Efficient and First-Order DPA Resistant Implementations of Keccak

Begül Bilgin[3,4], Joan Daemen[1], Ventzislav Nikov[2], Svetla Nikova[3], Vincent Rijmen[3], and Gilles Van Assche[1]

[1] STMicroelectronics, Belgium
[2] NXP Semiconductors, Belgium
[3] KU Leuven, ESAT/COSIC and iMinds, Belgium,
[4] University of Twente, DIES, The Netherlands

Abstract. In October 2012 NIST announced that the SHA-3 hash standard will be based on Keccak. Besides hashing, Keccak can be used in many other modes, including ones operating on a secret value. Many applications of such modes require protection against side-channel attacks, preferably at low cost. In this paper, we present threshold implementations (TI) of Keccak with three and four shares that build further on unprotected parallel and serial architectures. We improve upon earlier TI implementations of Keccak in the sense that the latter did not achieve uniformity of shares. In our proposals we do achieve uniformity at the cost of an extra share in a four-share version or at the cost of injecting a small number of fresh random bits for each computed round. The proposed implementations are efficient and provably secure against first-order side-channel attacks.

Keywords: Keccak, side-channel attacks, threshold implementation

1 Introduction

Keccak [6] is the best-known family of sponge functions. They can be used in a wide range of modes covering the full range of symmetric cryptographic functions [4]. These functions can take as argument a secret key (e.g., encryption, Message Authentication Code (MAC) computation, authenticated encryption, etc.) and require their internal state to remain secret for security (e.g., pseudorandom sequence generators). Such functions are subject to side-channel attacks.

A Differential Power Analysis (DPA) attack, which is a very powerful side-channel attack, exploits the dependencies between the instantaneous power consumption of a device and the intermediate results of a cryptographic algorithm. Since the security of cryptographic primitives inevitably relies on the fact that an adversary does not have access to intermediate computation results, any even partial knowledge of intermediate computation results can lead to a complete breakdown of security, e.g., by revealing the key. Several countermeasures against DPA [12] have been proposed on different levels. For example, a circuit design approach that aims to balance the power consumption of different data values has been proposed in [19]. Another popular method is to randomize the intermediate values of an algorithm by masking, namely on algorithm level [2,9], at the gate level [10,20] or in combination with circuit design approaches [17]. Since the amount of information that is leaked by hardware is unknown, the security proofs are based on an idealized hardware model, resulting in requirements on the hardware that are very expensive to meet in practice.

In a threshold implementation [14,15], the sharing can have three properties: correctness, non-completeness and uniformity. Correctness is an obvious requirement which simply states that the sum of the output shares of a sharing for a function f equals f applied to the sum of the input shares as in boolean masking. Non-completeness states that each output share of a function is independent of at least one input share.

When the input shares are uniformly distributed, then a correct and non-complete sharing is provably immune to first-order DPA even in presence of glitches [14,15]. In a sequential computation, e.g., such in a function composed of rounds or in multi-stage implementations of S-boxes, the output shares will be used as input in another stage of the computation. Hence, it is also interesting to preserve the uniformity of the shares. As a first option, a sharing can be uniform, which means that the output shares are uniform if the input shares are uniform. As another option, uniformity of the output shares can be obtained by the use of fresh randomness. This last option is also called re-masking and has been done before, e.g., for the TI of AES in [13]. Eventhough re-masking can restore the uniformity of the input shares, and with it the provable security agasint first-order DPA, this requires fresh randomness on each round, which may become expensive in practice.

The designers of Keccak proposed a hardware architecture that offers protection against first-order DPA [3]. They employ the threshold implementation method with three shares, but it is not uniform and hence not provably secure against first-order DPA.

Contribution. In Section 3 we propose an alternative way for re-masking that requires less random bits than the straightforward re-masking approach as described in [13]. In Section 4 we propose a sharing that uses four shares that achieves uniformity of sharing without the introduction of fresh randomness. In Section 5 we provide the area cost and the maximum frequency of our unprotected and threshold implementations for fully parallel and slice-based architectures. We show that the area requirement for our unprotected implementations are significantly smaller than the previous KECCAK implementations and have higher frequency. Moreover, the threshold implementations with serial architecture can be considered within the limits of a lightweight implementation. In addition, we discuss a way to reduce the area cost of the threshold implementations at the cost of extra randomness. First, we briefly recall the components of KECCAK in Section 2.

