University of Waterloo
Department of Electrical and Computer Engineering

Software Implementation of Gong-Harn Public-Key Cryptosystem

Supervised by: Prof G. Gong

Susana Sin
September 30, 2003

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

With the emergence of the third generation network for mobile communication, while speech transmission is still dominating the airway, the demands for fax, short messages and data transmissions are growing rapidly. Combined with the rapid development of Internet applications, data security becomes ever more important. Designing cryptosystems that meet both power constraints and computing constraints of mobile units is very challenging.

In the most widely used modern cryptosystems, such as Diffie-Hellman (DH) key exchange scheme [1] and the digital signature algorithm [2], increasing the size of the modulus is necessary in order to strengthen their security. The Gong-Harn Public-key Cryptosystem (GH-PKC) [3] [4], however, has the benefit in reducing the size of the modulus while speeding up the computations with the same degree of security as existing cryptosystems. This public-key distribution scheme is desirable as it minimizes computational cost.

The GH-PKC employs sequence terms generated by linear feedback shift registers (LFSR) over GF(p) or GF(p^2) where p is a prime number. The security is based on the difficulty in solving discrete logarithm problem in GF(q^3), where q equals to p or p^2 depending on the implementation. In order to implement the GH-PKC over GF(p) with 1024-bit security, a 341-bit p should be used. The scope of this report is to explain the software implementation of GH-PKC over GF(p) with 1024-bit security.

2.0 THIRD ORDER CHARACTERISTIC SEQUENCE AND RECIPROCAL SEQUENCE

Elements required to generate a third order characteristic sequence will be explained in this section. In order to understand the arithmetic in the next section, the idea of reciprocal sequence will also be illustrated here. [3]

2.1. Third Order Characteristic Sequence

A third order characteristic sequence is a sequence generated by an irreducible polynomial *f(x)* of degree three over GF(*p*), where *p* is a prime, using specific initial states. This sequence can also be regarded as a linear feedback shift register (LFSR) sequence generated by *f(x).*

An irreducible polynomial *f(x)* of degree three over GF(*p*) has the form:

$$f(x) = x^3 - ax^2 + bx - 1$$

where the coefficients *a* and *b* are elements in GF(*p*). The initial state has to be chosen as follows:

$$s_o = 3, s_1 = a, s_2 = a^2 - 2b$$

where s_k denotes the k^{th} term in the sequence. Then the sequence $\{s_k(a,b)\}$ is called a third order characteristic sequence generated by *f(x)* over GF(*p*). The k^{th} state is denoted as:

$$\underline{s}_k = (s_k, s_{k+1}, s_{k+2})$$

The period of this characteristic sequence, Q, is a factor of $p^2 + p + 1$ and the maximum period is:

$$Q = p^2 + p + 1$$

Any polynomial which has the form:

$$f_k(x) = x^3 - s_k(a,b)x^2 + s_{-k}(a,b)x - 1$$

is also an irreducible polynomial with the same period as *f(x)* if and only if the greatest common divisor (gcd) of *k* and Q equals 1.

2.2. Reciprocal Sequence

Given the irreducible polynomial *f(x)* in Section 2.1, the reciprocal polynomial is:

$$f^{-1}(x) = x^3 - bx^2 + ax - 1$$

By choosing the corresponding initial states as given in Section 2.1, the sequence generated by $f^{-1}(x)$ is also a third order characteristic sequence and it is denoted as $\{s_{-k}(a, b)\}$. This sequence is the reciprocal sequence of $\{s_k(a, b)\}$. The k^{th} term in the reciprocal sequence $\{s_{-k}(a, b)\}$ is the same as the $-k^{th}$ term in the sequence $\{s_k(a, b)\}$. An example of characteristic sequence is shown in Figure 1.

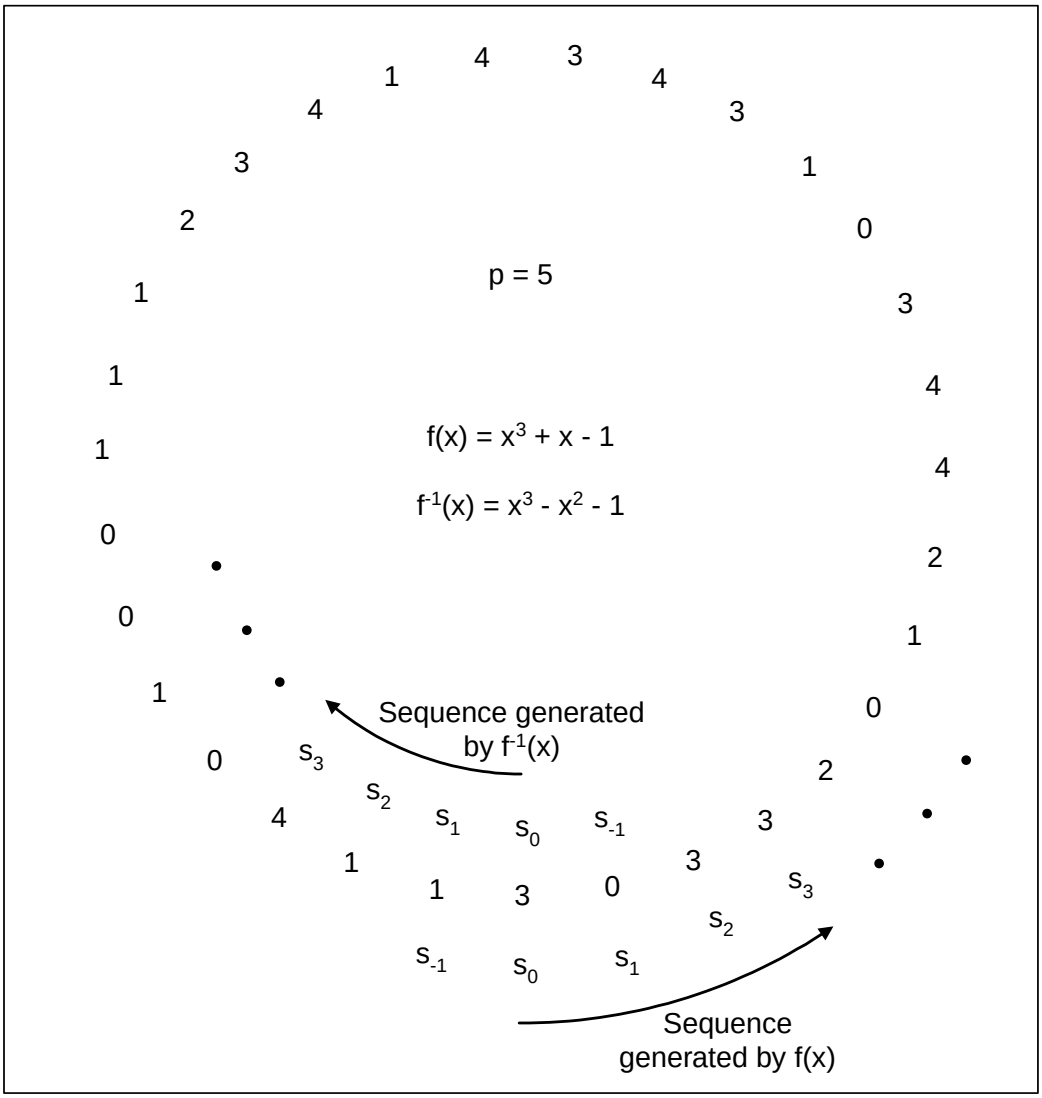

Figure 1: A characteristic sequence

3.0 ALGORITHMS

Algorithms for computing sequence terms given in references [3] and [4] are shown in this section.

3.1. Dual-State Fast Evaluation algorithm (DSEA Algorithm)

The DSEA algorithm is an efficient algorithm for computing the $\pm k^{th}$ terms in a third order characteristic sequence generated by an irreducible polynomial *f(x)* over GF(*p*). Basically, there are two sets of equations to be used depending on the bits in the binary representation of *k*.

