CHAPTER 3

LIGHTWEIGHT ELLIPTIC CURVE CRYPTOGRAPHY

This chapter presents the implementation of lightweight Elliptic Curve Cryptography using proposed Dynamic sliding window Non-Adjacent Form (DswNAF) and Twisted Edward curve in order to reduce the computational complexity involved in ECC. This chapter also deals with performance analysis of the proposed methods for ECC based techniques.

3.1 PROPOSED DYNAMIC SLIDING WINDOW NON-ADJACENT FORM FOR POINT MULTIPLICATION (DswNAF)

The flow diagram of ECC based cryptography is shown in Figure 3.1. ECC based cryptography implements schemes such as ECDH using Diffie-Hellman algorithm for key exchange and ECIES using Integrated Encryption Standard (IES) for confidentiality. For determining the Elliptic Curve (EC) parameters over prime field with the basic principles of NAF, this research proposes Dynamic sliding window Non-Adjacent Form (DswNAF). The key generated from the proposed DswNAF is used by ECDH and ECIES. To perform point multiplication in ECC, the proposed DswNAF uses a rule engine to select the rules based on the number of point addition, point doubling and pre-computation to vary the window size. Rule engine helps to reduce the computation complexity in the ECC based schemes.

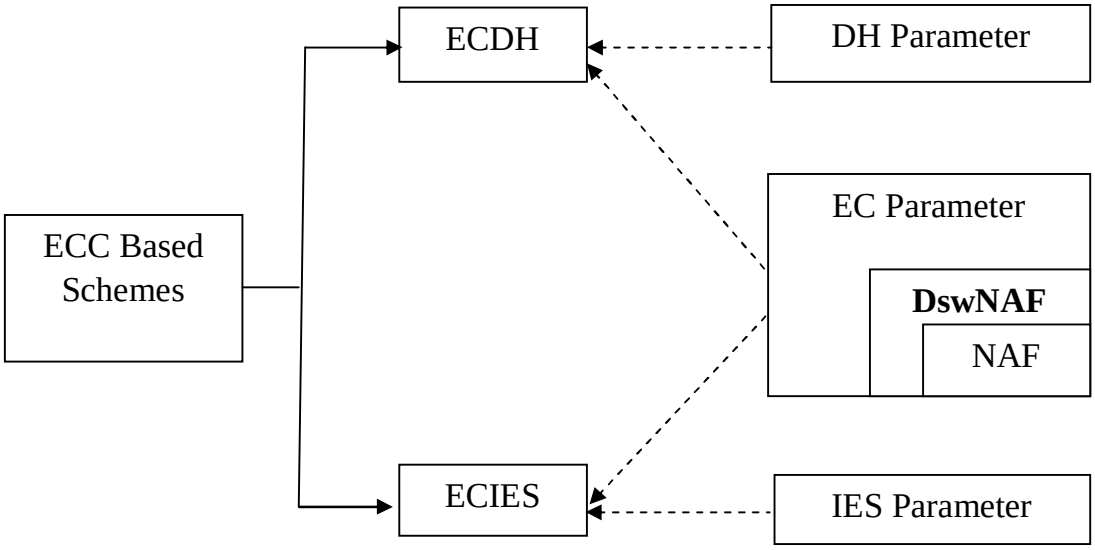

Figure 3.1 Inherent location of the proposed work

3.1.1 Non-Adjacent Form (NAF) for Point Multiplication

Algorithm 3.1 presents the steps involved in NAF. Repeatedly dividing the random positive integer (k) by 2, allowing remainders 0 or ±1 generates the digits of NAF. If k is odd, then the remainder r $\in \{-1,1\}$ such that the quotient (k −r)/2 is even (Hankerson et al 2004). The occurrence of consecutive 1's in k (represented as bit) is replaced by -1 or +1. Number of 1's in k represents number of addition operation and 0's represents number of doubling operation. Therefore, replacing consecutive 1's by ±1 reduces the addition operation. This process leads to reduced computation time (Hankerson et al 2004).

Here k is a positive integer, it is expressed as $k = \sum_{i=0}^{l-1} k_i 2^i$, where $k_i \in \{0, \pm 1\}$, l is the length of k. The length of NAF i$\frac{2^l}{3} < k < \frac{2^{l+1}}{3}$s where *l* is approximately 1/3.

Algorithm 3.1 NAF Algorithm

Input: A positive integer k
Output: NAF
1. $i \leftarrow 0$
2. *while* $k \geq 1$ do
2.1. *if* k is odd then $k_i = 2 - (k \bmod 4)$, $k = k - k_i$;
where i=1,2,...j = 160
2.2. *else* $k_i \leftarrow 0$
2.3. $k = \frac{k}{2}$; $i = i + 1$;
3. *return* $(k_{i-1}, k_{i-2}, ..., k_0)$

The expected running time of the NAF algorithm (shown in Algorithm 3.1) is $\frac{m}{3}A + mD$, where A and D are addition and doubling process respectively. To reduce the memory space and number of addition operations in NAF, the window based NAF or width based NAF is used.

3.1.2 Width Non-Adjacent Form (*w*NAF) for Point Multiplication

NAF algorithm can handle fixed window size, where *w*=2. For high memory space, the running time of Algorithm 3.1 can be reduced by window method, which process *w* digits of *k* at a time. For increased window size, number of 1's is reduced in *w*NAF compared to NAF. This in turn decreases the computation time. The steps involved in *w*NAF process are presented as Algorithm 3.2 (Hankerson et al 2004). The Algorithm 3.2 is executed for various bit sizes as per NIST standard to evaluate *w*NAF algorithm.

Algorithm 3.2: *w*NAF Algorithm

Input: A positive integer k, width w

Output: wNAF(k)

1. $i \leftarrow 0$

2. *while* $k \geq 0$ do

2.1. *if* k is odd then $k_i = 2 - (k \bmod 2^w)$, $k = k - k_i$;

2.2. *else* $k_i \leftarrow 0$

2.3. $k = \frac{k}{2}$; $i = i + 1$;

3. *return* $(k_{i-1}, k_{i-2}, ..., k_0)$

Figure 3.2 shows the Big Integer representation of 192 bit key size and *w*NAF representation for varying window size.

Big integer:
437352739857664006357930435496927561584355920663 2

W=2

00010-100-1000-10001010100100-100-1010-1000-10-10-100-10010-10010100-1000-1000010100-1000-10-10000010000010101010 0-10-10-100100100000-1000-1000-10000-100101010000000-10001010000-1000000-101

W=3

000-30000-1000-1000-3003000100-1003000-1000300-3000-100-30000-300-300000-10000-300-300000300-10000100000-300300100300-3000100100000-1000-1000-10000-100-3003000000000-1000-3001000-10000003

W=4

000-30000-1000-10005000-7000-300003000-1000-500070003000700 00-7000000-1000050000-1000-5000000010000050005000-5000700000100000-1000-1000-10000700005000000000-10005000000-10000003

Figure 3.2 *w*NAF output

*w*NAF is computed using Algorithm 3.2, where $k \bmod 2^w$ denotes the integer u, where $u \equiv k (mod\ 2^w)$ and $-2^{w-1} \leq u < 2^{w-1}$. The expected running time of the Algorithm 3.2 is approximately $[1D + (2^{w-2} - 1)A] + \left[\frac{m}{w+1}A + mD\right]$, where m, A and D are size of k, addition and doubling process respectively. A key is represented in binary format such that the total

number of ones in the entire bit length is reduced by increasing the window size. Therefore, there is no existence of adjacent 1's. It implies that the average hamming distance and average hamming weight are reduced significantly. The mathematical operation over 0-bit is less costly compared to 1-bit.

Binary representation of an integer has more number of 0's in the Most Significant Bit (MSB). In this case, the *w*NAF skips the consecutive zeros starting from right-to-left, whereas sliding window Non-Adjacent Form (swNAF) considers the process left-to-right to avoid 0 bits in the given k. Hence, it is possible to reduce the pre-computation time in swNAF. Though it is advantageous in terms of time, the window size is variable in swNAF. Therefore, adjustment of window size leads to run-time error. Hence, the window size is fixed as 4 in the existing ECC based cryptosystems. To avoid this issue, a rule based engine is proposed in this research to fix the window size based on the available resources. In the proposed research work, the existing swNAF adopts rule-based engine to fix the window size for a given point on elliptic curve.

3.1.3 Rule Engine

There is a trade-off between the computational cost and the window size in point multiplication process as shown in Table 1.4. In real-time applications, the number of bits for representation may vary depending on the user. Varying window size leads to an erroneous result and calculations are more complex and expensive. Therefore, there is a need for an optimal window size estimator (Shah et al 2010, Huang et al 2011). To overcome these issues, this research proposes rule engine to ensure the optimum window size.

Rule Engine for selecting the window size is shown in Figure 3.3. Number of pre-computation, number of addition operation and memory required to store the intermediate result is estimated for the given kP. These outputs are given to Rule Engine. The rules are divided into three parts such as decrease, stay and increase. The sample rules framed by the rule engine is shown below. Overall, there are 14 rules and rules are designed based on the window control system. Rules determine the size of the window. For example: In increment rules, if the pre-computation time is low (number of 1's is half the total bits from MSB), addition operation is low (number of 1's is 1/3 of the total bits) and memory space is low (depending on the sensor mote) then window size is increased gradually.

