IET Information Security

Special Issue: Lightweight and Energy-Efficient Security Solutions for Mobile Computing Devices

Hash-One: a lightweight cryptographic hash function

ISSN 1751-8709
Received on 14th September 2015
Revised 22nd March 2016
Accepted on 24th March 2016
E-First on 11th May 2016
doi: 10.1049/iet-ifs.2015.0385
www.ietdl.org

Puliparambil Megha Mukundan[1] ✉, Sindhu Manayankath[1], Chungath Srinivasan[1], Madathil Sethumadhavan[1]

[1]TIFAC-CORE in Cyber Security, Amrita School of Engineering, Coimbatore, Amrita Vishwa Vidyapeetham, Amrita University, India
✉ *E-mail: pmegha.mukundan@gmail.com*

Abstract: The increased demand for lightweight applications has triggered the need for appropriate security mechanisms in them. Lightweight cryptographic hash functions are among the major responses toward such a requirement. The authors thus have a handful of such hash functions such as QUARK, PHOTON, SPONGENT and GLUON introduced already. The cryptanalysis of these hash functions is crucial in analysing their strength and often calls for improvement in designs. Their performance, are also to be taken care of, in terms of both software and hardware implementations. Here, they propose a lightweight hash function with reduced complexity in terms of hardware implementation, capable of achieving standard security. It uses sponge construction with permutation function involving the update of two non-linear feedback shift registers. Thus, in terms of sponge capacity it provides at least 80 bit security against generic attacks which is acceptable currently.

1 Introduction

The improvements in the field of cryptography were mandatory to enhance the strength in primitive cryptographic techniques such as hash functions, symmetric and asymmetric ciphers. As technology took the diversion to compact designs, so has much expectations from its security perspectives. Thus rightly we have the field of 'lightweight cryptographic hash functions' for various applications. A designer of lightweight cryptography has to manage with the trade-off between security, costs and performance. Usually, any two of the three design goals – security and low costs, security and performance, low costs and performance can be optimised effortlessly, whereas it is very difficult to optimise all the three design goals at the same time. For example, a secure and high-performance hardware implementation can be achieved by a pipelined architecture which also can be integrated with many countermeasures against physical cryptanalysis. The resulting design would have a high area requirement, which results in high costs. On the other hand, it is possible to design a secure and low-cost hardware implementation with the drawback of limited performance. A security level of 128 bit is typical for high-performance applications, 80 bit security is often reasonable for lightweight applications. There are a number of lightweight hash functions designed for different applications and requirements with varying security levels. Some of them are QUARK [1], PHOTON [2], SPONGENT [3] and GLUON [4]. Lightweight versions of the new SHA-3 standard KECCAK [5] have also been introduced. We have analysed them and designed a lightweight hash function with improved performance and achieving 80 bit security. The initial motivation is derived from the design of QUARK family of hash functions.

This paper consists of mainly six sections apart from introduction and related works. In Section 2, we discuss the design methodology of the proposed system and the construction scheme is given in Section 3. Section 4 describes the design motivations of the proposed hash function. The results of security analysis are given in Section 5. Hardware implementation estimates are conveyed in Section 6 and we conclude this paper with Section 7.

1.1 Related works

In this section, we mainly discuss about four lightweight hash functions QUARK, SPONGENT, PHOTON and GLUON. QUARK is a lightweight hash function that incorporates design methodologies from two existing works, Grain [6] which is a stream cipher and the block cipher KATAN [7]. It uses an iterative hash construction method known as the sponge construction which leads to the advantage of obtaining variable length output. The designers of QUARK have proposed three instances known as U-QUARK, S-QUARK and D-QUARK with U-QUARK being the lightest of all the three. QUARK uses two non-linear feedback shift registers (NFSRs) and a linear FSR (LFSR) which are updated using three different non-linear functions. The internal state is clocked $4b$ times, where b is the width of the permutation. The state update for registers is based on a linear Boolean function for LFSR, and three non-linear Boolean functions for NFSRs. Apart from this, a non-linear Boolean function is introduced to influence the changes in the NFSRs.

Since QUARK uses sponge construction, the security parameter for QUARK involves the capacity c. Thus, it offers 2^c preimage resistance and $2^{c/2}$ collision and second preimage resistance.

Another lightweight hash function which uses sponge construction is SPONGENT. It is a family of lightweight hash functions based on a wide PRESENT-type permutation [8]. It accepts an arbitrary length input and produces fixed length output. The sponge structure consists of a state of b bits known as the width of the permutation π_b. The permutation $\pi_b: \mathbb{F}_2^b \rightarrow \mathbb{F}_2^b$ can be considered as a round transformation applied on the bits of the internal sponge state. The different variants are referred to as SPONGENT-$n/c/r$ for different hash size n, capacity c and rate r. Out of the 13 proposed variants, already implemented ones are SPONGENT-160/160/80 and SPONGENT-256/256/128. Since SPONGENT uses PRESENT-type permutation, its security advantages are enjoyed by the hash function to a great extent. In case of collision attacks, it is found that the probability for differential paths is considerably low and an attack is not possible in full number of rounds.

PHOTON is a sponge-based, lightweight and hardware-oriented hash function. The internal state is represented as a matrix with 4 or 8 bit entries, depending on which the five PHOTON flavours have been selected. It uses advanced encryption standard (AES) like fixed key permutation. There are about 12 rounds which include AddConstants, SubCells, ShiftRows and MixColumnSerial similar to AES. The GLUON family of lightweight hash functions are inspired by two stream ciphers F-FCSR-v3 and X-FCSR-v2.