First, express *k* in binary representation:

$$k = \sum_{i=0}^{n} k_i 2^{n-i} = k_0 2^n + k_1 2^{n-1} + \cdots + k_n$$

where *n* is the number of bits required to represent *k* in the binary form and it can be calculated as:

$$n = \lfloor \log_2 k \rfloor$$

A set of variable T_j is defined as follows:

$$T_0 = k_0 = 1,$$
$$T_j = k_j + 2T_{j-1}, \; for 1 \le j \le n$$

Since T_0 equals to k_0, which is the most significant bit in the binary representation of *k*, it must have the value of 1. Using the recursive equation given above to compute T_j, for *j* from 1 to *n*, the last value T_n equals to *k*.

To make it easier to see the terms in the two sets of equations in the algorithm, two variables are defined as:

$$t = T_{j-1}, t' = T_j$$

The two sets of equations to be used that depend on the bits in the binary representation of *k* are:

For $k_j = 0$ For $k_j = 1$

$$s_{t'+1} = s_t s_{t+1} - a s_{-t} + s_{-(t-1)} \qquad s_{t'+1} = s_{t+1}^2 - 2 s_{-(t+1)}$$
$$s_{t'} = s_t^2 - 2 s_{-t} \qquad s_{t'} = s_t s_{t+1} - a s_{-t} + s_{-(t-1)}$$
$$s_{t'-1} = s_t s_{t-1} - b s_{-t} + s_{-(t+1)} \qquad s_{t'-1} = s_t^2 - 2 s_{-t}$$

By having a for-loop for j from 1 to n, depending on the value of k_j, the corresponding set of equation will be chosen in the iteration to compute the $s_{t'+1}$, $s_{t'}$ and $s_{t'-1}$ terms. Notice that in the two sets of equations, there are some terms that belong to the reciprocal sequence *{s-k(a, b)}* such as s_{-t}. The s_t term used in the j^{th} iteration is the $s_{t'}$ term computed in the $(j-1)^{th}$ iteration. Similarly, the s_{-t} term used in the j^{th} iteration is the $s_{-t'}$ term computed in the $(j-1)^{th}$ iteration. The $s_{-t'}$ term can be computed by interchanging a and b to form the reciprocal polynomial $f^{-1}(x)$ and by interchanging all s_k terms with s_{-k} terms. Therefore, in each iteration of j, the corresponding set of equations need to be executed twice to obtain the $s_{t'+1}$, $s_{t'}$, $s_{t'-1}$, $s_{-(t'+1)}$, $s_{-t'}$ and $s_{-(t'-1)}$ terms. After the last iteration is performed, the $s_{t'}$ and $s_{-t'}$ terms are the $\pm k^{th}$ terms of the sequence.

In each iteration, because the corresponding set of equation needs to be executed twice there is a total of n iterations. The number of multiplications in the sets of equations corresponding to k_j equals 0 and k_j equals 1 are five and four respectively. On average, the probability that k_j equals 0 or 1 is ½. Therefore, on average, the total number of multiplications in GF(p) is:

$$2 \times r \times \left(\Pr(k_j = 0) \times 5 + \Pr(k_j = 1) \times 4 \right) = 2 \times \lfloor \log_2 k \rfloor \times \left(\frac{1}{2} \times 5 + \frac{1}{2} \times 4 \right) = 9 \lfloor \log_2 k \rfloor$$

3.2. Computation of a Previous Sequence Term

In the paper, it is proven that the third-order characteristic sequence has properties of duality and redundancy. Three elements in any state of the third-order characteristic sequence are not independent. If any two consecutive elements are known, the third remaining one can be uniquely determined according to the following:

Let

$$delta = s_{k+1} s_{-(k+1)} - s_1 s_{-1}$$

Then $s_{\pm(k-1)}$ can be computed as:

$$s_{k-1} = \frac{es_{-(k+1)} - s_{-1}D(e)}{delta}$$
$$s_{-(k-1)} = \frac{D(e)s_{k+1} - s_1 e}{delta}$$

where

$$D(s_k) = s_{-k}$$
$$e = -s_{-1}D(c_1) + c_2$$
$$c_1 = s_1 s_{k+1} - s_{-1}s_k$$
$$c_2 = s_k{}^2 - 3s_{-k} + (b^2 - a)s_{-(k+1)}$$

3.3. Computation of a Mixed Term $s_{\pm u(k+v)}$ with Known $\underline{s}_{\pm(k-1)}$ State

The procedure to compute a mixed term $s_{u(k+v)}$ is as follows. First, compute the sequence term $s_{(k+v)}$. Then, construct another irreducible polynomial as:

$$g(x) = x^3 - s_{(k+v)}x^2 + s_{-(k+v)}x - 1$$

and compute the $\pm u^{th}$ sequence terms generated by *g(x)*. This gives $s_{\pm u(k+v)}$ terms.

The algorithm for computing $s_{(k+v)}$ is given in reference [5]. This is a general result for LFSR sequence.

Define a transitional matrix, matrix *A*, as:

$$A = \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -b \\ 0 & 1 & a \end{bmatrix}$$

When the k^{th} state is multiplied by matrix *A*, it gives:

$$\underline{s}_k \cdot A = (s_k, s_{k+1}, s_{k+2}) \cdot \begin{bmatrix} 0 & 0 & 1 \\ 1 & 0 & -b \\ 0 & 1 & a \end{bmatrix} = (s_{k+1}, s_{k+2}, s_k - bs_{k+1} + as_{k+2}) = \underline{s}_{k+1}$$

which is the next state of the LFSR.

Define a state matrix, matrix *Mn*, as:

$$M_n = \begin{bmatrix} s_{n-2} & s_{n-1} & s_n \\ s_{n-1} & s_n & s_{n+1} \\ s_n & s_{n+1} & s_{n+2} \end{bmatrix}$$

The following properties are given in reference [5]:

(1) $\underline{s}_v = \underline{s}_0 \cdot A^v = \underline{s}_1 \cdot A^{v-1} = \underline{s}_2 \cdot A^{v-2} = \cdots = \underline{s}_{v-1} \cdot A^1$, $v = 1, 2, \ldots$

(2) $M_v = M_0 \cdot A^v \Rightarrow A^v = M_0^{-1} \cdot M_v$, if $\det(M_0) \neq 0$

In general,

$$\underline{s}_{k+v} = \underline{s}_k \cdot A^v = \underline{s}_k \cdot \left(M_0^{-1} \cdot M_v\right) = \underline{s}_k \cdot \left(\begin{bmatrix} s_{-2} & s_{-1} & s_0 \\ s_{-1} & s_0 & s_1 \\ s_0 & s_1 & s_2 \end{bmatrix}^{-1} \begin{bmatrix} s_{v-2} & s_{v-1} & s_v \\ s_{v-1} & s_v & s_{v+1} \\ s_v & s_{v+1} & s_{v+2} \end{bmatrix} \right), \text{ if } \det(M_0) \neq 0$$

In particular, the s_{k+v} term is equal to $\underline{s}_k$ multiplies to the first column of $(M_0^{-1} * M_v)$. Since state $\underline{s}_{k-1}$ is known instead of state $\underline{s}_k$, the s_{k+v} term is equal to $\underline{s}_{k-1}$ multiplied by the middle column of $(M_0^{-1} * M_v)$. To construct the matrix M_v, s_{v-2}, s_{v-1}, s_v, s_{v+1} and s_{v+2} are required. Sequence terms (s_{v-1}, s_v, s_{v+1}) can be obtained using the DSEA algorithm. The terms s_{v-2} and s_{v+2} can be obtained by:

$$s_{v+2} = a s_{v+1} - b s_v + s_{v-1}$$

$$s_{v-2} = s_{v+1} - a s_v + b s_{v-1}$$

Similarly for the $s_{-(k+v)}$ term.

The matrix M_0,

$$M_0 = \begin{bmatrix} s_{-2} & s_{-1} & s_0 \\ s_{-1} & s_0 & s_1 \\ s_0 & s_1 & s_2 \end{bmatrix} = \begin{bmatrix} b^2 - 2a & b & 3 \\ b & 3 & a \\ 3 & a & a^2 - 2b \end{bmatrix}$$

is invertible if and only if its determinant is not zero:

$$\det(M_0) = \left(b^2 - 2a\right)\left[3\left(a^2 - 2b\right) - a^2\right] - b\left[b\left(a^2 - 2b\right) - 3a\right] + 3\left[ab - 3 \cdot 3\right] \neq 0$$

Since it depends only on a and b, the determinant of M_0 can be guaranteed to be non-zero by choosing a and b correspondingly.

