Matrix-Based RSA Encryption of Streaming Data

Dr. Michael Prendergast
Michael.David.Prendergast@gmail.com
August 4, 2021

Abstract

This paper describes a new method for performing secure encryption of blocks of streaming data. This algorithm is an extension of the RSA encryption algorithm. Instead of using a public key (e, n) where n is the product of two large primes and e is relatively prime to the Euler Totient function, $\varphi(n)$, one uses a public key (n, m, E), where m is the rank of the matrix E and E is an invertible matrix in $GL(m, \varphi(n))$. When m is 1, this last condition is equivalent to saying that E is relatively prime to $\varphi(n)$, which is a requirement for standard RSA encryption. Rather than a secret private key $(d,\varphi(n))$ where d is the inverse of e $(mod\ \varphi(n))$, the private key is $(D,\varphi(n))$, where D is the inverse of E $(mod\ (\varphi(n))$. The key to making this generalization work is a matrix generalization of the scalar exponentiation operator that maps Z_n^m, the set of m-dimensional vectors with integer coefficients modulo n, onto itself.

1. Background

In public key encryption systems, a transmitter and receiver share a public key and the receiver possesses a secret key known only to himself. The transmitter encrypts and sends a secret message using the public key. Without expending considerable time and resources, only the receiver with the private key is capable of decrypting the secret message.

The RSA algorithm, named after its inventors, Rivest-Shamir-Adleman, is a public key encryption system described in [**Rivest, 1983**]. In its simplest form, keys are generated as follows:

Step 1: Choose two large primes, p and q and compute $n = p * q$.

Step 2: Compute $\varphi(n) = (p - 1) * (q - 1)$.

Step 3: Find an exponent e such that $gcd(\varphi(n), e) = 1$.

Step 4: Compute $d = e^{-1}\ mod(\varphi(n))$.

Step 5: Publish (n, e) as the public key, and keep $(d, \varphi(n))$ as the secret private key.

To encrypt, the sender first maps the message to a value M between 1 and $n - 1$. M is then encrypted according to the formula

$$C = M^e (mod\ n). \tag{1}$$

The receiver decrypts the message by computing

$$(M^e)^d (mod\ n) = M^{ed}\ (mod\ n) = M\ (mod\ n) \tag{2}$$

by Fermat's little theorem, since $ed = 1\ mod(\varphi(n))$.

The RSA algorithm is not used on plaintext messages of arbitrary length for several reasons.

First, note from above that each data block must be mapped to a value M between 1 and $n - 1$. This prevents outright encryption of an entire message unless the message is small.

However, breaking the message into smaller data blocks creates other problems. We see from (1) that repeated encryption of the same text with the same encryption key always yields the same result. This can make the technique vulnerable to both frequency analysis attacks and chosen ciphertext attacks.

To see an example of a frequency analysis attack, suppose that a long message is encrypted using standard RSA encryption, and that each letter in the message is individually encrypted using the same RSA encryption key. Since each encryption of the letter "e" is the same, one might be able to discern which encryption represented the letter "e" from a frequency analysis.

To illustrate the RSA vulnerability to a chosen ciphertext attack, suppose that an attacker knew a value k which was easily invertible $(mod\ n)$. Suppose also that he did not know the decryption of an encrypted message $(M)^e\ (mod\ n)$, but he was given access to the decryption of $k(M)^e\ (mod\ n)$ for some scalar k. The decryption is given by

$$(k(M)^e)^d = k^d M^{ed} = k^d M = k^{-e} M(mod\ n).$$

But since k is easily invertible and e is known, k^{-e} can also be computed and hence M can be discovered.

Therefore, repeated application of the standard RSA algorithm with the same encryption key is not recommended for decrypting streaming data. Instead, a more secure approach is to use the RSA encryption to agree on a block hash. This hash is then used in a block encryption algorithm (such as AES, see [**NIST, 2001**]), where the message is broken into data blocks of fixed length, and each data block is encrypted using a function dependent on the hash and the previous data blocks.

This paper describes a matrix generalization of the RSA encryption algorithm that overcomes these obstacles. This generalization permits encryption of a data block to be dependent on previous data blocks or random nonce, guaranteeing that repeated encryption of the same message does not yield the same ciphertext. This blocks the chosen ciphertext and frequency analysis attack vulnerabilities described above.

Previously proposed matrix-based public key encryption systems (such as [**Backal, 2001**], [**Chuang, 1991**], [**Singh, 2004**] and [**Slavin, 2008**]) have mapped a message into one or more matrices, and then performed encryption on these matrices. For example, [**Chuang, 1991**] encrypts by mapping the message into the off-diagonal elements of a triangular matrix, and applies the Cayley-Hamilton theorem to perform scalar exponentiation of the matrix.

This approach is different, because it generalizes the RSA algorithm to operate on entire matrices, instead of applying standard RSA to operate on scalar data that resides within a matrix.

For example, the key generation process is modified to choose an invertible matrix E in the general linear group of mxm invertible matrices ($mod\ \Phi(n)$) instead of a scalar e. In step 4, the matrix inverse D is computed instead of a scalar d.

