Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives

Applications to AES, PRESENT and SKINNY

Jérémy Jean[1], Amir Moradi[2], Thomas Peyrin[3,4], and Pascal Sasdrich[2]

[1] ANSSI Crypto Lab, Paris, France
Jeremy.Jean@ssi.gouv.fr

[2] Horst Görtz Institute for IT Security, Ruhr-Universität Bochum, Bochum, Germany
{firstname.lastname}@rub.de

[3] Temasek Laboratories, Nanyang Technological University, Singapore

[4] School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
Thomas.Peyrin@ntu.edu.sg

Abstract. Area minimization is one of the main efficiency criterion for lightweight encryption primitives. While reducing the implementation data path is a natural strategy for achieving this goal, Substitution-Permutation Network (SPN) ciphers are usually hard to implement in a bit-serial way (1-bit data path). More generally, this is hard for any data path smaller than its Sbox size, since many scan flip-flops would be required for storage, which are more area-expensive than regular flip-flops.

In this article, we propose the first strategy to obtain extremely small bit-serial ASIC implementations of SPN primitives. Our technique, which we call *bit-sliding*, is generic and offers many new interesting implementation trade-offs. It manages to minimize the area by reducing the data path to a single bit, while avoiding the use of many scan flip-flops.

Following this general architecture, we could obtain the first bit-serial and the smallest implementation of AES-128 to date (1560 GE for encryption only, and 1738 GE for encryption and decryption with IBM 130nm standard-cell library), greatly improving over the smallest known implementations (about 30% decrease), making AES-128 competitive to many ciphers specifically designed for lightweight cryptography. To exhibit the generality of our strategy, we also applied it to the PRESENT and SKINNY block ciphers, again offering the smallest implementations of these ciphers thus far, reaching an area as low as 1065 GE for a 64-bit block 128-bit key cipher. It is also to be noted that our bit-sliding seems to obtain very good power consumption figures, which makes this implementation strategy a good candidate for passive RFID tags.

Key words: Bit-serial implementations, bit-slide, lightweight cryptography.

1 Introduction

Due to the increasing importance of pervasive computing, lightweight cryptography has attracted a lot of attention in the last decade among the symmetric-key community. In particular, we have seen many improvements in both primitive design and their hardware implementations. We currently know much better how a lightweight encryption scheme should look like (small block size, small nonlinear components, very few or even no XORs gates for the linear layer, etc.).

Lightweight cryptography can have different meanings depending on the applications and the situations. For example, for passive RFID tags, power consumption is very important, and for battery-driven devices energy consumption is a top priority. Power and energy consumption depend on both the area and throughput of the implementation. In this scenario, so-called round-based implementations (i.e., one cipher round per clock cycle) are usually the most efficient trade-offs with regards to these metrics. For example, the tweakable block cipher SKINNY [6] was recently introduced with the goal of reaching the best possible efficiency for round-based implementations.

Yet, for the obvious reason that many lightweight devices are very strongly constrained, one of the most important measurement remains simply the implementation area, regardless of the throughput. It was estimated in 2005 that only a maximum of 2000 GE can be dedicated to security in an RFID tag [19]. While these numbers might have evolved a little since then, it is clear that area is a key aspect when designing/implementing a primitive. In that scenario, round-based implementations are far from being optimal since the data path is very large. In contrast, the serial implementation strategy tries to minimize the data path to reduce the overall area. Some primitives even specialized for this type of implementation (e.g., LED [15], PHOTON [14]), with a linear layer crafted to be cheap and easy to perform in a serial way.

In 2013, the National Security Agency (NSA) published two new ciphers [5], SIMON (tuned for hardware) and SPECK (tuned for software) targeting very low-area implementations. SIMON is based on a simple Feistel construction with just a few rotations, ANDs and XORs to build the internal function. The authors showed that SIMON's simplicity easily allows many hardware implementation trade-offs with regards to the data path, going as low as a 1-bit-serial implementation.

For Substitution-Permutation Network (SPN) primitives, like AES [12] or PRESENT [7], the situation is more complex. While they can usually provide more confidence concerning their security, they are known to be harder to implement in a bit-serial way.To the best of the authors' knowledge, as of today, there is no bit-serial implementation of an SPN cipher, mainly due to the underlying structure organized around their Sbox and linear layers. While this construction offers efficient and easy implementation trade-offs, it seems nontrivial to build an architecture with a dapa path below the Sbox size. Thus, there remains a gap to bridge between SPN primitives and ciphers with a general SIMON-like structure.

Our Contributions. In this article, we provide the first general bit-serial Application-Specific Integrated Circuit (ASIC) implementation strategy for SPN

ciphers. Our technique, that we call *bit-sliding*, allows implementations to use small data paths, while significantly reducing the number of costly scan flip-flops (FF) used to store the state and key bits.

Although our technique mainly focuses on 1-bit-serial implementations, it easily scales and supports many other trade-offs, e.g., data paths of 2 bits, 4 bits, etc. This agility turns to be very valuable in practice, where one wants to map the best possible implementation to a set of constraints combining a particular scenario and specific devices. We applied our strategy to `AES`, and together with other minor implementation tricks, we obtained extremely small `AES-128` implementations on ASIC: only 1560 Gate Equivalent (GE) for encryption (incl. 75% for storage), and 1738 GE for encryption and decryption using IBM 130nm library (incl. 67% for storage).[5] By comparison, using the same library, the smallest ASIC implementation of `AES-128` previously known requires 2182 GE for encryption [22] (incl. 64% of storage), and 2402 GE for encryption and decryption [3] (incl. 55% of storage).[6] Our results show that `AES-128` could almost be considered as a lightweight cipher.

Since our strategy is very generic, we also applied it to the cases of `PRESENT` and `SKINNY`, again obtaining the smallest known implementations. More precisely, for the 64-bit block 128-bit key versions and using the IBM 130nm library, we could reach 1065 GE for `PRESENT` and 1054 GE for `SKINNY` compared to the to-date smallest `PRESENT-128` with 1230 GE [31]. Our work shows that the gap between the design strategy of `SIMON` and a classical SPN is smaller than previously thought, as `SIMON` can reach 958 GE for the same block/key sizes.

In terms of power consumption, it turns out that bit-sliding provides good results when compared to currently known implementation strategies. This makes it potentially interesting for passive RFID tags for which power is a key constraint. However, as for any bit-serial implementation, due to the many cycles required to execute the circuit, the energy consumption figures will not be as good as one can obtain with round-based implementations.

We emphasize that for fairness, we compare the various implementations to ours using five standard libraries: namely, UMC 180nm, UMC 130nm, UMC 90nm, NanGate 45nm and IBM 130nm.

2 Bit-Sliding Implementation Technique

We describe in this section the conducting idea of our technique, which allows to significantly decrease the area required to *serially* implement any SPN-based cryptographic primitive. To clearly expose our strategy, we first describe the general structure of SPN primitives in Section 2.1 and we recall the most common types of hardware implementation trade-offs in Section 2.2. Then, in Section 2.3, we explain the effect of reducing the data path of an SPN implementation, in

---

[5]The same library used to benchmark `SIMON` area footprints in [5].

