Building power analysis resistant implementations of KECCAK

Guido Bertoni[1], Joan Daemen[1], Michaël Peeters[2], and Gilles Van Assche[1]

[1] STMicroelectronics
[2] NXP Semiconductors

Abstract. In this paper we report on KECCAK implementations that offer a high level of resistance against power analysis by using the technique of masking (secret sharing). In software, we show that two shares are required and if implemented carefully, sufficient. In dedicated hardware, three shares are required. We show that Multi-Gbit/s. throughput can be obtained with cores of area around 100 KGates. We demonstrate that there is a trade-off between area and performance by detailing two different architectures. Finally, we give arguments why the technique of secret sharing offers a very high level of protection against power analysis.

Keywords: KECCAK, side-channel attacks, secret sharing

1 Introduction

Sponge functions, among which KECCAK, can be used in a wide range of modes covering the full range of symmetric cryptography functions. We refer to [4,5,3] for examples. This includes functions that take as argument a secret key such as encryption, decryption, message authentication code (MAC) computation, authenticated encryption and key derivation. Some other functions do not take a secret key but take as input data that should remain secret such as pseudorandom sequence generators or commit-challenge-response protocols. If such functionality is desired on devices to which an adversary has some kind of physical or logical access, protection against side channel and fault attacks is appropriate [2].

Side channel and fault attacks are attacks that do not exploit an inherent weakness of an algorithm, but rather a characteristics of the implementation. For their security cryptographic primitives inevitably rely on the fact that an adversary does not have access to intermediate computation results. As a consequence, even partial knowledge of intermediate computation results can give a complete breakdown of security, e.g., by allowing computation of the key. However, actual implementations may leak information on such results via characteristics such as computation time, power consumption or electromagnetic radiation. Another condition for the security of cryptographic primitives is that they are executed without faults. An implementation that makes faults, or can be manipulated to make faults, may be completely insecure.

In this paper we concentrate on countermeasures against power analysis and electromagnetic radiation that make use of the algebraic properties of the step functions of KECCAK. In the remainder of this paper we will speak only about power analysis implying also electromagnetic analysis. The main difference between power analysis and electromagnetic analysis is that in the latter the adversary can make more sophisticated measurements and that the physical and electronic countermeasures are different. The countermeasures at the algorithmic level are however the same for both.

As far as timing attacks are concerned, it is straightforward to implement KECCAK in such a way that its execution time is independent of the input it processes, both in software as in hardware. Protection against fault attacks can be achieved by countermeasures that are independent of the cryptographic primitive being protected: fault detection at software level, at hardware level and

by performing computations multiple times and verifying that the results are equal. Particularly good sources of information on side channel attacks and countermeasures are the proceedings of the yearly Cryptographic Hardware and Embedded Systems (CHES) conferences (`http://www.chesworkshop.org/`) and the text book [19]. A good source of information on fault attacks and countermeasures are the proceedings of the yearly Fault Diagnosis and Tolerance in Cryptography (FDTC) workshops (`http://conferenze.dei.polimi.it/FDTC10/index.html`).

This paper is organized as follows. In Section 2 we remind the reader of the nonlinearity in KECCAK, especially relevant in the implementation of power analysis countermeasures. In Section 3 we give an overview of the different types of power analysis and the countermeasures, thereby focusing on secret sharing. In Section 4 we discuss how for a software implementation two shares are sufficient to give a high level of protection against power analysis. In Section 5 we show that in dedicated hardware implementations three shares are required and propose two architectures for power analysis resistant KECCAK implementations. In Section 6 we discuss the security of these three-share architectures. Finally in Section 7 we report on the preliminary power analysis simulations we did on one of the proposed hardware architectures.

2 The nonlinearity in KECCAK

KECCAK is a sponge function family that makes use of an underlying set of permutations called KECCAK-f. KECCAK is specified in [1] and its design is motivated and described in [4].

The security-relevant processing in KECCAK consists in the GF(2) addition (XOR) of the input blocks (key, message, IV, etc.) to a part of the state and the application of KECCAK-f on the state. The KECCAK-f permutations consist of the alternation of a round function with the bitwise addition of round constants ι. The round function consists of a sequence of steps that can be decomposed in a step λ linear over GF(2) with $\lambda = \pi \circ \rho \circ \theta$ followed by a step χ that has algebraic degree 2 in GF(2). More specifically, χ modifies each bit of the state by adding to it the product (AND) of two other state bits:

$$x_i \leftarrow x_i + (x_{i+1} + 1)x_{i+2} \ . \tag{1}$$

3 Power analysis

The general set-up of power (and electromagnetic) analysis is that the attacker gets one or more traces of the measured power consumption. If only a single trace suffices to mount an attack, one speaks about simple power analysis (SPA). However, the dependence of the signal in the variables being computed is typically small and obscured by noise. This can be compensated for by taking many traces, each one representing an execution of the cryptographic primitive with different input values. These many traces are then subject to statistical methods to retrieve the key information. These attacks are called differential power analysis (DPA) [12]. An important aspect in these attacks is that the traces must be *aligned*: they must be combined in the time-domain such that corresponding computation steps coincide between the different traces.

In DPA one distinguishes between *first order* DPA and *higher order* DPA. In first-order, the attacker is limited to considering single time offsets of the traces. In m-th order the attacker may incorporate up to m time offsets in the analysis. Higher-order attacks are in principle more powerful but also much harder to implement [19].

In correlation power analysis (CPA) [6] one exploits the fact that the power consumption may be correlated to the value of bits (or bitwise differences of bits) being processed at any given moment: there is a difference in the expected value of the power consumption. In short, one exploits this by

taking many traces and partitioning them in two subsets: in one set, a particular bit, the *target bit*, is systematically equal to 0 and in the other it is equal to 1. Then one adds the traces in each of the two sets and subtracts the results giving the compound trace. If now at any given time the power consumption is correlated to the target bit, one sees a high value in the compound trace. One can use this to retrieve key information by taking a target bit that depends on part of the key and trying different partitions based on the hypothesis for that part of the key. If wrong key guesses result in a partition where the bits of intermediate results are more or less balanced, the compound trace of the correct key guess will stand out. Note that if the power consumption is correlated to bit values or differences, it is also correlated to the Hamming values of words or Hamming distances between words.

