Generic Architecture for Lightweight Block Ciphers: A First Step Towards Agile Implementation of Multiple Ciphers

Etienne Tehrani, Jean-Luc Danger, Tarik Graba

Generic Architecture for Lightweight Block Ciphers: a first Step towards Agile Implementation of Multiple Ciphers

Etienne Tehrani, Jean-Luc Danger and Tarik Graba

LTCI, Télécom ParisTech, Université Paris-Saclay, 75 013 Paris, France.
firstname.lastname@telecom-paristech.fr

Abstract. Lightweight cryptography is at the heart of today's security needs for embedded systems. The standardised cryptographic algorithms, such as the Advanced Encryption Standard (AES), hardly fits the resource restrictions of those small and pervasive devices. From this observation a plethora of Lightweight Block Ciphers have been proposed. Every algorithm has its own advantages in terms of security, complexity, latency, performances. This paper presents first a classification of some popular Substitution-Permutation-Networks (SPN) class of lightweight ciphers according to their architecture and features which share many common operators. From this last point, we studied a round-based generic hardware architecture that allows a security architect to dynamically change the lightweight cryptographic algorithms to be executed. The results of the ASIC implementation show that the configuration part of the proposed flexible architecture adds significant complexity. If compared with the parallel implementation of several algorithms, the complexity ratio becomes interesting when the number of algorithms (or the level of agility) increases. For instance, if we consider 6 SPN ciphers, the configurable architecture provides a complexity reduction of 62.5%, whereas there is no reduction with 4 algorithms.

Keywords: Lightweight cryptography, SPN, ASIC, Configurable Architecture.

1 Introduction

With the intense development of small and pervasive computing devices, as IoTs and their ability to communicate through insecure networks, it becomes essential to add security features. Indeed, from connected homes to connected vehicles or medical devices, security is at the heart of tomorrow's issues as it will affect our private lives as much as our health. Hence, all these devices need to be protected using sound cryptography in order to get a good level of confidentiality, integrity and privacy. Such devices are constrained by a low complexity with restricted chip area, memory and energy, while still needing communication channels. The security of these devices and the sensitive data they process have to be addressed by a new generation of Lightweight Cryptography algorithms which provide a good compromise between security and complexity in terms of area, latency and energy.

The most common standard symmetric-key cryptographic algorithm used to secure digital information is the Advanced Encryption Standard (AES) [22]. However, AES does not fit the strong restrictions of small embedded systems. This observation led to the development of Lightweight Block Ciphers. Plethora of such ciphers are available in the literature [7]. The NIST has started a project [19] to develop a strategy for the standardisation of lightweight cryptographic algorithms. Some of these algorithms are based on the same basic computation steps as AES since they use a Substitution Permutation Network (SPN) structure [17], some others use Feistel networks as DES [24]. Out of these numerous algorithms, none have emerged as a clear favorite in terms of becoming a NIST standard. PRESENT [8] has been accepted as an ISO standard [15], but the impact of standardisation was not as significant as AES. Actually, this means first, that the security of these algorithms is not officially recognised by an authority. Second, that some industrial may use modified versions of those algorithms, loosing interoperability and eventually weakening the algorithm. Third, since the demand for such algorithms is increasing and the maturity of some of them is becoming evident, the NIST might eventually standardise one of them rendering obsolete actual devices or protocol implementations. From these three last points, it appears that the possibility to use a flexible architecture allowing the user to change afterwords the cryptographic algorithm is an interesting feature. Moreover, this "agility" characteristic increases the security level as the attacker is thwarted by the unsteady nature of the cryptographic computing. The point is to know if such flexible architecture is feasible in terms of complexity and other physical constraints.

We could think this objective could be made possible thanks to the common operators of SPN algorithms. But the cryptographic algorithms were developed focussing on different features and the SPN operators have thus been implemented in different manners Most algorithms focus on area, others such as PRINCE [9] and MANTIS [6] focus on latency thanks to a specific structure, inspired by SPN, called $\alpha - reflection$. The objective of low energy consumption is tackled by MIDORI [3] and PICCOLO [23]. It arises the question of how similar those algorithms are, and if these similarities can be used to design a unique hardware to implement them. This would permit the deployment of those block ciphers, while being ready to switch as soon as a standard emerges. This Generic Architecture would also have the advantage of allowing algorithms to be slightly modified or tweaked if necessary. The basic idea to develop such an architecture is that the SPN structured algorithms share similar functions. Those functions can be grouped within *steps* which have the same function but do not work in the same way. These *steps* are further detailed in chapter 2.