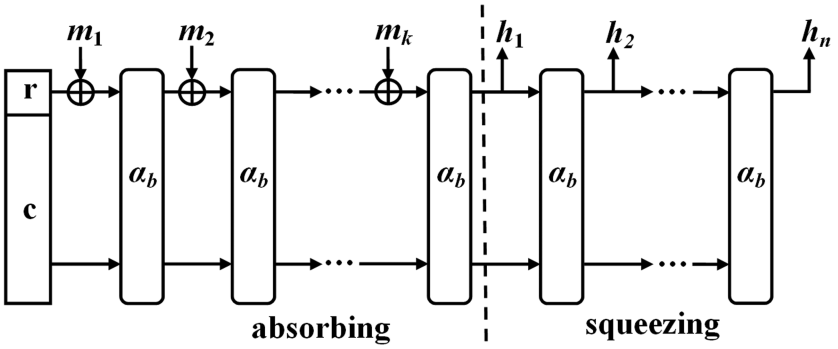

Fig. 1 *Sponge construction*

The design involves a word ring FCSR composed of a main shift register and a carry register. The construction is purely based on sponge criterion which involves a filtered FCSR inspired from the two stream ciphers.

We aimed at constructing a lightweight hash function with minimum building blocks (for permutation) achieving standard security. In terms of gate equivalent (GE), the construction of SPONGENT and PHOTON has attracted us. However, in order to reduce the components in permutation, we opted for a construction involving shift registers. QUARK has thus became our main source of inspiration. We propose Hash-One, a lightweight hash function with reduced GE achieving the same level of security compared with QUARK.

In Section 6, we have compared Hash-One with QUARK, SPONGENT, PHOTON and GLUON in terms of security and hardware performance.

2 Design methodology

As sponge construction is one of the efficient ways of constructing hash functions, we opt for sponge-based construction in our lightweight hash design. We use sponge construction producing a message digest of length 160 bits which is one bit less than the size of the sponge state. The rate r of the sponge is chosen as the minimum possible value which is one bit. This ensures proper mixing of the message bits with that of the state bits. As the rate is decreased, the speed of the hash function is won back by reducing the number of iterations to a lower level. This ensures minimum complexity in terms of time and computation.

Two NFSRs are used to update the sponge state. We use Goli ̀c's design criterion [9] to choose the tap positions from shift registers to the filter functions. According to Goli ̀c, in order to avoid inversion attacks in shift registers, the tap positions of the register should form a full positive difference set also known as a Golomb ruler. We have selected a Golomb ruler of order 8 and length 161 for this design.

3 Scheme of Hash-One

This section explores the details of sponge construction, initialisation of shift registers and permutation function for updating the sponge state. Toward the end, the pseudocode for Hash-One is also provided for better understanding.

3.1 Sponge-based construction

A sponge is an iterated cryptographic model developed by Bertoni *et al.* and is shown in Fig. 1 [10]. This iterated construction consists of a state S of b bits, which operates on r bits of message block and provides n bits of output. The state size of sponge is represented as $b = r + c \geq n$, which is also known as the width of the permutation. A permutation α_b is applied on the b bits of the sponge state. The parameter r is the rate or size of message block and c is the capacity.

Three different phases of sponge include:

- *Initialisation phase*: Padding bits are included with the message bits so as to make the whole length a multiple of rate r. This makes the message bits suitable to be cut into r-bit blocks for processing.
- *Absorbing phase*: The r-bit blocks from the message are XORed with the first or last r-bits of the state bits. The state bits are then fed into the permutation function α_b, and repeated until all message blocks $m_1, m_2, \ldots, m_k$ are consumed. The first or last r-bits of the state bits from the final sponge state in this phase forms the first r-bits of the hash.
- *Squeezing phase*: Permutation α_b is applied on the b bits of sponge state, with first r bits returned as output after each permutation. This is repeated until the required output length of n bits are squeezed out. Thus, during each permutation we get $h_1, h_2, \ldots, h_n$ as the output. The hash is the concatenation of $h_1, h_2, \ldots, h_n$, i.e $H = h_1 \| h_2 \| \ldots \| h_n$.

The sponge construction is considered to have a hermetic strategy [10] because it is known to be resistant against generic attacks. It offers 2^c preimage resistance and $2^{c/2}$ collision and second preimage resistance.

In Hash-One, the sponge state is of size $b = 161$ bits, initialised with the first 161 bits of the binary equivalent of the mathematical constant Pi (π). The rate $r = 1$ bit, size of hash is $n = 160$ bits and capacity $c = 160$ bits providing a security level of 2^{80} against collision and second preimage attacks.