Later, more advanced ways to measure the distance between distributions were introduced. In particular, mutual information analysis (MIA) [10] is a generalization of CPA in the sense that instead of just exploiting the fact that different bit values may result in different expected power consumption values, it is able to exploit the difference between the distributions of the power consumption for a bit being 0 or 1 respectively. So in short, when the power consumption distributions of a bit equal to 0 or 1 have equal mean values but different shapes, CPA will not work while MIA may still be able to distinguish the two distributions.

3.1 Different types of countermeasures

In the light of power analysis attacks, one must attempt implementing the cryptographic primitives such that the effort (or cost) of the adversary for retrieving the key is too high for her to be interesting. An important countermeasure is implementing the cryptographic primitives such that the power consumption and electromagnetic radiation leak as little as possible on the secret keys or data. Countermeasures can be implemented at several levels:

Transistor level Logical gates and circuits are built in such a way that the information leakage is reduced;

Platform level The platform supports features such as irregular clocking (clock jitter), random insertion of dummy cycles and addition of noise to power consumption;

Program level The order of operations can be randomized or dummy instructions can be inserted randomly to make the alignment of traces more difficult;

Algorithmic level The operations of the cryptographic algorithm are computed in such a way that the information leakage is reduced;

Protocol level The protocol is designed such that it limits the number of computations an attacker can conduct with a given key.

As opposed to protection against cryptographic attacks, protection against side channel attacks is never expected to be absolute: a determined attacker with a massive amount of resources will sooner or later be able to break an implementation. The engineering challenge is to put in enough countermeasures such that the attack becomes too expensive to be interesting. Products that offer a high level of security typically implement countermeasures on multiple levels.

The countermeasures at transistor level are independent of the algorithm to be implemented. Examples are WDDL [20] or SecLib [11]. These types of logic are evolutions of the dual rail logic, where a bit is coded using two lines in such a way that all the logic gates consume the same amount of energy independently of the values. They imply dedicated hardware for cryptography which takes more area, requires dedicated industrialization processes and is in general more expensive. Moreover, while these countermeasures may significantly reduce the information leakage, there always remains some leakage.

The countermeasures at platform level are also independent of the algorithm to be implemented. By randomizing the execution timing, alignment of power traces is made more difficult. The addition of noise to the power consumption increases the required number of traces. The timing randomization is particularly efficient against higher-order DPA as there the signals must be aligned in multiple places and any misalignment severely limits the effectiveness of attack. The addition of noise is very efficient against attacks that look for dependencies in higher moments of the distributions, such as MIA (see Section 6).

The countermeasures at program level are partially dependent on the algorithm to be implemented. Insertion of dummy instructions is possible in any algorithm while changing the order of operations may be easier for some algorithms than for others.

The countermeasures at protocol level are also independent of the algorithm. However, what can be done at this level depends on the requirements of the application and in many cases the possibilities are limited.

Finally, the countermeasures at algorithmic level depend on the basic operations used in the algorithm. This is the type of countermeasures where the choice of operations in the cryptographic primitive is relevant. One of the countermeasures of this type is that of secret sharing (or masking) and KECCAK is particularly well suited for it.

3.2 Secret sharing

Masking is a countermeasure that offers protection against DPA at the algorithmic level. It consists of representing variables processed by a cryptographic primitive by two or more shares (as in secret sharing) where the (usually bitwise) sum of the shares is equal to the *native* variable. Subsequently the program or circuit computes the cryptographic primitive using the shares in such a way that the processed variables are independent from the native variables. Whether this is possible depends on the details of the cryptographic primitive and the type of masking. In any case, to achieve independence for a native variable, all but one of its shares must be generated (pseudo-)randomly for each execution of the cryptographic primitive. Clearly, the generation of the shares, the masking operation of the input words and unmasking operation of output, usually considered out of scope of DPA attacks, must also be carefully implemented to limit information leakage. For masking to be effective, the adversary shall have as little information as possible on the value of the shares.

Taking two shares offers protection against first-order DPA under the condition that all processed bits and their joint behavior at any time are independent from native variables. Providing protection against m-th order DPA requires at least $m + 1$ shares.

Computing a linear function λ on the shares of a variable is straightforward. If we represent a native variable x by its shares x_i with $x = \sum_i x_i$ we can compute the shares y_i of $y = \lambda(x)$ by simply applying λ on the individual shares: $y_i = \lambda(x_i)$. A function that consists of the addition of a constant can be performed by adding it to a single share. As all separate operations are performed on shares that are independent of native variables, this provides protection against first-order DPA.

Computing a nonlinear function on the shares assuring all variables processed are independent of native variables depends on the nonlinear function at hand. This is in general a non-trivial problem. We refer to [2] for some examples. In this paper we limit ourselves to the nonlinear step mapping χ in the round function of KECCAK-f.

4 Software implementation using two-share masking

For software implementations, we have studied the application of masking with two shares, denoted by a and b. The step mapping χ is very similar to nonlinear step mapping γ in BASEKING, the

cipher that is the subject of [7] and hence the techniques shown there can be readily applied. We must now compute the shares of x at the lefthand side of Equation (1) in such a way that all intermediate variables are independent of native variables. This can be realized by implementing following equations:

$$\begin{aligned} a_i &\leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} \\ b_i &\leftarrow b_i + (b_{i+1} + 1)b_{i+2} + b_{i+1}a_{i+2} \ . \end{aligned} \tag{2}$$

To achieve independence from native variables, the order in which the operations are executed is important. If the expressions are evaluated left to right, it can be shown that all intermediate variables are independent from native variables. The computations of all terms except the rightmost one involve only variables of a single share, hence here independence from x is automatic. For the addition of the mixed term to the intermediate result of the computation, the presence of $a[i]$ (or $b[i]$) as a linear term in the intermediate variable results in independence.

