Protecting the NOEKEON Cipher Against SCARE Attacks in FPGAs by using Dynamic Implementations

Julien Bringer[1], Hervé Chabanne[1,2], Jean-Luc Danger[2]

[1] Sagem Sécurité
[2] Télécom ParisTech

Abstract. Protecting an implementation against Side Channel Analysis for Reverse Engineering (SCARE) attacks is a great challenge and we address this challenge by presenting a first proof of concept. White-box cryptography has been developed to protect programs against an adversary who has full access to their software implementation. It has also been suggested as a countermeasure against side channel attacks and we examine here these techniques in the wider perspective of SCARE. We consider that the adversary has only access to the cryptographic device through its side channels and his goal is to recover the specifications of the algorithm. In this work, we focus on FPGA (Field-Programmable Gate Array) technologies and examine how to thwart SCARE attacks by implementing a block cipher following white-box techniques. The proposed principle is based on changing dynamically the implementations. It is illustrated by an example on the Noekeon cipher and feasibility in different FPGAs is studied.

Keywords. SCARE attacks, white-box cryptography, FPGA.

1 Introduction

White-box cryptography has been introduced in the domain of Digital Rights Management with the ambitious goal of protecting keys of a block cipher while leaving to an adversary whole access to the software implementing this algorithm. Practically, this leads ciphers to be represented by a network of look-up tables. White-box implementations for DES and AES have been given in [2, 3]. These protections can affect either an encryption algorithm E (naked variant) or F o E o G where F and G are secret bijections. Today, there has been large cryptanalytic efforts on many implementations (naked or not) [1, 10, 12, 16, 18, 24].

We here retain from white-box cryptography the use of look-up tables which hides the structure of a block cipher. In this paper, we take back

this technique but in a different context. Our aim is now to protect a block cipher implemented in hardware. Moreover, we move from a white-box environment to a grey-box one where adversaries only get various side channels from the running algorithm (see for instance The Side Channel Cryptanalysis Lounge, `http://www.crypto.ruhr-uni-bochum.de/en_sclounge.html`). The idea of using white-box cryptography as a possible countermeasure against side channel attacks is not new. However, we consider here the situation where the adversary has not all the elements of the algorithm in his possession and wants to recover the missing details by Side Channel Attack Reverse Engineering (SCARE).

SCARE was introduced in [19] with a proprietary algorithm for GSM phone. These results were improved by [4]. [9] studies DES in this context and is extended to Unknown Hardware Feistel Implementation by [20]. Use of proprietary algorithms and the protection of their specifications can be conceived in organizations that can afford the risk of relying on secret algorithms such as military groups or pay-tv / mobile network operators. Nevertheless, achieving an implementation which actually resists SCARE attacks is still a great challenge.

We explain why white-box cryptographic implementations indeed provide an effective solution against SCARE. In particular, by constantly renewing the look-up tables - as in classical counter-measures against side channels - we might reduce the side channels information on the specifications of the running block cipher to almost nothing. We also underline that our proposal could be implemented on today's chips as FPGAs and we illustrate this aspect with the cipher Noekeon [6].

The remainder of this paper is organized as follows. Section 2 explains why, in a theoretical model, dynamic white-box cryptography actually counteracts SCARE. Section 3 gives an overview of the Noekeon cipher. Section 4 focuses on practical aspects of our proposal where we illustrate our ideas by giving a complete FPGA implementation of the Noekeon cipher in this white-box context.

We emphasize that the reader should see this work as a first proof of concept. This might lead to a solution practically addressing the challenge of resisting SCARE.

2 Security Model and General Principle of our Protection

In white-box cryptography, the classical representation of the operations and the use of the cryptographic keys within a cipher are generally turned

into a representation of look-up table network, namely a set of look-up tables $\{T_l\}_{l \in L}$ with some arcs between the tables (an arc corresponds to the output of one table becoming an input of another one). Then the look-up tables are obfuscated by encoding their input and output with encoding bijections; for instance T_l would be replaced by $f_{out} \circ T_l \circ f_{in}$. The choice of encoding functions of input and output is made according to the existing arcs between the tables so that the entire implementation does not change: if there is an arc from T_l to $T_{l'}$, the output encoding of T_l must be the inverse of the corresponding input encoding of $T_{l'}$. Encodings at the same time of several tables gathered together are possible as well.

From a side channel perspective, this implementation technique has yet some advantages. The implementation following a network of look-up tables does not give a direct access to the algorithm specifications and the main advantage is that a look-up table implementation improves the resistance against side channel analysis as the activity of the internal variables processed in the memory core are hardly discernible.

Remark 1. A look-up table access seems hard to exploit by Simple Power Analysis and the resistance against Differential Power Analysis (DPA) is improved when the size of tables decreases. Indeed the size of the tables has an effect onto the DPA signal-to-noise ratio; this is illustrated for instance by [11].