4.0 GONG-HARN PUBLIC-KEY DISTRIBUTION CRYPTOSYSTEM

GH-PKC is based on computing sequence terms in a third order characteristic sequence. The GH-PKC is explained in this section including GH Diffie-Hellman key agreement protocol and GH digital signature algorithm.

4.1. GH Diffie-Hellman (GH-DH) Key Agreement Protocol

The GH-DH is basically a Diffie-Hellman-like key agreement protocol using sequence terms. The DSEA algorithm was developed to compute these sequence terms efficiently. A pair of common keys between two users is generated by polynomials using each other's public key pair as coefficients.

The system parameters include a prime number p and an irreducible polynomial $f(x) = x^3 - ax^2 + bx - 1$ over GF(p). The period of the third order characteristic sequence is denoted as Q, and the protocol is executed as follows.

Two users, Alice and Bob, choose their own private keys K_A and K_B that satisfy:

$$0 < K_A, K_B < Q\ ,\ \gcd(K_A, Q) = 1 \text{ and } \gcd(K_B, Q) = 1$$

to ensure the polynomial they generate later is also irreducible over GF(p) with the same period as the period of sequence terms generated by $f(x)$.

Alice and Bob compute the $\pm K_A^{th}$ and $\pm K_B^{th}$ sequence terms generated by $f(x)$, namely $s_{\pm KA}$ and $s_{\pm KB,}$ as their public keys using the DSEA algorithm.

They send their public keys to each other to form polynomials $f_A(x)$ and $f_B(x)$ using the public keys as the coefficients. The $\pm K_A^{th}$ and $\pm K_B^{th}$ terms in the sequences generated by $f_A(x)$ and $f_B(x)$ respectively are the common key pair between the two users. For example, Alice forms a polynomial:

$$f_A(x) = x^3 - s_{+KB}x^2 + s_{-KB}x - 1.$$

She generates the common key pair by computing the $\pm K_A^{th}$ terms in the sequence generated by $f_A(x)$, namely $s_{KA}(s_{KB},s_{-KB})$. Figure 2 summarizes the GH-DH key agreement protocol.

System Parameters:

$f(x) = x^3 - ax^2 + bx - 1$, an irreducible polynomial over GF(p), where p is a prime number

	Alice	**Bob**
Secret Key:	K_A, $0 < K_A < Q$, $\gcd(K_A, Q) = 1$	K_B, $0 < K_B < Q$, $\gcd(K_B, Q) = 1$
Public Key Pair:	(s_{KA}, s_{-KA})	(s_{KB}, s_{-KB})
	(s_{KA}, s_{-KA}) →	
	← (s_{KB}, s_{-KB})	
Polynomial generated using the received public key pair:	$f_A(x) = x^3 - s_{KB}x^2 + s_{-KB}x - 1$	$f_B(x) = x^3 - s_{KA}x^2 + s_{-KA}x - 1$
Common Key Pair:	$\pm K_A$ terms generated by $f_A(x)$ $= s_{\pm KA}(s_{KB},s_{-KB})$	$\pm K_B$ terms generated by $f_B(x)$ $= s_{\pm KB}(s_{KA},s_{-KA})$

$$s_{\pm KA}(s_{KB},s_{-KB}) = s_{\pm KB}(s_{KA},s_{-KA})$$

Figure 2: GH-DH Key Agreement Protocol

4.2. GH Digital Signature Algorithm (GH-DSA)

The GH-DSA is similar to the ElGamal signature algorithm. In this report, Alice is assumed to be the signer and Bob is assumed to be the verifier. The system parameters are the same as the last section. Alice chooses her private key and computes her public key pair as detailed in the last section.

4.2.1. Signing Procedure

Alice randomly chooses k that satisfies:

$$0 < k < Q \text{ and } \gcd(k, Q) = 1$$

and computes (s_{k-1}, s_k, s_{k+1}) and its dual using the DSEA algorithm, such that s_k is co-prime with Q. If $gcd(s_k, Q) \neq 1$, then Alice should select another k.

To sign a message, m, Alice will compute the hash value of m:

$h = h(m)$, where $h(\ )$ is a hash function that Alice and Bob agree upon

and compute t in the signing equation:

$$h = kt + xr \mod Q$$

where r equals to the sequence term s_k. Then (r, t) is a digital signature of the message m.

Alice sends Bob (m, r, t), her public key pair (s_x, s_{-x}) together with (s_k, s_{k+1}) and its dual. Notice that only (s_k, s_{k+1}) are sent instead of (s_{k-1}, s_k, s_{k+1}). Since the previous sequence term can be computed using the two consecutive sequence terms as described in Section 3.2, it is not necessary for Alice to send all three terms to the verifier. Therefore, only (s_k, s_{k+1}) are sent to minimize the use of bandwidth.

4.2.2. Verification Procedure

In the verification procedure, Bob first computes $s_{\pm(k-1)}$ terms using the algorithm given in Section 3.2, (s_k, s_{k+1}) sequence terms and its dual. There are two different cases in verifying Alice's signature that Bob has to choose from depending on whether t and Q are co-prime.

Case 1: gcd(t, Q) = 1

In the signing equation, k and x are unknown to Bob, therefore, Bob cannot simply verify the equation by checking each parameter. However, Bob has Alice public key pair (s_x, s_{-x}). Instead of verifying each parameter in the signing equation, Bob can put both sides of the signing equation as sequence term indexes and verify to see if these sequence terms match:

$$s_{r^{-1}(h-kt)} = s_x \text{ and } s_{-r^{-1}(h-kt)} = s_{-x}$$

Let

$$u = -r^{-1}t \bmod Q \text{ and } v = -ht^{-1} \bmod Q$$

where h is the hash value of m using the same hash function as in the signing process. Then,

$$s_x = s_{-r^{-1}t(k-ht^{-1})} = s_{u(k+v)} \text{ and } s_{-x} = s_{r^{-1}t(k-ht^{-1})} = s_{-u(k+v)}$$

Bob computes u and v as shown above. By using the algorithm given in Section 3.3 to compute a mixed term, Bob can computes $s_{\pm u(k+v)}$ sequence terms and verify to see if they are the same as $s_{\pm x}$. If they match, then Bob accepts Alice's signature.

Case 2: gcd(t, Q) $\neq$ 1

If t and Q are not co-prime, $t^{-1} \bmod Q$ does not exist; therefore, Bob cannot use the same algorithm as in Case 1. Instead, Bob can verify to see if these sequence terms match:

$$s_{h-rx} = s_{kt} \text{ and } s_{-(h-rx)} = s_{-kt}$$

Let

$$u = -r \bmod Q \text{ and } v = -r^{-1}h \bmod Q$$

where h is the hash value of m using the same hash function as in the signing process. Then,

$$s_{kt} = s_{u(x+v)} = s_{-r(x-r^{-1}h)} \text{ and } s_{-kt} = s_{-u(x+v)} = s_{r(x-r^{-1}h)}$$

Similarly, Bob computes u, v as shown above. By using the algorithm given in Section 3.3 to compute a mixed term, Bob can computes $s_{\pm u(x+v)}$ and $s_{\pm kt}$ sequence terms. If these sequence terms match, then Bob accepts Alice's signature.

5.0 SOFTWARE IMPLEMENTATION

The software design, design issues and testing processes are explained in this section. Coding is done in C++ and a partially complete code listing is given in Appendix A.

5.1. Software Design

To facilitate the handling and calculation of these large integers, the ZZ [6] data type is used. This data type provides basic mathematical operators as well as other mathematical functions such as GCD( ) for use with large integers.

The first step in the program is to set up the system parameters. This involves setting the values of *a*, *b* and *p*, with *a* and *b* being the coefficient of the irreducible polynomial *f(x)* over GF(*p*). Since the system parameters should remain constant, they can be hard-coded in the program. The parameter, *Q*, is calculated to be the maximum sequence period for valid key selection tests unless the actual sequence period is known.

The program will then display a menu for the user to choose which operation to perform: to compute a shared key pair, to sign a message, to verify a signature or to terminate the program. Flowcharts for these operations except program termination are illustrated in Figure 3, Figure 4 and Figure 5.