An original fine-grain FPGA [20] has already been proposed. It is based on the utilization of Full-adder configurable cells. Other flexible cryptographic architectures have been proposed [25], [12] or [18] which focuses on an agile permutation layer but none of these targeted specifically lightweight cryptography. We propose in this paper to study a coarse-grain approach by taking into account the common operators found in the Lightweight algorithms. However, even if some operators are very close, the Feistel class of algorithms are too far from the SPN in terms of scheduling. Therefore, the study targets the SPN class only.

Our contributions. In this paper, we first propose a classification of lightweight cryptographic algorithms according to their internal scheduling and functions. From this classification and a detailed study of the differences between the selected SPN algorithms, a generic round-based hardware architecture has been proposed. It can be configured for different levels of agility in order to handle some or all families of ciphers. The permutation layer (P-layer) which is quite specific to each algorithm has been deeply investigated. A third contribution is the evaluation of the generic architecture and the comparison with implementations of independent lightweight cryptographic ciphers.

The paper is organised as follows. Section 2 presents the classification of the different families of SPN lightweight algorithms. It focuses on the different functions which require specific hardware modules. Section 3 describes the design of the generic architecture. The overall architecture alongside the design of each *step*. Section 4 justifies the different design choices of the P-Layer, as the architectural optimisations in the P-Layer are strongly related to the requirements of each family of algorithms. Section 5 discusses the results of the architectural implementations. The hardware resources usage for different levels of agility are presented and compared to direct implementation of the algorithms. Finally, Section 6 concludes the paper.

2 Classification of the algorithms

There are multiple ways to classify symmetric cryptographic algorithms, the main category used is the structure of the algorithm. An algorithm has either a Feistel structure (SIMON and SPECK [5]), an SPN structure (AES) or a structure derived from SPN (PRINCE [9], PICCOLO [23]).

Our study focuses on implementing SPN and SPN-like algorithms, as their functions are similar enough to be handled by the same hardware, and a large proportion of lightweight cryptographic algorithms use this structure. The SPN algorithms are composed of three *steps*:

- The Sbox: **S**, which corresponds to the confusion
- The P-Layer: **P**, which corresponds to the diffusion
- The AddKey: **K**, which corresponds to the addition of the secret

SPN Algorithms can differ by the order in which they execute those *steps*. This allow to classify them into three families based on the relative order of these *steps*. The first type is the SKP-type (SKINNY [6]), for which **S** is followed by **K**, itself followed by the P-layer. The second type is the SPK-type (MIDORI [3], GIFT [4], PRESENT [8]), for which **S** is followed by the P-layer, itself followed by **K**. On the one hand, each family can be implemented without changing the order, simply by changing the first *step* of the algorithm. On the other hand, the two families are not coherent and require extra modules to be handled by the same architecture. Note that it is also possible to handle both these families by adding computation to the key since the **P** and **K** are both linear layers. The last family is that of the $\alpha - reflective$ algorithms (PRINCE [9], MANTIS [6]) which change the order of the *steps* during the encryption.

Table 1 presents a classification proposal.

Table 1: Classification of Lightweight Ciphers sub-blocs

Algorithms			GIFT	PRESENT	SKINNY	MANTIS	PRINCE	MIDORI
Step order		SKP			X			
		SPK	X	X				X
		α reflection				X	X	
P-Layer	Permutation	bit-level	X	X				
		nibble-level			X	X	X	X
	Matrix Multiplication	None	X	X				
		4×4 matrix			X	X		X
		64×64 matrix					X	
Key Scheduling		Sub key extraction	A	B		C	C	C
		Rotation	X	X	X			
		Sbox		X				
		Round constant	X	X	X	X	X	X

The P-layer **P** has two main types of permutations, either at the bit-level (PRESENT [8], GIFT [4]) or at the nibble-level (SKINNY [6], PRINCE [9]). These two different approaches require more or less complex permutation modules and implementing one but not the other has an influence over the cost of the permutation. An additional matrix multiplication can be used. It can be a 4×4 multiplication of the nibbles (SKINNY [6], MANTIS [6]) or a 64×64 multiplication on the whole word (PRINCE [9]). PRESENT [8], GIFT [4] do not use matrix multiplication. Each type represents different levels of complexity, especially if we consider the resources to store the matrix values. Finally, though most algorithms only use one Sbox, $\alpha - reflective$ algorithms (PRINCE [9], MANTIS [6]) use two Sboxes: the Sbox and its inverse.