At the algorithm level, the effect of the introduction of two shares in the computation of the KECCAK-f round function is rather simple. The linear part λ can be executed on the two shares separately, roughly doubling the workload. In the nonlinear step mapping χ the computation of a state word according to Equation (1), taking a XOR, AND and NOT instruction, is replaced by the computation of the shares according to Equations (2), taking in total 4 XOR, 4 AND and 2 NOT instructions. The addition of round constants ι and addition of input blocks can be performed on one share only.

As the order of execution is important, it is not sufficient to write a program in C or some other high-level language and compile it. The compiler may optimize away the desired order. An option is to compile and inspect the machine code afterwards, but the method that provides the highest level of assurance is to program in assembly language. It is however not sufficient to check only the sequencing of instructions. In general, the operations on two shares of the same variable are preferably executed in registers *physically isolated* from each other. More particularly, the program shall be such that at any given moment there is no register (or bus) content or transition that is correlated to a native variable. For example, if the number of shares is two and a register containing x_0 is loaded with x_1, the power consumption can depend on the number of bits switched (e.g., on the Hamming weight of $x_0 \oplus x_1$) and it is likely to leak information on x. This can be solved by setting the register to zero in between. Another example is the simultaneous processing of two shares of a variable in different parts of the CPU. As the power consumption at any given moment depends on all processing going on, it depends on both shares and and hence there may be a dependence on the native variable. Clearly, care must be taken when attempting to build side-channel resistant implementations. So if sufficient care is taken, this provide provable resistance against first-order DPA. Higher-order DPA is in principle still possible but as explained in [7, Appendix A] very sensitive to noise and clock jitter. On smart cards, the addition of noise, dummy cycles and clock jitter are typically supported by the platform.

5 Hardware implementation using three-share masking

At first sight masking with two shares may work as well for a dedicated hardware implementation of KECCAK as in software. However, as explained in Section 4, it is crucial that the computation of Equations (2) is performed in a specific order. In dedicated hardware this is hard to achieve due to the occurrence of *glitches* [14,15]: in a combinatorial circuit computation is typically not monotonous but intermediate signals may switch several times per clock cycle. Hence in combinatorial circuits for implementing the formulas of Equation (2), there may well be an intermediate signal that leaks information on one of the native variable x_{i+1} or x_{i+2}. Hence, due to the occurrence of glitches, two-share masking cannot provide provable protection against first-order CPA.

A solution to this problem was proposed in [16]. We refer to [16,17] for an in-depth treatment and limit ourselves here to the explanation of the basic concept. The simple but powerful idea is to take as many shares as needed such that in any computation at least one of the shares is not taken as input. In this way, all intermediate variables are guaranteed to be independent from native variables for the same reason that the one-time pad offers perfect secrecy. For linear functions two shares are sufficient to realize this. For nonlinear functions, the required number of shares depends on the particular function.

As can be read in [16,17], for most cryptographic algorithms, the mere number of shares required makes the application of this technique very expensive. Fortunately, χ with its simple structure is well suited for applying this technique and only three shares are required. This suitability is quite unique among the SHA-3 candidates.

If we denote the shares by a, b and c, a possible computation of the three shares is given by:

$$\begin{aligned} a_i &\leftarrow b_i + (b_{i+1} + 1)b_{i+2} + b_{i+1}c_{i+2} + c_{i+1}b_{i+2} \\ b_i &\leftarrow c_i + (c_{i+1} + 1)c_{i+2} + c_{i+1}a_{i+2} + a_{i+1}c_{i+2} \\ c_i &\leftarrow a_i + (a_{i+1} + 1)a_{i+2} + a_{i+1}b_{i+2} + b_{i+1}a_{i+2} \ . \end{aligned} \tag{3}$$

Clearly, the computation of each share takes as input only components of the other two shares and it provides provable security against first-order CPA, even in the presence of glitches. Still, if the three shares are processed simultaneously, the power consumption depends on the three shares being processed and a dependence on the native variable may exist. If a native bit is 0, either all its three shares are 0 or two of the three shares are 1. If the native bit is 1, either all its three shares are 1 or one of the three shares is 1. The mean number of shares equal to 1 is 1.5 in both cases but they have different distributions. If the power consumption depends on the value of these bits, this may leak information on the native bit. However, as shown in Section 6, the possibility of exploiting this strongly depends on the noise level.

The three lines of Equation (3) are equivalent and only differ in the input shares and output share. We denote the computation of the nonlinear step χ' resulting in a share given the two other shares by, e.g., $a \leftarrow \chi'(b, c)$.

In the following subsections we present two architectures that make use of three shares. Other architectures can be derived based on different partitioning or sequences of the computational steps composing the round. For instance a low area coprocessor, as those presented in [4], can be protected using the secret sharing techniques.

5.1 One-cycle round architecture

A first proposal is to adopt an implementation computing one round in one clock cycle. This architecture is depicted in Figure 1.

Note that the round of Keccak-f is not so different, in terms of complexity, from the first stage of the Noekeon S-box implemented in [16].

Before processing, the three shares a, b and c are generated from a random source. As the initial state of Keccak should be set equal to zero implying $a + b + c = 0$, the shares a and b can be generated randomly and c computed as $c = a + b$. The hardware for generating the three shares is out of the scope of our study, we just consider them as input to the core.