[6]We note that the 2400 GE reported in [22] are done on a different library, namely UMC 180nm. The numbers we report here are obtained by re-synthesizing the code from [22] on IBM 130nm.

particular how the choice of the various flip-flops used for state storage strongly affects the total area. Finally, we describe our bit-sliding implementation strategy in Section 2.4 and we tackle the problem of bit-serializing any Sbox in Section 2.5. Applications of these techniques to AES-128 and PRESENT block ciphers are conducted in the subsequent sections of the paper (the case of SKINNY is provided in the long version of the paper [17]). For completeness, we provide in Section 2.6 a quick summary of previous low-area implementations of SPN ciphers such as AES-128 and PRESENT.

2.1 Substitution-Permutation Networks

Even though our results apply to any SPN-based construction (block cipher, hash function, stream cipher, public permutation, etc.), for simplicity of the description, we focus on block ciphers.

A *block cipher* corresponds to a keyed family of permutations over a fixed domain, $E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$. The value k denotes the key size in bits, n the dimension of the domain on which the permutation applies, and for each key $K \in \{0,1\}^k$, the mapping $E(K,\cdot)$, that we usually denote $E_K(\cdot)$, defines a permutation over $\{0,1\}^n$.

From a high-level perspective, an SPN-based block cipher relies on a round function f that consists of the mathematical composition of a nonlinear permutation S and a linear permutation P, which can be seen as a direct application of Shannon's confusion (nonlinear) and diffusion (linear) paradigm [27].

From a practical point of view, the problem of implementing the whole cipher then reduces to implementing the small permutations S and P, that can either be chosen for their good cryptographic properties, and/or for their low hardware or software costs. In most known ciphers, the nonlinear permutation $S : \{0,1\}^n \to \{0,1\}^n$ relies on an even smaller permutation called Sbox, that is applied several times in parallel on independent portions on the internal n-bit state. We denote by s the bit-size of these Sboxes. Similarly, the linear layer often comprises identical functions applied several times in parallel on independent portions on the internal state. We denote by l the bit-size of these functions.

2.2 Implementation Trade-offs

We usually classify ASIC implementations of cryptographic algorithms in three categories: round-based implementations, fully unrolled implementations and serial implementations. A round-based implementation typically offers a very good area/throughput trade-off, by providing the cryptographic functionalities (e.g., encryption and decryption). The idea is this case consists in simply implementing the full round function f of the block cipher in one clock cycle and to reuse the circuit to produce the output of the cipher. In contrast, to minimize the latency a fully unrolled implementation would implement *all* the rounds at the expense of a much larger area, essentially proportional to the number of the cipher rounds (for instance PRINCE [8] or MANTIS [6] have been designed to satisfy such low-latency requirements). Finally, serial implementations (the focus of this article) trade

throughput by only implementing a small fraction of the round function f, for applications that require to minimize the area as much as possible.

2.3   Data Path Reduction and Flip-Flops

From round-based to serial implementations, the data path is usually reduced. In the case of SPN primitives, reducing this data path is natural as long as the application independence of the various sub-components of the cipher (s-bit Sboxes and l-bit linear functions) is respected. This is the reason why all the smallest known SPN implementations are serial implementations with an s-bit data path (l being most of the time a multiple of s). Many trade-offs lying between an s-bit implementation and a round-based implementation can easily be reached. For example, in the case of AES, depending on the efficiency targets, one can trivially go from a byte-wise implementation, to a row- or column-wise implementation, up to a full round-based implementation.

   The data path reduction in an ASIC implementation offers area reduction at two levels. First, it allows to reduce the number of sub-components to implement (n/s Sboxes in the case of a round-based implementation versus only a single Sbox for a s-bit serial implementation), directly reducing the total area cost. Second, it offers an opportunity to reduce the number of scan flip-flops (scan FF), at the benefit of regular flip-flops (FF) for storage. A scan FF contains a 2-to-1 multiplexer to select either the data input or the scan input. This scan feature allows to drive the FF data input with an alternate source of data, thus greatly increasing the possibilities for the implementer about where the data navigates. In short: in an ASIC architecture, when a storage bit receives data only from a single source, regular FF can be used. If another source must potentially be selected, then a scan FF is required (with extra multiplexers in case of multiple sources). However, the inner multiplexer comes at a non-negligible price, as scan FF cost about 20-30% more GE than regular ones.

2.4   The Bit-Sliding Strategy

Because of the difference between scan FF and regular FF, when minimizing the area is the main goal, there is a natural incentive in trying to use as many regular FF as possible. In other words, the data should flow in such a way that many storage bits only have a single input source. This is hard to achieve with a classical s-bit data path, since the data usually moves from all bits of an Sbox to all bits of another Sbox. Thus, the complex wiring due to the cipher specifications impacts all the Sbox storage bits at the same time. For example, in the case of AES, the ShiftRows forces most internal state storage bits to use scan FFs.

   This is where the bit-sliding strategy comes into play. When enabling the bit-serial implementation by reducing the data path from s bits to a single bit, we make the data bits *slide*. All the complex data wiring due to the cipher specifications becomes handled only by the very first bit of the cipher state. Therefore, this first bit has to be stored in a scan FF, while the other bits can simply use regular FF. Depending on the cipher sub-components, other state

bits should also make use of scan FF, but the effect is obviously stronger as the size of the Sbox grows larger.

We emphasize that minimizing the ratio of scan FF is really the relevant way to look at the problem of area minimization. Most previous works concentrated on the optimization of the ciphers sub-components. Yet, in the case of lightweight cryptography where implementations are already very optimized for area, these sub-components represent a relatively small portion of the total area cost, in opposition to the storage costs. For example, for our PRESENT implementations, the storage represents about 80-90% of the total area cost. For AES-128, the same ratio is about 65-75%.

2.5 Bit-Serializing any Sbox

A key issue when going from an s-bit data path to a single bit data path, is to find a way to implement the Sbox in a bit-serial way. For some ciphers, like PICCOLO [28] or SKINNY [6], this is easy as their Sbox can naturally be decomposed into an iterative 1-bit data path process. However, for most ciphers, this is not the case and we cannot assume such a decomposition always exists.

We therefore propose to emulate this bit-serial Sbox by making use of s scan FFs to serially shift out the Sbox output bits at each clock cycle, while reusing the classical s-bit data path circuit of the entire Sbox to store its output.

Although the cost of this strategy is probably not optimal (extra regular FFs should change to scan FF), we nevertheless argue that this is not a real issue since the overall cost of this bit-serial Sbox implementation is very small when compared to the total area cost of the entire cipher. Moreover, this strategy has the important advantage that it is very simple to put into place and that it generically works for any Sbox.

2.6 Previous Serial SPN Implementations

Most of the existing SPN ciphers such as AES or PRESENT have been implemented using word-wise serialization, with 4- or 8-bit data paths. For AES, after two small implementations of the encryption core by Feldhofer et al. [13] in 2005 and Hämäläinen et al. [16] in 2006, one can emphasize the work by Moradi et al. [22] in 2011, which led to an encryption-only implementation of AES-128 with 2400 GE for the UMC 180nm standard-cell library. More recently, a follow-up work by Banik et al. [2] added the decryption functionality, while keeping the overhead as small as possible: they reached a total of 2645 GE on STM 90nm library. According to our estimations (see Section 3), this implementation requires around 2760 GE on UMC 180nm, which therefore adds decryption to [22] for a small overhead of about 15%. In [3] Banik et al. further improved this to 2227 GE on STM 90nm (about 2590 GE on UMC 180nm).