The **Key Scheduling** block is in charge of generating the round keys. Key Scheduling is done in very different ways from one algorithm to the other. Despite basic operations such as rotation or the use of round constants, the Key scheduling uses different functions applied to different parts of the key. Indeed, the sub-key generation is rather complex to unify as extracting different groups of bits of variety of size is hard to achieve in hardware without great increase in the design size. For example, GIFT [4] divides the entire key in 16-bit words and extracts two of them (A type), PRESENT [8] extracts two nibbles (B type) and use the Sbox. In other algorithms such as PRINCE [9] and MANTIS [6] the key is divided into two parts (C type) used separately. For these reasons, and because making the key scheduling generic in hardware would be complex and costly, we have decided to use a buffer that will contain pre-computed round keys. The round keys will be generated by software in the configuration phase.

Table 2 presents a selection of Lightweight Block Cipher and their implementations results from the literature.

Table 2: Comparison of Area, Latency and Throughput for implementations of 64-bit block size Lightweight Block Ciphers

Cipher	Ref.	Tech	Architecture	Area	Latency	TPmax	TP@100kHz
		(nm)	(cycle/round)	(GE)	(ns)	(Gbps)	(kbps)
Block size : 64-bit							
PRINCE	[9]	90	1/32	7996	13.9	4.56	-
	[9]	90	11/32	3286	58.3	1.10	-
PRESENT	[4]	90	32/32	1560	52.16	1.23	-
Midori64	[3]	90	16/16	2450	33.92	1.89	-
SKINNY 64	[6]	180	32/32	1696	59.84	0.95	177.78
	[6]	180	1/32	17454	51.59	1.24	6400
	[6]	180	128/32	1399	121.60	0.09	8.12
	[6]	180	2048/32	1172	2170.88	0.02	2.03
	[4]	90	32/32	1477	58.88	0.96	-
Mantis7	[6]	180	1/14	11209	20.5	3.12	-
	[6]	180	1/14	23926	11.0	5.82	-
GIFT-64	[4]	90	28/28	1345	51.24	1.25	-
	[4]	90	112/28	1113	239.68	0.06	-
	[4]	90	2048/32	930	4784.64	0.01	-

3 Generic Architecture

In order to handle multiple algorithms with the same hardware, the architecture is designed to be configurable.

The overall architecture is divided between the key scheduling and the configurable SPN structure. Each cipher computes the three steps of the SPN structure in a specific order, and can skip some of them (completely or at a specific round). In our design, this is achieved through the use of four 3-way multiplexers (see figure 1) which are controlled using configuration bits. These configuration bits need also to be different from one round to another.

For the desired functionality, there are only 15 different possible options to order the three steps, therefore all the multiplexers can be configured with only 4 bits per round. Thus, the number of necessary configuration bits is 4× the maximum number of rounds that the implemented ciphers need. This can be further reduced if we store a unique configuration for all the rounds that use the same configuration.

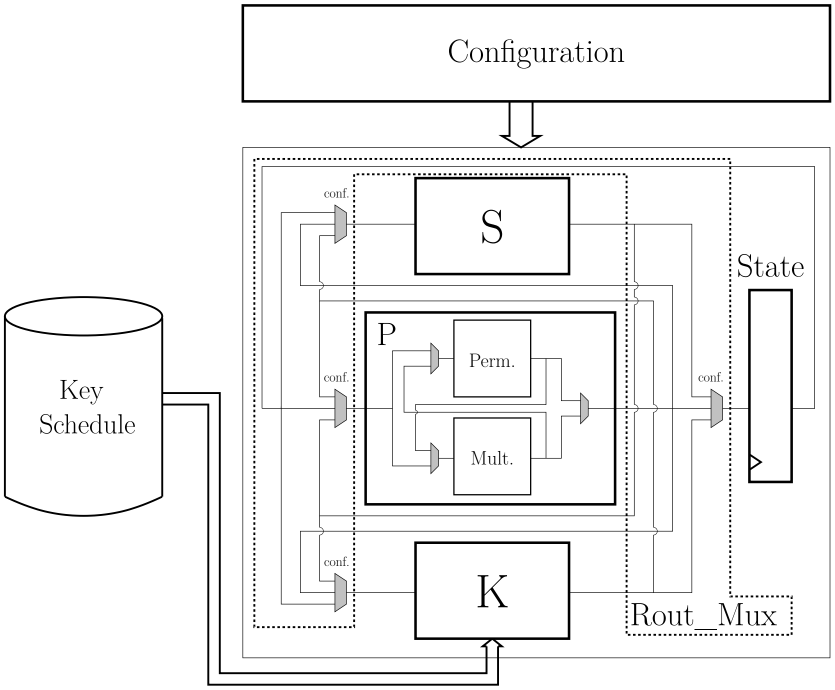


Fig. 1: Overall organisation of the Generic Architecture

3.1 Key Scheduling

Initially, as with the rest of the architecture, the key scheduling of each algorithm was studied in order to identify similarities between each of them. The issue is that unlike the algorithm itself, which uses a set SPN structure and is broadly similar to the standardised AES, key scheduling is achieved through a plethora of methods. Each of these methods applies calculation specific to the algorithm. An example would be isolating parts of the key, a few bits long, apply a certain processing such as an LFSR or the algorithm's Sbox to them, then each part is reordered and the result is shifted. Not only does this general description not fit most of the key scheduling but even if it were, being able to isolate different amounts of different-sized sample represents a real challenge and requires an important amount of hardware no matter the solution.