2 Introduction to Keccak

KECCAK is a function with variable-length input and arbitrary-length output based on the sponge construction [4]. In this construction, a b-bit permutation f is iterated. First, the input is padded and its blocks are *absorbed* sequentially into the state, with a simple XOR operation. Then, the output is *squeezed* from the state block by block. The size of the blocks is denoted by r and called the *bitrate*. The remaining number of bits $c = b - r$ is called the *capacity* and determines the security level of the function.

The simplest use case of a sponge function is to use it as a hash function. However, a MAC function can be built by taking the concatenation of a secret key and a message as input. It is also possible to use a sponge function as a stream cipher. To this purpose, it suffices to use the secret key and a nonce as input so that the resulting output can be used as a key stream. More modes of use are described in [5].

Seven permutations, denoted KECCAK-$f[b]$, are defined with width $b = 25w$ ranging from 25 to 1600 bits, with w increasing in powers of two. The state of KECCAK-$f[b]$ is organized as a set of $5 \times 5 \times w$ bits with (x, y, z) coordinates. Coordinates are taken modulo 5 for x and y and modulo w for z. A *row* is a set of 5 bits with given (y, z) coordinates, a *column* is a set of 5 bits with given (x, z) coordinates and a *lane* is a set of w bits with given (x, y) coordinates. Moreover, the set of 5×5 bits with given z coordinates is called a *slice*.

The round function of KECCAK-$f[b]$ consists of the following steps, which are only briefly summarized here. For more details, we refer to the specifications [6].

- θ is a linear mixing layer that adds a pattern depending solely on the parity of the columns of the state.
- ρ and π displace bits without altering their value.
- χ is a degree-2 non-linear mapping that processes each row independently. It can be seen as the application of a translation-invariant

5-bit quadratic S-box:

$$a_{(x,y,z)} \leftarrow a_{(x,y,z)} + (a_{(x+1,y,z)} + 1)a_{(x+2,y,z)}.$$

- ι adds a round constant.

The number of rounds in Keccak-f is determined by the width b of the permutations. It is 12 for Keccak-$f[25]$ and increases by two for each doubling of the size. So Keccak-$f[1600]$ has 24 rounds.

3 Achieving uniformity with limited extra randomness

In this section, we focus on the three-share implementation proposed in [3]. A value x is shared as (A, B, C) if $x = A + B + C$ in $\mathbb{F}_2^n$. Seen as random variables over $\mathbb{F}_2^n$, shares (A, B, C) are said to be *uniform* if and only if $\Pr[A + B + C = x] = 1$ and for any fixed values $a, b \in \mathbb{F}_2^n$, $\Pr[A = a, B = b] = 2^{-2n}$. This definition is slightly more restrictive than the one in [14,15], as we do not consider probability distributions over native values but only over their shared representation. As the computation of cryptographic primitives such as Keccak is deterministic, this restriction does not play a role here.

3.1 The original three-share TI implementation of χ

The non-linear step of the Keccak round function is called χ. In [7] we proposed a three-share TI implementation called χ'. We denote the three shares by A, B and C and the position of the bit within a row by i (to be taken modulo 5):

$$\begin{aligned}
A'_i &\leftarrow \chi'_i(B, C) \triangleq B_i + (B_{i+1} + 1)B_{i+2} + B_{i+1}C_{i+2} + B_{i+2}C_{i+1}, \\
B'_i &\leftarrow \chi'_i(C, A) \triangleq C_i + (C_{i+1} + 1)C_{i+2} + C_{i+1}A_{i+2} + C_{i+2}A_{i+1}, \qquad (1) \\
C'_i &\leftarrow \chi'_i(A, B) \triangleq A_i + (A_{i+1} + 1)A_{i+2} + A_{i+1}B_{i+2} + A_{i+2}B_{i+1}.
\end{aligned}$$

This maps a 15-bit vector (A, B, C) to a 15-bit vector (A', B', C'). Upon inspection, we found that this mapping is not invertible and hence not uniform [14,15]. The consequence is that even if (A, B, C) is a uniform sharing of a native value x, (A', B', C') is not a uniform sharing of $\chi(x)$.

3.2 Straightforward injection of fresh random bits

Keccak-$f[1600]$ has 320 rows. For a three-share TI implementation, this means the application of Eq. (1) 320 times per round.

To convert (A', B', C') into a uniform sharing again, we can inject random bits. Re-masking is based on the following lemma.

