Extending the Hill Cipher

John Chase & Matthew Davis

November 29, 2010

Contents

1 Introduction

The classic Hill Cipher is a symmetric cipher using matrix multiplication, first introduced by Lester Hill in a short paper published in 1929 [1]. Hill's cipher consists of breaking encoded plaintext into blocks (vectors) of size n, then multiplying each by an $n \times n$ key matrix K and reducing modulo some number m to yield the ciphertext. More simply, the plaintext block X encrypts to Y by

$$Y = XK \pmod{m}$$

Decryption is only possible if the key matrix is invertible modulo m.

At the time when Hill introduced this encryption scheme, this was computationally intensive to do by hand. Hill had a machine built (Figure 1) to perform the encryption with a fixed encryption matrix, but his machine never sold and his cipher has never been used in practice.

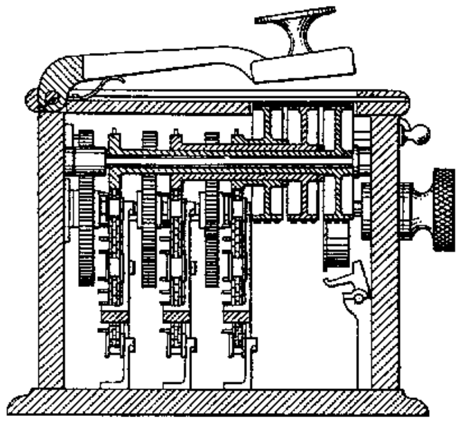
Figure 1: Hill's Machine [2]

The Hill Cipher described above is easily cryptanalyzed by the known-plaintext attack. If the block size of the Hill Cipher is n, we would need n pairs of plaintext and ciphertext row vectors. We would enter them in two $n \times n$ matrices: plaintext matrix X and ciphertext matrix Y. For the key matrix K, we know that $Y = XK$. Provided that X is invertible mod m, the key is recoverable as $K = X^{-1}Y$. If X is singular, we require more plaintext-ciphertext pairs with which we repeat the process.

From modern symmetric ciphers we expect strength against the ciphertext-only attack, the known-plaintext attack, the chosen-plaintext attack, and the chosen-ciphertext attack [3]. Thus, the classic Hill Cipher is of little practical use. Despite its lack of acceptance, several authors have suggested modifications to the Hill Cipher which make it more secure against these attacks [4, 5, 6, 7, 8]. Efforts to redeem the Hill Cipher are not without merit. If the cipher can be strengthened against these attacks, it could actually be used in practice. In this paper, we discuss and implement two such modified Hill ciphers.

2 SVK Hill Cipher

In 2010, Sastry, Varanasi, and Kumar introduced a variant of the Hill Cipher, which uses a pair of key matrices and a permutation scheme. In this variant, the plaintext takes the form of a square matrix with the same dimensions as the key matrices. The plaintext block is then multiplied by one key on the left and the other key on the right. Following this multiplication, the entries of the resulting matrix are converted to binary form, and the 128 resulting bits are permuted before being converted back to decimal form as a completely different matrix. Finally, the entire procedure is repeated a specific number of times before the ciphertext takes its final form and is transmitted to the receiver [4].

The procedure is probably best illustrated with an example. Suppose Alice wants to send the message "Abbot=A. Square" to Bob. First, the characters in Alice's message are converted to decimal numbers between 0 and 255 using the EBCDIC code. The numbers representing the characters in the message are placed into a 4×4 matrix. (Longer messages would be placed into multiple 4×4 matrices; shorter messages would have extra characters added to fill out the 16 entries in a 4×4 matrix.) Alice's message yields the following matrix:

$$\begin{bmatrix} 193 & 130 & 130 & 150 \\ 163 & 163 & 126 & 193 \\ 175 & 64 & 226 & 152 \\ 164 & 129 & 153 & 133 \end{bmatrix}$$

Suppose Alice and Bob have agreed on the following two keys, K, and L.

$$K = \begin{bmatrix} 18 & 33 & 109 & 210 \\ 78 & 43 & 102 & 64 \\ 133 & 17 & 29 & 89 \\ 99 & 87 & 114 & 12 \end{bmatrix}$$

$$L = \begin{bmatrix} 19 & 74 & 20 & 103 \\ 88 & 30 & 41 & 19 \\ 211 & 201 & 136 & 87 \\ 77 & 40 & 92 & 126 \end{bmatrix}$$

The matrix K is to be used as a multiplier on the left of the plaintext block, and L is to be used as a multiplier on the right of the plaintext block. So Alice's first step in the encrypting process is to find the product KPL, which is given by:

$$KPL = \begin{bmatrix} 37 & 84 & 53 & 207 \\ 252 & 250 & 237 & 134 \\ 122 & 149 & 90 & 88 \\ 115 & 94 & 211 & 45 \end{bmatrix}$$

After this product is found, the permutation step is completed in three stages. First these 16 entries are converted into binary form, and their binary forms are entered as rows in a new 16×8 matrix. Here, the binary form of 37 (00100101) would be the first row of this matrix, the binary form of 84 (01010100) would be the second row, etc. Here is that matrix in its entirety:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 \end{bmatrix}$$

Next, the 128 bits that make up this matrix are permuted according to the following scheme. The matrix is divided horizontally into two equal sections, so the first eight rows are considered separate from the last eight rows.

In the top (first eight rows), beginning with the last (eighth) row, the bits are taken from the right to the left and used to fill the first column in the new (permuted) matrix. That is, the first column of the new matrix is filled first by the 8 bits in row 8 of the old matrix, taking them from right to left (column 8 back to column 1), and then by the 8 bits from row 7, taking them the same way. The next three columns of the new matrix are filled the same way using pairs of rows of the upper half of the old matrix.

Next, the lower half (rows 9 through 16) are used to fill in the remaining columns of the new matrix. The columns are filled in a similar way, using the rows from the old matrix, but this time the rows are taken in order, beginning with row 9, and the bits are taken in their usual order, from left to right. So column 5 in the new matrix is filled using the bits (from left to right) of row 9, followed immediately by the bits from row 10. Columns 6-8 are filled in the same way.

