On Physical Obfuscation of Cryptographic Algorithms*

Julien Bringer[1], Hervé Chabanne[1,2], and Thomas Icart[1,3]

[1] Sagem Sécurité
[2] Télécom ParisTech
[3] Université du Luxembourg
`firstname.name@sagem.com`

Abstract. We describe a solution for physically obfuscating the representation of a cipher, to augment chips resistance against physical threats, by combining ideas from masking techniques and Physical Obfuscated Keys (POKs). With embedded chips – like RFID tags – as main motivation, we apply this strategy to the representation of a Linear Feedback Shift Register (LFSR).

The application of this technique to LFSR-based stream ciphers, such as the Self Shrinking Generator, enables to share key materials between several chips within a system while increasing the resistance of the system against compromise of chips. An extension of our ideas to non-linear ciphers is also presented with an illustration onto Trivium.

Keywords: RFID tags, POK, PUF, masking, stream ciphers.

1 Introduction

Physical Obfuscated Keys (POK) [14] are a means to store keys inside an Integrated Circuit (Section 2). Their use is based on the paradigm that an unauthorized access to a value represented by a POK will affect the behavior of this POK and make it non-operational. This way, when an adversary compromises a chip to read a key, he will not be able to use the same POK again. In particular, when different values are represented by the same POK, a compromise at some time will render activation of further values impossible. This type of situation has been considered in [14] with a general line of defense for POKs: split the computations with a key K in two steps, one related to a random key K' and the other one to another key K'' where the pair (K', K'') depends solely on the chip implementing the POK. Doing that, when a chip is tampered with, this will not allow an adversary to recover the key K or to interfere on the value of K contained in another chip. Here the difficulty is to find a way to split the cryptographic computations. This is illustrated with public key encryption schemes based on exponentiation by the key in [14]. [7] describes the modification of an existing protocol relying on an XOR with a key to incorporate POKs' trick.

* This work has been partially funded by the ANR T2TIT project.

In the paper, we extend this idea by combining physical obfuscated keys and classical masking techniques, similar to that used to counter Side Channel Analysis (SCA) attacks, to construct physically obfuscated ciphers. Indeed splitting the computation in several steps is essentially the goal of masking techniques or masked logic to thwart SCA attacks (see for instance [29, 3] for masked AND and XOR applied to the AES). And we illustrate here how this strategy can be employed successfully via POKs for obfuscating the secret material of a cipher. Some other related techniques are secret sharing techniques [31].

As a proof-of-concept, we apply this strategy to the general case of linear feedback shift registers (LFSR). LFSRs are easy to design and to implement with low hardware requirements. The operations of a LFSR are deterministic, which means that a polynomial and a state completely determine the next output values. For a system with shared materials between several tags, the use of the same LFSR and initial value, for instance to generate a key stream, would thus face the problem of resistance of tags against compromise: the opening of one tag gives the possibility to know the key stream of other tags. We explain here how to hide the value of the state and the polynomial during the execution by implementing the operations with POKs.

Our main achievement is to show that it is possible to hide their content and their connections by making use of POKs (Section 3). As an immediate application of our proposal, we introduce an implementation of LFSR-based hashing for message authentication [23] (Section 4.1). As LFSR is a very popular primitive in the design of stream-ciphers, we also give examples in this context and explain how POKs can be adapted to handle some small non-linear operations. As a relevant example, we give details of an obfuscated version of the Self-Shrinking Generator [27] (Section 4.2). Finally, we modify further our techniques to be able to protect the Trivium stream-cipher [8] with POKs (Section 4.3). Moreover our strategy is quite general and can be applied to other ciphers.

To conclude, we want to stress the fact that POKs do not make use of memories to store keys and need only few hardware resources to be implemented. They are well-suited for very constrained chips or those only allowing a small amount of their capacity to cryptographic computations. Our constructions show that they can also provide some inherent resistance against tampering. We thus think that RFID tags are targets of choice for implementing the results of our paper.

2 Physically Obfuscated Key

In [14], it is shown how to implement a key with a Physical Unclonable Function (PUF) by applying a fixed hard-wired challenge to the PUF (cf. Appendix A for description of the notion of PUF); this implementation is called a Physically Obfuscated Key (POK). In fact, using different challenges, several POKs can be obtained from one PUF. In the sequel, we refer to a POK as a value, stored in a tag, which is accessible only after the underlying PUF is stimulated; once erased from volatile memory, the value of the key is no longer available.

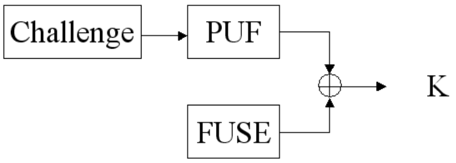

Fig. 1. Example of POK

To be able to set a POK to a chosen value, the output of the PUF – which cannot be chosen – can be combined to some fuse (or any data stored in the EEPROM) via an exclusive-or operation (Figure 1). In the sequel, we assume that any implemented POK is obtained via an I-PUF (cf. Appendix A), which gives us the following property.

Property 1. Any corruption of or intrusion into the chip leads to the end-of-life of the chip: an adversary could not continue to use the chip, particularly he can only obtain information on the content of the memory at this time and could not access the others POK*s* (if any) implemented in it.

More generally, inside a chip, we say that a primitive is **physically obfuscated** when it is implemented via some POKs so that an adversary could not learn any secret information from an intermediate result obtained by opening the chip.

3 Physically Obfuscated Linear Feedback Shift Register

A linear feedback shift register (LFSR) is a binary shift register where the update of the input bits are made via a linear function of the previous ones. The initial value of a LFSR is called the seed, the current input bits in the register are the current state and the update of the state is the result of the evaluation of a feedback function – represented by a so-called feedback polynomial – corresponding to exclusive-or of some bits at given positions (the connecting taps) of the state.