Once it had been made obvious that the variety of key scheduling does not allow a unified implementation at a reasonable cost, a different path had to be considered in order to obtain the used round keys. Using a configurable architecture means this architecture would most likely not work independently and thereby require to be included within a processor system. Computing the round keys could be done by software, prior to or in parallel with the configuration phase and the unrolled key could be stored in a specific buffer. Obviously this processor system, used for security applications, would have to ensure a secure way to store and handle the unrolled key. Moreover, as the key is generally not changed at each encryption, handling the Key Scheduling through software is not an issue.

3.2 The Sbox: S

The **S** module is rather straight forward, for each of the 16 possible nibble inputs there is an output defined through information stored within a RAM bloc. This module is therefore similar to a LUT. The substitution of each of the 16 nibbles is done in parallel and therefore requires 16 actual Sboxes. Most algorithms use the same Sbox throughout the entire encryption process but some of them require two different Sboxes. This is the case with $\alpha - reflective$ algorithms which use both an Sbox and its inverse, such as PRINCE [9] and Mantis [6]. Including a second Sbox means having to store twice as much information and add a mechanism to switch from one Sbox to the other.

3.3 The Key Addition: K

Each algorithm uses a different key for each round, called a round key, which is computed during the key scheduling. The *round key* considered here is not necessarily what algorithms call their round key, it also encompasses the round constant if any. Thereby, the **K** *step* is simply composed of 64 Xor gates in parallel to add the state to the software precomputed round key. The main security issue is that the round keys need to be stored in a secure environment, which is coherent with the natural use of encryption where the key needs to be stored securely. This mechanism also echoes the fact that the **K** needs to access this secure memory once per round, therefore once per cycle. Memory does not usually have this feature but this can be handled by using a bypass between this section of the memory and the **K** module. That part was not implemented and is a theoretical solution.

3.4 The P-Layer: P

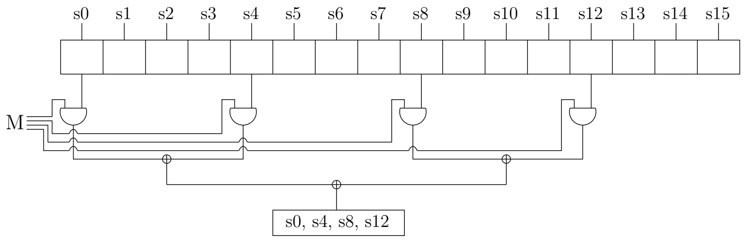


Fig. 2: The Generic Matrix Multiplication

The permutation bloc **P** for an SPN structured algorithm can include two different parts. The first part, equivalent to the MixColumn function of AES, is a matrix multiplication.

This matrix multiplication (see figure 2) uses the same four bits of the input with four different bit sets of the matrix to compute four bits of the output. This means that the matrix multiplication uses four times 64 bits values or a unique 256 bits value. This seems like a lot but it is still smaller then a full 64 × 64 matrix.

The second part of the **P** bloc is equivalent to the Shift Row function of the AES cipher. This part required a lot of attention for multiple reasons. First, in a classic Lightweight Bloc Cipher, this part generally implemented as a simple reordering of wires and uses no specific hardware. Second, the algorithms which do not use a matrix multiplication require permutation at a bit level rather than the at the nibble level as with AES's Shift Row. Third, designing a configurable permutation meant being able to route a signal through a crossbar-like module, but crossbars are expensive in terms of both area and configuration memory. It was therefore essential to find a lighter solution. The most efficient solution to optimise those crossbars was using Banyan switches [13].