Following with our example, the new (permuted) matrix would look like this:

$$\begin{bmatrix} 0&0&1&0&0&0&0&1\\ 1&1&1&0&1&1&1&1\\ 1&0&1&1&1&0&1&0\\ 0&1&1&0&1&1&1&1\\ 0&1&0&1&1&1&0&0\\ 0&1&0&0&0&0&0&0\\ 0&1&1&1&1&1&1&1\\ 1&1&1&0&0&0&1&1\\ 1&0&1&1&1&0&0&0\\ 0&0&0&0&0&1&1&0\\ 1&1&1&1&0&0&0&1\\ 1&1&0&0&1&1&1&0\\ 0&1&1&0&0&1&1&1\\ 1&1&1&1&1&0&1&1\\ 1&1&0&0&0&0&1&0\\ 1&1&0&0&1&0&0&1 \end{bmatrix}$$

Finally, the third stage of the permutation process is carried out, as this new binary matrix is converted back into decimal form as a 4×4 matrix, where each entry comes from converting the binary expression given in a row to its decimal form. In our example, we would produce this matrix:

$$\begin{bmatrix} 33&239&186&111\\ 92&64&127&227\\ 184&6&241&206\\ 103&251&194&201 \end{bmatrix}$$

This is the final result of one iteration of the SVK Hill Cipher encryption scheme. In practice, along with the two key matrices, Alice and Bob also agree on a random integer, known as r, that will indicate how many iterations of this process the plaintext goes through before its resulting ciphertext is sent to Bob. Suppose Alice and Bob have agreed that $r = 19$. This would cause the following ciphertext matrix to be sent to Bob:

$$\begin{bmatrix} 248&139&132&44\\ 232&218&237&165\\ 81&189&155&86\\ 182&10&65&19 \end{bmatrix}$$

When Bob receives the ciphertext, he knows K, L, and r. So he can easily decrypt the message by using the inverses of K and L. The decryption scheme simply reverses the steps laid out in the encryption scheme. The ciphertext is converted to a binary matrix, the inverse permutation is performed, the matrix is converted back into decimal form, and then the product $K^{-1}CL^{-1} \pmod{256}$ is computed. This procedure is repeated r times (in our example that would be 19) and finally he arrives back at the original plaintext that Alice sent.

The Avalanche Effect

Sastry, et. al identify a major strength of this procedure against potential cryptanalytical attacks, which they refer to as the avalanche effect [4]. What they are referring to is the dramatic change in the ciphertext that is observed if only a slight change is made to either the original plaintext or to one of the key matrices. We can illustrate this effect here.

Suppose we change Alice's original message from"Abbott=A. Square" to "Abbott=A. Sqvare". This would change only one entry in the original plaintext matrix, changing the first column, fourth row entry from 164 to 165.

Here is the binary representation of the final version of the ciphertext after 19 iterations of the encryption process:

$$\begin{bmatrix}
1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0
\end{bmatrix}$$

By comparison, here is the binary representation of the ciphertext after 19 iterations using the original plaintext:

$$\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 1 & 1 & 1 & 1 & 0 & 1 \\
1 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1
\end{bmatrix}$$

These two matrices differ in 64 of their 128 bits, an indication that the cipher would hold up well against cryptanalytical attacks.

Similarly, we can observe the avalanche effect on a slight change to one of the key matrices. Suppose we change the first row, first column entry of the key matrix K from 18 to 17.

After applying the same encryption scheme for 19 iterations, we get the following binary representation of the ciphertext matrix that would be sent to Bob:

$$\begin{bmatrix}1&0&1&0&0&0&1&0\\1&0&1&0&0&0&1&0\\0&1&1&1&1&1&0&1\\0&0&1&0&0&0&1&1\\1&0&1&1&0&1&1&1\\1&1&0&1&1&1&0&0\\0&1&1&0&1&1&0&0\\1&1&0&1&1&1&1&0\\0&0&1&1&1&0&0&1\\1&0&0&1&0&0&1&1\\0&1&1&1&0&1&1&0\\1&1&1&1&0&1&0&1\\1&0&0&1&1&1&1&0\\1&0&1&0&0&0&0&0\\0&0&0&0&1&1&1&1\\1&0&0&0&1&1&1&1\end{bmatrix}$$

Comparing this to the original binary representation of the ciphertext shown above, we observe this matrix also (coincidentally) differs from the original in 64 of the 128 bits. Again, this is an indication of the strength of the SVK Hill Cipher against cryptanalytic attacks.

3 TF Hill Cipher

Another secure variant of the Hill Cipher was introduced by Toorani and Falahati in 2009 [5]. The scheme is based on the *Affine Hill Cipher* [3, p. 42]. The classic Hill Cipher encrypts plaintext row vector X as ciphertext Y by the matrix multiplication $Y = XK \pmod{m}$. The Affine Hill Cipher introduces a row vector V which is part of the cryptosystem key. In this case, the secret key (K, V) is shared and Alice encrypts X as $Y = XK + V \pmod{m}$, passing the ciphertext Y to Bob. This simple affine modification of the cipher provides little additional security. Toorani and Falahati purpose a much more secure version of the Affine Hill Cipher.

The basic encryption algorithm for Alice is

$$Y_t = v_0 X_t K + V_t \pmod{p}$$

For the Toorani-Falahati Hill Cipher (TFHC), we require p to be prime. For each plaintext block X_t, a new a_t is recursively generated from a random initial value a_0 using a one-way hash function. Then the constant v_0 and vector V_t are generated using a_t. Among other things, this scheme results in two identical plaintext blocks X_a and X_b being encrypted as completely different ciphertext blocks. In more detail, we have the following scheme (rendered slightly different than in [5]):

Encryption.

1. Alice selects random integers a_0 and b such that $0 < a_0 < p - 1$ and $1 < b < n^2$.
2. She computes $r = a_0 k_{ij} \pmod{p}$ where $i = \lceil b/n \rceil$ and $j = b - n(i - 1)$.
3. She encodes the plaintext message into t row vectors $X_1 = [x_1\ x_2\ \cdots\ x_n]$, $X_2 = [x_1\ x_2\ \cdots\ x_n]$, ... , $X_t = [x_1\ x_2\ \cdots\ x_n]$.
4. For each plaintext block X_t, she computes $a_t = H(a_{t-1})$ using a recursive one-way hash function H.
5. She then computes v_0: If a_t is invertible mod p, that is $a_t \not\equiv 0 \pmod{p}$, she lets $v_0 = a_t \pmod{p}$. Otherwise $v_0 = 1$.

6. She then computes row vector $V_t = [v_1\ v_2\ \cdots\ v_n]$ with the recursive expression $v_i = k_{ij} + \tilde{v}_{i-1} a_t \pmod{p}$ for $i = 1, \ldots, n$ and $j = (v_{i-1} \mod n) + 1$, in which $\tilde{v}_{i-1} = 2^{\lceil \gamma/2 \rceil} + \left(v_{i-1} \mod 2^{\lceil \gamma/2 \rceil}\right)$ and $\gamma = \lfloor \log_2 v_{i-1} \rfloor + 1$ denotes the bit length of v_{i-1}.
7. She encrypts each X_t, as $Y_t = v_0 X_t K + V_t \pmod{p}$.
8. Alice passes to Bob all ciphertext blocks Y_t, b, and r over a public channel.