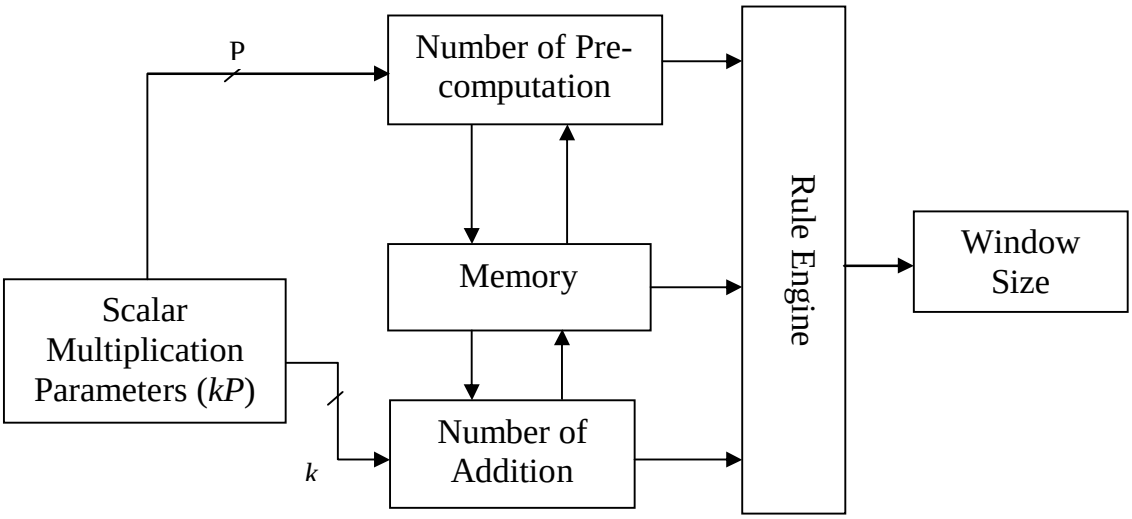

Figure 3.3 Rule engine for selecting window size

IF ((memory, pre-computation, addition) == low) || IF (((memory, pre-computation) == low) && (addition == average)) || IF (((memory, addition) == low) && (pre-computation == average)) || IF ((memory == low) && (pre-computation, addition == average)) || IF (((memory, addition) == low) && (pre-computation == high)) ||IF ((memory == average) && ((pre-computation, addition) == low)) ||IF ((memory, addition == average) && (pre-computation == low)) || IF (((memory, pre-computation) == average) && (addition == low)) then

{

Increment (window_size) by 1

}

else

{

IF (((memory, pre-computation) == average) && (addition == high)) || IF (((memory, addition) == high) && (pre-computation == low)) || IF ((memory, addition == high) && (pre-computation == average)) || IF (((memory, pre-computation) == high) && (addition == low)) || IF ((memory, pre-computation == high) && (addition == average)) || IF ((memory, pre-computation, addition) == high) then

{

Decrement (window_size) by 1

}

}

else

{

Stay (window_size)

}

Depending on the antecedent part, the rules are selected for fixing the window size. If the values of pre-computation, addition and memory space belongs to stay part rules then there is no change in window size. Similarly, in decrement rules, if the pre-computation time is high (number of 1's is half the total bits from MSB), addition operation is high (number of 0's is 1/3 of the total bits) and memory space is high (depending on the sensor mote) then window size is decreased gradually. The proposed work is said to

be dynamic due to its property of adjusting the window size based on rules developed. Therefore, the proposed DswNAF is most suitable for dynamically changing real-time applications. The steps involved in algorithm is given in Algorithm 3.3.

Algorithm 3.3: DswNAF Algorithm

Input: *FindMinComputationalTime*(k), positive integer k represented in binary form.
Output : f_w

1. $w \leftarrow 2$, where w is the window size,
   $f_w \leftarrow 2$, where f_w is the temprary window size,
   $T_w \leftarrow 0$, where T_w is the time taken to compute k,
   $i \leftarrow l-1$, where l is the bit length of k.
2. While $i \geq 0$
2.1. *if* $k_i = 0$, w++; f_w++; goto step 2;
2.2. *else*: Find the largest $f_w \leq w$ such that $u \leftarrow (k_i .....k_{i-w+1})$ is odd
2.3 *if* $u > 0$ then $i \leftarrow i - w$; goto step 2;
3. *if* $T_w > T_{w+1}$ *then* f_w++;
4. *return*(f_w, u);

Input: *ExecuteDswNAF*(w, k, P) where window width w, positive integer k, $P \in F_p$.
Output : $Q = kP$.

1. Use WNaF algorithm to compute NAF for k with $w = 2,3,4...14$ and
   $f_w = FindMinComputationalTime(k)$;
2. Compute $P_i = iP$ for $i \in \left\{1,3,...\frac{2(2^{f_w} - (-1)^{f_w})}{3} - 1\right\}$;
3. $Q \leftarrow 1$;
4. While $i \geq 0$
4.1. $Q \leftarrow 2^t Q$;
4.2 *if* $u > 0$ then $Q \leftarrow Q + P_u$; else if $u < 0$ $Q \leftarrow Q - P_{-u}$;
4.3 $i \leftarrow i - t$;
5. *return*(Q);

In step 1, NAF is computed for the window size 2-14 and the window size f_w is fixed for the given k, which takes the minimum time to

obtain the NAF representation. Step 2 is the pre-computation process of point P(x,y). Step 3 initializes the algorithm starting from left-to-right and process the doubling operation if the bit is zero else perform addition operation for a positive or negative (±ve) value. In step 4 – step 4.3, if 0 is present then the window size is incremented by 1 and left shift is performed for doubling based on the number of 0's present in *k.* For other ±ve integers, addition operation is performed in step 4.2. Then decrement the iterations with respect to number of zeros present. Finally, return Q=kP. Average length of zeros in DswNAF is given by Equation (3.1).

$$v(w) = \frac{2^w}{w-1} + \frac{(-1)^w}{(w-1).(2)^{w-2}} \quad (3.1)$$

where *v(w)* is the average length of zeros and *w* is the window size. Ds*w*NAF is computed using Algorithm 3.3, where $k \bmod 2^w$ denotes the integer *u*, where $u \equiv k(mod\ 2^w)$ and $-2^{w-1} \leq u < 2^{w-1}$. The expected running time of the Algorithm 3.3 is approximately $[mD] - [(2^{w-5})A]$, where m, A and D are the size of k, addition and doubling operation respectively. The implementation of DswNAF is carried out in BouncyCastle (BC), which is the security package that supports cryptographic algorithms.

3.2 IMPLEMENTATION IN MICAz

TinyECC is a code package that provides a base for arithmetic operation of ECC in TinyOS. ECC operations on domain F_p, F_{2m} includes point addition, doubling and scalar multiplication. TinyECC follows National Institute of Standards and Technology (NIST).

ECC arithmetic operations based on prime (F_p), and finite field (F_{2m}) such as addition, subtraction, doubling, multiplication and inverse are built in the NN module for different sensor nodes such as Micaz and TELOSB. ECC module in Micaz performs many basic operations on elliptic

curve such as initialization of an elliptic curve; point adding, point doubling, point scalar multiplication and sliding window. ECDH and ECIES modules in Micaz perform key exchange, authentication and encryption on ECC.

3.2.1 Structure of TinyECC-Twisted Edward (TinyECC-TE)

The relation between the components of TinyECC-TE is shown in Figure 3.4. A unique feature of TinyECC is its configurability. It provides a number of optimization switches which in turn does ON or OFF switching based on developers' need. Thus flexibility in integrating TinyECC into sensor network applications is achieved.

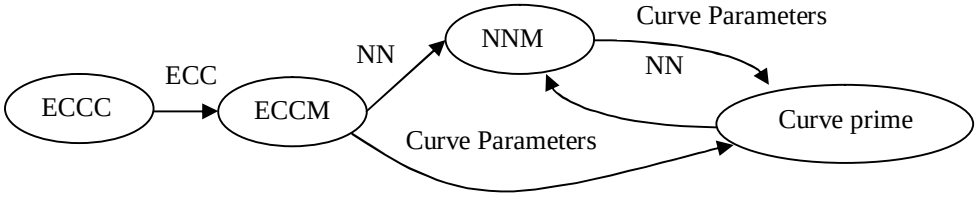

Figure 3.4 Component diagram of TinyECC

Elliptic Curve Cryptography Component (ECCC) provides an interface between the software program and the hardware. The basic definitions of the software program are given in Elliptic Curve Cryptography Module (ECCM). It consists of initialization, start, fire and stop functions to execute the arithmetic operation on elliptic curve points. Natural Number Module (NNM) provides the basic arithmetic operation such as addition, subtraction, multiplication, shift operations and inverse based on integer. Curve Prime initializes the NIST curves based on prime number.

TinyECC provides an interface curve parameter to initialize an elliptic curve. It defines 7 different elliptic curves of 128,160 and 192bits. This interface is implemented using secp128r1, secp128r2, secp160k1, secp160r1, secp160r2, secp192r1 and secp192k1 curves, where r1, r2 represent random selection of Weistress curve and k1 represents Koblitz

curve (Liu & Ning 2008). The proposed research performs analysis on secp160k1 curve of 160-bits. The curve is modified with the specification of 160-bits Twisted Edward curve. The NN module performs the basic operations involved in ECC. The existing addition, doubling and multiplication process operations are modified based on Twisted Edward law and DswNAF. The Twisted Edward Curves is a highly optimized form of an elliptic curve cryptography (Chu et al 2013). The Twisted Edward based point addition and point doubling operations are presented in Algorithm 3.4 and 3.5 respectively.

