Guards in Action: First-Order SCA Secure Implementations of Ketje without Additional Randomness

Victor Arribas
KU Leuven, imec-COSIC, Belgium
varribas@esat.kuleuven.be

Svetla Nikova
KU Leuven, imec-COSIC, Belgium
svetla.nikova@esat.kuleuven.be

Vincent Rijmen
KU Leuven, imec-COSIC, Belgium
vincent.rijmen@esat.kuleuven.be

***Abstract*—Recently the CAESAR competition has announced several finalists among the submitted authenticated encryption algorithms, after an open selection process during the last 5 years. Applications using these algorithms are rapidly increasing today. Devices implementing these applications are enormously susceptible to physical attacks, which are able to retrieve secret data through side-channel information such as the power consumption or the electromagnetic radiations. In this work we present a Side-Channel Analysis resistant hardware implementation of the whole family of authenticated encryption schemes KETJE. By changing just one parameter, any of the KETJE designs can be obtained, and tailored for different applications, either lightweight or high throughput.**

We introduce a new protected KECCAK implementation, as well as unprotected and protected KETJE implementations, which allow both encryption and decryption modes in the same module. In order to secure these implementations we make use of the masking scheme known as Threshold Implementations and complement it with the technique of "Changing of the Guards", achieving a first-order Side-Channel Analysis protected implementation with zero extra randomness needed. This way, no dedicated PRNG needs to be additionally implemented, avoiding issues such as the security of the PRNG itself or the quality of the randomness.

***Index Terms*—Authenticated Encryption, KETJE, SHA-3, Side-Channel Analysis, Threshold Implementations, Changing of the guards.**

I. INTRODUCTION

The use of IoT devices is growing enormously, with predictions of an increase of 250% in their use in the next three years. This makes hardware security an important issue, needed to protect devices implementing cryptographic algorithms. Several attacks have proliferated in the literature in the past few years, Side-Channel Analysis (SCA) being one of the most significant given its reduced cost and its ease of use. They exploit computation time, power consumption or electromagnetic radiation to extract sensitive information such as cryptographic keys. A distinct and very powerful SCA is Differential Power Analysis (DPA) [1].

Multiple techniques have been proposed to secure the hardware, as can be hiding or masking [2] among others. Masking has caught most of the attention in the literature, with several masking schemes proposed [3]–[8]. The aim of masking is to randomize the intermediate variables, making the power consumption independent of their value. The first works of Ishai *et al.* [3] and Trichina [4] were proven flawed, since they did not provide security in the presence of glitches. The work of Nikova *et al.* [5], known as Threshold Implementations (TI), was the first to provide first-order SCA security in the presence of glitches and later extended to higher-order by Bilgin *et al.* [6]. Reparaz *et al.* [7], in Consolidated Masking Schemes (CMS), reduce the area overload to achieve the same order of security at the cost of introducing extra randomness, and provide security against multivariate attacks (higher-order attacks where wires in different time frames are probed). Subsequently, Gross *et al.* [8], in Domain Oriented Masking (DOM), propose a reduction in the randomness with respect to CMS.

Our contribution. In this work we present a new protected KECCAK implementation with new trade-offs: slightly bigger area compared to the smallest state-of-the-art parallel implementations, but two or three times lower latency. Additionally, we introduce the implementation of the whole KETJE family, both unprotected and first-order SCA protected with zero extra randomness added. Furthermore, we extend these implementations to allow encryption and decryption in the same module, both protected and unprotected. We deploy the first-order protected KECCAK in FPGA and test it, proving the implementation secure for up to 100 million traces. We expand this implementation to get the secure KETJE implementation that inherits the previously proven security, since the expansion only involves linear operations and we use a TI sharing scheme.

We begin by summarizing the specifications of KETJE in Section II, then we discuss important concepts and methodologies applied to secure our implementations in Section III. Details of the protected implementations are given in Section IV, followed by the results and evaluation details in Sections V and VI respectively.

II. KETJE

KETJE [9] is a CAESAR (Competition for Authenticated Encryption: Security, Applicability and Robustness) third round candidate. Although recent updates on the CAESAR competition do not include KETJE as a finalist, we still believe it is a highly valuable algorithm. It is based on the SHA-3 standard and it is very suitable for side-channel resistant hardware implementations. KETJE is a set of four authentication encryption schemes that allow messages with associated data.

They target memory-constrained devices and strongly rely on nonce uniqueness for security. Ketje is built following a layered design, where KECCAK [10] is the underlying permutation, called by the construction MONKEYDUPLEX, which at the same time is instantiated by the mode MONKEYWRAP.

A. KECCAK

KECCAK is a family of hash functions that were standardized by NIST, becoming SHA-3. It runs the permutation KECCAK-$f[b]$ where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ defines different state sizes. KETJE uses a round reduced permutation KECCAK-$p[n_r, b]$, which runs the last n_r rounds of KECCAK-$f[b]$. KECCAK works on a three dimensional state (Fig. 1), where a round consists on five operations: $R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$. We refer to [10] for further details on the operations.

