be used to generate Message Authentication Codes. To minimize the memory used, the sponge is created with a capacity equal to the output size of the hash. Moreover very small bitrates are used in order to keep the costs of each iteration quite low. Another point for choosing a small bitrate is that the information processed in a standard RFID chip are normally very small, therefor even small bitrates do not hurt the performance that much.

Table 1. Overview of PHOTON's properties

	PHOTON-128/16/16	PHOTON-160/36/36	PHOTON-224/32/32
Energy consumption (mean) [μW]	2,29/3,45	2,74/4,35	4,01/6,50
Gate equivalents [GE]	1122/1707	1396/2117	1736/2786
Implementation preference	SW+HW		
Security			
Collision resistance	64	80	64
Preimage resistance	112	124	112

An AES-like internal permutation AES-like permutations are being chosen because of the confidence in the design strategy of AES and AES based hash algorithms. In each step, a fixed permutation is used. I comparison to AES, in every application of the algorithm the same key is used. This is possible because the purpose of PHOTON is not to cipher but to create a hash value, which needs

to be the same for every application of the algorithm. The SBoxes used are quite similar to the ones of PRESENT. Each iteration constists of these four steps:

AddConstants The constants were chosen by Guo et. al. that each round computation is different. In this step they also reduce the security in order to archive better performance by modifying only the first column.

SubCells The size of the SBoxes was chosen even to avoid odd message block sizes or even capacities (if also the amount of message blocks is odd). Remember, using the sponge function each time a message of the size of the bitrate is squeezed out of or absorbed by the sponge in order to apply these permutations on it. In order to do no unnecessary work, the message block size should devide the sponge capacity in order to do no unnecessary rounds. In the end, Guo et al use SBoxes of 4 or 8 bit.

ShiftRows Because of the fact that the chosen SBox got as many columns as rows, each row is shifted by its row index to the left.

MixColumnsSerial The underlying matrix is a circulant matrix. The AES MixColumnsSerial is not applicable in this case because the coefficients of these matrices are very bad for smaller implementations. This issue is solved by updating only the last cell of the input vector with the matrix. After this, we rotate the vector by one position towards the top. Therefore, the new MixColumnsSerial is composed of d applications of our matrix on the input vector.

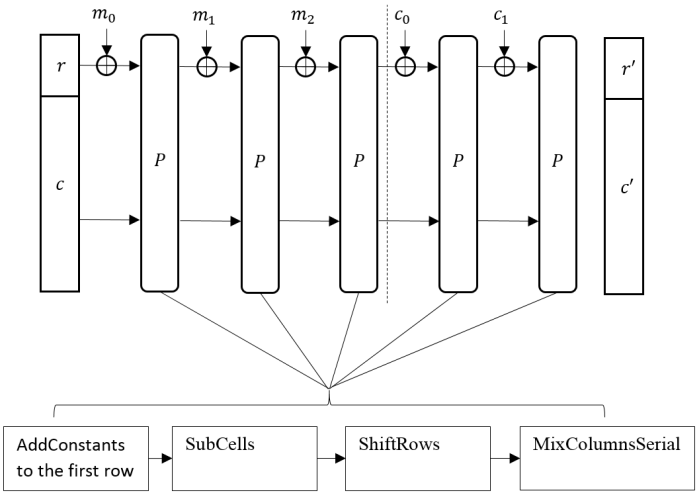

Fig. 2. Sponge construction in PHOTON, adapted from Fig. 1 of [18]

The PHOTON hash function family Every variant of the PHOTON hash-algorithm is fully defined by the size of its hash output, which is between $64 \leq n \leq 256$, its input and its in-/output bitrate r/r'. The input bitrate defines how many bytes are absorbed by the sponge. After each absorption the permuation steps are executed. After the whole message has been absorbed by the sponge, it is squeezed out with the output bitrate. After each squeeze, the permutations are applied, too.

