An efficient hardware data structure for cryptographic applications

Hala A. Farouk

Abstract**—** Cryptography currently plays a crucial role in the era where millions of people are connected to the internet and exchanging valuable and sensitive information. It is important for companies, banks, government departments and any other institution not only to create a secure connection over the ever-expanding networks but also not to slow down their system throughput by the implementation of these security solutions. Confidentiality, data integrity, authentication and non-repudiation are implemented using cryptographic algorithms. Applications for these algorithms are considered compute-intensive applications. Therefore, cryptographic algorithms are implemented in custom hardware seeking higher performance than the software implementation running on general-purpose processors. In this paper we present a new hardware data structure, namely the ShuffleBox. This hardware data structure is composed of simple registers and XOR gates. However, these components are connected in a certain way to allow fast implementation of important cryptographic procedures like permutation, affine transformation and rotation across a number of registers. The ShuffleBox is a rectangular array of bits that can store, XOR and rotate all bits in all directions. The hardware implementation that employs this hardware data structure achieves a speedup between 6x and 18x over conventional implementations.

Keywords**—** Cryptographic Architecture, Hardware Data Structure, Permutation, Rotation, Security Processor.

I. Introduction

CRYPTOGRAPHIC techniques provide a very strong data security infrastructure. However, they are composed of very special operations that are rarely found in any other application. These operations transform the input into a scrambled output, which can be recovered only using the appropriate key. These transforms are inherently complex and very compute intensive. They can consume a great deal of system resources if computed in software, i.e. on a general-purpose processor. Moreover, general-purpose processors' instruction sets are not optimized for these transforms. For these reasons, the current trend is towards customized hardware implementations of these security algorithms. However, hardware implementations have a broad spectrum. The hardware implementation can be as specific as a special circuit performing only one algorithm and can be as general as an instruction set architecture employing optimized instructions for cryptographic algorithms. Special circuits exploit all inherent parallelism in the algorithm being implemented. However, they lack flexibility and offer no control over their internal parameters. For example, if some vulnerability in the implemented algorithm is discovered, the system is risked to be useless unless a new circuit with the improved algorithm is installed. The custom hardware that implements only one algorithm is replaced by more flexible architectures. These architectures provide the computational data-path with more flexibility. The first step from custom hardware is reconfigurable devices, for example Field Programmable Gate Arrays (FPGA). FPGAs have abundant logic and routing resources, which can be configured or programmed to compute a large set of functions. These resources can be reconfigured or programmed post-fabrication. This feature compensate for the programmability of general-purpose processors. In reconfigurable devices, application parallelism is matched with as many function units as needed and with as many wires or buses as needed. FPGAs are considered fine-grained reconfigurable devices. They carry some drawbacks for being fine-grained [1]. Firstly, they need a large amount of configuration data which affects the configuration time. This is important in applications where the reconfigurable area is reconfigured during implementation and therefore the performance of the cryptographic application is affected by the configuration time. The other drawback is that they are general purpose. Any application can be implemented on an FPGA. Abundant routing resources are needed to maintain this abstractness. These resources increase chip area and consume power. Coarse-grained reconfigurable fabrics are developed to overcome the drawbacks of the fine-grained reconfigurable devices. The functional units in the coarse-grained reconfigurable devices are customized for the cryptographic applications [1,2]. These functional units implement the most common operations in any cipher. These common operations are as follows:

- Galois field addition and subtraction
- Galois field multiplication
- Logic operations like AND, OR and XOR.
- Shifting and rotation
- Table lookups

There are many research papers on the efficient implementation of these operations [3,4,5]. However, there is not enough concern with new hardware data structure that enhances performance. In this paper we present the ShuffleBox as a new hardware data structure that can enhance cryptographic systems' performance.

 Manuscript Received April 19, 2007; Revised October 21, 2007

II. THE SHUFFLEBOX

The ShuffleBox comprises a group of flip-flops connected in a grid as shown in Figure 1. Each square, except for the vertical and horizontal command registers, symbolizes a flip-flop. There are 16 command registers; eight for the horizontal rotation and are located at the left and another eight for the vertical rotation and are shown at the top. The vertical (horizontal) command register is composed of eight registers and each has three bits. The 24 bits are loaded into the vertical (horizontal) command registers in one cycle. The figure shows an 8x8 ShuffleBox just for the sake of explanation.