Decryption.

1. Bob knows the secret key K. He receives from Alice ciphertext blocks Y_t, b, and r over a public channel.
2. He computes $u = k_{ij}^{-1} \pmod{p}$ and $a_0 = ru \pmod{p}$ in which $i = \lceil b/n \rceil$ and $j = b - n(i-1)$.
3. Since Bob now has a_0, he computes V_t for each ciphertext block just as Alice did.
4. Each plaintext block is given by $X_t = v_0^{-1} (Y_t - V_t) K^{-1} \pmod{p}$.

The scheme above is sufficiently dense that an example serves to elucidate. The program we wrote to perform the following computations can be seen in appendix B.1 and the full output for this particular example can be seen in appendix B.2.

Encryption example.

Suppose our plaintext is "over the hill and through the woods". We might encode it using the following alphabet with $p = 29$:

a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	2	3	4	5	6	7	8	9	10	11	12

n	o	p	q	r	s	t	u	v	w	x	y	z
13	14	15	16	17	18	19	20	21	22	23	24	25

[space]	' [apostrophe]	. [period]	
26	27	28	

Our encoded plaintext would be

[14 21 4 17 26 19 7 4 26 7 8 11 11 26 0 13 3 26 19 7 17 14 20 6 7 26 19 7 4 26 22 14 14 3 18]

Suppose we choose a block size of $n = 5$.
Let the key matrix be

$$K = \begin{bmatrix} 1 & 5 & 17 & 25 & 16 \\ 9 & 0 & 18 & 19 & 23 \\ 26 & 4 & 10 & 13 & 2 \\ 3 & 4 & 8 & 9 & 22 \\ 7 & 24 & 15 & 17 & 12 \end{bmatrix}$$

1. We choose a random a_0 and b such that $0 < a_0 < p - 1$ and $1 < b < n^2$. We choose

$$a_0 = 20$$

$$b = 17$$

since $0 < a_0 < 28$, and since $1 < b < 25$.

2. Next we compute $r = a_0 k_{ij} \pmod p$ where $i = \lceil b/n \rceil$ and $j = b - n(i-1)$. We find
$$r = 22$$
3. We encode the plaintext into row vectors X_t of length $n = 5$
$$\begin{aligned} X_1 &= [14\ 21\ 4\ 17\ 26] \\ X_2 &= [19\ 7\ 4\ 26\ 7] \\ X_3 &= [8\ 11\ 11\ 26\ 0] \\ X_4 &= [13\ 3\ 26\ 19\ 7] \\ X_5 &= [17\ 14\ 20\ 6\ 7] \\ X_6 &= [26\ 19\ 7\ 4\ 26] \\ X_7 &= [22\ 14\ 14\ 3\ 18] \end{aligned}$$
4. For each plaintext block X_t, we compute $a_t = H(a_{t-1})$ using a recursive one-way hash function H. In our case, we will use the MD5 hash function. For example, to encrypt the first plaintext block X_1, we calculate $a_1 = H(a_0)$, obtaining
$$a_1 = H(20) = 203295115078782523880027993804846545796$$
5. We then compute v_0: If a_t is invertible mod p, that is $a_t \not\equiv 0 \pmod p$, we let $v_0 = a_t \pmod p$. Otherwise $v_0 = 1$. In our case
$$v_0 = a_1 \pmod{29} = 7$$
6. We then compute row vector $V_t = [v_1\ v_2\ \cdots\ v_n]$ with the recursive expression $v_i = k_{ij} + \tilde{v}_{i-1} a_t \pmod p$ for $i = 1, \ldots, n$ and $j = (v_{i-1} \mod n) + 1$, in which $\tilde{v}_{i-1} = 2^{\lceil \gamma/2 \rceil} + \left(v_{i-1} \mod 2^{\lceil \gamma/2 \rceil}\right)$ and $\gamma = \lfloor \log_2 v_{i-1} \rfloor + 1$ denotes the bit length of v_{i-1}. In our case,

$i = 1$	$j = 3$	$\gamma = 3$	$\tilde{v}_0 = 7$	$v_1 = 8$
$i = 2$	$j = 4$	$\gamma = 4$	$\tilde{v}_1 = 4$	$v_2 = 18$
$i = 3$	$j = 4$	$\gamma = 5$	$\tilde{v}_2 = 10$	$v_3 = 25$
$i = 4$	$j = 1$	$\gamma = 5$	$\tilde{v}_3 = 9$	$v_4 = 8$
$i = 5$	$j = 4$	$\gamma = 4$	$\tilde{v}_4 = 4$	$v_5 = 16$

So $V_1 = [8\ 18\ 25\ 8\ 16]$.

7. We encrypt each X_t, as $Y_t = v_0 X_t K + V_t \pmod p$. In the case of X_1, we have
$$Y_1 = (7)\,[14\ 21\ 4\ 17\ 26]\,K + V_1 \pmod p = [18\ 12\ 5\ 7\ 21]$$
8. We repeat this for all the other plaintext blocks, yielding:
$$\begin{aligned} Y_1 &= [18\ 12\ 5\ 7\ 21] \\ Y_2 &= [18\ 22\ 14\ 5\ 21] \\ Y_3 &= [23\ 24\ 13\ 14\ 8] \\ Y_4 &= [20\ 9\ 1\ 2\ 4] \\ Y_5 &= [10\ 18\ 26\ 1\ 26] \\ Y_6 &= [5\ 18\ 8\ 3\ 24] \\ Y_7 &= [4\ 5\ 0\ 12\ 0] \end{aligned}$$

We pass these ciphertext blocks along with $b = 17$ and $r = 22$ over a public channel to the recipient.

Decryption example.

1. We know the secret key K. We receive from the ciphertext blocks Y_t (above), $b = 17$, and $r = 22$ over the public channel.
2. We compute $u = k_{ij}^{-1} \pmod p$ and $a_0 = ru \pmod p$ in which $i = \lceil b/n \rceil$ and $j = b - n(i-1)$ and find
$$a_0 = 20$$
3. Since we now have $a_0 = 22$, we compute V_t for each ciphertext block just as the sender did.
4. We then decrypt each ciphertext block by $X_t = v_0^{-1}(Y_t - V_t)K^{-1} \pmod p$, obtaining:
$$\begin{aligned} X_1 &= [14\ 21\ 4\ 17\ 26] \\ X_2 &= [19\ 7\ 4\ 26\ 7] \\ X_3 &= [8\ 11\ 11\ 26\ 0] \\ X_4 &= [13\ 3\ 26\ 19\ 7] \\ X_5 &= [17\ 14\ 20\ 6\ 7] \\ X_6 &= [26\ 19\ 7\ 4\ 26] \\ X_7 &= [22\ 14\ 14\ 3\ 18] \end{aligned}$$

Decoding this, we have our plaintext message "over the hill and through the woods".