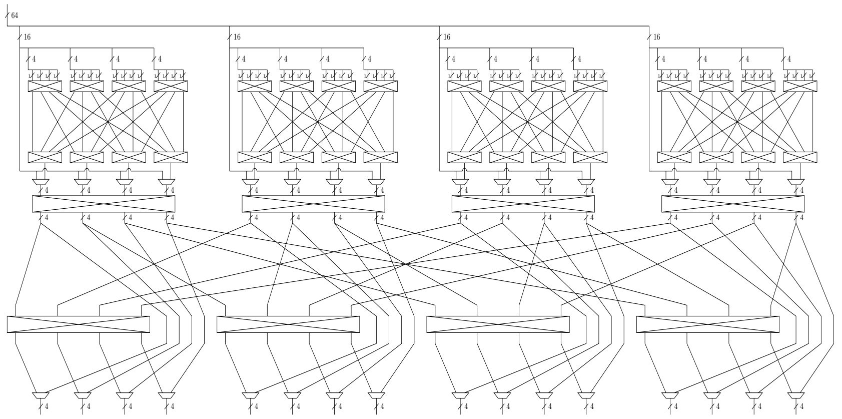


Fig. 3: The Generic Permutation Function

The chosen solution (see figure 3) was based on a thorough study of each algorithms' requirements in terms of permutation flexibility and the crossbar-like modules were optimised to better fit the situation (these optimised crossbar-like modules will be referred to as Banyan switches for the rest of the paper). More details on **P** will be given in section 4.

3.5 The Configuration

The generic architecture uses configuration parameters to select an algorithm. These parameters are set during the configuration phase, before the encryption begins. This allows the algorithm used to be changed dynamically.

The configuration memory bits are distributed throughout the architecture. A shift register mechanism is used (see figure 4) to limit the complexity of the external configuration interface. Once the shift register's content is valid, the

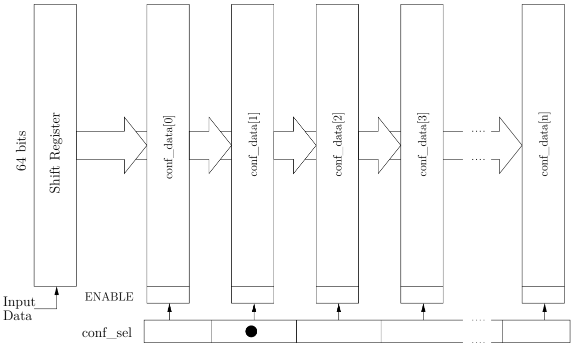


Fig. 4: The configuration scheme

configuration data is written by bloc in a configuration register chosen by a selection signal.

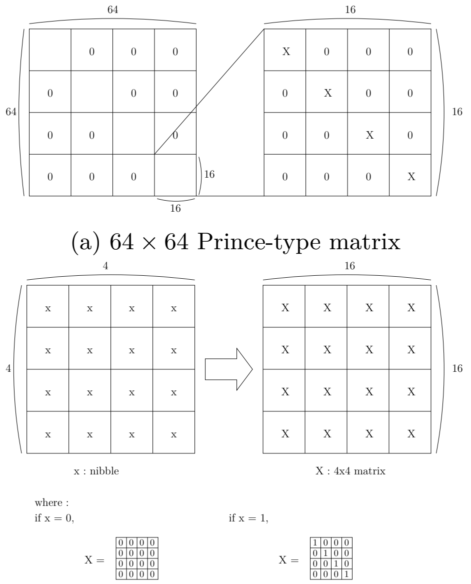
4 Detailed analysis of the P-Layer

Each module of the Generic Architecture has been designed to allow several levels of agility depending on the acceptable area overhead. This was achieved by identifying the common characteristics of different algorithms and designing the architecture accordingly. The most key part in unifying the architecture to each algorithm was **P**.

4.1 Unifying the Matrix Multiplication

The matrices used for the matrix multiplication are of two types. The first type is considering a 4×4 matrix composed of nibbles whose value is either `F` or `0` (SKINNY [6]). They are represented as 4×4 matrix but the multiplication actually applies to each bit of the state's 4×4 matrix of nibbles, they can therefore be considered as 4×4 matrix of *nibbles*. The second type is a 64×64 bit matrix which is mostly filled with 0s but has 4 16×16 sub-matrices composed of 1s and 0s (Prince [9]). The latter matrix are themselves composed of 16 4×4 diagonal matrices, therefore the 16×16 matrix can only have 1s placed along the 4×4 matrix's diagonals (see figure 5a).

This property makes it possible to reduce the 64×64 matrix to the information on the 4×4 matrix's diagonal which compose the 16×16 sub-matrix. There are 16 4×4 matrix in each of the 4 16×16 sub-matrices, each of which have 4 bits on their diagonal, which amount to a total of $4 \times 16 \times 4 = 256 bits$ of useful information. The first type only consists of a 4×4 matrix of nibbles with a single bit either at 1 or 0. In this type of matrix, the value of a bit is the same as the other bits of the same nibble, in other word the $4 \times 4 \times 4 = 64 bits$ of the matrix can be summed up as $4 \times 4 = 16 bits$ of useful information.



