ISSN 2278-3091
Volume 5, No.4, July – August 2016
International Journal of Advanced Trends in Computer Science and Engineering
Available Online at http://www.warse.org/IJATCSE/static/pdf/file/ijatcse08542016.pdf

Implementation of 163-bit Elliptic Curve Diffie Hellman (ECDH) Key Exchange Protocol Using BigDigits Arithmetic

Irfan Ahmad[1], Muhammad Waseem[2]
[1]Satellite Research and Development Center (SRDC), Lahore 54590, Pakistan, Email: irfahmad@gmail.com
[2]Satellite Research and Development Center (SRDC), Lahore 54590, Pakistan, Email: vasim98@gmail.com

ABSTRACT

For recommended number of bits Elliptic Curve Diffie Hellman (ECDH) key exchange protocol can be implemented in a variety of ways. But due to lack of built-in data types in structured programming language like C to manipulate larger numbers, we require the aid of a third party library that can accommodate and provide mathematical manipulations of the larger custom built data types. To implement ECDH key exchange protocol for 163 bits at application level, we have implemented an abstract API on the top of BigDigits. This API enriches existing BigDigits by providing all necessary mathematical manipulations and variables to accommodate larger numbers for an efficient design of ECDH key exchange protocol. Using this library, standard algorithms are implemented efficiently as hierarchical API's, by providing abstraction for the application programmers to implement ECDH key exchange protocol without knowing the details of implementation of the BigDigits source code library. The final implementation is reusable as compact dynamic linked library and is language independent, therefore useable at application level wherever necessary.

Key words: Cryptography, Elliptic Curve, API, Diffie Hellman, BigDigits, Key exchange.

1. INTRODUCTION

Key exchange protocols always play an important role in encryption algorithms. Two approaches are normally adopted to exchange keys, namely Symmetric and Asymmetric. Both of these differ in a sense that the former uses the same key at both receiver and transmission end. However, later uses two keys namely private and public to carry out the whole session of key exchange.

In Public-key encryption also known as 'Asymmetric Encryption' normally two keys, a public key and a private key are incorporated. Everyone can have an access to the public keys but the corresponding private keys are not disclosed to anyone. The public key is used to encrypt the data meant for one user and private key is used to decrypt the data. This approach allows the publication of a public key and using this public key, only the intended communicant will be able to read encrypted data [9].

Elliptic Curve Cryptography lies under the category of asymmetric encryption algorithms. In elliptic curve cryptography, points on elliptic curve are used to derive a public key i.e. a generator point in an elliptic curve group is agreed upon by the communicating parties. By multiplying this generator point by a randomly generated number, corresponding private key is generated. In case, the generator point and public keys are compromised, it is very hard problem for the intruder to get the private key by backtracking. Once computed, this public key can be utilized to achieve various cryptographic objectives e.g. the key exchange using Diffie-Hellman method.

In mid 1970s, Diffie and Hellman put forward a practical public key algorithm for exchange of secret keys. Using this algorithm, two participants can securely exchange a secret key also known as session key. Using a symmetric key encryption algorithm, this secret key can be used in encryption of messages. Basically, this algorithm is restricted to key exchange only. The effectiveness of Diffie Hellman algorithm lies in the difficulty of calculating logarithms in discrete domain. In this scheme involvement of a third party is not required for key exchange [1]. Only two parties i.e. a receiver and a sender is involved.

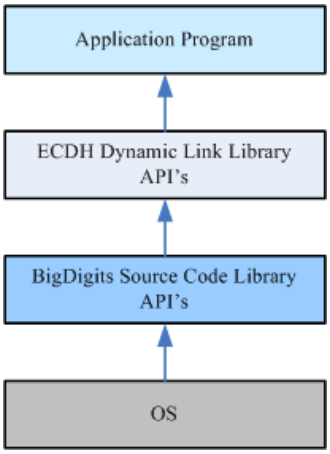

Figure 1: API Structure for Implementation of ECDH

The ECDH key exchange protocol is an extension of Diffie-Hellman protocol using elliptic curve cryptography. In Elliptic Curve Diffie-Hellman, two approaches are adopted together to carry out the key exchange more securely between the intended parties. To implement the ECDH key exchage protocol at application level, we have extended BigDigits [6] with an abstract API's of ECDH key exchange protocol. The variables in the BigDigits library [6] serve as containers of

larger numbers and rich mathematical functions which are used to carry out the calculations required in the underlying ECDH algorithms. Implementation is performed in a structured programming language as a dynamically linked library (DLL) and which has separate interface for all the illustrated algorithms of the protocol. These algorithms can be hooked from application programs in any programming language that can include this DLL, as illustrated in Figure 1.

