Rijndael Fast Implementation in Reconfigurable Hardware

Ahmed R. Darwish
School of Computer Engineering
George Mason University
adarwish@gmu.edu

Abstract—This paper presents an evaluation of the Rijndael Fast Algorithm in Xilinx FPGA. The Rijndael Fast Algorithm was initially proposed for fast software implementations in 32-bit processors. Throughput improvement from the original algorithm is estimated at about 100 times.

The Rijndael Fast Algorithm using 8x32 bit T-Boxes was first analyzed in hardware by V. Fischer and M. Drutarovský on the Altera CPLD series. The purpose of this paper is to present an analysis of the Rijndael Fast Algorithm Implementation on the Xilinx FPGA series, and compare the results of the analysis with those produced from Altera CPLD. The paper also compares the Fast Implementation throughput improvements over the original Rijndael algorithm in software to those achieved in Hardware.

Although the results obtained by the Fast Implementation in Hardware are faster than the original algorithm, the conclusion of this paper is that the gain in speed is counteracted by a multiplicative increase in the FPGA area, which makes the fast algorithm implementation in hardware uneconomical.

Index Terms—Cryptography, Field programmable gate arrays, Hardware design languages, Design Automation, Software

I. INTRODUCTION

Between 1997 and 2000 NIST was involved in the selection of a new standard to replace DES and 3DES, due to the vulnerability of DES, the low performance of 3DES in software. NIST required an algorithm that would offer at least 3DES security, but would offer much better performance in software and hardware.

Rijndael’s minimum number of rounds for making differential and linear cryptanalysis less practical than exhaustive key search is 6 rounds, giving it a security margin of at least 25% when using a 128-bit key (10 rounds) and 66% against the best estimated attack.

Among the five different candidates, Rijndael was selected by NIST for AES implementation. The designers of the Rijndael algorithm submitted the algorithm along with its possible software implementation in C. They have also proposed various algorithm implementation modes.

These implementations modes included: 1) the basic implementation of the algorithm using four main functions of: Bye Substitution, Shift Row, Mix Column and Add Round Key, 2) An 8-bit processor implementation for Smart Cards, 3) A 32-bit processor implementation suggested for the highest achievable software throughput.

V. Fischer and M. Drutarovsky in their paper [1] have compared the implementation of the Basic Mode and Fast Mode (based on the 32-bit processor implementation) in Reconfigurable Hardware.

The purpose of this paper is to analyze the advantages and disadvantages of using the Fast Implementation as compared to the Basic Implementation in Reconfigurable Hardware, and to present some preliminary information on my testing procedure for the Fast Implementation as part of a hardware implementation cryptology class project.

The analysis of the two implementations will be based on the results achieved in [1] for the Fast Implementation and other results achieved in [3] for the Basic Implementation using both an Iterative Architecture and Inner and Outer Round Pipelining.

II. OVERVIEW

A. Cipher Architecture

The standardized Rijndael algorithm is based on 128-bit blocks of input plaintext/ciphertext with the options of 128, 192 or 256 bit-key. The algorithm is based on an SP network structure with substitutions and permutations implemented.
iteratively with the number of rounds depending on the key-length. The iterative rounds are 8, 10 or 12 for key sizes of 128, 192 and 256 respectively. This structure is depicted in figure 1 below.

In addition, there are an initial and final round, with the initial round for encryption being a simple round key addition, and the final round is comprised of three of the four main functions, that is Byte Sub, Shift Row and Add Round Key.