2.1 Model

We detail here our security model and the corresponding assumptions. Following the previous remark, we consider that an adversary can obtain information on a look-up table neither by reading directly on it nor by measuring some signals during its execution. That is why we assume that a look-up table does not leak information by itself. However, we consider that the adversary can obtain information (partial or not) on the input and output of a look-up table execution. We assume that the adversary can make use of High-Order differential Side Channel Analysis (HO-SCA; introduced in [15,17] for High-Order DPA, see also [13]) to obtain several such information during an execution of the encryption algorithm.

We assume that the encryption algorithm implementation is made of consecutive steps where each step corresponds to the parallel evaluation of several look-up tables. We consider that the adversary can make several measures during an execution but only one measure per step. In the sequel, such an adversary is called an HO-SCA adversary. We define also a

generalized HO-SCA adversary as an HO-SCA adversary which is enabled to make several measures at the same step.

2.2 Our Protection in a Theoretical Nutshell

In a white-box implementation, an attacker can read at any moment of the execution the result provided by a look-up table, i.e. an encoded output $f(x)$ (for some encoding function f), where x corresponds to a non-obfuscated intermediate result of the underlying cipher. In [18], the authors explain – under some conditions on the cipher structure – how this property can be exploited to recover the keys. In our context of grey-box attacks, the attacker would encounter more difficulties to read an entire result but the same situation may occur. To thwart this and at the same time to achieve security against SCA, we introduce a dynamic implementation by renewing the encoding bijections after each execution of the cipher. This is somewhat a generalization of [5] which applies a same random permutation to all the intermediate values during an AES execution in order to achieve first-order DPA resistance.

Given an encryption algorithm E which can be implemented as a network of look-up tables with the set of tables $\{T_l\}_{l\in L}$, the implementation at a time t is given by the tables $\{T_l[t] = f_{l,out}[t] \circ T_l \circ f_{l,in}[t]\}_{l\in L}$. After the implementation execution, a new set of random encoding bijections $\{g_{l,in}, g_{l,out}\}$ is chosen and the implemented tables are transformed into $g_{l,out} \circ T_l[t] \circ g_{l,in}$, i.e. the implementation of the tables evolves into $\{T_l[t'] = f_{l,out}[t'] \circ T_l \circ f_{l,in}[t']\}_{l\in L}$ with $f_{l,out}[t'] = g_{l,out} \circ f_{l,out}[t]$ and $f_{l,in}[t'] = f_{l,in}[t] \circ g_{l,in}$.

Remark 2. Constantly renewing the look-up tables might seem a very costly solution but we explain later on in Section 4.5 why it might be feasible for practical hardware implementations. In our model, we moreover assume that the renewal process does not leak. Consequently, the adversary is seen here as a kind of constrained HO-SCA adversary. Note that a thorough practical analysis of the implementation would be necessary in the future to verify this claim. We thus see these points as an interesting challenge for further researches.

Given a fixed input message and a fixed algorithm (and key), the use of such random encodings which are renewed after each execution implies that an HO-SCA adversary cannot distinguish the intermediate values from uniform ones. If there are no correlated encodings within any step, then the resistance holds against a generalized HO-SCA adversary.

3 Noekeon Cipher [6]

We give here an overview of the Noekeon cipher. Noekeon was proposed to the NESSIE project in 2000 [6, 7, 14]. Noekeon is a 128-bit block cipher over 16 rounds. Noekeon maintains a state of four 32-bit words: a_0, a_1, a_2, a_3. Each round is constituted by the following operations:

1. A first round constant is XORed to a_0,
2. A linear transformation θ is applied to the four words a_0, a_1, a_2, a_3. During the execution of θ, the round key is introduced by an XOR into the state.
   Consider the involutive mapping that modifies four 32-bit words by XORing a linear transformation of the XOR of the other two words. This linear transformation consists of taking a word X, rotating it over a byte to the left to give Y and rotating it over a byte to the right to give Z and XORing X, Y and Z, $Z \leftarrow X \oplus Y \oplus Z$. θ consists of applying the described mapping, where the state words in odd positions are modified ($X = a_0 \oplus a_2$, Z is XORed to a_1 and a_3), followed by XORing the key to the state, followed by again applying the described mapping, where the state words in even positions are modified.
   For k the working key and a the state, each formed by four 32-bit words, the computation of $\theta(k, a)$ is illustrated by Table 1.

