A Lightweight Implementation of Keccak Hash Function for Radio-Frequency Identification Applications

Elif Bilge Kavun and Tolga Yalcin

Department of Cryptography
Institute of Applied Mathematics, METU
Ankara, Turkey
{e168522,tyalcin}@metu.edu.tr

Abstract. In this paper, we present a lightweight implementation of the permutation *Keccak-f*[200] and *Keccak-f*[400] of the SHA-3 candidate hash function Keccak. Our design is well suited for radio-frequency identification (RFID) applications that have limited resources and demand lightweight cryptographic hardware. Besides its low-area and low-power, our design gives a decent throughput. To the best of our knowledge, it is also the first lightweight implementation of a sponge function, which differentiates it from the previous works. By implementing the new hash algorithm Keccak, we have utilized unique advantages of the sponge construction. Although the implementation is targeted for Application Specific Integrated Circuit (ASIC) platforms, it is also suitable for Field Programmable Gate Arrays (FPGA). To obtain a compact design, serialized data processing principles are exploited together with algorithm-specific optimizations. The design requires only 2.52K gates with a throughput of 8 Kbps at 100 KHz system clock based on 0.13-µm CMOS standard cell library.

Keywords: RFID, Keccak, SHA-3, sponge function, serialized processing, low-area, low-power, high throughput.

1 Introduction

In recent years, the developments on digital wireless technology have improved many areas such as the mobile systems. Mobile and embedded devices will be everywhere in times to come, making it possible to use communication services and other applications anytime and anywhere. Among these mobile devices, radio-frequency identification (RFID) tags offer low-cost, long battery life and unprecedented mobility [1-2]. Today, we see RFID tags everywhere, in electronic toll collection systems, product tracking systems, libraries, passports, etc. Due to this rise in the usage of RFID tags in the past few years, research activities were started in RFID security area and security challenges have been identified.

However, security of the RFID tags is a main concern. The autonomously interacting capability of these digital devices makes them inherently unsecure. The authentication of devices and privacy are among the major critical problems. As a result, new cryptographic protocols have been proposed to preserve user privacy,

S.B. Ors Yalcin (Ed.): RFIDSec 2010, LNCS 6370, pp. 258–269, 2010.

authenticate the RFID tag communication and make it anonymous. Many works have been made on this subject, and most of them use symmetric cryptography because of the severe constraints for hardware implementations of RFID tags. In applications that demand low-cost and low-power such as RFID, the use of cryptographic functions requires the low gate count. To provide a low gate count in RFID tags, the researches have focused on block ciphers and hash functions. In [3] and [4], the use of block ciphers is discussed in more detail.

The compact realization of hash functions for RFID applications is still a major research subject. In [5], a lightweight implementation of the standard SHA-1 hash function is presented, while in [7] the hash function MAME, which is specifically designed for lightweight applications, was implemented at a very low gate count for protecting RFID tags. Our implementation differs from both via its unusual sponge construction, which offers a better gate count with a decent throughput value. We used two different permutations of SHA-3 candidate hash function Keccak - *Keccak-f*[200] and *Keccak-f*[400]. Keccak is based on sponge functions that use the sponge construction, and exploit all its advantages such as permutation-based structure, variable-length output, flexibility, functionality and security against generic attacks [6].

The paper is organized as follows: Section 2 and 3 briefly describe the sponge functions and Keccak, respectively. In Section 4, we present our serialized compact Keccak architecture and implementations of *Keccak-f*[200] and *Keccak-f*[400] for RFID applications. Section 5 summarizes the performance results for our implementations as well as comparison with straightforward parallel implementations. Section 6 is the conclusion.

2 Sponge Functions

Sponge functions can be used to generalize cryptographic hash functions to more general functions with arbitrary output lengths. They are based on the sponge construction, where the finite memory is modeled in a very simple way.

To specify the difference between the sponge functions and the sponge construction, we use the term sponge construction to define a fixed-length permutation for building a function that maps inputs of any length to arbitrary-length outputs and the term sponge functions for functions that are built using this sponge construction [8]. In section 2.1, the sponge construction will be explained.

2.1 The Sponge Construction

The sponge construction is a repetitive construction to build a function F with variable-length input and arbitrary-length output based on a fixed-length permutation f operating on a fixed number of b-bit, which is called the width. The sponge construction operates on a state of $b=r+c$ bits. r is called the bit rate and c is called the capacity. In the first step, the bits of the state are all initialized to zero. Then, the input message is padded and cut into blocks of r-bit. The construction consists of two phases, namely the absorbing phase and the squeezing phase.

