

FPGA implementation of improved version of the Vigenere cipher

Massoud Sokouti¹, Babak Sokouti², Saeid Pashazadeh¹ and Leili Mohammad Khanli³

¹Faculty of Electrical and Computer Engineering, University of Tabriz, Tabriz, Iran;

²Faculty of Engineering, Islamic Azad University-Tabriz Branch, Tabriz, Iran;

³Faculty of Mathematical Sciences, University of Tabriz, Tabriz, Iran.

m_sokouti@yahoo.com

Abstract

The use of cryptography has become increasingly important in recent years. Currently there are several good methods for encryption like AES and DES. Both of these algorithms require several rounds to encrypt a relatively small block of data. Stream ciphers, like Vigenere and Caesar in particular, only require one round. The Vigenere and Caesar ciphers, however, can be easily broken. Improved version of the Vigenere algorithm is obtained by adding random bits of padding to each byte to diffuse the language characteristics and this make the cipher unbreakable. In this paper we will present an efficient method for hardware implementation of the improved Vigenere algorithm.

Keywords: Cryptography; Vigenere algorithm; FPGA.

Introduction

The use of cryptography in modern day communication systems is becoming increasingly important as more and more people making use of electronic transactions (Dreyere, 2004). As we know AES and DES are good ways for encryption but the only big problem of these two ciphers is that the key needs several rounds to encrypt a small block of data. For a long period of time, the data encryption standard (DES) was considered as a standard for the symmetric key encryption. This standard has a key length of 56 bits. For the time being, this key length is considered small and can easily be broken. For this reason, the National Institute of Standards and Technology (NIST) announces, after a competition between 15 algorithms, that the Rijndael cipher will replace the DES cipher and will become a new advanced encryption standard (AES). The Rijndael cipher has three possible block and key lengths: 128, 192 or 256 bits. Therefore, the problem of breaking the key becomes more difficult (Stallings, 2003). The advanced encryption standard (AES) is based on arithmetic block cipher in finite Galois field, $GF(2^8)$, and is a symmetric block cipher that encrypts 128-bit plaintext data with a 128 bit, 192 bit, or 256 bit cipher key (NIST, 2001). As we know AES requires 11 rounds. Since Nov 2001, various AES implementations using ASICs or

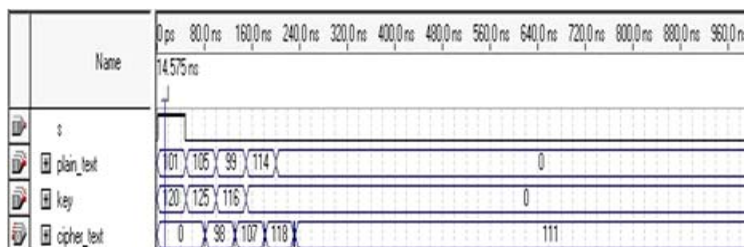
FPGAs have been reported some of them are focusing on small chips and some are focusing on high throughput IVV is a good choice for encryption because it uses the Vigenere cipher and it needs only one round (Sokouti & Sokouti, 2009). In this paper we will implement the IVV (Sokouti & Sokouti, 2009) by FPGA. This simulation is done in Quartus II 6 and the devices which are used are Startix.

Vigenere

This cipher is one of the best known of manual polyalphabetic ciphers and is named after Blaise de Vigenere. Just like the other classic ciphers we have a message that we want to encrypt which is named plaintext and another stream for encrypting which is named key. After encrypting plaintext and the key a message is produced that is named cipher text. In this cipher we should repeat key characters in order to make the length of the key as same as the length of the plain text. In other words, having keywords length m , an alphabetic character can be mapped to one of m possible alphabetic characters (assuming that the keyword contains m distinct characters). Such a cryptogram is called polyalphabetic cryptosystems.