The combinatorial logic implements the round function and input data block absorbing. It has a layered structure. In a first (linear) layer, the absorbing of the input data block, DIN, is implemented by adding it to one of the shares and then λ is applied to the three shares by three separate combinatorial blocks. In a second layer, the nonlinear step mapping χ is computed on the output of the first layer according to Equations (3) by three separate combinatorial blocks

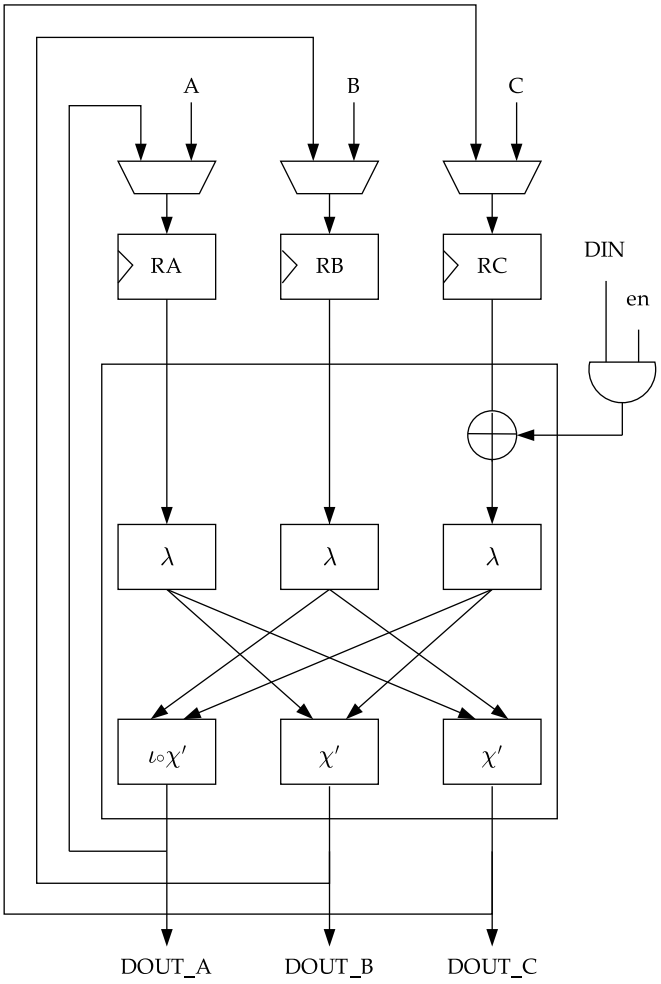

Fig. 1. Protected architecture computing one round in one clock cycle.

implementing χ'. Each block takes as input two shares and generates one share. The blocks only differ in the presence of ι in the leftmost block as ι only needs to be added to a single share.

We can estimate the cost of the secret sharing technique in terms of silicon area by comparing this architecture with our unprotected one-cycle round architecture in [4], based on Figure 1 and Equations (3). The number of registers required for storing the state is three times larger than the unprotected version. The cost of the linear part is three times larger as well. Regarding the nonlinear part, we have three blocks χ' instead of one χ and the cost of every χ' is also larger than that of χ. While χ requires basically a AND gate, a NOT and a XOR for each bit of the state, χ' requires three AND gates, one NOT and three XOR for a single share. So roughly, the protected nonlinear part is expected to be nine times larger than the unprotected χ.

5.2 Three-cycle round architecture

In this section a second architecture is presented, which reduces the amount of silicon at the cost of performance. The architecture is depicted in Figure 2. Since the round logic is composed of three equal blocks for λ and three equal blocks for χ', we instantiate only one block of each and try to use them as much as possible.

Instead of three registers this architecture requires four registers, some multiplexing and a careful schedule. The schedule has some similarity to pipelining techniques.

For explaining the schedule we refer to Table 1. We use $\lambda(R_0)$ to denote the application of λ to register R_0, and $\chi'(R_1, R_2)$ to denote the application of χ' using registers R_1 and R_2 as inputs. The three initial values of the shares are indicated as A, B and C. The values after the first round by A', B' and C', after the second round by A'', etc. At clock cycle zero the registers do not yet contain any relevant data, as indicated with a $-$. Instead of loading the three shares in parallel, as done in the previous architecture, one share per clock cycle is loaded into register R_0 during the three initial clock cycle.

The content of R_1 is in general the result of λ applied to R_0, with the DIN enabled and XORed at the input of $\lambda(R_0)$ before an initial KECCAK-f round.

The content of R_0 is the result of χ' applied to either the couple R_2 and R_1 or R_2 and R_3. While in the one-cycle round architecture ι was applied only to one of the three shares, here it is applied to all of the three shares. This does not change the result of the computation and simplifies the control logic.

The registers R_2 and R_3 are used for maintaining the values required for the computation of χ'. In the second clock cycle B is loaded into R_0, and R_1 receives $\lambda(A)$. In the third clock cycle C is loaded into R_0, R_1 receives $\lambda(A)$ and $\lambda(A)$ is moved from R_1 to R_2. In the fourth clock cycle no more shares need to be loaded. Instead R_0 receives χ' applied to the content of R_1 and R_2, which means $\lambda(A)$ and $\lambda(B)$. It follows that R_0 now contains the share C' after the first round. R_1 receives $\lambda(C)$, $\lambda(B)$ is moved from R_1 to R_2 and $\lambda(A)$ is moved from R_2 to R_3. In the fifth clock cycle R_0 receives χ' applied to the content of R_1 and R_2, being $\lambda(B)$ and $\lambda(C)$ and hence contains the share A after the first round. R_1 receives $\lambda(C')$, $\lambda(C)$ is moved from R_1 to R_2, while $\lambda(A)$ remains in R_3. Since shares A' and C' as input of the second round have been already computed, in the next clock cycle share B' must be computed, requiring $\lambda(A)$ and $\lambda(C)$, and they are in R_1 and R_3 respectively before clock cycle five. Thus we have described also what will be computed in the next clock cycle and this is basically the end of a round. Starting from the next clock cycle there will be a loop of three clock cycles where the alternation of data contained in the registers are those for computing a round.