The SFHC is a strong symmetric cipher with respect to all the attacks mentioned in [3]. Since the key used for encrypting each plaintext block is unique, the SFHC scheme resists the known-plaintext attack, the chosen-plaintext attack, and the chosen-ciphertext attack. In reality, a large prime number p would be used as the modulus, thus providing security against the ciphertext-only attack.

4 Conclusion

In this paper we have given a brief overview of the classic Hill Cipher and explained its weaknesses with respect to common attacks. We reviewed two recently purposed variants on Hill's scheme (among many other variants suggested in the literature). We described their strength, explained the encryption scheme, and implemented them using GAP and Python. From the discussion, both are found to be strong against the known-plaintext, known-ciphertext, chosen-plaintext, and chosen-ciphertext attacks. Since this is the performance we expect from a modern symmetric cipher, these modifications are well-suited for actual implementation. Perhaps Hill's Cipher is not simply a curiosity in the halls of cryptographic history, but a cipher ready (with slight modification) for practical use in modern applications.

References

[1] L.S. Hill, "Cryptography in an Algebraic Alphabet," *American Mathematical Monthly*, Vol.36, No.6, pp.306-312, 1929.

[2] L. Weisner and L.S. Hill, "Message Protector," U.S. Patent 1,845,947, Feb. 16, 1932. Accessed on Nov. 29, 2010: <http://www.google.com/patents/about?id=rFtHAAAAEBAJ>

[3] D.R. Stinson, *Cryptography Theory and Practice*, 3rd edition, Chapman & Hall/CRC, pp.13-37, 2006.

[4] V.U.K. Sastry, A. Varanasi, and S.U. Kumar, "A Modified Hill Cipher Involving a Pair of Keys and a Permutation," *International Journal of Computer and Network Security*, Vol.2, No.9, pp.105-108, September 2010.

[5] M. Toorani and A Falahati, "A Secure Variant of the Hill Cipher," *Proceedings - IEEE Symposium on Computers and Communications*, pp.313-316, 2009.

[6] I. Gupta, J. Singh, R. Chaudhary, "Cryptanalysis of an Extension of the Hill Cipher," *Cryptologia*, Vol.31, pp.246-253, 2007.

[7] Shahrokh Saeednia, "How to Make the Hill Cipher Secure," *Cryptologia*, Vol.24, No.4, pp.353-360, October 2000.

[8] I.A. Ismail, Mohammed Amin, and Hossam Diab, "How to repair the Hill cipher," *Journal of Zhejiang University-Science A*, Vol.7, No.12, pp.2022-2030, December 2006.

A VFK Hill Cipher Implementation

A.1 SVKHillCipher.txt

```
#This function takes an integer between 0 and 255
# and returns a row vector that indicates
# the integer's binary representation.
# The components of the vector are
# the coefficients of the powers of 2 in
# ascending order from 0 to 7.

base_2:= function(c)
  local i,q,l;
  q:=c;
  l:=[0,0,0,0,0,0,0,0];
  i:=1;
  while q<>0 do
     l[i]:=RemInt(q,2);
     q:=QuoInt(q,2);
     i:=i+1;
  od;
  return(l);
end;;


#This function is identical to the greatest integer
# function, where the value returned is the largest
# integer less than or equal to the argument.

Floor:=function(c)
   if c<0 and not IsInt(c) then c:=Int(c)-1;
   else c:=Int(c);
   fi;
   return(c);
end;;


#This is the first of three functions used in the
# permutation process.  This function takes the elements
# of a 4x4 matrix in order from the element in the first
# row, first column, moving right across each row and then
# down through the following rows.  It finds the binary
# representation of each element and builds a new 16x8
# matrix where each row is the binary representation of
# an entry in the original 4x4 matrix.

CreateBinaryMatrix:= function(m)
local P,Q,i,j,k,l,r;
i:=1;
l:=1;
P:=m;
Q:=NullMat(16,8);
  while l<17 do
  while i<5 do
    j:=1;
    while j<5 do
      r:=base_2(P[i][j]);
        for k in [0..7] do
          Q[l][k+1]:=r[8-k]; # <- This line makes sure to
        od; # put the binary string in the
      l:=l+1; # usual order (units digit
      j:=j+1; # on the right) when it forms
    od; # a row in this matrix.
    i:=i+1;
  od;