In the absorbing phase, the r-bit input message blocks are XORed with the first r-bit of the state, then interleaved with the function f. After processing all of the message blocks, the squeezing phase begins.

In the squeezing phase, the first *r*-bit of the state is returned as output blocks, and then interleaved with the function *f*. The number of output blocks is chosen by the user. The block diagram of the sponge construction is shown in Figure 1.

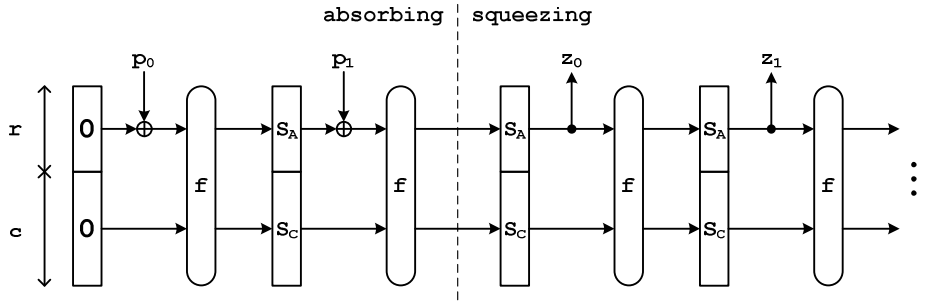

Fig. 1. The sponge construction: The sponge construction operates on a state of $b=r+c$ bits. The value r is called the *bitrate* and the value c the *capacity*.

The least significant *c*-bit of the state is never directly affected by the input blocks and never output during the squeezing phase. The capacity *c* determines the attainable security level of the construction. In sponge functions, indifferentiability framework [9] is used to assess the security of the construction. In [10], the expected complexity resistance level was approximated by the expression $2^{c/2}$. This value is independent of the output length. For example, the collisions of a random sponge which has output length shorter than *c*-bit, has the same expected complexity as a random source.

The sponge construction provides many advantages with its permutation-based structure, variable-length output and security against generic attacks. In addition, it has flexibility to choose the adequate bit rate/capacity values while using the same permutation and it is functional because it can also be used as a stream cipher, deterministic pseudorandom bit generator or mask generating function with its long output and proven security bounds to generic attacks properties.

3 Keccak

Keccak [11] is a cryptographic hash function submitted to the NIST SHA-3 hash function competition by Guido Bertoni, Joan Daemen, Michaël Peeters and Gilles Van Assche. Keccak is a family of hash functions that are based on the sponge construction and used as a building block of a permutation from a set of seven permutations. The basic component is the *Keccak-f* permutation, which consists of a number of simple rounds with logical operations and bit permutations.

3.1 The Structure of Keccak

The fundamental function of Keccak is a permutation chosen from a set of seven *Keccak-f* permutations, denoted by *Keccak-f*[*b*], where $b \in \{25,50,100,200,400,800,1600\}$ is the width of the permutation. The width *b* of the permutation is also the width of the state in the sponge construction. The state is organized as an array of 5×5

lanes, each of length w-bits, where $w \in \{1,2,4,8,16,32,64\}$, ($b$=25$w$). The *Keccak*[$r$,$c$,$d$] sponge function can be obtained by applying the sponge construction to *Keccak-f*[r+c] with the parameters capacity c, bit rate r and diversifier d and also padding the message input specifically. The pseudo-code of *Keccak-f* is given in Algorithm 1. The number of rounds n_r depends on the permutation width which is calculated by $n_r = 12+2\times l$, where $2^l = w$. This yields 12, 14, 16, 18, 20, 22, 24 rounds for *Keccak-f*[25], *Keccak-f*[50], *Keccak-f*[100], *Keccak-f*[200], *Keccak-f*[400], *Keccak-f*[800], *Keccak-f*[1600], respectively.