Using this circuit, a KECCAK computation takes 3 initial cycles plus 24 cycles per execution of KECCAK-f.

clock cycle	Formula				Contents			
	R_0	R_1	R_2	R_3	R_0	R_1	R_2	R_3
0	-	-	-	-	-	-	-	-
1	input	-	-	-	A	-	-	-
2	input	$\lambda(R_0 \oplus DIN)$	-	-	B	$\lambda(A \oplus DIN)$	-	-
3	input	$\lambda(R_0)$	R_1	-	C	$\lambda(B)$	$\lambda(A \oplus DIN)$	-
4	$\chi'(R_2, R_1)$	$\lambda(R_0)$	R_1	R_2	C'	$\lambda(C)$	$\lambda(B)$	$\lambda(A \oplus DIN)$
5	$\chi'(R_2, R_1)$	$\lambda(R_0)$	R_1	R_3	A'	$\lambda(C')$	$\lambda(C)$	$\lambda(A)$
6	$\chi'(R_2, R_3)$	$\lambda(R_0)$	R_1	R_2	B'	$\lambda(A')$	$\lambda(C')$	$\lambda(C)$
7	$\chi'(R_2, R_1)$	$\lambda(R_0)$	R_1	R_2	B''	$\lambda(B')$	$\lambda(A')$	$\lambda(C')$
8	$\chi'(R_2, R_1)$	$\lambda(R_0)$	R_1	R_3	C''	$\lambda(B'')$	$\lambda(B')$	$\lambda(C')$
9	$\chi'(R_2, R_3)$	$\lambda(R_0)$	R_1	R_2	A''	$\lambda(C'')$	$\lambda(B'')$	$\lambda(B')$
10	$\chi'(R_2, R_1)$	$\lambda(R_0)$	R_1	R_2	A'''	$\lambda(A'')$	$\lambda(C'')$	$\lambda(B'')$
11	$\chi'(R_2, R_1)$	$\lambda(R_0)$	R_1	R_3	B'''	$\lambda(A''')$	$\lambda(A'')$	$\lambda(B'')$
12	$\chi'(R_2, R_3)$	$\lambda(R_0)$	R_1	R_3	C'''	$\lambda(B''')$	$\lambda(A''')$	$\lambda(A'')$

Table 1. The content of the registers during the computation.

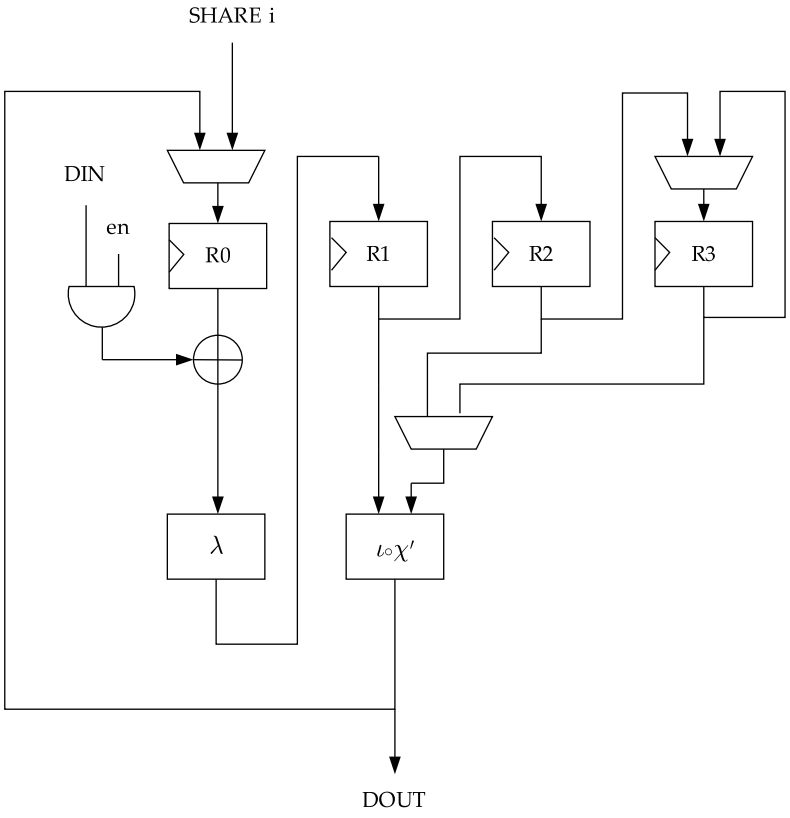

Fig. 2. Protected architecture computing one round in three clock cycles.

5.3 Synthesis results

We have implemented both architectures in VHDL and synthesized it for understanding the maximum frequency and silicon area demand. We have used the same technology library adopted for the unprotected implementation reported in [4], a 130 nm general purpose library from STMicroelectronics, and the Synopsys Design Compiler.

Table 2 summarizes the gate count and performance numbers of the different implementations, together with the numbers for the one-cycle round unprotected architecture described in [4]. It gives the gate count split up over the I/O buffer, the linear part and nonlinear part of the round logic, the state registers and finally multiplexors and input logic (together under the heading MUX). In the three rightmost columns is the total gate count, the maximum frequency and the throughput assuming a bitrate of 1024 bits.

The gate count figures are in line with the estimations made in Section 5.1. All implementations have an I/O buffer of 9 Kilo gate equivalent (KGE) for connecting the core to a system bus as described in [4]. This allows to load the I/O buffer, 64-bit per clock cycle, simultaneously with the application of KECCAK-f on the previous input block.

In the three-cycle round architecture the computation of one round is executed in 3 clock cycles and the initialization of the state requires also 3 clock cycles. Thanks to the very short critical path, the fast variant of the three-cycle round architecture can reach 714 MHz, still resulting in a very competitive speed of 10 Gbit/s. If we compare the one-cycle round and the three-cycle round

architectures both running at 500Mhz, we can see that the three-cycle requires 40% less silicon area at a cost of a throughput reduction by a factor 3.

It is interesting to note that the strategy adopted in the design of the function allows implementing this countermeasure with a cost consisting only of silicon area with almost no penalty in terms of throughput: it is reduced only by 5%, from 22.4 Gbit/s. to 21.3 Gbit/s. when using a rate of 1024 bits.