This paper is organized as follows: Section 2 gives mathematical concepts of the Elliptic Curve Diffie Hellman key exchange protocol, Section 3 covers some overview of BigDigits Multiple precision arithmetic source code library, Section 4 is about the implementation details of the ECDH key exchange protocol using BigDigits library, Section 5 contains the utilization of the ECDH APIs. Conclusion and future work are described in Section 6.

2. ELLIPTIC CURVE DIFFIE HELLMAN KEY EXCHANGE PROTOCOL

Elliptic Curve Diffie Hellman (ECDH) is a key exchange protocol that allows two communicants to exchange a shared secret over an insecure channel [7] [8], while each having an elliptic curve public-private key pair. This shared key or public key is used to derive another key namely session key that is used for encryption. To exchange keys between two parties by using ECDH, initially following domain parameters [11] are agreed upon:

$$(m, f(x), a, b, tt, n, h) \qquad (1)$$

where a and b are the elliptic curve parameters, *tt* is a base point with coordinates (G_x, G_y) at the elliptic curve with an order *n* and $f(x)$ is an irreducible polynomial with degree *m* [11]. In order to carry out the secure communication using ECDH key exchange protocol between the intended parties, a scenario has been discussed below based on the illustrations in [9]. For this, each party must have private key $K_S User$ and a public key *P* where

$$P = K_S User \times G \qquad (2)$$

Let's first party's key pair be $(K_S User_A, P_A)$ and other party's key pair be $(K_S User_B , P_B)$. Both parties mutually exchange public keys. Using these public keys, each user generates his session key as follows:

$$K_A = K_S User_A \times P_B \qquad (3)$$

$$K_B = K_S User_B \times P_A \qquad (4)$$

Since Eq. 2 for User A and User B implies,

$$P_A = K_S User_A \times G \qquad (5)$$

$$P_B = K_S User_B \times G \qquad (6)$$

Therefore Eq. 3 and Eq. 4, by substituting P_A and P_B from Eq. 5 and Eq. 6, become:

$$(K_x, K_y) = K_A = K_S User_A \times K_s User_B \times G \qquad (7)$$

$$(K_x, K_y) = K_B = K_S User_B \times K_S User_A \times G \qquad (8)$$

The shared key is K_x between the two users. The shared key as depicted by Eq. 7 and Eq. 8 above are equivalent. Public key is the only information about the private key that User A exposes. So, no user other than User A can determine User B's private key, unless that user can solve the Elliptic Curve Discrete Logarithm problem [13]. User B's private key is similarly secure. No user other than User A or User B can compute the shared secret, unless it solves the elliptic curve Diffie-Hellman problem.

3. BIGDIGITS MULTIPLE PRECISION ARITHMETIC SOURCE CODE LIBRARY

Cryptography calculations require calculations of large natural numbers. To carry out these calculations, a library BigDigits written in ANSI-C has been introduced by D.I. Management Services [6]. BigDigits is a library of multiple-precision arithmetic routines and its implementation has been built using the algorithms in [2] and [3] as the primary references. The classical multiple-precision arithmetic algorithms like add, subtract, multiply and divide are from [2]. This library also includes other functions such as modular multiplication, exponentiation and inversion; number theory function such as greatest common divisor and the Rabin-Miler Probabilistic Primality Test procedure from [4] and [5]. Rabin- Miler Probabilistic Primality Test procedure is used to show that a large integer is probably a prime [6]. Except the initial creation and final release of resources, this library also has a capability to handle memory allocation automatically [6].