Table 1. Computation of $\theta(k, a)$

$temp \leftarrow a_0 \oplus a_2;\ temp \leftarrow temp \oplus (temp >> 8) \oplus (temp << 8);$
$a_1 \leftarrow a_1 \oplus temp;$
$a_3 \leftarrow a_3 \oplus temp;$
$a_0 \leftarrow a_0 \oplus k_0;\ a_1 \leftarrow a_1 \oplus k_1;\ a_2 \leftarrow a_2 \oplus k_2;\ a_3 \leftarrow a_3 \oplus k_3;$
$temp \leftarrow a_1 \oplus a_3;\ temp \leftarrow temp \oplus (temp >> 8) \oplus (temp << 8);$
$a_0 \leftarrow a_0 \oplus temp;$
$a_2 \leftarrow a_2 \oplus temp;$

3. A second round constant is XORed to a_0.
4. π_1: The words a_1, a_2, a_3 are rotated of 1, 5, and 2 bits, respectively, to the left.
5. Γ: All bits in the same position in a_0, a_1, a_2, a_3 are grouped together into nibbles (i.e. words of 4 bits) which go through the same non-linear bijection γ (i.e. γ is applied 32 times, once for each possible nibble).
6. π_2: The words a_1, a_2, a_3 are rotated of 1, 5 and 2 bits, respectively, to the right.

Finally, after the last round, a final constant is XORed to a_0 and θ is applied.

To sum up, each round of the cipher can be decomposed in one non-linear step Γ and several linear ones. We have: 16 matrices M_j, $j = 1, \ldots, 16$ representing the steps 1 to 3 (from first round constant XOR to the second round constant XOR), one matrix for each round; 16 applications of π_1, Γ and π_2; and a matrix $M' = M_{\text{final}}$ for the final step (the final constant XOR and the application of θ).

4 Our Implementation of the NOEKEON Cipher

Compared to AES, Noekeon is a softer candidate for the technique we have introduced in Section 2.2, mainly due to the non-linear transformation where the corresponding look-up table is smaller than for the AES. Nonetheless, it would be interesting to investigate its adaptation to other similar block ciphers.

4.1 General Description

Our implementation follows the strategy of Section 2 by the use of several tables look-up representation with the inclusion of input and output encoding functions to hide the key and the running values during computations.

Each of the 32 applications of γ in the non-linear step Γ of each round is implemented by a table look-up. A different 4×4 table is used for each γ. Moreover, instead of γ, our table represents $f_i \circ \gamma \circ g_i^{-1}, i = 0, \ldots, 31$ where the f_i's, g_i's are random bijections over nibbles. We need $16 \times 32 = 512$ different tables for the whole algorithm, which takes $512 \times 2^4 \times 4$ bits (4KBytes). Following this, almost the whole implementation will operate on nibbles. We call the nibble of index i (for $i \in \{0, \ldots, 31\}$), denoted nib_i, the nibble containing all the bits of index i of the current state a_0, a_1, a_2, a_3, i.e. $nib_i = a_0^i a_1^i a_2^i a_3^i$ where a_k^i denotes the i-th bit of a_k. Only π_1, π_2 are seen as operations on bits. In fact, as rotations of different order of the words a_0, a_1, a_2, and a_3, they correspond to permutations of bits between nibbles and can be simply hardwired.

Concerning the 128×128 binary matrices $M_1, \ldots, M_{16}$ and M', we observe from Table 1 that the output bits of a nibble do not depend on all the input bits. By construction of θ, which is based on XORs and four rotations of 8 bits (to the left and to the right), and for a given round

key, a nibble of the output state depends only on three nibbles of the input state. The corresponding formulas for the update of nib_i are given below (where the additions of index are taken modulo 32): ${}^T\left(a_0^i\, a_1^i\, a_2^i\, a_3^i\right)$ is updated as

$$\begin{pmatrix} a_0^i \oplus k_0^i \oplus \left(a_1^i \oplus k_1^i \oplus a_3^i \oplus k_3^i\right) \oplus \left(a_1^{i+8} \oplus k_1^{i+8} \oplus a_3^{i+8} \oplus k_3^{i+8}\right) \\ \oplus \left(a_1^{i+24} \oplus k_1^{i+24} \oplus a_3^{i+24} \oplus k_3^{i+24}\right) \\ a_1^i \oplus \left(a_0^i \oplus a_2^i\right) \oplus \left(a_0^{i+8} \oplus a_2^{i+8}\right) \oplus \left(a_0^{i+24} \oplus a_2^{i+24}\right) \oplus k_1^i \\ a_2^i \oplus k_2^i \oplus \left(a_1^i \oplus k_1^i \oplus a_3^i \oplus k_3^i\right) \oplus \left(a_1^{i+8} \oplus k_1^{i+8} \oplus a_3^{i+8} \oplus k_3^{i+8}\right) \\ \oplus \left(a_1^{i+24} \oplus k_1^{i+24} \oplus a_3^{i+24} \oplus k_3^{i+24}\right) \\ a_3^i \oplus \left(a_0^i \oplus a_2^i\right) \oplus \left(a_0^{i+8} \oplus a_2^{i+8}\right) \oplus \left(a_0^{i+24} \oplus a_2^{i+24}\right) \oplus k_3^i \end{pmatrix}.$$

For instance, nib_0 is updated thanks to the input bits of nib_0, nib_8 and nib_{24}. This enables us to split the representation of a such matrix into 32 smaller (4×12) binary matrices which take less room to be represented with look-up tables than a 128×128 binary matrix. For $i \in \{0, \ldots, 31\}$, the matrix used to update nib_i at round j, as it would have been done by M_j, is denoted below by U_j^i ($j \in \{1, \ldots, 16\}$) and the matrix used to update nib_i at the final step is denoted by U_{final}^i.

