Key Generation in Elliptic Curve Cryptosystems over GF(2^n)

Tai-Chi Lee

Abstract This paper proposes a public key generation for an ECC (Elliptic Curve Cryptosystem) using FPGA's. To improve the strength of encryption and the speed of processing, the public key and the private key of EC(Elliptic curve) over GF(2^n) are used to form a shared private key (X,Y). And then the X is used with an initial point on HEC(Hyper-Elliptic Curve) over GF(2^n) to generate session keys, which are used with 3BC (Block Byte Bit Cipher) algorithm for the data encryption. We are investigating a novel approach of software/ hardware co-design implemented in Verilog Hardware Description Language (VHDL), which produces hardware algorithm components to place onto the FPGAs, thereby creating adaptive software overlays differentiated by use of a Universal Unique Identifier (UUID) as a functional operand to a custom arithmetic Logic Unit (ALU).

Index Terms: EEC, HEC, FPGA, VHDL, UUID, ASIC, 3BC algorithm.

I. INTRODUCTION

Instead of RSA algorithm, ECC has been widely used for public-key cryptosystem for encryption/decryption. As a matter of fact, the key length for secure RSA has increased over the years. This would demand a heavy computing power for applications, especially for electronic commerce site that process a large number of transaction. Recently, a different approach of generating public key based on elliptic curve cryptography (ECC) [2, 4, 10] has begun to challenge the weakness of RSA [11]. Its security relies on the problem of computing logarithms on the points of an elliptic curve. However, the use of hyperelliptic curve has recently attracted some researchers' interest because it gives the same security level with a smaller key length compared to cryptosystems using elliptic curves The main attraction of combining EC with HEC is that it appears to offer equal security for a far smaller key size, thereby saving the processing overhead. To improve the strength of encryption and the speed of processing, the public key and the private key of ECC are used with initial point on HEC to generate session keys for the data encryption. Fundamentally, HECC (Hyper-Elliptic Curve Cryptosystem) technique is more mathematics involved. We only give a brief review of the basic concept in the next section and explain elliptic curve ciphers later.

II. THE MATHEMATICAL OVERVIEW

The elliptic curve cryptosystem makes use of elliptic curve in which the variables and coefficients are all restricted to elements of a finite field. Two families of elliptic curves are used in cryptographic applications. They are prime curves defined over Zp and binary curves constructed over GF(2^n). It has been found that prime curves are best for software applications, because the extended bit-fiddling operations needed by binary curves are not required. Binary curves are best for hardware applications, where it takes logic gates to create a powerful, fast cryptosystem [1]. In this paper, we will examine only the family of elliptic curves defined over GF(2^n). .

2.1 EC (Elliptic Curves) over $E(F_{2^m})$

An elliptic curve with underlying field F_{2^m} is formed by choosing the element a and b within F_{2^m} (the only condition is that b is not 0. As a result of the field F_{2^m} having a characteristic 2, the elliptic curve equation is slightly adjusted for binary representation:

$$y^2 + xy = x^3 + ax^2 + b.$$

The elliptic curve includes all points (x, y) which satisfy the elliptic curve equation over F_{2^m} (where x and y are elements of F_{2^m}). An elliptic curve group over F_{2^m} consists of the points on the corresponding elliptic curve, together with a point at infinity, O. The number of points in $E(F_{2^m})$ is denoted by $\#E(F_{2^m})$. It follows from the Hasse theorem that

$$q+1-2\sqrt{q} \leq \#E(F_{2^m}) \leq q+1+2\sqrt{q},$$

where q = 2^m. Furthermore, $\#E(F_{2^m})$ is even.

The set of points $E(F_{2^m})$ is a group with respect to the following addition rules:

(1) 0 + 0 = 0.
(2) (x, y) + 0 = (x, y) for all (x, y) $\in E(F_{2^m})$.
(3) (x, y) + (x, x + y) = 0 for all (x, y) $\in E(F_{2^m})$.
(i.e., the inverse of the point (x, y) is the point (x, x + y)).
(4) Rule for adding two distinct points that are not inverses of each other :

Let $(x_1, y_1) \in E(F_{2^m})$ and $(x_2, y_2) \in E(F_{2^m})$ be two points such that $x_1 \neq x_2$.