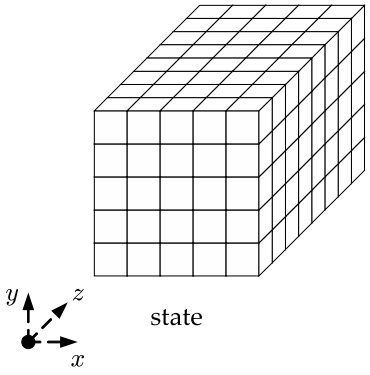

Fig. 1: Three-dimensional KECCAK state [10]

B. MONKEYDUPLEX

MONKEYDUPLEX calls a round-reduced version of KECCAK-f where different types of calls are supported invoking different numbers of rounds. The performance of the functions can be optimized by reducing the number of rounds, at the expense of requiring nonce uniqueness for the scheme to provide security against key retrieval. MONKEYDUPLEX follows the structure of a duplex construction, where there is an absorption phase and a squeezing phase alternating. The absorption phase introduces new bits to the state, while the squeezing phase extracts bits from the state. Fig. 2 depicts this idea. The function takes as input (absorbs) a binary string of any length σ and returns (squeezes) a binary string of the requested length Z.

Four parameters (r, n_{start}, n_{step}, n_{stride}) determine the efficiency and security strength of the module. Two types of calls are supported by MONKEYDUPLEX:

- D.start: at the beginning the string $I = K||N$ that concatenates the Key (K) and the Nonce (N) is introduced in the function after a simple padding is applied, initializing the state. Then, the function f runs for n_{start} rounds, $f[n_{\text{start}}]$.
- D.step or D.stride: the construction absorbs a data block σ , extended with a multi-rate padding until the string length is equal to the $rate$ r. Then, either $f[n_{\text{step}}]$ or $f[n_{\text{stride}}]$ is applied to the state f, and afterwards part of this state is extracted (Z).

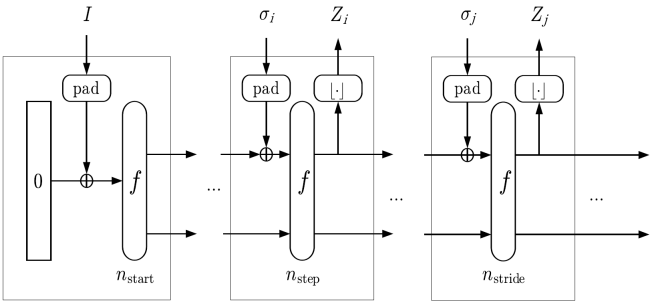

Fig. 2: The MONKEYDUPLEX construction [9]

This construction can be used for mainly three different applications: first, the simplest one, as a stream cipher; second, as a reseedable (fresh seed every block) pseudorandom bit sequence generator; third, and the most relevant one, as authenticated encryption.

Finally, it is important to stress that both uniqueness and secrecy of I are crucial to comply with the security claims. In addition to this, multiple instances of MONKEYDUPLEX with the same input I are not allowed, since, otherwise, an attacker could retrieve the full state easily by observing differences in the output.

C. MONKEYWRAP

The authentication process takes as input a header A or Associated Data, a data body B or Plaintext, and returns a Ciphertext C and a Tag T. The encryption process is known as *wrapping* and then the decryption process is known as *unwrapping*. The functionality of the encryption is outlined below and illustrated in Fig. 3:

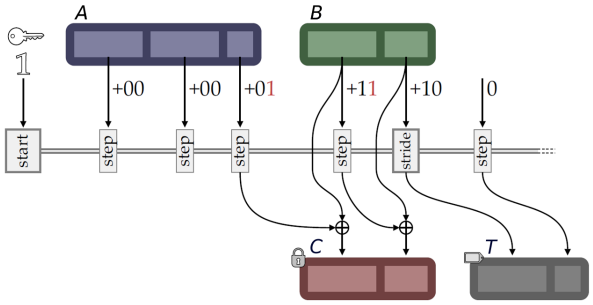

Fig. 3: The MONKEYWRAP construction [9]

1) The state is initialized with Key and Nonce executing D.start(keypack($K, |K| + 16)||N$) where the keypack is a way of encoding the secret key given a string input.
2) The associated data is processed block by block (A_i) by executing D.step($A_i||00$), and D.step($A_i||01$) for the last block of A. The block size ($|A_i|$) of the mode is given by ρ. The last output is XORed with the first plaintext block to get the first ciphertext block.
3) The plaintext is similarly processed with D.step($B_i||11$) and the output of every step is XORed with the next

block of the plaintext to get a ciphertext block. The last block of B is treated with D.step$(B_i||10)$ to produce the first tag block.

4) Subsequent tag blocks are produced by executing D.step(0) until the required T length is achieved.

The decryption is essentially identical to the encryption except for few differences: the scheme receives as input A, C and T, where the same procedure is followed, using the ciphertext data instead of the plaintext. The new tag T' is calculated out of the plaintext blocks decrypted, and finally the entire plaintext B is output only if $T = T'$.

MONKEYWRAP supports *sessions*, allowing the encryption of several messages (with the respective associated data) with the same input I. The state is never reset or reinitialized during the same session, so that the function keeps running with the previously updated state. This means that intermediate tags are produced at the end of every encryption, authenticating the current encryption together with the previous ones.