5.1.1. Shared Key Computation

The program will prompt the initiator, Alice, to choose her private key from 0 to *Q*. After Alice has entered the secret key, *x*, the program will check to see if *x* is valid by testing if it is co-prime with *Q* and if *x* is less than *Q*. If any of these cases fails, the program will prompt Alice to choose another value for the private key. After validating the choice of private key, the program will express *x* in its binary representation. The program will then computes the key pair as the $\pm x^{th}$ terms of the sequence generated by *f(x)*. This key pair is Alice's public key pair and it will be given to Alice.

In setting up a session with Bob, Alice will send her public key pair to Bob and Bob will response by sending his public key pair to Alice. The program will prompt Alice to enter the received public key pair. The keys in the key pair will be used to set up another irreducible polynomial $f_A(x)$ over GF(p) and the $\pm x^{th}$ terms of the sequence generated by $f_A(x)$ will be the shared key pair between Alice and Bob. This can be done by setting a and b equal to the keys in the received key pair and by using the same algorithm for computing the $\pm x^{th}$ terms of the sequence generated by $f(x)$. This is the key pair shared between Alice and Bob.

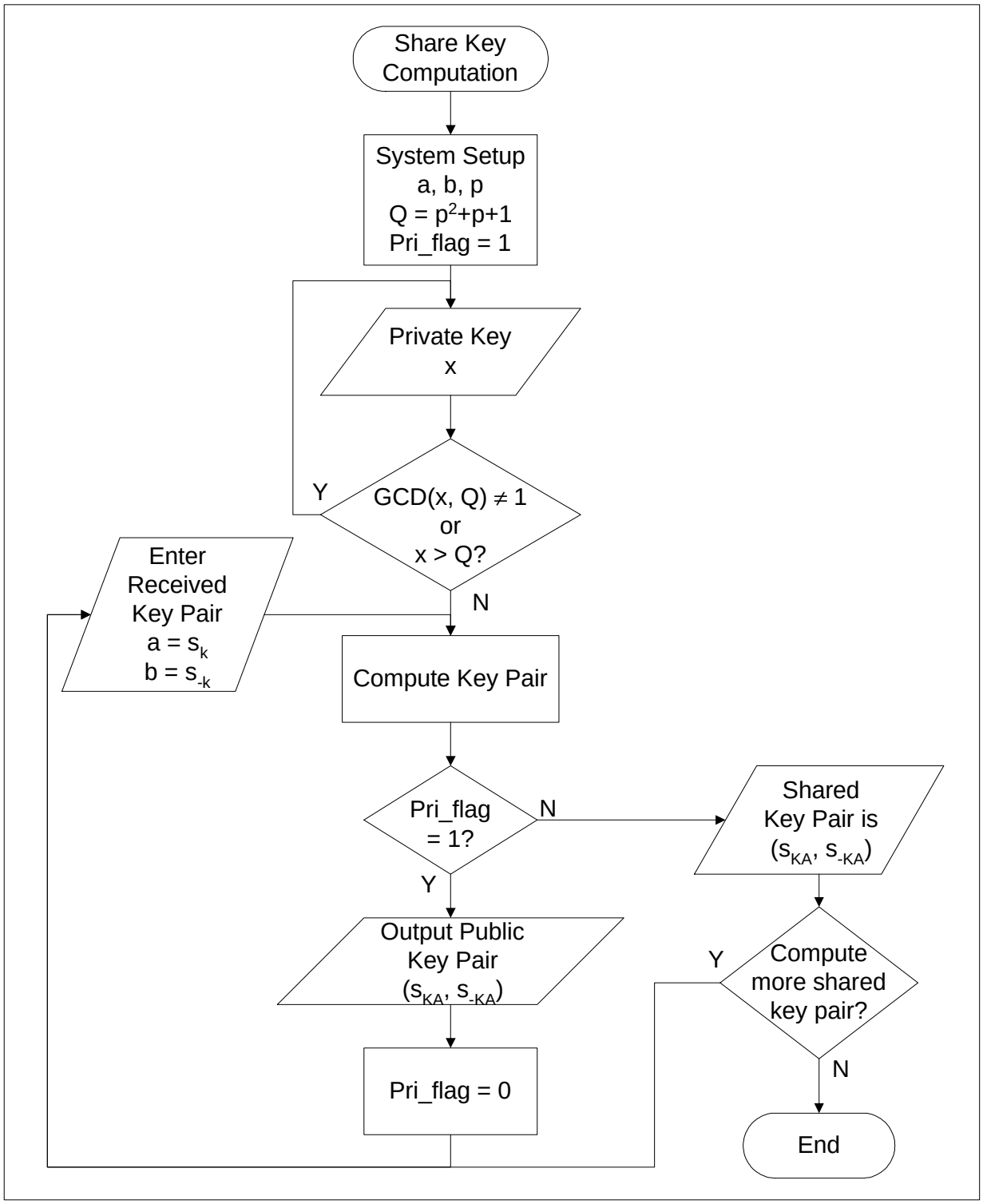

Figure 3: Flowchart Diagram for Shared Key Computation

5.1.2. Signature Generation

Since signature generation requires the public key pair of the user, the program will prompt the signer, Alice, to choose her private key as in the first part of Section 5.1.1. After computing the public key pair, the program will prompt Alice to enter a value for the signing key, *k*, with the same restrictions as her private key described in Section 5.1.1. The program computes (s_{k-1}, s_k, s_{k+1}) and its duals after a valid *k* value is entered. However, this is not the final validation of the signing key choice yet.

As describe in the signing equation in Section 4.2.1, the parameter *r* equals to the sequence term s_k and it should be co-prime with *Q*. Therefore, the program will perform the co-prime test on *r* and *Q* after s_k is computed. In computing a previous sequence term as describe in Section 3.2, *delta* must be a non-zero number. This algorithm would be used in signature verification as Bob has to compute s_{k-1} and its dual using (s_k, s_{k+1}) and its dual. Therefore, in the signing process, the program should also compute *delta* and check to see if it equals zero. If any of these tests fails, the program will prompt Alice to select another value for *k*.

The program will then ask Alice to enter the message that she needs to sign. The hash function is ignored in the implementation since it is only a hash function agreed upon between two users. The hash value, *h*, is set to be the same as the message, *m*. The program will compute the parameter *t* according to the signing equation. If *t* equals zero, *x* will be compromised as the term with *t* in the signing equation disappears. Therefore, *t* must be a non-zero number. If *t* is zero, the program will prompt Alice to select another value for *k*.

After the tests on *r*, *t* and *delta*, the choice of *k* is now validated, and the program will output (s_x, s_{-x}) as Alice's public key pair and *(m, r, t)* as digital signature of message *m*. It will also output (s_k, s_{k+1}) and its dual.