Then $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$, where

$x_3 = L + L + x_1 + x_2 + a$, $y_3 = L(x_1 + x_3) + x_3 + y_1$ and $L =. (y_1+ y_2)/(x_1 + x_2)$

(5) Rule for doubling a point :

Let $(x_1, y_1) \in E(F_{2^m})$ be a point with $x_1 \neq 0$. Then $2(x_1, y_1) = (x_3, y_3)$, where

$x_3 = L^2 + L + a$, $y_3 = x_1^2 + (L + 1) x_3$ and $L = x_1 + y_1/ x_2$.

Manuscript submitted December 15, 2011; received December 17, 2011; revised January 18, 2012.

Tai-Chi Lee is with the Department of Computer Science and Information Systems, Saginaw Valley State University, University Center, MI 48710 USA (e-mail: lee@svsu.edu).

The group $E(F_{2^m})$ is abelian, which means that P + Q = Q + P for all points P and Q in $E(F_{2^m})$

2.3 ECC (Elliptic Curve Cryptosystem) over $E(F_{2^m})$

The concept of ECC, which was proposed by N. Kobiltz [4] and V. Miller [10] in 1985 is that when any two points are selected and added, the point of the sum is generated and is used for cryptosystem. The elliptic curve (EC) over F_{2^m} is a set of points (x, y) to satisfy the equation $y^2 + xy = x^3 + ax^2 + b$.

The procedure to generate a public key in ECC is outlined as follows:

(1) [common] Select any irreducible polynomial f(x)
(2) [common] Select any vector value a, b for EC such that $y^2 + xy = x^3 + ax^2 + b$
(3) [common] Select randomly an initial point P among points on EC
(4) [sender] Receives p, E, P, k_rP from common
(5) [sender] Generates a random integer k_s as a private key
(6) [sender] Computes a public key k_sP by multiplying P by k_s and registers it in the common directory.
(7) [sender] Computes a shared secret key $k_s(k_rP)$ by multiplying k_s
(8) [receiver] Generates a random integer as private key k_r
(9) [receiver] Computes a public key k_rP by multiplying P by k_r and registers it in the common directory.
(10) [receiver] Computes a shared secret key $k_s(k_rP)$ by multiplying k_s

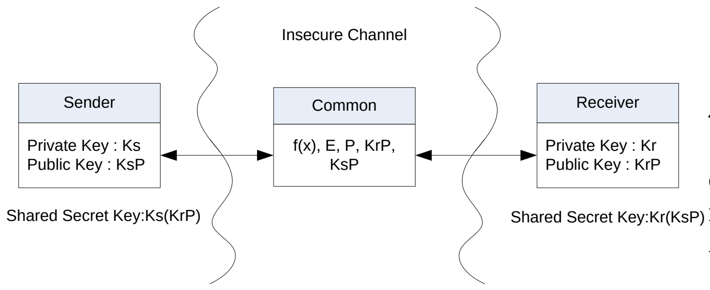


Figure 1. Concept of ECC public key

2.4 HEC (Hyper-Elliptic Curve)

In the study of cryptosystem, hyper-elliptic curve has recently attracted some researchers' interests because it gives The main attraction of combining EC with HEC is that it appears to offer equal security for a far smaller key size, thereby saving the processing overhead because it gives the same security level with a smaller key length as compared to cryptosystems using elliptic curves. From the fact it is expected to be possible to use hyper-elliptic curves to factor integers, since elliptic curve method exploits the property of the Abelian groups in the same way as the cryptosystems.

A hyper-elliptic curve H of genus g($g \geq 1$) over a field F is a nonsingular curve that is given by an equation of the following form:

$H : v^2 + h(u)v = f(u)$ (in F[u, v])

Where h(u)$\in$F[u] is a polynomial of degree$\leq$ g, and f(u)$\in$F[u] is a monic polynomial of degree 2g+1.

2.4.1 Divisors

Divisors of a hyper-elliptic curve are pairs denoted div(a(u), b(u)), where a(u) and b(u) are polynomials in $GF(2^n)$[u] that satisfy the congruence

$$b(u)^2 + h(u)b(u) \equiv f(u) \text{ (mod a(u))}.$$

They can also be defined as the formal sum of a finite number of points on the hyper-elliptic curve. Since these polynomials could have arbitrarily large degree and still satisfy the equation, the notion of a reduced divisor is needed. In a reduced divisor, the degree of a(u) is no greater than g, and the degree of b(u) is less than the degree of a(u).