We consider here a linear feedback shift register (LFSR) of length L with a feedback polynomial $P \in GF(2)[X]$ of degree L. The polynomial is often chosen to be primitive to obtain a LFSR of maximal period $2^L - 1$. This avoids the occurrence of repeating cycle after a short delay. Let $P = \sum_{k=0}^{L} a_k X^k$, let $S^0 = (s_0, \ldots, s_{L-1})$ be the initial state of the LFSR. Then the next state is $S^1 = (s_1, \ldots, s_{L-1}, s_L)$ where s_L is the value $a_0 s_0 \oplus \cdots \oplus a_{L-1} s_{L-1}$, and s_0 is outputted. More generally, S^n denotes the n-th state.

3.1 Obfuscation of Basic Operations

Let $l \geq 1$. For $i \in \{1, 2, 3\}$, let $K[i]$ be a l-bit vector, implemented by two POKs $K[i]'$, $K[i]''$ such that $K[i] = K[i]' \oplus K[i]''$. Let x and y be two l-bit vectors.

Definition 1. *A physically obfuscated XOR, denoted by* $_{\mathsf{PO}}\mathsf{XOR}$*, corresponds to the computation of a masked XOR of two masked inputs.* $_{\mathsf{PO}}\mathsf{XOR}_l(x \oplus K[1], y \oplus$

Table 1. $_{\mathsf{PO}}\mathsf{XOR}_l(x \oplus K[1], y \oplus K[2], K[1], K[2], K[3])$ implementation

```
1. Set $z = (x \oplus K[1]) \oplus (y \oplus K[2]) = (x \oplus y) \oplus (K[1] \oplus K[2])$
2. For $i = 1$ to 3 do
      activate $K[i]'$, update $z \leftarrow z \oplus K[i]'$, erase $K[i]'$ from memory
End For
3. For $i = 1$ to 3 do
      activate $K[i]''$, update $z \leftarrow z \oplus K[i]''$, erase $K[i]''$ from memory
End For
4. Output $z$
```

$K[2], K[1], K[2], K[3])$ *is the computation of* $x \oplus y \oplus K[3]$ *with the inputs* $x \oplus K[1]$ *and* $y \oplus K[2]$ *and is implemented as in Table 1.*

It is straightforward to check that $_{\mathsf{PO}}\mathsf{XOR}_l(x \oplus K[1], y \oplus K[2], K[1], K[2], K[3])$ outputs the desired value, $x \oplus y \oplus K[3]$. Moreover, thanks to the POKs Property 1 (see page 90), we ensure the physical obfuscation of the XOR.

Lemma 1. $_{\mathsf{PO}}\mathsf{XOR}$ *is a physically obfuscated primitive.*

Proof. (sketch) The implementation of $_{\mathsf{PO}}\mathsf{XOR}$ does not leak information to an adversary on $x, y, x \oplus y, K[1], K[2]$ or $K[3]$. Indeed, recall that an adversary $\mathcal{A}$ can eavesdrop on the memory only once before destroying the chip. We assume that $\mathcal{A}$ already knows the inputs $x \oplus K[1]$ and $y \oplus K[2]$ and the output $x \oplus y \oplus K[3]$. If he corrupted the chip in step 1, he would learn nothing more. If he corrupted the chip during the step 2, say when $i = 2$, he would learn $K[2]'$ and $(x \oplus y) \oplus (K[1]'' \oplus K[2]'')$. If he corrupted the chip during the step 3, say whence $i = 1$, he would learn $K[1]''$ and $(x \oplus y) \oplus K[2]'' \oplus K[3]'$. In any case, $\mathcal{A}$ does not gain information on the un-masked result $x \oplus y$ or on the un-masked inputs x, y. □

From $_{\mathsf{PO}}\mathsf{XOR}$, we deduce another interesting physically obfuscated operation, $_{\mathsf{PO}}\mathsf{Convert}$, which converts the mask of a physically masked value into another mask. $_{\mathsf{PO}}\mathsf{Convert}_l(x \oplus K[1], K[1], K[3])$ takes as input $x \oplus K[1]$ and outputs $x \oplus K[3]$; it is implemented via $_{\mathsf{PO}}\mathsf{XOR}_l(x \oplus K[1], 0, K[1], 0, K[3])$. And the property of lemma 1 holds.

Let now $K[3]$ be restricted to a 1-bit vector.

Definition 2. *We define the function* $_{\mathsf{SemiPO}}\mathsf{Scalar}$ *as the masked scalar product of a first non-masked input with a second masked input (the term* Semi *underlines this asymmetry).* $_{\mathsf{SemiPO}}\mathsf{Scalar}_l(x, y \oplus K[2], K[2], K[3])$ *is the computation of* $(x \cdot y) \oplus K[3]$ *with the inputs* x *and* $y \oplus K[2]$ *(cf. Table 2).*

As for $_{\mathsf{PO}}\mathsf{XOR}$, this implementation with sequential activation of the POKs implies that $_{\mathsf{SemiPO}}\mathsf{Scalar}$ is physically obfuscated: no information on $y, K[2], K[3]$ or $(x \cdot y)$ are leaked.