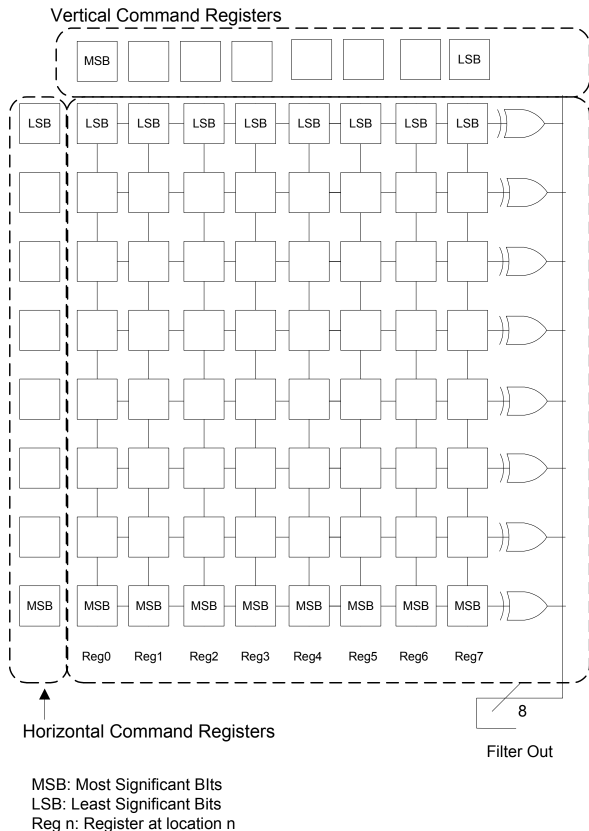

Figure 1 Abstract view of an 8x8 ShuffleBox

However, the size of the ShuffleBox should start from 64x64 for the implementation of the permutations in the DES algorithm and blowfish algorithms, for example.

In the following sections, examples are given for the implementation of some of the AES algorithm's transformations and also for an implementation of permutations employing the ShuffleBox.

A. Applications of the ShuffleBox in the AES Algorithm

There are two transformations in the AES algorithm that benefit the new data structure, the ShuffleBox. The two transformations are the ByteSub and the ShiftRow [6]. The ByteSub transformation is composed of two functions. First, taking the multiplicative inverse in GF(2^8) of every byte and then applying an affine transformation defined by:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \bullet \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

This affine transformation can be implemented by using vector x five times in different rotations and then bitwise EXORing the five results. Finally, the result is EXORed with the hexadecimal value 63. This procedure requires the implementation of five shifters and five 8-bit EXOR units or the execution of the shift and the EXOR instruction five times on vector x. This routine is performed on every byte in the data-block and the key-block and in every round of the algorithm.