Lemma 1. *Let (A, B, C) be n-bit shares (not necessarily uniform) of a fixed native value and (X, Y, Z) be uniform m-bit shares. Let (D, E, F) be uniform n-bit shares statistically independent of (A, B, C) and (X, Y, Z). Then, $((A+D, X), (B+E, Y), (C+F, Z))$ are uniform $n+m$-bit shares.*

Proof. First, since $A + B + C$, $D + E + F$ and $X + Y + Z$ take a fixed value with probability one, so does $(A+B+C+D+E+F, X+Y+Z)$. Then, it suffices to verify that for each fixed value $a + d, x, b + e, y$:

$$\begin{aligned}
&\Pr[A + D = a + d, B + E = b + e, X = x, Y = y] \\
=&\sum_{d,e} \Pr[D = d, E = e] \Pr[A = (a + d) + d, B = (b + e) + e, X = x, Y = y] \\
=&2^{-2n} \sum_{d,e} \Pr[A = (a + d) + d, B = (b + e) + e, X = x, Y = y] \\
=&2^{-2n} \Pr[X = x, Y = y] \\
=&2^{-2(n+m)}.
\end{aligned}$$

We get a realization of χ that satisfies the uniformity property at the cost of 2 uniformly distributed random bits P_i, S_i per bit of the state. The implementation of χ becomes:

$$\begin{aligned}
A'_i &\leftarrow \chi'_i(B, C) + P_i + S_i, \\
B'_i &\leftarrow \chi'_i(C, A) + P_i, \\
C'_i &\leftarrow \chi'_i(A, B) + S_i,
\end{aligned} \tag{2}$$

Eq. (2) can be seen as the addition of $(\chi'_i(B, C), \chi'_i(C, A), \chi'_i(A, B))$ and $(P_i + S_i, P_i, S_i)$. The result is uniform thanks to Lemma 1 as $(P_i + S_i, P_i, S_i)$ is a uniform sharing of the native value 0 obtained from independently drawn random bits.

Although from a theoretical point of view this re-masking method solves the uniformity issue raised above, the solution is not satisfactory since it requires a RNG which generates many high-quality random bits at each clock cycle.

3.3 Less randomness per row

In this section we reduce the number of required fresh random bits per round by using specific properties of χ'.

The function χ in Keccak operates on 5-bit rows. It can be seen as a specific case of a convolutional mapping operating on an n-bit circular array with updating function $x_i \leftarrow x_i + (x_{i+1} + 1)x_{i+2}$. Next Lemma is a general result that holds for any value n.

Lemma 2. *If the input (A, B, C) to χ' is shared uniformly, the output truncated to any $n-2$ consecutive bits, e.g., $(A', B', C')_{0\ldots n-3}$, is shared uniformly.*

Proof. First, consider $(A'_{n-3}, B'_{n-3}, C'_{n-3})$. It is the result of summing $(B_{n-3}, C_{n-3}, A_{n-3})$ with bits computed from A, B and C in positions $n-2$ and $n-1$. As $(B_{n-3}, C_{n-3}, A_{n-3})$ is a uniform sharing of x_{n-3} independent of input bits in positions $n-2$ and $n-1$, Lemma 1 applies and hence $(A'_{n-3}, B'_{n-3}, C'_{n-3})$ is a uniform sharing.

Assuming $(A', B', C')_{i+1\ldots n-3}$ is a uniform sharing, we can prove that $(A', B', C')_{i\ldots n-3}$ is a uniform sharing. (A'_i, B'_i, C'_i) is the result of summing (B_i, C_i, A_i) with bits computed from $(A, B, C)_{i+1\ldots i+2}$. As (B_i, C_i, A_i) is a uniform sharing of x_i and is independent of input bits in positions $i+1$ and $i+2$ and of $(A', B', C')_{i+1\ldots n-3}$, Lemma 1 applies and hence $(A', B', C')_{i\ldots n-3}$ is a uniform sharing. This can be extended till $(A', B', C')_{0\ldots n-3}$. □

Further (cyclic) extensions to include $(A', B', C')_{n-1}$ or $(A', B', C')_{n-2}$ is not possible as $(B_{n-2}, C_{n-2}, A_{n-2})$ is not independent of $(A', B', C')_{0\ldots n-3}$ and Lemma 1 no longer applies.

Lemma 2 says that the truncated output with two successive bits removed is uniform. As a consequence, one can repair uniformity using only 4 fresh random bits P_{n-2}, P_{n-1}, S_{n-2}, S_{n-1}. In particular, we just apply Eq. (2) with $P_i = S_i = 0$ for $i \leq n-2$.