The KETJE specifications allow four different authenticated encryption schemes, namely: KETJE JR, KETJE SR, KETJE MINOR and KETJE MAJOR. They are characterized by the underlying permutation and their block size (ρ). They can be used in either lightweight applications or higher-throughput implementations. Tab. I summarizes the different possibilities:

TABLE I: Four KETJE authenticated encryption schemes [9]

Name	f	ρ	Main use
KETJE JR	KECCAK-p*[200]	16	lightweight
KETJE SR	KECCAK-p*[400]	32	lightweight
KETJE MINOR	KECCAK-p*[800]	128	lightweight
KETJE MAJOR	KECCAK-p*[1600]	256	high performance

Important features of the KETJE schemes to take into account:

- As already mentioned above, all but KETJE MAJOR are lightweight.
- The greater block size of KETJE MINOR and KETJE MAJOR allows the output of 128-bit tags without extra costs.
- They are very well suitable for side-channel countermeasures and a perfect fit for secure messaging with secured chips, for instance, smart cards.

Another important distinguisher is the fact that KETJE supports sessions, meaning that a sequence of messages can be authenticated rather than a single one. This offers an easy way of sending successive commands while preventing the interference of an attacker.

III. SECURING THE HARDWARE

Hereunder we provide an overview of the concepts needed and used throughout the paper to secure a hardware design, namely: the adversary model, where the adversary capabilities are introduced; Threshold Implementations, where the key concepts to correctly implement this sharing scheme are specified, and finally, the technique known as "Changing of the Guards" to complement the previously mentioned sharing methodology.

A. Adversary model

Along this work we consider the model proposed in [3] known as the d-Probing Model. A single probe that monitors the power consumption (or the electromagnetic radiance) of a single wire. The adversary gains information of all the intermediate values of the wire from the last register to the probe. Similarly, an adversary probing d wires acquires the knowledge of every intermediate value of all d wires. A dth-order security scheme provides security against up to dth-order attacks.

B. Threshold Implementations

Threshold Implementations (TI) was first introduced by Nikova *et al.* [5] for first-order security, and broadened further to higher-order by Bilgin *et al.* [6]. Like multiple other masking schemes, TI is based on secret sharing, where each sensitive, i.e. key dependent, data element x is divided into s pieces ($\mathbf{x} = (x_1, \ldots, x_s)$), such that $x = x_1 \perp \ldots \perp x_s$. We consider Boolean masking, where $\perp$ is exclusive addition $\oplus$, and all s shares are needed to derive x. The aim of this methodology is to provide formal methods to correctly randomize the intermediate variables and hence achieve security in hardware implementations. TI imposes three mandatory properties for a design to be secure:

- *Correctness*: a shared function $\mathbf{f}$ such that $f_i(\mathbf{x}) = Y_i$, where $i = 1, \ldots, s$, is correctly shared if $\sum Y_i = Y = f(x)$.
- *Non-completeness*: a shared function $\mathbf{f}$ is d^{th}-order non-complete if any combination of up to d component functions f_i is independent of at least one input share. Here d is the degree of security.
- *Uniformity*: the outputs of a shared function have to conform a uniform distribution. Note that given a uniform sharing, it is sufficient to preserve the uniformity across operations to achieve univariate security (together with non-completeness).

TI provides security even when the gates behavior is non-ideal, that is, any gate can glitch depending on prior inputs of the same clock cycle before stabilizing without giving any advantage to an attacker thanks to the *non-completeness* property.

The number of shares s define the degree of security. The lower bounds for the number of input and output shares are given in [6], in function of d and the algebraic degree of the function t, following the expressions:

$$\begin{aligned} s_{in} &\geq td + 1, \\ s_{out} &\geq \begin{pmatrix} s_{in} \\ t \end{pmatrix}. \end{aligned} \tag{1}$$

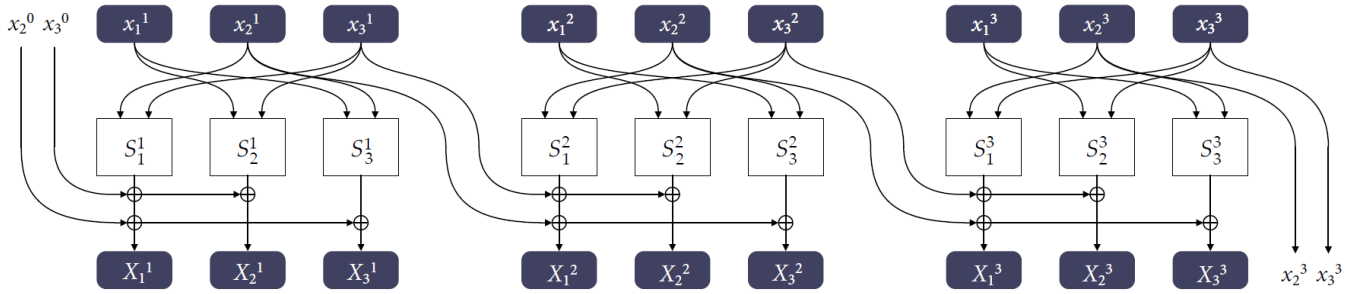

Fig. 4: "Changing of the Guards" structure for three adjacent S-Boxes from a three shares secure implementation [11]

We provide an example of how to protect a simple AND/XOR gate ($z = w \oplus xy$) in Eq. 2, where $(d, t, s_{in}, s_{out}) = (1, 1, 3, 3)$ with TI:

$$\begin{aligned} Z_1 &= w_1 \oplus x_1y_1 \oplus x_1y_2 \oplus x_2y_1, \\ Z_2 &= w_2 \oplus x_2y_2 \oplus x_2y_3 \oplus x_3y_2, \\ Z_3 &= w_3 \oplus x_3y_3 \oplus x_1y_3 \oplus x_3y_1. \end{aligned} \tag{2}$$

We denote an input x with lower case letters, an output Z with upper case letters, and we refer to a specific bit of an S-Box with index t as $x[t]$.