Algorithm 1. Pseudo-code of *Keccak-f*

$Keccak-f[b](A)\{$

for i *in* $0 \ldots n_r - 1$

$\quad A = Round[b](A, RC[i])$

return A

$\}$

$Round[b](A, RC)\ \{$

θ *step*:

$\quad C[x] = A[x,0] \oplus A[x,1] \oplus A[x,2] \oplus A[x,3] \oplus A[x,4], \quad \forall x$ *in* $0...4$

$\quad D[x] = C[x-1] \oplus ROT(C[x+1],1), \quad \forall x$ *in* $0...4$

$\quad A[x,y] = A[x,y] \oplus D[x], \quad \forall (x,y)$ *in* $(0...4, 0...4)$

ρ *and* π *steps*:

$\quad B[y, 2x+3y] = ROT(A[x,y], r[x,y]), \quad \forall (x,y)$ *in* $(0...4, 0...4)$

χ *step*:

$\quad A[x,y] = B[x,y] \oplus ((NOT\ B[x+1,y])\ AND\ B[x+2,y]), \quad \forall (x,y)$ *in* $(0...4, 0...4)$

ι *step*:

$\quad A[0,0] = A[0,0] \oplus RC$

return A

$\}$

In Algorithm 1, all of the operations on the indices are done in modulo 5. A denotes the complete permutation state array, and *A[x,y]* denotes a particular lane in that state. *B[x,y], C[x], D[x]* are intermediate variables, the constants *r[x,y]* are the rotation offsets and *RC[i]* are the round constants. *rot(w,r)* is the bitwise cyclic shift operation which moves the bit from position i into position i+r, in the modulo lane size.

The pseudo-code of the sponge function *Keccak*[r,c,d] is given in Algorithm 2, again with parameters capacity c, bit rate r and diversifier d. This description is restricted to the case of messages that span a whole number of bytes. For messages with a number of bits not dividable by 8, the details are given in [12]. In the algorithm, S denotes the state as an array of lanes. The padded message P is organized as an array of blocks P_i. The operator || denotes the byte string concatenation.

Algorithm 2. Pseudo-code of the sponge function *Keccak*[*r*,*c*,*d*]

$Keccak[r,c,d](M)\{$

Initialization and padding:

$S[x,y] = 0, \qquad \forall (x,y) \text{ in } (0...4, 0...4)$

$P = M \parallel 0x01 \parallel byte(d) \parallel byte(r/8) \parallel 0x01 \parallel 0x00 \parallel ... \parallel 0x00$

Absorbing phase:

$\forall$ block P_i in P

$S[x,y] = S[x,y] \oplus P[x+5y], \qquad \forall (x,y) \text{ such that } (x+5y) < (r/w)$

$S = Keccak-f[r+c](S)$

Squeezing phase:

$Z = empty\ string$

while output is requested

$Z = Z \parallel S[x,y], \qquad \forall (x,y) \text{ such that } (x+5y) < (r/w)$

$S = Keccak-f[r+c](S)$

return Z

$\}$

---

4 Lightweight Keccak

Fast and parallel implementations of the Keccak have already been reported [13-14]. The main components in these implementations are the *Keccak-f* round function module and the state register. Sizes of both modules depend on the choice of width, *b*, of the *Keccak-f*[*b*] permutation. In the official SHA-3 proposal, this width is chosen to be 1600 [11]. In a fully parallel implementation, this corresponds to a minimum gate count of 1600 flip-flops, 1600 inverters, 1600 AND gates, and 4864 XOR gates. Table 1 lists the equivalent gate counts for fully parallel implementations of *Keccak-f*[1600] and a few other SHA-3 candidates. Only the ones with the lowest gate counts are listed for convenience. As seen from the table, the gate count for a fully parallel implementation of Keccak is beyond the acceptable numbers for a lightweight implementation [15].

The gate count can be lowered by a serialized implementation, where the internal state is kept in a RAM based memory instead of registers, and a single datapath serves as the *Keccak-f* round function module, processing one lane at a time. However, such an approach is not really applicable to the Keccak round function. Both θ and χ steps require data from 3 lanes on the *x*-axis to compute a single lane data, whereas π transposes lanes on the *y*-axis to *x*-axis after shuffling their locations. Each of these operations will require several temporary storage registers in addition to the state

RAM. It would also be practical to replace the *Keccak-f* datapath with a simple arithmetic-logic unit, resulting in a micro-processor rather than a dedicated hardware. The number of cycles required to complete the processing of all lanes will be rather large.