We would like to point out that this result can also be obtained using virtual variables as proposed in [8]. Namely, let us consider each of the first two equations of χ as equations depending on one more variable Y and Z, respectively. Let (A_i, B_i, C_i) be a sharing of x_i and (Y_1, Y_2, Y_1+Y_2), (Z_1, Z_2, Z_1+Z_2) be a (therefore virtual) sharing of Y, Z then exactly the same result is obtained as in Lemma 2: 4 additional bits suffice to make the sharing uniform.

We decreased the number of fresh random bits per round from 10 to 4 bits per row. However, for Keccak-f[1600] this is $320 \times 4 = 1280$ bits, still too expensive in practice.

3.4 Jointly satisfying uniformity

In this section we consider the uniformity at the level of the full state rather than in the individual rows. We propose a TI implementation of χ with interaction between the rows that achieves almost uniformity at the level of the full state, greatly reducing the required number of fresh random bits per round.

Let us for convenience number the rows with index $j = y + 5z$. The idea is to make the sharing at the output of row $j + 1$ uniform by using input at row j. In straightforward way, we add $(A + B, A, B)$ at the input of row j to the output (A', B', C') of row $j + 1$. This is again a straightforward application of Lemma 1. Note that to satisfy the independence required by Lemma 1, the last row still requires injection of four fresh random bits for achieving uniformity, as in Eq. (2). The circuit complexity can be reduced greatly by combining this with Lemma 2. As a matter of fact, we have to add $(A + B, A, B)$ at the input of row j to the output (A', B', C') of row $j + 1$ in only two successive bit positions. Care must be taken in the bit positions used in each row so as to be able to rely on Lemma 2.

The above reasoning points out that each row individually can become uniform. The key point, however, is to show that the joint application on the entire state yields a uniform realization of χ. This is what the theorem below will show.

We denote the three shares of the whole state by (A, B, C), and a 5-bit row of the state as $(A^{(j)}, B^{(j)}, C^{(j)})$ with $j \in \mathbb{Z}_{5w}$. Then, the implementation of χ becomes:

$$\begin{aligned} A_i'^{(j)} &\leftarrow \chi_i'(B^{(j)}, C^{(j)}) + A_i^{(j-1)} + B_i^{(j-1)}, \\ B_i'^{(j)} &\leftarrow \chi_i'(C^{(j)}, A^{(j)}) + A_i^{(j-1)}, \\ C_i'^{(j)} &\leftarrow \chi_i'(A^{(j)}, B^{(j)}) + B_i^{(j-1)}, \end{aligned} \tag{3}$$

if $j > 0$ and $i \in \{3, 4\}$. Otherwise, Eq. (2) applies when $j = 0$, and Eq. (1) suffices for positions $i \leq 2$.

Theorem 1. *If the (whole state) input (A, B, C) to Eq. (3) if $j > 0$ and $i \in \{3, 4\}$, to Eq. (2) if $j = 0$ and $i \in \{3, 4\}$ and to Eq. (1) if $i \leq 2$, is shared uniformly, then the (whole state) output (A', B', C') is shared uniformly.*

Proof. We can apply Lemma 1 recursively, with j starting at $j = 5w - 1$ and going down to $j = 0$. Everytime, the reasoning is to show that if $(A'^{(j+1 \dots 5w-1)}, B'^{(j+1 \dots 5w-1)}, C'^{(j+1 \dots 5w-1)})$ is uniform, then it is also uniform for rows j to $5w - 1$.

Following Eq. (3), the sharing $(A'^{(j)}, B'^{(j)}, C'^{(j)})$ is obtained by adding $(\chi'(B^{(j)}, C^{(j)}), \chi'(C^{(j)}, A^{(j)}), \chi'(A^{(j)}, B^{(j)})$ and $(A^{(j-1)}+B^{(j-1)}, A^{(j-1)}, B^{(j-1)})$ for bit positions $i \in \{3, 4\}$. The latter expression is a uniform sharing of 0 and independent of the rows with indexes j and higher. From Lemma 2,

$(\chi'(B^{(j)}, C^{(j)}), \chi'(C^{(j)}, A^{(j)}), \chi'(A^{(j)}, B^{(j)})$ is already uniform when restricted to bit positions 0 to 2. The conditions of Lemma 1 are thus satisfied and $(A'^{(j \ldots 5w-1)}, B'^{(j \ldots 5w-1)}, C'^{(j \ldots 5w-1)})$ is uniform

If $j = 0$, the same reasoning applies, except that bit positions $i \in \{3, 4\}$ are obtained as in Eq (2). $\square$

The cost is four random bits per round, some additional XORs, registers and extra routing. As far as randomness is concerned, this amounts to 96 bits for the 24 rounds of Keccak-f[1600], which is small compared to the 3200 random bits needed to represent the input state in three shares.