C. Changing of the Guards

In [11] Daemen proposes a method to achieve uniformity in a shared S-Box by just appending few dummy bits to the states, also known as "guards". This method uses these first bits to randomize the first S-Box x^1, then a few of the output bits of this S-Box are similarly used to randomize the second S-Box x^2, and so on until the last S-Box x^j. The last output bits will be the new "guards" of the next round. By feedforwarding these bits and the small extension of the shares, uniformity in the whole S-Box layer is achieved instead of making every single shared S-Box uniform. Fig. 4 illustrates a simple example of a sequence of three shared S-Boxes with three shares each.

This technique has been used previously to protect ASCON and KEYAC [12], and also in a first-order secure AES [13].

IV. IMPLEMENTATIONS

In this section we provide the details of the whole KETJE implementation. The design follows the layered structure given in the specifications of Sect. II. We start from the very bottom, with the underlaying permutation KECCAK. We continue with MONKEYDUPLEX block that calls the permutation. Finally, we implement MONKEYWRAP, which instantiates the MONKEYDUPLEX module, for encryption, and further extended to support encryption/decryption together. The implementation is made generic, where, by just varying the width of the permutation (b), the four instances of KETJE can be obtained.

A. KECCAK-p*

a) Twisted permutations: In KETJE v2, the permutation is modified to KECCAK-p* $= \pi \circ$ KECCAK-$p \circ \pi^{-1}$. The purpose of this is to more effectively re-order the state bits.

b) Design choice: There are several protected implementations of KECCAK proposed in the literature [14]–[16]. The first two use TI to secure their implementations: the first one was shown to use a non-uniform sharing, which was fixed by Bilgin *et al.* in the second one by using a four shares scheme instead of three to achieve uniformity. Gross *et al.* in [16] propose several serial and parallel implementations using DOM to secure their designs. KETJE is designed to operate with a parallel implementation to get the most optimal implementation performance/area wise. We discard implementations using DOM since a single round needs two or three cycles to be computed in the parallel implementations. We go with TI to get the best latency, and we apply the "Changing of the Guards" method explained before to get a uniform sharing with three shares. With this method we are able to optimize area with respect to the four shares version, and no extra randomness is needed. Thus, no dedicated PRNG is needed to feed this module.

Securing linear operations is trivial, where just mere repetition is needed to get correctness. Hence, we focus on how to secure the non-linear operation χ and how the guards are implemented. Eqn. (3) shows the equations followed in our first-order implementation with tree shares, similarly given in [11].

It is difficult to compare our implementations area-wise with previous works given that different synthesis libraries were used. However, latency can be compared precisely. Since no registers are needed to secure the non-linear operation, the corresponding KECCAK-f implementation takes respectively two times and three times less cycles than the PARALLEL double clocked and the PARALLEL pipelined implementations from [16]. On the other hand, our implementation is between 6% to 28% (depending on the library) larger in area than the previous works.

B. MONKEYDUPLEX

MONKEYDUPLEX implements three functions in its framework: D.start, D.step and D.stride. All of them call the same permutation, but for different numbers of rounds: $n_{\text{start}} = 12$, $n_{\text{step}} = 1$, $n_{\text{stride}} = 6$. The first one gets as inputs a string forged with the Key and the Nonce, while the others get a

$$
\begin{aligned}
&\text{For } 0 \le t \le 2\text{:} \\
X_1^j[t] &\leftarrow x_3^j[t] \oplus (x_3^j[t+1] \oplus 1)x_3^j[t+2] \oplus x_3^j[t+1]x_1^j[t+2] \oplus x_1^j[t+1]x_3^j[t+2] \;\; for\; j > 0 \\
X_2^j[t] &\leftarrow x_1^j[t] \oplus (x_1^j[t+1] \oplus 1)x_1^j[t+2] \oplus x_1^j[t+1]x_2^j[t+2] \oplus x_2^j[t+1]x_1^j[t+2] \;\; for\; j > 0 \\
X_3^j[t] &\leftarrow x_2^j[t] \oplus (x_2^j[t+1] \oplus 1)x_2^j[t+2] \oplus x_2^j[t+1]x_3^j[t+2] \oplus x_3^j[t+1]x_2^j[t+2] \;\; for\; j > 0 \\
&\text{For } t = 3, 4\text{:} \\
X_1^j[t] &\leftarrow x_3^j[t] \oplus (x_3^j[t+1] \oplus 1)x_3^j[t+2] \oplus x_3^j[t+1]x_1^j[t+2] \oplus x_1^j[t+1]x_3^j[t+2] \oplus x_3^{j-1}[t] \;\; for\; j > 0 \\
X_2^j[t] &\leftarrow x_1^j[t] \oplus (x_1^j[t+1] \oplus 1)x_1^j[t+2] \oplus x_1^j[t+1]x_2^j[t+2] \oplus x_2^j[t+1]x_1^j[t+2] \oplus x_2^{j-1}[t] \;\; for\; j > 0 \\
X_3^j[t] &\leftarrow x_2^j[t] \oplus (x_2^j[t+1] \oplus 1)x_2^j[t+2] \oplus x_2^j[t+1]x_3^j[t+2] \oplus x_3^j[t+1]x_2^j[t+2] \oplus x_2^{j-1}[t] \oplus x_3^{j-1}[t] \;\; for\; j > 0 \\
X_2^0[t] &= CG_2[t-3] \\
X_3^0[t] &= CG_3[t-3]
\end{aligned}
\tag{3}
$$