As for PRESENT, the first result appeared in the specifications [7], where the authors report a 4-bit serial implementation using about 1570 GE on UMC 180nm. In 2008, Rolfes et al. [25] presented an optimization reaching about 1075 GE on the same library, which was further decreased to 1032 GE by Yap et al. [31].

Finally, we remark that bit-serial implementations of SKINNY and GIFT [4] have already been reported, which are based on the work described in this article.

3   Application to AES-128

3.1   Optimizations of the Components

Since its standardization, the AES has received many different kind of contributions including the attempts to optimize its implementations on many platforms. We review here the main results that we use in our implementations, which specifically target two internal components of the AES: the 8-bit Sbox from SubBytes and the matrix multiplication applied in the MixColumns.

SubBytes. One crucial design choice of any SPN-based cipher lies in the Sbox and its cryptographic strength. In the AES, Daemen and Rijmen chose to rely on the algebraic inversion in the field $GF(2^8)$ for its good resistance to classical differential and linear cryptanalysis. Based on this strong mathematical structure, Satoh et al. in [26] used the tower field decomposition to implement the field inversion using only 2-bit operations, later improved by Mentens et al. in [21]. Then, in 2005, Canright reported a smaller implementation of the combined Sbox and its inverse by enumerating all possible normal bases to perform the decomposition, which resulted in the landmark paper [10]. In our serial implementation supporting both encryption and decryption, we use this implementation.

However, when the inverse Sbox is not required, especially for inverse-free mode of operations like CTR that do not require the decryption operation, the implementation cost can be further reduced. Indeed, Boyar, Matthews and Peralta have shown in [9] that solving an instance of the so-called Shortest Linear Program NP-hard problem yields optimized AES Sbox implementations. In particular, they introduce a 115-operation implementation of the Sbox, further refined to 113 logical operations in [11], which is, to the best of our knowledge, the smallest known to date. We use this implementation in our encryption-only AES cores, which allows to save 20-30 GE over Canright's implementation.

We should also refer to [29], where the constructed Sbox with small footprint needs in average 127 clock cycles. The work has been later improved in [30], where the presented Sbox finishes the operation after at most 16 (in average 7) clock cycles. Regardless of the vulnerability of such a construction to timing attacks [20], we could not use them in our architecture due to their latency higher than 8 clock cycles.

MixColumns. Linear layers of SPN-based primitives have attracted lots of attention in the past few years, mostly from the design point of view. Here, we are interested in finding an efficient implementation of the fixed MixColumns transformation, which can either be seen as multiplication by a 4×4 matrix over $GF(2^8)$ or by a 32×32 matrix over $GF(2)$. For 8-bit data path, similar to previous works like [1, 2, 33], we have considered the 32×32 binary matrix to
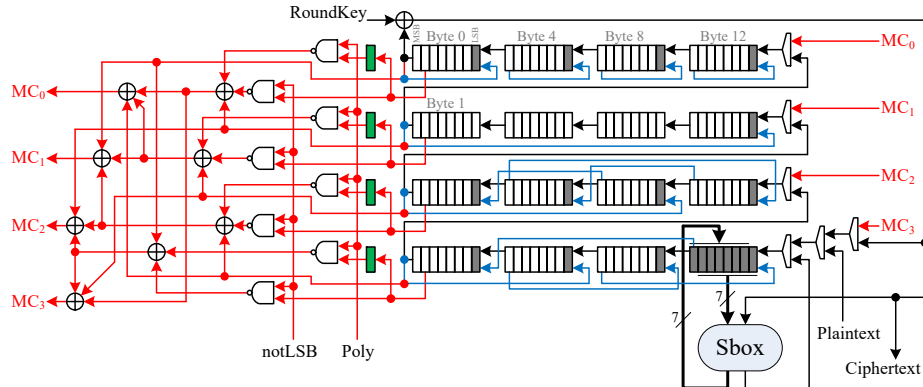
7

Figure 1: Bit-serial architecture for `AES-128` (encryption only, data path).

implement the MixColumns. An already-reported strategy can implement it in 108 XORs, but we tried to slightly improve this by using a heuristic search tool from [18], which yielded two implementations using 103 and 104 XORs, where the 104-XOR one turned to be more area efficient.

3.2   Bit-Serial Implementations of `AES-128` Encryption

We first begin by describing an implementation that only supports encryption, and then complete it to derive one that achieves both encryption and decryption.

Data Path. The design architecture of our bit-serial implementation of `AES-128` is shown in Figure 1. The entire 128-bit state register forms a shift register, which is triggered at every clock cycle. The white register cells indicate regular FFs, while the gray ones scan FFs. The plaintext bits are serially fed from most significant bit (MSB) down to least significant bit (LSB) of the Bytes `0,4,8,12,1,5,9,13,2,6,10,14,3,7,11,15`. In other words, during the first 128 clock cycles, first 8 bits (MSB downto LSB) of plaintext Byte `0` and then that of Byte `4` are given till the 8 bits (MSB downto LSB) of plaintext Byte `15`.

The AddRoundKey is also performed in a bit serial form, i.e., realized by one 2-input XOR gate. For each byte, during the first 7 clock cycles, the AddRoundKey result is fed into the rotating shift register, and at the 8th clock cycle, the Sbox output is saved at the last 8 bits of the shift register and at the same time the rest of the state register is shifted. Therefore, we had to use scan FFs for the last 8 bits of the state shift register (see Figure 1). For the Sbox module, as stated before, we made use of the 113-gate description given in [11] by Cagdas Calik. After 128 clock cycles, the SubBytes is completely performed.

The ShiftRows is also performed bit-serially. The scan FFs enable us to perform the entire ShiftRows in 8 clock cycles. We should emphasize that we have examined two different design architectures. In our design, in contrast to [2,3,22],
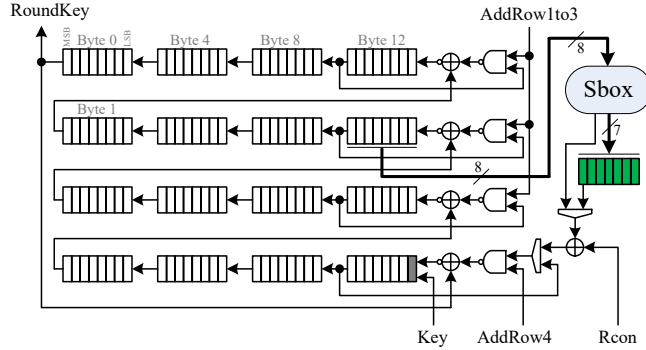
Figure 2: Bit-serial architecture for `AES-128` (encryption only, key path).

the state register is always shifted without any exception. This avoids extra logic to enable and disable the registers. In [3], an alternative solution is used, where each row of the state register is controlled by clock gating. Hence, by freezing the first row, shifting the second row once, the third row twice and the forth row three times, the ShiftRows can be performed. We have examined this fashion in our bit-serial architecture as well. It allows us to turn 9 scan FFs to regular FFs, but it needs 4 clock gating circuits and the corresponding control logic. For the bit-serial architecture, it led to more area consumption. We discuss this architecture in Section 3.4, when we extend our serial architecture to higher bit lengths.