Algorithm 3.4: Point Addition

Input: Point $J = (X_1 : Y_1 : Z_1)$ *with* $E_1H_1 = T_1 = X_1Y_1/Z_1$ *and*
$K = (X_2 : Y_2 : Z_2)$ *with* $Z_2 = X_2Y_2$.
Output: Sum $L = P_1 + P_2 = (X_3 : Y_3 : Z_3)$.
where $A, B, C, D, E_3, F, G, H_3$ *are temporary variables*
$A \leftarrow (Y_1 - X_1) \cdot (Y_2 + X_2)$
$B \leftarrow (Y_1 + X_1) \cdot (Y_2 - X_2)$
$C \leftarrow 2 \cdot Z_1 \cdot T_2$
$D \leftarrow 2 \cdot T_1$
$E_3 \leftarrow D + C$
$H_3 \leftarrow D - C$
$F \leftarrow B - A$
$G \leftarrow B + A$
$X_3 \leftarrow E_3 \cdot F$
$Y_3 \leftarrow G \cdot H_3$
$Z_3 \leftarrow F \cdot G$
return $(X_3 : Y_3 : E_3 : H_3 : Z_3)$

Considering, $P_1 = (X_1/Z_1, Y_1/Z_1, Z_1)$ and $P_2 = (X_2/Z_2, Y_2/Z_2, Z_2)$ points on ECC, the point addition $P_3 = (X_3/Z_3, Y_3/Z_3, Z_3)$ is obtained where $P_3 = P_1 + P_2$. Similarly point doubling is obtained as $P_3 = P_1 + P_1$. P_1, P_2 and P_3 are the projective points of Twisted Edward elliptic curve and X_i, Y_i, Z_i are the coordinates of the ECC point on the curve. The subtraction operation is

obtained from addition operation. -X is the imaginary part of X coordinate that is given to the addition operation. ie., the subtraction operation changes the sign of the X coordinate and uses the value as such.

Algorithm 3.5: Point Doubling

Input: Point $P_1 = (X_1 : Y_1 : E_1 : H_1 : Z_1)$.
Output: Double $P_3 = 2 \cdot P_1 = (X_3 : Y_3 : E_3 : H_3 : Z_3)$.
where A, B, C, E_3, F, G, H_3 *are temporary variables*
$A \leftarrow X_1^2$
$B \leftarrow Y_1^2$
$C \leftarrow 2 \cdot Z_1^2$
$H_3 \leftarrow A + B$
$E_3 \leftarrow (X_1 + Y_1)^2 - H_3$
$G \leftarrow B - A$
$F \leftarrow C - G$
$X3 \leftarrow E_3 \cdot F$
$Y_3 \leftarrow G \cdot H_3$
$Z_3 \leftarrow F \cdot G$
return $(X_3 : Y_3 : E_3 : H_3 : Z_3)$

The multiplication in the traditional algorithm is replaced by repeated addition operation. Addition makes the process simple and finding the inverse of it is much simpler in the elliptic curve than multiplicative group. Even though the process is simple, the computation time increases due to the repeated addition and doubling process. In order to reduce the time, point addition and doubling in TinyECC is replaced by Twisted Edward's point addition and doubling. Twisted Edward curve has fewer modular multiplications than Weierstrass curves.

The Tiny Elliptic Curve Cryptography (TinyECC) is implemented using Koblitz and Twisted Edward curves. The arithmetic operations of Elliptic curves are implemented in TinyOS platform using nested C (nesC) (Gay et al 2003, 2009) programming. The implemented curve specifications are fused in two different nodes namely $Node_A$ and $Node_B$. Both $Node_A$ and $Node_B$ are synchronized with SerialForwarder.java and the results are collected from COM port 4.

All the above mentioned optimization techniques except sliding window require more memory (RAM & ROM) and higher execution cost. To overcome these issues, the proposed research work modifies sliding window technique to achieve variable window size in ECCM module. Key generated by DswNAF using TinyECC-TE is given to ECIES for encryption process.

3.2.2 Elliptic curve Integrated Encryption Scheme (ECIES) with Proposed DswNAF

ECIES (Martínez et al 2010) is a variant of the Integrated Encryption Standard (IES) used over elliptic curve cryptography. ECIES provides better security with lower key size for higher computation time compared to ECDSA.

The rule based engine is applied to the existing ECIES in BouncyCastle (BC), which is a security package implemented in java. The existent ECIES generally passes the EC parameters to a separated java class file called multiplier that computes the scalar multiplication. Existing definitions for the multiplier are modified to bring the optimization technique into effect. A variable value of pre-computation and addition help the rule engine to change dynamically. This leads to efficient decision making to select the suitable window size and fix the threshold value.

The rule system is modeled in java platform using jFuzzyLogic to interface it along with the ECIES. The Big Integer is a defined data type used in Java platform to represent integers with byte size greater than 64 (i.e. byte size greater than long data type). The working of the rule engine is adjusted by setting the rules with weights. The weights are distributed in such a way that the total weight is equal to unity. For example:

IF ((storage IS low) AND (precomp IS average)) AND (doubling IS low) THEN windowsize IS incremented by 1.

The rules are defined to decrease the latency. The window size of output parameter is modeled with triangular membership function where each state has midpoints at 0, 0.5 and 1.0 respectively.

3.2.3 Elliptic Curve Diffie Hellman (ECDH) in Micaz

ECDH is a key agreement protocol that allows two parties to establish a shared secret key over an insecure channel. A shared secret key is established between two parties by generating private and public key using elliptic curve arithmetic operations.

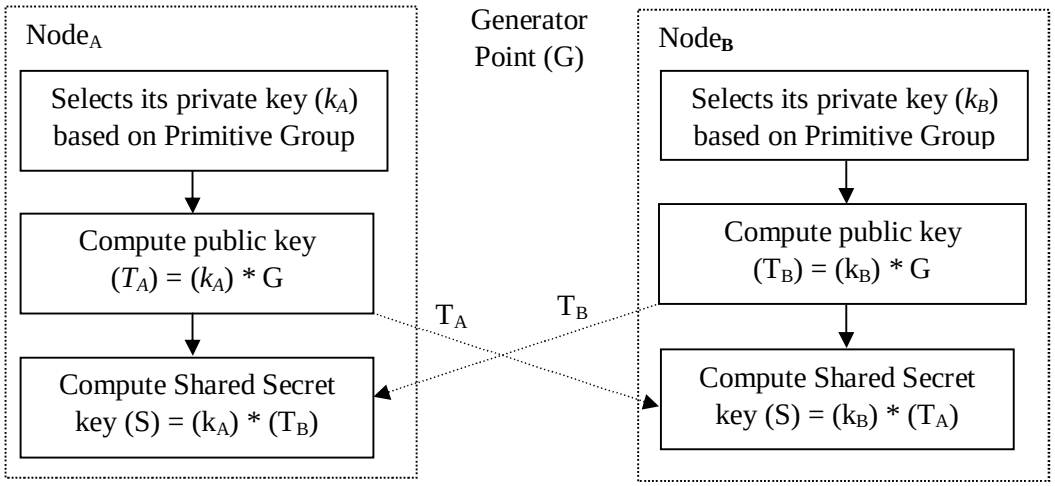

Figure 3.5 Elliptic Curve Diffie Hellman (ECDH) key exchange

A common generator point ‘G’ is assigned between two parties to derive the shared secret key over a prime field or finite fields. Figure 3.5 shows the Elliptic Curve Diffie Hellman (ECDH) Key Exchange Algorithm between $Node_A$ and $Node_B$ based on elliptic curve over prime field. Steps involved in Elliptic Curve Diffie Hellman Key Generation and Exchange process are as follows:

1. k_A, and k_B are the two random value chosen with the value of k in the range of *1* to *2p* $(1 < k < 2p)$ for $Node_A$ and $Node_B$. k_A is the private key of $Node_A$, k_B is the private key of $Node_B$.
2. $Node_A$ computes the public key $T_A = (k_A * G_x)$, $Node_B$ performs the public key calculation $T_B = (k_B * G_x)$, *where* G_x is the x-coordinate of the generator point G.
3. $Node_A$ sends public key T_A to $Node_B$. Similarly $Node_B$ sends public key T_B to $Node_A$.
4. $Node_A$ computes the shared secret key $S = (k_A * T_B)$ and $Node_B$ compute shared secret key $S = (k_B * T_A)$.

In sensor network based security, the broadcast public key from one node is multiplied with the private key of the receiving node. By doing this, the shared secret key is obtained. After the key exchange process, it is highly essential to include confidentiality mechanism for recognizing the relevant data. Therefore, this research uses ECIES module in WSN to ensure confidentiality of the data.

3.2.4 Elliptic Curve Integrated Encryption Scheme (ECIES) in Micaz

Integrated Encryption Scheme (IES) is a hybrid encryption scheme, which provides semantic security against an adversary (Martínez et al 2010). The schematic diagram of ECIES shown in the Figure 3.6. The steps involved in encryption and decryption process, is shown in Algorithm 3.6.