If the message is an English text containing n characters with all punctuation and spaces removed, and if we use the correspondence $a = 0, b = 1... z = 25$, for

Fig. 1. Sample Vigenere encryption. Cipher text is our output after encryption, and clk is our clock, and key is the key of the Vigenere cipher, and plain_text is the text which should be encrypted, s is starting simulation.



plaintext $p_1, p_2... p_n$ and key $k_1, k_2 ... k_n$, the i^{th} character of the cipher text is given by this method (Piper & Murphy, 2002; Bishop, 2003; Stallings, 2003):
 $C_i = (p_i + k_i) \text{ mod } 26$
 We can see a sample simulation

Table 1. Diffusion of random bits, r is a random bit; m is the bit value of the message.

F(x)	Byte 1	Byte 2
0	rmmmmmmm	mrrrrrrr
1	rmmmmrmm	mrrrrrrr
2	mrmmmmmm	mrrrrrrr
3	rmrmmrmm	mrrrrrrr
4	mrrmmrmm	mrrrrrrr
5	rmrmmrmm	mrrrrrrr
6	mrrmmrmm	mrrrrrrr
7	rmrmmrmm	mrrrrrrr

with 20 MHz clock which is meant 120 Mbps as in Fig. 1. As we see after getting the first character, the encryption operation is started.

Review of improved version of Vigenere algorithm (IVV)

In this cipher we want to diffuse random bits between the bits of the plaintext and after that encrypt it by the Vigenere cipher. This makes the cipher text length double. There is a function $F(x)$ to find that how to diffuse

the Table 1 (Sokouti & Sokouti, 2009; Sokouti *et al.*, 2009).

$$F(x) = (g^x + c) \text{ mod } p \quad (1)$$

According to Table 1:

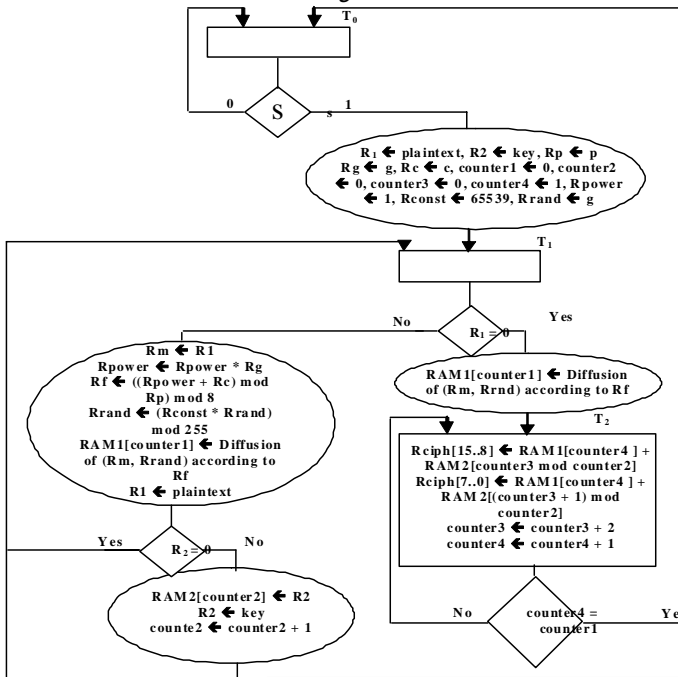
In $F(x)$ function if $g = 10$, $c = 7$, $p = 11$ then $F(0) = 0$, $F(1) = 6$ as shown below:

rmmmmmmm mrrrrrrr mrrmrrmr mrrmrrmr

The implementation of the IVV

We can see the ASM chart diagram in Fig. 2 and also the implementation of it in Fig. 3. For implementing this method we need 5 registers for input consisting of R plain for the plaintext characters, R key for Vigenere key characters, Rp for the value p, Rg for the value of g and Rc for the value of c. R plain and R key gets the characters in every clock until the user stops entering the plaintext and the key characters. While the system is getting the plain text characters the system produces the random numbers and calculates the $F(x)$ function for every character. Producing the random numbers and calculating the $F(x)$ function is one clock late from getting the plain text characters so we will use another register for the plaintext named Rm. This helps us to have the plain text characters and the random number and the value of $F(x)$ in the same clock. To implement the producing of random numbers we need 3 registers consisting if Rrand1 (this register first will be initialized by the value of g but during getting plaintext characters is initialized by the numbers which are produced by the Rrand2), Rconst. which is set by a big constant number for eg. 65539 and Rrand2 is initialized by the random