With appropriately defined vector exponentiation, equations (1) and (2) continue to hold. These changes allow for confusion across encrypted data blocks, and allow RSA encryption on streaming data.

1.1. Mathematical Notation

Throughout this paper, we use $\Phi(n)$ to represent the number of positive integers not exceeding the integer n and relatively prime to n. This is commonly known as Euler's Totient function.

We also use the expression $GL(m, s)$ to represent the general linear group of invertible $m\ x\ m$ matrices with coefficients in the field of integers *modulo s*.

For integer *x* and *z*, the expression $x(mod\ z)$ denotes the integral remainder when *x* is divided by *z*. When the expression preceding $(mod\ z)$ is a matrix, the integral remainder is computed for each component of the matrix. Hence

$$\begin{pmatrix} 13 & 27 \\ 41 & -4 \end{pmatrix} = \begin{pmatrix} 3 & 7 \\ 1 & 6 \end{pmatrix} (mod\ 10).$$

Now we define a generalization of scalar exponentiation that can be applied to vectors of length $m > 1$. Let $Z_s$be the ring of integers modulo *s*, and let

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{m,1} \\ a_{2,1} & a_{2,2} & & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & & a_{m,m} \end{bmatrix}$$

be an mxm matrix with coefficients in Z_s.

Suppose that

$$X = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

is an m-dimensional vector with non-zero components which are also in Z_s.

We denote the transpose of X by $X^T = (x_1, x_2, \dots, x_m)$ and note that $(X^T)^T = X$.

Next, define a new operator - the vector exponentiation operator "^^". This operator maps the set of m-dimensional vectors with non-zero components over Z_s to itself, by:

$$X\text{^^}A = \begin{pmatrix} \prod_{j=1}^{m} {x_j}^{a_{1,j}} \\ \prod_{j=1}^{m} {x_j}^{a_{2,j}} \\ \vdots \\ \prod_{j=1}^{m} {x_j}^{a_{m,j}} \end{pmatrix}, \qquad (3)$$

Note that (3) confuses the inputs; each output y_i depends on all inputs x_j for which $a_{i,j} \neq 0$. Two additional important properties of the definition (3) are:

$$X\text{^^}I = X \qquad (4)$$

and

$$((X\text{^^}A)\text{^^}B) \;=\; (X\text{^^}(B * A)) \qquad (5).$$

Equation (4) follows directly from (3). For (5), note that the i-th component of $(X\text{^^}A)\text{^^}B$ is given by

$$z_i = \prod_{k=1}^{m} \left(\prod_{j=1}^{m} {x_j}^{a_{k,j}}\right)^{b_{i,k}} = \prod_{k=1}^{m} \left(\prod_{j=1}^{m} {x_j}^{a_{k,j}b_{i,k}}\right)$$

$$= \prod_{j=1}^{m} {x_j}^{\sum_{k=1}^{m} a_{k,j}b_{i,k}} \qquad (6).$$

But the (i, j) component of BA is given by

$$(BA)_{i,j} = \sum_{k=1}^{m} b_{i,k}a_{k,j} \qquad (7).$$

Hence, combining (6) and (7),

$$z_i = \prod_{j=1}^{m} {x_j}^{\sum_{k=1}^{m} a_{k,j}b_{i,k}} = \prod_{j=1}^{m} {x_j}^{(BA)_{i,j}} \quad ,$$

which proves (5).

1.2. Paper Overview

Section 2 provides a high-level overview of how matrix-based RSA encryption works, and illustrates encryption and decryption of a simple message using this approach.

A detailed description of the algorithm is provided in section 3. Section 3.1 presents several alternatives for key generation, including randomized selection of matrix components, similarity transformation on diagonal matrices, and LU compositions with randomized off-diagonal elements. Streaming data encryption approaches are outlined in section 3.2, and include encryption with previously encrypted data blocks, encryption with random nonce, and combinations of both previous data blocks and random nonce. Section 3.3 describes the decryption process steps.

Security and performance notes are found in section 4, and the author's summary and conclusions are found in section 5. References cited in this paper are listed in section 6.

2. Matrix-Based RSA Encryption Overview

This section provides a high-level overview of the matrix-based RSA encryption approach, and illustrates the approach with an example.

2.1. Matrix-Based RSA Encryption Approach

A summary of the approach for encrypting a stream of digital block data is as follows:

1. (Key Generation) Select two large primes, p and q, and compute $n = p * q$.