4. API IMPLEMENTATION

For security purpose, elliptic curve domain parameters over F m must have following set of degrees of irreducible polynomials [11]:

$$m = \{113, 131, 163, 193, 233, 239, 283, 409, 571\} \qquad (9)$$

But for all these numbers there are no data types except custom data types in any programming language to contain such large numbers and perform mathematical operation on such variables. Several modern programming languages have built-in support for bignums [12], others have libraries for floating point mathematics and arbitrary-precision integers. These implementations use variable length arrays of digits instead of storing values as fixed numbers of binary bits compatible with the register size of the microprocessor. We can use arbitrary precision in applications where we require precise results with very large numbers or where speed of arithmetic is not a limiting factor. One such library is BigDigits library for which attributes are listed in Table 1. It is a collection of C library routines with natural number types for calculation of large numbers.

Table 1: BigDigits Library

Package / Library Name	Number Type	Language	License
BigDigits	Naturals	C	Freeware

The BigDigits library [6] files, we have used to perform our calculations are,

- `bigdigits.c`
- `bigdigits.h`
- `bigdtypes.h`

The functions that need to be implemented for Elliptic Curve Diffie Hellman are:

- Field Adder Function
- Field Multiplier Function
- Field Inverter Function
- EC Point Adder Function
- EC Point Doubler Function
- EC Point Multiplier Function

We have selected $m = 163$ as Elliptic Curve Domain Parameter to perform our calculations. The irreducible polynomial over F^{163}_2 is given in Eq. 10:

$$f(x) = x^{163} + x^7 + x^6 + x^3 + 1 \qquad (10)$$

Decimal value of the above equation is calculated in a spread-sheet and given in Table 2.

Table 2: Decimal & Hex Values of Irreducible Polynomial

f (x)	1169201309864720000000000000000000000000000000000000.0
f (x)	08 00000000 00000000 00000000 00000000 000000C9 (HEX)

This large value cannot be converted in Hexadecimal value with ordinary calculator; therefore APICalc 2 [10] is used for this conversion.

Other Elliptic Curve Domain Parameters over F^{163}_2 for the curve in Eq. 11, as recommended by "Standards for Efficient Cryptography (SEC)" are given in Table 3. The corresponding equation of Elliptic Curve, irreducible polynomial and related domain parameters are selected from [11]:

$$E : y^2 + xy = x^3 + ax^2 + b \qquad (11)$$

Scalar multiplication is the core operation involved in ECDH key exchange protocol. In order to carry out this operation we have to perform EC point addition and EC point code segments we have used the field F^{163}_2 with irreducible polynomial as given in Eq. 10. We have incorporated algorithms in [14] and [15] to implement the ECDH key exchange protocol using BigDigits library [6].

To implement the functions as depicted in Figure. 2, we will require a minimal set of functions from BigDigits library [6]. The essential functions that we have imported are listed below:

- mpSetEqual() Function
- mpXorBits() Function
- mpSetZero() Function
- mpGetBit() Function
- mpShiftLeft() Function
- mpShiftRight() Function

Table 3: Standard Elliptic Curve Domain Parameters

a	07 B6882CAA EFA84F95 54FF8428 BD88E246 D2782AE2
b	07 13612DCD DCB40AAB 946BDA29 CA91F73A F958AFD9
G	040369 979697AB 43897789 56678956 7F78 7A78 76A65400 435EDB42 EFAFB298 9D51FE FC E3C80988 F41FF883

The implementation requirements are not limited to above listed functions. These library functions are utilized in the ECDH field arithmetic functions such as Field Adder, Field Multiplier and Field Inverter. These field arithmetic functions are then used to achieve the functionality of EC Point Adder and EC Point Doubler functions. Once implemented, EC Point Adder and EC Point Doubler are employed in EC Point Multiplier function. The implementation of four API functions requires the EC Point Multiplier function.