3.2 Transitional permutation

The sponge state S can be represented as $S \leftarrow (S_0, S_1, \ldots, S_{160})$. Hash-One uses two NFSRs P and Q of sizes $b_1 = 80$ and $b_2 = 81$, respectively, to represent the internal state of the sponge construction. The NFSR P takes the first 80 bits from sponge state and NFSR Q takes the remaining 81 bits. The contents of two registers can be represented as follows:

- P:$(P_0, P_1, \ldots, P_{79}) \leftarrow (S_0, S_1, \ldots, S_{79})$
- Q:$(Q_0, Q_1, \ldots, Q_{80}) \leftarrow (S_{80}, S_{81}, \ldots, S_{160})$

In a single round, the permutation α_b updates the sponge state S as follows: $\alpha_b(S) = \alpha_b(P_0, P_1, \ldots, P_{79}, Q_0, \ldots, Q_{80}) \rightarrow (P_1, P_2, \ldots, P_{79}, T_1, Q_1, Q_2, \ldots, Q_{80}, T_2)$ where $T_1 = P_f(P_0, P_{11}, Q_{23}, P_{55}) \oplus L_f(P_1, Q_1, P_{50})$ and $T_2 = Q_f(Q_0, Q_{25}, Q_{41}, P_{48}) \oplus L_f(P_1, Q_1, P_{50})$ detailed in Section 3.3. In the absorbing phase, each bit of the message is XORed with the last bit S_{160} of the sponge state. After XORing each message bit, the 161 bits of the sponge state are fed into the permutation α_b and this process is repeated until all message bits are absorbed. During the absorption of the first and last message bits, α_b involves 324 (*i.e.* $4b_2$) rounds of state update. While absorbing the intermediate message bits, a total of 162 (*i.e.* $2b_2$) rounds of state update are included. As opposed to this, the squeezing phase necessitates a single round of state update for each permutation α_b. Here, the permutation is repeated until the required output length of $n = 160$ bits are squeezed out.

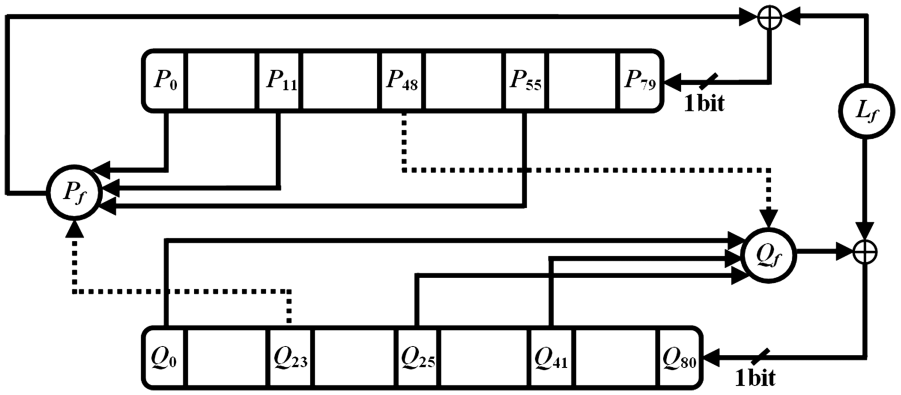

Fig. 2 *Permutation in Hash-One*

3.3 Update function

Two non-linear Boolean functions in four variables and a linear function in three variables are used to update the state registers in each clock cycle. The non-linear functions P_f and Q_f updates the registers P and Q, respectively. The linear function L_f is used to update both registers P and Q. The bits in registers are shifted toward the left with one bit at a time and updating the rightmost bits P_{79} and Q_{80} with the help of the non-linear functions. The clocking and the register update of the proposed hash function is shown in Fig. 2. To ensure proper diffusion of bits in both registers, we choose three tap positions from a selected register and one from the supporting register.

The rightmost bits P_{79} and Q_{80} are updated by the functions P_f and Q_f, respectively, and by the linear function L_f commonly as follows:

$$P_f(P_0, P_{11}, Q_{23}, P_{55}) = (P_0P_{11} \oplus P_0P_{55} \oplus P_{11}Q_{23} \oplus Q_{23}P_{55} \oplus Q_{23} \oplus P_{55}) \oplus 1$$

$$Q_f(Q_0, Q_{25}, Q_{41}, P_{48}) = (Q_{25}P_{48} \oplus Q_{25}Q_{41} \oplus Q_0P_{48} \oplus Q_0Q_{41} \oplus Q_{25} \oplus Q_{41})$$

$$L_f(P_1, Q_1, P_{50}) = (P_1 \oplus Q_1 \oplus P_{50})$$

We use the symbol $\oplus$ to indicate exclusive-OR or XOR throughout in this paper. The choice of binary '1' in the Boolean function P_f helps the registers not to go into all zeros state.

Since the 161-bit internal state of the permutation function is updated in a non-linear way, finding its periodicity is difficult. During the absorption phase, the permutation is clocked 324 times in a non-linear way making it to behave as a random function. Hence, the periodicity of the permutation for a given input is considered to be equiprobable. Assuming a maximum periodicity of 2^{161} for the permutation function, the probability for a given input to cause a periodicity smaller than 2^{100} is 2^{-61}, which is negligibly small.

The pseudocode of Hash-One is given as follows:

Pseudocode

The sponge state S is initialised with the first 161 bits of the binary equivalent of the decimal digits of the mathematical constant Pi (π). Let $m[0], m[1], \ldots, m[k-1]$ be a k bit message with iteration count, l set to 0.

i. During the absorption phase of sponge do the following:

   (a) if $l = 0$, then set $S[160] \leftarrow S[160] \oplus m[l]$ and update S for 324 times. Increment l by one.
   (b) for $l = 1$ to $k - 2$ do the following:

   (i) Set $S[160] \leftarrow S[160] \oplus m[l]$.
   (ii) Update S for 162 times.
   (c) if $l = k - 1$, then do the following:

   (i) Set $S[160] \leftarrow S[160] \oplus m[l]$.
   (ii) Update S for 324 times.

ii. During the squeezing phase do the following:

   (a) $h[1] \leftarrow S[160]$
   (b) for $i = 2$ to 160

   update S once
   $h[i] \leftarrow S[160]$

iii. Output the hash value $H \leftarrow h[1] \| h[2] \| \ldots \| h[160]$.