<table>
<thead>
<tr>
<th>Encryption</th>
<th>Decryption (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round Key[0] ➔ Add Round Key</td>
<td>Round Key[Nr] ➔ Add Round Key</td>
</tr>
<tr>
<td>i=1 ➔ Byte Sub</td>
<td>i=NR ➔ Inv Byte Sub</td>
</tr>
<tr>
<td>i=+1 ➔ Shift Row</td>
<td>i=+1 ➔ Inv Shift Row</td>
</tr>
<tr>
<td>Add Round Key</td>
<td>Add Round Key</td>
</tr>
<tr>
<td>Mix Column</td>
<td>Inv Mix Column</td>
</tr>
<tr>
<td>Shift Row</td>
<td>Shift Row</td>
</tr>
<tr>
<td>Byte Sub</td>
<td>Byte Sub</td>
</tr>
<tr>
<td>Shift Row</td>
<td>Shift Row</td>
</tr>
<tr>
<td>Round Key[Nr] ➔ Add Round Key</td>
<td>Round Key[0] ➔ Add Round Key</td>
</tr>
</tbody>
</table>

![Fig. 1. Structure of Rijndael cipher a) encryption algorithm b) standard decryption algorithm.](image)

In a table lookup implementation, the only non-linear function (Byte Sub) should be the first operation in a round. However, as depicted in fig. 1 b) above, in the standard decryption algorithm Inverse Byte Sub is the last operation. In[2] it is shown that the order of Byte Sub and Inverse Shift Row can be changed without affecting the algorithm.

Also, since Inv Mix Col is a linear transformation, the following equation is valid:

\[
Inv Mix Column (d \oplus K) = Inv Mix Column (d) \oplus Inv Mix Column (K).
\]

From the above properties, the standard decryption algorithm can be changed to its modified form in fig. 2.

![Fig. 2. Structure of Rijndael modified decryption algorithm](image)

**B. Cipher Operations**

**Byte Sub ad Inverse Byte Sub**

Rijndael uses 8x8 S-Boxes for such operation, 16 S-Boxes are required for encryption and 16 for decryption. The implementation of the S-Box in reconfigurable hardware can be realized via CLB blocks or Block Select RAM. Block Select RAM is preferable as S-Boxes consumes a lot of CLB resources. However, the choice of the implementation depends on the particular FPGA resources. For example, Xilinx FPGA series is rich in CLB resources as compared to Block RAM, while the reverse is true for the Altera devices. As shown, in [3], using RAM for S-Boxes in pipelined architectures should be discouraged, as such architecture consume a lot of RAM resources in pipelining, and thus S-Boxes would be better realized via CLB(s) in this case for an efficient implementation.
These implementation are simple permutations that can be implemented by utilizing routing resources and reordering CLB interconnections. The ordering is different between encryption and decryption.

**Mix Column and Inverse Mix Column**

These are the most complex operations in Rijndael as they involve multiplication in the Galois Field \(2^8\). Also, both encryption and decryption are treated differently.

Since these functions include multiplication by constants in the Galois field it can be shown as per [1] and [3] that they can be implemented using multiple layers of XOR gates.

### III. Rijndael’s Two Alternative Implementations in FPGA

In this paper we focus on the Fast Implementation architecture and then compare it with the Basic Implementation. A possible realization of the Rijndael implementation is shown in fig. 3.

**Fig. 3 Possible realization of T-Box Encryption Unit**

As shown in the above figure, The Fast Implementation can be realized via several RAM blocks using the above Iterative Looping architecture.

The Iterative Looping architecture is the only possible architecture for the Fats Implementation, as the Loop Unrolling architecture would require a huge amount of RAM far above the FPGA available Block RAM, while the Inner and Outer Round Pipelining would not be applicable for the Fast Implementation since there are no pipelining stages that exist in this simple RAM architecture that can achieve any gain in throughput. Thus, from now our focus will be on only the Iterative Looping architecture.

The first round is a simple round key addition to the input block, which is realized via a simple XOR.