  od;
return(Q);
end;;


#This is the function that actually does the permuting.


PermuteMatrix:= function(m)
   local P,Q,i,j,u,v;
   P:=m;
   Q:=NullMat(16,8);
   i:=1;

   # The procedure takes the top half of original matrix first,
 # and from the 8th row, backwards up to the first row it
 # takes the elements in a row in reverse order (from the 8th
 # column to the first) and uses them to fill the new
 # matrix's first four columns.  For example, the 8th row's
 # elements, in reverse order, fill the first 8 entries in
 # column 1 of the new (permuted) matrix.  The 7th row's
 # elements (again in reverse order) fill the last 8 entries
 # (rows 9 through 16) in column 1.  This continues with the
 # remaining six rows of the top half of the matrix, filling
 # columns 2 though 4 of the new matrix.

   while i<9 do
      j:=1;
      while j<9 do
        if i mod 2 = 0 then u:=9-j;
        else u:= 17-j;
        fi;
        v:=Floor((2-i)/2)+4;
        Q[u][v]:=P[i][j];
        j:=j+1;
      od;
      i:=i+1;
    od;

   #Now, in this section attention is turned to the bottom of
   # the original matrix.  Here the procedure is similar, rows
   # of the original matrix are used to fill columns of the
   # new matrix, but here the rows are taken in order (9, 10,
   # through 16, and the elements are taken in order (not
   # reversed) to fill the columns.  So rows 9 and 10 are used
   # to fill column 5, then rows 11 and 12 fill column 6, etc.

   while i<17 do
      j:=1;
      while j<9 do
        if i mod 2 = 0 then u:=j+8;
        else u:= j;
        fi;
        v:=Floor((i+1)/2);
        Q[u][v]:=P[i][j];
        j:=j+1;
      od;
      i:=i+1;
   od;
return(Q);
end;;


#This is the function that performs the permuatation
# described above, in reverse.  It is needed for the
# decryption procedure.

InvPermuteMatrix:=function(m)
   local P,Q,i,j,u,v;
   Q:=m;
   P:=NullMat(16,8);
   j:=1;
   while j<5 do
     i:=1;
     while i<17 do
       if i<9 then
          v:=9-i;
          u:=10-2*j;
       else
          v:=17-i;
          u:=9-2*j;
       fi;
       P[u][v]:=Q[i][j];
       i:=i+1;
     od;
     j:=j+1;
   od;
  while j<9 do
    i:=1;
    while i<17 do
      if i<9 then
        v:=i;
        u:=2*j-1;
      else
```

```
        v:=i-8;
        u:=2*j;
      fi;
      P[u][v]:=Q[i][j];
      i:=i+1;
    od;
    j:=j+1;
  od;
return(P);
end;;


#This is the third stage of the permutation process.
# This takes the newly permuated matrix that is a
# 16x8 matrix, where each row represents an entry of the
# 4x4 matrix in its binary form, and converts the numbers
# to their decimal equivalents, and then rebuilds the matrix
# into its orignial 4x4 form.

BintoDecMatrix:=function(m)
   local P,Q,i,j,k,l,u;
   P:=m;
   Q:=NullMat(4,4);
   i:=1;
   while i<17 do
      u:=0;
      j:=1;
      while j<9 do
         if P[i][j]<>0 then u:=u+2^(8-j);
         fi;
         j:=j+1;
      od;

      #This part of the function determines the location in the
      # 4x4 matrix where each entry, based on its row position in
      # the 16x8 matrix, belongs.

      k:= Floor((i+3)/4);
      if i mod 4 <> 0 then l:= i mod 4;
      else l:=4;
      fi;
      Q[k][l]:=u;
      i:=i+1;
   od;
   return(Q);
end;;


Permute:= function(P)
   P:= CreateBinaryMatrix(P);
   P:= PermuteMatrix(P);
   P:= BintoDecMatrix(P);
   return(P);
```

```
end;;


InvPermute:= function(P)
   P:= CreateBinaryMatrix(P);
   P:= InvPermuteMatrix(P);
   P:= BintoDecMatrix(P);
   return(P);
end;;


# The encryption function takes a plaintext matrix, two keys (left and right),
#  and r, an integer # indicating the number of iterations to be performed.

Encrypt:= function(P,K,L,r)
   local i, C;
   for i in [1..r] do
      P:= K*P*L mod 256;
      P:= Permute(P);
   od;
   C:=P;
   PrintArray(C);
   return(C);
end;;


Decrypt:= function(C,K,L,r)
   local i, P;
   for i in [1..r] do
      C:= InvPermute(C);
      C:= K^(-1)*C*L^(-1) mod 256;
   od;
   P:=C;
   PrintArray(P);
   return(P);
end;;
```

