White-Box Cryptography to Counteract SCARE

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

[1] Sagem Sécurité
[2] Télécom ParisTech

Abstract. 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 this perspective, but one step further. We consider that the adversary has only access to the cryptographic device through its side channels and his goal is to recover the algorithm specifications. Moreover, in this work, we focus on hardware implementations.
As a proof of concept, we here examine how to thwart Side Channel Analysis for Reverse Engineering (SCARE) attacks by implementing a block cipher following white-box techniques. We explain that with our proposal no details on the running symmetric cipher is revealed. Our proposal is illustrated by an example on Noekeon cipher and with the study of implementation feasibility on Field-Programmable Gate Arrays (FPGA).

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

1 Introduction

White-box cryptography has been introduced in the domain of DRM with the ambitious goal of protecting keys of a block cipher while letting the whole access to the software implementing this algorithm to an adversary. 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 [3, 4]. 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 had been big cryptanalytic efforts on many implementations (naked or not) [2, 11, 13, 17, 19, 25].

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 [1]). 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 Attacks Reverse Engineering (SCARE).

SCARE has been introduced in [20] with a proprietary algorithm for GSM phone. These results have been improved by [5]. [10] studies DES in this context and is extended to Unknown Hardware Feistel Implementation by [21]. Use of proprietary algorithms and the protection of their specifications can be conceived in organizations which can afford the risk of relying on secret algorithms such as military groups or pay-tv / mobile network operators.

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 can reduce the side channels information on the specifications of the running block cipher to almost nothing. We also want to show that our proposal can be implemented on today chips and we illustrate this aspect with the cipher Noekeon [7].

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.

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 tables 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 of several tables at one time 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 distinguish by Simple Power Analysis and the resistance against Differential Power Analysis (DPA) is improved when the size of tables decreased. Indeed the size of the tables has an effect onto the DPA signal-to-noise ratio; this is illustrated for instance by [12].

2.1 Model

We detail here our security model and the corresponding assumptions.

Following the previous remark, we consider that an adversary cannot obtain information on a look-up table neither by reading directly on it nor by measuring some signals during its execution. That is 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 [16, 18] for High-Order DPA, see also [14]) 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 [19], 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 [6] 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 $f_{l,in}[t'] = f_{l,in}[t] \circ g_{l,in}$. We assume that the renewal process does not leak. We assume moreover that every choice of input or output encodings is made independently except if there is an arc between two tables imposing the next encoding is the inverse of the previous one. More generally, we say that two tables are correlated if the same input (resp. output) encoding is used at the input (resp. output) of these tables because these tables are related to a common table by an arc. In case of such correlated encodings, we consider that implementation is arranged so that the correlated tables – those either with a same input encoding, or with a same output encoding – are evaluated in parallel during the same step.

Therefore, any intermediate value is of the form $f_{l,out}[t](x)$ where the $f_{l,out}[t]$ are different from one table to other ones as soon as they are not related to a common next input by an arc. This last assumption holds as the correlated tables (if any) are executed in a same step. As each $f_{l,out}$ varies randomly after each execution, this leads to uniformity. This would lead to the following resistance property against side channel analysis:

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, then the resistance holds against a generalized HO-SCA adversary.

3 Noekeon Cipher [7]

We give here an overview of the Noekeon cipher. In the next section, we describe a practical application of our idea to it.

Noekeon has been proposed to the NESSIE project in 2000 [7, 8, 15].

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, two vectors of four 32-bit words, the computation of $\theta(k, a)$ is illustrated by Table 1.

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

$$
\begin{array}{l}
temp \leftarrow a_0 \oplus a_2;\ temp \leftarrow temp \oplus (temp >> 8) \oplus (temp << 8);\\
a_1 \leftarrow a_1 \oplus temp;\\
a_3 \leftarrow a_3 \oplus temp;\\
a_0 \leftarrow a_0 \oplus k_0;\ a_1 \leftarrow a_1 \oplus k_1;\ a_2 \leftarrow a_2 \oplus k_2;\ a_3 \leftarrow a_3 \oplus k_3;\\
temp \leftarrow a_1 \oplus a_3;\ temp \leftarrow temp \oplus (temp >> 8) \oplus (temp << 8);\\
a_0 \leftarrow a_0 \oplus temp;\\
a_2 \leftarrow a_2 \oplus temp;
\end{array}
$$

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 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.

Roughly speaking, each round of the cipher can be decomposed in a non-linear step Γ and some 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' for the final step (the final constant XOR and the application of θ).