2. (Key Generation) Select an invertible matrix E in $GL(m, \varphi(n))$. Find its inverse $D = E^{-1}\ (mod\ \varphi(n))$. (n, m, E) is the public key. $(D, \varphi(n))$ is the private key.
3. (Mapping) Map each block of digital data in a message to a value between 1 and $n - 1$.
4. (Encryption) For each new block of data B_k, encrypt B_k together with $(m - 1)$ other blocks $(S_1, S_2, \dots, S_{m-1})$ using the encryption matrix E by calculating

$$(E_1, E_2, \dots, E_{m-1}, E_m)^T = [(S_1, S_2, \dots, S_{m-1}, B_k)^T \text{^^} E]\ (mod\ n).$$

   Transmit all m encrypted data blocks to the receiver. (The S_1 can be generated from prior data blocks, prior data block encryptions, or random nonce.)
5. (Decryption). For each set of m encrypted blocks of data received from the sender, decrypt them using the decryption matrix D by computing

$$(S_1, S_2, \dots, S_{m-1}, B_k)^T = [(E_1, E_2, \dots, E_{m-1}, E_m) \text{^^} D]^T (mod\ n).$$

2.2. Matrix-Based RSA Encryption Example

In this section we give a very simple example of the techniques described in this section, including key generation, encryption and decryption.

Let $p = 11$ and $q = 17$ so that $n = p * q = 187$ and $\varphi(n) = 10 * 16 = 160$.

We will encrypt two blocks at a time, so set $m = 2$.

Our secret message will contain only letters, so we map A to 1, B to 2, C to 3, and so on. The message we will choose to decrypt will be "HI", which is mapped to the numbers 8 and 9.

We will compute *E* using a similarity transformation on a diagonal matrix Λ,

$$\Lambda = \begin{pmatrix} 3 & 0 \\ 0 & 13 \end{pmatrix}.$$

Note that

$$\Lambda^{-1} = \begin{pmatrix} 107 & 0 \\ 0 & 37 \end{pmatrix}\ (mod\ 160).$$

It should also be noted that it is very important to ensure that matrix inversion is computed using the correct modulus. The modulus should be $\varphi(n)$ during key generation but encryption and decryption are performed modulus *n*. Mixing these up will return incorrect results.

We choose the invertible matrix *P* to be

$$P = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} (mod\ 160)$$

so that

$$P^{-1} = \begin{pmatrix} 1 & -1 \\ -1 & 2 \end{pmatrix} (mod\ 160).$$

Then our encryption matrix E is computed as

$$E = P\Lambda P^{-1} = \begin{pmatrix} 153 & 20 \\ 150 & 23 \end{pmatrix} (mod\ 160).$$

The decryption matrix D is also

$$E = P\Lambda^{-1}P^{-1} = \begin{pmatrix} 17 & 20 \\ 70 & 127 \end{pmatrix} (mod\ 160).$$

Our secret message is "HI", which is mapped to the vector

$$X = \begin{pmatrix} 8 \\ 9 \end{pmatrix}.$$

Its encryption is therefore

$$X\text{^^}E = \begin{pmatrix} 8^{153} * 9^{20} \\ 8^{150} * 9^{23} \end{pmatrix} (mod\ 187) = \begin{pmatrix} 94 \\ 25 \end{pmatrix} (mod\ 187).$$

Note that, since we are now doing encryption and decryption, the modulus has switched from $\varphi(n)$=160 to $n = 187$.

The decryption is given by

$$(X\text{^^}E)\text{^^}D = \begin{pmatrix} 94^{17} * 25^{20} \\ 94^{70} * 25^{127} \end{pmatrix} (mod\ 187) = \begin{pmatrix} 8 \\ 9 \end{pmatrix} (mod\ 187),$$

as we expected.

3. Detailed Description of Matrix-Based RSA Algorithm

3.1. Key Generation

The key generation process will construct a public key (n, m, E) and private key $(\varphi(n), D)$, where E and $D = E^{-1}\ (mod\ \varphi(n))$ are matrices in $GL(m, \varphi(n))$. The matrix rank m represents the number of data blocks that are confused and encrypted. To compute n, we select two large primes, p and q, each at least as large as 2^{64}, and set $n = p * q$ and $\varphi(n) = (p-1) * (q-1)$, just as with standard RSA encryption.

The rank of the encryption matrix may be either pre-selected or randomly selected as part of the key generation process. The matrix rank m determines how many data blocks are confused together in a single encryption round. If $m = 1$, then the matrix-based RSA encryption is equivalent to standard RSA encryption as defined in [**Rivest, 1983**], and there is no confusion between data blocks.

There is no theoretical limit to how large m can be, but very large values of m will increase the computation burden for encryption and decryption. Reasonable results can be achieved with m between 4 and 7.

When the matrix E is used to encrypt a sequence of consecutive data blocks together, E determines how data blocks are confused with each other during encryption. A dense matrix results in greater confusion across data blocks. If E is lower triangular, then encryption requires knowledge of only previous data blocks, not subsequent blocks. Tridiagonal matrices confuse only nearest neighbors.

With standard RSA key generation, care must be made in the selection of e so that the inverse d cannot be easily discovered. As recommended in [**Rivest, 1983**], small factors that might divide $\varphi(n)$ should be checked to see if they yield solutions for $e^d = 1\ (mod\ n)$. If a factor is found, then the ciphertext can be decrypted without knowledge of $\varphi(n)$.