A.2 Sample output of SVKHillCipher.txt

A.2.1 Sample 1

```
gap> PrintArray(P);
[ [  193,  130,  130,  150 ],
  [  163,  163,  126,  193 ],
  [  175,   64,  226,  152 ],
  [  164,  129,  153,  133 ] ]
gap> PrintArray(K);
[ [   18,   33,  109,  210 ],
  [   78,   43,  102,   64 ],
  [  133,   17,   29,   89 ],
  [   99,   87,  114,   12 ] ]
```

```
gap> PrintArray(L);
[ [   19,   74,   20,  103 ],
  [   88,   30,   41,   19 ],
  [  211,  201,  136,   87 ],
  [   77,   40,   92,  126 ] ]
gap> Q:=K*P*L mod 256;;
gap> PrintArray(Q);
[ [   37,   84,   53,  207 ],
  [  252,  250,  237,  134 ],
  [  122,  149,   90,   88 ],
  [  115,   94,  211,   45 ] ]
gap> R:=CreateBinaryMatrix(Q);;
gap> PrintArray(R);
[ [  0,  0,  1,  0,  0,  1,  0,  1 ],
  [  0,  1,  0,  1,  0,  1,  0,  0 ],
  [  0,  0,  1,  1,  0,  1,  0,  1 ],
  [  1,  1,  0,  0,  1,  1,  1,  1 ],
  [  1,  1,  1,  1,  1,  1,  0,  0 ],
  [  1,  1,  1,  1,  1,  0,  1,  0 ],
  [  1,  1,  1,  0,  1,  1,  0,  1 ],
  [  1,  0,  0,  0,  0,  1,  1,  0 ],
  [  0,  1,  1,  1,  1,  0,  1,  0 ],
  [  1,  0,  0,  1,  0,  1,  0,  1 ],
  [  0,  1,  0,  1,  1,  0,  1,  0 ],
  [  0,  1,  0,  1,  1,  0,  0,  0 ],
  [  0,  1,  1,  1,  0,  0,  1,  1 ],
  [  0,  1,  0,  1,  1,  1,  1,  0 ],
  [  1,  1,  0,  1,  0,  0,  1,  1 ],
  [  0,  0,  1,  0,  1,  1,  0,  1 ] ]


gap> S:=PermuteMatrix(R);;
gap> PrintArray(S);
[ [  0,  0,  1,  0,  0,  0,  0,  1 ],
  [  1,  1,  1,  0,  1,  1,  1,  1 ],
  [  1,  0,  1,  1,  1,  0,  1,  0 ],
  [  0,  1,  1,  0,  1,  1,  1,  1 ],
  [  0,  1,  0,  1,  1,  1,  0,  0 ],
  [  0,  1,  0,  0,  0,  0,  0,  0 ],
  [  0,  1,  1,  1,  1,  1,  1,  1 ],
  [  1,  1,  1,  0,  0,  0,  1,  1 ],
  [  1,  0,  1,  1,  1,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  1,  1,  0 ],
  [  1,  1,  1,  1,  0,  0,  0,  1 ],
  [  1,  1,  0,  0,  1,  1,  1,  0 ],
  [  0,  1,  1,  0,  0,  1,  1,  1 ],
  [  1,  1,  1,  1,  1,  0,  1,  1 ],
  [  1,  1,  0,  0,  0,  0,  1,  0 ],
  [  1,  1,  0,  0,  1,  0,  0,  1 ] ]
gap> T:=BintoDecMatrix(S);;
gap> PrintArray(T);
[ [   33,  239,  186,  111 ],
  [   92,   64,  127,  227 ],
  [  184,    6,  241,  206 ],
  [  103,  251,  194,  201 ] ]
gap> C:=Encrypt(P,K,L,1);;
[ [   33,  239,  186,  111 ],
  [   92,   64,  127,  227 ],
  [  184,    6,  241,  206 ],
  [  103,  251,  194,  201 ] ]
gap> D:=CreateBinaryMatrix(C);;
gap> PrintArray(D);
[ [  0,  0,  1,  0,  0,  0,  0,  1 ],
  [  1,  1,  1,  0,  1,  1,  1,  1 ],
  [  1,  0,  1,  1,  1,  0,  1,  0 ],
  [  0,  1,  1,  0,  1,  1,  1,  1 ],
  [  0,  1,  0,  1,  1,  1,  0,  0 ],
  [  0,  1,  0,  0,  0,  0,  0,  0 ],
  [  0,  1,  1,  1,  1,  1,  1,  1 ],
  [  1,  1,  1,  0,  0,  0,  1,  1 ],
  [  1,  0,  1,  1,  1,  0,  0,  0 ],
  [  0,  0,  0,  0,  0,  1,  1,  0 ],
  [  1,  1,  1,  1,  0,  0,  0,  1 ],
  [  1,  1,  0,  0,  1,  1,  1,  0 ],
  [  0,  1,  1,  0,  0,  1,  1,  1 ],
  [  1,  1,  1,  1,  1,  0,  1,  1 ],
  [  1,  1,  0,  0,  0,  0,  1,  0 ],
  [  1,  1,  0,  0,  1,  0,  0,  1 ] ]


gap> F:=InvPermuteMatrix(D);;
gap> PrintArray(F);
[ [  0,  0,  1,  0,  0,  1,  0,  1 ],
  [  0,  1,  0,  1,  0,  1,  0,  0 ],
  [  0,  0,  1,  1,  0,  1,  0,  1 ],
  [  1,  1,  0,  0,  1,  1,  1,  1 ],
  [  1,  1,  1,  1,  1,  1,  0,  0 ],
  [  1,  1,  1,  1,  1,  0,  1,  0 ],
  [  1,  1,  1,  0,  1,  1,  0,  1 ],
  [  1,  0,  0,  0,  0,  1,  1,  0 ],
  [  0,  1,  1,  1,  1,  0,  1,  0 ],
  [  1,  0,  0,  1,  0,  1,  0,  1 ],
  [  0,  1,  0,  1,  1,  0,  1,  0 ],
  [  0,  1,  0,  1,  1,  0,  0,  0 ],
  [  0,  1,  1,  1,  0,  0,  1,  1 ],
  [  0,  1,  0,  1,  1,  1,  1,  0 ],
  [  1,  1,  0,  1,  0,  0,  1,  1 ],
  [  0,  0,  1,  0,  1,  1,  0,  1 ] ]
gap> G:=BintoDecMatrix(F);;
gap> PrintArray(G);
[ [   37,   84,   53,  207 ],
  [  252,  250,  237,  134 ],
  [  122,  149,   90,   88 ],
  [  115,   94,  211,   45 ] ]
gap> H:=K^(-1)*G*L^(-1) mod 256;;
gap> PrintArray(H);
[ [  193,  130,  130,  150 ],
  [  163,  163,  126,  193 ],
  [  175,   64,  226,  152 ],
  [  164,  129,  153,  133 ] ]
```