For the MixColumns, we also provide a bit-serial version. More precisely, each column is processed in 8 clock cycles, i.e., the entire MixColumns is performed in 32 clock cycles. In order to enable such a scenario, when processing a column, we need to store the MSB of all four bytes, which are used to determine whether the extra reduction for the `xtime` (i.e., multiplication by 2 in $\mathrm{GF}(2^8)$ under `AES` polynomial) is required. The green cells in Figure 1 indicate the extra register cells which are used for this purpose. The input of the green register cells come from the 2nd MSB of column bytes. Therefore, these registers should store the MSB one clock cycle before the operation on each column is started. During the ShiftRows and at the 8th clock cycle of MixColumns on each column, these registers are enabled. This enables us to fulfill our goal, i.e., always clocking the state shift register. The bit-serial MixColumns circuit needs two control signals: `Poly`, which provides the bit representation of the `AES` polynomial `0x1B` serially (MSB downto LSB) and `notLSB`, which enables `xtime` for the LSB.

Therefore, one full round of the `AES` is performed in $128 + 8 + 32 = 168$ clock cycles. During the last round, MixColumns is ignored, and the last AddRoundKey is performed while the ciphertext bits are given out. Therefore, the entire encryption takes $9 \times 168 + 128 + 8 + 128 = 1776$ clock cycles. Similar to [2, 3, 22], while the ciphertext bits are given out, the next plaintext can be fed inside. Therefore, similar to their reported numbers, the clock cycles required to fed plaintext inside are not counted.

Key Path. The key register is similar to the state register and is shifted one bit per clock cycle, and gives one bit of the RoundKey to be used by AddRoundKey (see Figure 2). The key schedule is performed in parallel to the AddRoundKey and SubBytes, i.e., in 128 clock cycles. In other words, while the RoundKey bit is given out the next RoundKey is generated. Therefore, the key shift register needs to be frozen during ShiftRows and MixColumns operations, which is done by means of clock gating. As shown in Figure 2, the entire key register except the last one is made by regular FFs, which led to a large area saving. During key schedule, the Sbox module, which is shared with the data path,[7] is required 4 times. We instantiate 7 extra scan FFs, those marked by green, which save 7 bits of the Sbox output and can shift serially as well. It is noteworthy that 4 of such register cells are shared with the data path circuit to store the MSBs required in MixColumns.[8] At the first clock cycle of the key schedule, the Sbox is used and its output is stored in the dedicated green register. It is indeed a perfect sharing of the Sbox module between the data path and key path circuits. During every 8 clock cycles, the Sbox is used by the key path at the first clock cycle and by the data path at the last clock cycle. During the first 8 clock cycles, the Sbox output $S(\text{Byte}_{13})$ is added to Byte_0, which is already the first byte of the next Roundkey. Note that the RoundConstant `Rcon` is also provided serially by the control logic. During the next 16 clock cycles, by means of `AddRow4` signal, $S(\text{Byte}_{13}) \oplus \text{Byte}_0 \oplus \text{Byte}_4$ and $S(\text{Byte}_{13}) \oplus \text{Byte}_0 \oplus \text{Byte}_4 \oplus \text{Byte}_8$ are calculated, which are the next 2 bytes of the next RoundKey. The next 8 clock cycles, Byte_{12} is fed unchanged into the shift register, that is required to go through the Sbox later. This process is repeated 4 times and at the last 8 clock cycles, i.e., clock cycles 121 to 128, by means of `AddRow1to3`, the last XORs are performed to make the Bytes 12, 13, 14, and 15 of the next RoundKey. During the next 8+32 clock cycles, when the data path circuit is performing ShiftRows and MixColumns, the entire key shift register is frozen.

3.3 Bit-Serial `AES-128` Encryption and Decryption Core

Data Path. In order to add decryption, we slightly changed the architecture (see Figure 3). First, we replaced the last 7 regular FFs by scan FFs, where Byte_0 is stored. Then, as said before, we made use of Canright `AES` Sbox [10].

The encryption functionality of the circuit stays unchanged, while the decryption needs several more clock cycles. After serially loading the ciphertext bits, at the first 128 clock cycles, the AddRoundKey is performed. Afterwards, the ShiftRows^{-1} should be done. To do so, we perform the ShiftRows three times since ShiftRows$^3 = $ ShiftRows^{-1}. This helps us to not modify the design architecture, i.e., no extra scan FF or MUX. Therefore, the entire ShiftRows^{-1} takes $3 \times 8 = 24$ clock cycles. The next SubBytes^{-1} and AddRoundKey are performed at the same time. For the first clock cycle, the Sbox inverse is stored in 7 scan FFs, where Byte_0 is stored, and the same time the XOR with the RoundKey bit and the

---

[7]Eight 2-to-1 MUX at the Sbox input are not shown.
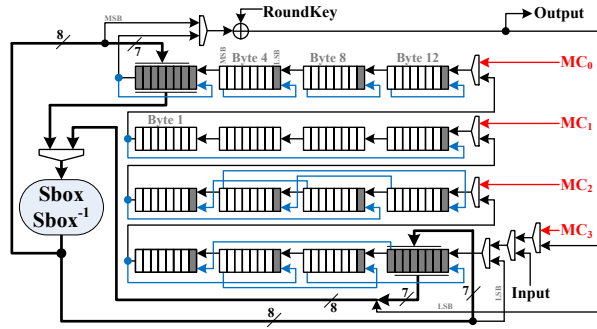[8]It requires four 2-to-1 MUX which are not shown.

Figure 3: Bit-serial architecture for `AES-128` (encryption and decryption, data path).

shift in the sate register happen. In the next 7 clock cycles, the AddRoundKey is performed. This is repeated 16 times, i.e., 128 clock cycles. For the MixColumns^{-1}, we followed the principle used in [3] that MixColumns^3 = MixColumns^{-1}. In other words, we repeat the MixColumns process explained above 3 times, in $3 \times 32 = 96$ clock cycles. Note that for simplicity, the MixColumns circuit is not shown in Figure 3. At the last decryption round, first the ShiftRows^{-1} is performed, in 24 clock cycles, and afterwards, when the SubBytes^{-1} and AddRoundKey are simultaneously performed, the plaintext bits are given out. Therefore, the entire decryption takes $128 + 9 \times (24 + 128 + 96) + 24 + 128 = 2512$ clock cycles. Note that the state register, similar to the encryption-only variant, is always active.