2.4.2 Reduced Divisors

Let H be a hyper-elliptic curve of genus g over a field F. A reduced divisor (defined over F) of H is defined as a form div(a, b), where a, b$\in$F[u] are polynomial such that

(1) a is monic, and deg b < deg a $\leq$ g,
(2) a divides $(b^2 - bh - f)$.

In particular div(1,0) is called zero divisor.

[Algorithm 1] Reduction of a divisor to a Reduced Divisor.

```
Input : A semi-reduced divisor, D=div(a, b)
Output : The equivalent reduced divisor, $D' = div(a', b') \sim D$
 1. Set $a' = (f - bh - b^2)/a$ and $b' = (-h - b)(\mathrm{mod}\, a')$
 2. If $\deg_u a' > g$ then set $a = a'$, $b = b'$ and go to step 1.
 3. Let c be the leading coefficient of $a'$. Set $a' = c^{-1}a'$.
 4. Output $D' = div(a', b')$
```

2.4.2 Adding Divisors

If $D_1 = div(a_1, b_1)$ and $D_2 = div(a_2, b_2)$ are two reduced divisors defined over F, then Algorithm 2 finds a semi-reduced divisor or reduced divisor $D_3 = div(a, b)$. To find the unique divisor, $D_3 = div(a, b)$, Algorithm 1 should be used just after the addition of two divisors.

[Algorithm 2] Addition defined over the group of divisors

```
Input: Two reduced divisors, $D_1 = div(a_1, b_1)$ and $D_2 = div(a_2, b_2)$
Output : A reduced divisor or semi-reduction divisor, $D_3 = div(a, b)$
 1. Compute $d_1, e_1$ and $e_2$ which satisfy
        $d_1 = GCD(a_1, a_2)$ and $d_1 = e_1a_1 + e_2a_2$
 2. If $d_1 = 1$, then $a := a_1a_2$, $b := (e_1a_1b_2 + e_2a_2b_1)\,\mathrm{mod}\, a$
    otherwise do the following:
   (1) Compute d, $c_1$ and $s_3$ which satisfy
       $d = GCD(d_1, b_1 + b_2 + h)$ and $d = c_1d_1 + s_3(b_1 + b_2 + h)$.
   (2) Let $s_1 := c_1e_1$ and $s_2 := c_1e_2$, so that
       $d = s_1a_1 + s_2a_2 + s_3(b_1 + b_2 + h)$.
   (3) Let $a := a_1a_2 / d^2$,
    $b := (s_1 a_1 b_2 + s_2 a_2 b_1 + s_3(b_1 b_2 + f))/d \bmod a$
3. output $D_3 = div(a,b)$
```

III. ENCRYPTION AND DECRYPTION WITH MESSAGE EMBEDDED IN A POINT

After having generated a public key, the encryption/decryption can be implemented using different approaches. The simplest one is to embed the message P_m to be sent as a point (x, y) in Ep(a,b) . Since not all (x, y) are in Ep(a,b) we have to select a point P'_m in Ep(a,b) that is sufficient close to the point in (x, y) and work with P'_m as it were P_m and recover it by removing the offsets in x or y. To encrypt and send a message P_m to user B, A choose a random positive integer k and generate the ciphertext C_m consisting of the pair of points

$$C_m = \{ kG, P_m + kP_B \},$$

where G is base point and $P_B = K_B G$ is the public key of user B. Note that user A has masked the message P_m by adding kP_B to it. No one but A knows the value of k, so nobody can remove the mask kP_B. However, to decrypt the ciphertext, B can multiply the first point in the pair by B's secret key K_B and subtract the result from the second point, which gives

$$P_m + kP_B - K_B(kG) = P_m + k(K_BG) - K_B(kG) = P_m, \text{ since } P_B = K_BG.$$

3.1 Encryption and Decryption with 3BC Algorithm

With 3BC algorithm, the procedure of data encryption is divided into three parts, inputting plaintext into data block, byte-exchange between blocks, and bit-wise XOR operations between data and session key.