The S-Box index j assumes a one-dimensional array extracted from the three-dimensional state of KECCAK(Fig. 1). For $x = 3$ and $x = 4$, two arrays are derived where $0 < j \le y + 5z$.

block σ of either the Associated Data or the Plaintext. Only one KECCAK-p* block is instantiated, which means that the input string and the number of rounds are decided based on the function to run. The three aforementioned functions are encoded in a two bit control signal `mode` as: 00, 10 and 11 respectively.

Two padding blocks are instantiated, one for D.start operation and another one for D.step and D.stride that append a string with the following structure: 10...01 with enough zeros to get the required length. Fig. 5 depicts the diagram of this block: the unprotected version is represented by the black datapath, and the protected version by the red dotted one.

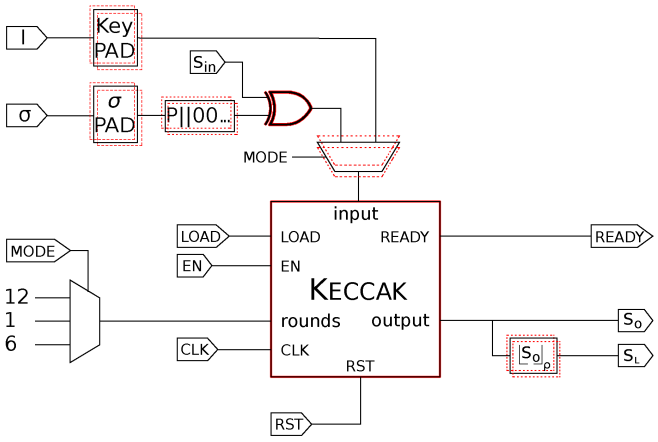

Fig. 5: Unprotected and protected MONKEYDUPLEX implementation

C. MONKEYWRAP

The last block of the building process is MONKEYWRAP, which instantiates a single MONKEYDUPLEX block. The function call can be selected with the `mode` signal, which is decided by a Control module. This module receives two inputs that indicate when the last block of the Associated Data and the last block of the Plaintext are put in. Together with a counter, the Control module decides the value of `mode` signal, when to load, what to input, and enables MONKEYDUPLEX. Since the input σ is variable and in the hardware we can not instantiate a bus to, for example, drive 16 bits sometimes and 20 bit some other times, we declare the bus with a fix length and an integer length σ_{ln} is attached so that the padding is correctly applied.

Very few extra hardware is needed to extend the encryption block to support *decryption* as well. We add a general control signal `decrypt`, which decides if the block encrypts or decrypts. New control logic is needed to decide the input to MONKEYDUPLEX, between the input plaintext and the just decrypted ciphertext, after the associated data is fully processed. The decrypted plaintext is released all at once, but only after the new tag T' is calculated and checked that it corresponds to the received one T. Thus, we implement the tags comparison and add new registers to keep the already calculated plaintext while the new tag is calculated.

D. Securing the implementations

To protect these last two blocks is trivial, for the only non-linear operation in the whole design is inside the permutation. Thus, simple repetition suffices to secure the surroundings. The control signals do not carry sensitive information, hence we do not need to instantiate more than one Control block. When securing the decryption datapath, a problem arises. How do we compare the new calculated tag shared with the input tag? To do the comparison we have to unshare T', resulting in a *non-completeness* violation. This is not a problem since the attacker already knows the tag, which is given as input to the decryption. Probing, and getting a different tag, just means that the attacker learns something went wrong in the authentication process, but he never learns secret data.

The top module details are given in Fig. 6: the unprotected encryption is represented in solid line, and the extension to