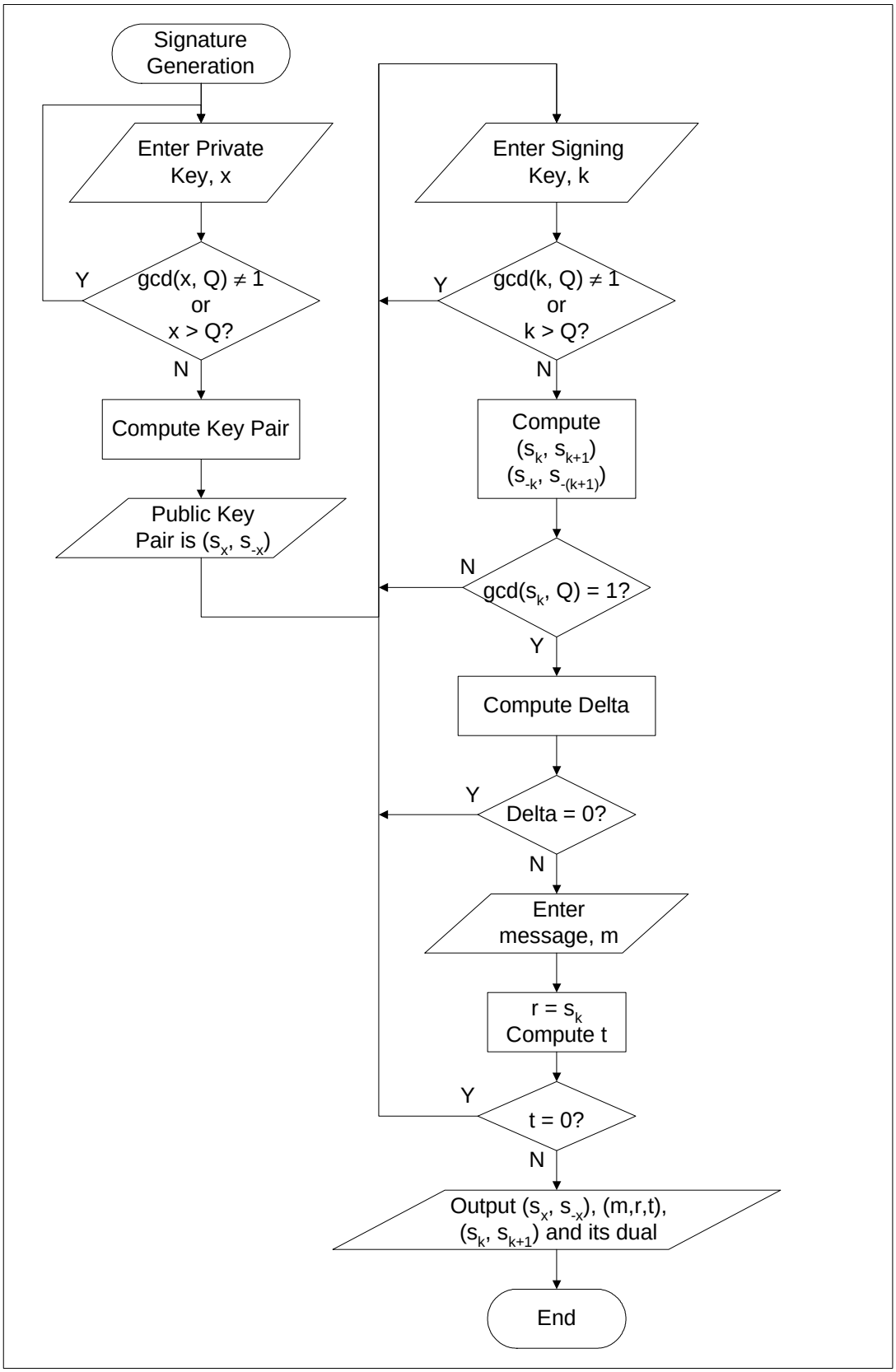

Figure 4: Flowchart Diagram for Signing Generation

5.1.3. Signature Verification

The program will prompt the verifier, Bob, to entered the received (m, r, t), (s_k, s_{k+1}) and its dual. As described in Section 4.2.2, the verifier, Bob, first computes $s_{\pm(k-1)}$ terms using (s_k, s_{k+1}) sequence terms and its dual. Since the signing key, k, is chosen such that *delta* is a non-zero number, the $s_{\pm(k-1)}$ terms can be computed easily.

Depending whether t is co-prime with Q, different value of u and v will be used. The program determine the corresponding u and v as described in Section 4.2.2 and compute $s_{\pm u(k+v)}$ or $s_{\pm u(x+v)}$ terms using the algorithm given in Section 3.3.

If t is co-prime with Q, then the program will prompt Bob to enter the received public key pair (s_x, s_{-x}). It will then compare to see if $s_{\pm u(k+v)}$ equals $s_{\pm x}$. If t is not co-prime with Q, then the program will compute $s_{\pm kt}$ terms using the same algorithm as it computes the $s_{\pm u(x+v)}$ terms. Then, it will compare to see if $s_{\pm u(x+v)}$ equals $s_{\pm kt}$. Again, the signature is valid if and only if these sequence terms match.

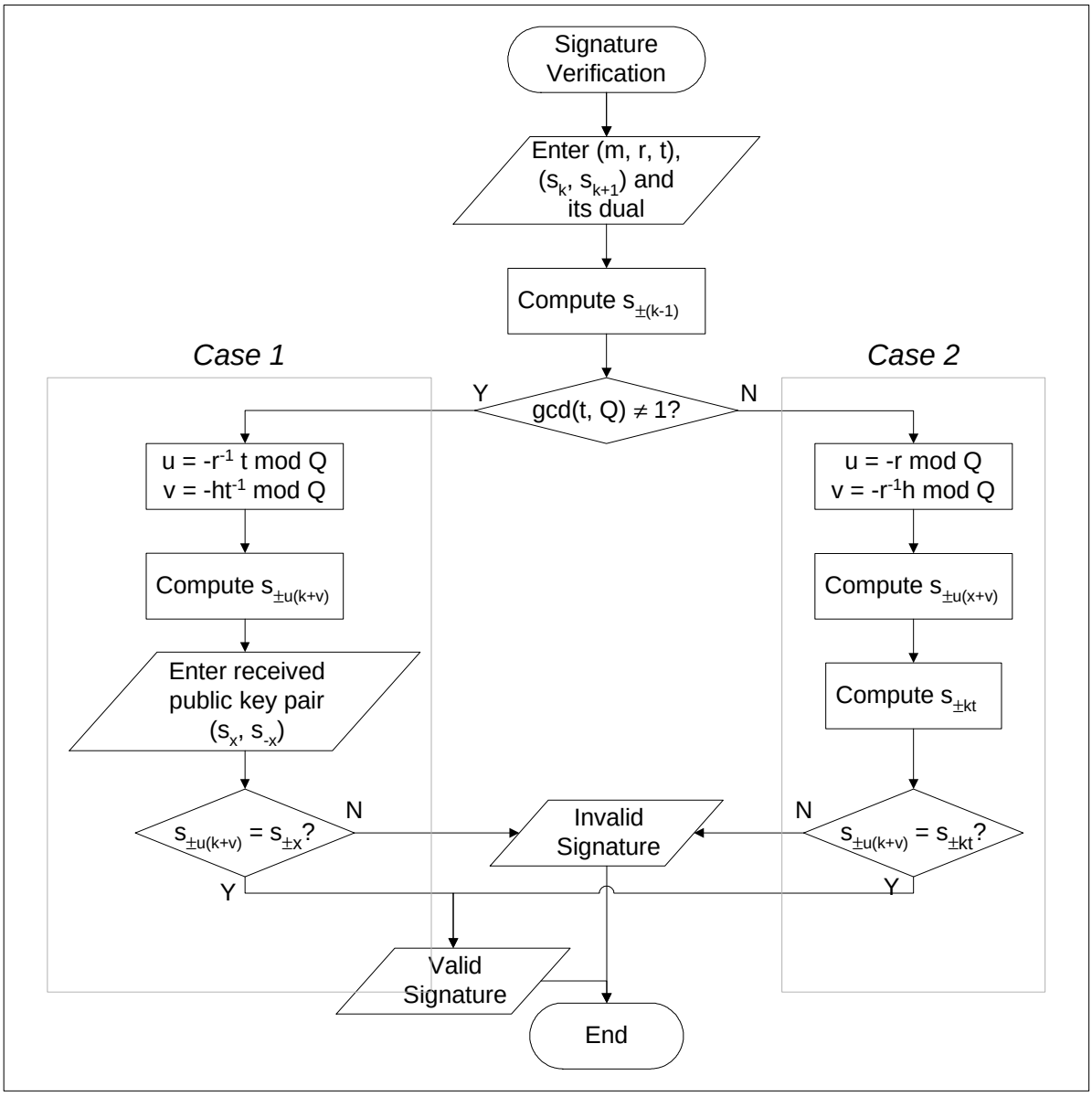

Figure 5: Flowchart Diagram for Signature Verification

5.2. Design Issues

Two design issues arise during the process of software design: the storage requirement for recursive iteration for DSEA algorithm and the percentage of *delta* that equals zero.

5.2.1. The Storage Requirement for Recursive Iterations in DSEA Algorithm

As stated in Section 3.1, since a for-loop from 1 to n is needed to choose the corresponding set of equation to use depending on the value of k_j, it is necessary to consider the storage requirement for this for-loop. In the two sets of equations for k_j equals 0 or 1, the six sequence terms to be computed in each iterations $s_{t'+1}$, $s_{t'}$, $s_{t'-1,}$ $s_{-(t'+1)}$, $s_{-t'}$ and $s_{-(t'-1)}$ require the use of s_{t+1}, s_t, $s_{t-1,}$ $s_{-(t+1)}$, s_{-t} and $s_{-(t-1),}$ which are the six sequence terms generated in the previous iteration. Therefore, after the sequences terms $s_{t'+1}$, $s_{t'}$, $s_{t'-1,}$ $s_{-(t'+1)}$, $s_{-t'}$ and $s_{-(t'-1)}$ are computed, the terms s_{t+1}, s_t, $s_{t-1,}$ $s_{-(t+1)}$, s_{-t} and $s_{-(t-1)}$ can be discarded as they will not be used in future iterations. Hence, instead of storing the whole sequence, only six values need to be kept after each iteration.