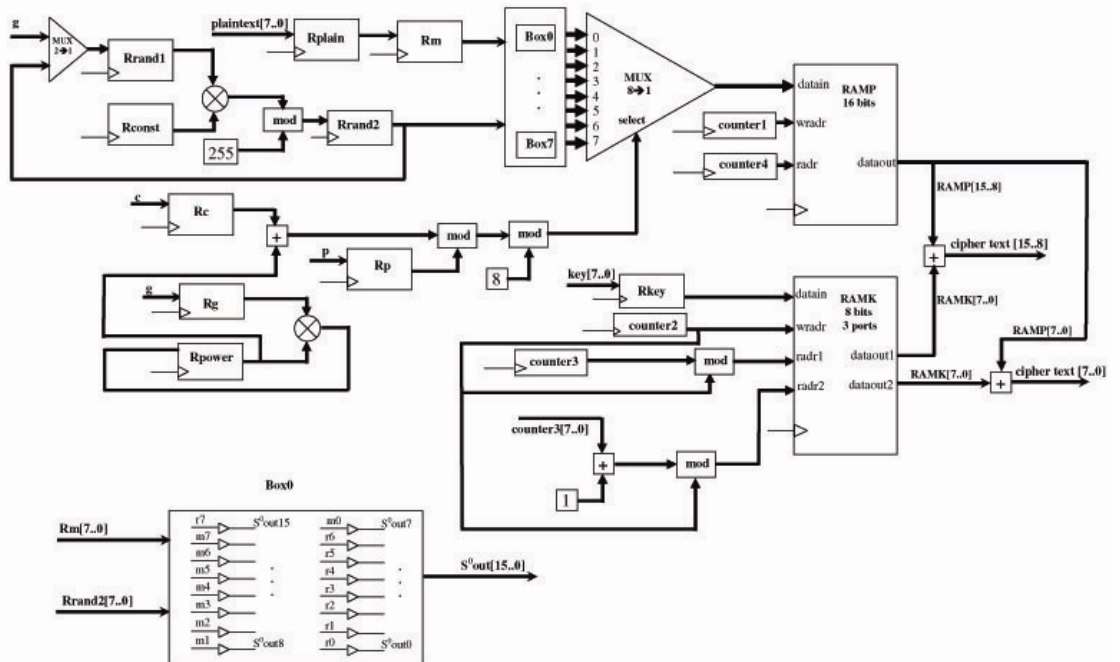
Fig. 2. ASM chart of the IVV



the random bits between the bits of plaintext. One way function, $F(x)$, consisting of a prime, p , a generator less than p , g , and a positive constant less than eight, c . Prime number p should be more than the length of the plain text to prevent the possible detection of cycles. The eq. (1) shows $F(x)$ where x represents the $(n-1)$ th character of plain text that is started from 0. To reduce

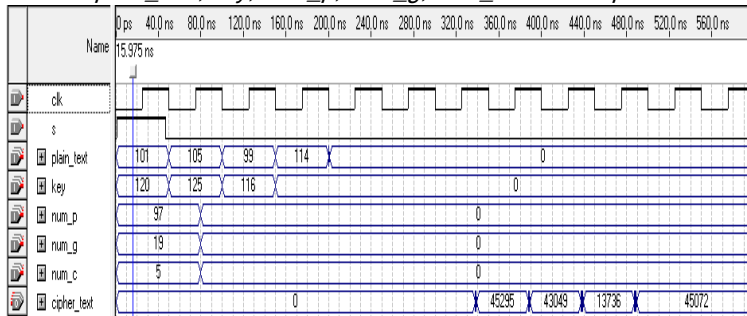
the size of the pad to a reasonable number, $F(x)$ is reduced by performing $F(x) \text{ mod } 8$. This allows the $F(x)$ range from 0 to 7. After calculating $F(x)$ we should use