The implementation of a 4×12 binary matrix U is realized as follows. U is split into three 4×4 submatrices $U[0]$, $U[1]$, $U[2]$ and the computation of $U.^T(x_0, \ldots, x_{11})$ becomes $U.^T(x_0, \ldots, x_{11}) = U[0].^T(x_0, \ldots, x_3) \oplus U[1].^T(x_4, \ldots, x_7) \oplus U[2].^T(x_8, \ldots, x_{11})$ where $\oplus$ corresponds to an XOR on vectors of $GF(2)^4$. Each $U[l]$ is represented as a 4×4 look-up table, for a size of 8 Bytes, and each XOR is seen as a 8×4 look-up table, for a size of 128 Bytes. Instead of considering 2 XOR of 8 inputs, a complexity reduction is achieved by using 4 XOR of 3 inputs as the size is only of 4 Bytes (8 Bytes if we consider the 4×4 look-up table as $U[l]$).

It leads to three 4×4 encoded look-up tables and one 4×4 encoded look-up tables (for the XORs). Thus, one 4×12 binary matrix is implemented on 32 Bytes. Note that the same input/decoding strategy as for γ is also respected: random bijections over 4 bits are used at each 4 bits inputs and outputs of the tables.

By applying this method to all the U_j^i ($i \in \{0, \ldots, 31\}$, $j \in \{1, \ldots, 16\} \cup \{\text{final}\}$), we obtain $32 \times 17 \times 4$ look-up tables of 4×4 size for an overall size of $32 \times 17 \times 32$ Bytes, i.e. 17408 Bytes.

In addition to the 4×4 input/output encodings, we also insert mixing linear bijections to further disguise the representation of these matrices (as introduced in [3] to hide the separation of a bit strings into nibbles).

For this aim, we randomly choose an invertible 12×12 matrix MB_j^i for each U_j^i, and instead of implementing directly the 5 look-up tables related to U_j^i, we write U_j^i as the product of the two matrices $U_j^i.MB_j^i$ and $(MB_j^i)^{-1}$. The implementation of $U_j^i.MB_j^i$ as look-up tables is realized as explained above for U_j^i and we add the implementation of the 12×12 matrix $(MB_j^i)^{-1}$ by following the same principle, i.e. splitting into nine 4×4 submatrices with the associated XOR and the additional input/output encodings. For one matrix $(MB_j^i)^{-1}$, it gives 3 times the size of $U_j^i.MB_j^i$, i.e. 52224 Bytes.

With these representations of the matrices $M_1, \ldots, M_{16}$, M', Γ and hardwired π_1 and π_2, the total number of 4×4 look-up tables is:

$(MB_j^i)^{-1}$: $32 \times 17 \times 12$
$U_j^i.MB_j^i$: $+ 32 \times 17 \times 4$
$\pi_2 \circ \Gamma \circ \pi_1$: $+ 32 \times 16$

This gives a total size of 73728 Bytes for all the 4×4 look-up tables.

4.2 Choice of Encoding Functions

The above description is made with the choice of random bijections as encoding functions. Although it is the classical strategy with static encodings, our aim is to renew the encoding functions after each execution of the algorithm, which enables us to select bijective functions with a simpler representation.

Note also that all the input/output encoding functions are not fully independent because the output encoding function which acts on a nibble at one step must be followed by its inverse as the input encoding functions of the next operation on this nibble. In particular, as π_1 and π_2 operate like permutation of bits between nibbles, the encoding functions have to be taken accordingly. For an easy compatibility of the encodings with these permutations, we design specific functions for the input and output encoding which are before or after the application of π_1 and π_2, i.e. around γ, at the output of the $U_j^i.MB_j^i$ (for $j \in \{1, \ldots, 16\}$) and at the input of $(MB_j^i)^{-1}$ (for $j \in \{2, \ldots, 16\} \cup \{\text{final}\}$). We design these encoding functions $f : GF(2)^4 \rightarrow GF(2)^4$ as an XOR with a random padding $c_f = (c_{f,0}, c_{f,1}, c_{f,2}, c_{f,3})$ – that is $f(x) = x \oplus c_f$ – so that the inverse can be evaluated bit by bit. For instance, given all the output encoding functions $g_j^0, \ldots, g_j^{31}$ of the matrices $U_j^0, \ldots, U_j^{31}$ at some round j, it enables us to design π_1-compatible input encoding functions $f_j^0, \ldots, f_j^{31}$ of the 32 look-up tables for γ.