Note that in this paper the symbol $\|$ represents the concatenation of bits.

4 Design motivations

In this section, we justify the use of different components in the proposed design. This includes our choice of sponge construction, two NFSRs, two 4-variable Boolean functions, a linear function in three variables, number of rounds for permutation and the minimal rate used in sponge construction.

According to Bertoni *et al.* [11], the sponge construction is claimed to be the only alternative to classical Merkle–Damgård (MD) construction involving a compression function. Majority of the other existing construction methods are modified versions of MD, with varying internal states, prefix-free encoding etc. Owing to the iterative nature of sponge construction, a single permutation is all it takes to process the multiple messages without the need for storing any of the message bits. Sponge construction being hermetic (no chances of structural distinguishers for the underlying permutation) and flexible (capable of producing variable length outputs) makes it an ideal choice.

The aim of this hash function design is to keep things simple yet secure. Keeping this in mind we first concentrated on using minimal number of shift registers. Shift registers were used instead of S-boxes to reduce the complexity of circuits in the permutation. To achieve the property of non-linearity, two non-linear Boolean functions P_f and Q_f in four variables were selected. The Boolean functions were formulated by considering their balancedness with non-linearity 4, correlation immunity 1 and algebraic immunity 2. To ensure the permutation to be random we add P_1, P_{50}, Q_1 linearly to both P_f and Q_f.

The rate of absorbing and squeezing in sponge are reduced to one bit at a time to ensure proper mixing of message bits with the state bits. During the absorption phase, after absorbing the first and last message bits, the sponge state is updated 324 times so as to obtain the avalanche effect. While during the absorption of the intermediate message bits, the sponge state updates 162 times. We can even let all the absorbed message bits undergo 324 times of update, which ensures thorough mixing of the entire message bits with that of state bits. However, this forfeits our aim of achieving lower computing power and time consumption. To generate 160-bit hash, the sponge state is updated only once in every permutation followed by squeezing a single bit after each permutation in the squeezing phase.

5 Security analysis

As part of the security analysis of Hash-One, the scope of differential attack and algebraic attack in Hash-One are analysed. We have also applied three different cryptographic randomness tests: namely, collision test, coverage test and strict avalanche criterion (SAC) test to examine the randomness of Hash-One. The test results for different cryptographic randomness tests are included in this section.

5.1 Cryptographic randomness testing

To evaluate the randomness of outputs of block ciphers and hash functions, a new test package was proposed in [12]. The following different cryptographic properties are evaluated in this randomness testing:

- *SAC:* A single input change should change every output bit with probability $p = (1/2)$.
- *Collision:* Two different inputs having the same output results in collision.
- *Coverage:* Hash functions are required to behave such as random mappings.

5.1.1 SAC test: The SAC test was originally introduced for S-boxes. According to this test when a single bit in the input is changed, then every output bit must change with probability (1/2). This test can also be used to evaluate the effect on the outputs for a single bit flip in the inputs of hash function.

In SAC test, a ***SAC*** matrix of size $x \times y$ (x is the block size of the message and y is the chaining variable size) is to be created as follows: the ***SAC*** matrix entries are initialised to 0. Out of 2^{20} random inputs to be tested, a random input is taken and the corresponding output is calculated. The rth bit (such that $0 \le r \le n - 1$, where n is the number of hash bits) of the same input is flipped, output is calculated and the two outputs are XORed. For each non-zero bit c of the output, (r, c)th entry of the ***SAC*** matrix is incremented by 1. The process is repeated for each input bit r. We analyse if the distribution of the values in the ***SAC*** matrix follow a binomial distribution. For this we use χ^2 goodness of fit test to compare the expected results with observed results. The parameters chosen by us for this SAC test are given below:

Table 1 SAC test

Range	Expected count	Observed count
0–523,857	5126	5196
523,858–524,158	5118	5053
524,159–524,417	5112	5087
524,418–524,718	5118	5058
524,719–1,048,576	5126	5206

Table 2 Collision test

Range	Expected count	Observed count
0–116	13,517	13,519
117–122	12,714	12,651
123–128	14,407	14,552
129–134	12,056	11,873
135–4096	12,842	12,941

Table 3 Coverage test

Range	Expected count	Observed count
0–2572	13,053	13,003
2573–2584	13,414	13,315
2585–2594	12,967	13,075
2595–2606	13,319	13,359
2607–4096	12,783	12,784

- *Size of **SAC** matrix:* 160×160.
- *Degrees of freedom:* 4.
- *Significance level:* 0.01.
- Number of inputs tested: 2^{20}.

Table 1 shows the values obtained for the SAC test. For the 4 degrees of freedom, we got χ^2 value as 3.85 and it corresponds to the p-value as 0.42. Since this p-value is much greater than the significance level 0.01, it passes the test.

5.1.2 Collision test: The property of collision resistance in hash function ensures that it is computationally infeasible to find any two inputs that output the same message digest. As the property of collision resistance is crucial in cryptographic hash functions, we use collision test to make sure the same. The number of collisions in specific number of bits in the output is considered for evaluation. Thereby, we are testing the near collision property in Hash-One.