The same is true for the selection of E. An attacker can try various small integers s that might solve $E^s = I\ (mod\ n)$. If one is discovered, then E^{s-1} can be used to decrypt the encrypted message by *s*-fold application of exponentiation by *E*, since

$$X\text{^^}E^s = X\text{^^}I\ =\ X(mod\ n).$$

However, with matrix-based encryption, E^s may still reveal information about the original information even if $E^s \neq I\ (mod\ n)$. Each individual component of E^s should be checked to prevent information leakage.

For example, let $n = 11 * 13 = 143$, so $\varphi(n) = 10 * 12 = 120$. If we set

$$E = \begin{pmatrix} 2 & 0 \\ 0 & 142 \end{pmatrix} = \begin{pmatrix} 2 & 0 \\ 0 & -1 \end{pmatrix}$$

E has order 60, since 2 has order 60 in Z_{143}. But

$$E^s = \begin{pmatrix} 2^s & 0 \\ 0 & (-1)^s \end{pmatrix}$$

which leaves the second component of $X\text{^^}(E^s)$ unchanged for even values of s. Therefore, with this choice of E, information is revealed about the plaintext when $s = 2$, even though the order of E is $O(E) = \frac{\varphi(n)}{2} = 60$. Hence this example is not a good choice for E.

3.1.1. Random Key Selection

There are a number of methods for selecting $E = (e_{i,j})$ in $GL(m, \varphi(n))$. For example, the encryptor can choose E through random selection of components and then test to determine whether or not the randomly selected matrix is invertible. There are a number of algorithms available to determine whether a matrix is invertible, including computing the determinate and Gaussian elimination with pivoting to directly compute the inverse, if it exists (see [**Strang, 2009**]). When applying Gaussian elimination to matrices in $GL(m, \varphi(n))$, pivoting around an element $e_{i,j}$ requires application of Euclid's algorithm to determine $d = {e_{i,j}}^{-1}\ (mod\ \varphi(n))$ (see [**LeVeque, 1996**]).

When p and q are chosen so that $\varphi(n)$ has "mostly" large factors, a randomly selected matrix E will be invertible with enough regularity that choosing an E at random and then testing it for invertibility is a reasonable approach to take.

To summarize the random component selection approach:

- Randomly select the components of E, subject to any constraints imposed by the problem (i.e., block diagonal, or lower triangular)
- Test E for invertibility
- Test E^s using small to moderate factors s of $\varphi(n)$ to ensure no data leakage occurs

3.1.2. Key Selection by Similarity Transformation

A better approach, however, is to construct E by starting with an invertible diagonal matrix Λ, and apply a similarity transformation matrix P to it, so that $E = P\Lambda P^{-1}(mod\ \varphi(n))$. The diagonal matrix Λ will be invertible in $GL(m, \varphi(n))$ if and only if each of the diagonal elements $\lambda_{i,i}$ is relatively prime to $\varphi(n)$. This embodiment has the nice property that the powers of E can be easily computed, and $E^{\varphi(n)} = I\ (mod\ n)$, since all of the diagonal elements $\lambda_{i,i}$ satisfy ${\lambda_{i,i}}^{\varphi(n)} = 1\ (mod\ n)$. With this approach, one chooses Λ so that all of the diagonal elements have large order (at least 1000), and then select an invertible P as above.

3.1.3. Key Selection Using LU Decomposition

Another good approach for creating the encryption matrix E is to use LU decomposition. With this approach, P can be the product of randomly generated lower-triangular and upper-triangular matrices, $P = LU$. When combined with similarity transformation, one constructs E as the product $E = (LU)\Lambda(LU)^{-1} mod\ \varphi(n)$, where L is an invertible lower triangular matrix in $GL(m, \varphi(n))$, U is an invertible upper triangular matrix in $GL(m, \varphi(n))$, and Λ is a diagonal matrix with all elements relatively prime to $\varphi(n)$. Since the diagonal elements of Λ can be any number relatively prime to $\varphi(n)$, we can assume without loss of generality that the diagonal elements of L and U are all ones.

A flowchart for a key generation process that uses LU decomposition where $E = (LU)\Lambda(LU)^{-1}$, is shown in Figure 1.