4 A White-Box Implementation of Noekeon

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):

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

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 8Bytes, and each XOR are seen as 8×4 look-up tables, for a size of 128Bytes. Moreover, the same input/decoding strategy as for γ is respected, i.e. any 4×4 look-up table T is encoded as $f \circ T \circ g$ for some random bijections f, g over nibbles and any 8×4 look-up table T is encoded as $f \circ T \circ (g', g'')$ for some random

bijections f, g', g'' over nibbles. It leads to three 4×4 encoded look-up tables and two different 8×4 encoded look-up tables (one for each XOR). Thus, one 4×12 binary matrix is implemented on 280Bytes.

By applying this method to all the U_j^i ($i \in \{0, \ldots, 31\}, j \in \{1, \ldots, 16\} \cup \{'final'\}$), we obtain $32 \times 17 \times 5$ look-up tables for an overall size of $32 \times 17 \times 280$Bytes, i.e. 152 320 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 [4] 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 $3 \times 3 = 9$ encoded look-up tables of size 4×4 and $3 \times 2 = 6$ encoded XOR look-up tables (840Bytes).

With these representations of the matrices $M_1, \ldots, M_{16}$, M', Γ and hardwired π_1 and π_2, the total number of look-up tables is $16 \times 32 + 17 \times 32 \times 5 + 17 \times 32 \times 3 \times 5 = 11392$ for an overall size of 599 kBytes.

Remark 2. In the section 4.3, another architecture with 16×16 matrices is also explained to optimize further the implementation and the independence of the 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 $j \in \{2,\ldots,16\} \cup \{'final'\}$). 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 $U_j^0, \ldots, U_j^{31}$ 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 by $f_j^0(x) = x \oplus (c_{g_j^0,0}, c_{g_j^{31},1}, c_{g_j^{27},2}, c_{g_j^{30},3})$.

In the next section, we consider that encodings for inputs of 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 is sufficient to this purpose. Moreover, this simplifies the renewal (cf. end of section 4.3).

4.3 Implementation Complexity in FPGAs

The Noekeon ciphering is based on nibble calculations, a nibble being a four-bit bundle. Therefore the FPGA architectures based on 4-bit look-up tables (LUT4) are well suited to implement the Noekeon algorithm. For instance the π_1, Γ and π_2 functions can be implemented by 32 4×4 tables for each round. A table being a set of four FPGA LUT4s, there is a need of 32×16×4=**2048** LUT4s for the whole functions (π_1, Γ and π_2).

Architecture of the M_j and M' A first implementation consists in considering 32 parallel computations of nibbles at each round as explained in section 4.1. Hence each M_j function, for $j \in \{1,\ldots,16\}$, or M' is implemented by means of 32 basic blocks U_j^i ($i \in \{1,\ldots,32\}$) having the

same architecture. Figure 1 represents the basic block which computes one nibble from 3 input nibbles.

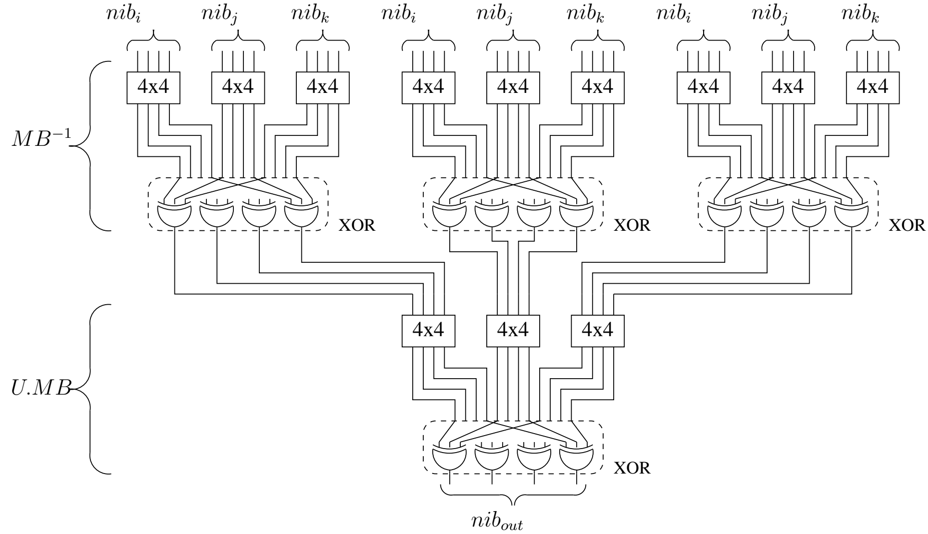

Fig. 1. Nibble computation block for U^i_j.