Core	I/O	Round logic		State	MUX	Total size	Frequency	Throughput
		λ	χ	registers				($r = 1024$)
	KGE	KGE	KGE	KGE	KGE	KGE	MHz	Gbit/s.
Unprotected one-cycle [4]	9	11	8	9	11	48	526	22.4
One-cycle (fast)	9	33	87	27	27	183	500	21.3
One-cycle (compact)	9	22	50	27	19	127	200	8.5
Three-cycle (fast)	9	13	33	36	24	115	714	10.1
Three-cycle (medium)	9	11	26	36	24	106	500	7.1
Three-cycle (compact)	9	8	18	36	24	95	200	2.8

Table 2. Performance and gate count of the different implementations

6 Computing in parallel or sequentially?

The use of three shares computed at different times give a provable resistance against first-order DPA, but not against DPA of higher order. Higher-order DPA involves taking measurements corresponding to the computation of the different shares, and this task is more complex due to its higher sensitivity to noise than first-order DPA [7, Appendix A].

In the architectures presented in the previous sections, several computations take place in parallel and the power consumption at a given time depends on all these computations. For this reason, these architectures are not provably resistant against first-order DPA. For instance, in [17], the authors analyzed an implementation with three shares working in parallel. Storing the three shares in a register causes a power consumption to depend on the three shares simultaneously. In the absence of noise, the distribution of the consumption depends on the native variable being stored and hence is vulnerable to MIA. In this section, we explain that the introduction of noise has a much stronger impact on the feasibility of this attack for a two-share or three-share implementation than for an unmasked one, and that the qualitative difference is similar to the one between first-order and higher-order DPA. Furthermore, we argue that a masked implementation has per construction a higher noise level than an unmasked one.

With three shares, a native bit equal to 0 (resp. 1) can be represented as 000, 011, 101 or 110 (resp. 001, 010, 101 or 111). If the three shares are processed simultaneously, the power consumption can leak the Hamming weight of the shares, which means 0 or 2 for a native bit 0, or 1 or 3 for a native bit 1. Clearly, these two distributions are different and can be distinguished.

We start the discussion by constructing a simple model where the power consumption is equal to the Hamming weight plus an additive Gaussian noise of standard deviation σ, expressed in the same unit as the impact of the Hamming weight on the power consumption. In this model, the distribution of the power consumption for three shares is as in Figure 3(c), with $\sigma = 0.2$. Similarly, we can look at an unmasked implementation, where the power consumption is 0 or 1 plus the Gaussian noise, and at masking with two shares. In this last case, the Hamming weight is 0 or 2

for a native bit 0 (represented as 00 or 11) or 1 for a native bit 1 (represented as 01 or 10). The two cases can be found in Figures 3(a) and 3(b), respectively.

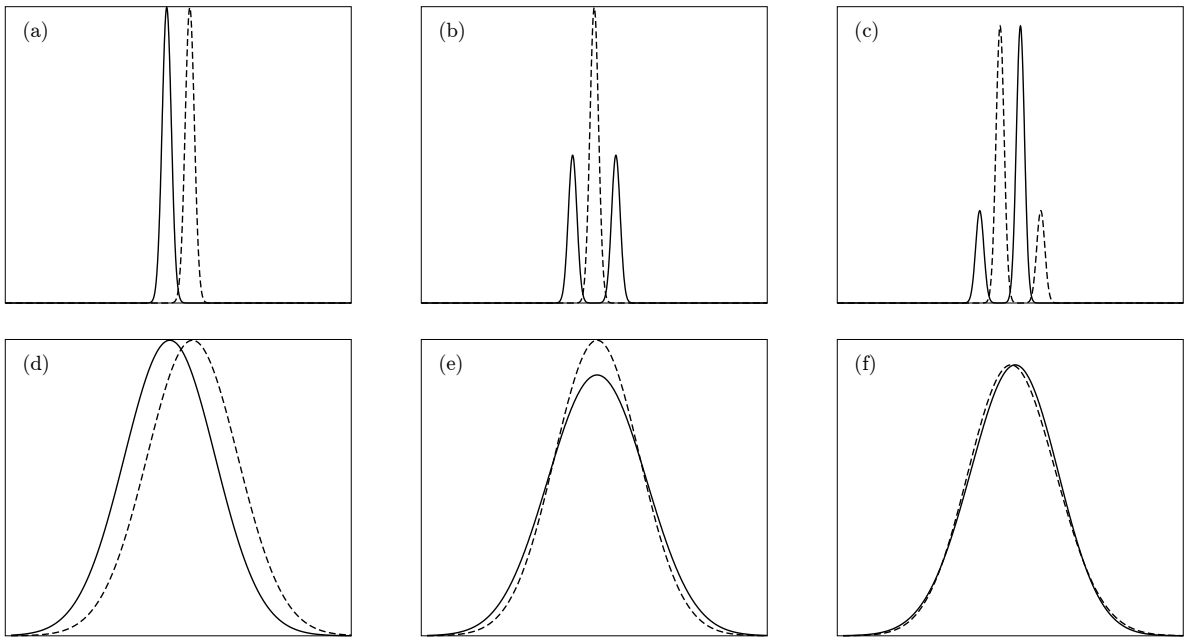

Fig. 3. Distribution of the power consumption for a simple model. The solid line shows the distribution for a bit with native value 0 and the dashed line for a bit 1. Sub-figures (a), (b) and (c) show the case of one, two and three shares, respectively, with a noise level of $\sigma = 0.2$. Sub-figures (d), (e) and (f) follow the same model with $\sigma = 2$.