A.2.2 Sample 2

```
gap> C:=Encrypt(P,K,L,19);;
[ [  248,  139,  132,   44 ],
  [  232,  218,  237,  165 ],
  [   81,  189,  155,   86 ],
  [  182,   10,   65,   19 ] ]
gap> A:=Decrypt(C,K,L,19);;
[ [  193,  130,  130,  150 ],
  [  163,  163,  126,  193 ],
  [  175,   64,  226,  152 ],
  [  164,  129,  153,  133 ] ]
```

A.2.3 Sample 3

```
gap> P[4][1]:=165;
165
gap> PrintArray(P);
[ [  193,  130,  130,  150 ],
  [  163,  163,  126,  193 ],
  [  175,   64,  226,  152 ],
  [  165,  129,  153,  133 ] ]
gap> C:=Encrypt(P,K,L,19);;
[ [  241,   37,  254,  163 ],
  [   41,   33,   56,   22 ],
  [   58,  183,  130,  170 ],
  [  118,   46,   15,  144 ] ]
gap> D:=CreateBinaryMatrix(C);;
gap> PrintArray(D);
[ [  1,  1,  1,  1,  0,  0,  0,  1 ],
  [  0,  0,  1,  0,  0,  1,  0,  1 ],
  [  1,  1,  1,  1,  1,  1,  1,  0 ],
  [  1,  0,  1,  0,  0,  0,  1,  1 ],
  [  0,  0,  1,  0,  1,  0,  0,  1 ],
  [  0,  0,  1,  0,  0,  0,  0,  1 ],
  [  0,  0,  1,  1,  1,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  1,  1,  0 ],
  [  0,  0,  1,  1,  1,  0,  1,  0 ],
  [  1,  0,  1,  1,  0,  1,  1,  1 ],
  [  1,  0,  0,  0,  0,  0,  1,  0 ],
  [  1,  0,  1,  0,  1,  0,  1,  0 ],
  [  0,  1,  1,  1,  0,  1,  1,  0 ],
  [  0,  0,  1,  0,  1,  1,  1,  0 ],
  [  0,  0,  0,  0,  1,  1,  1,  1 ],
  [  1,  0,  0,  1,  0,  0,  0,  0 ] ]
gap> P[4][1]:=164;
164
gap> C:=Encrypt(P,K,L,19);;
[ [  248,  139,  132,   44 ],
  [  232,  218,  237,  165 ],
  [   81,  189,  155,   86 ],
  [  182,   10,   65,   19 ] ]
gap> D:=CreateBinaryMatrix(C);;
gap> PrintArray(D);
[ [  1,  1,  1,  1,  1,  0,  0,  0 ],
  [  1,  0,  0,  0,  1,  0,  1,  1 ],
  [  1,  0,  0,  0,  0,  1,  0,  0 ],
  [  0,  0,  1,  0,  1,  1,  0,  0 ],
  [  1,  1,  1,  0,  1,  0,  0,  0 ],
  [  1,  1,  0,  1,  1,  0,  1,  0 ],
  [  1,  1,  1,  0,  1,  1,  0,  1 ],
  [  1,  0,  1,  0,  0,  1,  0,  1 ],
  [  0,  1,  0,  1,  0,  0,  0,  1 ],
  [  1,  0,  1,  1,  1,  1,  0,  1 ],
  [  1,  0,  0,  1,  1,  0,  1,  1 ],
  [  0,  1,  0,  1,  0,  1,  1,  0 ],
  [  1,  0,  1,  1,  0,  1,  1,  0 ],
  [  0,  0,  0,  0,  1,  0,  1,  0 ],
  [  0,  1,  0,  0,  0,  0,  0,  1 ],
  [  0,  0,  0,  1,  0,  0,  1,  1 ] ]
gap> P[4][1]:=165;
165
gap> C:=Encrypt(P,K,L,19);;
[ [  241,   37,  254,  163 ],
  [   41,   33,   56,   22 ],
  [   58,  183,  130,  170 ],
  [  118,   46,   15,  144 ] ]
gap> D:=CreateBinaryMatrix(C);;
gap> PrintArray(D);
[ [  1,  1,  1,  1,  0,  0,  0,  1 ],
  [  0,  0,  1,  0,  0,  1,  0,  1 ],
  [  1,  1,  1,  1,  1,  1,  1,  0 ],
  [  1,  0,  1,  0,  0,  0,  1,  1 ],
  [  0,  0,  1,  0,  1,  0,  0,  1 ],
  [  0,  0,  1,  0,  0,  0,  0,  1 ],
  [  0,  0,  1,  1,  1,  0,  0,  0 ],
  [  0,  0,  0,  1,  0,  1,  1,  0 ],
  [  0,  0,  1,  1,  1,  0,  1,  0 ],
  [  1,  0,  1,  1,  0,  1,  1,  1 ],
  [  1,  0,  0,  0,  0,  0,  1,  0 ],
  [  1,  0,  1,  0,  1,  0,  1,  0 ],
  [  0,  1,  1,  1,  0,  1,  1,  0 ],
  [  0,  0,  1,  0,  1,  1,  1,  0 ],
  [  0,  0,  0,  0,  1,  1,  1,  1 ],
  [  1,  0,  0,  1,  0,  0,  0,  0 ] ]
gap> PrintArray(P);
[ [  193,  130,  130,  150 ],
  [  163,  163,  126,  193 ],
  [  175,   64,  226,  152 ],
  [  165,  129,  153,  133 ] ]
gap> P[4][1]:=164;
164
gap> K[2][2]:=42;
42
gap> PrintArray(K);
[ [   18,   33,  109,  210 ],
  [   78,   42,  102,   64 ],
  [  133,   17,   29,   89 ],
  [   99,   87,  114,   12 ] ]
gap> K^(-1) mod 256;
ModRat: for <r>/<s> mod <n>, <s>/gcd(<r>,<s>) and <n> must be coprime at
result[i] := list[i] mod scalar;
 called from
MOD_LIST_SCL_DEFAULT( list, nonlist ) called from
list[i] mod scalar called from
MOD_LIST_SCL_DEFAULT( list, nonlist ) called from
<function>( <arguments> ) called from read-eval-loop
Entering break read-eval-print loop ...
you can 'quit;' to quit to outer loop, or
you can replace the integer <n> via 'return <n>;' to continue
brk> quit;
gap> K[2][2]:=43;
43
gap> K[1][1]:=17;
17
gap> PrintArray(K);
[ [   17,   33,  109,  210 ],
  [   78,   43,  102,   64 ],
  [  133,   17,   29,   89 ],
  [   99,   87,  114,   12 ] ]
gap> K^(-1) mod 256;
[ [ 176, 247, 212, 217 ], [ 182, 229, 148, 192 ], [ 29, 204, 130, 107 ],
  [ 153, 188, 231, 4 ] ]
gap> PrintArray(P);
[ [  193,  130,  130,  150 ],
  [  163,  163,  126,  193 ],
  [  175,   64,  226,  152 ],
  [  164,  129,  153,  133 ] ]
gap> C:=Encrypt(P,K,L,19);;
[ [  162,  162,  125,   35 ],
  [  183,  220,  108,  222 ],
  [   57,  147,  118,  245 ],
  [  158,  160,   15,  143 ] ]
gap> D:=CreateBinaryMatrix(C);;
gap> PrintArray(D);
[ [  1,  0,  1,  0,  0,  0,  1,  0 ],
  [  1,  0,  1,  0,  0,  0,  1,  0 ],
  [  0,  1,  1,  1,  1,  1,  0,  1 ],
  [  0,  0,  1,  0,  0,  0,  1,  1 ],
  [  1,  0,  1,  1,  0,  1,  1,  1 ],
  [  1,  1,  0,  1,  1,  1,  0,  0 ],
  [  0,  1,  1,  0,  1,  1,  0,  0 ],
  [  1,  1,  0,  1,  1,  1,  1,  0 ],
  [  0,  0,  1,  1,  1,  0,  0,  1 ],
  [  1,  0,  0,  1,  0,  0,  1,  1 ],
  [  0,  1,  1,  1,  0,  1,  1,  0 ],
  [  1,  1,  1,  1,  0,  1,  0,  1 ],
  [  1,  0,  0,  1,  1,  1,  1,  0 ],
  [  1,  0,  1,  0,  0,  0,  0,  0 ],
  [  0,  0,  0,  0,  1,  1,  1,  1 ],
  [  1,  0,  0,  0,  1,  1,  1,  1 ] ]
```