Fig. 3. The implementation of IVV



numbers by implementing $(Rrand1 * Rconst) \text{ mod } 255$. Now we have produced the random numbers. For calculating $F(x)$ we will use 4 registers consisting of Rc,

Fig. 4. Simulation of the method: cipher_text is the output, clk, s, plain_text, key, num_p, num_g, num_c are the inputs.



Rg, Rp that we talked about them in the input part and Rpower that is a register which is initialized by the powers of g (for eg. in g^x it will calculated for every value of x in $(0 < x < (\text{size of plain text}-1))$ in every clock. This register is in the first

time to 1. Initializing Rpower or calculating g^x is easy by implementing $Rpower * Rg$. Now producing $F(x)$ is easy by implementing $((Rpower + Rc) \text{ mod } Rp) \text{ mod } 8$. Now we should use a box consisting of 8 boxes (box0... box7) and also a multiplexor with eight 8-bit inputs and one output for implementing Table 1 for the diffusion of plain text bits and the random bits. We can see the implementation of one of the 8 boxes in fig. 3. As we see the input of the box is $Rm[7..0]$ or plaintext bits and $Rrand2[7..0]$ or random bits (two 8bits) and the output is $S^0out[15..0]$ (16 bits). Box_i will diffuse the bits when $F(x) = i$. To choose one of the boxes which are implementing Table1 we will use the multiplexer $8 \rightarrow 1$ and it's selector is $F(x)$. The output of multiplexer is our new 16 bit plaintext. Now it is time to save the new 16 bit plaintext in a 16 bit RAM (M4K) and also saving the 8 bit key in an 8 bit 3_port RAM (M4K). While the system is getting the plaintext characters it gets the key characters too and when the user stops typing the key characters the system stops getting the key characters but continues getting plaintext characters until the user stops typing the plaintext characters. The next step is implementing the Vigenere algorithm. As we know Vigenere cipher encrypts every 8 bit plaintext with every 8 bit key but now we have 16 bit plaintext so that, we have used a 3 port RAM to have access to every 2 characters of the key or 16 bits. If we suppose the cipher text is $cihertext[15..0]$ and suppose the output of the RAM 16 bit is $RAMP[15..0]$ and the output1 of RAM 8 bit is $RAMK1[7..0]$ and the output2 of RAM 8bit is $RAMK2[7..0]$ the cipher text[15..0] will be calculated by $cihertext[15..8] \leftarrow RAMP[15..8] + RAMK1[7..0]$, $cihertext[7..0] \leftarrow RAMP[7..0] + RAMK2[7..0]$.

Discussion

As we see this cipher is different from the Vigenere cipher because it changed the characteristic model of characters. Also it increases the length of the cipher text and diffuses the bits of the plaintext by random bits. These random bits have a top rule in our cipher because by using them no one can guess which bits are related to the plaintext and which bits are random and we can easily say it is an NP-hard problem (Impaglizzo & Levin, 1990). The other basic

Table 2. Comparison of IVV and the published works.