5.2.2. The Percentage of *Delta* that Equals Zero

As described in Section 5.1.2, Alice must choose a value for k such that *delta* is a non-zero number. This is a necessary condition for Bob to compute the s_{k-1} and its dual using (s_k, s_{k+1}) and its dual. Therefore, it is important to investigate the percentage of delta that equals zero to see the tradeoff between bandwidth usage and time required to choose a valid k.

Programs written in C++ and Maple are used to compute the value of *delta* for all sequence terms generated by all irreducible polynomial over GF(p) for a given p. When *delta* equals zero, it means that:

$$s_{k+1}s_{-(k+1)} = s_1 s_{-1} = ab$$

If sequence terms are organized in two rows as shown in Figure 6, the product $s_{k+1}s_{-(k+1)}$ for a given k is the product between sequence terms in the same column. For example, if k equals to 2, the product $s_{k+1}s_{-(k+1)}$ is:

$$s_3 s_{-3} = s_3 s_{Q-3} = s_{(Q-3)} s_{-(Q-3)}$$

which is the same product term for *k* equals to *Q* - 4. Therefore, it is sufficient to compute *delta* for *k* from 1 to (*Q* - 1) / 2 only.

Prime numbers, *p*, between 5 and 127 are used in the testing and the results are analyzed. A sample result for *p* equals 17 is provided in Appendix C. It is found out that for any prime *p*,

$$s_p = a = s_{-(p+1)}$$
$$s_{p+1} = b = s_{-p}$$

In other words, if *k* equals to either *p-1* or *p*, the product $s_{k+1}s_{-(k+1)}$ equals to *a*b* for any prime *p*. This restriction is added to the signing key selection.

Besides, if either *a* or *b* is zero, *k* will be more likely to yield a zero delta. This is due to the fact that *a*b* equals zero if either *a* or *b* is zero. A zero *delta* will be resulted if either s_{k+1} or $s_{-(k+1)}$ equals zero. To avoid this, system parameters should be chosen such that both *a* and *b* are non-zero integers.

Moreover, from the result, as *p* increases the percentage of *delta* that equals zero decreases. The percentage is low enough to conclude that the cost of re-selecting value for *k* is lower than the cost associated with increasing the bandwidth usage.

With the additional restrictions on system parameters and the additional rules on selecting *k*, it is reasonable to reduce bandwidth usage as a way of minimizing the cost.

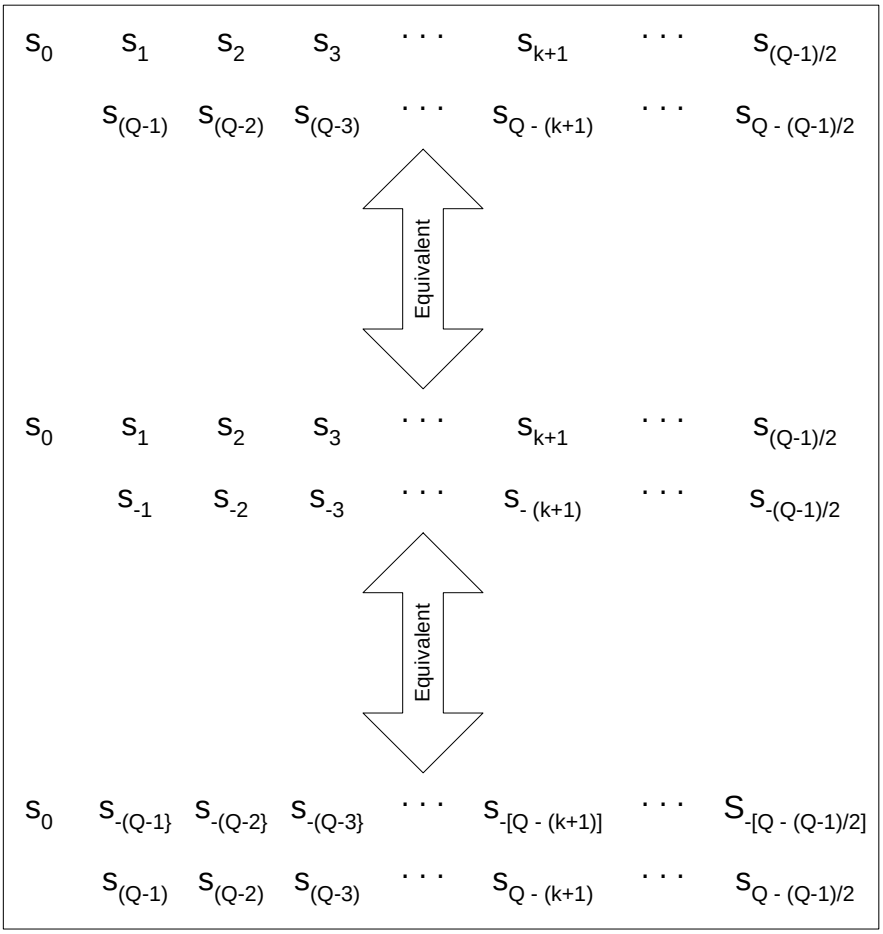

Figure 6: Organizing Sequence Terms

5.3. Testing

Two cases, toy case and real system case, are used to verify the coding. The testing of these two cases is described below.

5.3.1. Toy Case

In the initial phase of testing, an irreducible polynomial *f(x)* over a small field is used. Working in small field makes it possible to verify all the sequence term computations including the intermediate ones.

In order to verify that all computed sequence terms are correct, a program is written to generate all the terms in one period of the characteristic sequence as shown in Appendix B. Sequence terms are computed using a recursive formula:

$$s_{j+3} = as_{j+2} - bs_{j+1} + s_j, \; for \; i = 0,1,2,\ldots$$

derived from *f(x)*. This program can display any $\pm k^{th}$ terms in the sequence.

An irreducible polynomial in GF(5) is chosen for the initial phase of testing:

$$f(x) = x^3 + x - 1$$

The maximum period of the characteristic sequence generated by any irreducible polynomial *f(x)* over GF(5) is:

$$Q = 5^2 + 5 + 1 = 31$$

All computed sequence terms, such as public key pairs, shared key pairs and *r* in the signing equation are compared to the sequence terms generated by the recursive formula to ensure there is no mismatch.

5.3.2. Real System Case

Since the security of the GH-PKC is based on the difficulty in solving the discrete logarithm problem in GF(p^3), to implement a GH-PKC with 1024-bit security, a 341-bit prime p is required. System parameters are chosen as:

$p = 252410014280206509131998647534662043944278252812238164081281638438436419$
$589262881844002472940759520929 1$

$a = 100967846246663453437323616599547897779132286415320714933049077620914827 9$
$733077179938397109115148708951$

$b = 206216022644184759815024549954227848108708723659854548174088293500293906 2$
$370689540637392192938836162683$

To test this "Real System", it is not practical to verify all the generated sequence terms as there are too many terms in one period. As all the sequence terms were verified in the "Toy Case", there is no need to verify these terms again here. Instead, to see if the coding is correctly implemented, two copies of the program are executed at the same time to simulate two user Alice and Bob, namely Program A and Program B respectively. By feeding random private keys K_A and K_B to Program A and Program B respectively, the programs should return their public key pairs. After the public key pairs have been generated, the public key pair from Program B is entered into Program A to simulate Alice receiving the public key from Bob. Similarly, copying the public key pair from Program A to Program B simulates Bob receiving the public key from Alice. Then each program outputs a common key pair that is the common key pair shared between Alice and Bob, and the output key pairs in the two programs are verified to be the same. Similar testing would be done to the signature scheme.

6.0 CURRENT RESULTS AND REMARKS

The GH-DH key agreement protocol and the signature generation in the GH-DSA scheme are successfully implemented in C++ and are tested thoroughly. The implementation of signature verification is still in progress.