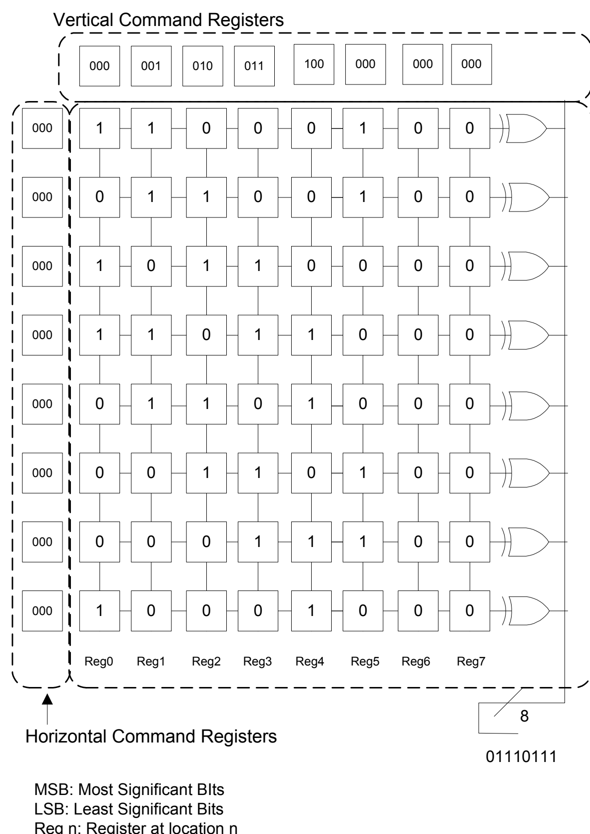

Figure 2 ShuffleBox implementing the affine transformation

The ShuffleBox simplifies the implementation of the above described routine. First, the multiplicative inverse of the input byte is loaded into five consecutive registers inside the ShuffleBox. Then, the vertical command registers is loaded with the value 053800_h in hexadecimal format. The value 63_h is also loaded into the sixth register, namely Reg5. The output bus (FilterOut) carries the output of the affine transformation after the rotation is enabled. An example is shown in Figure 2. The input data byte is 02_h. The multiplicative inverse of 02_h, which is $8D_h$, is loaded into the first five registers. The vertical command register is loaded with the value 053800_h. The figure shows the register values and the output after the rotation, which is 77_h. The rotation in all register is done in one cycle. The rotation is a down rotation according to the value written in the vertical command registers and a right rotation according to the value written in the horizontal command register.

The algorithm implementation using the ShuffleBox speeds up the execution six times over architectures with registers, ALU and Shifter. The ShuffleBox takes however five times the area of a register file of the same size, one EXOR unit and one shifter. Higher speedups are noticeable in algorithms that need bit replacements in more than one register or a rotation between bits in a certain position in more than one register as required by the ShiftRow operation in the AES algorithm.

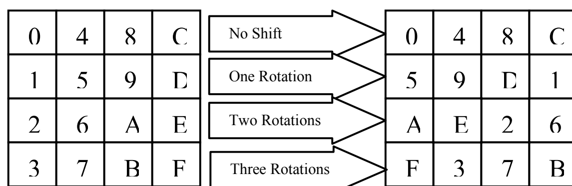

Figure 3 The ShiftRow transformation in the AES algorithm

In the AES algorithm bytes are organized over four rows as shown in Figure 3. The execution of such a shifting across the bytes is actually swapping bytes between the words. Therefore, this transformation requires 12 cycles on a machine with no special hardware and without taking the loading into registers into account. However, the execution of this transformation using an 8x8 ShuffleBox requires three cycles and using 32x8 ShuffleBox would require only one cycle without taking the register loading into account. As we have mentioned above, the flip-flops inside the ShuffleBox are connected vertically as well as horizontally. Therefore, the rotation in either direction takes only one cycle.

B. Permutations using the ShuffleBox

Keyed permutation can be easily implemented using the ShuffleBox in two cycles without the cycles of loading the registers. Assume that the value M=6E in hexadecimal needs to be permutated as follows:

$M_5\ M_2\ M_7\ M_1\ M_3\ M_0\ M_4\ M_6$

This permutation will transform the 6E input value into D9 hexadecimal value.