Design	Devices	Memory for key	Memory for core	Cycles	fclk (MHz)	Th (Gbps)	Eff (Mbps/ K_bit)
IVV(Sokouti & Sokouti, 2009)	EP1S10F484C5	4 K-bit (1 M4K)	4 K-bit (1 M4K)	3	20	0.312	80
Standaert et al. (2003) (unrolling)	XCV 3200E-8	80 K-bit (20 BRAMs)	320 K-bit (80 M4K)	21	-	11.77	36.78
Saggese et al. (2003)(unrolling)	XVE 2000-8	80 K-bit (20 BRAMs)	320 K-bit (80 M4K)	50	-	20.3	63.44
UF10-PP3B (Zambreno et al. 2004)(Unrolling)	XC2V 4000	80 K-bit (20 BRAMs)	320 K-bit (80 M4K)	30	-	23.5	73.44
UF1-PP0B (Zambreno et al. 2004) (unrolling)	XC2V 4000	8 K-bit (2 BRAMs)	32 K-bit (8 M4K)	10	-	1.41	44.06
Helion, 2009 (rolling)	Starix -C5	8 K-bit (2 M4K)	32 K-bit (8 M4K)	10	-	1.4	43.75

BRAM- Block selected RAM (4 K-bit)

advantage of this cipher is using the $F(x)$ function, so that we will have different kinds of encrypting for a character. Also this cipher uses 256 character of ASCII code. It is very difficult to break this cipher because we use 3 keys (p, g & c) with the Vigenere key and also to distinguish the place of random bits we have to compute the $F(x)$. (Sokouti & Sokouti, 2009) If we think about it we will see it's an NP_hard problem.

Our method has some drawbacks too. According to Table 1 we can get that this method doubles the size of the text and for areas with low band width or limited storage capacity this cipher is not suitable. The other drawback is the number of calculations for every character. So the number of calculating the $F(x)$ function is length of the text. Also we need a good random generator in $F(x)$ in order to make the cipher effective. All of the bits in this method are important so, if we miss only one bit the whole sent item will be trashed and we can guess that we can't use this method in image processing.

As we see this system stores the data at first and then it starts encryption so, after finishing plaintext it takes 3 clocks to see the cipher text. The advantages of this implementation is that it has a great random generator; the power operation and calculating $F(x)$ and also diffusion is done and stored just as we get the plain text.

Performance and comparisons

We evaluated the performance of our method and compared with the unrolling AES implementation of other published works as there are no implementations regarding our design or along side our improved algorithm in the literature. We will compare our implementation with the other unrolling implementations. We have implemented our implementations in Quartus II

6 for synthesis, place & route and timing analysis. Finally, we used Quartus II 6 simulator to test the logical operation and to do worst-case timing analysis for the design in the target FPGA. The maximum clock rate f_{clk} was obtained by Quartus II 6 simulator. In Table 2, the upper row shows our implementation and the other rows are the implementation of published works. The column "Device" denotes the FPGA used. The column "Memory for Key" denotes the amount of memory utilized for the key expansion, and "Memory for core" denotes the amount of memory utilized for the core, where one M4K is equivalent to 4096 bits. The column "Cycles" denotes the number of clock cycles for the process of the implementations. The column "Th" denotes the maximum throughput calculated by:

$$Th = 16 \times f_{clk} \quad (2)$$

The column "Eff" shows the memory efficiency calculated by:

$$Eff = \frac{\text{Throughput (Mbps)}}{\text{Memory_for_core (Kbits)}} \quad (3)$$

In the Altera startix device family (Altera, 2010), EP1S10F484C5 has the minimum devices, containing 60 M4Ks. The EP1S20F780C5 and contains 82 M4Ks. We will use EP1S10F484C5 for our implementation. As shown in Table 2, our method achieved 320 Mbps with 1 memory block (M4k). The amounts of memory utilized for our method is reduced by 98.75% and the memory efficiency is improved by 16.56%. Direct comparison among various FPGA implementations of the AES algorithms is difficult, since FPGA target devices are usually different. However, many AES implementations have provided the maximum throughputs and the amount of the memory utilized for the core. Thus, we can compare the memory efficiency defined in eq. (3). Compared with the published unrolling AES implementations signified with '(unrolling)' in the column 'Design', the memory efficiency of our method is very close to the fastest implementation (UF10-PP3B). Note that the number of cycles for UF10-PP3B is 30. Besides, the amounts of memory utilized for the core of our method are reduced by 98.75%. Compared with the published rolling AES implementation signified with '(rolling)' in column 'Design', both the throughputs and the memory efficiencies of the proposed implementation is much higher than the fastest rolling implementation of AES (UF1-PP0B). Our implementation has much higher throughput than the software implementations. An AES rolling implementation achieves 1.538 Gbps on a 3.2 GHz Pentium 4 processor (Limpma, 2003) and a 640 Mbps on a 1 GHz embedded processor (Nadehara *et al.*, 2004). Table 3 summarizes the features of the two different architectures: unrolling, rolling in the column 'memory efficiency' let the value of Helion (Helion, 2009) be 1, since, it is approximately the same as the value of UF1-PP0B (Zambreno *et al.*, 2004). We can see that our