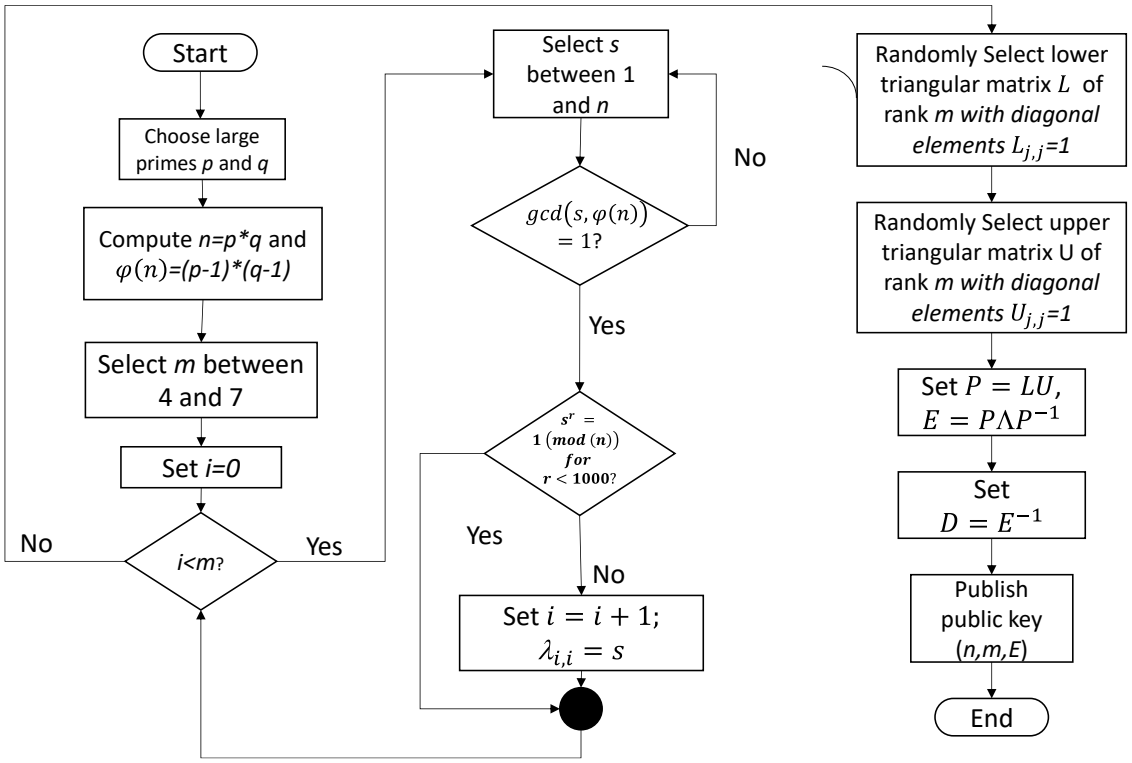

Figure 1: Key Generation Process Using LU-Decomposition

First p, q and the rank m are chosen. We set $n = pq$ and $\varphi(n) = (p-1) * (q-1)$.

Next, we choose the diagonal elements λ_i to be relatively prime to $\varphi(n)$, and create two random lower- and upper-triangular matrices with ones on the diagonal. We set $P = LU\big(mod\ \varphi(n)\big)$ and define the encryption matrix by $E = P\Lambda P^{-1}\ (mod\ \varphi(n))$. Set $D = E^{-1}\ (mod\ \varphi(n))$. We publish the public key (n, m, E) and keep the hidden secret key $(\varphi(n), D)$.

Since Λ is a diagonal matrix in $GL(m, \varphi(n))$, its inverse

$$\Lambda^{-1} = \begin{bmatrix} {d_{1,1}}^{-1} & 0 & \dots & 0 \\ 0 & {d_{2,2}}^{-1} & 0 & \vdots \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & {d_{m,m}}^{-1} \end{bmatrix}$$

exists since each diagonal element of Λ, $d_{i,i}$, is relatively prime to $\varphi(n)$. The diagonal elements can be calculated by solving $a_i * d_{i,i} + b_i * \varphi(n) = 1$ for a_i and b_i using Euclid's algorithm. The solutions a_i satisfy $a_i = {d_{i,i}}^{-1}$, and

$$\Lambda^{-1} = \begin{bmatrix} a_1 & 0 & \dots & 0 \\ 0 & a_2 & 0 & \vdots \\ 0 & 0 & \ddots & 0 \\ 0 & 0 & \dots & a_m \end{bmatrix} .$$

Since L and U are triangular matrices, their inverses can also be readily computed by Gaussian elimination. For example, if

$$L = \begin{bmatrix} 1 & 0 & 0 \\ l_{2,1} & 1 & 0 \\ l_{3,1} & l_{3,2} & 1 \end{bmatrix},$$

then repeated application of Gaussian Elimination to find L^{-1} yields:

$$L^{-1} = \begin{bmatrix} 1 & 0 & 0 \\ -l_{2,1} & 1 & 0 \\ -l_{3,1} + l_{2,1}l_{3,2} & -l_{3,2} & 1 \end{bmatrix},$$

Note that if $m = 1$ and $gcd(E, \varphi(n)) = 1$, this procedure is equivalent to the RSA key generation procedure.

3.2. Encryption Method

During key generation, we saw that the encrypting agent selected two large primes, p and q, and computed the Euler totient function $\varphi(n) = (p-1) * (q-1)$. The encrypting agent then selected an invertible $m\ x\ m$ matrix E $(m \geq 1)$ from $GL(m, \varphi(n))$, the general linear group of invertible $m\ x\ m$ matrices $(mod\ \varphi(n))$, and computed the matrix inverse $D = E^{-1}$. Note that D and E are both matrices with elements in the field of integers $(mod\ \varphi(n))$. The encrypting agent then published (n, m, E) as the public key, and retains $(\varphi(n), D)$ as the secret private key.