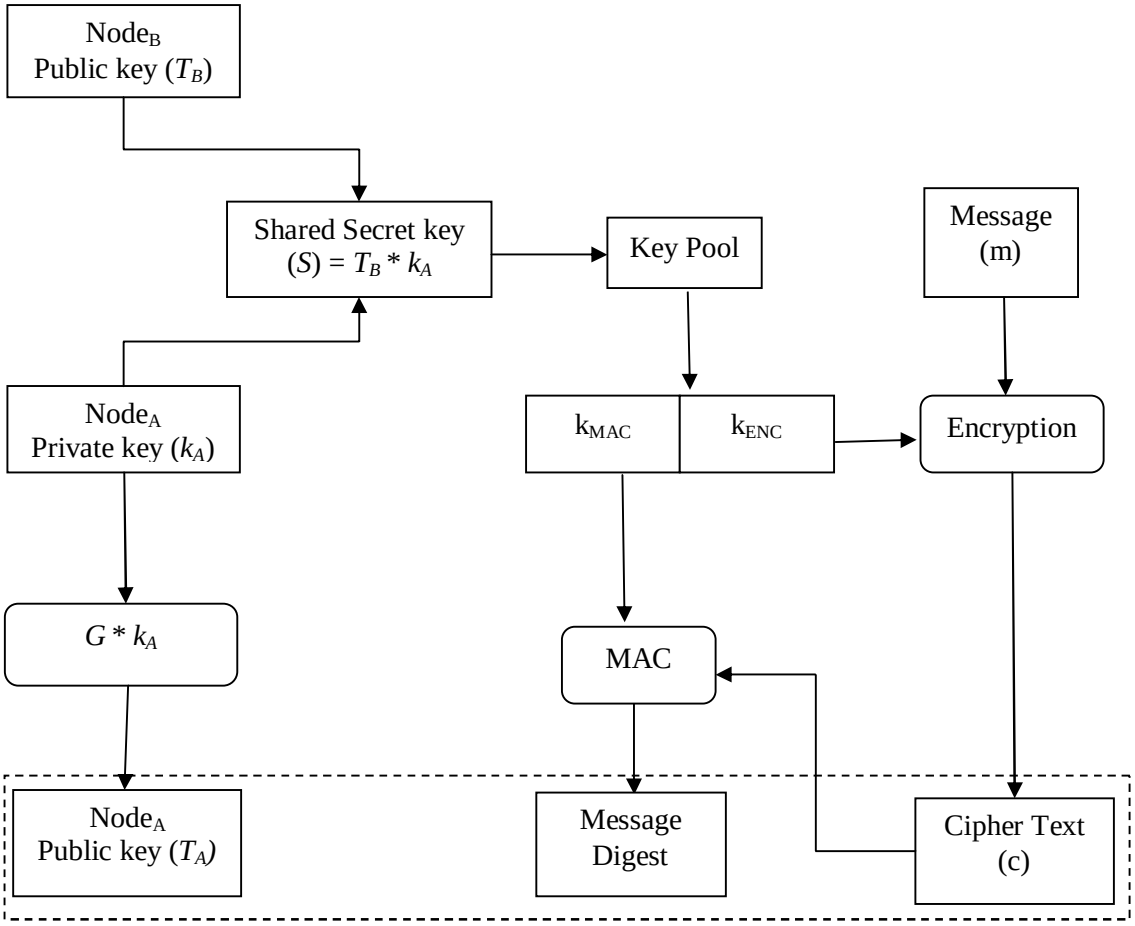

Figure 3.6 Functional diagram of ECIES

3.3 RESULT AND DISCUSSION

In this section, the performance of the proposed DswNAF based on Koblitz and Twisted Edward curve is used in Elliptic Curve Diffie Hellman (ECDH) and Elliptic Curve Integrated Encryption Standard (ECIES) for key generation, authentication and encryption respectively are evaluated in Micaz mote. Hence, this process ensures message confidentiality, authentication and message integrity for WSN to overcome MITM attack.

Algorithm 3.6: ECIES

To encrypt a message, $Node_A$ does the following:

1. A random number r = £ [1, *p*-1] is generated and $R = r*G_x$ is calculated.
2. A shared secret key $S = x_A$ is derived, where $P = (x_A, y_A) = r\ k_B$ (and $P \neq 0$) is stored in Key Distribution Function (KDF) using ECDH with SHA-1.

   $k_{ENC} \| k_{MAC} = KDF (S \| S_1)$
3. Symmetric encryption is performed using Advanced Encryption Standard with Code Block Chaining (AES-CBC) mode.

   $c = E(k_{ENC}, m)$
4. Hash function (HMAC-SHA-1-160) is used with 160-bit key for Message Authentication Code (MAC) generation process.

   $d = MAC (k, c \| S_2)$
5. The random value, cipher text and authentication code (R||c||d) are concatenated and transmitted to $Node_B$.

To decrypt the ciphertext (R||c||d) $Node_B$ does the following:

1. The shared secret key $S = x_B$ is derived such that point $P = (x_B, y_B)$. k_B*R.
2. The derived key is same as $Node_A$.

   $k_E \| k_M = KDF (S \| S_1)$.
3. MAC is used to verify the received message authentication code.

   $d = MAC (k_{MAC}, c \| S_2)$.
4. Symmetric encryption scheme is used to decrypt the message.

   $m = E^{-1}(k_{ENC} , c)$.

3.3.1 Performance Analysis of Proposed DswNAF

Existing point multiplication techniques such as wNAF and sliding window NAF (swNAF) are discussed in literatures (Win et al 1998) to perform elliptic curve arithmetic operation. These optimization techniques reduce the required computation time to perform point multiplication. To compute point multiplication (Q=*k*P), pre-computation is required for point P, wNAF and swNAF are used in integer *k* to reduce the computation time. However, the number of pre-computations is increased to compensate the computational overheads in point multiplication process. Table 3.1 gives the amount of computation time taken for processing various scalar values of *k* using wNAF algorithm as recommended by NIST.

Table 3.1 Computation time of various scalar values using wNAF algorithm

	Computation Time Taken for Various Scalar Value *k* (ms)					
Window Size	**160**	**192**	**224**	**256**	**384**	**521**
2	759.33	990.30	1645.21	1651.95	2779.40	2583.07
3	652.99	746.33	1266.99	1378.63	1592.28	1482.09
4	627.46	686.67	1003.78	781.46	1375.26	893.10
5	603.42	616.89	669.82	616.41	863.80	597.16
6	522.10	499.48	667.42	577.11	739.12	564.44
7	514.88	460.50	515.84	**474.46**	656.35	496.59
8	478.31	398.43	484.56	486.49	640.95	**373.41**
9	476.87	**335.32**	422.01	494.67	460.98	415.27
10	**475.30**	581.28	**356.56**	499.65	**353.68**	386.40
11	475.42	571.42	624.58	768.95	1225.48	1698.54
12	483.60	502.48	614.52	669.85	1012.54	925.46
13	482.64	512.56	584.75	654.78	525.45	1198.54
14	476.38	507.85	598.56	568.74	524.21	857.46

Existing wNAF has fixed window size as 2. The minimum window size is considered as 2 and the maximum window size is fixed according to the size of bit length as recommended by NIST. However, the window size is restricted to 14 because of the storage constraint in WSN. From the given Table 3.1, it is observed that the amount of computation time taken for 160-bit length k is less when the window size 10. The amount of time taken to perform wNAF is decreased linearly upto window size 10, but the computation time increases beyond the window size 10. Similarly, the computation time for various key length *k* is listed in Table 1.4 and the minimum time is highlighted for which the corresponding window size may be fixed. Figure 3.7 shows the computation time analysis of wNAF algorithm using various window sizes.

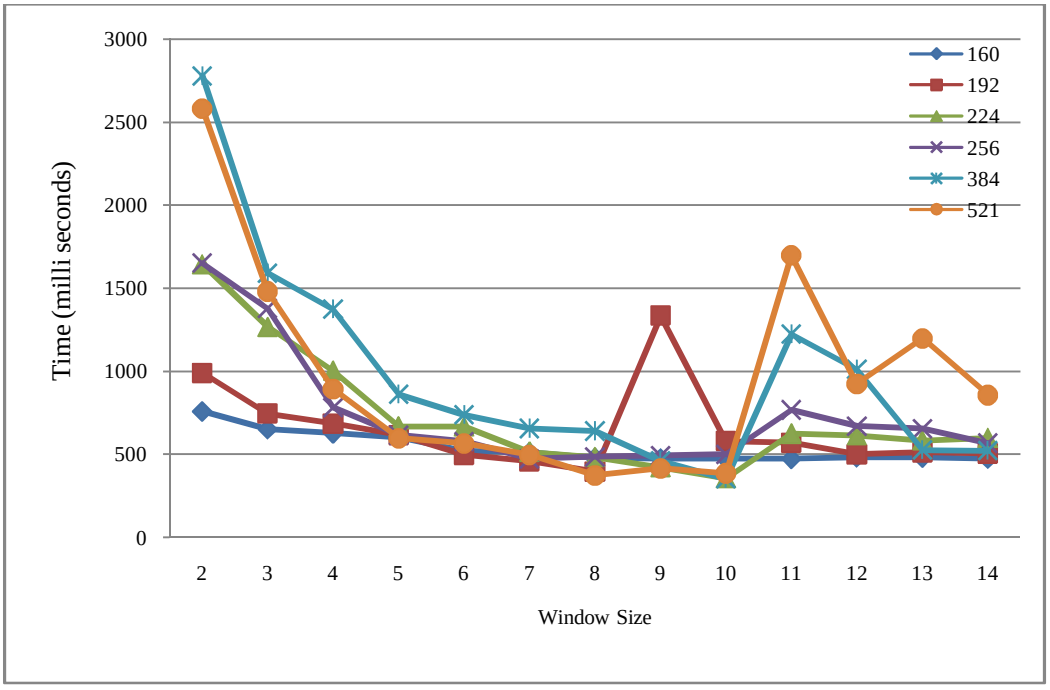

Figure 3.7 Performance of computation time analysis for wNAF algorithm using various window sizes

The size of window and efficient representation of *k* decides the speed of the point multiplication process. Sliding window NAF (swNAF) representation of *k* increases the speed of point multiplication by increasing

the size of window size compared to the *w*NAF. However, the number of pre-computations is increased in swNAF to reduce the required number of point addition process. Further, to increase the speed of point multiplication process, a Rule Engine is adopted in the proposed DswNAF by adjusting the window size.

Figure 3.8 shows the output of the proposed rule engine. The window size is adjusted based on the rules defined. The rule engine increases the window size if the number of point addition is less and number of pre-computation is high. Figure 3.8 shows the output of dynamic controller which decides the adjusted tradeoff between the pre-computations and point addition. The rules are defined in the FCL tool for the controller. This is simulated in java using jFuzzyLogic and the performance of the proposed Rule Engine is tested. The animated output of the rule engine controls the window size and decides the window size based on the rules defined in FCL.