implementation offers high memory efficiency and a high throughput using low memory area.

Conclusion

We presented architecture for an improved version of Vigenere encryption processor and implemented this design on Altera Startix EP1S10F484C5 FPGA. It achieves a throughput of 320 Mbps by using a M4K and key memory by using a 3 port M4K compared with the unrolling AES implementation that achieves a throughput of 20.48 Gbps by using 80 M4Ks. IVV implementation improves the memory efficiency by 16.56% and reduced the amount of memory by 98.75% and fits on less expensive FPGAs. Our next work will be on implementing this cipher algorithm in combinational form.

References

- Bishop M (2003) Computer security art and security. Person education Inc., NY.
- Dreyere C (2004) A pipelined implementation of AES for altera FPGA platforms, *ECE575, Fall*. 1-5.
- Helion Technology Ltd. (2009) High performance AES (rijndael) cores for altera FPGA, <http://www.heliontech.com/aes.htm>.
- Impaglizzo R and Levin L (1990) No better ways to generate hard NP instances than picking uniformly at random foundations of computer science. *Proc. Of 31th IEEE Annual Symp. On Foundations of Computer Sci.* 2, 812-821.
- Limpma H (2003) AES implementation speed comparison, <http://www.tsc.hut.fi/~aes/rijndael.html>.
- Nadehara K, Ikekawa M and Kuroda I (2004) Extended instructions for the AES cryptography and their efficient implementation. *IEEE workshop on signal processing system (SIPS'04)*, Oct. 13-15, FA-1.3.
- NIST (2001) Advanced encryption standard (AES), *Federal Inform. Proc. Standards Pub.* 197 (FIPS197). <http://www.altera.com>
- Piper F and Murthy S (2002) Cryptography: a very short introduction. Oxford Univ. Press.
- Saggese G P, Mazzeo A and Strollo AGM (2003) An FPGA-based performance analysis of the unrolling, tiling, a pipelining of the AES algorithm. *FPL 2003. LNCS 2778*, 292-302.
- Sokouti M and Sokouti B (2009) Improved version of Vigenere algorithm, *12th conf. of ISCEE*, 13-15.
- Sokouti M, Sokouti B and Pashazadeh S (2009) an approach in improving transposition cipher system. *Indian J. Sci. Technol.* 2(8), 9-15.
- Stallings W (2003) Cryptography and network security: Principles and practice, Prentice Hall, 3rd ed.
- Standert FX, Rouvory G, Quisqater JJ and Legat JD (2003) Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs. *Proc. of CHES 2003. Lecture notes in computer Sci.* 2523, 334-550.
- Stinson DR (2005) Cryptography theory and practice, 3rd ed. University of Waterloo, Ontario, Canada.
- Zambreno J, Nguyen D and Choudhary AN (2004) Exploring area/delay tradeoffs in an AES FPGA implementation, *FPL 2004, LNCS3203*, 575-585.

Table 3. Comparison of different architectures.

Architecture	Memory area	Throughput	Memory efficiency
Unrolling	Large	High	0.84 ~ 1.68
Rolling	Small	Low	1
IVV	small	High	1.83