Figure 2 shows one way in which these keys can be used to encrypt a fixed sequence of data blocks $(B_1, B_2, \dots)$, Each data block must be of length less than $Log_2(n)$ bits. Using an assignment method, each data block is mapped to an integer between 1 and $n-1$, each relatively prime to n. (Note that 0 is not used in the mapping, because each mapped element must be relatively prime to both p and q.)

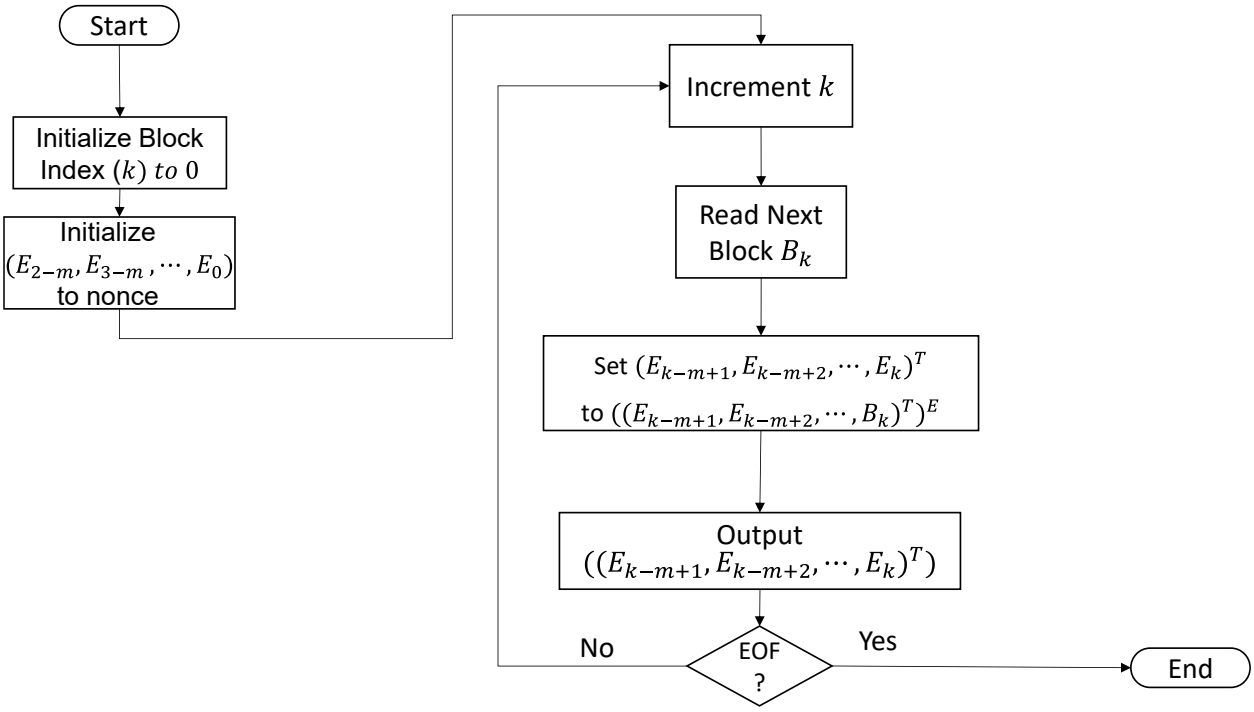

Figure 2: Matrix-RSA Encryption of a Sequence of Data Blocks

An initial set of $m-1$ blocks $(E_{2-m}, E_{3-m}, \ldots, E_0)$ are set to random nonce. Then each new data block B_k is encrypted with the previously $m-1$ data blocks $(E_{i+1}, E_{i+2}, \ldots, E_{i+m-1})$. The first block is therefore encrypted with random nonce. Subsequent iterations, however, will encrypt a combination of the current data block, previous encryptions and the initial random nonce together.

In a computer program, this is performed in two steps ((8) and (9)). First the encrypted data are encrypted and confused:

$$(F_{i+1}, F_{i+2}, \ldots, F_{i+m})^T = ((E_{i+1}, E_{i+2}, \ldots, E_{i+m-1}, B_{i+m})^T)\text{^^}E (mod\ n) \quad (8)$$

Then the encryption results are stored for the next block:

$$(E_{i+1}, E_{i+2}, \ldots, E_{i+m})^T = (F_{i+1}, F_{i+2}, \ldots, F_{i+m})^T \quad (9)$$

Two points should be noted with this approach. First, changing one data block in a message stream will in general change all of the ciphertext that follows that data block. This is because if a bit in data block B_j changes, then the encryption E_j changes, which causes all subsequent encryptions to change, due to (8). Second, two encryptions of the same data blocks $(B_1, B_2, \ldots)$ will have different cyphertexts, because they start with different random nonces.