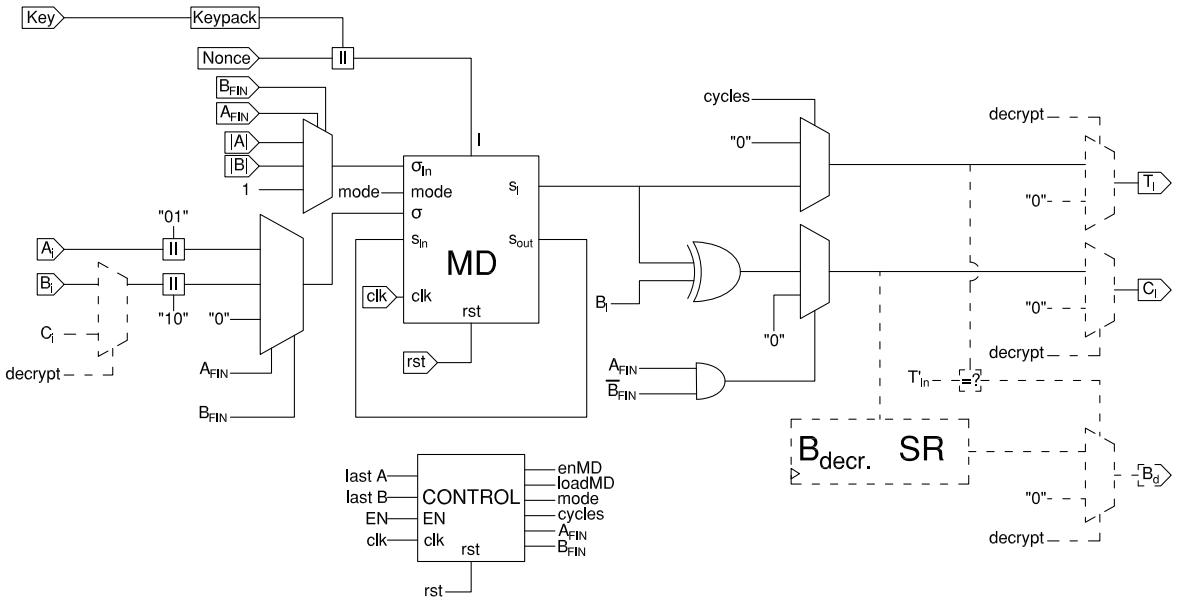

Fig. 6: The MONKEYWRAP implementation including encryption&decryption

allow decryption in dashed line. For the sake of clarity, we draw just a single datapath. The scheme of the protected version follows the same procedure as in the previous block: the protected MONKEYDUPLEX is instantiated, and all the surrounding logic including Muxes, paddings, the shift register, and the comparator are repeated three times, one per each share. As mentioned before, the Control module is left as it is, feeding the same signals to all shares.

V. RESULTS

Here we present the synthesis results for all the implementations: area, latency, max. frequency, and power. Other than the four bits appended to the state to act as "guards", which are given together with the initial sharing, there is no extra randomness needed. Tab. II summarizes them.

Three shares are used to secure our implementations. Intuitively, the protected implementations should be three times larger in area than the unprotected ones. Instead, they are around 3.6 times larger. The extra overhead comes from the more complex implementation of the non-linear function.

Since decryption mode does not reveal the plaintext until the tags are verified, we need to store it. We implement a shift register that keeps the decrypted plaintext block by block. To be able to decrypt any message in the session, the shift register has to be the size of the longest cipher. This means the area of the decryption depends on the length of the ciphertext/plaintext. The area of the shift register is given in function of the number of plaintext bits $\#B$, where the expression has two coefficients: the first one refers to the combinational area per bit, calculated experimentally, and the second one refers to the sequential area, 5.67 GE being the area of a single bit register. Note that, apart from the shift register area overhead, the area increase of the encryption&decryption module is minimal.

We use a special notation to indicate the latency in the KETJE implementations: Cycles per block for processing the Associated Data + Cycles per block for the Plaintext + 6 (stride cycles) + #cycles to produce 128 bit tag. We do not include here the cycles needed to initialize a session, which are always 12 cycles. In encryption mode the Ciphertext is available after Associated Data and Plaintext have been processed. In decryption mode both Plaintext and Tag are given at the end. Note that smaller versions take longer to precess since the absorption rate is smaller.

The results gathered in Tab. II are given using the NanGate 45nm Open Cell Library [17] and synthesized with Synopsys DC Compiler 2017.09. We *compile* with `-exact_map` option, and medium effort for mapping, area, and power to minimize the differences between the written and the synthesized code, and to avoid optimizations that could affect the security.

VI. EVALUATION

To test our implementations in practice, we deploy them into FPGA and conduct a leakage detection experiment. No information leakage is detected with up to 100 million traces.

1) TVLA: We use the method presented in [18], known as non-specific test vector leakage assessment (TVLA) to detect leakage, or similarly, to detect if intermediate variables have correlation with secret data. This test is not an attack, and retrieving the key is not the aim. This test gives information about the leakage, whose presence is a necessary condition, but not sufficient, for an attack to succeed. The fact that no leakage is observed, gives confidence to the designer on the security of the design.

We mount the experiment following the methodology described in [19]. We collect power traces from two different

TABLE II: Synthesis results for the entire KETJE family using the NanGate 45nm Open Cell Library