Table 2. Overview of the PHOTON versions

Instance	Input Rate	Output Rate	Hash Size
PHOTON-80/20/16	16	20	80
PHOTON-128/16/16	16	16	128
PHOTON-160/36/36	36	6	160
PHOTON-224/32/32	32	32	224
PHOTON-256/32/32	32	32	256

4.3 Attacks

There are different attacks on PHOTON, but yet none of them managed to break PHOTON. One point for the security of PHOTON is the usage of the extended sponge function. Especially for low bitrates this adds a lot of security because there are many round in which either clearcode is absorbed by the sponge or the hash is squeezed of the sponge. In every round the permutation is applied and therefor a lot of complexity is added.

Differential/linear cryptanalysis The differential cryptanalysis analyses the effects of changes in the plaintext to the ciphertext. As already mentioned, PHOTON's internal primitives are AES-based and therefore the work done in the past to break AES can be reused. By adapting the wire-trail strategy, we can easily obtain a bound for the number of active Sboxes for four rounds. For any non-null differential path, there will be at least $(d+1)^2$ Sboxes active. Due to the fact that the differential probability of PRESENT SBoxes[4] is 2^{-2} (for AES it is 2^{-6}), the best differential path probability on four rounds is maximum $2^{-2\cdot(d-1)^2}$ for $s = 4$ and $2^{-6\cdot(d-1)^2}$ for $s = 8$. In case of PHOTON this does not work due to the fact that PHOTON does not require any key material in the internal permutations. Therefore it is not possible to leverage the key by attacking this permutations.

Rebound and super-sbox attacks The original rebound attack [22] has been introduced recently and is currently the best known method to analyse AES-like

[4] The SBoxes used by PHOTON are quite similar to the ones of PRESENT.

permutations, which do not depend on a certain secret. Currently, all versions of PHOTON are quite safe against this kind of attack. The attacker can only calculate the results for 8 rounds, more rounds can not be reached even with more calculation power. The fact that in PHOTON no key is involved in the internal permutations, impairs that there can not be any improvement of the attack using a certain weakness of the secret key.

Cube testers and algebraic attacks Cube tester attacks have already been applied to other hash algorithms like Trivium and MD6 [2]. This approach was applied by Guo et al to PHOTON and was only able to recover 3 rounds of the algorithm. In PHOTON there are two types of Sboxes being used, the ones of AES and the ones of PRESENT. In case of P_{144}, which is used in PHOTON-128/16/16, around 9000 equations in around 3500 variables are need to be solved. Compared that to AES, where 6400 equations in around 2500 variables need to be solved, it is assumed that PHOTON is quite save against this kind of attack[5].

Other cryptanalysis Recently the slide attack normally being used to attack block cipher has been applied to attack sponge-like hash functions. This attack tries to exploit the degree of self-similarity of a permutation. Due to the AddConstants operation (compare 4.2), all permutations are made different. Because of that, it is impossible to perform the slide attack.

5 Quark

5.1 General information

One of the recently introduced lightweight hash function is Quark [3]. Instead of a standard algorithm's lighter version such as reduced Keccak [20], Quark is a new scheme that uses components of the stream cipher Grain belonging to the eStream Project [5] and the block cipher KATAN [17] for its core algorithm. The developers of Quark wanted the whole design to be lightweight: the security level, the construction and the core algorithm. This means that the security level does not reach a value that would be sufficient for standard applications. In this use case, the lower security level is sufficient, because only the aggregated information of all sensor nodes (or something similar) is important and should be protected. An attacker would have to decrypt all the information of all nodes to get to a valuable result. Due to the mass of information to attack, a lightweight cipher can prevent the decryption of all devices. The construction and core algorithm are conformist to the hardware of chips. In summary, Aumasson et al. wanted to create a hash function that fulfils the security and hardware requirements of RFID applications as example for an IoT device. An exception is the new C-Quark [4] that offers a higher security level for non-lightweight applications with the requirement of less gates that are used by the algorithm.