Several restrictions on system parameters and signing key selections are added to the system to increase system efficiency. System parameters *a* and *b* should be chosen such that determinant of state matrix M_0 does not equal to zero. Besides, both *a* and *b* should be non-zero integers to minimize the probability of *delta* equals zero. The signing key, *k*, cannot equal to either *p-1* or *p* as these values will cause delta to be zero.

7.0 REFERENCES

[1] W. Diffie and M. E. Hellman, “New directions in cryptography”, IEEE Trans on Information Theory vol IT-22, no. 6, pp644-654, Nov 1976

[2] NIST, FIPS PUB 186-2, Digital Signature Standard (DSS), Jan 2000, available at http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf

[3] G. Gong and L. Harn, “Public-key cryptosystems based on cubic finite field extensions”, IEEE IT vol 45, no 7, pp 2061-2065, Nov. 1999

[4] G. Gong, L. Harn, and H. Wu, “The GH public-key cryptosystem”, in the Preceedings of Selected Areas in Cryptography-SAC 2001, Toronto, ON, Canada, Aug 2001, pp 301-316.

[5] S.W. Golomb, *Shift Register Sequences*, California: Aegean Park Press, 1982, pp 34-35.

[6] V. Shoup, *Number Theory Library (NTL) Version 5.3*, available at http://www.shoup.net/ntl

APPENDIX A CODE LSTING FOR GH-PKC

```
/*************************************************************************
Susana Sin
Gong-Harn Public-key Cryptosystem
*************************************************************************/
#include <NTL/vec_ZZ_p.h>
#include <stdlib.h>
#include <math.h>

void k_equal_0(ZZ_p a, ZZ_p b, vec_ZZ_p& s){
   vec_ZZ_p st_prime;
   st_prime.SetLength(6);
   int run = 0;
   while (run < 2) {
      st_prime[0+run*3] = s[1+run*3]*s[0+run*3] - b*s[(4+run*3)%6] +
s[(5+run*3)%6];
      st_prime[1+run*3] = s[1+run*3]*s[1+run*3] -
to_ZZ_p(to_ZZ("2"))*s[(4+run*3)%6];
      st_prime[2+run*3] = s[1+run*3]*s[2+run*3] - a*s[(4+run*3)%6] +
s[(3+run*3)%6];
      run++;
      ZZ_p temp = a;
      a = b;
      b = temp;
   }
   for (int z = 0; z < 6; z++) s[z] = st_prime[z];
}

void k_equal_1(ZZ_p a, ZZ_p b, vec_ZZ_p& s){
   vec_ZZ_p st_prime;
   st_prime.SetLength(6);
   int run = 0;
   while (run < 2) {
      st_prime[0+run*3] = s[1+run*3]*s[1+run*3] -
to_ZZ_p(to_ZZ("2"))*s[(4+run*3)%6];
      st_prime[1+run*3] = s[1+run*3]*s[2+run*3] - a*s[(4+run*3)%6] +
s[(3+run*3)%6];
      st_prime[2+run*3] = s[2+run*3]*s[2+run*3] -
to_ZZ_p(to_ZZ("2"))*s[(5+run*3)%6];
      run++;
      ZZ_p temp = a;
      a = b;
      b = temp;
   }
   for (int z = 0; z < 6; z++) s[z] = st_prime[z];
}

void Initial_State(ZZ_p a, ZZ_p b, vec_ZZ_p& s) {
   s[0] = to_ZZ_p(to_ZZ("3"));
   s[1] = a;
   s[2] = a * a - to_ZZ_p(to_ZZ("2")) * b;
   s[3] = to_ZZ_p(to_ZZ("3"));
   s[4] = b;
   s[5] = b * b - to_ZZ_p(to_ZZ("2")) * a;
}

void Seq_term(ZZ_p a, ZZ_p b, vec_ZZ_p& s, ZZ K) {
ZZ K1,num;
long int r = 0;
int k[r+1];

    /****** Binary Representation of K ******/
```

```
    K1 = K;
    while (K > power_ZZ(2,r)) r++;
    if (K!= power_ZZ(2,r)) r--;
    k[0] = 1; // Most Significant Bit
    K1 = K1 - power_ZZ(2,r);
    for (int i = 1; i <= r; i++) {
       num = power_ZZ(2,(r-i));
       if(K1 >= num) {
          k[i] = 1;
          K1 = K1 - num;
       }
       else k[i] = 0;
    }

   /***************** DSEA *****************/
   for (long j = 1; j <= r; j++) {
      if (k[j] == 0) k_equal_0(a, b, s);
      else k_equal_1(a, b, s);
   }

}

void GH_DH(ZZ_p a, ZZ_p b, ZZ Q){
ZZ X;
vec_ZZ_p s;
s.SetLength(6);
ZZ gcd;
bool pri_key = 1;
bool cont = 1;

   while(cont) {
      if (pri_key) { // A public key pair computation
         cout << "Select your private key x, 0 < x < " << Q << ": ";
         cin >> X;
         gcd = GCD(X, Q);
         while ((X==0) || (X > Q) || (gcd != to_ZZ("1")))  {
            cout << "Invalid entry.  Re-enter value: ";
            cin >> X;
            gcd = GCD(X, Q);
         }
      }
      else { // A shared key pair computation
         cout << "Enter the received key pair:" << endl;
         cout << "sx = ";
         cin >> a; // Form fA(x) = x^3 - sx * x^2 + s-x * x - 1
         cout << "s-x = ";
         cin >> b;
      }

      Initial_State(a, b, s);
      Seq_term(a, b, s, X);

      if (pri_key) {// Output public key pair
         cout << "Your public key pair is (" << s[1] << "," << s[4] << ")" <<
endl << endl;
         pri_key = 0;
      }
      else {// Output shared key pair
         cout << "Your shared key pair is (" << s[1] << "," << s[4] << ")" <<
endl << endl;
         cout << "Any more computation? (Input 1 = Yes, 0 = No) ";
         cin >> cont;
      }
   }
```

```
   cout << endl;
}


void Signature(ZZ_p a, ZZ_p b, ZZ Q){
ZZ H, t, X, K;
vec_ZZ_p s;
s.SetLength(6);
ZZ gcd;
bool delta_r_flag;
bool rt_flag;


   cout << "Select your private key x, 0 < x < " << Q << ": ";
   cin >> X;
   gcd = GCD(X, Q);
   while ((X==0) || (X > Q) || (gcd != to_ZZ("1")))  {
      cout << "Invalid entry.  Re-enter value: ";
      cin >> X;
      gcd = GCD(X, Q);
   }

   Initial_State(a, b, s);
   Seq_term(a, b, s, X);
   cout << "Your public key pair is (" << s[1] << "," << s[4] << ")" << endl
<< endl;

   cout << "Enter a message, m, to be signed: ";
   cin >> H;
   cout << endl;


   rt_flag = 1;
   while (rt_flag) { // Choose k such that both r and t !=0
      delta_r_flag = 1;
      while (delta_r_flag) { // Choose k such that delta != 0 and gcd(r,Q) = 1
         cout << "Select signing key k, 0 < k < " << Q << ": ";
         cin >> K;
         gcd = GCD(K, Q);
         while ((K==0) || (K > Q) || (gcd != to_ZZ("1")))  {
            cout << "Invalid entry.  Re-enter value: ";
            cin >> K;
            gcd = GCD(K, Q);
         }

         Initial_State(a, b, s);
         Seq_term(a, b, s, K);
         gcd = GCD(rep(s[1]), Q);
         if (gcd != to_ZZ("1")) cout << "r is not co-prime with Q.  Choose
another k." << endl;
         else {
            if ((s[2] * s[5] - a * b) != to_ZZ_p(to_ZZ("0"))) delta_r_flag =
0; // valid delta and r
            else cout << "Delta is Zero for the chosen k.  Choose another k."
<< endl;
         }
      }

      /********** Compute Signature **********/
      InvMod(t,K,Q);
      t = ((t * (H - rep(s[1]) * X) % Q) + Q) % Q;
      if (s[1] != to_ZZ_p(to_ZZ("0")) && t != to_ZZ("0")) {
         rt_flag = 0;
         cout << endl << "The signed message is (";
```

```
         cout << H << "," << s[1] << "," << t << ")" << endl;
         cout << "(sk, sk+1) is its dual are (" << s[1] << "," << s[2] << ")";
         cout << " and (" << s[4] << "," << s[5] << ") respectively" << endl
<< endl;
      }
      else cout << "Either one of the resultant signature parameter is Zero.
Pick another k" << endl;
   }
   cout << endl;
}

void Verify(ZZ_p a, ZZ_p b, ZZ Q){
   Working in progress.
}

int main()
{

/*
// 341-bit p for 1024-bit security
ZZ p =
to_ZZ("252410014280206509131998647534662043944278252812238164081281638438436411
9589262881844002472940759520929 1");
ZZ a =
to_ZZ("100967846246663453437323616599547897779132286415320714933049077620914827
9733077179938397109115148708951");
ZZ b =
to_ZZ("206216022644184759815024549954227848108708723659854548174088293500293900
6237068954063739219293883616 2683");
ZZ Q = to_ZZ("164705219395020291376758884936962412458513495611 1");
*/

// Toy Case
ZZ p = to_ZZ("5");
ZZ_p::init(p);
ZZ_p a = to_ZZ_p(to_ZZ("0"));
ZZ_p b = to_ZZ_p(to_ZZ("1"));
ZZ Q = to_ZZ("31");

int option;
int cont = 1;

   while (cont){
      cout << "Choose from the followings:" << endl;
      cout << "1. Compute shared key pair" << endl;
      cout << "2. Sign a hashed value of a message" << endl;
      cout << "3. Verify a signature" << endl;
      cout << "4. Terminate program" << endl;
      cout << "Enter the number of operation you would like to perform: ";
      cin >> option;
      cout << endl;
      while ((option != 1) && (option !=2) && (option !=3) && (option !=4)) {
         cout << "Invalid entry.  Please re-enter value: ";
         cin >> option;
      }
      if (option == 4) cont = 0;
      else {
         if (option == 1) GH_DH(a, b, Q); // GH-DH key agreement
         else if (option == 2) Signature(a, b, Q); // Signing
         else Verify(a, b, Q); // Verifying
      }
   }
   return 0;
}
```