According to collision test a random input is chosen and its first 12 bits are given all the possible values to generate an input set of size 2^{12}. The corresponding hash outputs are calculated and are stored in a hash table. To find the collisions, only the first 16 bits of the hash are considered. Thus, number of collisions in the hash table for each of the 2^{16} random inputs are computed. Let r_1, r_2, …, $r_2 16$ denote the random inputs. For each random input r_x ($1 \le x \le 2^{16}$), we find 2^{12} number of inputs i_1, i_2, …, $i_2 12$ and their corresponding outputs j_1, j_2, …, $j_2 12$ which are stored in a hash table (with only the first 16 bits of each hash). The χ^2 goodness of fit test is utilised to reveal the randomness. The parameters for this collision test are given below:

- *Size of message:* 20 bits.
- *Number of inputs tested:* $2^{16} \times 2^{12}$.
- Significance level: 0.01.
- Degrees of freedom: 4.

In Table 2, we have shown the details of collision test performed on the proposed system. For the 4 degrees of freedom, we got the χ^2 value as 4.73 and it corresponds to the p-value as 0.31. Since the p-value is greater than the significance level, we conclude that the proposed hash function is random according to the collision test specified in [12].

5.1.3 Coverage test: The coverage of an output set for a given input set is evaluated in the coverage test. Coverage of an output set is expected to be around 63% of its input set, provided there exists a random mapping. The input set size and coverage are equal for a random permutation, when all the bits in the output are considered.

As in the collision test described above, in coverage test too we use 2^{16} random inputs. For a given random input r_x ($1 \le x \le 2^{16}$), 2^{12} possible input combinations are formed by changing its first 12 bits. The corresponding hashes are calculated and the hash table is filled with only the first 12 bits of each hash value. Coverage is calculated for each random input from the hash table. We applied the χ^2 goodness of fit test to find out the p-values and understand the randomness. The coverage test parameters are given below:

- *Size of message:* 20 bits.
- *Number of inputs tested:* $2^{16} \times 2^{12}$.
- Significance level: 0.01.
- Degrees of freedom: 4.

The test results are shown in Table 3. The p-value and χ^2 value are 0.74 and 1.93, respectively. Here too, we have the p-value greater than significance level, supporting the fair conclusion that the proposed hash function is random according to coverage test.

5.2 Differential attack

Differential cryptanalysis is one of the significant cryptanalytic attacks developed by Biham and Shamir [13]. It is a type of chosen plaintext attack where the cryptanalyst searches for high probability of certain occurrences of plaintext differences and the corresponding differences in the ciphertext. These kinds of attacks are primarily applicable in block ciphers [13], stream ciphers [14] and also in some hash functions [15].

In a perfectly randomised cipher, the probability of an output difference δY, given an input difference δX is $(1/2)^n$, where n is the number of hash bits and $(\delta X,\ \delta Y)$ is a differential pair. In Hash-One, a change in the 161th (*i.e.* Q_{80}) bit of sponge state enters the register of P in round 58, that is when the change reaches S_{103} (*i.e.* Q_{23}). When the change in Q reaches bit position S_{81} (*i.e.* Q_1), the P register is again affected. Hence for a change made in position Q_{80}, the differences are propagated into the whole 161-bit internal state in 261 rounds which is much <324 rounds of permutation used to absorb the first and last bit of the messages in the absorption phase. We have also made changes individually to the registers Q_{40}, Q_{26}, Q_{24}, Q_2, P_{56}, P_{51}, P_{49}, P_{12}, P_1 and tracked the number of differences that propagated into the 161-bit internal state of the permutation for reduced number of rounds. Table 4 below gives the results of an automated search of the differences for reduced rounds. Hence we observed that for a change in any state bit, the differences are propagated into the whole 161-bit internal state in <262 rounds of the permutation, which shows that this type of attack may not be feasible in Hash-One.

5.3 Algebraic attack

An algebraic attack is one of the recent techniques in cryptography that mainly deals with obtaining and breaking a system of polynomial equations. It can be considered as the process of converting a problem of retrieving secret key into solving a system of non-linear equations. As this is an NP-complete problem, it is not simple as it seems to be. Linear components in a hash construction are the main targets of algebraic attack. The system of equations are solved usually by linearisation.

The size of the internal sponge state of Hash-One is 161 bits and we consider it as 161 unknown variables. For each clock cycle of the hash function, the internal state is updated which results in two new equations and one new variable of degree at most 2. After 40 clock cycles, m (=80) equations with maximum algebraic degree d (=2) and n (=201) variables will be generated. The XL algorithm [16] consists of multiplying these initial m equations by all possible monomials of degree up to $D-d$, so that the total degree of resulting equations is D and let R be the number of equations and T be the number of all monomials of this newly generated system of equations. Here, R is evaluated as $R=m(n^{D-d}/D-d)$ and T is evaluated as $T=(n^D/D!)$ where $D \geq (n/\sqrt{m})$. If most of the equations are linearly independent, then XL algorithm will succeed as long as $R \geq T$. Thus we need to take $D \geq 23$, which will result in very large values of R and T ($>2^{101}$). Since the time complexity for solving the system of equations is $\mathcal{O}(T^3)$, this algebraic attack is not feasible on the proposed hash function.