[5] Remember that the value of the information protected by PHOTON is in general much smaller than the value protected by AES.

5.2 Design and implementation

The central component in the design of a hash function is the construction method. For Quark, sponge construction was preferred to the classical Merkle Damgard construction that is explained in section 3.1. The general construction scheme is introduced in section 3.2.

Sponge construction in Quark The construction scheme in Quark is the same as in figure 1. The permutations of sponge consist in Quark of *shift registers* and *boolean functions* like proposed in Grain and KATAN [17]:

Feedback shift registers: Shift registers are chained flip-flops. They are known as building block for stream ciphers. Feedback shift registers have additional XOR registers for the feedback function. They can be implemented in hardware, for example with field programmable gate arrays, or software.
A Quark permutation consists of two non-linear feedback shift registers (NFSR or NLFSR) and one linear feedback shift register (LFSR). The one linear register was introduced in KATAN for replacing a simple digital counter [17]. A LFSR is a shift register with n flip flops. Because of its linearity, a pseudo random number generator consisting only of LFSRs is predictable. The non-linear feedback shift registers prevent the cipher from being attacked.

Boolean Functions: The permutation p not only consists of shift registers but also of the boolean functions f, g and h. They are combined with XORs to the shift registers. The properties of the functions are: non-linearity, resilience, algebraic degree and density. An alternative would have been S-boxes like in AES.

In general, the construction has three phases: the initialization phase where the m is changed so that it is a multiple of r, the absorbing phase where the permutations are applied with the XORed part of the message m and the squeezing phase where the output z is produced. The second and the third phase are modelled in Fig. 5.2. In summary, for the output the state of Quark is updated 4b times.

Instances of Quark Quark has three standard instances: U-Quark, D-Quark and S-Quark. Per instance, there are differences in the parameters and boolean functions. In Table 3, the values of the parameters to the sponge construction are stated for the different instances. The values are given in bits.

The last instance that was introduced is C-Quark. It is also called heavy Quark [4]. C-Quark is used for multimedia systems-on-chip. These systems have more ressources than RFID applications for example and therefore can reach a higher security level.

The general phases of C-Quark are the same, but the sponge construction is not used in its pure form. It is adapted for an authenticated encryption scheme with associated data (AEAD). The first cipher that introduced AEAD is Spritz

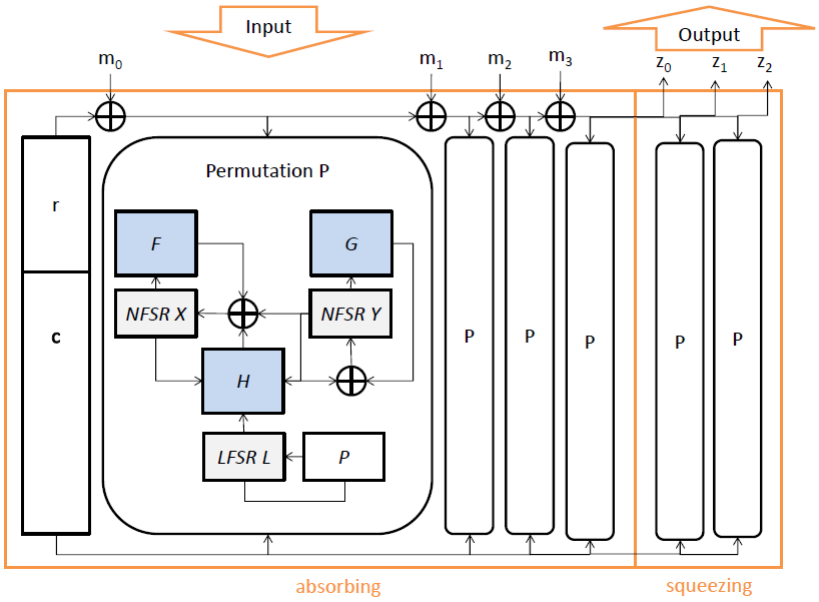

Fig. 3. Sponge construction in Quark, adapted from Fig. 1 and 2 of [3]