An example: The first look-up table for γ operates on the first nibble of the state. This nibble after permutation of the state by π_1 comes from the bit 0 of nib_0, the bit 1 of nib_{31}, the bit 2 of nib_{27} and the bit 3 of nib_{30}. If $g_j^i(x) = x \oplus (c_{g_j^i,0}, c_{g_j^i,1}, c_{g_j^i,2}, c_{g_j^i,3})$, then f_j^0 is defined as $f_j^0(x) = x \oplus (c_{g_j^0,0}, c_{g_j^{31},1}, c_{g_j^{27},2}, c_{g_j^{30},3})$. In the next section, we consider that encodings for inputs of an XOR table are also chosen in this form. This enables us to lighten further the architecture.

Remark 3. Note that in our grey-box context, we can relax constraints on the encoding functions as the adversary has no direct access to the look-up tables but only to their side channels. We here only have to periodically renew these tables and a simple linear mask might be sufficient to this purpose. Moreover, this simplifies the renewal (cf. end of Section 4.3).

4.3 Implementation Complexity in FPGAs

The FPGA architectures are based on Look-Up Tables (LUTs) which have at least 4 inputs (LUT4). LUT4 are well suited to implement the Noekeon algorithm as a 4×4 look-up table is implemented by 4 LUT4s. A first implementation consists in considering 32 parallel computations of nibbles at each round as explained in section 4.1. Figure 1 represents the corresponding architecture for one nibble path.

This gives a total of **36864** LUT4s for the whole function. This architecture can be optimized by considering eight parallel processings of four nibbles. Each bundle is composed of four nibbles

$$(nib_i, nib_{i+8}, nib_{i+16}, nib_{i+24}).$$

The architecture is composed of 8 bundle paths, a bundle path being illustrated by Figure 2. An advantage of this architecture is that all output encodings at the same level can be chosen independently thus leading to resistance against a generalized HO-SCA adversary (cf. Section 2).

There is 44 4×4 look-up tables for each bundle path. Hence the implementation requires a total of **23808** LUT4s for the whole Noekeon implementation (22528 for the 16 rounds and 1280 for the last round).

An important complexity optimization is performed by serializing the 16 rounds calculation. Hence the same synchronized structure is used for the 16 rounds. The robustness is not damaged if a specific care is taken, as described in chapter 4.5. In this case 1408 LUT4s are required for the round implementation and 1280 for the last round. If we consider the

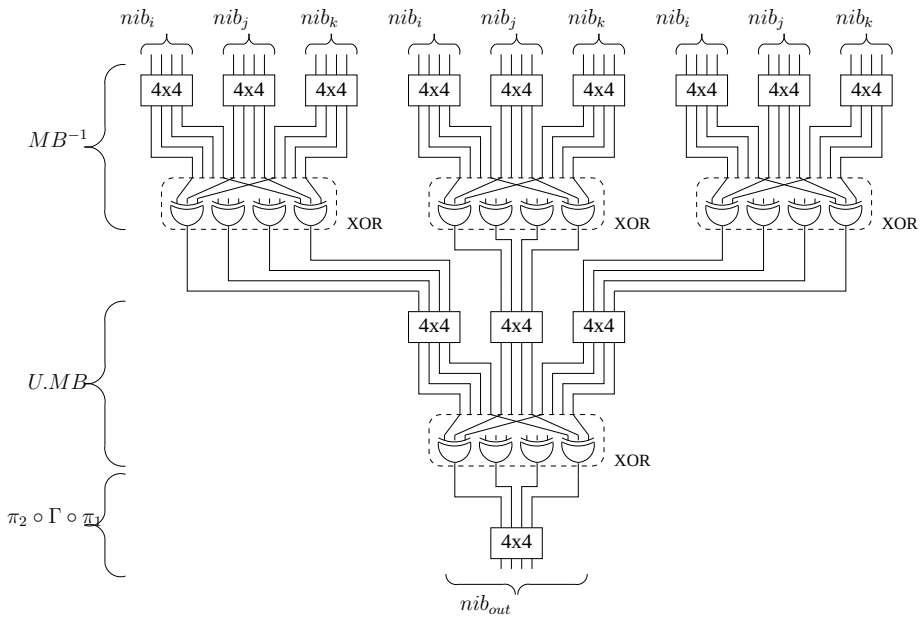

Fig. 1. Architecture of the nibble path.