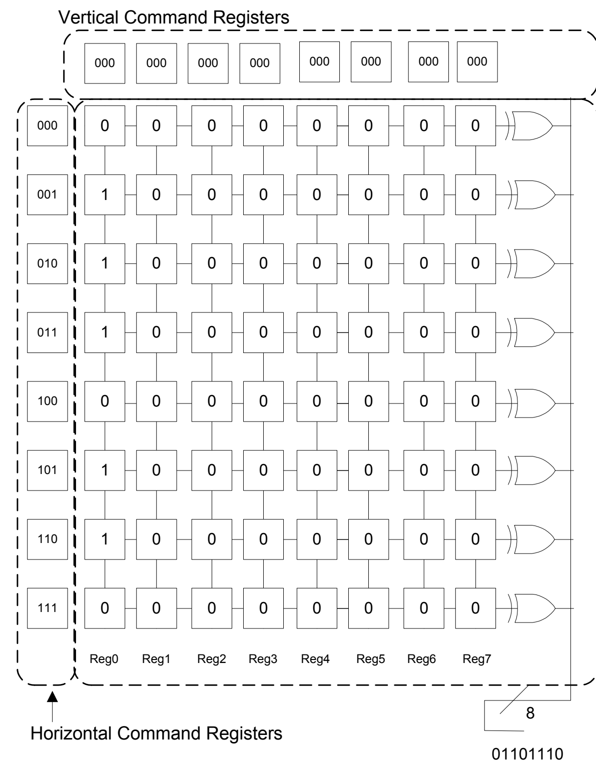

Figure 4 ShuffleBox state before any rotation for the permutation operation

The horizontal command register is loaded with successive values from 0 to 7 and the first register is loaded with the value $6E_h$ as shown in Figure 4.

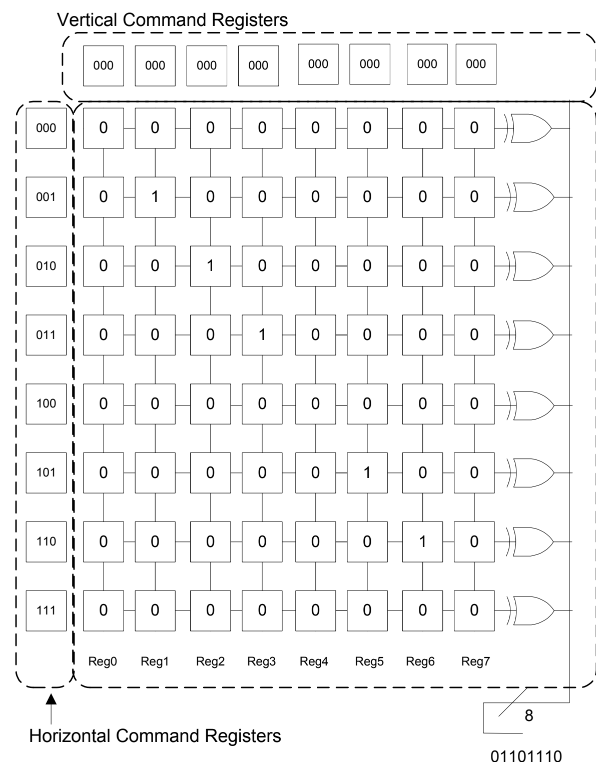

Figure 5 The ShuffleBox after the horizontal rotation

Each row of the eight rows is rotated according to the value on its left in the horizontal command register. This rotation will spread the bits over ShuffleBox so that each column holds only one bit of the input data as shown in Figure 5.

The vertical command register can be loaded during the rotate operation in the second cycle. The signal of vertical rotation is then asserted and each column is rotated down n positions according to the value in the vertical command register.