Table 1. Area comparison for parallel (fast) implementation of SHA-3 candidates

Function	Area (KGE)
BLAKE-32	45.64
CubeHash16/32-h	58.87
Fugue-256	46.25
Grøstl-256	58.40
Hamsi-256	58.66
JH-256	58.83
Keccak-256	**56.32**
Luffa-224/256	44.97
Shabal-256	54.19
SHAvite-3	57.39
Skein-256-256	58.61

The RAM can be replaced with flip-flop based registers, making it possible to reach more than a single lane at a time. However, this time register cost will be equal to that of a parallel implementation, since the internal state size is independent of implementation strategy.

Variable permutation width characteristic of Keccak, together with the low data rate requirements of lightweight hash functions, presents us with an alternative solution to deal with the large internal state size. We can choose a Keccak permutation with a lower data width, without altering the overall structure of the *Keccak-f*[b] permutation.

4.1 Serialized *Keccak* Architecture

We propose the serialized architecture given in Figure 2 for our lightweight Keccak implementations. The architecture utilizes area advantages of serialized processing to the full extend. Data is processed in lanes (1/25 of the whole state). The state (circled) registers numbered 24-0 are used to store the internal state, while the four summation registers (rightmost registers numbered 4-0) store the row sums. The operational blocks which implement step of a Keccak round are the θ, ρ, π, χ and ι-modules. All of these modules, except for the π-module, operate on a single lane, reducing the combinational gate count drastically. π-step is executed in parallel on all 25 lanes. However, since it is just a fixed permutation operation, its only area cost comes from the additional multiplexers and routing. There is an additional area cost caused by the sum registers, required for the θ-step, and the two temporary registers, required for the χ-step. These extra registers are well compensated by the huge area saving caused by the serialized processing and the resulting single lane combinational blocks.

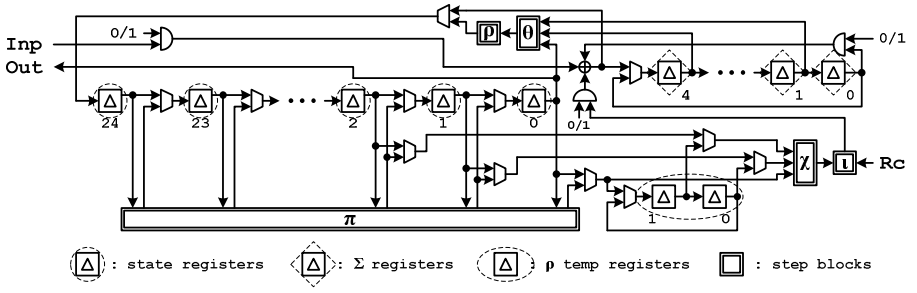

Fig. 2. Serialized Keccak architecture

In the first phase of each Keccak round, data is written in lanes into the state registers column by column while each row sum is accumulated in the sum registers in parallel. The first incoming lane is *lane*(0,0) and shifted into *state_register*[24] while *sum_register*[4] is initialized to the same value. The next incoming data is *lane* (1,0); it is shifted into *state_register*[24], *state_register*[24] into *state_register*[23]. At the same time, *sum_register*[4] is shifted into *sum_register*[3], and *sum_register*[4] is re-initialized with *lane*(1,0). At the end of the first 5 cycles, the first 5 lanes of data are in *state_registers*[24] to [20], while *sum_registers*[4] to [0] have the first column lanes of each corresponding column. In the following cycles, incoming lane data are added on to sum registers and shifted into the state registers, so that at the end of the first 25 cycles, state registers contain the full state and sum registers contain the row sums.

Starting with the next cycle, θ and ρ operations are run in parallel on each lane starting with *lane*(0,0), continuing with *lane*(1,0), *lane*(2,0), …, all the way to *lane*(4,4), covering the whole state. This phase is completed in 25 cycles. It is followed by another 25 cycles, where π, χ and ι operations are performed. π can only be executed on the whole state, therefore done in parallel with the calculation of χ for the very first lane. ι operation (round constant addition) is also done in the same cycle. In the following 24 cycles, χ operation are performed on the remaining lanes, completing the first round. We name each of these 25 cycles as "half rounds".

As an additional optimization, the row summations for the following round are also performed in parallel with π, χ and ι operations of the current round. In average, a full round takes 50 cycles to complete. The very first half round (half round "0") is used for the "absorption" of the first input block, while the following input block absorptions can be done during the second half round of each last round. The final half round (following the last input block) is used for "squeezing" of the message digest. This scheme is illustrated in Figure 3.