Lemma 2. $_{\mathsf{SemiPO}}\mathsf{Scalar}$ *is a physically obfuscated primitive.*

Table 2. $_{\mathsf{SemiPO}}\mathsf{Scalar}_l(x, y \oplus K[2], K[2], K[3])$ implementation

```
1. Set $z = x \cdot (y \oplus K[2])$
2. Activate $K[2]'$, update $z \leftarrow z \oplus (x \cdot K[2]')$, erase $K[2]'$ from memory
3. Activate $K[3]'$, update $z \leftarrow z \oplus K[3]'$, erase $K[3]'$ from memory
4. Activate $K[2]''$, update $z \leftarrow z \oplus (x \cdot K[2]'')$, erase $K[2]''$ from memory
5. Activate $K[3]''$, update $z \leftarrow z \oplus K[3]''$, erase $K[3]''$ from memory
6. Output $z$
```

For $l = 1$, $_{\mathsf{SemiPO}}\mathsf{Scalar}$ corresponds to a AND operator. Here only one input can be masked; for both inputs to be masked, i.e. for a general obfuscated scalar product, we need a slightly more complex implementation as the operations related to the POKs are not linear anymore. This is illustrated on the AND operator in section 4.3.

For all above primitives, note that no mask ($KS[3]$) needs to be applied to the output whence the latter does not need to be protected. In that case, it does not alter the physical obfuscation of the others values (un-masked inputs, $K[1]$ and $K[2]$).

3.2 Obfuscating the Taps

We now represent the operation of the feedback polynomial $P = \sum_{k=0}^{L} a_k X^k$ as a scalar product by its coefficients, $s_{n+L} = S^n \cdot KF$, with $KF = (a_0, \ldots, a_{L-1})$ and $S^n = (s_n, \ldots, s_{n+L-1})$. KF can be seen as a feedback key and for some cryptographic primitives (cf. section 4) we want to thwart an adversary to recover it by opening a tag.

Let KF' be a random L-bit vector and $KF'' = KF \oplus KF'$; we also assume that KF', KF'' are implementing as physically obfuscated keys (POKs). The computation of s_{n+L} is thus seen as

$$s_{n+L} = {}_{\mathsf{SemiPO}}\mathsf{Scalar}_L(S^n, 0, KF, 0).$$

In contrast to the general use of $_{\mathsf{SemiPO}}\mathsf{Scalar}$ in section 3.1, the output is not masked here; only KF is to be kept obfuscated during execution. In addition to the value of s_{n+L}, an adversary who opens a tag during execution only learns information either on KF' or KF'', but not both at the same time, thanks to the POKs property 1. As the separation of KF into $KF' \oplus KF''$ can be made different for each tag, he cannot recover KF from this information.

However from the knowledge of the value of s_{n+L}, he gains some information on KF if he also knows the value of S^n. And from about L values S^{n_i} and s_{n_i+L} obtained by opening as many tags sharing the same KF, it is easy to recover KF by solving a linear system. This issue is addressed in the sequel.

3.3 Towards Obfuscating the Taps and the State Simultaneously

To hide the state during execution of the register, we introduce another key KS, called key state, with the intended goal to manage the state S masked by the key $KS = (KS_0, \ldots, KS_{L-1})$ without letting the state appearing in clear. Here the key KS can be different from one tag to another. It is the state S which may be shared and consequently has to be protected, in particular if the initial state corresponds to a shared key for a set of tags within the system.

Rather than the state S^n, we store the value $M^n = S^n \oplus KS$ and want to update the register directly via the masked state M^n and the feedback key KF. We think again of the $_{\mathsf{SemiPO}}\mathsf{Scalar}$ solution, but as explained in section 3.1, it is not straightforward to apply it when both inputs are masked. Here, we choose KS such that $KS \cdot KF = 0$ which leads to the simplification $M^n \cdot KF = (S^n \oplus KS) \cdot KF = S^n \cdot KF$. To enable the update of the masked register, we split the key KS into two POKs as in the previous section for KF. Let KS' be a random L-bit vector and $KS'' = KS \oplus KS'$. The operations of the previous section are completed as follows.

After the computation of $s_{n+L} = {}_{\mathsf{SemiPO}}\mathsf{Scalar}_L(M^n, 0, KF, 0)$, the register outputs $m_n = s_n \oplus KS_0$ and the state becomes

$$temp = ((s_{n+1} \oplus KS_1, \ldots, s_{n+L-1} \oplus KS_{L-1}, s_{n+L})).$$

Then, $_{\mathsf{PO}}\mathsf{Convert}_L(temp, K[1], K[3])$ is run to update $temp$ where $K[3] = KS$ and $K[1] = (KS_{1\ldots L-1}||0)$ the vector resulting from the concatenation of the bits $KS_1, \ldots, KS_{L-1}$ and the bit 0. Subsequently, the register state is updated as $M^{n+1} = temp$ which is equal to $S^{n+1} \oplus KS$.

This doing and thanks to the splitting of the operations with two POKs, the state is not available in clear for an adversary in the second step. Note that the value s_{n+L} is not hidden in the first step. In the next section, we fuse the two previous process to enable obfuscation of this value too.

3.4 Fill in the Gap

Given $2L$ consecutive bits of an outputted stream from an LFSR, it is known that one can reconstruct an LFSR by using the Berlekamp-Massey [26] algorithm which will produce the same stream. It emphasizes the interest to mask any bit of the state whence the tags share the same feedback function and the same initial state. We describe now the whole process which achieves obfuscation of the feedback key and the state, including the new input bits s_{n+L} and the outputted bits.

We assume below that within a tag the LFSR is used as a key stream generator to encrypt a message by xoring it. In the sequel, let x be the current bit to be encrypted and assume that the current masked state is $M^n = S^n \oplus KS = (s_n \oplus KS_0, s_{n+1} \oplus KS_1, \ldots, s_{n+L-1} \oplus KS_{L-1})$. Note that all bits KS_i of KS can be seen as well as a combination of two 1-bit POKs, $KS_i = KS'_i \oplus KS''_i$. The algorithm is split in consecutive steps as detailed in Table 3.