4 Achieving uniformity with four shares

A uniform 3-share threshold implementation for χ or for any of its affine equivalent is not found yet. We present a uniform sharing of χ with 4 shares. For $i = 0, 1, 2, 4$, we have:

$$\begin{aligned}
&A'_i \leftarrow B_i + B_{i+2} + ((B_{i+1} + C_{i+1} + D_{i+1})(B_{i+2} + C_{i+2} + D_{i+2})),\\
&B'_i \leftarrow C_i + C_{i+2} + (A_{i+1}(C_{i+2} + D_{i+2}) + A_{i+2}(C_{i+1} + D_{i+1}) + A_{i+1}A_{i+2}),\\
&C'_i \leftarrow D_i + D_{i+2} + (A_{i+1}B_{i+2} + A_{i+2}B_{i+1}),\\
&D'_i \leftarrow A_i + A_{i+2},
\end{aligned} \tag{4}$$

and for the remaining 3rd coordinate function we have:

$$\begin{aligned}
&A'_3 \leftarrow B_3 + B_0 + C_0 + D_0 + ((B_4 + C_4 + D_4)(B_0 + C_0 + D_0)),\\
&B'_3 \leftarrow C_3 + A_0 + (A_4(C_0 + D_0) + A_0(C_4 + D_4) + A_0A_4),\\
&C'_3 \leftarrow D_3 + (A_4B_0 + A_0B_4),\\
&D'_3 \leftarrow A_3.
\end{aligned} \tag{5}$$

We found this sharing by using Theorem 2 of [8]. Namely, we first searched through all affine equivalent S-boxes of χ, i.e., $\chi'' = \chi(A(x))$, where $A(x)$ is an affine permutation and we found the ones that can be shared with a direct sharing. Next, we applied the corresponding inverse affine transformation to the found direct sharing to generate a uniform sharing for the function χ. We chose the one that has the smallest area over all the candidates. Therefore, this uniform sharing (although derived and close to direct) is not a direct sharing and that is why the shares can not be computed in a circular manner.

5 Hardware implementations

There are several reports on different implementations of unprotected Keccak-f that uses different platforms, architectures and libraries [1]. In this work, we provide unprotected (plain) and threshold implementations of Keccak-f with a round-based (parallel, Fig. 1) and a slice-based (serial, Fig. 2) architecture. We used ModelSim to verify the correctness of our implementations and Synopsys with FARADAY, FSA0A-D and FSC0H-D libraries which are standard cell libraries tailored for UMC 0.18μm and UMC 0.13μm logic processes respectively to observe and compare the accurate area cost and maximum frequency with the previous works. For all our designs, we also provide the results with NANGATE $45nm$ standard cell library which is free and can be used for further comparison. The D flip-flops that take the output of a 2×1 MUX as input are implemented as scan flip-flops to reduce the area.

In the following sections, we first describe the unprotected Keccak architectures. Then, we build our threshold implementations on those architectures.

5.1 Unprotected implementations

In our parallel implementation (Fig. 1), we fixed the rate to be at most 1024 bits. The architecture of the round function Keccak-f for this implementation is straightforward with 320 parallel instances of χ. The function θ is implemented in a slice-based manner. Namely, the 5-bit XOR of every row in each slice (i.e., the column parity) X_i, where $i \in \{0, \ldots, 63\}$ is calculated in parallel [7]. For each slice, the rotated values of X_i and X_{i-1} are XORed. This new value is concatenated five times to generate a 25-bit value which will then be XORed to its corresponding slice. With this method, the θ function can be calculated with a low cost. The rest of the linear layer, i.e., ρ and π, are executed on the whole 1600-bit state as a simple wiring and the output in each round is written to a 1600-bit register. Hence, one iteration of Keccak-f[1600] takes 24 clock cycles.