This architecture is very regular as it is composed of four identical structures. This structure corresponds to 12×4 tables composed of three 4×4 tables whose output are XORed to form the outputs of $(MB^i_j)^{-1}$ or output of $U^i_j.MB^i_j$

Each 4×4 table and group of four three-input XOR need four LUT4s. This gives a total of 64 LUT4s for each nibble and 17×32×64=**34816** LUT4s for the whole M_j and M' functions.

This architecture can be optimized by considering eight parallel processings of four nibbles. Each bundle is composed of four nibbles

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

and can be calculated from the same bundle at the input. Therefore the architecture can use 16×16 tables composed of 16×4 tables based on four 4×4 tables as illustrated by Figure 2. Another 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).

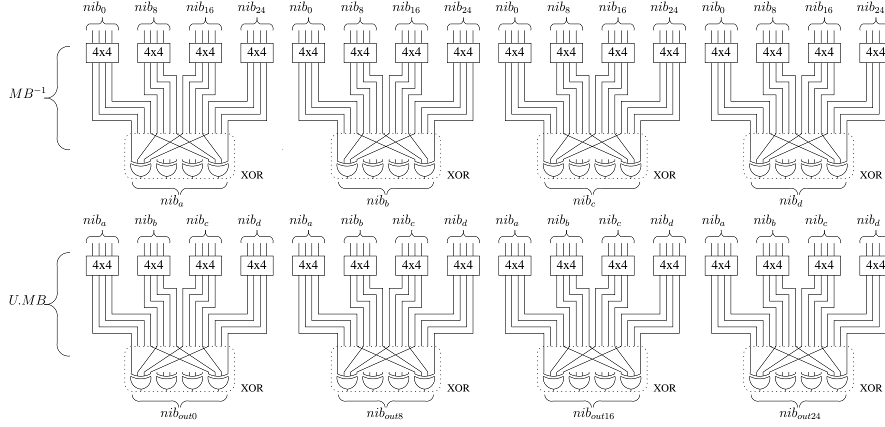

Fig. 2. four-Nibble computation block for U_j^i.

This architecture is made up of two identical structures corresponding to $(MB_j^i)^{-1}$ and $U_j^i.MB_j^i$. Each 4×4 table and group of four four-input XOR can be implemented with four LUT4s. This gives a total of 160 LUT4s for each bundle of nibbles and 17×8×160=**21760** LUT4s for the whole M_j and M' functions.

By including the π_1, Γ and π_2 functions the implementation requires **23808** LUT4s for the whole Noekeon implementation.

Feasibility in Current FPGAs Most SRAM-based FPGAS [22–24] 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 Adaptative LUT Module 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 biggest SRAM-based FPGA devices. Columns **4×4**, **16×4** and **TOTAL** indicate respectively the number of cells for the 4×4 table, the 16×4 table and the total number of cells for the proposed Noekeon implementation. In the rightmost column the minimum occupancy rate is indicated for the biggest devices. It remains relatively low and proves the feasibility in most devices.

Encoding Renewal The tables have to be updated periodically in order to change the masking functions $f_{l,in}$ and $f_{l,out}$ (cf. section 2). Ideally they have to be changed after each ciphering. In FPGA a solution could be to

Table 2. Occupation rate

Device	cell type	4×4	16×4	TOTAL	min cell	max cell	min occ.
CYCLONE II	LUT4	4	20	23808	4608	68416	34.8%
CYCLONE III	LUT4	4	20	23808	5136	119088	20%
STRATIX	LUT4	4	20	23808	10570	79040	30.1%
STRATIX II	ALM	2	10	11904	6240	71760	16.6%
STRATIX III	ALM	2	10	11904	19000	135200	8.8%
STRATIX IV	ALM	2	10	11904	91200	212480	5.6%
LATTICE ECP2	LUT4	4	20	23808	6000	68000	35%
LATTICE SC/M	LUT4	4	20	23808	15000	115000	20.7%
SPARTAN3	LUT4	4	20	23808	1728	74880	31.8%
VIRTEX2 PRO	LUT4	4	20	23808	3168	99216	24%
VIRTEX4	LUT4	4	20	23808	13824	200448	11.9%
VIRTEX5	LUT6	2	10	11904	12480	207360	5.7%

reconfigure it either completely or partially (only for the XILINX devices) but the time needed for reconfiguration would reduce significantly the ciphering rate. Another solution is to take advantage of LUT4 which are configurable in distributed memory. This feature is available in LATTICE and XILINX devices. In this case the LUT4 can be dynamically changed by the FPGA itself. For this, the FPGA should embed a fast True Random Number Generator (TRNG), as e.g. the one presented in [9], to generate new $f_{l,in}$ and $f_{l,out}$, and calculate the inverse functions (note that the latter operation is quite direct for encodings which consist of padding). At each cycle, one word of all the tables except the ones of index j being read at round j, can be written. As Noekeon has 17 rounds and all the tables have 16 words, almost all the tables can be updated during one ciphering.