Key Path. Enabling the inverse key schedule in our bit-serial architecture is a bit more involved than in the data path. According to Figure 4, we still make use of only one scan FF and the rest of the key shift register is made by regular FFs. We only extended the 7 green scan FFs to 8. At the first 8 clock cycles, Byte$_1 \oplus$ Byte$_5$ is serially computed and shifted into the green scan FFs, and at
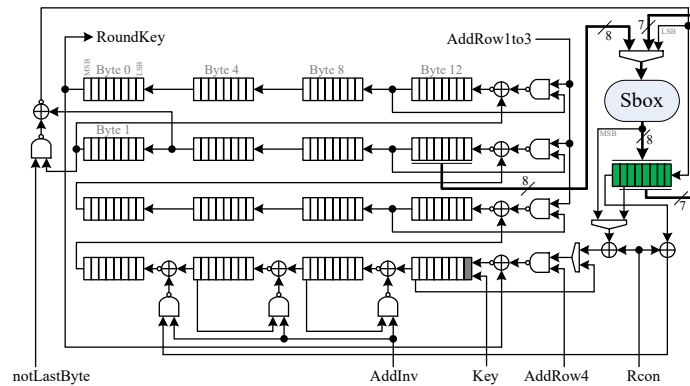

Figure 4: Bit-serial architecture for `AES-128` (encryption and decryption, key path).

the 8th clock cycle the entire 8-bit Sbox output is stored in the green scan FFs. Within the next 16 clock cycles, the key state is just rotated. During the next 8 clock cycles, the green scan FFs are serially shifted out and its XOR results with Byte_0 is stored. At the same time, by means of `AddInv` signal, $\text{Byte}_0 \oplus \text{Byte}_4$, $\text{Byte}_4 \oplus \text{Byte}_8$, and $\text{Byte}_8 \oplus \text{Byte}_{12}$ are serially computed, that are the first 4 bytes of the next RoundKey upwards. For sure, RoundConstant is also provided (serially) in reverse order (by the control logic). This process is repeated 4 times with one exception. At the last time, i.e., at Clock cycles 97 to 104, by means of the `notLastByte` signal, the XOR is bypassed when the green scan FFs are serially loaded. This is due to the fact that such an XOR has already been performed. Hence, the key scheduleinv takes again 128 clock cycles, and is synchronized with the AddRoundKey of the data path circuit. During other clock cycles, where ShiftRows^{-1} and MixColumns^{-1} are performed, the key shift register is disabled.

3.4 Extension to Higher Bit Lengths

We could relatively easily extend our design architecture(s) to higher bit lengths. More precisely, instead of shifting 1 bit at every clock cycle, we can process either 2, 4, or 8 bits. The design architectures stay the same, but every computing module provides 2, 4, or 8 bits at every clock cycle. More importantly, the number of scan FFs increases almost linearly. For the 2-bit version, the 9 scan FFs that enabled ShiftRows must be doubled. Its required number of clock cycles is also half of the 1-bit version, i.e., 888 for encryption and 1256 for decryption.

However, we observed that in 4-bit (resp. 8-bit) serial version almost half (resp. full) of the FFs of the state register need to be changed to scan FF, that in fact contradicts our desire to use as much regular FFs as possible instead of scan FFs. In these two settings (4- and 8-bit serial), we have achieved more efficient designs if the ShiftRows is realized by employing 4 different clock gating, each of which for a row in state shift register. This allows us to avoid replacing 36 (resp. 72) regular FFs by scan FF. This architecture forces us to spend 4 more clock cycles during MixColumns since not all state registers during ShiftRows are shifted, and the MSB for the MixColumns cannot be saved beforehand. Therefore, for the 4-bit version, the AddRoundKey and SubBytes are performed in 32 clock cycles, the ShiftRows in 6 cycles, and the MixColumns in $4 \times (1 + 2) = 12$ cycles, hence $9 \times (32 + 6 + 12) + 32 + 6 + 32 = 520$ clock cycles for the entire encryption.

For the decryption, the ShiftRows^{-1} does not need to be performed as ShiftRows^3, and it can also be done in 6 clock cycles. However, the MixColumns^{-1} still requires to apply 3 times MixColumns, i.e., $3 \times 12 = 36$ cycles. Thus, the entire decryption needs $32 + 9 \times (6 + 32 + 36) + 6 + 32 = 736$ clock cycles.

In the 8-bit serial version, since the Sbox is required during the entire 16 clock cycles of SubBytes, we had to disable the state shift register 4 times to allow the Sbox module to be used by the key schedule. Since MixColumns now computes the entire column in 1 clock cycle, there is no need for extra registers (as well as clock cycles) to save the MSBs. Therefore, AddRoundKey and SubBytes need 20 clock cycles, ShiftRows 3 clock cycles, and MixColumns 4 clock cycles, i.e., $9 \times (20 + 3 + 4) + 20 + 3 + 16 = 282$ clock cycles in total. The first step of

Table 1: `AES-128` implementations for a data path of δ bits @ 100 KHz.

Func.	δ bits	UMC180 GE / μW	UMC130 GE / μW	UMC90 GE / μW	Ngate45 GE / μW	IBM130 GE / μW	Latency Cycles	Ref.
NAND	μm^2	9.677	5.120	3.136	0.798	5.760		
Enc	1	1727 / 3.510	1902 / 0.845	1596 / 0.666	1982 / 100.2	1560 / 0.823	1776	**New**
Enc	2	1796 / 3.640	1992 / 0.904	1667 / 0.699	2054 / 104.6	1625 / 0.842	888	**New**
Enc	4	1920 / 4.040	2168 / 1.040	1784 / 0.800	2146 / 111.4	1731 / 0.892	520	**New**
Enc	8	2112 / 3.990	2360 / 1.020	1968 / 0.784	2337 / 122.2	1912 / 0.874	282	**New**
Enc	8	2400 / 6.240	3574 / 1.270	2292 / 0.768	2768 / 136.6	2182 / 0.984	226	[22]
EncDec	1	1917 / 3.670	2142 / 0.944	1794 / 0.713	2171 / 112.1	1738 / 0.852	1776/2512	**New**
EncDec	2	2028 / 3.920	2269 / 0.972	1916 / 0.761	2286 / 119.8	1855 / 0.922	888/1256	**New**
EncDec	4	2212 / 4.590	2509 / 1.200	2097 / 0.942	2436 / 130.4	2069 / 1.070	520/736	**New**
EncDec	8	2416 / 4.490	2713 / 1.170	2329 / 0.945	2621 / 142.3	2293 / 1.070	282/354	**New**
EncDec	8	2577 / 3.560	2893 / 0.915	2332 / 0.645	2793 / 139.1	2402 / 0.753	246/326	[3]
EncDec	8	2772 / 5.860	3233 / 1.280	2639 / 0.832	3105 / 160.2	2503 / 1.110	226/226	[2]

decryption is AddRoundKey, but at the same time the next RoundKey should be provided. In order to simplify the control logic, the first sole AddRoundKey also takes 20 clock cycles, and MixColumns^{-1} 12 clock cycles. Hence, the entire decryption is performed in $20 + 9 \times (3 + 20 + 12) + 3 + 16 = 354$ clock cycles.

Compared to [2,3,22], our design is different with respect to how we handle the key schedule. For example, our entire key state register needs only 8 scan FFs; we could reduce the area, but with a higher number of clock cycles. It is noteworthy that we have manually optimized most of the control logic (e.g., generation of Rcon) to obtain the most compact design.