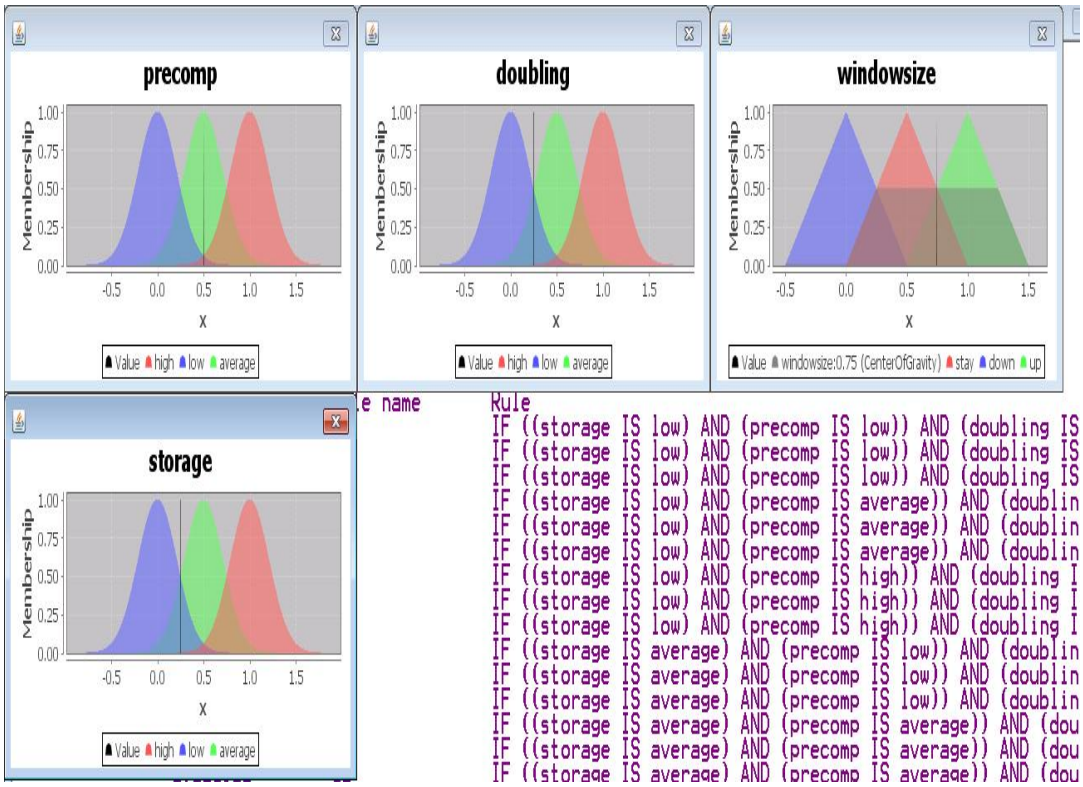

Figure 3.8 Simulated output of dynamic controller

From Figure 3.8, it is observed that the average number of pre-computation, less number of doubling or addition results in less utilization of storage, which in turn increases the window size. The rule engine controls the window size either by increment or decrement process.

Table 3.2 Computation time of various scalar values using the proposed DswNAF algorithm

	Time Taken for various scalar value k in bits (ms)					
Window Size	**160**	**192**	**224**	**256**	**384**	**521**
2	119.33	140.988	152.537	219.423	286.789	95.758
3	121.74	144.356	153.499	191.994	271.391	97.683
4	142.43	144.838	142.914	189.108	286.789	98.645
5	100.56	126.553	139.063	170.823	271.391	81.803
6	105.86	124.629	144.357	125.591	254.068	85.653
7	98.644	128.477	78.434	1064.34	154.943	84.69
8	100.56	108.268	39.458	42.826	106.342	780.5
9	93.832	83.246	37.052	40.901	56.299	25.022
10	70.734	40.42	44.751	38.976	58.224	627.48
11	72.179	30.315	36.571	41.863	53.893	25.503
12	401.79	29.833	43.788	37.533	52.45	25.022
13	27.909	34.645	39.939	38.495	51.006	23.578
14	25.022	36.089	58.705	35.608	48.6	58.225
15	36.571	42.322	8.18	36.571	50.043	5.774
16	29.353	34.164	8.662	35.127	48.119	5.774
17	25.503	48.119	8.661	36.089	48.6	6.255
18	25.503	26.465	8.661	34.646	46.675	6.255
19	32.24	31.277	10.586	34.646	46.675	6.256
20	24.059	26.466	8.661	33.202	56.299	5.774
21	17.804	27.428	8.661	35.127	57.262	5.775
22	17.804	30.315	11.067	33.202	12.03	5.774
23	17.804	31.278	7.699	34.646	11.549	5.774
24	94.794	43.307	8.18	42.344	11.548	5.294
25	50.525	5.293	8.18	9.142	11.549	5.775
26	4.812	5.774	8.662	8.662	12.03	5.775
27	4.812	5.293	7.699	8.661	11.549	5.293

Table 3.2 gives the computation time taken for processing various size of scalar value *k* using DswNAF algorithm. The proposed DswNAF algorithm extends the window size upto 27 by processing the scalar value from left-to-right in computing the point multiplication. The proposed DswNAF computes pre-computation values faster than *w*NAF using the increased window size. The window size decides the number of consecutive one's present in the scalar value k in the proposed DswNAF. The number of zero is increased to perform point-doubling operation. The major operation in the proposed DswNAF is the point-doubling or addition operation that consumes less number of computations. The amount of computations carried for point multiplication in DswNAF is less when compared to the wNAF due to its window size.

It is observed that the computation time taken for 160-bit key length is less, because of the resulting window size as 5, in the proposed DswNAF algorithm. The performance of the proposed DswNAF is analyzed in terms of the amount of computation time and window size. The time taken to perform the proposed DswNAF is changed abruptly for the window size. The computation time for various length of k is listed in Table 3.2. The window size of the corresponding minimum time taken to obtain the scalar value *k* is fixed as the window size in the proposed DswNAF. The window size is fixed as 5 for the proposed DswNAF to obtain the minimum computation time. However, the existing wNAF takes minimum computation time for window size 14. Figure 3.9 shows the computation time analysis of DswNAF algorithm using various window size.

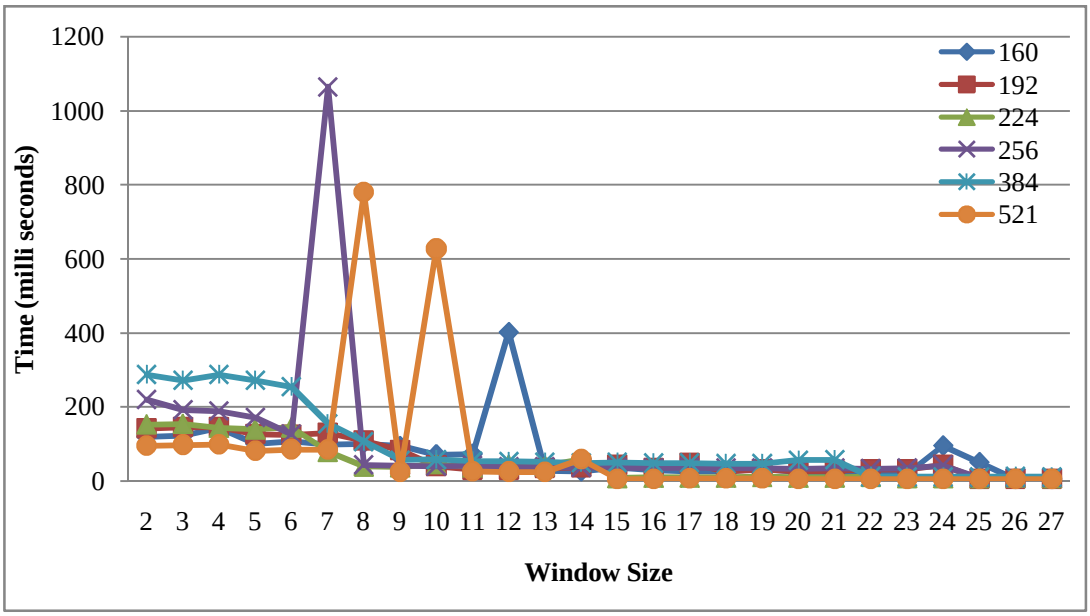

Figure 3.9 Timing analysis of the proposed DswNAF

Figure 3.9 shows the timing analysis of the proposed DswNAF algorithm for various scalar values vs. different window size. From the analysis, it is observed that the increase in window size decreases the computation time. For all scalar values, there is a substantial increase in computational complexity below the window size 5. Similarly, beyond value of window size 15, the computation time is invariably maintained as constant. The existing swNAF algorithm used in WSN application fixes the window size 4 as a threshold value to increase the performance of point multiplication operation in ECC. However, the proposed rule engine in DswNAF observes considerable amount of reduction in computation time for point multiplication operation with window size 5. The proposed DswNAF is adopted in point multiplication operation of ECIES cryptosystem. The performance analysis of the proposed DswNAF algorithm based ECIES cryptosystem shows more reduction in computation time than the existing swNAF. Though the number of point addition and point doubling operations is reduced, the number of pre-computation is increased in the proposed DswNAF.

The proposed DswNAF algorithm is implemented in BouncyCastle (BC), which is a security provider in java. The 160-bit size of scalar value is used to implement the proposed DswNAF for ECIES cryptosystem as recommended by NIST. The execution time of encryption and decryption algorithm is same as similar operations are performed. Figure 3.10 shows the execution time of proposed DswNAF in ECIES cryptosystem. From the result, it is observed that the ECIES algorithm consumes 75396 ms using the existing swNAF algorithm. The execution time of ECIES is reduced to 37975 ms using the proposed DswNAF algorithm and the time reduction is approximately 50% as given in Table 3.3.