Following this model, we compute the number of samples that are needed to distinguish between the distribution for a native bit 0 and the distribution for a native bit 1. We follow the same reasoning as in [7, Appendix A]. The number z of samples needed to distinguish one distribution over the other is inversely proportional to the Kullback-Leibler divergence between the two distributions [13], $z = 1/D(f\|g)$ with

$$D(f\|g) = \int f(x)(\log(f(x)) - \log(g(x))dx.$$

In this model, the scaling of z as a function of σ is different for one, two or three shares. For one share, $z \sim 2\sigma^2$ samples are needed to distinguish a native bit 0 from a native bit 1 from unmasked values, whereas about z^2 samples are needed for the same noise level when two shares are used, and this number grows to about z^3 samples for three-share masking. Hence, the difference between the one-share and two-share and three-share implementation is qualitatively the same as the one between first-order and second-order and third-order DPA.

The real-world behavior is likely to differ from this simple model. Nevertheless, we expect a significantly higher sensitivity to noise for three shares than for one. Qualitatively, the three pairs of distributions are different. For one share, the mean is different for native bits 0 and 1. For two shares, the two distributions have the same mean but a different variance. For three shares, the two

distributions have the same mean and variance; they differ only starting from their third moment. Figures 3(d), 3(e) and 3(f) illustrate this with the simple model and a higher noise level $\sigma = 2$.

So far, we have assumed that the three levels of masking are subject to the same noise level σ. However, the masking by itself introduces noise, as $m - 1$ shares are randomly and independently chosen for every execution of the algorithm. The very dependence of the processing of a bit in the power consumption that an attacker exploits turns against her, as it becomes an additional source of noise due the randomization. For instance, in the one-cycle-one-round implementation of KECCAK-f[1600] with three shares, the noise due to the Hamming weight of 3200 random bits must be compared to a small number of unmasked bit values that a differential power analysis attempts at recovering.

7 Power analysis simulations

In this section we report on some preliminary power analysis simulations we did of the 3-share one-cycle architecture compared with the plain fast core architecture specified in [4].

We have adopted the following approach, inspired by Regazzoni et al. [18]. The approach is based on standard CAD tools, used in different steps for estimating the power consumption of a hardware design without the real fabrication of the chip.

The first two steps, RTL simulation and silicon synthesis, are those presented in section 5.3 and are used for generating the gate level model of the architecture.

The next step is to use a tool, such as Model Sim, to simulate the gate level with a set of inputs and to extract so-called value change dump (VCD) files. Such a file contains all the switching activity of a design during a given simulation. The VCD file and the power characterization of the technology library are used as input by PrimeTime from Synopsys for creating the power consumption trace. Finally the power consumption trace is imported into Matlab for elaborating it using OpenSCA [8]. Compared to what has been presented in [18] we have limited ourselves to the simulation at gate level. The next level would be to perform the so called place-and-route and simulate at this lower level.

Note that the simulation is noise-free and sampling of the input by the registers is perfectly synchronized.

We have applied gate-level simulations to two architectures: to the KECCAK plain fast core specified in [4] and to the three-share one-cycle architecture as described in Section 5.1. Each core has been simulated in 10,000 executions. Each execution is a sequence of two KECCAK-f permutations, the first for absorbing the secret key, while the second for absorbing a random (and known) message. In the case of the three-share architecture, the three shares are randomized at the beginning of every execution, before absorbing the secret key. The target of an attacker is to recover the key or the value of the state after the first permutation, knowing the random messages and the power consumption of each execution.

First we analyze the simulated power consumptions of the plain architecture for understanding where are the leakage points exploitable by the attacker. Then we analyze the same leakage points in the three-share architecture to check if the secret-sharing technique has removed the vulnerability.

More specifically, we compute for each time step in the execution the correlation coefficient between the simulated power consumption on the one hand and a function of the native values in the permutation on the other. The function of the native values can either be the Hamming weight of the output of a round, or the Hamming distance between the input of a round and its output.

Note that an attacker does not know the key and cannot predict the value of the native data. Here, we rather wish to characterize which native quantity is correlated to the power consumption.

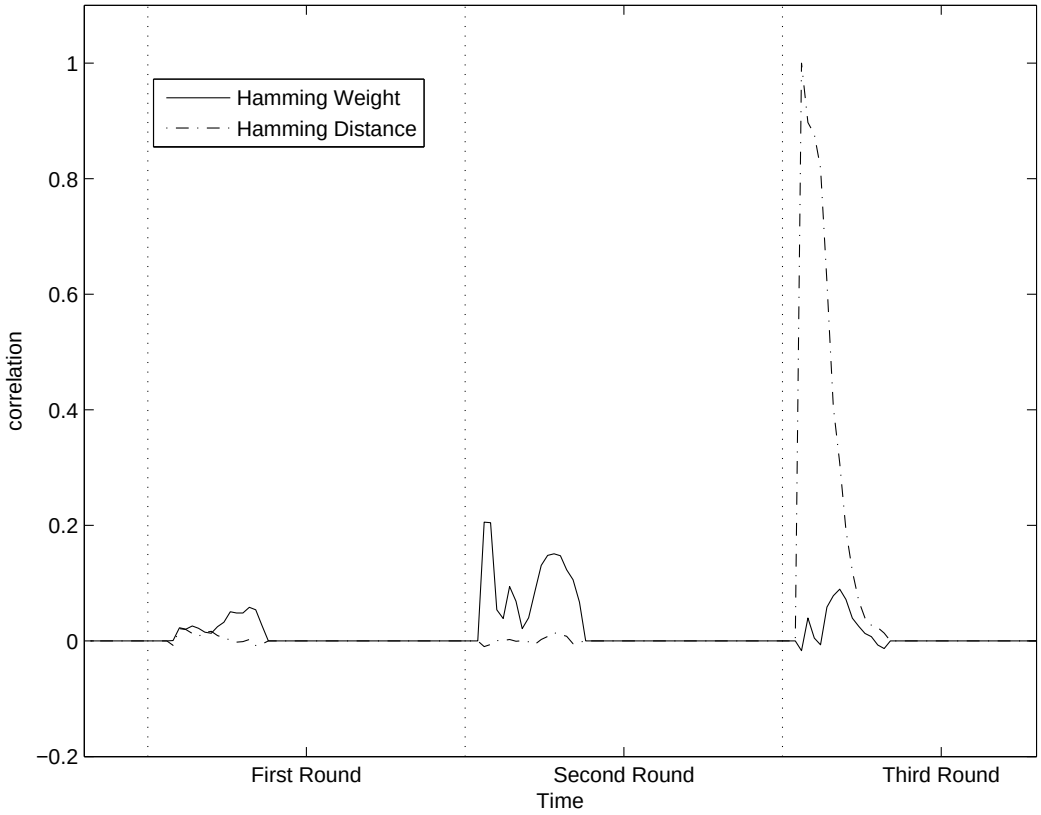

Fig. 4. Correlation between the power consumption and either the Hamming weight of the state at the output of the first round or the Hamming distance between the states at the input and output of the second round.