APPENDIX B CODE LISTING FOR COMPUTING SEQUENCE TERM

```
/*************************************************************************
Susana Sin
This program gives one perod of a third-order characteristic sequence and
returns (sk, s-k) for any k.
*************************************************************************/
#include<iostream.h>

int LFSR(int s[],int n, int p, int g[]) {
   bool flag = 0;
   int i = 0;

   while (flag == 0) {
      s[i + n] = (s[i] * g[0] + s[i + 1] * g[1] + s[i + 2] * g[2]) % p;
      if (s[i + n] < 0) s[i + n] = s[i + n] + p;
      if ((i > 3) && (s[i +n- 2] == s[0]) && (s[i +n- 1] == s[1]) && (s[i+n]
== s[2])) flag = 1;
      i++;
   }
return i;
}

void main(){
int k;
const int n = 3;
const int p = 5;
int a = 0, b = 1;

//const int p = 53;
//int a = 45, b = 37;

int g[n] = {1, -b, a};
int q = p * p + p + 1;
int s[q + n];
int per = 0;

   /****Initial State****/
   s[0] = 3;
   s[1] = a;
   s[2] = (a * a - 2 * b) % p;
   if (s[2] < 0) s[2] = s[2] + p;

   per = LFSR(s, n, p, g);
   cout << "Period is " << per << endl;

   cout << "k = ? <0 to stop> ";
   cin >> k;
   while (k > 0) {
      if (k > per) cout << "Invalid k" << endl << endl;
      else cout << "(s" << k << ", s" << -k<< ") = (" << s[k] << ", " <<
s[per-k] << ")" << endl << endl;;
      cout << "k = ? <0 to stop> ";
      cin >> k;
   }
}
```

APPENDIX C SAMPLE OUTPUT FOR ZERO *DELTA* TESTING

```
p = 17

(a, b)        Seq Period    k values
-------------------------------------------------------------
( 1, 5)       307           16 17
( 8, 6)       307           5 16 17 101 107
( 6,11)       307           16 17
( 1, 3)       307           16 17
(13,10)       307           16 17
(14, 7)       307           16 17
( 1,11)       307           16 17
(12, 7)       307           16 17 45 92 138
( 4,14)       307           16 17
(13, 6)       307           16 17
( 5, 6)       307           16 17 19 32 52
(12, 4)       307           2 16 17 50 53
( 0, 3)       307           5 13 14 16 17 36 42 51 54 61 68 101 107 111 116
132 146
(13, 0)       307           6 7 12 16 17 20 29 49 70 72 73 85 103 118 125 135
143
( 8, 1)       307           16 17
(11, 8)       307           16 17
( 5, 1)       307           16 17
( 4,10)       307           16 17
( 5, 9)       307           16 17
(13, 9)       307           16 17
( 9,12)       307           16 17
( 0, 9)       307           7 16 17 21 22 24 66 79 83 88 94 106 117 131 135
142 143
( 2, 8)       307           16 17
(11, 9)       307           16 17
( 1,12)       307           16 17 38 48 87
(16, 8)       307           16 17
( 5, 3)       307           16 17
(11, 2)       307           16 17
( 2,15)       307           16 17 45 92 138
( 3, 8)       307           11 16 17 90 102
( 8,11)       307           16 17
( 6, 8)       307           5 16 17 101 107
( 3,13)       307           16 17
(13, 7)       307           16 17
( 1,13)       307           16 17
( 7, 3)       307           16 17
(15, 9)       307           16 17
(15, 4)       307           16 17
(15,14)       307           16 17
( 6, 7)       307           16 17
(14, 8)       307           16 17
(16,13)       307           16 17
( 1, 8)       307           16 17
(10, 4)       307           16 17
(11, 6)       307           16 17
( 2,16)       307           16 17
( 9,14)       307           16 17
(15,16)       307           16 17
( 6,14)       307           16 17 19 32 52
( 8, 0)       307           12 16 17 39 40 60 65 72 78 82 85 105 112 114 115
123 129
(10, 2)       307           16 17
(10, 7)       307           16 17
( 0,13)       307           6 7 12 16 17 20 29 49 70 72 73 85 103 118 125 135
143
```

```
( 8, 3)       307          11 16 17 90 102
( 9, 5)       307          16 17
( 3, 1)       307          16 17
( 2,12)       307          16 17 26 127 151
( 0,10)       307          2 16 17 28 41 43 44 50 53 91 99 110 120 128 133
141 150
( 0,11)       307          9 16 17 24 29 61 73 78 103 111 112 114 117 126 132
136 142
( 2, 6)       307          16 17 62 93 149
( 3, 0)       307          5 13 14 16 17 36 42 51 54 61 68 101 107 111 116
132 146
(15, 2)       307          16 17 45 92 138
(13, 3)       307          16 17
( 9,13)       307          16 17
(10,13)       307          16 17
( 9, 1)       307          16 17
( 4,12)       307          2 16 17 50 53
( 2,11)       307          16 17
(13,16)       307          16 17
( 7,13)       307          16 17
(12, 9)       307          16 17
( 7,14)       307          16 17
( 6, 5)       307          16 17 19 32 52
( 3, 5)       307          16 17
( 8,14)       307          16 17
(16, 2)       307          16 17
(13, 1)       307          16 17
( 9, 0)       307          7 16 17 21 22 24 66 79 83 88 94 106 117 131 135
142 143
(11, 1)       307          16 17
( 6,13)       307          16 17
( 8,16)       307          16 17
( 7, 6)       307          16 17
( 7,10)       307          16 17
(14, 9)       307          16 17
( 3, 7)       307          16 17
( 8, 2)       307          16 17
( 7,12)       307          16 17 45 92 138
(14, 4)       307          16 17
(12, 1)       307          16 17 38 48 87
(14,15)       307          16 17
( 2,10)       307          16 17
(12, 2)       307          16 17 26 127 151
(16,15)       307          16 17
( 9,15)       307          16 17
( 9,11)       307          16 17
( 4,15)       307          16 17
( 0, 8)       307          12 16 17 39 40 60 65 72 78 82 85 105 112 114 115
123 129
( 6, 2)       307          16 17 62 93 149
(10, 0)       307          2 16 17 28 41 43 44 50 53 91 99 110 120 128 133
141 150
(14, 6)       307          16 17 19 32 52
(11, 0)       307          9 16 17 24 29 61 73 78 103 111 112 114 117 126 132
136 142
( 1, 9)       307          16 17
```