3.1.2 Session Key Generation

As we know that the value which is obtained by multiplying one's private key by the other's public key is the same as what is computed by multiplying one's public key to the other's private key. The feature of EC is known to be almost impossible to estimate a private and a public key. The proposed key generation combines EC and HEC with 3BC algorithm to generate session keys and cipher text. The encryption and decryption processes are shown in Figure 4. First, an x of shared secret key (x, y) from ECC is inputted as a private key x of HEC, and then xD (where, xD means x times D) is computed, which D is an initial point of HECC. The result of xD generates a session key for 3BC [5]. With this advantage and the homogeneity of the result of operations, the proposed 3BC algorithm uses a 64-bit session key to perform the encryption and decryption. Given the sender's private key Ks and the receiver's public key Pr, we multiply Pr by Ks to obtain a point KsPr = (X, Y) on EC, where $X = X_1 X_{2,\ldots} X_m$ and $Y = Y_1 Y_{2,\ldots} Yn$. Then we form a key N by concatenating X and Y (i.e. $N = X_1 X_{2,} X_m Y_1 Y_{2,\ldots} Yn$), and generate the session keys as follows:

i) If the length (number of digits) of X or Y exceed four, then the extra digits on the left are truncated. And if the length of X or Y less than four, then they are padded with 0's on the right. This creates a number $N' = X_1' X_2' X_3' X_4' Y_1' Y_2' Y_3' Y_4'$. Then a new number N'' is generated by taking the modulus of each digit in N' with 8.

ii) The first session key sk1 is computed by taking bit-wise OR operation on N" with the reverse string of N".

iii) The second session key sk2 is generated by taking a circular right shift of sk1 by one bit. And repeat this operation to generate all the subsequent session keys needed until the encryption is completed. For more details on the use of public key and session key for encryption and decryption process, see [5].

3.2.2 Block Data Input

The block size is defined as 64 bytes. A block consists of 56 bytes for input data, 4 byte for the data block number, and 4 byte for the byte-exchange block number (see Figure 2). During the encryption, input data stream are blocked by 56 bytes. If the entire input data is less than 56 bytes, the remaining data area in the block is padded with each byte by a random character. Also, in the case where the total number of data blocks filled is odd, then additional block(s) will be added to make it even, and each of those will be filled with each byte by a random character as well. Also, a data block number in sequence is assigned and followed by a byte-exchange block number, which is either 1 or 2.

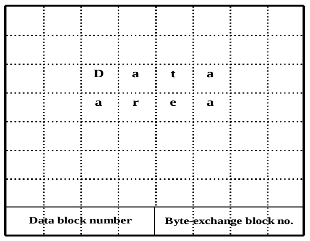


Figure 2. Structure of block

3.2.3 Byte-Exchanges between Blocks

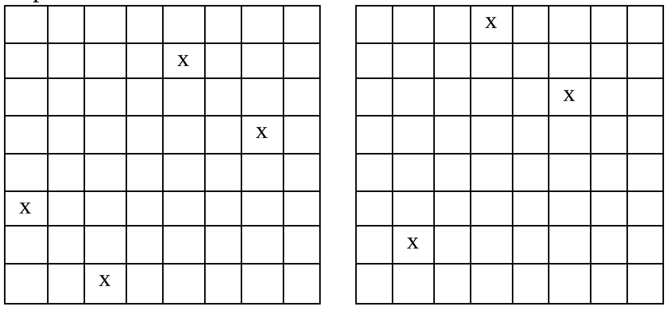
After filling the data into the blocks, we begin the encryption by staring with the first data block and select a block, which has the same byte-exchange block number for the byte exchange. In order to determine which byte in a block should be exchanged, we compute its row-column position as follows:

For the two blocks whose block exchange number, n = 1, we compute the following:

byte-exchange row=$(N_i{*}n) \bmod 8$ (i=1,2 …,8)
byte-exchange col=$((N_i{*}n)+3) \bmod 8$ (i=1,2 …,8),

where N_i is a digit in N". These generate 8 byte-exchange postions. Then for n = 1, we only select the non-repeating byte position (row, col) for the byte-exchange between two blocks whose block exchange numbers are equal to 1.

Similarly, we are repeating the procedure for n = 2. The following examples illustrates the process of byte exchange operation.