This knowledge can help defining an appropriate selection function depending only a small number of key bits, whose value can be guessed.

As illustrated by Figure 4, our experiments show that at some time steps in the execution the simulated power consumption is proportional (and thus perfectly correlated) to the Hamming distance between the states at the input of two consecutive rounds. This implies that the simulated power consumption is proportional to the switching activity of the register. Together with Hamming distance over the second round, Figure 4 depicts also the correlation between the power consumption and the Hamming weight of the output state of the first round. The leakage is not as evident as the case of Hamming distance but it is still exploitable.

The same experiments have been performed on the three-share architecture. As expected in this model, the simulation shows no significant correlation with the native values.

8 Conclusions

In this paper we have applied the secret sharing method for protecting Keccak software and hardware implementations against power analysis.

References

1. G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, KECCAK *specifications, version 2*, NIST SHA-3 Submission, September 2009, `http://keccak.noekeon.org/`.
2. ———, *Note on side-channel attacks and their countermeasures*, Comment on the NIST Hash Competition Forum, May 2009, `http://keccak.noekeon.org/NoteSideChannelAttacks.pdf`.
3. ———, *Duplexing the sponge: single-pass authenticated encryption and other applications*, Second SHA-3 candidate conference, August 2010.
4. ———, KECCAK *sponge function family main document*, NIST SHA-3 Submission (updated), June 2010, `http://keccak.noekeon.org/`.
5. ———, *Sponge-based pseudo-random number generators*, CHES (S. Mangard and F.-X. Standaert, eds.), Lecture Notes in Computer Science, vol. 6225, Springer, August 2010, pp. 33–47.
6. E. Brier, C. Clavier, and F. Olivier, *Correlation power analysis with a leakage model*, CHES (M. Joye and J.-J. Quisquater, eds.), Lecture Notes in Computer Science, vol. 3156, Springer, 2004, pp. 16–29.
7. J. Daemen, M. Peeters, and G. Van Assche, *Bitslice ciphers and power analysis attacks*, Fast Software Encryption 2000 (B. Schneier, ed.), Lecture Notes in Computer Science, vol. 1978, Springer, 2000, pp. 134–149.
8. E. Oswald et al., *OpenSCA, an open source toolbox for Matlab*, University of Bristol, Department of Computer Science, 2008, `http://www.cs.bris.ac.uk/home/eoswald/opensca.html`.
9. G. Gielen and J. Figueras (eds.), *2004 design, automation and test in Europe conference and exposition (DATE 2004), 16-20 February 2004, Paris, France*, IEEE Computer Society, 2004.
10. B. Gierlichs, L. Batina, P. Tuyls, and B. Preneel, *Mutual information analysis*, CHES (E. Oswald and P. Rohatgi, eds.), Lecture Notes in Computer Science, vol. 5154, Springer, 2008, pp. 426–442.
11. S. Guilley, P. Hoogvorst, Y. Mathieu, R. Pacalet, and J. Provost, *CMOS structures suitable for secured hardware*, in Gielen and Figueras [9], pp. 1414–1415.
12. P. C. Kocher, J. Jaffe, and B. Jun, *Differential power analysis*, Advances in Cryptology – Crypto '99 (M. Wiener, ed.), Lecture Notes in Computer Science, vol. 1666, Springer, 1999, pp. 388–397.
13. S. Kullback and R. A. Leibler, *On information and sufficiency*, Ann. Math. Statist. **22** (1951), no. 1, 79–86, `http://projecteuclid.org/euclid.aoms/1177729694`.
14. S. Mangard, T. Popp, and B. M. Gammel, *Side-channel leakage of masked CMOS gates*, CT-RSA (A. Menezes, ed.), Lecture Notes in Computer Science, vol. 3376, Springer, 2005, pp. 351–365.
15. S. Mangard, N. Pramstaller, and E. Oswald, *Successfully attacking masked AES hardware implementations*, CHES (J.R. Rao and B. Sunar, eds.), Lecture Notes in Computer Science, vol. 3659, Springer, 2005, pp. 157–171.
16. S. Nikova, V. Rijmen, and M. Schläffer, *Secure hardware implementation of nonlinear functions in the presence of glitches*, ICISC (P. J. Lee and J. H. Cheon, eds.), Lecture Notes in Computer Science, vol. 5461, Springer, 2008, pp. 218–234.
17. ———, *Secure hardware implementation of nonlinear functions in the presence of glitches*, Journal of Cryptology, 2010, to appear.
18. F. Regazzoni, A. Cevrero, F. Standaert, S. Badel, T. Kluter, P. Brisk, Y. Leblebici, and P. Ienne, *A design flow and evaluation framework for DPA-resistant instruction set extensions*, CHES (C. Clavier and K. Gaj, eds.), Lecture Notes in Computer Science, vol. 5747, Springer, 2009, pp. 205–219.
19. E. Oswald S. Mangard and T. Popp, *Power analysis attacks — revealing the secrets of smartcards*, Springer-Verlag, 2007.
20. K. Tiri and I. Verbauwhede, *A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation*, in Gielen and Figueras [9], pp. 246–251.