On the other hand, the serial implementation (Fig. 2) operates on the 25-bit slices. It takes 25 bits in each clock cycle starting from slice 0. The input is written to the register R_{63} after the implementation of θ, which takes as input the 5-bit XOR of every row of each input slice in the mentioned clock cycle and the previous cycle with the exception of the first slice. This is repeated for 64 cycles as the data in the registers are shifted from R_{i+1} to R_i for $i \in \{0, \ldots, 62\}$. θ for the first slice is completed in the 64th clock cycle together with the last slice. ρ and π

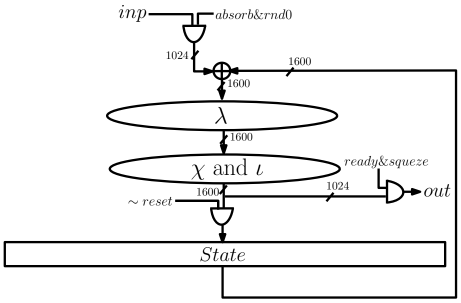


Fig. 1: Schematic of the round-based implementation of Keccak-f

are simple wirings executed on the same clock cycle as well. We can consider this one round of 64 cycles as the initialization round. For the following rounds, the input to θ is the output of the five χ functions executed in parallel on the slice R_0 followed by the XOR of the round constant. The output is taken from the output of the round constant injection starting from the first clock cycle of the 25th round. With this implementation, one iteration of Keccak-f[1600] takes $64 \times 25 = 1600$ clock cycles and costs around 10kGE in area. We should note that it is possible to have implementations that work on 2 or 4 slices per cycle and are faster but require larger area as a trade-off. In this paper, we focus on a small implementation.

Both of these unprotected implementations are noticeably smaller than the implementations reported so far which use standard cell libraries for state storage and still provide a high frequency. On the other hand, the smallest design so far, that is proposed in CHES'13 [16] uses RAM macros and requires more clock cycles for one iteration. More detailed comparison for after synthesis results is given in Table 1.

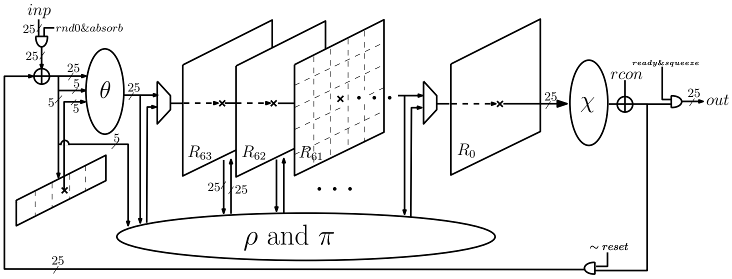


Fig. 2: Schematic of the slice-based implementation of Keccak-f

5.2 Threshold implementations

We propose two different types of threshold implementations. In the first type, we use as little random bits as we can. Namely, except for the initial sharing, we use at most four bits of randomness per round. In the second type, however, we relax this restriction on using minimum amount of randomness in order to reduce the area. In all these versions, we assume that the input shares are provided from an outside source, such that the sum of the shares is the unshared message.

For the first type of TI, we implement two versions as described in Section 3 and 4 and we use three and four shares, respectively, throughout the entire implementations. Hence, we need respectively three and four times the registers compared to the unprotected implementation. The linear layer is also tripled (and quadrupled), such that each works on one share only. During the χ operations, these shares are used together as described in (3)-(5). The round constant is inserted in one share only. In the case of the parallel three share implementation, we need 640-bit extra registers to store the re-masking masks since we need to complete the re-masking one clock cycle later as described in Section 3. Also, because of this re-masking, the output is ready one clock cycle after the last χ operation therefore one KECCAK-f takes 25 clock-cycles.

As expected, for the parallel implementations the cost of the combinational logic exceeds the cost of the register, since there are too many instances of θ and χ. Even though these implementations are fast, the parallel threshold implementations are quite big and can no longer be called efficient implementations, when applied to bigger versions of KECCAK.

When the serial implementations are considered, the register cost is the dominant cost in the architecture whereas the θ and χ layers together is only 4% of the overall implementation (Table 1). Note that for the three-share implementation, we need to keep the random bits from the previous χ function to the next (as described in Section 3) in every clock cycle which requires 4-bit register. Also, for proper re-masking, we need to use an extra 10-bit register to store the values after the χ operation which leads to a decrease of one clock cycle per round in speed.

The threshold implementations of the serial architecture have the same size as the unprotected parallel implementation. One can, of course, have an implementation operating on more than one slice to increase the speed with a relatively small cost.