Other approaches are also possible. If storage is not an issue, each data block B_{i+m} could be uniquely encrypted with $m-1$ data blocks of random nonce $(R_{i+1}, R_{i+2}, \ldots, R_{i+m-1})$ according to the mapping

$$(E_{i+1}, E_{i+2}, \ldots, E_{i+m})^T = ((R_{i+1}, R_{i+2}, \ldots, R_{i+m-1}, B_{i+m})^T)\text{^^}E\ (mod\ n)$$

Here only the last data block is of interest, the others are just noise.

Matrix-based encryptions that include combinations of random nonce and previously encrypted blocks together with the current block are also possible,

3.3. Decryption Method

Decrypting a sequence of data blocks $(B_1, B_2, \dots)$ simply reverses the encryption process. Figure 3 is a flowchart for decryption of the encryption example shown in Figure 2.

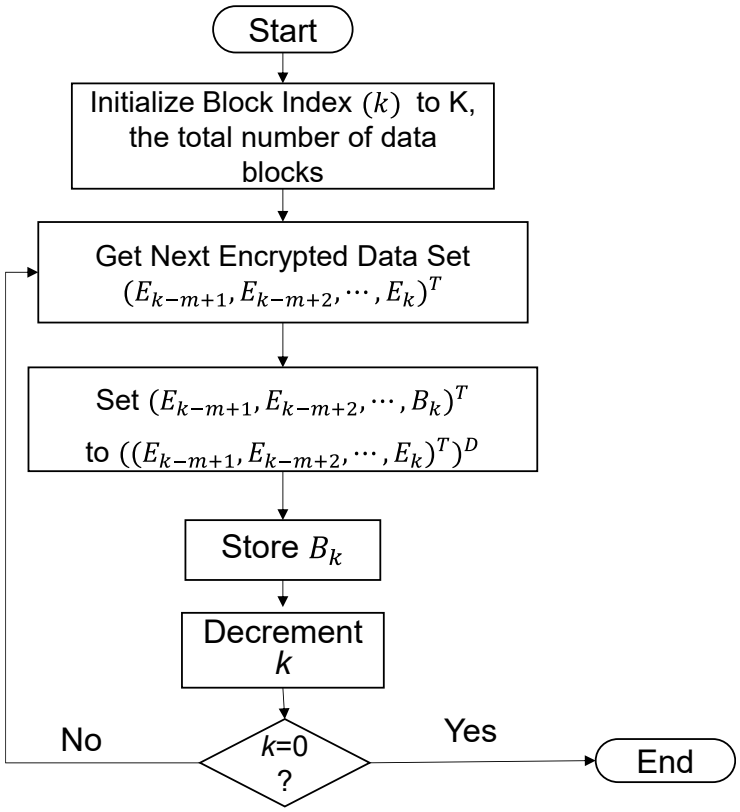

Figure 3: Decryption Reverses the Encryption Process

As before, we use a counter k to keep track of the data blocks that have been received for decryption. Since decryption reverses the encryption process, the counter k is initially set to the last block number in the set of data blocks.

Just as with encryption, decryption is performed m data blocks at a time, where m is the rank of the encryption matrix E.

If m data blocks $(M_{i+1}, M_{i+2}, \dots, M_{i+m-1}, M_{i+m})$ are encrypted by

$$((E_{i+1}, E_{i+2}, \dots, E_{i+m})^T)(mod\ n) = (((M_{i+1}, M_{i+2}, \dots, M_{i+m-1}, M_{i+m})^T)\text{^^}E)(mod\ n),$$

then decryption using the private key is

$$((E_{i+1}, E_{i+2}, \dots, E_{i+m})^T)\text{^^}D(mod\ n = (((M_{i+1}, M_{i+2}, \dots, M_{i+m-1}, M_{i+m})^T)\text{^^}E)\text{^^}D(mod\ n)$$

$$= (M_{i+1}, M_{i+2}, \dots, M_{i+m-1}, M_{i+m})^T \text{^^} (D * E) (mod\ n) \quad (10)$$

$$= (M_{i+1}, M_{i+2}, \dots, M_{i+m-1}, M_{i+m})^T\ (mod\ n).$$

This last equation is true by Fermat's little theorem since $D * E = I\ (mod\ \varphi(n))$.

For the example of Figure 2, each data block is encrypted with the encryptions of the previous $m-1$ data blocks,

$$((F_{i+1}, F_{i+2}, \dots, F_{i+m})^T)(mod\ n) = (((E_{i+1}, E_{i+2}, \dots, E_{i+m-1}, B_{i+m})^T)\text{^^}E)(mod\ n)$$

and so, from (10), decryption using the private key is performed according to the formula

$$((F_{i+1}, F_{i+2}, \dots, F_{i+m})^T)\text{^^}D(mod\ n) = (((E_{i+1}, E_{i+2}, \dots, E_{i+m-1}, B_{i+m})^T)\text{^^}E)\text{^^}D(mod\ n)$$

$$= (E_{i+1}, E_{i+2}, \dots, E_{i+m-1}, B_{i+m})^T \text{^^} (D * E) (mod\ n) \quad (11)$$

$$= (E_{i+1}, E_{i+2}, \dots, E_{i+m-1}, B_{i+m})^T\ (mod\ n).$$

Recall that when this set of data blocks was originally encrypted, it consisted of one unencrypted value, B_k, and $m-1$ previously encrypted data elements, E_{k-1}, E_{k-2}, …, E_{k-m+1}, Since decryption simply reverses this process, the output of decryption is one unencrypted value B_k and $m-1$ previously encrypted values.