(a) n = 1 ***(b) n = 2***

Figure 3. Exchange bytes at (row, col) for a selected pair of blocks

Example: Given the values of a sender's public key 21135 and a receiver's private key 790, we compute the position of row and col for byte-exchange as follows:

For n = 1, It follows from 3.2.1 that N″= 11357900 (after truncation, padding and concatenation), and
row = ((1,1,3,5,7,9,0,0)*1) mod 8 = (1,1,3,5,7,1,0,0) and
col = (((1,1,3,5,7,9,0,0)*1+3) mod 8) = (4,4,6,0,2,4,3,3)
This results 8 byte-exchange positions, (1,4), (1,4), (3,6), (5,0), (7,2), (1,4), (0,3) and (0,3). However, counting only once for repeating pairs, the four bytes at (1,4) (3,6), (5,0), and (7,2) will be selected for byte-exchange between two blocks (see Figure 3 (a)).
For n = 2, we have
row = ((1,1,3,5,7,1,0,0)*2) mod 8 = (2,2,6,2,6,2,0,0) and
col = (((1,1,3,5,7,1,0,0)*2+3) mod 8 = (5,5,1,5,1,5,3,3),
which results 8 byte-exchange positions, (2,5), (2,5), (6,1), (2,5), (6,1), (2,5), (0,3) and (0,3). Similarly, only three byte positions at (2,5), (6,1), and (0,3) are used for byte-exchanges between two blocks as shown in Figure 3 (b).

3.2.4 Bit-wise XOR between Data and Session Keys

After the byte-exchange is done, the encryption proceeds with a bit-wise XOR operation on the first 8 byte data with the session sk1 and repeats the operation on every 8 bytes of the remaining data with the subsequent session keys until the data block is finished (see Figure 4).

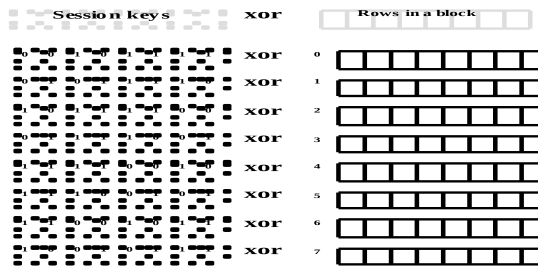


Figure 4. The bit-wise XOR on rows with session keys

Note that the process of byte-exchange hides the meaning of 56 byte data and the exchange of the data block number hides the order of data block, which needs to be assembled later on. In addition, the bit-wise XOR operation transforms a character into a meaningless one, which adds another level of complexity to deter the network hackers. Figure 4 shows an encryption procedure using session keys as described in 3.2.1 deriving from a private key and a public key [11, 12].

IV. COMPLEXITY OF ALGORITHM

The addition operation in ECC is the counterpart of modular multiplication in RSA, and multiple additions are the counterpart of modular exponentiation. Especially, with the use of hyperelliptic curve over $E(F_{2^m})$, to compute a public key and the order of base point, it requires much more intensive computations [5]. Therefore, from computational aspect, we like to offer few suggestions, which could speed up the process.

1) Multiple additions:
Given a point P ε $E(F_{2^m})$, to compute a public key kP for an integer k requires (k – 1) point-additions and each point-addition needs a number of integer additions, subtractions , multiplications and divisions based on modular arithmetic as described in section 2. To reduce the computation time, we can compute the point Q = (k/2)P. Then kP can be readily obtained from 2Q or 2Q+P respectively depending on k is even or odd, which will cut the time in half. For a large k, this is a great time saving.

2) Modular arithmetic:
As seen from the section 2, to obtain the point (x, y) of the sum of two point P and Q on the elliptic curve, we repeatedly involve using modular arithmetic with respect to the prime number p. One can shorten the computation if more efficient techniques dealing with modular arithmetic is used by either an improved algorithm or custom computing machine, which is the purpose of this research.

V. IMPLEMENTATIONS

To generate a public key, the most time consuming process is to find an initial point P on the given elliptic curve and to compute kP for an integer k < p for a large prime number p. The approach we investigate in this paper is to create a 64bit ALU with its own custom instructions added to an Altera EP1C12 NIOS II embedded processor. Custom instructions are designed to be small, re-arrangeable portions of a C implementation of key generation. This will allow sections of the algorithm to be in C and other sections to be expressed as custom instructions. These sections can be easily reordered and re-factored by recompiling the algorithm and uploading the overlay to the FPGA via TCP/IP in order to handle the distribution of the algorithms over the network. See Figure 5.