3.5 Results

The synthesis result of our designs under five different standard cell libraries and the corresponding power consumption values – estimated at 100 KHz – are shown in Table 1. We have also shown that of the designs reported in [2,3,22]. It should be noted that we had access to their designs and did the syntheses by our considered libraries. It can be seen that in all cases our constructions outperform the smallest designs reported in literature. The numbers listed in Table 1 obtained

under the highest optimization level (for area) of the synthesizer. For all designs (including [2,3,22]), we further forced the synthesizer to make use of the dedicated scan FFs of the underlying library when needed. It can be seen that all of our designs need smaller area footprints compared to the other designs. In case of the estimated power consumption, our designs also outperform the others except the one in [3]. However, as an important observation by increasing the δ, the estimated power consumption is increased. We should highlight that our target is the smallest footprint, and our designs would not provide better results if either area×time or energy is considered as the metric.

Based on the results presented in Table 1, it can be seen that comparing the area based on GE in different libraries does not make much sense. For instance, the synthesis results reported in [2,3] that are based on STM 65nm and STM 90nm libraries cannot be compared with that of another design under a different library. Indeed, such a huge difference comes from the definition of GE, i.e., the relative area of the NAND gate compared to the other gates: an efficient NAND gate (compared to the other gates in the library) will yield larger GE numbers than an inefficient one. The area of the NAND gate under our considered libraries are also listed in Table 1. The designs synthesized by Nangate 45nm show almost the highest GE numbers, that is due to its extremely small NAND gate. More interestingly, using IBM 130nm, it shows the smallest GE numbers while the results with UMC 130nm (with the same technology size) are amongst the largest ones. One reason is the larger NAND gate in IBM 130nm.

4 Application to PRESENT

4.1 Optimization of the Components

Substitution Layer. To help the synthesizer reach an area-optimized implementation, we use the tool described in [18] to look for an efficient implementation of the PRESENT Sbox. We have found several ones that allow to significantly decrease the area of the Sbox, in comparison to a LUT-based VHDL description: namely, while the LUT description yields an area equivalent to 60-70 GE, our Sbox implementation decreases it to about 20-30 GE. In our serial implementations described below, we have selected the PRESENT Sbox implementation described in [18] using 21.33 GE on UMC 180nm In our serial implementations described below, we have selected the PRESENT Sbox implementation described in [18] using 21.33 GE on UMC 180nm, which is the world's smallest known implementation to date of the PRESENT S-box, about 1 GE smaller than the one provided in [32].

Permutation Layer. The diffusion layer of PRESENT is designed as a bit permutation that is cheap and efficient in hardware, particularly for round-based architectures since then the permutation simply breaks down to wired connections. However, for serialized architectures, such as for our bit-sliding technique, the bit permutation seems to be an obstacle. Although the permutation layer has some underlying structure, adapting it for a bit-serial implementation seems nontrivial.
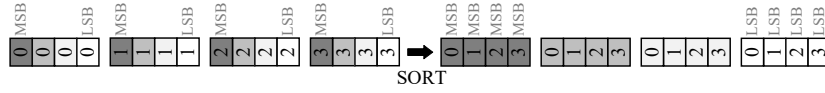
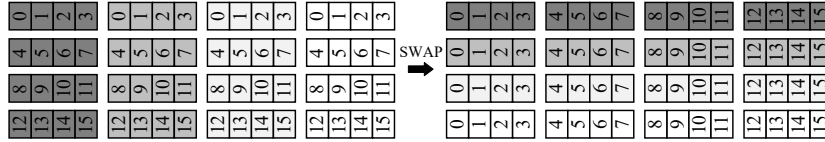**Figure 5:** Local Permutation (`SORT`). Re-ordering of bits according to their significance.

Figure 6: Global Permutation (`SWAP`). Column- and row-wise re-ordering of nibbles.

However, we present in the following an approach that allows to decompose the permutation into two independent operations that can be easily performed in a bit-serial fashion. We note that a two-stage decomposition of the `PRESENT` permutation has also been described in [23]. The first operation performs a local permutation at the bit-level, whereas the second operation performs a global permutation at the nibble-level, comparable to `ShiftRows` in the `AES`.

Local Permutation. Essentially, the local permutation sorts all bits of a single row of the state (in its matrix representation) according to their significance as show in Figure 5. Hence, given four nibbles `0,1,2,3` (with bit-order: MSB downto LSB), the first nibble will contain the most significant bits (in order `0,1,2,3`) after the sorting operation, whereas the fourth nibble will hold the least significant bits. Fortunately, this operation can be applied to each row individually and independently. As a direct consequence, only one row of the state register needs to implement the local permutation, which can then be applied to the state successively.

Global Permutation. After the local permutation has been performed on all rows of the state, all bits are sorted according to their significance and, for instance, the first column will contain all MSBs. However, for a correct implementation of the permutation layer, the bits should be sorted row-wise instead of column-wise. Therefore, the global permutation restores the correct ordering by rearranging the nibbles as shown in Figure 6, which can also be visualized as a mirroring of the state to its diagonal. Then, either by swapping two nibbles or by holding a nibble in its position, the global permutation can be mapped to structures that are very similar to the `ShiftRows` operation of `AES` or `SKINNY` and we can adapt some design strategies.

4.2   Bit-Serial Implementations of `PRESENT`

Data Path. We illustrate in Figure 7 the basic architecture of our bit-serial implementation of `PRESENT`. Similar to the bit-serial `AES` design described in Section 3, the 64-bit state of `PRESENT` is held in a shift register and shifted at every

clock cycle. Again, the white cells represent regular FFs, while the gray ones indicate the positions of scan FFs. During the initialization phase, the plaintext is provided starting from its MSB to its LSB of each nibble in the order of `0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15`. Hence, each nibble is provided within 4 clock cycles, starting from MSB to LSB and the entire plaintext is stored in the state register after 64 clock cycles starting from Nibble_0 to Nibble_{15}.

Similar to our bit-serial `AES` implementation, the addition of the round key is performed in a bit serial fashion using a single 2-input XOR gate. However, since `PRESENT` has a 64-bit state of 16 nibbles, only during the first 3 clock cycles, the result of the XOR-operation is fed into the state register. At the 4th clock cycle, the Sbox is applied and the result is saved in the last 4 bits of the state register (using the indicated scan FFs) while the remaining part of the state is shifted.

At the 16th clock cycle, the first stage of the permutation (local permutation) is applied to the last row in parallel to the 4th Sbox operation. The red lines in Figure 7 indicate the data flow that realizes the sorting of the bits according to their significance. Since this operation could be interleaved with the continuous shifting of the state register, we could save a few scan FFs for the last row.

After 64 clock cycles, the round key has been added, all 16 Sboxes have been evaluated, and each row has been sorted according to the local permutation. To finalize the round computation, the second stage of the permutation (global permutation) is performed in 4 clock cycles by means of the blue lines in Figure 7. In total, a full round of the cipher is performed in $4 \times 16 + 4 = 68$ clock cycles. After 31 rounds (2108 clock cycles), the ciphertext is returned as the result of the final key addition, whereby the next plaintext can be loaded into the state register simultaneously.