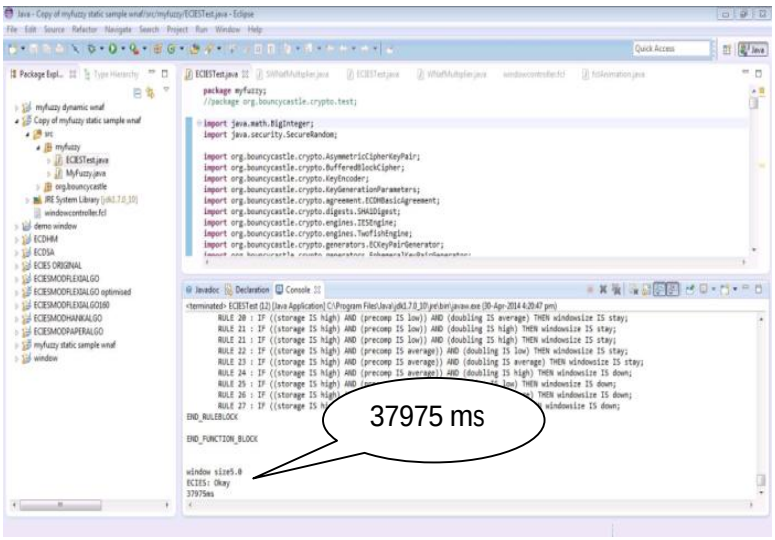

Figure 3.10 Execution time of proposed DswNAF in ECIES

Table 3.3 Timing analysis of ECIES

Scheme	Timing (ms)
Traditional ECIES	75396
ECIES with Proposed DswNAF	37975

The proposed DswNAF of ECIES cryptosystem exhibits a tradeoff between the window size and the computation time. However, it reduces the computation time that requires larger memory for pre-computation. Hence, the proposed rule engine fixes threshold window size as 5. However, increase in window size requires large memory in real time applications. This large memory requirement is not suitable for WSN due to resource constraint.

Moreover, the proposed DswNAF is prone to timing attack due to the bit-pattern analysis on Weierstrass curves. The computation time required to perform the conventional point multiplication depends on the bit-patterns in koblitz curve. To prevent timing attack, a specific bit-pattern should be avoided in these curves. Further, to reduce the computation time for processing point addition, Hisil et al (2008) suggested the extended coordinates of Twisted Edward that provides a fast point addition algorithm.

3.3.2 Implementation of TinyECC and TinyECC-Twisted Edward

The Tiny Elliptic Curve Cryptography (TinyECC) is implemented using Koblitz and Twisted Edward curves. The elliptic curve arithmetic is implemented in TinyOS platform using nested C (nesC) (Gay et al 2003, Gay et al 2009) programming. The specification of two curves are fused into different two nodes namely $Node_A$ and $Node_B$. Both $Node_A$ and $Node_B$ are synchronized with SerialForwarder.java and the results are observed from COM port 4.

The different optimization techniques in scalar multiplication process of TinyECC-TE includes Barrett Reduction, Hybrid Multiplication, Squaring, Projective Coordinates, Sliding Window, and Shamir's Trick (Hankerson et al 2004, Liu & Ning 2008). The optimization techniques help to reduce the running time of the algorithm. From the observed results, it is identified that the proposed work needs 9 multiplication, 4 squaring and 4

addition operations (9M+4S+4A) to perform point multiplication operation. However, the existing Koblitz curve based point multiplication technique requires 10 multiplication, 1 squaring and 7 addition operations (10M+1S+2D+7A). Therefore, the proposed method reduces 1 multiplication, 2 doubling and 3 addition operations for point multiplication. The performance of Algorithm 3.4 and Algorithm 3.5 is measured in terms of computation time. The performance results of Koblitz curve based optimization technique and Twisted Edward curve based optimization technique are discussed in the following subsections.

3.3.2.1 Performance analysis of ECDH using Koblitz and Twisted Edward curves

In this research, the performance of Koblitz curve and Twisted Edward curve based Elliptic Curve Diffie Hellman (ECDH) is analyzed using Micaz motes. It is observed that the twisted Edward curve based key generation using ECDH takes lesser time than the existing koblitz curve. However, the generated shared secret key between $Node_A$ and $Node_B$ is taken only once, during the entire communication. Table 3.4 shows the execution time for key generation using ECDH with koblitz curve (TinyECC) and twisted Edward curve (TinyECC-TE). It also shows the execution time for key generation using ECDH by adopting the optimization techniques such as Barrett Reduction, Hybrid Multiplication and Squaring, Projective Coordinates, proposed DswNAF for Scalar Multiplications and Shamir's Trick.

The initialization of private key takes place in $Node_A$ & $Node_B$ for the selection of private key in random. Hence, the time taken for initialization gets different. The generation of public key by the $Node_A$ & $Node_B$ varies due to the multiplication process. Assuming the private key is k_A, the generator point G(x,y) is multiplied with the coordinates of x and y to obtain the public

key $T_A = k * G(x,y)$ using point multiplication operation. This public key T_A is transmitted to the $Node_B$ through Zigbee transceiver (CC2420).

Table 3.4 Computational time of ECDH using TinyECC and proposed TinyECC-TE

ECDH (in Seconds)	Without Optimization		With Optimization	
	TinyECC using Koblitz curve	**TinyECC using Twisted Edward curve**	**TinyECC using Koblitz curve with swNAF**	**TinyECC using Twisted Edward with Proposed DswNAF**
ECDH Initialization	0.000591363	0.000591	0.9271501	0.9271658
Public key generation in $Node_A$	13.735425	14.512952	0.8993658	0.90565334
Public key generation in $Node_B$	14.403715	15.434399	1.8067827	0.9063373
Key Agreement in $Node_A$	13.755545	14.561885	1.0607042	1.0602349
Key Agreement in $Node_B$	29.685535	28.823105	2.125352	2.1059017
Total Execution Time for ECDH	**71.58081136**	**73.332932**	**6.8193548**	**5.90529304**

Table 3.4 shows the execution time for various steps involved in ECDH algorithm such as initialization of private key, public key generation and key agreement in $Node_A$ & $Node_B$. It also shows that the time taken for generating public key changes due to the point multiplication operation on $Node_A$ & $Node_B$. The existing TinyECC fixes window size 4 as threshold in swNAF to increase the performance of point multiplication operation of ECC.

The result shows the different amount of time taken for $Node_A$ & $Node_B$ to compute public key in ECC. The proposed rule engine in DswNAF provides considerable amount of reduction in computation time for point multiplication operation and aids in deciding the minimum time required to share the secret key. The computation time for sharing the secret key using Twisted Edward is increased by 2.44% on $Node_A$ & $Node_B$ than the Elliptic curve using Koblitz. By adding swNAF with Twisted Edward in TinyECC, the computation time is increased by 4.4%. The proposed DswNAF reduces the overall computation time by 13.4% compared to the existing swNAF in TinyECC-TE.

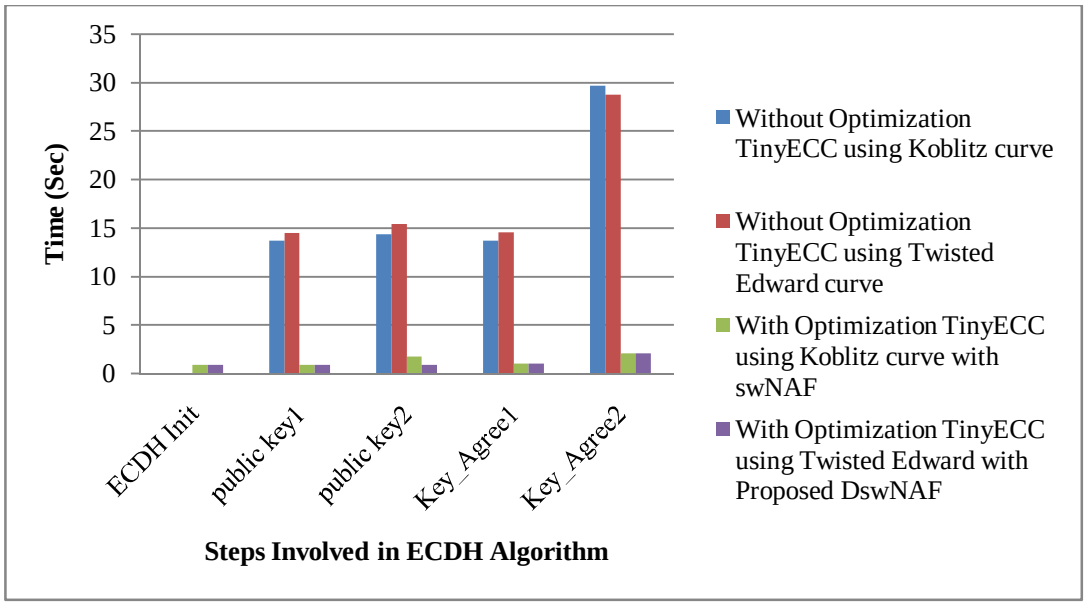

Figure 3.11 Various steps involved in ECDH and its execution time compared with the existing Koblitz curves and twisted Edward curves

In Figure 3.11, for each iteration, the key generation process uses different private key. Hence, different private key leads to produce the public key with different computation time. This is due to different point additions and point doubling operation involved in point multiplication operation to produce both public key and shared secret key in ECDH. The security of shared secret key depends on the complexity involved in the retrieval of the key and the number of elliptic curve point selected from Koblitz curve or

Twisted Edward curve. More number of elliptic curve points decides the high level of the security and computational complexity of shared secret key generation process. However, Twisted Edward curve provides more security with less number of selected elliptic curve points.

3.3.2.2 Performance analysis of ECIES using Koblitz and Twsited Edward curve

ECIES algorithm is chosen for WSN application to ensure confidentiality, integrity and authentication. The computation time for ECIES encryption using elliptic curves is nonlinear. The time taken to generate the keys using Twisted Edward curve is almost constant throughout all iterations. The performance results of ECIES computation time is given in Table 3.5.

Table 3.5 Comparison of computation time of ECIES using TinyECC and proposed TinyECC-TE

ECIES (in Seconds)	**Without Optimization**		**With Optimization**	
	TinyECC using Koblitz curve	**TinyECC using Twisted Edward curve**	**TinyECC using Koblitz curve with swNAF**	**TinyECC using Twisted Edward with Proposed DswNAF**
ECIES Initialization	0.00119032	0.001190321	1.8543272	1.853632
public key generation	28.810425	28.811281	1.8016483	1.802146
Encryption	116.38396	58.199398	3.9743664	3.883139
Decryption	58.78016	58.782898	2.6308713	2.6208713
Total Execution Time for ECIES	**203.975735**	**145.7947673**	**10.261213**	**10.1597883**

Table 3.5 shows the execution time for various steps involved in ECIES algorithm such as initialization of private key, public key generation, encryption and decryption process in $Node_A$ & $Node_B$. The execution time of the adopted Twisted Edward curve is same as Koblitz curve for initialization of private key, public key generation, and decryption. The computation time for encryption is reduced by 49.99% using Twisted Edward curve. Twisted Edward curve reduces the overall computation time for initialization of private key, public key generation, encryption and decryption by 28.52%. Further, DswNAF and Twisted Edward curve reduces the computation time by 0.98% in ECIES. Figure 3.12 shows the comparison of the computation time for various steps as discussed above.