[1] Uses RAM macros

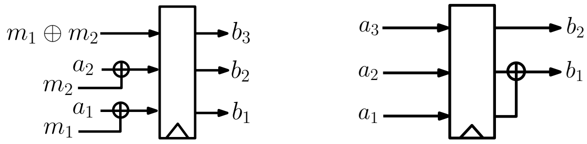
Table 1: Synthesis results for different implementations of Keccak

	Area (kGE)						Rand. bit	Clock	Freq.
Design	State	θ	χ	ANDs/XORs	Other	TOTAL	per round	Cycles	MHz
	UMC 0.18μm standard cell library								
Parallel	9.0	9.3	7.0	8.1	0.1	33.5	-	24	572
Parallel-3sh	27.2	27.8	55.4	31.4	3.5	145.3	4	25	516
Parallel-4sh	36.3	37.1	68.8	31.9	0.1	174.2	-	24	513
Serial	10.1	0.1	0.1	0.2	0.3	10.8	-	1600	555
Serial-3sh	30.4	0.4	0.8	0.7	0.8	33.1	4	1625	553
Serial-4sh	40.5	0.6	1.0	0.7	0.3	43.1	-	1600	572
	UMC 0.13μm standard cell library								
Parallel	8.0	8.6	6.4	7.5	0.1	30.6	-	24	855
Parallel-3sh	24.0	25.7	52.8	29.4	3.3	135.2	4	25	746
Parallel-4sh	32.0	34.2	61.6	29.7	0.1	157.6	-	24	735
Serial	10.0	0.1	0.1	0.2	0.2	10.6	-	1600	752
Serial-3sh	30.0	0.4	0.8	0.7	0.7	32.6	4	1625	820
Serial-4sh	40.0	0.5	0.9	0.7	0.3	42.4	-	1600	775
	NANGATE 45nm standard cell library								
Parallel	9.0	6.4	5.6	7.0	0.1	28.1	-	24	690
Parallel-3sh	27.2	19.2	40.6	25.9	3.7	116.6	4	25	592
Parallel-4sh	36.3	25.6	48.7	28.7	0.1	139.4	-	24	588
Serial	12.2	0.1	0.1	0.2	0.2	12.8	-	1600	775
Serial-3sh	36.8	0.3	0.6	0.5	0.8	39.0	4	1625	645
Serial-4sh	49.0	0.4	0.8	0.6	0.3	51.1	-	1600	633
	UMC 0.18μm standard cell library								
Parallel-[18]	N/A	N/A	N/A	N/A	N/A	56.7	-	25	488
	STM and UMC 0.13μm standard cell library								
Parallel Keccak team	N/A	N/A	N/A	N/A	N/A	48.0	-	24	526
Serial-[11]	N/A	N/A	N/A	N/A	N/A	20.0	-	1200	N/A
Serial-[16][1]	N/A	N/A	N/A	N/A	N/A	5.9	-	15427	61

5.3 An architecture with 2 shares for the linear part

Working on three or four shares throughout the whole implementation leads to a high area since the size of the state is big as a result of adopting the 1600-bit permutation. Furthermore, the cost of the linear θ layer is very close to the register cost as we converge to the parallel implementation (Table 1) because of multiple XORs per bit. For this second type of threshold implementation, we propose a way to reduce the area at the cost of extra random bits.

We can use two shares for the linear part λ of the Keccak-f. Then we face the problem of increasing or decreasing the number of shares for the nonlinear layer. The re-sharing from 2 to 3 shares can be done as in Fig. 3a one clock cycle before going through the χ layer as these three new shares need to be written into registers to avoid leakage. Note, that we do not anymore need to have a uniform χ implementation as this re-sharing will also serve as re-masking in the input of the nonlinear function. Therefore, we will only consider the χ implementation with three shares and direct sharing. Moreover, reducing the number of shares from 3 to 2 can be done by only a single XOR as shown in Fig. 3b since linear layers do not require uniform input shares.



(a) 2 to 3

(b) 3 to 2

Fig. 3: Resharing

With this approach, we will need 1 more clock cycle per round for the round-based architecture and 10 extra bits of randomness for each instance of the χ function. Applying the method in a straightforward way will cost 3200 bits of extra randomness. However it is possible to use the idea of Section 3 and borrow randomness from the input of the previous instances of the χ function.

For a parallel implementation, this approach decreases the cost of the linear layer and the ANDs and XORs only. We need to put a register between the 2-to-3 re-sharing and the χ layer, in order to safeguard against the possibility that some of the masks do not arrive on time. Moreover, there is the extra cost of the XORs during the re-masking that compensates the area saved in the linear layer. In the end, such a parallel implementation will not save area and moreover it needs more randomness which is not preferable.

For a serial architecture, this approach is more efficient. To give an example from our slice-based implementation, we need to increase the number of shares when we shift the data in the register R_1 to the register R_0 and decrease the number of shares with the shift from R_{63} to R_{62}. Even though the θ layer is still applied on three shares, the registers from R_1 to R_{62} only requires two instances. Besides, the extra cost of re-masking is small since we only need to increase or decrease the number of shares on one slice. As a result, this implementation will require 30% less area for the cost of four extra random bits per round and 96 extra random bits for one Keccak-f as we need 10 bits of randomness per round.