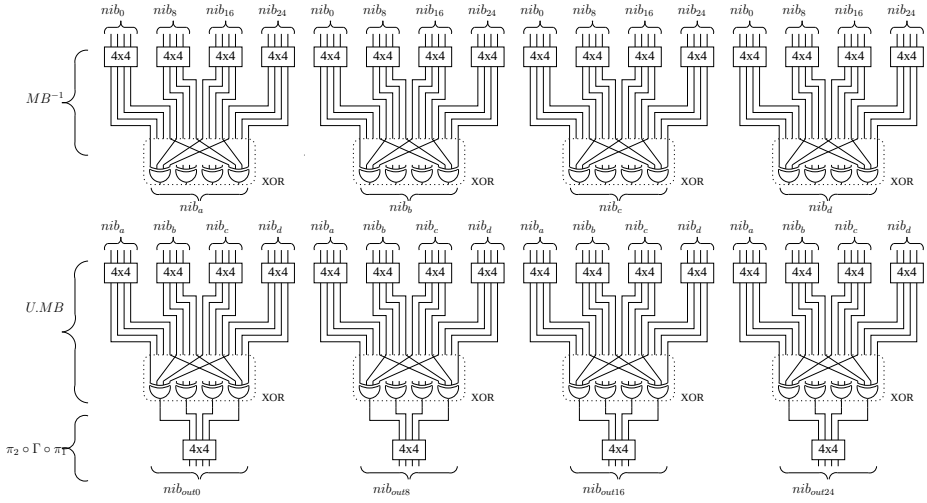

Fig. 2. Optimized architecture with Four-Nibbles bundle path.

128-bit register to store the intermediate results at each round, this gives a total of **2816** FPGA cells of LUT4.

4.4 Feasibility in Current FPGAs

Most SRAM-based FPGAs (those from ALTERA, XILINX and Lattice for instance) [21–23] have cells composed of LUT4 but recent families have a more advanced cell structure. For instance the STRATIX II, III and IV from ALTERA take advantage of the Adaptive LUT Module (ALM) which could be configured as two independent LUT4. The VIRTEX 5 family from XILINX has six-input look-up (LUT6) which can output two signals. Table 2 summarizes the occupation percentage in the SRAM-based FPGA devices. Columns **4×4** and **TOTAL** indicate respectively the number of cells for the 4×4 table and the total number of cells for the proposed Noekeon implementation with the serial implementation. In the rightmost column the occupancy rate is indicated for the biggest devices. It remains relatively low and proves the white box feasibility of the Noekeon implementation in most FPGA devices.

Table 2. Occupation rate

Device	cell type	4×4	TOTAL	max cell	occ. rate
CYCLONE II	LUT4	4	2816	68416	4.1%
CYCLONE III	LUT4	4	2816	119088	2.3%
STRATIX	LUT4	4	2816	79040	3.6%
STRATIX II	ALM	2	1408	71760	2%
STRATIX III	ALM	2	1408	135200	1%
STRATIX IV	ALM	2	1408	212480	0.56%
LATTICE ECP2	LUT4	4	2816	68000	4.1%
LATTICE SC/M	LUT4	4	2816	115000	2.4%
SPARTAN3	LUT4	4	2816	74880	3.8%
VIRTEX2 PRO	LUT4	4	2816	99216	2.8%
VIRTEX4	LUT4	4	2816	200448	1.4%
VIRTEX5	LUT6	2	1408	207360	0.7%

4.5 Dynamic implementation for encoding renewal

The tables are updated periodically in order to change the masking functions $f_{l,in}$ and $f_{l,out}$ (cf. Section 2). Moreover they have to be changed after each ciphering round for the serial implementation in order to keep a high robustness level . In FPGA a solution could be to reconfigure it

either completely or partially (only for the XILINX devices) [23] but the time needed for reconfiguration would reduce significantly the ciphering rate. Another solution – as a trade-off between practical and theoretical security – is to add extra pins to input random number values issued from a Random Number Generator (RNG) or ideally a True RNG (TRNG) [8] based on non deterministic physical phenomenon. The extra RNG pins do not add logic in some FPGAs as in ALTERA STRATIX II, III and IV [21]. The cost to implement a 4×4 look up tables is two ALMs (Adaptive Logic Modules) as shown in Table 2. If two extra pins for random numbers are added, there is no supplementary cost as the ALM is able to implement two LUT6 having four common inputs, as illustrated in Figure 3.

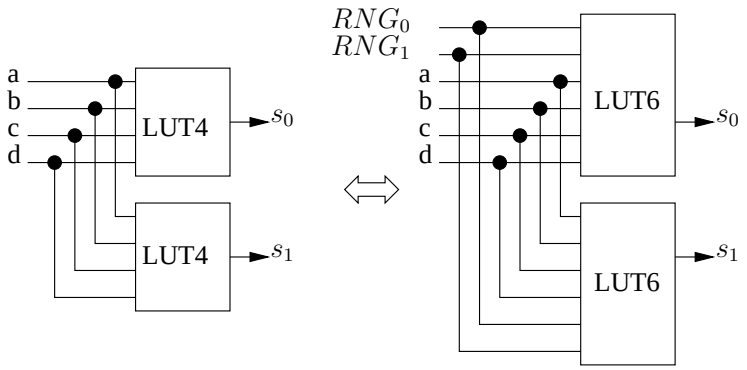

Fig. 3. Using two RNG pins does not add ALMs.