Table 3. Overview of the instances of Quark

Instance	Rate	Capacity	Width	Rounds	Digest
Quark	r	c	b=r+c	4b	n
U-Quark	8	128	136	544	136
D-Quark	16	160	176	704	176
S-Quark	32	224	256	1024	256
C-Quark	64	320	384	1536	384

[24]. AEAD is a duplex construction [10]. The duplex construction has the same rate as the sponge construction. This means that the input blocks m and the output blocks z in Fig. 5.2 are called by the authenticated encryption and sponge construction in parallel. With this sort of a AEAD that is named by the Aumasson et al. [4] as C-QuarkWrap it is possible to implement a hash function. The implementation only has 2b rounds instead of the 4b rounds in the other instances. The functions differ and the shift registers consist of 16 bits instead of 10 bits.

Implementation in hardware The Quark instances are optimized for hardware. There are three characteristics that support the lightweight hashing:

1. Feed forward values that would normally acquire a lot of memory are missing.
2. Shift registers that are easily implemented in hardware.

3. Space or time implementation trade-offs can be made.

Quark can be implemented in hardware two ways: as serial or parallel architecture. The serial architecture contains one permutation module and the three functions with the shift once per cycle in a most compact design. The parallel architecture makes it possible to have more permutation modules that can increase the number of rounds computed per cycle.
Aumasson et al. [3] used for the performance analysis of the Quark instances a generic way. They simulated a hardware implementation to measure the power and ground distribution. With a frequency of 100kHz, two consecutive messages with a length of 512b were processed with 2,5 μW. The peak volume was never too high for restrained systems (27% of the mean value). For the statistical power analysis, generated Value Change Dumps (VCD) were used. The number of gate equivalents (GE) can be found in Table 4. They depend on the Quark instance and influence the energy consumption and the evaluation speed. The different security levels that Quark can reach also depend on the instance and will be explained in the next section.

Table 4. Overview of Quark's properties

	U-Quark	D-Quark	S-Quark
Energy consumption (mean) [μW]	2,44/4,07	3,1/4,76	4,35/8,39
Gate equivalents [GE]	1379/2392	1702/2819	2296/4640
Implementation preference	HW	HW	HW

5.3 Attacks and security

Table 5. Security overview of the standard instances introduced by [3] and the extended version C-Quark [4].

Instance	c	Collision resistance	2nd preimage resistance	Preimage resistance
Quark	c	$2^{c/2}$	$2^{c/2}$	2^{c}
U-Quark	128	2^{64}	2^{64}	2^{128}
D-Quark	160	2^{80}	2^{80}	2^{160}
S-Quark	224	2^{112}	2^{112}	2^{224}
C-Quark	320	2^{160}	2^{160}	2^{320}

Aumasson et al. [3] report the difference of the hash function's digest to the security level. The capacity c stands for the security level in the construction. In Table 5 the security levels are listed. But these levels do not only show how secure an algorithm is. Exact security analyses are necessary. For an overview,

we explain some attacks in this section. But this list only contains examples and is not exhaustive.

Differentiating attacks Bertoni et al. [9] proved the sponge construction indifferentiable from a random oracle. The concrete bound is the capacity c. The (multi)collision resistance, (2nd) pre-image resistance, resistance against long-message and herding attacks requires only the usage of random transformation or permutation. These attacks are directed against the properties of the construction that is called *hermetic sponge* strategy by Aumasson et al. [3].
They also applied two sorts of differential attacks on Quark: the simple truncated differential attacks and conditional differential attacks. The results show that the methods are unlikely to be successful in the full 4b-round permutation of Quark.

Cube attacks and testers The cryptanalysis technique of cube attacks and testers is explained in general in section 1.3. It is also tested on several ciphers and hash functions:

1. The stream cipher Trivium [16] of the eStream project was broken by a cube attack. The complete key of 673 initialization rounds was extracted in 2^{19} bit operations.
2. Cube testers were applied to Grain-128.
3. Aumasson et al. [3] tested these two methods against Quark and found a "comfortable security margin", but a concrete comparison of the algorithms in a security analysis is missing.