Then, each Rijndael iterative round transformation can be represented as:

\[
e_j = T_0[a_{0,j}] \oplus T_1[a_{1,j-1}] \oplus T_2[a_{2,j-2}] \oplus T_3[a_{3,j-3}] \oplus K_j
\]

Where \(K_j\) is the round key in round j, and the T-boxes are represented as:

\[
T_0[a] = \begin{bmatrix} S[a] \cdot 02 \\ S[a] \\ S[a] \cdot 03 \end{bmatrix}, \quad T_1[a] = \begin{bmatrix} S[a] \cdot 02 \\ S'[a] \\ S[a] \end{bmatrix}, \quad T_2[a] = \begin{bmatrix} S[a] \cdot 03 \\ S[a] \\ S[a] \cdot 02 \end{bmatrix}, \quad T_3[a] = \begin{bmatrix} S[a] \\ S[a] \cdot 03 \\ S[a] \cdot 02 \end{bmatrix}
\]

Thus we can implement these rounds using a Single 256x32 Block RAM for each T-Box, and XOR these T-Boxes together along with the round key to get the round output states, which can then be fed to the multiplexer to index the block RAM(s) in the next cipher round. This will require a total of 16 256x32 Block RAM(s) for the 16 T-Boxes formed by the four state equations for the encryption unit. The decryption unit will require another 16 KB of Block RAM, resulting in a total of 32 KB RAM for the iterative rounds for both encryption and decryption.

The final round is different from the iterative rounds in that only three functions are implemented: Byte Sub, Shift Row, and Add Round Key, while the Mix Column function is skipped. This round could then be implemented via CLB. Alternatively, by realizing that S-Boxes can be obtained from T-Boxes as shown in the above tables where S(a) is multiplied by one, then this round can also be implemented via new T-Boxes realized via Block RAM. However, this round alone would cost 16 KB for encryption and 16 KB for decryption, which is equal to the amount of RAM consumed by the n iterative rounds, thus the achieved gain in speed/area will not be optimal as compared to the previous rounds. The total RAM requirements for this case would b 64KB for both encryption and decryption.

Key Scheduling can be achieved via computing the round keys in the fly using CLB resources. Alternatively, it can be computed in the fly using T-Boxes as well resulting in an extra 32K of block RAM for both encryption and decryption.

The third alternative, will be to pre-compute the round keys and store them in Block RAM which will require much less
amount of memory as compared to the T-Box implementation of the key scheduling algorithm (NR+1) 128-bit keys or a maximum of 3840 bits for 256-bit keys for both encryption and decryption. This alternative could be useful when large bulks of data are going to be encrypted and the keys do not need to change frequently. However, in applications which require fast key changes, such as encrypting ATM cells with different keys such approach would not applicable.

IV. COMPARISON OF BASIC AND FAST IMPLEMENTATIONS OF Rijndael

In this section we analyze the results obtained in [1] and [3] for both Rijndael implementations using the Basic and Fast Implementations.

<table>
<thead>
<tr>
<th>Logic Elements Used</th>
<th>Speed Mbits/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast (T-boxes, 128 bit blocks)</td>
<td>750</td>
</tr>
<tr>
<td>Fast (S-boxes, 128 bit blocks)</td>
<td>612</td>
</tr>
<tr>
<td>NSA</td>
<td>606</td>
</tr>
<tr>
<td>GAJ</td>
<td>414</td>
</tr>
<tr>
<td>DAN</td>
<td>353</td>
</tr>
<tr>
<td>ELB</td>
<td>300</td>
</tr>
<tr>
<td>MUT</td>
<td>248</td>
</tr>
<tr>
<td>Economic (S-boxes, 64 bit blocks)</td>
<td>212</td>
</tr>
<tr>
<td>Economic (T-boxes, 32 bit blocks)</td>
<td>115</td>
</tr>
</tbody>
</table>

Table 1: Comparison of Basic vs. Fast Implementations using Iterative Looping Architecture

As shown in Table 1 above, the Fast Implementation outperforms the S-Box implementation in terms of throughput. However, the throughput gain is not that significant, especially when compared to the Fast S-Box implementation (a factor of 1.2). The Fast Implementation is optimized for software implementations and it has been reported that throughputs up to 100 times faster in software than the basic implementation can be achieved.