(a) 64 × 64 Prince-type matrix

(b) Turning the information of a 4 × 4 matrix to the information of a 16 × 16 matrix

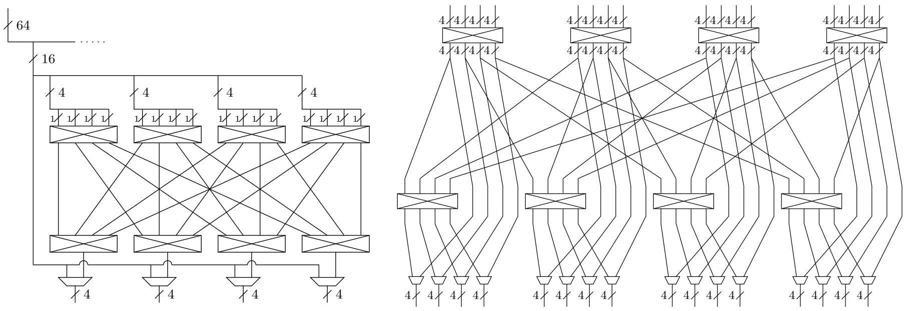
Fig. 5: Unifying the different types of Matrix Multiplication

This meant that either some algorithms were discarded in order to maintain a lower amount of required information, or they had to be harmonised. Doing so meant turning 16 bits of information into 256 bits without changing the result of the multiplication. This was achieved by changing every nibble of the 4 × 4 matrix into a 4 × 4 matrix, either filled with 0s if the nibble was a 0 or the 4×4 identity matrix if the nibble was a 1 (see figure 5b). The result is a 16×16 matrix with $4 \times 4 \times 16 = 64 bits$ of useful information, which is the same as one of the 16 × 16 sub matrix of the 64 × 64 matrix. The 16 × 16 matrix thereby obtained was then duplicated four times in order to have the 256 bits of useful information as with the 64 × 64 matrix. This choice is still costly as 256 bits is meaningful but is much less than the $64 \times 64 = 2048 bits$ of the entire matrix. The question remains nonetheless on whether adding the second type of matrix is worth the cost, this will be discussed later.

4.2 Minimizing the Cost of the Permutation

Permutation was a key issue as it is usually achieved by just reordering the wires. The overhead of making this bloc configurable could thus be significant. The simplest way to go is to consider a 64 × 64 crossbar which would allow any permutation but would be incredibly costly both in terms of area and in terms of the size of configuration memory. It was therefore essential to identify similarities between the different permutations in order to limit the area of this module.

The result was two levels of permutations, as explained beforehand, the bit-level permutation and the nibble-level permutation. The bit-level permutation (PRESENT [8], GIFT [4]) allows any layer-input bit to end up as any layer-output bit within the same 16-bit word, the restriction being that two bits from the same input nibble may not end up in the same output nibble. This restriction is coherent with the diffusion properties of a cipher as a bit level permutation needs to spread the information as much as possible in order to ensure the security. It requires four sets of Banyan switches each using the same parameters, which apply to each of the four 16-bit words of the 64-bits state.



(a) Bit-level permutation layer

(b) Nibble-level permutation layer

Fig. 6: Detailed permutation design

It is composed of two layers (see figure 6a). Between these layers, the connection wires are fixed and cannot be configured. They link each bit of a nibble to a different nibble. The first layer defines which bit of each nibble will be connected to which nibble of the second layer, through the use of a 4×4 Banyan switch for each nibble. The second layer reorders the bits within each nibble with a 4×4 Banyan switch for each nibble.

Nibble-level permutation (see figure 6b) works similarly and therefore, once again, any layer-input nibble may end up as any layer-output nibble and the restriction is that two nibbles of the same 16-bit input may not end in the same 16-bit word output. There is an exception to this rule in the case of an actual Shift Register where each nibble is reordered but every nibble stays within the same 16-bit word. It also requires two layers of Banyan switches separated with transition wires, which can be configured. There is a set of multiplexers which allows either to connect the four nibbles of a 16-bit word to four different 16-bit words or to keep each nibble within the same 16-bit word, which is needed for the Shift Row function. The first layer defines which nibble goes to which 16-bit word, and the second layer reorders each 16-bit word. They each use four 4-bit 4×4 Banyan switches.