5.4 Cube attack

Cube attack [17] is successful against any symmetric cipher where at least one of its outputs has a very low algebraic degree. For a cube attack, we need only a black-box access to the cipher, whereby we can assign various values to the publicly known input bits and obtain the output bits. Since cube attacks are key recovery attacks, they are only effective on cryptographic algorithms such as symmetric ciphers and message authentication codes which have keys as one of its inputs. This type of attack is not directly applicable to hash functions, as the only secret input to it is the message, which is processed till the absorption phase is complete. In the case of hash functions, we can use cube testers [18] to find distinguisher for the permutations used. Here, one can apply cube testers by choosing the initial state of the hash function uniformly at random, which may help us to find some distinguisher. As given in Section 5.1.2, the output of the Hash-One is tested for randomness by choosing random initial states, which is also a similar attempt to find a distinguisher as in the case of cube testers. This shows that Hash-One is unlikely to be broken by cube testers as we could not find any distinguishers.

6 Hardware estimation

This section describes the hardware implementation estimates of the proposed system. Since we use modest Boolean functions in our hash function, we have included the circuits for the same in here. The results in this section are the outcome of simulations performed on available synthesising tools.

6.1 Logic circuits

Our design when implemented in hardware used NAND only circuits for the Boolean functions Q_{f}, P_{f} and L_{f}

$$P_{\mathrm{f}}(P_0, P_{11}, Q_{23}, P_{55}) = (P_0P_{11} \oplus P_0P_{55} \oplus P_{11}Q_{23} \oplus Q_{23}P_{55} \oplus Q_{23} \oplus P_{55}) \oplus 1$$

$$Q_{\mathrm{f}}(Q_0, Q_{25}, Q_{41}, P_{48}) = (Q_{25}P_{48} \oplus Q_{25}Q_{41} \oplus Q_0P_{48} \oplus Q_0Q_{41} \oplus Q_{25} \oplus Q_{41})$$

$$L_{\mathrm{f}}(P_1, Q_1, P_{50}) = (P_1 \oplus Q_1 \oplus P_{50})$$

To reduce the area occupied by these functions, we need to reduce the NAND gates in the representation of the functions. We show the optimised representation of Q_{f} and P_{f}, which gives the minimum number of NAND gates. Figs. 3 and 4 give the NAND only circuits for P_{f} and Q_{f}, respectively. The Boolean functions P_{f} and Q_{f} can be re-written as

$$P_{\mathrm{f}}(P_0, P_{11}, Q_{23}, P_{55}) = (P_0 \oplus Q_{23})(P_{11} \oplus P_{55}) \oplus Q_{23} \oplus P_{55} \oplus 1$$

$$Q_{\mathrm{f}}(Q_0, Q_{25}, Q_{41}, P_{48}) = (P_{48} \oplus Q_{41})(Q_0 \oplus Q_{25}) \oplus Q_{25} \oplus Q_{41}$$

The output bits of these functions are used for updating the memory components P and Q, respectively. During the message absorption phase, this update is done 324 times when the first and the last message bits are induced into the memory S and just 162 times for processing all the intermediate message bits.

Table 4 Differential propagation

Rounds	Number of differences propagated									
	Q_{80}	Q_{40}	Q_{26}	Q_{24}	Q_2	P_{56}	P_{51}	P_{49}	P_{12}	P_1
100	12	37	53	50	46	59	54	34	67	65
132	31	78	102	100	91	102	98	77	112	113
152	57	111	130	129	119	125	124	109	134	136
173	87	139	148	147	139	143	142	135	151	151
216	146	160	161	161	161	160	161	161	161	161
261	161	161	161	161	161	161	161	161	161	161