The global architecture is shown in Figure 2 with its 5 layers of look up tables. Hence there is a need of 5 TRNGs, one for each layer, the same RNG being used for $f_{l,in}$ and $f_{l,out}$ of adjacent look up tables as shown in Figure 4.

2 bits of entropy per nibble for encoding renewal allows the generation of $2^{64\times5}$ different implementations per round. This implies a TRNG which has too stringent speed requirement to be feasible. A complexity reduction is to apply 2 bits of TRNG for a bundle or group of 4 nibbles. One bit is used for $f_{l,in}$ and the other for $f_{l,out}$. In this case the TRNG has to produce a total of 40 bits (One bit for each of the 5 layers, multiplied by 8 bundles). The internal code pointed by each bit value can change by static or dynamic reconfiguration. In order to reduce the TRNG speed, the random number can generate a seed at the beginning of every ciphering

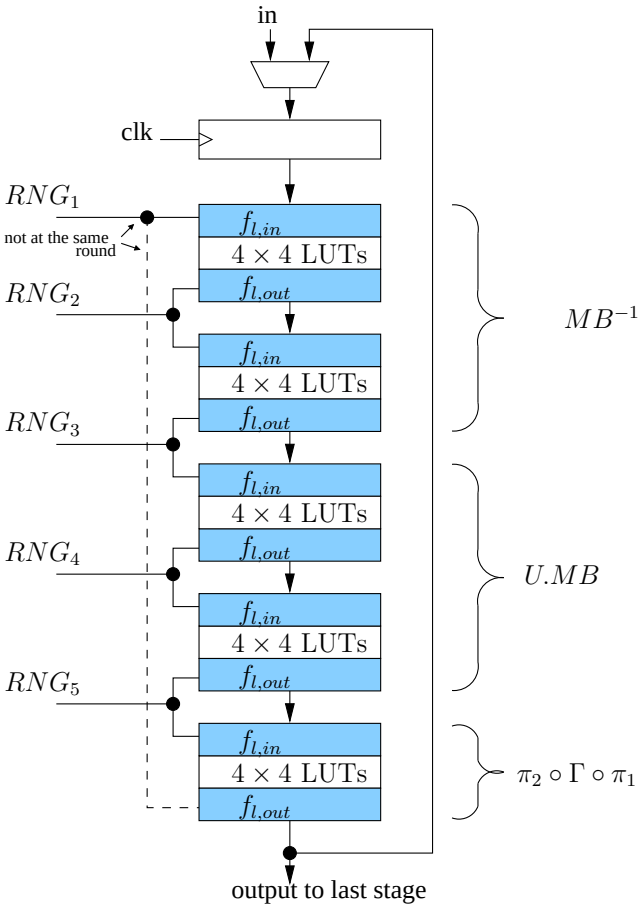

Fig. 4. Architecture with 5 layers of LUTs.

operation. At every round a specific sequence transforms the coding and allows to change the 40-bit RNG word. This RNG change allows to keep a high robustness level as the hamming distance between two rounds remains random. This RNG transformation can be based on a Non-Linear operation like a set of AES substitution boxes between two rounds. Five AES S-Boxes are needed in case of 40-bit TRNG seed. A padding stage is also necessary to feed correctly the encoding functions as explained in section 4.2. Figure 5 shows the general architecture of a Noekeon loop with the RNG and its associated processing.

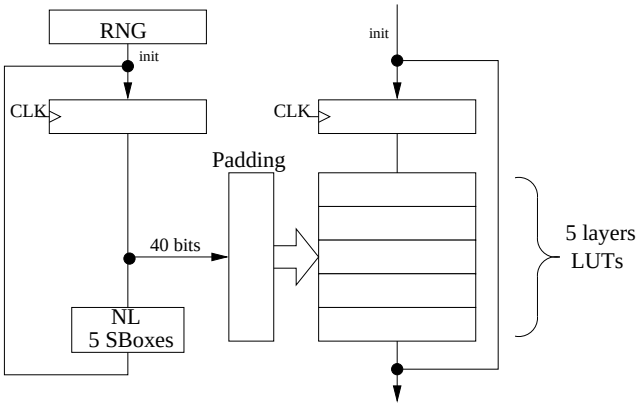

Fig. 5. RNG processing Architecture.

If the clock frequency f_{clock} is 50 MHz the TRNG data rate is of $40 \times \frac{f_{clock}}{17} = 118Mbps$. The number 17 comes from the 16 rounds plus the final Noekeon step. Fast TRNGs like [8] can then be used for dynamic implementations of Noekeon.