Design	State (bits)	Area(GE) χ	Area(GE) Total	Latency (per round/block)	Max.Freq. (MHz)	Power (mW)
Protected KECCAK						
p*[200]	200	6 319	15 113	1	892.85	1.58
p*[400]	400	12 639	30 104	1	892.85	2.87
p*[800]	800	25 279	60 092	1	909.1	5.47
p*[1600]	1 600	50 346	119 816	1	877.2	9.96
Unprotected KETJE encryption						
JR	200	707	5 447	1+1+6+8	632.9	0.682
SR	400	1 413	9 663	1+1+6+4	591.7	1.12
MINOR	800	2 827	19 665	1+1+6+1	585	2.34
MAJOR	1 600	5 653	37 650	1+1+6+1	543.5	4.13
Protected KETJE encryption						
JR	200	6 319	18 335	1+1+6+8	892.85	2.08
SR	400	12 639	35 136	1+1+6+4	892.85	3.63
MINOR	800	25 279	73 516	1+1+6+1	909.1	7.75
MAJOR	1 600	50 346	144 022	1+1+6+1	877.2	14
Unprotected KETJE encryption&decryption						
JR	200	707	$6\,109 + 3.6 \cdot \#B + 5.67 \cdot \#B$	1+1+6+8	632.9	0.93
SR	400	1 413	$10\,276 + 5.6 \cdot \#B + 5.67 \cdot \#B$	1+1+6+4	591.7	1.29
MINOR	800	2,827	$20\,554 + 19.6 \cdot \#B + 5.67 \cdot \#B$	1+1+6+1	585	2.7
MAJOR	1 600	5 653	$39\,596 + 40 \cdot \#B + 5.67 \cdot \#B$	1+1+6+1	543.5	4.86
Protected KETJE encryption&decryption						
JR	200	6 319	$20\,032 + 3 \cdot 3.6 \cdot \#B + 3 \cdot 5.67 \cdot \#B$	1+1+6+8	892.85	2.8
SR	400	12 639	$36\,645 + 3 \cdot 5.63 \cdot \#B + 3 \cdot 5.67 \cdot \#B$	1+1+6+4	892.85	4,24
MINOR	800	25 279	$77\,608 + 3 \cdot 19.6 \cdot \#B + 3 \cdot 5.67 \cdot \#B$	1+1+6+1	909.1	9.33
MAJOR	1 600	50 346	$145\,259 + 3 \cdot 40 \cdot \#B + 3 \cdot 5.67 \cdot \#B$	1+1+6+1	877.2	16.4
Related work [16]						
Protected KECCAK						
PARALLEL pip.*	1 600	57 600	111 800	3	837.5	-
PARALLEL d.c.*	1 600	44 200	100 500	2	803.9	-
PARALLEL pip.**	1 600	50 400	97 700	3	846.7	-
PARALLEL d.c.**	1 600	38 400	85 700	2	877.2	-

*Synthesis library UMC 130nm
**Synthesis library UMC 90nm

plaintext groups (one fixed and the other one random) and compare the two sets using the t-test statistic. When the t-statistic surpasses the threshold of ± 4.5, one can conclude with confidence 99.9995% that the two sets of power traces follow a different distribution and hence, the design leaks.

We first test an unprotected implementation to verify that our setup is sound and we can find leakage, which is expected in every order. We then activate the countermeasure and run the test again. Now only second-order leakage is expected, since our design has first-order protection. Fig. 7 illustrates this evaluation results.

2) Setup: To evaluate the security of our implementations we use a SAKURA-G board, specifically designed for side-channel evaluation, which includes two Spartan-6 FPGAs. To reduce the experimental noise, we split our implementation in the two FPGAs: the control FPGA, which handles the communication with the host computer and generates the shares for the cryptographic FPGA. In the crypto FPGA we deploy the actual scheme. This way we isolate the power consumption of the actual encryption, reducing considerably the noise. We use a very slow 3 MHz clock to ensure clear power traces with minimal overlap between consecutive time samples. The synthesis of the design is done using the Xilinx 14.7 tools with the `KEEP HIERARCHY` constraint, in order to avoid optimizations across different shares. We measure the power consumption using a Tektronix DPO 7254C oscilloscope, sampling at 1 GS/s with 5 000 points per frame, where fourteen and a half rounds can be appreciated.

3) The experiment: From all the designs, we deploy the smallest KECCAK to test, where $b = 200$. Given the properties of the permutation, the conclusions on a specific version of KECCAK transfer to versions of different width. This property is known as the Matryoshka principle [10].

To complete the full KETJE implementation, only linear operations are used on top of KECCAK. As shown in [20], linear operations prior to non-linear operations might introduce dependencies among shares and hence non-completeness failures. TI does not have this problem, given that every share always operates with one of the shares missing, ensuring protection even if there are dependencies. Thus, conclusions from the evaluation of the underlying permutation remain valid for the full implementation for first-order protection. This conclusion would not be valid in masking schemes based on CMS or DOM, which need independent inputs.

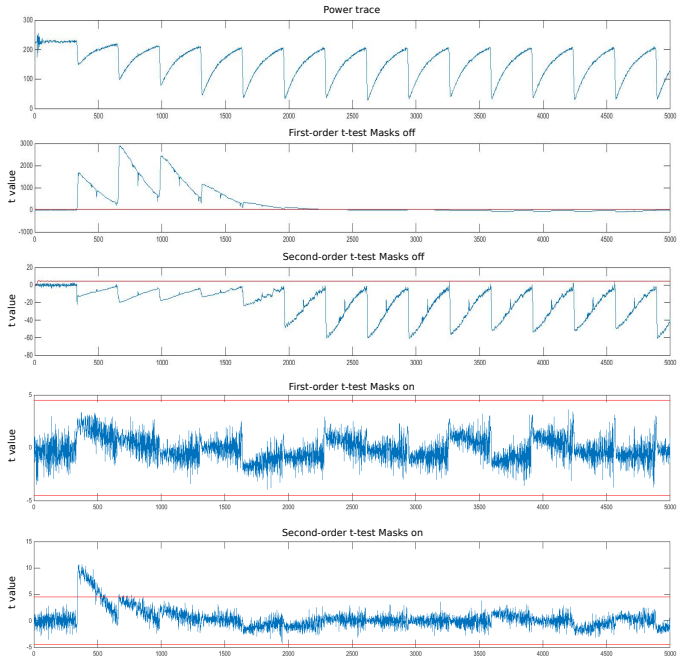

Fig. 7: Welch t-test evaluation of full KECCAK. Top to bottom: power trace, first- and second-order masks off test, and first- and second-order masks on test

VII. CONCLUSION

In this work we have presented a new first-order protected KECCAK implementation using TI and the "Changing of the Guards" technique. In addition to this, we introduce a general implementation of first-order side-channel protected KETJE authenticated encryption schemes that are suitable for different applications, from lightweight to high throughput purposes. Furthermore, we extend this implementations to include both encryption and decryption functionalities in the same module.