Finally, the Banyan Switches are composed of a set of five switches (see figure 7). Each switch allows the reordering of two inputs and is controlled by a

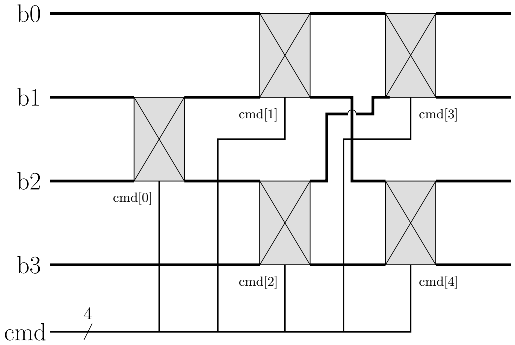


Fig. 7: Optimised crossbar-like module, the Banyan switch

single configuration bit. The 4 × 4 Banyan Switch can thus be configured to reorder its 4 inputs to get any permutation at the output. The $4 \times 3 \times 2 = 24$ permutations can be controlled with only 5 configuration bits. This structure allows to reduce the area in terms of logic and configuration memory.

5 Implementation Results

The architecture was implemented targeting the Cadence Free45PDK standard cells library. Post synthesis results are used to evaluate the area and complexity of our design.

First the generic architecture complexity is evaluated for different levels of agility. Second the generic architecture is compared to the cost of implementation of classic Lightweight Block Ciphers to identify the gain of using such an architecture.

Table 3: Cost of architecture's sub-parts for the level of agility **III**

Sub-part		Cost		Area percentage
		Area	GE	
Configuration	Route_Mux	695	678	26.9
	Sbox	348	339	
	Permutation	956	932	
	Multiplication	1390	1355	
	Other	87	85	
Route_Mux		2791	2720	21.6
S		1784	1739	13.8
P	Permutation	2059	2007	21.7
	Multiplication	744	725	
K		175	171	1.4

The Generic architecture can be divided in multiple sub-parts which have been presented in detail in chapter 3. The results of Table 3 show the cost of each of these parts. Making the architecture agile has important over-costs. For instance, Permutation is usually free in terms of area but making it configurable will obviously make it costly. It is also true for **S** which requires a configurable table and could not be optimised through the use of specific logic functions. The other two main parts are also new to such an architecture as they do not exist in a non-agile implementation of cryptographic algorithms. The Route_Mux allows to order each *step* at each round and therefore uses an important amount of multiplexers in order to select the path for the entire state. Finally, the most costly sub-part is the configuration which gathers all the parameters used to select which algorithm is implemented within the architecture. This last sub-part is divided between the different aspects which need configuration. It appears that the Multiplication requires the most important part of the parameters. Indeed, configuration of the Matrix multiplication has a cost of 256 bits (see section 4.1) which is a lot more that the 64 bits required for **S** or the 176 bits used to define the algorithm's route at each round. The overhead of the generic architecture is therefore important but most of it is due to the very nature of the architecture whose agility has a minimal cost which cannot be canceled. It would therefore seem that this architecture is not efficient when compared to a single algorithm but, the more algorithms it implements, the more interesting it becomes.

The next step was thereby comparing different levels of agility in order to identify how much adding new algorithms costs. This will then lead to a comparison between the cost of the generic architecture and the cost of implementing multiple algorithms.

Table 4: Different levels agility for the architecture

<table>
<tr><th>Algorithm</th><th colspan="3">Level of Agility</th></tr>
<tr><td>PRESENT</td><td rowspan="2"><b>I</b></td><td rowspan="4"><b>III</b></td><td rowspan="6"><b>IV</b></td></tr>
<tr><td>GIFT</td></tr>
<tr><td>SKINNY</td><td rowspan="2"><b>II</b></td></tr>
<tr><td>MIDORI</td></tr>
<tr><td>PRINCE</td><td></td><td></td></tr>
<tr><td>MANTIS</td><td></td><td></td></tr>
</table>

Each level of agility, **I**, **II**, **III** and **IV** from Table 4 allows the implementation of a certain set of algorithms. Each of these levels is compared to the cost of each of the algorithms it can handle in Table 5.

Figure 8 illustrates the complexity reduction provided by the generic architecture when the level of agility increases. It shows that balance is achieved around an agility of four algorithms and that once this limit is exceeded, the generic architecture offers a real gain.

Table 5: Comparison between different levels of agility and the sum of the algorithms it can implement

Level of Agility	Area of the Generic Architecture (in GE)	Sum of the Implementations (complexity ratio)
I	8494	1.72
II	8245	1.96
III	9212	1
IV	9631	0.625