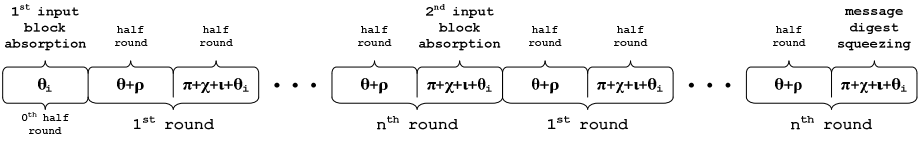

Fig. 3. Serialized Keccak data processing rounds

The whole data processing in each half round is explained by a tweaked version of Keccak, where there are 3×3 lanes in Figures 4 and 5. In our implementation, we apply the same timing to the actual 5×5 lanes configuration.

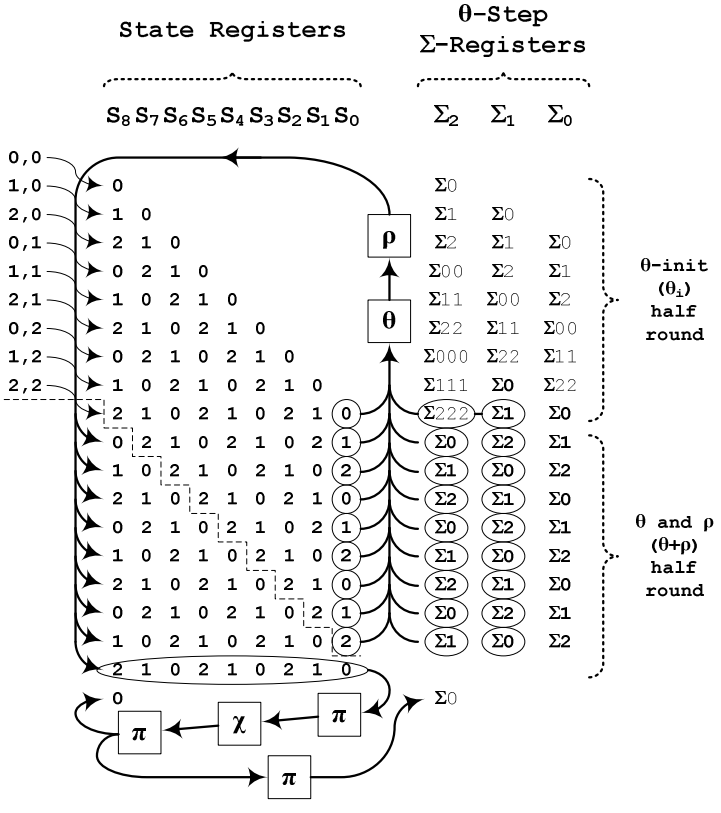

Fig. 4. Serialized Keccak operation flow and register contents during the θ-init and θ+ρ half cycles

4.2 *Keccak-f*[200] and *Keccak-f*[400] Implementations

Our first lightweight candidate is *Keccak-f*[200]. In this configuration, the lane width is chosen as 8-bits (2^l, where l=3) in accordance with the definition of Keccak [11]. The target message digest size is 64-bits. This corresponds to a capacity, c, value of 128-bits, limiting the highest achievable data rate, r, to 72-bits (200-128).

Our second candidate is *Keccak-f*[400], where the lane width is chosen as 16-bits (l=4). The target message digest size is 128-bits, resulting in a capacity value of 256-bits and data rate of 144-bits (400-256).

We have implemented both candidates using the serialized architecture presented in section 4.1, as well as using a fully parallel straightforward approach for a fair comparison. In addition, we have also implemented the original Keccak configuration (*Keccak-f*[1600]) using both the serialized and fully parallel architectures in order to demonstrate the compactness of our architecture. The comparison results and performance figures are presented in section 5.