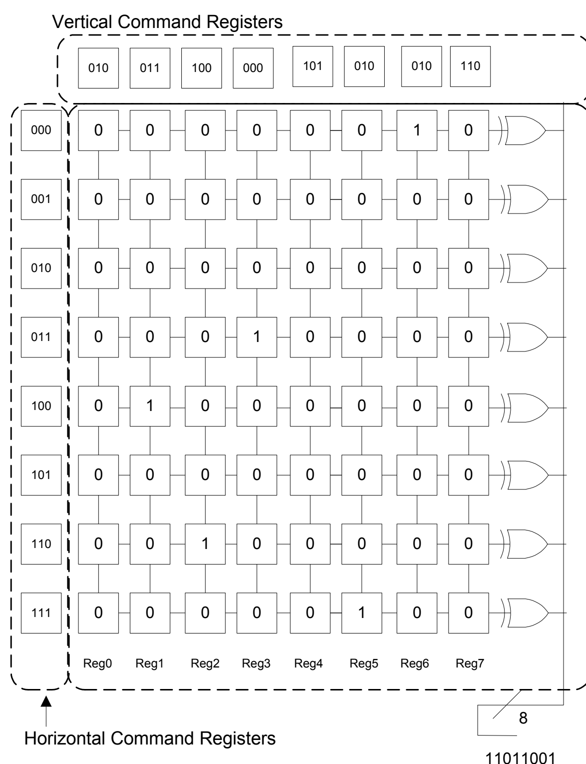

Figure 6 The ShuffleBox after the vertical rotation

The values in the vertical command register correspond to the number of times each bit is displaced in the permutation formula. In our example, the permutation follows this pattern M_5 M_2 M_7 M_1 M_3 M_0 M_4 M_6. The data bit that was in the zero location before the permutation moves after the permutation two places to the left. Therefore, the most significant register in the vertical command registers is loaded with the value 010 in binary format as shown in Figure 6. The same is performed on each bit to achieve the final permutation. The output on the FilterOut bus shows the value $D9_h$ as explained before.

III. ShuffleBox Implementation

Table 1 shows the number of slices consumed for the implementation of the 8x8 ShuffleBox and the 64x64 ShuffleBox. The table shows also the number of slices needed for a plain register file of the same size. The implementation is performed on the VirtexII pro40.

Table 1 Comparison of the circuit areas for the different sizes of ShuffleBoxes and Register Files

	Area in Slices
ShuffleBox 8x8	329
ShuffleBox 64x64	16403
Register file 8x8	48
Register file 64x64	3177

Table 2 Comparison of performance measures

	Performance in Cycles
AES Key Schedule on Pentium Pro	305
AES Key Schedule on security processor with ShuffleBox	59

Table 2 shows the number of cycles needed for the execution of the key schedule routine of the AES [7] on an instruction set architecture with and without the ShuffleBox.

IV. Conclusion

Cryptographic architectures are crucial nowadays in offloading the main system processor from the encryption, authentication and other security related functions. There are protocols to implement security in all network layers. Therefore, offloading the main processor of these functions enhances the overall performance [8]. The security processor that takes over the cryptographic functions must be optimized for such a task. Special hardware for Galois field algebra is being designed as dataflow components. These components are then either multiplied and assembled together to perform the desired algorithm or they constitute the instruction set of an application-specific instruction set architecture. Including a specialized hardware data structure into any kind of cryptographic engine can enhance its performance. In this paper we provided a new hardware data structure, namely the ShuffleBox. We also gave examples for its operation and how it speeds up basic and important cryptographic functions.

References

[1] R. R. Taylor, S.C. Goldstein, “A high-performance flexible architecture for cryptography,” *Proceedings of the 1st Workshop on Cryptographic Hardware and Embedded Systems*, Worcester, MA, USA, August 12–13, 1999. pp. 231–245.

[2] Kang Sun, Xuezeng Pan, Jiebing Wang, Jimin Wang, “Design of a novel asynchronous reconfigurable architecture for cryptographic applications,” *IMSCCS (2)*, 2006, pp. 751–757.

[3] M. Fayed, M. W. El-Kharashi, F. Gebali, “A low-area, high-speed, processor array architecture for field ALU over GF (2^m),” *ITI 5th International Conference on Information and Communications Technology*, Cairo, Egypt, December 16–18, 2007.

[4] A. . Elbirt and C. Paar, “An FPGA implementation and performance evaluation of the serpent block cipher,” *Eighth ACM International Symposium on Field Programmable Gate Arrays*, Montrey, California, February 10–11, 2000.

[5] A. J. Elbirt, W. Yip, B. Chetwynd, C. Paar, “An FPGA implementation and performance evaluation of the AES block cipher candidate algorithm finalists,” *Proc. 3rd Advanced Encryption Standard (AES) Candidate Conference*, New York, April 13–14, 2000.
[6] J. Daemen and V. Rijmen, "AES proposal: Rijndael," *1st Advanced Encryption Standard (AES) Conference*, California, USA, 1998.
[7] K. Gaj and P. Chodowiec, “Comparison of the hardware performance of the AES candidates using reconfigurable hardware,” *Proc. 3rd Advanced Encryption Standard (AES) Candidate Conference*, New York, April 13–14, 2000.
[8] A. Dandalis, V. K. Parsanna, “An Adaptive Cryptographic Engine for Internet Protocol Security Architectures,” *ACM Transactions on Design Automation of Electronic Systems*, Vol.9, No.3, July 2004. pp. 333–353.