Lemma 3. *The LFSR implementation of Table 3 leads to a physically obfuscated primitive.*

Proof. (sketch) Assume that x is unknown by the adversary then all bit values of the state are always masked along the different steps, either by x, or a bit of

Table 3. A physically obfuscated LFSR implementation

1. Set $z = {}_{\mathsf{SemiPO}}\mathsf{Scalar}_L(M^n, 0, KF, KS_{L-1})$.
2. The register outputs $m_n = s_n \oplus KS_0$ and the state becomes $temp = (s_{n+1} \oplus KS_1, \ldots, s_{n+L-1} \oplus KS_{L-1}, z)$ with $z = s_{n+L} \oplus KS_{L-1}$.
3. Then set $y = x \oplus m_n = x \oplus s_n \oplus KS_0$, erase x from memory, and output ${}_{\mathsf{PO}}\mathsf{Convert}_1(y, KS_0, 0)$.
4. Finally, $temp \leftarrow {}_{\mathsf{PO}}\mathsf{Convert}_L(temp, K[1], K[3])$ is run to update the register with $M^{n+1} = temp$ where $K[1] = (KS_{1\ldots L-1}||0)$ and $K[3] = (KS_{0\ldots L-2}||0)$.

KS, KS' or KS''. The important point now is that all these values are different from one tag to another one, this means that even if an adversary succeeds in obtaining N consecutive masked values, say $s_{k+1} \oplus \alpha_{k+1}, \ldots, s_{k+N} \oplus \alpha_{k+N}$ of the state by opening several tags (at least N, as opening a tag implementing a POK implies its end-of-life prematurely), he cannot recover the value of the state thanks to the bitwise independence of the bits $\alpha_{k+1}, \ldots, \alpha_{k+N}$. □

Corollary 1. *The implementation above without the execution of the step 3 remains physically obfuscated, i.e. the state and the feedback key stay hidden; which implementation we denote by* ${}_{\mathsf{PO}}\mathsf{LFSRupdate}$.

4 Applications

4.1 Krawczyk's MACs and LFSR-Based Hashing

[23] describes an efficient construction for Message Authentication Codes relying on traditional hashing techniques. The basic idea is to use a family H of linear *hash* functions which map $\{0,1\}^m$ to $\{0,1\}^L$ in a balanced way. Interestingly, such hashing family can be constructed as LFSR-based hashing. See Appendix B for a quick description of the MAC mechanism and the related notions.

An efficient solution using multiplication by matrices provided in [23] is to use specific Toeplitz matrices which can be described by a LFSR. Let the LFSR be represented by its feedback polynomial P, a primitive polynomial over $GF(2)$ of degree L, and an initial state $S^0 = (s_0, \ldots, s_{L-1}) \neq 0$. Then $h_{P,S^0} \in H$ is defined by the linear combinations $h_{P,s}(X) = \bigoplus_{j=0}^{m-1} x_j.S^j$ where $X = (x_0, \ldots, x_{m-1})$ and S^j is the j-th state of the LFSR. This leads to an ϵ-balanced family H (see Definition 4 in Appendix B) for at least $\epsilon \leq \frac{m}{2^{L-1}}$ as proved by [23]. Moreover, a hash function h_{P,S^0} is easily implemented as the message authentication can be computed progressively with an accumulator register which is updated after each message bit: the implementation does not depend on the size m of X.

Let $X = (x_0, \ldots, x_{m-1})$ be the message to be authenticated. We can manage the computation of $h_{P,S^0}(x)$ in an obfuscated way thanks to the previous algorithm for LFSR obfuscation. All updates of the LFSR are made thanks to ${}_{\mathsf{PO}}\mathsf{LFSRupdate}$ (modification of the method of section 3.4 where the step 3

Table 4. A physically obfuscated LFSR-based hashing implementation

```
$counter \leftarrow 0$
$result \leftarrow (0, \ldots, 0)$
For $n = 0$ to $m - 1$ do
    If $(x_j == 1)$ then
        $result \leftarrow result \oplus M^n$
        $counter \leftarrow counter \oplus 1$
    End If
    execute $_{\mathsf{PO}}\mathsf{LFSRupdate}()$ to obtain $M^{n+1}$
    $n \leftarrow n + 1$
End For
If $(counter == 0)$ then Output $result$
    Else Output $_{\mathsf{PO}}\mathsf{Convert}_L(result, KS, 0)$
End If
```

is skipped). Let *result* be the variable which will correspond to the value of $h_{P,S^0}(x)$ at the end of the execution. Starting from the initial masked state $M^0 = S^0 \oplus KS$, we update the register $m-1$ times and before each clocking, we update the value of *result*. The execution is summarized in Table 4. All computations to obtain h_{P,S^0} are made directly on the masked states and if necessary (when the weight of x is odd) KS is used at the end to unmask the result. Thanks to lemma 1 and corollary 1, we have the following results.

Lemma 4. *The LFSR-based hashing implementation in Table 4 is a physically obfuscated primitive.*

Remark 1. This obfuscation can be for instance applied to the implementation of an authentication protocol which makes use of LFSR-based hashing. It would be a way to answer the possible weakness of use of LFSR in RFID tags as underlined in [13] where the LFSR feedback polynomial is assumed to be known as soon it is the same in all tags. With our obfuscation technique this is not anymore the case.

4.2 Self-shrinking Generator

The self-shrinking generator (SSG) [27] consists of one LFSR combined with a so-called shrinking function. Let KF be the feedback function of the LFSR and $S^0 = (s_0, \ldots, s_{L-1})$ be its initial state. The shrinking function $f : GF(2) \times GF(2) \rightarrow GF(2) \cup \{\epsilon\}$ is defined as follows: for $(x, y) \in GF(2) \times GF(2)$, $f(x, y) = y$ if $x = 1$, $f(x, y) = \epsilon$ if $x = 0$, which could be interpreted as $f(0, y)$ outputs nothing. Hence a given output stream $s_0, \ldots, s_{2N}$ of length $2N$ from the LFSR is split in N couples and the shrinking function acts on each of them to output at the end the bitstream $f(s_0, s_1) \ldots f(s_{2N-1}, s_{2N})$ of length $\leq N$. As empty output may appear, the exact length is in fact hard to know in advance.