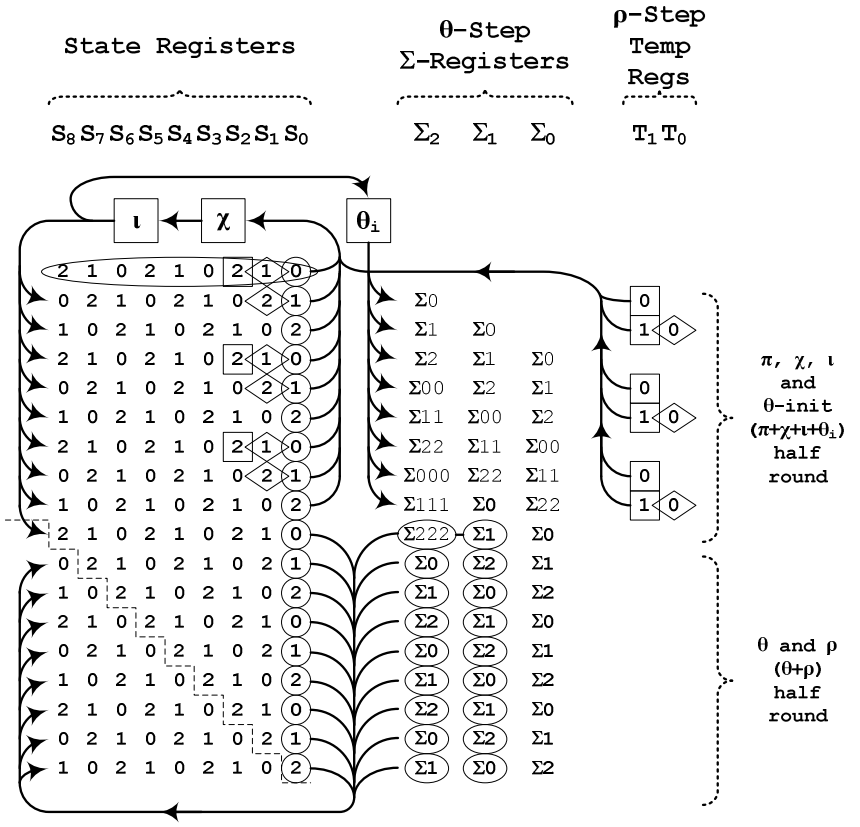

Fig. 5. Serialized Keccak operation flow and register contents during the $\pi+\chi+\iota+\theta_i$ and $\theta+\rho$ half cycles

In addition to their lane widths, the two implementations also differ in the total number of rounds. According to the Keccak specification, the total number of rounds is specified as $12+2l$, corresponding to 18 and 20 rounds for *Keccak-f*[200] and *Keccak-f*[400], respectively.

The sponge construction of Keccak hash function makes it possible to use *Keccak-f*[200] for message digests longer than 72-bits via consecutive squeezes. However, such a usage will add extra rounds to the overall hashing operation, which may be effective especially for short message lengths. Instead, we fix our functions and their respective message digest sizes, resulting in *Keccak-f*[200]-64 and *Keccak-f*[400]-128 lightweight hash functions.

We rely on our own statistical analyses as well as the security claims in the Keccak proposal for the security of these two hash function implementations. Furthermore, we assume that 64 and 128-bit message digest sizes are sufficient for RFID applications making our proposed variations and serialized architecture ideal lightweight hash functions.

5 Implementation and Performance Results

We have realized both straightforward parallel implementations and serialized implementations (using our proposed architecture) of Keccak for lane widths of 8-bits (l=3), 16-bits (l=4) and 64-bits (l=6), corresponding to *Keccak*-f[200], *Keccak*-f[400] and *Keccak*-f[1600], respectively, on a standard 0.13µm digital CMOS technology. The corresponding gate counts, throughput values and power consumptions are listed in Table 2. Additionally, we compare our lightweight candidates with MAME [7], a hash function specifically designed for lightweight applications, and a compact SHA-1 [5] implementation in Table 3.

Table 2. Performance comparison for parallel and serialized Keccak implementations

	Hash output size	Data path size	Input data size	Cycles per block	T/put at 100 KHz (Kbps)	Area (KGE)	Efficiency (bps/GE)	Power cons. (µW/MHz)
Parallel *Keccak-f*[1600]	256	64	1088	24	4533	47.63	95.40	315.1
Parallel *Keccak-f*[400]	128	16	144	20	720	10.56	68.18	78.1
Parallel *Keccak-f*[200]	64	8	72	18	400	4.9	81.63	27.6
Serial *Keccak-f*[1600]	**256**	**64**	**1088**	**1200**	**90.66**	**20.79**	**4.36**	**44.9**
Serial *Keccak-f*[800] (estimate)	**128**	**32**	**544**	**1100**	**49.45**	**13.00**	**3.80**	**28.2**
Serial *Keccak-f*[400]	**128**	**16**	**144**	**1000**	**14.4**	**5.09**	**2.83**	**11.5**
Serial *Keccak-f*[200]	**64**	**8**	**72**	**900**	**8**	**2.52**	**3.17**	**5.6**