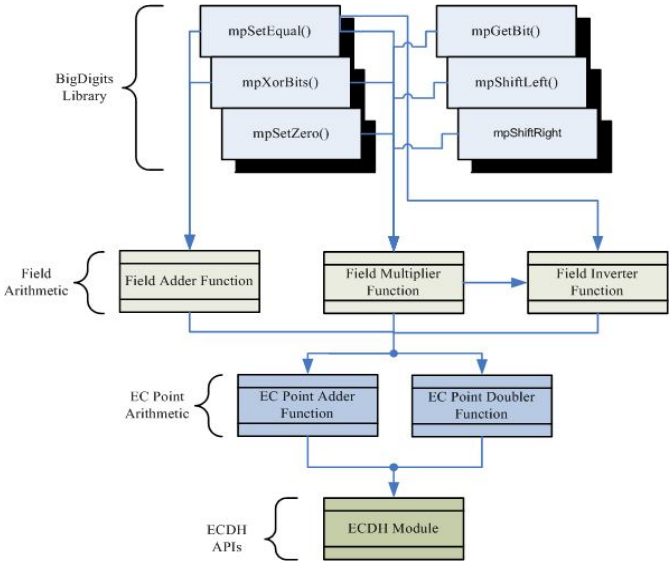

Figure 2: ECDH Module

Functional details and interfaces of each component described in Figure 2 are illustrated in the form of tables. Functions name, arguments and return value of the functions are given in the Table 4.

`DIGIT_T` is a custom type for 32-bit unsigned integer. The `ndigits` is the size of array elements and for 163-bit implementation of ECDH key exchange protocol, we have selected `ndigits=8`. The `ibit` is the bit number to extract.

4.1 Field Adder Function

XOR operation is used to perform the addition of any two elements in the field F_2^{163} . This function requires the `mpSetEqual()` and `mpXorBits()` of the BigDigits library as illustrated in Table 4 to carry out the addition between the two elliptic curve parameters. The functional interfaces of this function are depicted in Table 5 and Table 8 as Callee (the functions that are called by this function) and Caller (functions that call this function).

Table 4: Function name, Argument and Return values

Function Name	Arguments	Return Value	Remarks
`mpSetEqual()`	`DIGIT_T a[], size_t ndigits, const DIGIT_T b[]`	`void`	Sets a=b
`mpXorBits()`	`DIGIT_T a[], const DIGIT_T b[], const DIGIT_T c[], size_t ndigits`	`void`	Compute bitwise a=b
`mpSetZero()`	`volatile DIGIT_T, a[], size_t ndigits`	`volatile DIGIT_T`	Sets a=0
`mpGetBit()`	`DIGIT_T a[], size_t ndigits, size_t ibit`	`int`	1/0/-1
`mpShiftLeft()`	`DIGIT_T a[], const DIGIT_T [b], size_t shift, size_t ndigits`	`DIGIT_T`	Compute bitwise a=b <<Shift
`mpShiftRight()`	`DIGIT_T a[], const DIGIT_T b[], size_t shift, size_t ndigits`	`DIGIT_T`	Compute bitwise a=b >>Shift

4.2 Field Multiplier Function

Multiply function is used to achieve the functionality of modular multiplication. To implement this function, we require `mpSetEqual()`, `mpSetZero()`, `mpGetBit()`, `mpXorBits()`, `mpShiftLeft()` and `mpShiftRight()` functions of the BigDigits library. The callee and caller functions with required arguments and return values are given in Table 7 and Table 8 respectively.

Table 5: Field Adder Callee Functions

Function Name	Arguments	Return Value	Remarks
`mpSetEqual()`	`var1,a,8`	`void`	a∈ F_2^{163}
`mpSetEqual()`	`var2,b,8`	`void`	b∈ F_2^{163}
`mpXorBits()`	`Result,var1,var2,8`	`void`	

4.3 Field Inverter Function

The field inverter function is implemented using `FieldMul()` and `mpSetEqual()` functions. This function is required by the `ECPointAdd()` and `ECPointDoubler()` functions. The interfaces of this function are given in Table 9 and Table 10.