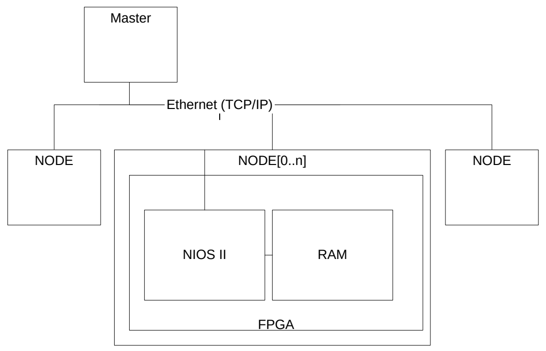


Figure 5 – System Diagram

5. 1 Hardware Design

The design of this approach consists of four components: A PC Master Controller, TCP/IP interconnect,

FGPA logic units that each contains a NIOS II processor and custom ALU, and the creation and selection of the custom instructions and overlays.

5.2 PC Master Controller

A PC Master Controller will provide benefits over existing designs. It is capable of systematically assigning algorithms to logic units based on the specific set of custom instructions included in the ALU. Our design will implement the most efficient way to delegate operations and also take advantage of the parallelism that can be obtained by using a FPGA [6, 7, 8].

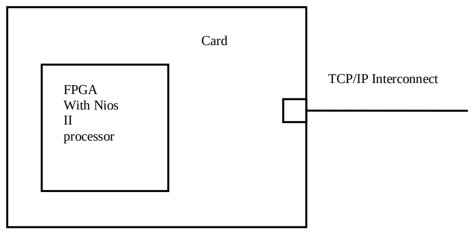


Figure 6. The basic design for card with FPGA

5.3 TCP/IP Interconnect

The PC Master Controller will communicate through a TCP/IP interface with one or more FGPAs in a cluster.

Each FPGA will execute the algorithm, using the custom instructions. See Figure 7.

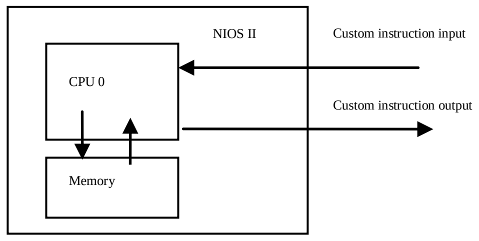


Figure 7. The layout of the FPGA

5.4 Custom Instructions

The Altera EP1C12 NIOS II embedded processor is a 32bit system. By adding a customized 64bit ALU and associated 64bit registers [11, 12], we can have custom instructions to handle algorithms specific to public key via EC, which include: XOR, Addition, Multiplication, Division, Right/Left Shift, and others. These instructions are given 32bit UUIDs as their opcode, allowing unique naming even when the full set is not within a single ALU. We are experimenting with various decompositions of Expansion and Permutation in order to create sub-algorithms the custom instructions reproduced in hardware. See Figure 8.

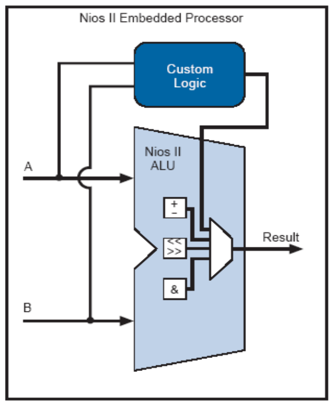


Figure 8. Nios II processor operation from Altera's Nios II Custom Instruction User Guide

VI. ANTICIPATED RESULTS

Based upon the above approach we are investigating the different decomposition of the subalgorithms used.. By using our approach on the encryption/decryption algorithm we expect to be able to process on average one key per clock cycle. This appears reasonable as the custom instructions allow our design to use several FPGAs to process multiple key ranges simultaneously. We are interested in locating the balance between the high implementation time with the low run time of the pure hardware approach, and the low implementation time with high run time of the pure software approach.

For instance, starting with encryption or decryption algorithm in C that has a nonexistent design time, its average run time is a constant. When the algorithm is translated into well optimized hardware, the design time is very high and the run time is very low. With our approach the design time and run times are between the pure hardware and pure software methods. When the number of data sets to run is in the bolded range on Figure 9, our method should be preferred.