We tested the underlying permutation finding no leakage with up to 100 million traces. By using Threshold Implementations, we are able to transport the security conclusions from the permutation to the whole scheme. Moreover, by implementing the trick of "Changing of the Guards", no extra randomness is needed to provide first-order security, and thus, we do not need to place any dedicated randomness source next to our design, avoiding any problems this might entail.

ACKNOWLEDGMENT

The authors would like to thank Thomas De Cnudde for his valuable help. This work was partially supported by the NIST Research Grant 60NANB15D346.

REFERENCES

[1] P. C. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, 1999, pp. 388–397. [Online]. Available: https://doi.org/10.1007/3-540-48405-1_25

[2] S. Mangard, E. Oswald, and T. Popp, *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007.

[3] Y. Ishai, A. Sahai, and D. A. Wagner, "Private circuits: Securing hardware against probing attacks," in *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, 2003, pp. 463–481. [Online]. Available: https://doi.org/10.1007/978-3-540-45146-4_27

[4] E. Trichina, "Combinational Logic Design for AES SubByte Transformation on Masked Data," Cryptology ePrint Archive, Report 2003/236, 2003. [Online]. Available: http://eprint.iacr.org/2003/236

[5] S. Nikova, C. Rechberger, and V. Rijmen, "Threshold implementations against side-channel attacks and glitches," In ICICS, volume 4307 of LNCS, pages 529-545. Springer, 2006.

[6] B. Bilgin, B. Gierlichs, S. Nikova, V. Nikov, and V. Rijmen, "Higher-order threshold implementations," In ASIACRYPT, volume 8874 of LNCS, pages 326-343. Springer, 2014.

[7] O. Reparaz, B. Bilgin, S. Nikova, B. Gierlichs, and I. Verbauwhede, "Consolidating masking schemes," in *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, 2015, pp. 764–783. [Online]. Available: https://doi.org/10.1007/978-3-662-47989-6_37

[8] H. Gross, S. Mangard, and T. Korak, "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order," Cryptology ePrint Archive, Report 2016/486, 2016, http://eprint.iacr.org/2016/486.

[9] G. Bertoni, J. Daemen, M. Peeters, G. V. Assche, and R. V. Keer, "Caesar submission: Ketje v2," September 2016.

[10] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "The keccak reference," http://http://keccak.noekeon.org/, January 2010.

[11] J. Daemen, "Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 137–153.

[12] N. Samwel and J. Daemen, "Dpa on hardware implementations of ascon and keyak," in *Proceedings of the Computing Frontiers Conference*, ser. CF'17. New York, NY, USA: ACM, 2017, pp. 415–424. [Online]. Available: http://doi.acm.org/10.1145/3075564.3079067

[13] F. Wegener and A. Moradi, "A first-order SCA resistant AES without fresh randomness," in *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings*, 2018, pp. 245–262. [Online]. Available: https://doi.org/10.1007/978-3-319-89641-0_14

[14] G. Bertoni, J. Daemen, M. Peeters, and G. V. Assche, "Building power analysis resistant implementations of Keccak," Second SHA-3 candidate conference, August 2010.

[15] B. Bilgin, J. Daemen, V. Nikov, S. Nikova, V. Rijmen, and G. V. Assche, "Efficient and first-order dpa resistant implementations of keccak," in CARDIS, volume 8419 of LNCS pp 187-199, June 2014.

[16] H. Gross, D. Schaffenrath, and S. Mangard, "Higher-order side-channel protected implementations of keccak," Cryptology ePrint Archive, Report 2017/395, 2017, https://eprint.iacr.org/2017/395.

[17] NANGATE, "The NanGate 45nm Open Cell Library," available at http://www.nangate.com.

[18] J. Cooper, E. D. Mulder, G. Goodwill, J. Jaffe, G. Kenworthy, and P. Rohatgi, "Test Vector Leakage Assessment (TVLA) methodology in practice," International Cryptographic Module Conference, 2013, http://icmc-2013.org/wp/wp-content/uploads/2013/09/goodwillkenworthtestvector.pdf.

[19] O. R. B. Gierlichs and I. Verbauwhede, "Fast leakage assessment," in *Cryptographic Hardware and Embedded Systems – CHES 2017*, W. Fischer and N. Homma, Eds. Cham: Springer International Publishing, 2017, pp. 387–399.

[20] V. Arribas, B. Bilgin, G. Petrides, S. Nikova, and V. Rijmen, "Rhythmic Keccak: SCA security and low latency in hw," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2018, no. 1, pp. 269–290, 2018. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/840