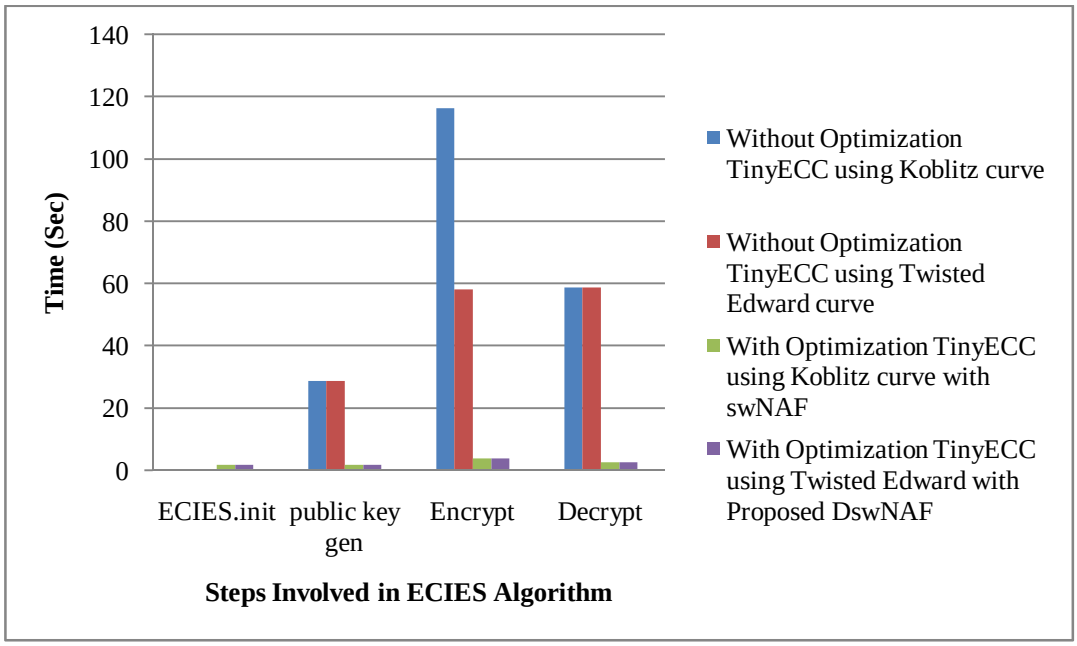

Figure 3.12 Comparison of computation time of various steps involved in ECIES using Koblitz curves and Twisted Edward curves

The computation time for key exchange, authentication and confidentiality algorithm using Twisted Edward and DswNAF is reduced in ECDH and ECIES. It is observed that the Twisted Edward curve based ECDH is suitable for time constraint applications.

3.3.2.3 Memory analysis

Table 3.6 shows the memory requirement of TinyECC and TinyECC-TE. The utilization of Read Only Memory (ROM) in TinyECC and the proposed TinyECC-TE of with and without optimization is 11% and 22.7% respectively.

Table 3.6 Memory requirement for TinyECC and proposed TinyECC-TE

Memory	Available memory in Micaz (Bytes)	without optimization (Bytes)		with optimization (Bytes)	
		TinyECC	TinyECC-TE	TinyECC	TinyECC-TE
ROM	128000	13912 (11%)	29056 (22.7%)	13890 (11%)	29098 (22.7%)
RAM	4000	938 (23%)	3028 (76%)	1440 (36%)	2357 (59%)

The utilization of Random Access Memory (RAM) in TinyECC and TinyECC-TE without optimization is 23% and 76% respectively. However, by adopting the optimization techniques with TinyECC and TinyECC-TE, the proposed method consumes 36% and 59% of RAM respectively with 23% of additional memory requirement. Even though, memory utilization is increased, Micaz board supports the proposed optimization DswNAF with TinyECC-TE. Therefore, this proposed research work is feasible for real-time applications.

3.4 SECURITY ANALYSIS OF TINYECC-TE

A Security Protocol ANimator (SPAN) is designed to help protocol developers in writing HLPSL specifications used in AVISPA. It provides an interactive graphical user interface for protocol execution. The proposed TinyECC-TE is tested using HLPSL in AVISPA tool. The following pseudo code illustrates the key exchange between two nodes. Assume node A and B are in the communication. Initially Node A and B select their private key Ka

and Kb respectively from the generator point (G), which is already defined in the nodes using elliptic curve parameters.

Step 1. A -> B: {Ka, Gx}Ta' // Node A computes its public key and send it to node B by authenticating the message using its private key.

Step 2. B -> A : {Ka, Kb}Tb // Node B receives the message from node A and multiply with private key of node B. Then it is authenticated by private key of node B and sent to node A.

Step 3. A -> B : {Kb}Ta // Node A sends the acknowledgement to node B.

The HLPSL notations used for the implementation are shown in Table 3.7. Table 3.8 shows the HLPSL code for key exchanging algorithm between two nodes.

Table 3.7 Notations used in HLPSL code

.	Multiplication
/\	Parallel process
{}	Encryption
=|>	Transition from one state to another state
A	Node A
B	Node B
Gx	x-coordinate of elliptic curve
inv	Multiplicative inverse
Ka, _inv(Ta)	Private key of node A
Ka’	New value of Ka
Kb	Private key of node B
Kb’	New value of Kb
new	New random value
RCV	Receive
SND	Send
Ta, ta	Public key of node A
Tb, tb	Public key of node B
witness	Authentication

Table 3.8 HLPSL code for communication between two nodes

State transitions in node A:

1. State=0 /\ RCV(start) =|> State':=1 /\ Gx':=new() /\ Ka':=new() /\ witness(A,B,auth_1,Ka') /\ SND({Ka'.Gx'}_inv(Ta))
   // Node A initiates the key exchange process and wait for the synchronization with node B (step 1) on left hand side (LHS). On right hand side (RHS) node A computes its public key by multiplying the Gx with Ka and performs the authentication and encryption process using its private key (Ka) and send it to node B.
2. State=1 /\ RCV({Ka.Kb'}_Tb) =|> State':=2 /\ SND({Kb'}_Ta)
   // Node A receives the message from node B after processing the step 1 in node B. This received message is decrypted with the public key (Tb) of node B and process the verification using its private key (Ka). The output produced is multiplied with private key (Ka). This output is the shared secret key between node A and Node B shown in LHS. Now node A send back the acknowledgement to node B containing the encrypted message of newly created private key using its public key (Ta) for the next transaction on RHS.

State transition in node B:

1. State=0 /\ RCV({Ka'.Gx'}_inv(Ta)) =|> State':=1 /\ request(B,A,auth_1,Ka') /\ Kb':=new() /\ SND({Ka'.Kb'}_Tb)
   // Node B waits for message from node A. After receiving the cipher (RCV({Ka'.Gx'}_inv(Ta))) sent by node A, node B decrypts and verifies this message using public key (Ka) of node A on LHS. Then node B multiplies the received public key of node A with its private key (Kb). This obtained value is authenticated and encrypted using public key (Tb) and send back to node A on RHS.
3. State=1 /\ RCV({Kb}_Ta) =|> State':=2
   //After the completion of step 2 in node A, node B receives the message and decrypts the message using the public key of node A and authenticate the message by its private key (Kb). This resultant is the shared secret key between node A and node B on LHS. RHS waits for the next transaction.

Figure 3.13 shows the output of the proposed TinyECC-TE key exchange between the node A and node B.

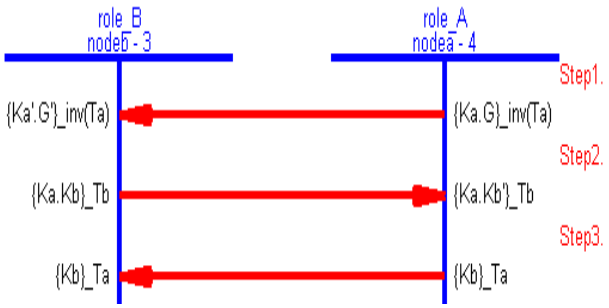

Figure 3.13 Proposed TinyECC-TE Key exchange between two nodes

3.4.1 MITM Attack

SPAN is also used to create an intruder to analyze the strength of the security. Here, the MITM attack is introduced to validate the secure relay metric of the proposed TinyECC-TE. Figure 3.14 shows that an intruder (I) eavesdrop all the messages from node A and transmits the received messages to node B as per steps shown below:

1. A → I:{Ka, Gx}Ta' // An intruder receives the message from node A
2. I →B:{Ka, Gx}Ta' // Received message is sent to node B with new nonce.
3. B →I:{Ka, Kb}Tb // Node B multiply the received value with its public key and authenticate using its private key and sent back to intruder.
4. I →A:{Ka, Kb}Tb // The received message of node B is sent to node A with new nonce.
5. A →I:{Kb}Ta // Node A sent back with new nonce to node B to intruder by authenticating the message using its public key
6. I -> B : {Kb}Ta // The received information is forwarded to node B