Slide resynchronisation and side channel attacks Slide resynchronisation attacks were applied on block ciphers like Grain [21]. The same method for hash functions is called *slide distinguisher* by Aumasson et al. [3]. This attack is avoided because each round is dependent on a bit counting from a fixed value. Thus, it is impossible to obtain two valid states.
The known algorithm of Quark is not protected against side channel attacks. Protection by hiding and masking would increase the number of gates by three times[6].

6 Comparison

6.1 Design

The design of PHOTON and Quark is quite similar: they are both implemented with the sponge construction scheme. Quark's permutations are boolean functions and shift registers influenced by KATAN and Grain. PHOTON implements the SBoxes of the AES-cipher. Quark is only optimized for hardware, whereas PHOTON can be applied to hardware and software equally.

[6] See section 5.5 of Aumasson et al. [3].

6.2 Security and efficiency

Balasch et al. [7] compared several lightweight hashes in their paper. A part of this comparison can be found in Tables 6 and 7. It is obvious that the compared instances of Quark and PHOTON in Table 6, namely S-Quark compared to PHOTON-256/32/32 and D-Quark to PHOTON-160/36/36, with a similar security bound require almost the same size of state RAM. The other two hash functions SPONGENT and reduced Keccak have increased values. PHOTON needs twice the stack than the other implemented functions. The code size of

Table 6. Memory consumption [7]

Hash-Function	digest size [bits]	code size [bytes]	RAM [bytes] data	state	stack
S-Quark	256	1106	4	60	5
PHOTON-256/32/32	256	1244	4	68	10
SPONGENT-256/256/128	256	364	16	96	5
Keccak (r=144, c=256)	256	608	18	92	4
D-Quark	160	974	2	42	5
PHOTON-160/36/36	160	764	29	39	11
SPONGENT-160/160/80	160	598	10	60	6
Keccak (r=40, c=160)	160	752	5	45	3

S-Quark and PHOTON-256/32/32 is much higher, but the memory consumption for data only a fourth or less than for SPONGENT-256/256/128 and Keccak (r=144, c=256). In Table 7, you see that the cycle count of PHOTON-256/32/32 is much more lower than S-Quark's. The difference between D-Quark and PHOTON-160/36/36 is relative small. Keccak is the winner in this topic and SPONGENT needs even more cycles than Quark.

Table 7. Performance Evaluation [7]

Hash-Function	digest size [bits]	code size [bytes]	cycle count (message length) (8-byte)	(50-byte)	(100-byte)	(500-byte)
S-Quark	256	1106	708 783	1 417 611	2 339 023	9 427023
PHOTON-256/32/32	256	1244	254 871	486 629	787 896	3 105 396
SPONGENT-256/256/128	256	364	1 542 923	3 856 916	6 170 900	25 454 100
Keccak (r=144, c=256)	256	608	90 824	181 466	37 221	1 313 291
D-Quark	160	974	631 871	1 516 685	2 570 035	10 996 835
PHOTON-160/36/36	160	764	620 921	1 655 364	2 793 265	11 999 914
SPONGENT-160/160/80	160	598	795 294	2 783 241	4 771 186	20 674 746
Keccak (r=40, c=160)	160	752	58 063	162 347	278 269	1 205 627

Guo et al. [19] published their research in 2011 concerning technology dependence of lightweight hash functions. Their work shows that within implementation comparison, the instances of PHOTON and Quark clearly have the best results for latency[7] and area[8] with increasing security level. In Table 1 and Table 4, the mean energy consumption for two implementations with different numbers of gates is compared. PHOTON requires a lesser number of gates and therefore has a lesser mean energy consumption.
Altogether, the two functions offer good security bounds while being very efficient. For a memory constrained scenario, Quark and PHOTON should be preferred to SPONGENT and Keccak with a tendency to use Quark. If the performance of the hash function is important in a use case, PHOTON or Keccak should be used.