The revealed value B_k is stripped off, *k* is decremented and the process repeats.

To illustrate this approach further, if $m = 3$ and the encrypted message contains $k = 5$ data blocks, then:

- Set (E_3, E_4, B_5) equal to the decryption of (E_3, E_4, E_5).
- Set (E_2, E_3, B_4) equal to the decryption of (E_2, E_3, E_4).
- Set (E_1, E_2, B_3) equal to the decryption of (E_1, E_2, E_3).
- Set (E_0, E_1, B_2) equal to the decryption of (E_0, E_1, E_2).
- Set (E_{-1}, E_0, B_1) equal to the decryption of (E_{-1}, E_0, E_1).

The decrypted data stream is then given by $(B_1, B_2, B_3, B_4, B_5)$.

In the case where each of the data blocks is encrypted, one block at a time, with $m-1$ blocks of random nonce, equation (10) becomes

$$((E_{i+1}, E_{i+2}, \dots, E_{i+m})^T)\text{^^}D = (((R_{i+1}, R_{i+2}, \dots, R_{i+m-1}, B_{i+m})^T)\text{^^}E)\text{^^}D(mod\ n)$$

$$= (R_{i+1}, R_{i+2}, \dots, R_{i+m-1}, B_{i+m})^T \text{^^} (D * E) (mod\ n)$$

$$= (R_{i+1}, R_{i+2}, \dots, R_{i+m-1}, B_{i+m})^T.$$

4. Security and Performance

In general, this method has the same security as RSA encryption, since computation of $D = E^{-1}$ in $GL(m, \varphi(n))$ generally requires knowledge of $\varphi(n)$. Computing $\varphi(n)$ from n is believed to require factoring n, which is believed to be very hard.

To compute X^^E requires no more than $O(m^2 Log(n))$ computations. To see this, note that computing $x^j (mod\ n)$ for $j \leq n$ using the squaring method requires $O(Log(n))$ multiplications. Therefore, the number of multiplications required to compute a single component of $X\hat{}\hat{}E$, does not exceed $O(mLog(n))$ computations, since each component is the product of *m* integers, each of which is exponentiated no more than *n* times. Since *X* has m components, this implies that $O(m^2 Log(n))$ are required to fully compute all components of $X\hat{}\hat{}E$.

It follows then that if a message consists of k data blocks, then the number of multiplications required to encrypt the entire message is on the order of $O(kmLog(n))$, assuming each data block is encrypted $O(1)$ times.

5. Conclusion

RSA encryption is rarely used for streaming data, in part because it can be susceptible to frequency analysis attacks and chosen ciphertext attacks.

In this paper, we have shown how the RSA algorithm can be generalized to support full matrix-based encryption using matrix encryption and decryption keys and a novel matrix generalization of scalar exponentiation. When random nonce is added to the data stream, encryption results will vary from run to run, defeating frequency analysis attacks. The algorithm also supports confusion across data blocks, which defeats chosen ciphertext attacks.

Security is just as strong as scalar RSA encryption and in general requires ability to factor the product of two large primes.

In short, matrix-based RSA is a valid encryption alternative for handling streaming data.

6. References

1. Backal, “Virtual matrix encryption (VME) and virtual key cryptographic method and apparatus”, US Patent US7346162, published April 17, 2001.

2. Chuang, Chih-Chwen and James George Dunham, “Matrix Extensions of the RSA Algorithm”, Advances in Cryptology – CRYPTO ’90, A.J. Menezes and S.A. Vanstone (Eds.): LNCS 537, pp. 140-155, 1991. © Springer-Verlag Berlin Heidelberg 1991

3. Glatfelter and Raab, “Cryptography using a symmetric frequency-based encryption algorithm”, US Patent US8855303, published October 7, 2014.

4. LeVeque, William J., Fundamentals of Number Theory, ISBN-13: 9780486689067, Dover Publications, 1996.

5. Rivest, et. al., “Cryptographic Communication System and Method”, US Patent US4405829, published September 20, 1983.

6. Singh, “Cryptosystems”, US Patent US20040174995, published September 9, 2004.

7. Slavin, “Public key cryptograph using matrices”, US Patent US7346162, published March 28, 2008.

8. Strang, Gilbert, Introduction to Linear Algebra, Fourth Edition, ISBN-13: 9780980232714, Wellesley-Cambridge Press, 2009.

9. United States National Institute of Standards and Technology (NIST), "Announcing the ADVANCED ENCRYPTION STANDARD (AES)" (PDF). Federal Information Processing Standards Publication 197, November 26, 2001.