The solutions described in section 3 can be applied to the LFSR of the self-shrinking generator, thus protecting the state and the feedback function[1] against an intrusive adversary. Nevertheless, one constraint arises with the use of the shrinking function on the output bits: the value of the output bit in the algorithm of section 3.4 can not be masked anymore by x in order to be able to compare it with 0 or 1, which leads to a potential source of leakage. We have to distinguish two situations:

- The tags share at most one of the following data – feedback key or (exclusive) initial state, which means that the opening of different tags will not give enough information to recover the shared data.
- The tags share both data, feedback key and initial state, and in that case, the leakage of the output bit can afford to an adversary the possibility to reconstruct the LFSR via Berlekamp-Massey, by opening many tags. To avoid this, we suggest below a small modification of the SSG.

Masked SSG. We consider the algorithm $_{\mathsf{PO}}\mathsf{LFSRupdate}$ of section 3.4 for the execution of the LFSR assuming that it outputs the value $m_n = s_n \oplus KS_0$. We operate two bits by two bits for the shrinking function. I.e. we use the output bits $m_n = s_n \oplus KS_0$ and $m_{n+1} = s_{n+1} \oplus KS_0$.

Let x be the current bit to be encrypted, i.e. to be xored with the keystream generated by the SSG. In our modification the shrinking check $s_n == 1$ is replaced by the check $s_n \oplus KS_0 == 1$ (cf. Table 5). Note that this modification does not change the standard analysis of SSG as KS_0 is a constant.

Table 5. A physically obfuscated SSG implementation

```
While $m_n \neq 1$
    execute $_{\mathsf{PO}}\mathsf{LFSRupdate}()$ twice to output two new bits $(m_n, m_{n+1})$ (where $n \leftarrow n+2$)
End While
Set $y = x \oplus m_{n+1}$
Output $_{\mathsf{PO}}\mathsf{Convert}_1(y, KS_0, 0)$ (i.e. $x \oplus s_{n+1}$).
```

Lemma 5. *The above implementation of our modification of the SSG is physically obfuscated.*

Remark 2. All LFSR-based stream ciphers are possible targets for our obfuscation method as soon as operations remain linear. For instance, it can be adapted for the shrinking generator [10] or for some generalizations of the self-shrinking generator [24, 25]. LFSR-based stream ciphers with irregular clocking are also good targets. These include as examples the Alternating Step Generator [18],

[1] Note that for analysis of the SSG security, it is generally assumed that the feedback polynomial is known; here we consider the case where the system may try to hide it too.

A5/1 [4], or W7 [33]. To activate the clocking of some registers, clocking bits at fixed position are used. If all initial data (states and feedback polynomials) are not shared between several tags then those bits may be managed unmasked (in some cases) to check whether a register might clock (e.g. for A5/1 this check is made by a majority vote between the values of 3 bits coming from the 3 registers of the cipher). If all data are shared, then to avoid the risk of compromise we can modify slightly the scheme by checking the clocking condition directly on the masked bits.

4.3 Trivium

Trivium [8, 9] is a stream cipher which has been elected as one of the three hardware oriented stream ciphers of the eStream project portfolio (`http://www.ecrypt.eu.org/stream/`). This is thus natural to consider it as a possible cipher for implementation into tags. Technically, it is not a linear LFSR-based stream cipher, but its structure remains quite simple and the small number of non-linear operations enables us to adapt our obfuscation technique.

Trivium is roughly a concatenation of 3 registers which are updated via quadratic feedback functions. It contains a 288-bit internal state $(s_0, \ldots, s_{287})$ and once initialized, the key stream generation of a bit y and the update of the state follow the algorithm in Table 6 (where $\otimes$ stands for a product of bits, i.e. a AND). Here, the feedback function is fixed, so we do not need to mask the feedback key KF in the same way as in section 3.2 but to simplify the analysis we can keep the method described in section 3.4 as a baseline. The method is similar for handling all the linear computations above. The only specificity concerns the steps (5), (6), (7) where a general obfuscated AND is needed.

AND obfuscation. Here we focus on an AND of two bits (this is easily generalizable to l-bit vectors). For $i \in \{1, 2, 3\}$, let $K[i]$ be a binary value, implemented by two POKs $K[i]'$, $K[i]''$ such that $K[i] = K[i]' \oplus K[i]''$. Compare to the XOR,

Table 6. Trivium key stream generation

$t_1 \leftarrow s_{65} \oplus s_{92}$	(1)
$t_2 \leftarrow s_{161} \oplus s_{176}$	(2)
$t_3 \leftarrow s_{242} \oplus s_{287}$	(3)
$y \leftarrow t_1 \oplus t_2 \oplus t_3$	(4)
$t_1 \leftarrow t_1 \oplus (s_{90} \otimes s_{91}) \oplus s_{170}$	(5)
$t_2 \leftarrow t_2 \oplus (s_{174} \otimes s_{175}) \oplus s_{263}$	(6)
$t_3 \leftarrow t_3 \oplus (s_{285} \otimes s_{286}) \oplus s_{68}$	(7)
$(s'_0, s'_1, \ldots, s'_{92}) \leftarrow (t_3, s_0, \ldots, s_{91})$	(8)
$(s'_{93}, s'_{94}, \ldots, s'_{176}) \leftarrow (t_1, s_{93}, \ldots, s_{175})$	(9)
$(s'_{177}, s'_{178}, \ldots, s'_{287}) \leftarrow (t_2, s_{177}, \ldots, s_{286})$	(10)

we also introduce a couple of POK*s* $K[4]'$, $K[4]''$. Also let x and y be two binary values.