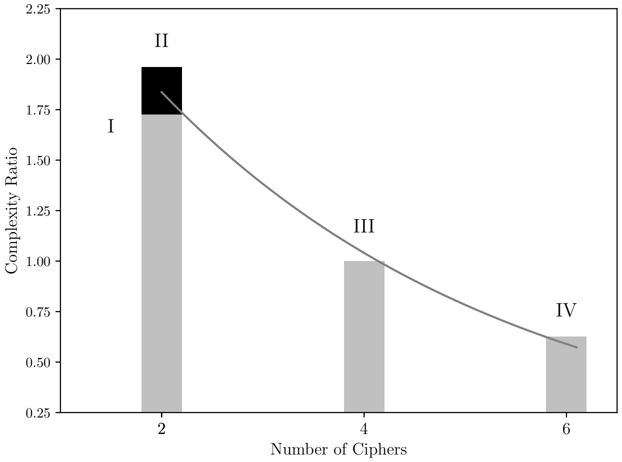


Fig. 8: Complexity Ratio for different levels of agility

6 Conclusion

An implementation of a round-based generic architecture of SPN lightweight ciphers has been presented. The results showed it was possible to compound multiple algorithms within the same architecture to provide agility features. The proposed architecture has the advantage of allowing to easily change the configuration at a round level and thus implementing the majority of SPN Lightweight algorithms.

However, the proposed architecture has a significant complexity cost, mainly due to the configuration logic. Compared to the complexity of a parallel implementation of different algorithms, we observe a complexity reduction if more than 4 ciphers are considered. The reduction can reach 62.5% if 6 different algorithms are considered. These results are promising if agility requirement is more important than complexity.

Apart from optimizing the design complexity by finding more optimal implementations for each sub bloc, a prospect is to search for ways to implement countermeasures against physical attacks. Indeed, specific countermeasures have to be implemented as they have to take into account the flexibility of our architecture without significantly increasing the global complexity.

References

1. Cryptographic technology guideline - lightweight cryptography (cryptrec-gl-0001-2016). Technical report, CRYPTREC Cryptographic Technology Guideline, 2017.
2. Stéphane Badel, Nilay Dağtekin, Jorge Nakahara, Khaled Ouafi, Nicolas Reffé, Pouyan Sepehrdad, Petr Sušil, and Serge Vaudenay. Armadillo: a multi-purpose cryptographic primitive dedicated to hardware. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 398–412. Springer, 2010.
3. Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 411–436. Springer, 2014.
4. Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.
5. Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.
6. Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In *Annual Cryptology Conference*, pages 123–153. Springer, 2016.
7. Alex Biryukov and Léo Paul Perrin. State of the art in lightweight symmetric cryptography. 2017.
8. Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer, 2007.
9. Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. ryptology ePrint Archive, Report 2012/529, 2012. `https://eprint.iacr.org/2012/529.pdf`.
10. Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 208–225. Springer, 2012.
11. Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: Noekeon. In *First Open NESSIE Workshop*, pages 213–230, 2000.
12. Adam J Elbirt and Christof Paar. An instruction-level distributed processor for symmetric-key cryptography. *IEEE Transactions on Parallel and distributed Systems*, 16(5):468–480, 2005.

13. L Rodney Goke and G Jack Lipovski. Banyan networks for partitioning multiprocessor systems. In *ACM SIGARCH Computer Architecture News*, volume 2, pages 21–28. ACM, 1973.
14. Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The led block cipher. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'11, pages 326–341, Berlin, Heidelberg, 2011. Springer-Verlag.
15. Neil Hanley and Maire ONeill. Hardware comparison of the iso/iec 29192-2 block ciphers. In *VLSI (ISVLSI), 2012 IEEE Computer Society Annual Symposium on*, pages 57–62. IEEE, 2012.
16. Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Tweaks and keys for block ciphers: the tweakey framework. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 274–288. Springer, 2014.
17. John B. Kam and George I. Davida. Structured design of substitution-permutation encryption networks. *IEEE Transactions on Computers*, (10):747–753, 1979.
18. Ruby B Lee, Zhijie Shi, and Xiiao Yang. Efficient permutation instructions for fast software cryptography. *IEEE Micro*, 21(6):56–69, 2001.
19. Kerry A. McKay, Lawrence E. Bassham, Meltem Sonmez Turan Report on lightweight cryptography (nist interagency/internal report (nistir) - 8114). Technical report, NIST Interagency/Internal Report (NISTIR), 2017.
20. Nele Mentens, Edoardo Charbon, and Francesco Regazzoni. Rethinking secure fpgas: Towards a cryptography-friendly configurable cell architecture and its automated design flow. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 215–215. IEEE, 2018.
21. AES NIST. Advanced encryption standard. *FIPS Publication*, 197, 2001.
22. Vincent Rijmen and Joan Daemen. The design of rijndael: Aes. *The Advanced Encryption Standard. Springer, Berlin*, 2002.
23. Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: an ultra-lightweight blockcipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 342–357. Springer, 2011.
24. Data Encryption Standard et al. Federal information processing standards publication 46. *National Bureau of Standards, US Department of Commerce*, 23, 1977.
25. R Reed Taylor and Seth Copen Goldstein. A high-performance flexible architecture for cryptography. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 231–245. Springer, 1999.