Table 8. Dependence between Security and Area

Hash Function	Collision Bits	Preimage Bits	Area (GE)
DM-PRESENT-80	32	64	1600
DM-PRESENT-128	32	64	1886
PHOTON-80/20/16	40	64	865
PHOTON-128/16/16	64	112	1122
H-PRESENT-128	64	128	2330
U-Quark	64	128	1379
ARMADILLO2-B	64	128	4353
PHOTON-160/36/36	80	128	1396
D-Quark	80	160	1702
ARMADILLO2-C	80	160	5406
C-PRESENT-192	96	192	4600
PHOTON-224/32/32	112	192	1736
S-Quark	112	224	2296
PHOTON-256/32/32	128	224	2177
ARMADILLO2-E	128	256	8653

6.3 Critic to custom performance comparisons

The given performance analysis refers to the attributes energy consumption, area (gate equivalents) and latency in cycles. Guo et al. [19] criticize current cost statements in research papers. They have done two case studies with Quark and CubeHash and an overall comparison that is partly shown in the table 8[9]. Cryptographic engineers refer in a different way to the general three design

[7] Compare with table 7.
[8] See table 8.
[9] The values are taken from the figure 'Technology dependence of existing lightweight hash implementations' of Guo et al. [19]

fields. The relation between algorithm specification and hardware architecture is known, but the influences of silicon implementation are broadly ignored. Aumasson et al. [3] only use a simulation to get the implementation measures.
A problem is the dependence of given attributes for efficiency measures to the silicon implementation. The necessary area and power depend on the hardware. Only latency is independent. Therefore, the given tables have a lower meaning because the values are only tendencies. The authors propose a technology dependent cost analysis.
Balasch et al. [7] implemented hash functions on ATMEL AVR ATtiny 8-bit micro controller. They discovered that RAM use and code sizes are sufficient low for all algorithms. However, the sponge construction additionally allows a reduction of RAM use. Lightweight primitives have in general large cycle counts. This means that the overall efficiency is generally low in this implementation context. The peak power (energy consumption) is important to restrained systems, but most of the time not mentioned.

7 Conclusion

The hash-algorithms Quark and PHOTON are quite similar to each other in terms of performance and security. Neither of them has been broken yet and they both offer sufficient security for the data processed by RFID chips for example. One disadvantage of Quark in comparison to PHOTON is the lack of optimization for software purposes. However, hardware implementations are in general more efficient and preferred for the use in the Internet of Things. Quark as well as PHOTON are much more performant compared to most of the other hash-functions. They use a quite little amount of gates for their calculation. All in all, PHOTON and Quark are lightweight hashes that have a balanced trade-off between security and efficiency.

References

1. Anandakumar, N.N., Peyrin, T., Poschmann, A.: A very compact fpga implementation of led and photon. In: Progress in Cryptology–INDOCRYPT 2014, pp. 304–321. Springer (2014)
2. Aumasson, J.P., Dinur, I., Meier, W., Shamir, A.: Cube testers and key recovery attacks on reduced-round md6 and trivium. In: Fast Software Encryption. pp. 1–22. Springer (2009)
3. Aumasson, J.P., Henzen, L., Meier, W., Naya-Plasencia, M.: Quark: A lightweight hash. Journal of cryptology 26(2), 313–339 (2013)
4. Aumasson, J.P., Knellwolf, S., Meier, W.: Heavy quark for secure aead. DIAC-Directions in Authenticated Ciphers (2012)
5. Babbage, S., Canniere, C., Canteaut, A., Cid, C., Gilbert, H., Johansson, T., Parker, M., Preneel, B., Rijmen, V., Robshaw, M.: The estream portfolio. eSTREAM, ECRYPT Stream Cipher Project (2008)
6. Badel, S., Dağtekin, N., Nakahara Jr, J., Ouafi, K., Reffé, N., Sepehrdad, P., Sušil, P., Vaudenay, S.: Armadillo: a multi-purpose cryptographic primitive dedicated to