Key Path. The state register of the key update function is implemented as shift register, which is shifted and rotated one bit per clock cycle, similar to the state
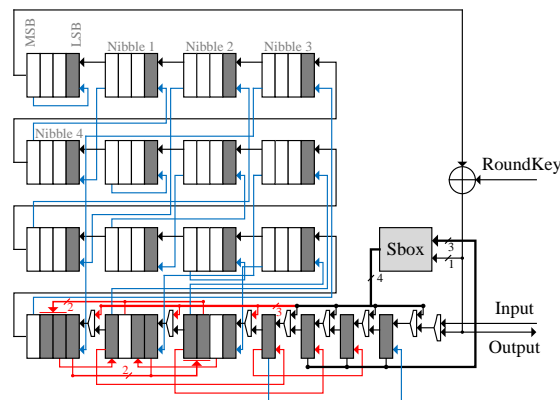

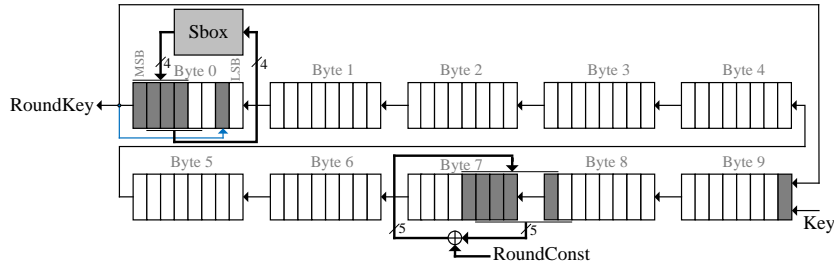**Figure 7:** Bit-serial architecture for `PRESENT` (encryption only, data path).

Figure 8: Bit-serial architecture for `PRESENT-80` (encryption only, key path).

of the data path (see Figure 8 for the 80-bit version). At each clock cycle, one bit of the round key is extracted and given to the data path module.

Besides, in order to derive the next round key, the current state has to be rotated by 61 bits to the left which can be done in parallel to the round key addition and Sbox computation of the data path. However, these operation takes 64 clock cycles in total, and the rotation of the round key needs only 61 clock cycles. Hence, we have to stop the shifting of the key register using a gated clock signal. However, since we would loose synchronization between key schedule and round function for the last 3 bits of the round key, we have to partition the key register into a higher (7 bits) and a lower part (73 bits). Then, after 61 clock cycles, the lower part is stopped, while the higher part still is rotated using an additional scan FF (see blue line in Figure 8) to provide the remaining 3 bits of the round key. Then, while the data path module performs the finalization of the permutation layer, the remaining 4 bits of the higher part are rotated to restore the correct order of the bits. In addition, during the last clock cycle, the round constant is added as well as the Sbox is applied (which is shared with the data path module[9]). Eventually, the key register holds the next round key and is synchronized with the round function in order to continue with the next round.

4.3 Extension to Higher Bit Lengths

In this section, we discuss necessary changes of our architectures to extend and scale the data path to higher bit lengths in order to increase the throughput and decrease the latency.

2-Bit Serial. Expansion of our 1-bit serial data path to a 2-bit serial one is straightforward. Essentially, every component is adapted such that it provides 2 bits at a time, i.e., the state register is shifted for two bits per clock cycle, while the Sbox is applied every 2 clock cycles. Similarly, the local permutation is performed every 8 clock cycles, and the finalization of the permutation takes another 2 clock cycles. Hence, an entire round is computed within $16 \times 2 + 2 = 34$ clock cycles, which is exactly half of the clock cycles of the 1-bit serial architecture.

---

[9]Again, necessary 2-to-1 MUX at the inputs are not shown.

Unfortunately, adaption of the key path to a 2-bit serial one is more complex. In particular the rotation of 61 bits is difficult since shifting 2 bits at a time does not allow a rotation of an odd number of bits. In order to overcome this issue, we decided to distinguish between odd and even rounds. During an odd round we use a rotation of 60 bits, while during even rounds the key state is rotated by 62 bits. However, this approach implies the need for additional multiplexers in order to select the correct round key as well as the correct positions to inject the round constant and the Sbox computation. Apart from that, the key state register is shifted 2 bits per clock cycle, still uses a gated clock signal for the lower part and a rotation of the most significant bits (eight or six, depending on the round) for synchronization.

4-Bit Serial. Further, we considered extending the data path to 4 bits using our bit-sliding technique and replacing all FFs of the state registers by scan FFs. Unfortunately, the bit permutation layer prevents an efficient scaling of our approach, which would result in an architecture that is even larger than the results reported in the literature (for nibble-serial implementations). In particular, the decomposition of the permutation layer, that allowed us an efficient realization for 1- and 2-bit serial data paths, is rather inefficient for nibble-serial structures. Although the global permutation could be realized using only scan FFs for the entire state, the local permutation would require additional multiplexers for the last row of the state. Eventually, performing the entire permutation in a single clock cycle after the substitution layer (as it is done in existing nibble-serial architectures), would be possible solely using scan FFs and without the need of further multiplexers. Hence, although our bit-sliding approach offers outstanding results for 1- and 2-bit serial data paths, it does not scale for larger structures and classical approaches appear to be more efficient.

4.4 Results

In Table 2 we report synthesis results and estimated power consumption of our designed architectures using the aforementioned five standard cell libraries based on various technologies (from 45nm to 180nm). We also report the results for the design published in [31] which is, to the best of our knowledge, the smallest PRESENT architecture reported in the literature. We emphasize again that we had access to the design sources from [31] and performed the syntheses using our considered libraries with the same set of parameters as for our architectures. It can be seen that our constructions outperform the smallest designs reported in the literature in terms of area and power.

5   Conclusion

In this paper, we have introduced a new ASIC implementation strategy, so-called bit-sliding, that allows to obtain efficient bit-serial implementations of SPN ciphers. Apart from the area savings due to a small data path, the bit-sliding

Table 2: Encryption-only `PRESENT` implementations for a data path of δ bits @ 100KHz.

Key	δ	UMC180		UMC130		UMC90		Ngate45		IBM130		Latency	Ref.
	bits	GE	μW	GE	μW	GE	μW	GE	μW	GE	μW	Cycles	
80	1	934	1.82	1006	0.44	872	0.32	1113	55.43	847	0.43	2252	**New**
80	2	1004	2.05	1096	0.47	949	0.33	1191	59.33	913	0.45	1126	**New**
80	4	1032	3.13	1088	0.53	990	0.33	1279	59.69	942	0.49	516	[31]
128	1	1172	2.41	1268	0.59	1090	0.43	1397	69.26	1065	0.57	2300	**New**
128	2	1265	2.61	1366	0.61	1189	0.44	1499	74.92	1150	0.58	1150	**New**
128	4	1344	4.00	1416	0.67	1289	0.53	1672	77.54	1230	0.71	528	[31]

strategy reduces the proportion of scan-flip flops to store the cipher state and key, greatly improving the performances compared to state-of-the-art area-optimized implementations.