5 Conclusion

In this paper, we show 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 proof of concept of our ideas as well as the implementation feasibility inside a FPGA of the block cipher Nokeon.

Some more work is still needed to tune the overall level of protection provided by our solution to keep good encryption performances. The look-up table renewal strategy is a key protection feature. Its implementation will be investigated in future works by using for instance on the fly FPGA reconfiguration. We in fact want to think in the future at our proposal in terms of System On Chip.

References

1. The Side Channel Cryptanalysis Lounge. Website: `http://www.crypto.ruhr-uni-bochum.de/en_sclounge.html`.
2. 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 *Lecture Notes in Computer Science*, pages 227–240. Springer, 2004.
3. 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, Washington, DC, USA, November 18, 2002, Revised Papers*, volume 2696 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2002.
4. 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 *Lecture Notes in Computer Science*, pages 250–270. Springer, 2002.
5. Christophe Clavier. An Improved SCARE Cryptanalysis Against a Secret A3/A8 GSM Algorithm. In Patrick Drew McDaniel and Shyam K. Gupta, editors, *ICISS*, volume 4812 of *Lecture Notes in Computer Science*, pages 143–155. Springer, 2007.
6. Jean-Sébastien Coron. A New DPA Countermeasure Based on Permutation Tables. In *SCN*, volume 5229 of *Lecture Notes in Computer Science*, pages 278–292. Springer, September 2008. Amalfi, Italy.
7. Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie Proposal: NOEKEON, 2000.
8. Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. On Noekeon, no!, 2001.
9. J-L Danger, S. Guilley, and P. Hoogvorst. High speed true random number generator based on open loop structures in fpgas. *Elsevier Microelectronics Journal*, to be published 2009.
10. Rémy Daudigny, Hervé Ledig, Frédéric Muller, and Frédéric Valette. SCARE of the DES. In John Ioannidis, Angelos D. Keromytis, and Moti Yung, editors, *ACNS*, volume 3531 of *Lecture Notes in Computer Science*, pages 393–406, 2005.
11. Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of White Box DES Implementations. In *Selected Areas in Cryptography, 14th International Workshop, SAC 2007, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*, volume 4876 of *Lecture Notes in Computer Science*, pages 278–295. Springer, 2007.
12. S. Guilley, Ph. Hoogvorst, and R. Pacalet. Differential Power Analysis Model and some Results. In *Proceedings of WCC/CARDIS*, pages 127–142, Aug 2004. Toulouse, France.

13. 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, Washington, DC, USA, November 18, 2002, Revised Papers*, volume 2696 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2002.
14. 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 *Lecture Notes in Computer Science*, pages 293–308. Springer, 2005.
15. Lars R. Knudsen and Havard Raddum. On Noekeon, 2001.
16. P. Kocher, J. Jaffe, and B. Jun. Differential Power Analysis. In *CRYPTO*, volume 1666 of *LNCS*, pages pp 388–397. Springer, 1999.
17. 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.
18. Thomas S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. In *CHES*, LNCS, pages 238–251. Springer-Verlag, 2000.
19. Wil Michiels, Paul Gorissen, and Henk D.L. Hollmann. Cryptanalysis of a Generic Class of White-Box Implementations, 2008. To appear in the proceedings of Selected Areas in Cryptography 2008, 15th International Workshop, SAC 2008.
20. Roman Novak. Side-Channel Attack on Substitution Blocks. In Jianying Zhou, Moti Yung, and Yongfei Han, editors, *ACNS*, volume 2846 of *Lecture Notes in Computer Science*, pages 307–318. Springer, 2003.
21. Denis Réal, Vivien Dubois, Anne-Marie Guilloux, Frédéric Valette, and M'hamed Drissi. SCARE of an Unknown Hardware Feistel Implementation. In Gilles Grimaud and François-Xavier Standaert, editors, *CARDIS*, volume 5189 of *Lecture Notes in Computer Science*, pages 218–227. Springer, 2008.
22. Altera FPGA designer: `http://www.altera.com/`.
23. Lattice FPGA designer: `http://www.latticesemi.com/`.
24. Xilinx FPGA designer: `http://www.xilinx.com/`.
25. 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, Ottawa, Canada, August 16-17, 2007, Revised Selected Papers*, volume 4876 of *Lecture Notes in Computer Science*, pages 264–277. Springer, 2007.