Table 6: Field Adder Caller Functions

Function Name	Arguments	Return Value	Remarks
`ECPointAdd()`	`y2,y1`	`(y2+y1)`	y1,y2∈ F_2^{163}
`ECPointAdd()`	`x2,x1`	`(x2+x1)`	x1,x2∈ F_2^{163}
`ECPointAdd()`	`x3,x1`	`(x3+x1)`	x1,x3∈ F_2^{163}
`ECPointDoubler()`	x1,$(y1.(x1)^{-1})$	λ	x1,y1∈ F_2^{163}

Table 7: Field Multiplier Callee Functions

Function Name	Arguments	Return Value	Remarks
`mpSetEqual()`	`var1,a,8`	`void`	$a \in F_2^{163}$
`mpSetEqual()`	`var2,b,8`	`void`	$b \in F_2^{163}$
`mpSetZero()`	`var3,8`	`var3`	`var3=0`
`mpXorBits()`	`var3,var3,var1,8`	`void`	-
`mpGetBit()`	`var1,8,1`	`1`	`var1(MSB)`
`mpShiftLeft()`	`var1,var1,1,8`	`var1`	-
`mpShiftRight()`	`var2,var2,1,8`	`var2`	-

Table 8: Field Multiplier Caller Functions

Function Name	Arguments	Return Value	Remarks
`ECPointAdd()`	`(y2+y1), (x2+x1)`$^{-1}$	λ	x1,y1, x2,y2 ∈ F_2^{163}
`ECPointAdd()`	λ, λ	λ^2	$\lambda \in F_2^{163}$
`ECPointAdd()`	λ ,(x3+x1)	λ(x3+x1)	x1,x3∈ F_2^{163}
`ECPointDoubler()`	y1,$(x1)^{-1}$	y1.$(x1)^{-1}$	x1,y1∈ F_2^{163}
`ECPointDoubler()`	λ, λ	λ^2	$\lambda \in F_2^{163}$
`ECPointDoubler()`	x1,x1	$x1^2$	x1∈ F_2^{163}
`ECPointDoubler()`	λ, x3	λ(x3)	λ,x3∈ F_2^{163}

4.4 EC Point Adder Function

This function is implemented using `FieldAdder()`, `FieldMul()` and `FieldInv()` functions discussed above. This function performs addition of any two points on

the elliptic curve in Eq. 11 such that the resultant also lies on the same curve. The Callee and Caller associated with this function are given in Table 11 and Table 12 respectively.

Table 9: Field Inverter Callee Functions

Function Name	Arguments	Return Value	Remarks
mpSetEqual()	var1,a,8	void	a∈ F_2^{163}
FieldMul()	var1,var1	var2	-
FieldMul()	var1,var2	var2	-
FieldMul()	var2,var2	var2	-

Table 10: Field Inverter Caller Functions

Function Name	Arguments	Return Value	Remarks
ECPointAdd()	(x2+x1)	$(x2+x1)^{-1}$	x1,x2∈ F_2^{163}
ECPointDoubler()	x1	$x1^{-1}$	x1∈ F_2^{163}

Table 11: EC Point Adder Callee Functions

Function Name	Arguments	Return Value	Remarks
FieldAdder()	y2,y1	(y2+y1)	y1,y2∈ F_2^{163}
FieldAdder()	x2,x1	(x2+x1)	x1,x2∈ F_2^{163}
FieldAdder()	x3,x1	(x3+x1)	x1,x3∈ F_2^{163}
FieldMul()	(y2+y1), $(x2+x1)^{-1}$	λ	x1,y1 x2,y2 ∈ F_2^{163}
FieldMul()	λ, λ	λ^2	$\lambda \in F_2^{163}$
FieldMul()	λ ,(x3+x1)	λ(x3+x1)	x1,x3∈ F_2^{163}
FieldInv()	(x2+x1)	$(x2+x1)^{-1}$	x1,x2∈ F_2^{163}

Table 12: EC Point Adder Caller Functions

Function Name	Arguments	Return Value	Remarks
ECPointMul()	R,S	R	R(x3,y3),S(x4,y4)∈ F_2^{163}

4.5 EC Point Doubler Function

Field arithmetic functions `FieldAdder(), FieldMul()` and FieldInv() functions are used to implement this function. This function doubles a point on the elliptic curve in Eq. 11 and resultant of this doubler function lies again on the same curve. The Callee and Caller functions details regarding this functions are given below in Table 13 and Table 14 respectively.