We have successfully applied bit-sliding to `AES-128`, `PRESENT` and `SKINNY`, and in some cases reduced the area figures by more than 25%. Even though area optimization was our main objective, it turns out that power consumption figures are also improved, which indicates that bit-sliding can be used especially for passive RFID tags, where area and power consumption are the key measures to optimize, notably affecting the proximity requirements.

However, as for any bit-serial implementation, it is to be noted that energy consumption necessarily increases when compared to round-based implementations, due to the higher latency. Therefore, depending on the area available for security on the device, bit-sliding might not be the best choice for battery-driven devices. All in all, this work shows that for some scenarios, `AES-128` can be considered as a lightweight cipher and can now easily fit in less than 2000 GE.

References

1. Banik, S., Bogdanov, A., Regazzoni, F.: Exploring Energy Efficiency of Lightweight Block Ciphers. In Dunkelman, O., Keliher, L., eds.: Selected Areas in Cryptography - SAC 2015 - 22nd International Conference, Sackville, NB, Canada, August 12-14, 2015, Revised Selected Papers. Volume 9566 of Lecture Notes in Computer Science., Springer (2015) 178–194

2. Banik, S., Bogdanov, A., Regazzoni, F.: Atomic-AES: A compact implementation of the AES encryption/decryption core. In Dunkelman, O., Sanadhya, S.K., eds.: Progress in Cryptology - INDOCRYPT 2016 - 17th International Conference on Cryptology in India, Kolkata, India, December 11-14, 2016, Proceedings. Volume 10095 of Lecture Notes in Computer Science. (2016) 173–190

3. Banik, S., Bogdanov, A., Regazzoni, F.: Atomic-aes v 2.0. IACR Cryptology ePrint Archive **2016** (2016) 1005

4. Banik, S. and Pandey, S. K. and Peyrin, T. and Sasaki, Y. and Sim, S. M. and Todo, Y.: GIFT: A Small PRESENT. to appear in Cryptographic Hardware and Embedded Systems - CHES 2017 - Taipei, Taiwan, September 25-28, 2017

5. Beaulieu, R., Treatman-Clark, S., Shors, D., Weeks, B., Smith, J., Wingers, L.: The SIMON and SPECK lightweight block ciphers. In: Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE, IEEE (2015) 1–6

6. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS. In Robshaw, M., Katz, J., eds.: Advances in Cryptology - CRYPTO 2016 - 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part II. Volume 9815 of Lecture Notes in Computer Science., Springer (2016) 123–153

7. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An ultra-lightweight block cipher. In Paillier, P., Verbauwhede, I., eds.: CHES 2007. Volume 4727 of LNCS., Springer (September 2007) 450–466

8. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knežević, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçin, T.: PRINCE - A low-latency block cipher for pervasive computing applications - extended abstract. In Wang, X., Sako, K., eds.: ASIACRYPT 2012. Volume 7658 of LNCS., Springer (December 2012) 208–225

9. Boyar, J., Matthews, P., Peralta, R.: Logic minimization techniques with applications to cryptology. Journal of Cryptology **26**(2) (April 2013) 280–312

10. Canright, D.: A very compact S-box for AES. In Rao, J.R., Sunar, B., eds.: CHES 2005. Volume 3659 of LNCS., Springer (August / September 2005) 441–455

11. CMT: Circuit Minimization Team: http://www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html

12. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)

13. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. IEE Proceedings-Information Security **152**(1) (2005) 13–20

14. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON family of lightweight hash functions. In Rogaway, P., ed.: CRYPTO 2011. Volume 6841 of LNCS., Springer (August 2011) 222–239

15. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.J.B.: The LED block cipher. [24] 326–341

16. Hamalainen, P., Alho, T., Hannikainen, M., Hamalainen, T.D.: Design and implementation of low-area and low-power AES encryption hardware core. In: Digital System Design: Architectures, Methods and Tools, 2006. DSD 2006. 9th EUROMICRO Conference on, IEEE (2006) 577–583

17. Jean, J., Moradi, A., Peyrin, T., Sasdrich, P.: Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives – Applications to AES, PRESENT and SKINNY. Cryptology ePrint Archive, Report 2017/600 (2017)

18. Jean, J., Peyrin, T., Sim, S.M.: Optimizing Implementations of Lightweight Building Blocks. Cryptology ePrint Archive, Report 2017/101 (2017)

19. Juels, A., Weis, S.A.: Authenticating pervasive devices with human protocols. In Shoup, V., ed.: CRYPTO 2005. Volume 3621 of LNCS., Springer (August 2005) 293–308

20. Kocher, P.C.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Koblitz, N., ed.: Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings. Volume 1109 of Lecture Notes in Computer Science., Springer (1996) 104–113

21. Mentens, N., Batina, L., Preneel, B., Verbauwhede, I.: A systematic evaluation of compact hardware implementations for the Rijndael S-box. In Menezes, A., ed.: CT-RSA 2005. Volume 3376 of LNCS., Springer (February 2005) 323–333

22. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the limits: A very compact and a threshold implementation of AES. In Paterson, K.G., ed.: EUROCRYPT 2011. Volume 6632 of LNCS., Springer (May 2011) 69–88

23. Poschmann, A.: Lightweight Cryptography - Cryptographic Engineering for a Pervasive World. Cryptology ePrint Archive, Report 2009/516 (2009)

24. Preneel, B., Takagi, T., eds.: CHES 2011. In Preneel, B., Takagi, T., eds.: CHES 2011. Volume 6917 of LNCS., Springer (September / October 2011)

25. Rolfes, C., Poschmann, A., Leander, G., Paar, C.: Ultra-lightweight implementations for smart devices–security for 1000 gate equivalents. In: International Conference on Smart Card Research and Advanced Applications, Springer (2008) 89–103

26. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A compact Rijndael hardware architecture with S-box optimization. In Boyd, C., ed.: ASIACRYPT 2001. Volume 2248 of LNCS., Springer (December 2001) 239–254

27. Shannon, C.E.: Communication theory of secrecy systems. Bell Systems Technical Journal **28**(4) (1949) 656–715

28. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: An ultra-lightweight blockcipher. [24] 342–357

29. Wamser, M.S.: Ultra-small designs for inversion-based s-boxes. In: 17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014, IEEE Computer Society (2014) 512–519

30. Wamser, M.S., Holzbaur, L., Sigl, G.: A petite and power saving design for the AES s-box. In: 2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015, IEEE Computer Society (2015) 661–667

31. Yap, H., Khoo, K., Poschmann, A., Henricksen, M.: EPCBC - A Block Cipher Suitable for Electronic Product Code Encryption. In Lin, D., Tsudik, G., Wang, X., eds.: Cryptology and Network Security - 10th International Conference, CANS 2011, Sanya, China, December 10-12, 2011. Proceedings. Volume 7092 of Lecture Notes in Computer Science., Springer (2011) 76–97

32. Yap, H., Khoo, K., Poschmann, A., Henricksen, M.: EPCBC-a block cipher suitable for electronic product code encryption. In: International Conference on Cryptology and Network Security, Springer (2011) 76–97

33. Zhang, X., Parhi, K.K.: High-speed VLSI architectures for the AES algorithm. IEEE transactions on very large scale integration (VLSI) systems **12**(9) (2004) 957–967