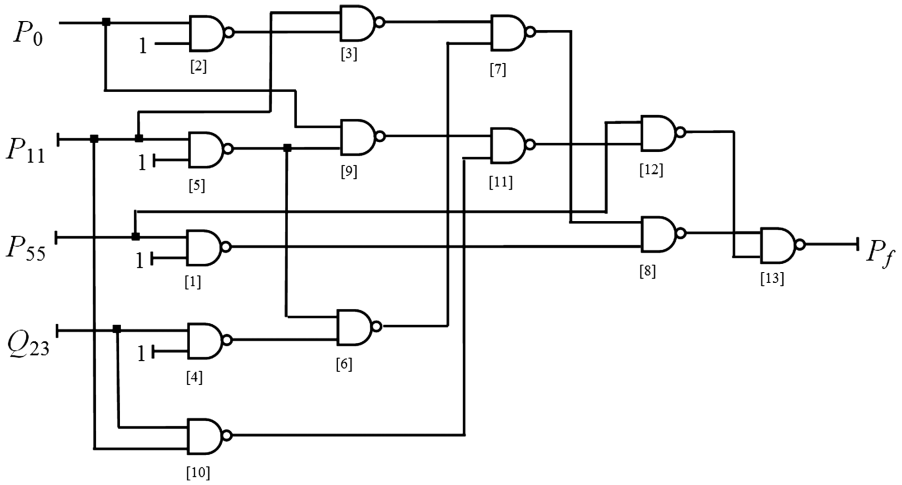

Fig. 3 *NAND only circuit for function P_f*

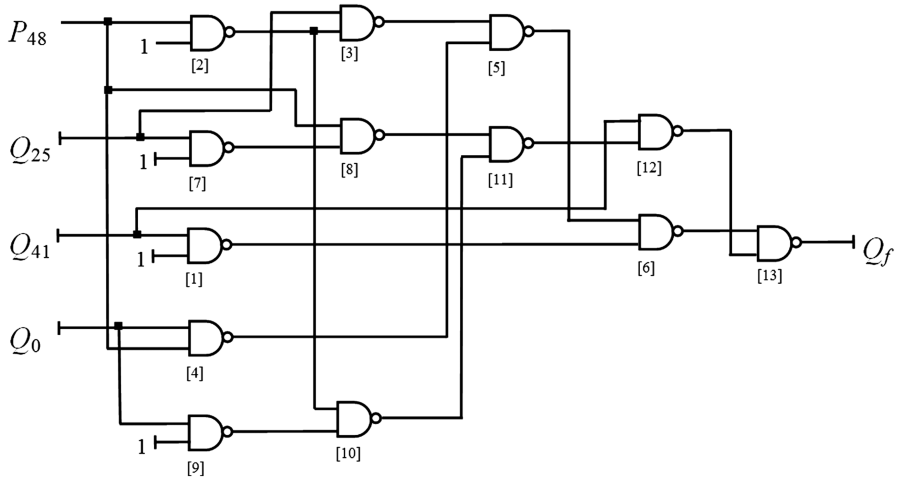

Fig. 4 *NAND only circuit for function Q_f*

An XOR gate is equivalent to four NAND gates and an AND gate is equivalent to two NAND gates for implementing in hardware. Hence, a minimum of 13 NAND gates each for P_f and Q_f are required when implemented in hardware. Linear function L_f can be realised with eight NAND gates.

6.2 Hardware simulation

We explain the theoretic basis of the hardware implementation of Hash-One considering the UMC 180 nm technology. After completing the optimisation process, we obtain the area consumed by the three Boolean functions to be 34 GE. Considering the memory requirements, flip-flops can be used for the storage of state bits. In general, for the implementation of an n-bit LFSR, a generic circuit with n flip-flops are required [19]. Two different methods of hardware implementation: namely, serial and parallel ways are described here.

6.2.1 Serial implementation estimates: By considering the two registers P and Q as a single 161-bit register, we prefer using a single multiplexer along with 161 flip-flops for the update process. The area requirements for a flip-flop is 6 GE and that of a multiplexer is 2.33 GE as in [20]. Thus essentially, Hash-One requires $(161 \times 6) + 2.33 + 38 = 1006.33$ GE (including the XOR of message bits) for serial implementation. Table 5 gives a fair comparison of the GE values of the existing lightweight hash functions with Hash-One.

6.2.2 Parallel implementation estimates: As the shift register P takes 24 cycles to bring a change in bit P_{55}, it allows to induce parallel computing in our implementation. Thus in order to achieve 324 rounds of state update in the absorption phase, we need to perform 14 rounds under the cost of extra hardware. This is due to the use of 24 self-replicated logic circuits which are simultaneously updated 14 times to get the same effect of 324 rounds. The outputs

Table 5 Hardware performance of Hash-One in comparison with some of the existing lightweight hash functions

Hash function	n	c	r	Preimage	Collision	Second preimage	Process, μm	Area (GE)	Cycles
Hash-One	160	160	1	160	80	80	0.18	1006	324/162
Hash-One	160	160	1	160	80	80	0.18	2130	14/7
SPONGENT [3]	176	160	16	144	80	80	0.13	1329	3960
SPONGENT [3]	176	160	16	144	80	80	0.13	2190	90
D-QUARK [1]	176	160	16	160	80	80	0.18	1702	704
D-QUARK [1]	176	160	16	160	80	80	0.18	2819	88
PHOTON [2]	160	160	36	124	80	80	0.18	1396	1332
PHOTON [2]	160	160	36	124	80	80	0.18	2117	180
GLUON [4]	160	160	16	160	80	80	0.18	2799	50

from 24 circuits are stored temporarily and wired directly to the last 24 bits of the 161-bit state register. The remaining 137 bits are also updated by means of direct wiring without the need for any extra GE. This implementation uses scan registers (6.67 GE in our library), instead of a flip-flop+multiplexer combination, corresponding to every bit. The area requirements for parallelised architecture is obtained as $(161 \times 6.67) + (24 \times 38) + (24 \times 6) = 2129.87$ GE (including the memory for storing the 24 bits). During the absorption of the intermediate message bits, we use 162 rounds of the permutation. Considering the parallel computation, we need to perform it only for seven rounds and it does not make any change (reduction) to the gate equivalents.

On the basis of the hardware performance of Hash-One, we have compared it with some of the existing lightweight hash functions and is shown in Table 5. The GE differences based on serial and parallel implementations are included in this table. For Hash-One, the cycles are represented as 324 for first and last message bits while 162 for all the intermediate message bits in the serial implementation. In parallel implementation, as we use 24 self-replicated circuits, 324 and 162 cycles are reduced to 14 and 7 cycles, respectively.

Hash-One shares some similarities with QUARK mainly in use of shift registers and sponge construction. To update the sponge state, QUARK uses three shift registers (two NFSRs and one LFSR), whereas Hash-One uses two NFSRs. The number of rounds in QUARK is four times the size of internal state while it is approximately twice the internal state size in Hash-One. By reducing the components in hash construction and the number of variables (tap positions in shift registers) in Boolean function, Hash-One can be implemented in hardware with 1006 GE while it is 1329 GE in SPONGENT, 1396 GE in PHOTON and 1702 GE in D-QUARK, all of which provides a security level of 80 bits.

7 Conclusion

We have proposed a lightweight hash function to meet the requirements in many of the constrained devices such as radio-frequency identification tags, wireless sensors and other embedded system devices. The security level has been set to 80-bit standard level, with minimal computational complexity and area requirements. Hash-One has achieved 160-bit preimage resistance and 80-bit collision resistance consuming 1006 GE which is an estimate of the hardware implementation showing much less area requirements. The randomness in Hash-One has been successfully verified using collision test, coverage test and SAC test. Thus in terms of compactness, we actually moved from highly constrained to ultra-constrained environment.