5 Conclusion

In this paper, we address the challenge of how to protect the content of a block cipher against reverse engineering by side-channel attacks. To this end, we use the look-up tables representation which prevail in the domain of white-box cryptography. When these tables are periodically renewed - while keeping the same functionality - we can reduce to few the information given to an adversary about the underlying algorithm. We describe a first proof of concept of our ideas as well as a preliminary implementation feasibility inside a FPGA of the block cipher Nokeon. The strategy could

be applied to other similar block ciphers as well. Nonetheless, the impact on a best-known cipher needs to be evaluated.

The look-up table renewal strategy based on TRNG is a key protection feature. Some more work is still needed to validate the exact implementation cost and tune the level of protection provided by our solution according to the complexity. Other ways as on the fly FPGA reconfiguration in XILINX FPGAs could also be investigated. In particular, the impacts on the efficiency and the exact overhead imposed by our dynamic reconfiguration must be measured on real implementations. Further works include also SCA through experiments to verify the efficiency of the countermeasure and to compare it to existing counter measures against SCARE attacks.

References

1. Olivier Billet, Henri Gilbert, and Charaf Ech-Chatbi. Cryptanalysis of a White Box AES Implementation. In Helena Handschuh and M. Anwar Hasan, editors, *Selected Areas in Cryptography*, volume 3357 of *LNCS*, pages 227–240. Springer, 2004.
2. Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. A White-Box DES Implementation for DRM Applications. In *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002*, volume 2696 of *LNCS*, pages 1–15. Springer, 2002.
3. Stanley Chow, Philip A. Eisen, Harold Johnson, and Paul C. van Oorschot. White-Box Cryptography and an AES Implementation. In Kaisa Nyberg and Howard M. Heys, editors, *Selected Areas in Cryptography*, volume 2595 of *LNCS*, pages 250–270. Springer, 2002.
4. Christophe Clavier. An Improved SCARE Cryptanalysis Against a Secret A3/A8 GSM Algorithm. In *ICISS*, volume 4812 of *LNCS*, pages 143–155. Springer, 2007.
5. Jean-Sébastien Coron. A New DPA Countermeasure Based on Permutation Tables. In *SCN'08*, volume 5229 of *LNCS*, pages 278–292.
6. Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie Proposal: NOEKEON, 2000.
7. Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. On Noekeon, no!, 2001.
8. J-L Danger, S. Guilley, and P. Hoogvorst. High speed true random number generator based on open loop structures in FPGAs. *Elsevier Microelectronics Journal*, doi: 10.1016/j.mejo.2009.02.004 2009.
9. Rémy Daudigny, Hervé Ledig, Frédéric Muller, and Frédéric Valette. Scare of the des. In *ACNS*, pages 393–406, 2005.
10. Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of White Box DES Implementations. In *Selected Areas in Cryptography, 14th International Workshop, SAC 2007*, volume 4876 of *LNCS*, pages 278–295. Springer, 2007.
11. S. Guilley, Ph. Hoogvorst, and R. Pacalet. Differential Power Analysis Model and some Results. In *Proceedings of WCC/CARDIS*, pages 127–142, 2004.

12. Matthias Jacob, Dan Boneh, and Edward W. Felten. Attacking an Obfuscated Cipher by Injecting Faults. In *Security and Privacy in Digital Rights Management, ACM CCS-9 Workshop, DRM 2002*, volume 2696 of *LNCS*, pages 16–31. Springer, 2002.
13. Marc Joye, Pascal Paillier, and Berry Schoenmakers. On second-order differential power analysis. In Josyula R. Rao and Berk Sunar, editors, *CHES*, volume 3659 of *LNCS*, pages 293–308. Springer, 2005.
14. Lars R. Knudsen and Havard Raddum. On Noekeon, 2001.
15. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages 388–397.
16. Hamilton E. Link and William D. Neumann. Clarifying Obfuscation: Improving the Security of White-Box DES. In *ITCC (1)*, pages 679–684. IEEE Computer Society, 2005.
17. Thomas S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In *CHES*, LNCS, pages 238–251. Springer-Verlag, 2000.
18. Wil Michiels, Paul Gorissen, and Henk D.L. Hollmann. Cryptanalysis of a Generic Class of White-Box Implementations. In *Selected Areas in Cryptography 2008, 15th International Workshop, SAC 2008*, 2008.
19. Roman Novak. Side-channel attack on substitution blocks. In *ACNS*, pages 307–318, 2003.
20. Denis Réal, Vivien Dubois, Anne-Marie Guilloux, Frédéric Valette, and M'hamed Drissi. Scare of an unknown hardware feistel implementation. In *CARDIS*, pages 218–227, 2008.
21. Altera FPGA designer: `http://www.altera.com/`.
22. Lattice FPGA designer: `http://www.latticesemi.com/`.
23. Xilinx FPGA designer: `http://www.xilinx.com/`.
24. Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings. In *Selected Areas in Cryptography, 14th International Workshop, SAC 2007*, volume 4876 of *LNCS*, pages 264–277. Springer, 2007.