Table 3. Performance comparison for lightweight Keccak implementation against MAME and SHA-1

	Hash output size	T/put at 100 KHz (Kbps)	Area (KGE)	Efficiency (bps/GE)
SHA-1 [5]	160	148.8	5.53	26.91
MAME [7]	256	146.7	8.1	18.10
Serialized *Keccak-f*[400]	**128**	**14.4**	**5.09**	**2.83**
Serialized *Keccak-f*[200]	**64**	**8**	**2.52**	**3.17**

The figures depict both the throughput and area drop in the serialized architecture. The throughput drop is much more drastic due to the extra cycles coming from the serialization. However, it should be noted that we are more interested in lower areas in lightweight applications, which only demand acceptable figures for throughput [16]. In that respect, even our more secure lightweight candidate *Keccak-f*[400]-128 offers a throughput of 14.4 Kbps at 100 KHz system clock, which is deemed acceptable for RFID applications, while occupying only 5.09KGE. In case, higher throughput is targeted at the expense of area, it is also possible to implement *Keccak-f*[800] using our serialized architecture, which gives an estimated throughput of 49.45 Kbps occupying 13KGE.

6 Conclusion

In this study, we have presented a pipelined serialized architecture for the SHA-3 candidate Keccak, which offers very low area and power consumption with acceptable throughput. Our architecture is especially attractive for lightweight applications when implemented with compact versions of Keccak. With its flexible structure, our architecture is very easy to modify for faster versions, at the expense of increased area. Lane-based processing can easily be turned into row or column based processing, raising the throughput by a factor of 5, while the estimated area increase is only about 50 percent. We have also shown that even straightforward parallel implementations of compact versions of Keccak offer acceptable areas (4.9KGE for parallel *Keccak-f*[200]) with very high data rates (400 Kbps at 100 KHz system clock).

References

1. European Commission, Draft Recommendation on the Implementation of Privacy, Data Protection and Information Security Principles in Applications Supported by Radio Frequency Identification (RFID), http://ec.europa.eu/yourvoice/ipm/forms/dispatch?form=RFIDRec
2. Finkenzeller, K.: RFID Handbook. John Wiley, Chichester (2003)
3. Feldhofer, M., Dominikus, S., Wolkerstorfer, J.: Strong Authentication for RFID Systems using the AES Algorithm. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 357–370. Springer, Heidelberg (2004)
4. Rolfes, C., Poschmann, A., Leander, G., Paar, C.: Ultra-Lightweight Implementations for Smart Devices - Security for 1000 Gate Equivalents. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 89–103. Springer, Heidelberg (2008)
5. O'Neill, M.: Low-Cost SHA-1 Hash Function Architecture for RFID Tags. In: Proceedings of RFIDSec (2008)
6. The KECCAK sponge function family, http://keccak.noekeon.org
7. Yoshida, H., Watanabe, D., Okeya, K., Kitahara, J., Wu, H., Kucuk, O., Preneel, B.: MAME: A compression function with reduced hardware requirements. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 148–165. Springer, Heidelberg (2007)
8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: ponge Functions. In: Ecrypt Hash Workshop (2007)
9. Maurer, U., Renner, R., Holenstein, C.: Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 21–39. Springer, Heidelberg (2004)
10. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: On the Indifferentiability of the Sponge Construction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 181–197. Springer, Heidelberg (2008)
11. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak sponge function family main document. NIST (2009) (submission to)
12. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak specifications, version 2, NIST (2009) (submission to)

13. Tillich, S., et al.: High-Speed Hardware Implementations of BLAKE, BMW, CubeHash, ECHO, Fugue, Grostl, Hamsi, JH, Keccak, Luffa, Shabal, SHAvite-3, SIMD, and Skein. In: Cryptography ePrint (November 2009)
14. Namin, A.H., Hasan, M.A.: Hardware Implementation of the Compression Function for Selected SHA-3 Candidates, CACR 2009-28 (July 2009)
15. Feldhofer, M., Rechberger, C.: A Case Against Currently Used Hash Functions in RFID Protocols. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM 2006 Workshops. LNCS, vol. 4277, pp. 372–381. Springer, Heidelberg (2006)
16. Bogdanov, A., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y.: Hash Functions and RFID Tags: Mind The Gap. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 283–299. Springer, Heidelberg (2008)