4.6 EC Point Multiplier Function

Implementation of `ECPointMul()` function requires `ECPointAdd()` and `ECPointDoubler()` functions. This function is used to perform multiplication of any two points on the elliptic curve in Eq. 11 and resulting point lies on the same curve. The functional interfaces of this function are depicted in Table 15 and Table 16 respectively.

Table 13: EC Point Doubler Callee Functions

Function Name	Arguments	Return Value	Remarks
FieldAdder()	x1,(y1.$(x1)^{-1}$)	λ	x1,y1∈ F_2^{163}
FieldMul()	y1,$(x1)^{-1}$	y1.$(x1)^{-1}$	x1,y1∈ F_2^{163}
FieldMul()	λ, λ	λ^2	$\lambda \in F_2^{163}$
FieldMul()	x1,x1	$x1^2$	x1∈ F_2^{163}
FieldMul()	λ, x3	λ(x3)	λ,x3∈ F_2^{163}
FieldInv()	x1	$x1^{-1}$	x1∈ F_2^{163}

Table 14: EC Point Doubler Caller Functions

Function Name	Arguments	Return Value	Remarks
ECPointMul()	S	2S	S(x4,y4)∈ F_2^{163}

Table 15: EC Point Multiplier Callee Functions

Function Name	Arguments	Return Value	Remarks
ECPointAdd()	R,S	R	R(x3,y3),S(x4,y4) ∈ F_2^{163}
ECPointDoubler()	S	2S	S(x4,y4)∈ F_2^{163}

Table 16: EC Point Multiplier Caller Functions

Function Name	Arguments	Return Value	Remarks
CalculateP$_A$()	K_sUser_A , $G(G_x, G_y)$	P_A	$G \in F_2^{163}$
CalculateP$_B$()	K_sUser_B , $G(G_x, G_y)$	P_B	$G \in F_2^{163}$
Ks$_A$()	K_sUser_A , P_B	Ks_A	--
Ks$_B$()	K_sUser_B , P_A	Ks_B	--

4. API UTILIZATION

In order to use ECDH key exchange protocol, a user will require four API functions. First, private numbers are selected by each user and based on these private numbers, public keys are generated by using API functions. These public keys are mutually exchanged and used in generation of session keys. One of the coordinates of the session keys is used for secure communication. The detailed functionality is depicted in Figure 3.

The proposed library of ECDH key exchange protocol provides four API functions for application programmers. To use these API functions, both User A and User B will have to select private number K_sUser_A and K_sUser_B respectively, both of these numbers should be less than the order n of the base point (G_x,G_y). The APIs "`CalculatePA()`" and

"CalculatePB()" are used to generate the public keys for User A and User B respectively, from the randomly selected private numbers. Interfaces of each of these APIs are given Table 17 and Table 18 respectively.

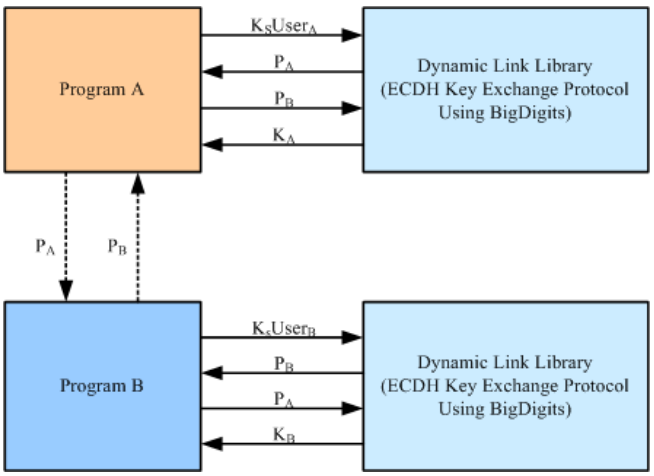

Figure 3: ECDH Key Exchange Protocol Utilization

The public keys are mutually exchanged between User A and User B. Using the API functions "$Ks_A()$" and "$Ks_B()$", session keys are generated based on the public numbers.