Definition 3. *A physically obfuscated AND, denoted by* ${}_{\mathsf{PO}}\mathsf{AND}$, *corresponds to the computation of a masked AND of two masked bits.* ${}_{\mathsf{PO}}\mathsf{AND}(x \oplus K[1], y \oplus K[2], K[1], K[2], K[3], K[4])$ *is the computation of* $(x \otimes y) \oplus K[3]$ *with the inputs* $x \oplus K[1]$ *and* $y \oplus K[2]$ *and is implemented as in Table 7.*

The implementation is based on the following relation:

$$x \otimes y = \Big((x \oplus K[1]) \otimes (y \oplus K[2]) \Big) \oplus \Big(K[1] \otimes (y \oplus K[2]) \Big)$$
$$\oplus \Big((x \oplus K[1]) \otimes K[2] \Big) \oplus \Big(K[1] \otimes K[2] \Big)$$

Table 7. ${}_{\mathsf{PO}}\mathsf{AND}(x \oplus K[1], y \oplus K[2], K[1], K[2], K[3], K[4])$ implementation

```
1. Set $z = ((x \oplus K[1]) \otimes (y \oplus K[2]))$
2. $z \leftarrow z \oplus {}_{\mathsf{SemiPO}}\mathsf{Scalar}_1(x \oplus K[1], 0, K[2], 0) \oplus {}_{\mathsf{SemiPO}}\mathsf{Scalar}_1(y \oplus K[2], 0, K[1], 0)$
3. $z \leftarrow {}_{\mathsf{PO}}\mathsf{Convert}_1(z, 0, K[3])$
4. Activate $K[1]'$ and $K[2]'$, update $z \leftarrow z \oplus (K[1]' \otimes K[2]')$
5. Erase $K[1]'$ and $K[2]'$ from memory
6. Activate $K[1]''$ and $K[2]''$, update $z \leftarrow z \oplus (K[1]'' \otimes K[2]'')$
7. Activate $K[4]', K[4]''$, set $temp_1 = K[4]' \oplus K[1]''$ and $temp_2 = K[4]'' \oplus K[2]''$
8. Erase $K[1]''$, $K[2]''$, $K[4]'$ and $K[4]''$ from memory
9. Activate $K[1]', K[2]'$, update $temp_1 \leftarrow temp_1 \otimes K[2]'$, $temp_2 \leftarrow temp_2 \otimes K[1]'$
10. Update $z \leftarrow z \oplus temp_1 \oplus temp_2$, erase $temp_1, temp_2$ from memory
11. Activate $K[4]', K[4]''$, update $z \leftarrow z \oplus (K[4]' \otimes K[2]') \oplus (K[4]'' \otimes K[1]')$
12. Erase $K[4]', K[4]''$, $K[1]'$ and $K[2]'$ from memory
13. Output $z$
```

The steps 4 to 12 are used to compute $K[1] \otimes K[2]$ as the XOR of $K[1]' \otimes K[2]'$, $K[1]' \otimes K[2]''$, $K[1]'' \otimes K[2]'$ and $K[1]'' \otimes K[2]''$.

Lemma 6. ${}_{\mathsf{PO}}\mathsf{AND}$ *is a physically obfuscated primitive.*

With this additional physically obfuscated primitive, it becomes possible to obfuscate non linear stream-ciphers such as Trivium.

Whole Description of Trivium Obfuscation. As in section 3.4, we assume that the current masked state is $M = (s_0, \ldots, s_{287}) \oplus (KS_0, \ldots, KS_{287})$ where the key state KS is computed thanks to the two POK*s* KS' and KS''.

Let x be the current bit to be encrypted, the obfuscated key stream generation is made as follows.

– Set $t_1' = M_{65} \oplus M_{92}$,
– $t_2' \leftarrow M_{161} \oplus M_{176}$,

– $t'_3 \leftarrow M_{242} \oplus M_{287}$,
– set $y = t'_1 \oplus t'_2 \oplus t'_3$, $y \leftarrow y \oplus x$ and erase x from memory.
– Output $_{\mathsf{PO}}\mathsf{Convert}_1(y, KS_{65} \oplus KS_{92} \oplus KS_{161} \oplus KS_{176} \oplus KS_{242} \oplus KS_{287}, 0)$.

At this stage the encrypted version of x has been obtained correctly.

– Update $t'_1 = t'_1 \oplus M_{170}$, $t'_2 = t'_2 \oplus M_{263}$, $t'_3 = t'_3 \oplus M_{68}$.
– Update $t'_1 = t'_1 \oplus {}_{\mathsf{PO}}\mathsf{AND}(M_{90}, M_{91}, KS_{90}, KS_{91}, KS_{93}, K[4])$
– Update $t'_2 = t'_2 \oplus {}_{\mathsf{PO}}\mathsf{AND}(M_{174}, M_{175}, KS_{174}, KS_{175}, KS_{177}, K[4])$
– Update $t'_3 = t'_3 \oplus {}_{\mathsf{PO}}\mathsf{AND}(M_{285}, M_{286}, KS_{285}, KS_{286}, KS_0, K[4])$

where KS_{93}, KS_{177} and KS_0 corresponds to the bits of KS whose indexes are the future positions of t'_1, t'_2, t'_3 for the register updating) and with $K[4]$ corresponding to a couple of two POKs as in Table 7.