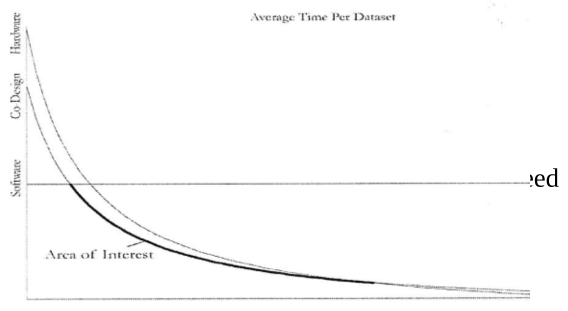


Figure 9 Area of interest

VII. CONCLUSION

Our approach has the benefit of some of the speed of ASIC, while maintaining some of the flexibility of C**.** The added use of storing and transferring the algorithms as an overlay allows the organizational aspects of the algorithm to be re-factored and delivered without the need of rebuilding the ASIC image and reconfiguring the FPGA [3, 9] Using a TCP/IP interconnect network to send both the overlay and the problem set allows for an efficient and easily scalable infrastructure.

The proposed 3BC, which uses byte-exchange and the bit operation increases data encryption speed. Even though cipher text is intercepted during transmission over the network. Because during the encryption process, the 3BC algorithm performs byte exchange between blocks, and then the plaintext is encoded through bit-wise XOR operation, it rarely has a possibility for cipher text to be decoded and has no problem to preserve a private key.

REFERENCES

[1] M.J. Bastiaans, *FPGA's as Cryptanalytic Tools, http://www.sps.ele.tue.nl/ members/m.j.bastiaans/spc/rouvroy.pdf.*

[2] A. Fernaades, *Elliptic Curve Cryptography*, Dr. Dobb's Journal, December, 1999.

[3] P. Glesner, and M. Zipf, Renovell (Eds.): *Programmable Logic and Applications. Reconfigurable Computing Is Going Mainstream*, Proceedings of 12th International Conference, FPL 2002, Montpellier, France, September 2-4, 2002.

[4] N. Koblitz, *Elliptic Curve Cryptosystems*. Math. Comp. 48 203-209, 1987.

[5] Tai-Chi Lee, ByungKwan Lee, *A HESSL(Highly Enhanced Security Socket Layer) Protocol*, The Proceedings of the Seventh IEEE International Conference *on* E-Commerce Technology, pp 456-460 July 19-22, 2005, Munich, Germany.

[6] Tai-Chi Lee, Mark White, *Using Software Emulation in FPGAs To Improve Co-Design Development Time,* The Proceedings of the Fourth International Conference of Applied Mathematics and Computing, Bulgaria, pp. 359-360, August 12-18, 2007.

[7] Tai-Chi Lee, Patrick Robinson, *A FPGA-Based Designed for an Image Compressor***,** International Journal of Pure and Applied Math, Academic Publications, Volume 33 No.1, pp. 63-67, 2006.

[8] Tai-Chi Lee, Richard Zeien, Adam Roach, and Patrick Robinson*, DES Decoding Using FPGA and Custom Instructions*, The Proceedings of The Third International Conference on Information Technology: New Generation, Las Vegas, Nevada, pp. 575-577, 2006.

[9] Tai-Chi Lee, Patrick Robinson, and Erik Henne, *Framework for executing VHDL code on FPGA*, The Proceedings of the International MultiConference in Computer Science & Computer, Las Vegas, NV, pp. 1296-1299, 2004.

[10] V.S. Miller, *Use of Elliptic Curve in Cryptography*. Advances in Cryptology-Proceedings of Crypto '85, Lecture Notes in Computer Science 218**,** pp. 417-426, Springer-Verlag, 1986.

[11] M. Robshaw, *Block Ciphers,* RSA LaboratoriesTechnical Report TR, 601, August 1995, http://www.rsasecurity.com/rsalabs/dindex.html.

[12] B. Schneier, *Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish),* Proceedings, Workshop on Fast 78 Software Encryption, New York: Springer-Verlag, 1993.

[13] B. Schneier, *The Blowfish Encryption Algorithm,* Dr. Dobb's Journal**,** April 1994.