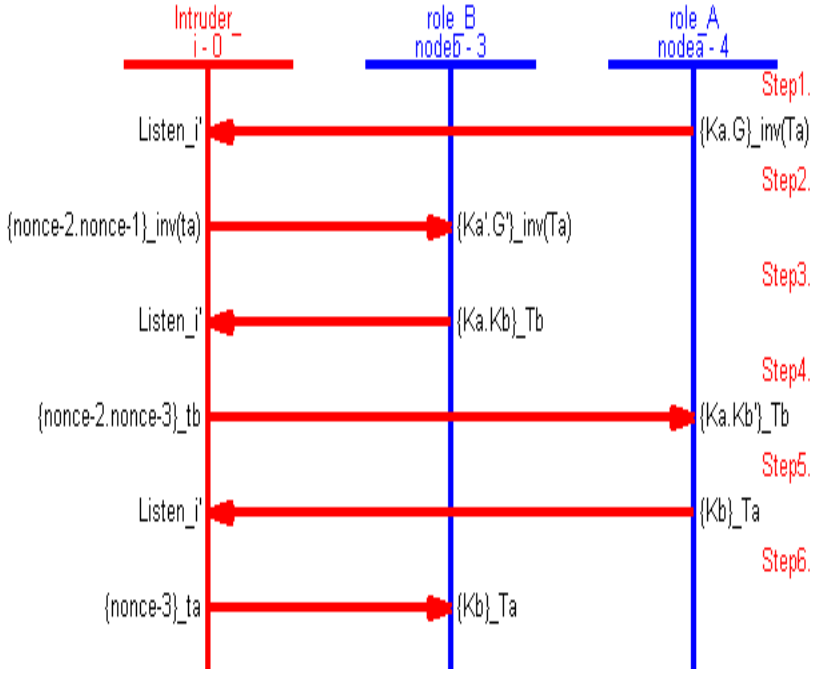

Figure 3.14 Proposed TinyECC-TE Key exchange between two nodes with MITM attack

After listening, an intruder performs multiplicative inverse process to obtain the private key of the node A and node B. The following HLPSL code shows the knowledge assumed for an intruder. “nodea”, “nodeb” are assumed as node ID and ta, tb are the public key of node A and node B respectively.

intruder_knowledge = {nodea, nodeb, ta, tb}

Figure 3.15 shows the security strength of the proposed TinyECC-TE. It can be seen that the system is safe even after introducing the MITM attack. From the experimental results, it is observed that MITM attack did not break the security of the proposed TinyECC-TE due to the complexity in the inverse process for finding private key (ta’) from the known public key (ta).

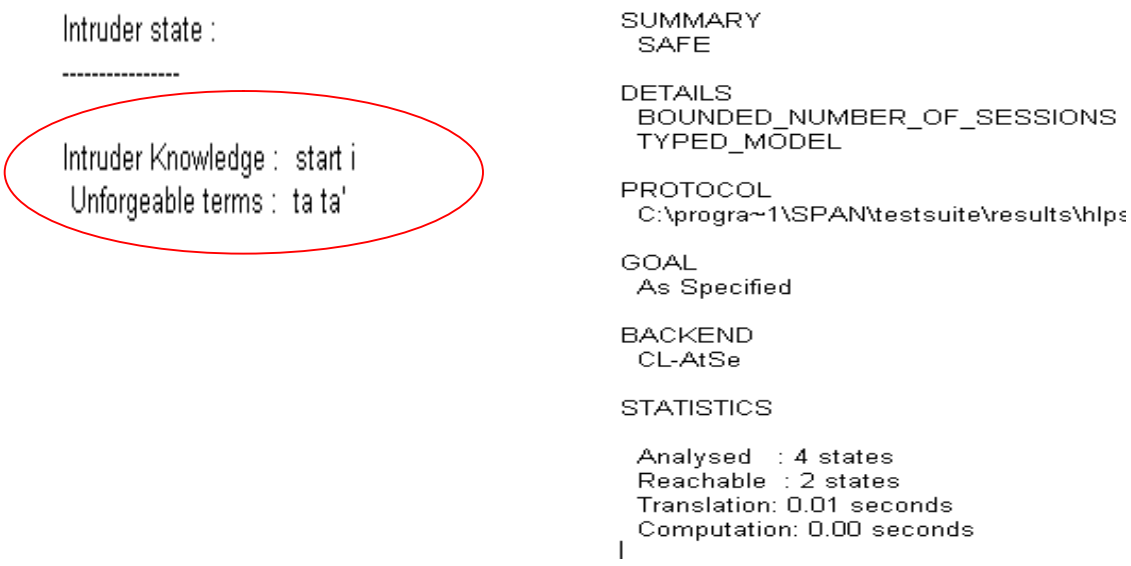

Figure 3.15 Evaluation of proposed TinyECC-TE under MITM attack

If the private key of a node is compromised then an intruder can act as a legitimate node else, it is a non-legitimate node. It is also observed that the network is always safe using the proposed ECDH based TinyECC-TE. The proposed ECDH protects the private key by increasing the computational complexity of multiplicative inverse process in ECC.

3.4.2 Simple Power Analysis (SPA) Attack

Figure 3.16 shows the power trace consumed on processing the secret key on ECC. Examining the power trace is known as Simple Power Analysis (SPA) attack (Hankerson et al 2004) that reveals the secret key. The existing elliptic curve point multiplication process ($Q=kP_x$) are vulnerable to SPA attack due to the difference in time taken by point addition and point doubling operation. In ECC, public key P_x is known publically and the random integer k is kept secret in ECC. If an attacker concentrates on power trace of NAF(k) which is used in the point multiplication operation then the k is easily revealed. The k is represented in the form of binary and converted into NAF(k) with window size as 2. In which '0' indicates the point doubling

operation and ±1 refers the point addition operation as shown in Figure 3.2. The number of 1's yields the substantial information about *k*, in which the time consumed for processing addition operation is longer than the doubling operation as shown in Figure 3.16. NAF(*k*) algorithm susceptible for SPA attack.

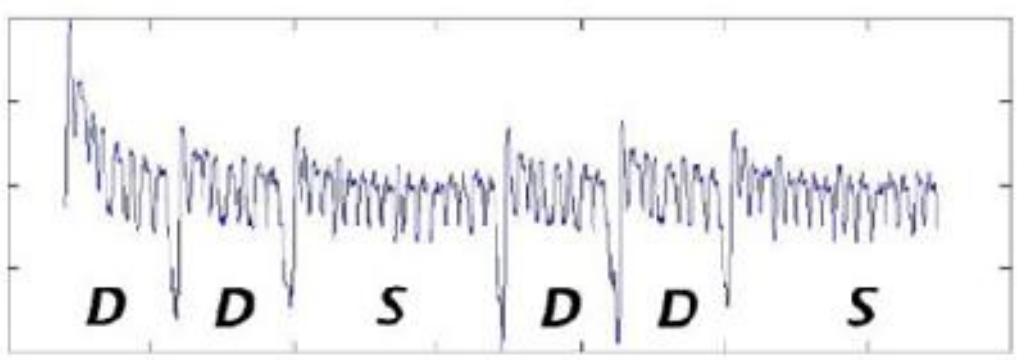

Figure 3.16 Power trace of addition (S) and doubling (D) operation on ECC

Bernstein & Lange 2011 suggested Twisted Edward curve based on Montgomery-form that depends on the bit-length and it is immune to timing attack like Simple Power Analysis (SPA) attack. In this research work, Twisted Edward curve is adopted in TinyECC for WSN applications. Since Montgomery based Twisted Edward curve has no point doubling operations, the required amount of time for point multiplication is reduced. Hence, it is immune to SPA attack.

Table 3.9 shows the output of 160-bit ECC of the proposed TinyECC-TE in micaz motes. However, the elliptic curve generates different keys in Koblitz curve and the proposed TinyECC-TE curve. In Koblitz curve the computation time varies in each step of key generation process, whereas the proposed TinyECC-TE consumes almost the same time for producing the secret keys on various rounds of different key. Hence, similar traces are

created by Twisted Edward curve in the point multiplication process. Thus the proposed algorithm is more resilient to SPA attack.

Table 3.9 Computation time of ECDH Using Koblitz curves and Twisted Edward curves

Computation Time of ECDH Algorithm using Optimization Techniques (in Seconds)					
Tiny ECC					
Rounds	**ECDH Parameter Initialization**	**Public Key Generation Node$_A$**	**Public Key Generation Node$_B$**	**Key Agreement Node$_A$**	**Key Agreement Node$_B$**
1	0.9271501	0.8993658	1.8067827	1.0607042	2.125352
2	1.2361965	0.60102284	1.8080502	1.4109129	2.0127983
3	1.3907303	0.8993723	1.8013559	1.06104	2.1159883
4	1.4834759	0.7228127	1.8036623	1.2706404	2.1192257
5	1.5452985	0.89935327	1.7986349	1.0605124	2.1116626
6	1.5894415	0.776072	1.8017596	1.2150532	2.1175132
Proposed Tiny ECC - TE					
1	0.9271658	0.90565334	0.9063373	1.0602349	2.1059017
2	1.0362051	0.90995667	1.0353536	1.0599551	2.0954018
3	1.0407248	0.90743953	1.0291634	1.0681451	2.109716

3.5 SUMMARY

This chapter presented two novel schemes. First scheme is based on Twisted Edward curve with TinyECC (TinyECC-TE). Another scheme is to select the optimum window size for swNAF using rule engine. The proposed Twisted Edward curve based TinyECC method adopts optimization technique for reducing the time complexity of point multiplication in ECC. In

this method point addition is adopted in TinyECC which reduces 1 multiplication, 2 doubling and 3 addition operations for point multiplication. In addition, the existing TinyECC uses swNAF in which the window size 4 is fixed as a threshold value to increase the performance of point multiplication operation of ECC. A significant reduction in computation time for point multiplication is obtained by including rule engine for selecting optimum window size dynamically in DswNAF with window size of 5 for 160-bits of key length. Hence, the proposed DswNAF performs well in Twisted Edward curve. The computation time of ECIES algorithm using proposed DswNAF is 50% lesser than the computation time of the existing swNAF algorithm. Hence, the proposed DswNAF based Twisted Edward reduces the overall computation time for key generation by 13.4% compared to the existing Koblitz curve. The experimental results show that the proposed TinyECC-TE is resilient to the MITM and SPA attack.