– Update $t'_1 = {}_{\mathsf{PO}}\mathsf{Convert}_1(t'_1, KS_{170} \oplus KS_{65} \oplus KS_{92} \oplus KS_{93}, KS_{93})$
– Update $t'_2 = {}_{\mathsf{PO}}\mathsf{Convert}_1(t'_2, KS_{177} \oplus KS_{263} \oplus KS_{161} \oplus KS_{176}, KS_{177})$
– Update $t'_3 = {}_{\mathsf{PO}}\mathsf{Convert}_1(t'_3, KS_0 \oplus KS_{68} \oplus KS_{242} \oplus KS_{287}, KS_0)$

At this stage, this leads to the equality between t'_1 and $s_{65} \oplus s_{92} \oplus s_{170} \oplus (s_{90} \otimes s_{91}) \oplus KS_{93}$, i.e. the value of t_1 at the original step (9) xored with KS_{93} (similar for t'_2, t'_3 with the values $t_2 \oplus KS_{177}$ and $t_3 \oplus KS_0$). To finish the register updating, we run these last operations.

– Compute M' as
$(M'_0, M'_1, \ldots, M'_{92}) \leftarrow (t'_3, M_0, \ldots, M_{91})$,
$(M'_{93}, M'_{94}, \ldots, M'_{176}) \leftarrow (t'_1, M_{93}, \ldots, M_{175})$,
$(M'_{177}, M'_{178}, \ldots, M'_{287}) \leftarrow (t'_2, M_{177}, \ldots, M_{286})$
– Then update M' with
$${}_{\mathsf{PO}}\mathsf{Convert}\Big(M', (0, KS_0, \ldots, KS_{91}, 0, KS_{93}, \ldots, KS_{175}, 0, KS_{177}, \ldots, KS_{286}), (0, KS_1, \ldots, KS_{92}, 0, KS_{94}, \ldots, KS_{176}, 0, KS_{178}, \ldots, KS_{287})\Big)$$

This leads to the update version of the state register obfuscated by KS.

Lemma 7. *This implementation of the key stream generation of Trivium is physically obfuscated.*

5 Conclusion

We describe in this paper physical obfuscation of binary operations (XOR, AND, Scalar Product) with a study of their applications to stream ciphers. As these binary operations enable any boolean operations to be computed, our ideas are useful for other kind of cryptographic primitives which use basic operations and where increasing resistance of tags against compromise is required. For instance, the HB-related RFID protocols (HB [21], HB$^+$ [22] and modified version [6, 12, 16, 28, 30, 5]) are good targets for our obfuscation techniques which can be seen as a enhancement of [20, 19] where PUF are introduced. Other RFID protocols based on binary operations can be improved as well, e.g. the scheme [11]. Efficient hash functions such as [2, 1] are also of interest.

Further works would include the analysis of the implementation overhead to achieve such physical resistance. In many settings, one chooses connection polynomials for LFSRs such that their Hamming weight is small. This lowers the cost of computing the state change. But the obfuscation technique essentially randomizes the state change. The expected Hamming weight of the masked connection polynomial is then half the length of the LFSR.

Acknowledgements. The authors thank the referees for their helpful comments.

References

1. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak specifications. Submission to NIST (2008)
2. Bertoni, G., Daemen, J., Assche, G.V., Peeters, M.: Radiogatún, a belt-and-mill hash function. NIST - Second Cryptographic Hash Workshop, August 24-25 (2006)
3. Blömer, J., Guajardo, J., Krummel, V.: Provably secure masking of AES. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 69–83. Springer, Heidelberg (2004)
4. Briceno, M., Goldberg, I., Wagner, D.: A pedagogical implementation of A5/1 (1999), `http://jya.com/a51-pi.htm`
5. Bringer, J., Chabanne, H.: Trusted-HB: A low-cost version of HB$^+$ secure against man-in-the-middle attacks. IEEE Transactions on Information Theory 54(9), 4339–4342 (2008)
6. Bringer, J., Chabanne, H., Dottax, E.: HB^{++}: a lightweight authentication protocol secure against some attacks. In: SecPerU, pp. 28–33. IEEE Computer Society, Los Alamitos (2006)
7. Bringer, J., Chabanne, H., Icart, T.: Improved privacy of the tree-based hash protocols using physically unclonable function. In: Ostrovsky, R., De Prisco, R., Visconti, I. (eds.) SCN 2008. LNCS, vol. 5229, pp. 77–91. Springer, Heidelberg (2008)
8. De Cannière, C., Preneel, B.: Trivium specifications. eSTREAM, ECRYPT Stream Cipher Project (2005)
9. De Cannière, C., Preneel, B.: Trivium - a stream cipher construction inspired by block cipher design principles. In: eSTREAM, ECRYPT Stream Cipher Project (2006)
10. Coppersmith, D., Krawczyk, H., Mansour, Y.: The shrinking generator. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 22–39. Springer, Heidelberg (1994)
11. Dolev, S., Kopeetsky, M., Shamir, A.: RFID authentication efficient proactive information security within computational security. Technical Report 08-2007, Department of Computer Science, Ben-Gurion University (July 2007)
12. Duc, D.N., Kim, K.: Securing HB+ against GRS man-in-the-middle attack. In: Proceedings of the Symposium on Cryptography and Information Security (SCIS 2007) (2007)
13. Frumkin, D., Shamir, A.: Un-trusted-HB: Security vulnerabilities of trusted-HB. Cryptology ePrint Archive, Report 2009/044 (2009), `http://eprint.iacr.org/`
14. Gassend, B.: Physical random functions. Master's thesis, Computation Structures Group, Computer Science and Artificial Intelligence Laboratory. MIT (2003)