Table 17: Calculate P_A Callee Functions

Function Name	Arguments	Return Value	Remarks
ECPointMul()	$Ks User_A, G(G_x, G_y)$	P_A	$G \in F_2^{163}$

The Session keys $Ks(K_x, K_y)$ generated at both ends are same and only one of the coordinates either K_x or K_y is used as session key for secure communication between the two parties. The interfaces of these two API functions are given in Table 19 and Table 20 respectively.

Table 18: Calculate P_B Callee Functions

Function Name	Arguments	Return Value	Remarks
ECPointMul()	$Ks User_B, G(G_x, G_y)$	P_B	$G \in F_2^{163}$

Table 19: Calculate Ks_A Callee Functions

Function Name	Arguments	Return Value	Remarks
ECPointMul()	$Ks User_A, P_B$	Ks_A	-

Table 20: Calculate Ks_B Callee Functions

Function Name	Arguments	Return Value	Remarks
ECPointMul()	$Ks User_B, P_A$	Ks_B	-

5. CONCLUSION

In this paper, we have focused mainly on the implementation details of the 163-bit ECDH key exchange protocol. This key exchange protocol is considered to be secure for larger bit lengths. Therefore, using the approach illustrated in this paper, the ECDH key management scheme can be implemented for any bit length efficiently.

The involved parameters used in computation are constant. However we can use random number generator to generate these parameters. The work done on this protocol library can be considered as a step towards the implementation of ECDH key management scheme in application programs and for real-time implementation in FPGA or ASIC based hardware. A hardware designer can use this library as a golden reference model and can validate the hardware implementation by comparing the results of ECDH key exchange protocol module with the software model.

REFERENCES

1. Tahir Mehmood, **"Security Services in Satellites"** University of Surrey, MS Thesis England: 2006.
2. Donald E. Knuth, ***"The Art of Computer Programming"***, Vol 2 Semi-numerical Algorithms, 3rd edition, Addison-Wesley, 1998.
3. Alfred J. Menezes, Paul C. van Oorschot, Scott A Vanstone, ***"Handbook of Applied Cryptography"***, CRC Press, 1997,
4. **"Digital Signature Standard (DSS)"**, FIPS PUB 186-2, U.S. Department of Commerce/National Institute of Standards and Technology, 2000.
5. ANSI X9.42-2003 "**Public Key Cryptography for the Financial Services Industry: Agreement of Symmetric Keys Using Discrete Logarithm Cryptography**", American National Standards Institute, 2003
6. DI Management, **"BigDigits Multiple-Precision Arithmetic Source code"**, Version 2.2 Released 31st July, 2008.
7. CCSDS "***Next Generation Space Internet (NGSI)End-to-End Security for Space Mission Communications***" CCSDS 733.5-O-1 (Apr 2003)
8. Roohi Banu **"Satellite Encryption Report"** University of Surrey, England.
9. "Cryptography and Network Security Principles and Practices",William Stalling 3rd Ed.
10. "**Beaconaut APICalc2**", Bignum Math, Beaconaut Communications [11] Certicom Research, "SEC 2: Recommended Elliptic Curve Domain
11. **Parameters",** Version 1.0, September 20, 2000
12. **"Cryptographic Services"**, .Net Framework Cryptography Model, MSDN Library.
13. A.M. Odlyzko**,"Discrete logarithms in finite fields and their crypto- graphic significance "**, Advances in Cryptology - Eurocrypt '84, Springer- Verlag (1984), 224-314.
14. Miguel Morales-Sandoval, **"Hardware architecture for elliptic curve cryptography and lossless data compression"**, a thesis presented to Computer Science Department National Institute for Astrophysics, Optics and Electronics, Tonantzintla, Puebla Mexico Dec. 2004
15. N. Mentens et al, **"An FPGA Implementation of an Elliptic Curve Processor over GF (2m)"** Proceedings of the 14th ACM Great Lakes symposium on VLSI 2004, Boston, MA, USA.