hardware. In: Cryptographic Hardware and Embedded Systems, CHES 2010, pp. 398–412. Springer (2010)
7. Balasch, J., Ege, B., Eisenbarth, T., Gérard, B., Gong, Z., Güneysu, T., Heyse, S., Kerckhof, S., Koeune, F., Plos, T., et al.: Compact implementation and performance evaluation of hash functions in attiny devices. Springer (2013)
8. Berger, T.P., Francq, J., Minier, M.: Cube cipher: A family of quasi-involutive block ciphers easy to mask. In: Codes, Cryptology, and Information Security, pp. 89–105. Springer (2015)
9. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the indifferentiability of the sponge construction. In: Advances in Cryptology–EUROCRYPT 2008, pp. 181–197. Springer (2008)
10. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Duplexing the sponge: single-pass authenticated encryption and other applications. In: Selected Areas in Cryptography. pp. 320–337. Springer (2012)
11. Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varıcı, K., Verbauwhede, I.: Spongent: A lightweight hash function. In: Cryptographic Hardware and Embedded Systems–CHES 2011, pp. 312–325. Springer (2011)
12. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. Springer (2007)
13. Buchmann, J.: Introduction to cryptography. Springer Science & Business Media (2004)
14. Coppersmith, D.: Another birthday attack. In: Advances in CryptologyCRYPTO85 Proceedings. pp. 14–17. Springer (1986)
15. Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-damgård revisited: How to construct a hash function. In: Advances in Cryptology–CRYPTO 2005. pp. 430–448. Springer (2005)
16. De Cannière, C.: Trivium: A stream cipher construction inspired by block cipher design principles. In: Information Security, pp. 171–186. Springer (2006)
17. De Canniere, C., Dunkelman, O., Knežević, M.: Katan and ktantana family of small and efficient hardware-oriented block ciphers. In: Cryptographic Hardware and Embedded Systems-CHES 2009, pp. 272–288. Springer (2009)
18. Guo, J., Peyrin, T., Poschmann, A.: The photon family of lightweight hash functions. In: Advances in Cryptology–CRYPTO 2011, pp. 222–239. Springer (2011)
19. Guo, X., Schaumont, P.: The technology dependence of lightweight hash implementation cost. In: ECRYPT Workshop on Lightweight Cryptography 2011. pp. 0–0 (2011)
20. Kavun, E.B., Yalcin, T.: A lightweight implementation of keccak hash function for radio-frequency identification applications. In: Radio frequency identification: security and privacy issues, pp. 258–269. Springer (2010)
21. Küçük, Ö.: Slide resynchronization attack on the initialization of grain 1.0. eSTREAM, ECRYPT Stream Cipher Project, Report 44, 2006 (2006)
22. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The rebound attack: Cryptanalysis of reduced whirlpool and grøstl. In: Fast Software Encryption. pp. 260–276. Springer (2009)
23. Poschmann, A.Y.: Lightweight cryptography: cryptographic engineering for a pervasive world. In: Ph. D. Thesis. pp. 0–0. Citeseer (2009)
24. Rivest, R.L., Schuldt, J.C.: Spritz-a spongy rc4-like stream cipher and hash function (2014)

25. Rogaway, P., Shrimpton, T.: Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: Fast Software Encryption. pp. 371–388. Springer (2004)
26. Shamir, A.: Squash–a new mac with provable security properties for highly constrained devices such as rfid tags. In: Fast Software Encryption. pp. 144–157. Springer (2008)
27. Standaert, F.X., Piret, G., Gershenfeld, N., Quisquater, J.J.: Sea: A scalable encryption algorithm for small embedded applications. In: Smart Card Research and Advanced Applications, pp. 222–236. Springer (2006)