6 Conclusions

We presented the first implementations of KECCAK that satisfy the three properties of threshold implementations. At the moment, it seems that at least four shares are required in order to be able to satisfy simultaneously correctness, non-completeness and uniformity. Implementations with three shares require extra random bits in each round. We showed how the amount of extra random bits can be brought down to as little as four per round. To illustrate our work, we made six hardware implementations and compared their merits. We have shown that even though threshold implementations increase the area significantly, by using a serial architecture instead of a parallel one, this increase can be compensated.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments. In addition, this work has been supported in part by the Research Council of KU Leuven (OT/13/071), B. Bilgin was partially supported by the Flemish Government by the project G.0B421.13N., and V. Nikov was supported by the European Commission (FP7) within the Tamper Resistant Sensor Node (TAMPRES) project with the contract number 258754.

References

1. ATHENa: Automated tool for hardware evaluation. http://cryptography.gmu.edu/athena/.
2. M.-L. Akkar and C. Giraud. An implementation of DES and AES, secure against some attacks. In *CHES*, volume 2162 of *LNCS*, pages 309–318. Springer, 2001.
3. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Building power analysis resistant implementations of KECCAK. Second SHA-3 candidate conference, August 2010.
4. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions, January 2011.
5. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. In *Selected Areas in Cryptography (SAC)*, 2011.
6. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The KECCAK reference, January 2011.
7. G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer. KECCAK implementation overview, September 2011.
8. B. Bilgin, S. Nikova, V. Nikov, V. Rijmen, and G. Stütz. Threshold implementations of all 3 × 3 and 4 × 4 S-boxes. In *CHES*, volume 7428 of *LNCS*, pages 76–91. Springer, 2012.

9. S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. Towards sound approaches to counteract power-analysis attacks. In *CRYPTO*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.
10. Y. Ishai, A. Sahai, and D. Wagner. Private circuits: Securing hardware against probing attacks. In *CRYPTO*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.
11. Elif Bilge Kavun and Tolga Yalcin. A lightweight implementation of Keccak hash function for radio-frequency identification applications. In *Proceedings of the 6th international conference on Radio frequency identification: security and privacy issues*, RFIDSec'10, pages 258–269, Berlin, Heidelberg, 2010. Springer-Verlag.
12. P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. In M. Wiener, editor, *Advances in Cryptology – Crypto '99*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999.
13. A. Moradi, A. Poschmann, S. Ling, C. Paar, and H. Wang. Pushing the limits: A very compact and a threshold implementation of AES. In *EUROCRYPT*, volume 6632 of *LNCS*, pages 69–88. Springer, 2011.
14. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. In P. J. Lee and J. H. Cheon, editors, *ICISC*, volume 5461 of *Lecture Notes in Computer Science*, pages 218–234. Springer, 2008.
15. S. Nikova, V. Rijmen, and M. Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *J. Cryptology*, 24(2):292–321, 2011.
16. Peter Pessl and Michael Hutter. Pushing the limits of SHA-3 hardware implementations to fit on RFID. In Springer, editor, *Cryptographic Hardware and Embedded Systems - CHES 2013, 14th International Workshop, Santa Barbara, California, USA, August 20-23, 2013, Proceedings.*, volume 8086 of *Lecture Notes in Computer Science*, pages 126 – 141. Springer, 2013.
17. T. Popp and S. Mangard. Masked dual-rail pre-charge logic: DPA-resistance without routing constraints. In *CHES*, volume 3659 of *LNCS*, pages 172–186. Springer, 2005.
18. Stefan Tillich, Martin Feldhofer, Mario Kirschbaum, Thomas Plos, Jörn-Marc Schmidt, and Alexander Szekely. Uniform evaluation of hardware implementations of the round-two SHA-3 candidates. In *The Second SHA-3 Candidate Conference, Santa Barbara, USA, August 23-24, 2010*, pages 1 – 16, 2010.
19. K. Tiri and I. Verbauwhede. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *DATE*, pages 246–251. IEEE Computer Society, 2004.
20. E. Trichina, T. Korkishko, and K.-H. Lee. Small size, low power, side channel-immune AES coprocessor: Design and synthesis results. In *AES Conference*, volume 3373 of *LNCS*, pages 113–127. Springer, 2005.