8 Acknowledgments

We thank N.R. Rajesh Pillai, Defence Research and Development Organisation, India and other referees whose comments induced us to clarify the exposition in this paper.

9 References

[1] Aumasson, J.P., Henzen, L., Meier, W., *et al.*: 'QUARK: a lightweight hash', *J. Cryptol.*, 2013, **26**, pp. 313–339

[2] Guo, J., Peyrin, T., Poschmann, A.: 'The PHOTON family of lightweight hash functions'. In Rogaway, P. (Ed.): '*Advances in Cryptology - CRYPTO 2011*' (Springer, Berlin Heidelberg 2011), pp. 222–239

[3] Bogdanov, A., Knežević, M., Leander, G., *et al.*: 'SPONGENT: A lightweight hash function'. In Preneel, B., Takagi, T. (Eds.): '*Cryptographic Hardware and Embedded Systems – CHES 2011*' (Springer, Berlin Heidelberg, 2011), pp. 312–325

[4] Berger, T.P., D'Hayer, J., Marquet, K., *et al.*: 'The GLUON family: a lightweight hash function family based on FCSRs'. In Mitrokotsa, A., Vaudenay, S. (Eds.): '*Progress in Cryptology – AFRICACRYPT 2012*' (Springer, Berlin Heidelberg, 2012), pp. 306–323

[5] Kavun, E.B., Yalcin, T.: 'A lightweight implementation of KECCAK hash function for radio-frequency identification applications'. In Yalcin, S.B.O. (Eds.): '*Radio Frequency Identification: Security and Privacy Issues*' (Springer, Berlin Heidelberg, 2010), pp. 258–269

[6] Hell, M., Johansson, T., Meier, W.: 'Grain: a stream cipher for constrained environments', *Int. J. Wirel. Mob. Comput.*, 2007, **2**, pp. 86–93

[7] De Cannière, C., Dunkelman, O., Knežević, M.: 'KATAN and KTANTAN – A family of small and efficient hardware-oriented block ciphers'. In Clavier, C., Gaj, K. (Eds.): '*Cryptographic Hardware and Embedded Systems – CHES 2009*' (Springer, Berlin Heidelberg, 2009), pp. 272–288

[8] Bogdanov, A., Knudsen, L.R., Leander, G., *et al.*: '*PRESENT: an ultra-lightweight block cipher*'. (Springer, 2007)

[9] Golić, J.D.: 'On the security of nonlinear filter generators'. In Gollmann, D. (Ed.): '*Fast software encryption*' (Springer, Berlin Heidelberg, 1996), pp. 173–188

[10] Bertoni, G.: The sponge functions corner. Available at http://www.sponge.noekeon.org/

[11] Bertoni, G., Daemen, J., Peeters, M., *et al.*: 'On the indifferentiability of the sponge construction'. In Smart, N. (Ed): '*Advances in Cryptology - EUROCRYPT 2008*', (Springer, Berlin Heidelberg, 2008), pp. 181–197

[12] Doganaksoy, A., Ege, B., Koçak, O., *et al.*: 'Cryptographic randomness testing of block ciphers and hash functions'. IACR Cryptology ePrint Archive 2010, 2010, vol. **564**

[13] Biham, E., Shamir, A.: 'Differential cryptanalysis of DES-like cryptosystems', *J. Cryptol.*, 1991, **4**, pp. 3–72

[14] Wu, H., Preneel, B.: 'Chosen IV attack on stream cipher WG'. ECRYPT stream cipher project report, Citeseer, 2005, vol. **45**

[15] Wang, X., Yu, H.: 'How to break MD5 and other hash functions'. In Cramer, R. (Eds): '*Advances in Cryptology - EUROCRYPT 2005*' (Springer, Berlin Heidelberg, 2005), pp. 19–35

[16] Courtois, N., Klimov, A., Patarin, J., *et al.*: 'Efficient algorithms for solving overdefined systems of multivariate polynomial equations'. In Preneel, B. (Ed.): '*Advances in Cryptology – EURO-CRYPT 2000*' (Springer, Berlin Heidelberg, 2000), pp. 392–407

[17] Dinur, I., Shamir, A.: 'Cube attacks on tweakable black box polynomials'. In Joux, A. (Ed.): '*Advances in cryptology – EUROCRYPT 2009*' (Springer, Berlin Heidelberg, 2009), pp. 278–299

[18] Aumasson, J.P., Dinur, I., Meier, W., *et al.*: 'Cube testers and key recovery attacks on reduced-round md6 and trivium'. In Dunkelman, O. (Ed.): '*Fast Software Encryption*' (Springer, Berlin Heidelberg, 2009), pp. 1–22

[19] Aloisi, W., Mita, R.: 'Gated-clock design of linear-feedback shift registers'. IEEE Transactions on Circuits and Systems II: Express Briefs, 2008, vol. **55**, pp. 546–550

[20] Khoo, K., Peyrin, T., Poschmann, A.Y., *et al.*: 'FOAM: Searching for hardware-optimal SPN structures and components with a fair comparison'. In Batina, L., Robshaw, M. (Eds.): '*Cryptographic Hardware and Embedded Systems – CHES 2014*', (Springer, Berlin Heidelberg, 2014), pp. 433–450