15. Gassend, B., Clarke, D.E., van Dijk, M., Devadas, S.: Silicon physical random functions. In: Atluri, V. (ed.) ACM Conference on Computer and Communications Security, pp. 148–160. ACM, New York (2002)
16. Gilbert, H., Robshaw, M., Seurin, Y.: HB#: Increasing the security and efficiency of HB$^+$. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 361–378. Springer, Heidelberg (2008)
17. Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: FPGA Intrinsic PUFs and Their Use for IP Protection. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 63–80. Springer, Heidelberg (2007)
18. Günther, C.G.: Alternating step generators controlled by de bruijn sequences. In: Price, W.L., Chaum, D. (eds.) EUROCRYPT 1987. LNCS, vol. 304, pp. 5–14. Springer, Heidelberg (1988)
19. Hammouri, G., Öztürk, E., Birand, B., Sunar, B.: Unclonable lightweight authentication scheme. In: Chen, L., Ryan, M.D., Wang, G. (eds.) ICICS 2008. LNCS, vol. 5308, pp. 33–48. Springer, Heidelberg (2008)
20. Hammouri, G., Sunar, B.: Puf-hb: A tamper-resilient hb based authentication protocol. In: Bellovin, S.M., Gennaro, R., Keromytis, A.D., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 346–365. Springer, Heidelberg (2008)
21. Hopper, N.J., Blum, M.: Secure human identification protocols. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 52–66. Springer, Heidelberg (2001)
22. Juels, A., Weis, S.A.: Authenticating pervasive devices with human protocols. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 293–308. Springer, Heidelberg (2005)
23. Krawczyk, H.: LFSR-based Hashing and Authentication. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 129–139. Springer, Heidelberg (1994)
24. Lee, D.H., Park, J.H., Han, J.W.: Security analysis of a variant of self-shrinking generator. IEICE Transactions 91-A(7), 1824–1827 (2008)
25. Lihua, D., Yupu, H.: Weak generalized self-shrinking generators. Journal of Systems Engineering and Electronics 18(2), 407–411 (2007)
26. MacWilliams, F., Sloane, N.: The theory of error-correcting codes, ch. 9. North-Holland, Amsterdam (1977)
27. Meier, W., Staffelbach, O.: The self-shrinking generator. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 205–214. Springer, Heidelberg (1995)
28. Munilla, J., Peinado, A.: HB-MP: A further step in the HB-family of lightweight authentication protocols. Computer Networks 51(9), 2262–2267 (2007)
29. Oswald, E., Mangard, S., Pramstaller, N., Rijmen, V.: A side-channel analysis resistant description of the AES S-box. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 413–423. Springer, Heidelberg (2005)
30. Piramuthu, S., Tu, Y.-J.: Modified HB authentication protocol. In: Western European Workshop on Research in Cryptology, WEWoRC (2007)
31. Shamir, A.: How to share a secret. ACM Commun. 22(11), 612–613 (1979)
32. Suh, G.E., Devadas, S.: Physical unclonable functions for device authentication and secret key generation. In: DAC, pp. 9–14. IEEE, Los Alamitos (2007)
33. Thomas, S., Anthony, D., Berson, T., Gong, G.: The W7 stream cipher algorithm. Internet Draft, April 2002 (2002)
34. Tuyls, P., Batina, L.: RFID-tags for anti-counterfeiting. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 115–131. Springer, Heidelberg (2006)

A Physical Unclonable Function

Gassend in [14] introduces the concept of Physical Unclonable Function (PUF): a function that maps challenges (stimuli) to responses, that is embodied by a physical device, and that has the following properties:

1. easy to evaluate,
2. hard to characterize, from physical observation or from chosen challenge-response pairs,
3. hard to reproduce.

For a given challenge, a PUF always gives the same answer. The hardness of characterization and reproduction means that it is impossible to reproduce or to characterize the PUF thanks to a reasonable amount of resources (time, money, ...). PUF can thus be viewed as pseudo-random function (note however that they can be limited in the number of possible challenge-response pairs as explained in [17]) where the randomness is insured thanks to physical properties.

[34] defines an Integrated Physical Unclonable Function (I-PUF) as a PUF with the additional interesting properties listed below:

1. The I-PUF is inseparably bound to a chip. This means that any attempt to remove the PUF from the chip leads to the destruction of the PUF and of the chip.
2. It is impossible to tamper with the communication (measurement data) between the chip and the PUF.
3. The output of the PUF is inaccessible to an attacker.

These properties ensure the impossibility to analyze physically a PUF without changing its output. Hence, physical attacks corrupt the PUF and the chip leaving the attacker without any information about the PUF. Particularly, volatile memory cannot be read out without destroying the I-PUF. Silicon PUF have been already described in [15] and can be taken as relevant examples of I-PUF, they are based on delay comparison among signals running through random wires. Moreover, they only require a few resources to be implemented. A practical example of implementation is described in [32]. The final output of a PUF should not contain any errors, whatever the external conditions are. This problem is generally handle thanks to error correcting techniques (cf. [34]).

B Krawczyk's MACs

The MAC mechanism described by [23] works as follows.

If two parties share a common key consisting of a particular function $h \in H$ and a random pad e of length L, then the MAC of a message X is computed as $t = h(X) \oplus e$. To break the authentication, an adversary should find X' and t' such that $t' = h(X) \oplus e$. For this, h and e must remain secret.

Definition 4. *A family H of hash functions is said ϵ-balanced (or ϵ-almost universal) if:* $\forall X \in \{0,1\}^m, X \neq 0$, $c \in \{0,1\}^L$, $\Pr[h \in H, h(X) = c] \leq \epsilon$.

[23] proves the property below.

Proposition 1. *If H is a family of linear hash functions and if H is ϵ-balanced then the probability of success of an adversary is lower than ϵ.*

The scheme is then said ϵ-secure.

Following the principle of a one-time pad, the same h can be reused but e must be a random pad different each time.