For applications which require very high speeds such as 10 Gigabit Ethernet or network-based gateways with OC-192 or 10 Gig-E speeds (though these might usually use ASIC instead), the increase in performance as achieved by the Fast Implementation by [1] would not be of avail. Instead, architectures that use Inner and Outer Round Pipelining would be more capable of achieving such high speeds to support line rate performance, such as the results reported by [3] of 12.2 Gbps throughput using an Inner and Outer Round Pipelining Architecture.

From the above analysis, it is clear that the Fast Implementation does not produce significant gain in throughput as compared to the Basic Implementation. For Reconfigurable Hardware with limited RAM resources as compared with CLB resources, such as Xilinx FPGA, it will be a clear disadvantage to use this architecture, as it will consume critical resources in the FPGA that could be better used for alternative implementations, as for example, for pipelining registers, that could achieve much better performance than the T-Box based implementation.

For Reconfigurable Hardware with limited CLB resources, and large RAM resources, such as Altera FPGA, it would be interesting to compare the results of the T-Box implementation vs. Inner and Outer Round Pipelining as can be maximally achieved using such RAM resources and CLB. Although [1] has reported results for the T-Box based implementation, they have not reported any testing of Inner/Outer Round Pipelining implementation of the Basic Implementation.

Such a comparison could be a good research to conclude on the usefulness of the Fast Implementation for different Reconfigurable Hardware devices.

VI. SIMULATION AND SYNTHESIS OF FAST IMPLEMENTATION IN XILINX FPGA

In this section, I present preliminary tests implemented as part of the research project for a Cryptology class.

VHDL was used for implementing the code for the Fast Implementation scenario. The code implemented the Fast Implementation using 32K RAM for both encryption and decryption as well as another 32K RAM for the Final Round, to test for the maximum achievable throughput. The round keys were pre-computed and used Internal Memory for storing both the encryption and decryption keys.

Initially, a behavioral model using sequential logic was used, and the simulation was performed using the NIST Test Vectors in FIPS 197 publication. The simulation tests were OK. However, the synthesis resulted in a Fully Unrolled design. Despite, the fact that a clock was applied to the T-Box equation (presented in section 3 ) within every loop iteration. The RAM synthesized was still that of a Fully Unrolled design. The conclusion of this test is that RAM should be separately clocked, in order for it to be synthesized for a synchronous iterative architecture.

Next, the code was redesigned with RAM specifically instantiated via the Xilinx Synthesis library. Although this
might limit the code’s portability into other FPGA vendors, it is not the purpose of this exercise to be portable, but rather to test the performance of the Rijndael Fast Implementation on Xilinx FPGA. Besides, such attributes can be easily changed using different vendors’ libraries without large impact on the code itself.

The code was redesigned using concurrent VHDL constructs with parallelism in mind, to achieve the highest throughput. The tests have not been completed yet, to conclude on Fast Implementation performance on Xilinx FPGA.

CONCLUSION

This paper briefly reviewed the Rijndael algorithm and discussed the Rijndael Fast Implementation in Reconfigurable Hardware and compared it to that of the Basic Implementation.

The conclusion of the analysis is that the Fast Implementation is more optimal for software rather than hardware implementations as it does not produce significant gain in throughput as compared to the Basic Implementation.

For Reconfigurable Hardware with limited RAM resources as compared with CLB resources, such as Xilinx FPGA, it will be a clear disadvantage to use this architecture.

For Reconfigurable Hardware with limited CLB resources, and large RAM resources, such as Altera FPGA, it would be interesting to compare the results of the T-Box implementation vs. Inner and Outer Round Pipelining as can be maximally achieved using such RAM resources and CLB.

ACKNOWLEDGMENT

I would like to thanks Professor Kris Gaj and Pawell Chodoweic for their support in this project.

REFERENCES

[1] Viktor Fischer, Milo’s Drutarovsk’y, “Two Methods of Rijndael Implementation in Reconfigurable Hardware”.
[2] FIPS 197 standard by NIST. As well as in “J. Daemen and V. Rijmen, AES Proposal: Rijndael, AES Algorithm Submission, September 3, 1999”.