B TF Hill Cipher Implementation

B.1 tfHill.py

The following is Python code that implements the Toorani-Falahati Hill Cipher variant mentioned in section 3. Note that it requires the package `numpy`. Its usage is described in the comments.

```
'''
Toorani-Falahati Hill Variant
  (implemented by John Chase)

--------------------------------------------------------------------------------

Usage:    tfHill.encrypt( plaintext: list of any length containing lists of length n,
                          key: n x n matrix,
                          prime modulus: positive integer,
                          random a0: integer such that 0<a0<p-1,
                          random b: integer such that 1<b<n^2 )

Example:  tfHill.encrypt( [[2,3,5],[13,25,0],[9,17,21],[2,3,5],[1,15,8]],
                          [[2,4,2],[18,7,9],[11,21,6]],29,23,8)

--------------------------------------------------------------------------------

Usage:    tfHill.decrypt( ciphertext: list of any length containing lists of length n,
                          key: n x n matrix,
                          prime modulus: integer,
                          r: integer
                          b: integer such that 1<b<n^2)

Example:  tfHill.decrypt( [[0,19,9],[11,24,2],[4,13,11],[10,19,10],[8,18,27]],
                          [[2,4,2],[18,7,9],[11,21,6]],29,19,8)

--------------------------------------------------------------------------------
'''

from numpy import linalg
from numpy import matrix
import numpy
import math
import hashlib

def encrypt(X,K,p,a0,b):

  n=len(X[0])

  # First, check to make sure arguments are valid
  if not isPrime(p):
    print "Modulus p="+str(p)+" must be prime."
    return False
  if not isInvertibleMatrix(K,p):
    print "Key matrix (second argument) must be invertible mod "+str(p)+"."
    return False
  if len(X[0])!=len(K):
    print "Key matrix must have the same dimension as the plaintext block."
    return False
  if not checka0(a0,p): return False
  if not checkb(b,n): return False

  # Compute r
  i=int(math.ceil(float(b)/float(n)))
  j=b-n*(i-1)
  r=(a0*K[i-1][j-1]) % p
  print "r=" + str(r)   #debug

  Y=[]
  a=a0
  for t in range(0,len(X)):
    print "t="+str(t)
    V,v0,a=generateV(K,a,p)
    print "   v0=" + str(v0)  #debug
    print "   V=" + str(V)    #debug

    # Compute the ciphertext Y for the current plaintext X
    ctext=(v0*matrix(X[t])*matrix(K)+matrix(V))%p
    Y.append(ctext.tolist()[0])
  return Y


def decrypt(Y,K,p,r,b):

  n=len(Y[0])

  # First, check to make sure arguments are valid
  if not isPrime(p):
    print "Modulus p="+str(p)+" must be prime."
    return False
  if not isInvertibleMatrix(K,p):
    print "Key matrix (second argument) must be invertible mod "+str(p)+"."
    return False
  if len(Y[0])!=len(K):
    print "Key matrix must have the same dimension as the ciphertext block."
    return False
  if not checkb(b,n): return False

  #Compute u and a0
  i=int(math.ceil(float(b)/float(n)))
  j=b-n*(i-1)
  u=modInv(K[i-1][j-1],p)
  a0=(r*u)%p
  print "a0="+str(a0)

  X=[]
  a=a0
  for t in range(0,len(Y)):
    print "t="+str(t)
    V,v0,a=generateV(K,a,p)
    print "   v0=" + str(v0)  #debug
    print "   V=" + str(V)    #debug
    #Compute the plaintext X from the ciphertext Y
    ptext=matrix(modInv(v0,p)*(matrix(Y[t])-matrix(V))*modMatInv(matrix(K),p))%p
    X.append(ptext.tolist()[0])
  return X


def generateV(K,a0,p):        # Generates the secret matrix V based on K and a0
  n=len(K)

  # Compute the next a with the MD5 hash function applied to the previous a
  #  (easy to change this to SHA1 or another hash if one desires)
  a=int(hashlib.md5(str(a0)).hexdigest(),16)
  print "   a="+str(a)      #debug

  # Compute V
  V=[]
  if a%p==0: V.append(1)
  else: V.append(a%p)
  for i in range(1,n+1):
    j=((V[i-1])%n) + 1
    gamma=math.floor(math.log(V[i-1],2))+1
    vTilde=int(2**(math.ceil(float(gamma)/2))+(V[i-1]%(2**(math.ceil(float(gamma)/2)))))
    print "   i="+str(i)+" j="+str(j)+" gamma="+str(gamma)+" vTilde="+str(vTilde)
    V.append((K[i-1][j-1]+vTilde*a)%p)
  v0=V[0]
  del V[0]
  # Return the secret vector V, v0, and the next a,
  #  in case there are more plaintext blocks to encrypt
  return [V,v0,a]


def isPrime(n):               # Tests for primality
  for x in range(2, int(n**0.5)+1):
    if n % x == 0: return False
  return True


def modInv(a,p):              # Finds the inverse of a mod p, if it exists
  for i in range(1,p):
    if (i*a)%p==1: return i
  print str(a)+" has no inverse mod "+str(p)


def modMatInv(A,p):           # Finds the inverse of matrix A mod p
  if not isInvertibleMatrix(A,p):
    print "The first argument must be an invertible matrix mod "+str(p)+"."
    return False

  n=len(A)
  A=matrix(A)
  adj=numpy.zeros(shape=(n,n))
  for i in range(0,n):
    for j in range(0,n):
      adj[i][j]=((-1)**(i+j)*intDet(minor(A,j,i)))%p
  return (modInv(intDet(A),p)*adj)%p


# Return matrix A with the ith row and jth column deleted (indexed from 0)
def minor(A,i,j):
  if not isMatrix(A):
    print "The first argument must be a matrix."
    return False

  A=numpy.array(A)
  minor=numpy.zeros(shape=(len(A)-1,len(A)-1))
  p=0
  for s in range(0,len(minor)):
    if p==i: p=p+1
    q=0
    for t in range(0,len(minor)):
      if q==j: q=q+1
      minor[s][t]=A[p][q]
      q=q+1
    p=p+1
  return minor


def intDet(A):                # Returns the determinant of your integer-valued matrix A
  return int(round(linalg.det(A)))


def isInvertibleMatrix(A,p):  # Checks to see that A is an invertible matrix mod p
  if intDet(A)%p!=0: return True
  else: return False


def checkb(b,n):              # Check the value of b
  if b>1 and b<n**2:
    return True
    print ('The random value b=' + str(b) +
           ' must be greater than 1 and less than the square of the'
           ' length of your plaintext.\n In this case, it must be true'
           ' that 1<b<' + str(n**2) + '.')
  else: return False


def checka0(a0,p):            # Check the value of a0
  if a0>0 and a0<p-1:
    return True
    print ('The random value a0=' + str(a0) +
           ' must be greater than 0 and less than'
           ' your modulus minus one.\n In this case, it must be true'
           ' that 0<a0<' + str(p-1) + '.')
  else: return False
```

B.2 Sample output of tfHill.py

The following sample output shows all the necessary computations for the example in section 3.

B.2.1 Encryption

We enter the plaintext blocks X_t and our key K, then call the encryption function.

```
>>>X = [ [14,21,4,17,26],
     [19,7,4,26,7],
     [8,11,11,26,0],
     [13,3,26,19,7],
     [17,14,20,6,7],
     [26,19,7,4,26],
     [22,14,14,3,18] ]

>>>K=[ [1,5,17,25,16],
   [9,0,18,19,23],
   [26,4,10,13,2],
   [3,4,8,9,22],
   [7,24,15,17,12] ]

>>>Y = encrypt(X,K,29,20,17)

r=22
t=0
   a=203295115078782523880027993804846545796
   i=1 j=3 gamma=3.0 vTilde=7
   i=2 j=4 gamma=4.0 vTilde=4
   i=3 j=4 gamma=5.0 vTilde=10
   i=4 j=1 gamma=5.0 vTilde=9
   i=5 j=4 gamma=4.0 vTilde=4
   v0=7
   V=[8L, 18L, 25L, 8L, 16L]
t=1
   a=245997980935699445360930870119064451886
   i=1 j=1 gamma=5.0 vTilde=9
   i=2 j=4 gamma=5.0 vTilde=15
   i=3 j=3 gamma=5.0 vTilde=9
   i=4 j=4 gamma=2.0 vTilde=3
   i=5 j=2 gamma=5.0 vTilde=10
   v0=25
   V=[23L, 17L, 3L, 26L, 13L]
t=2
   a=323176948216200043363809235424646874509
   i=1 j=5 gamma=2.0 vTilde=2
   i=2 j=5 gamma=5.0 vTilde=8
   i=3 j=2 gamma=5.0 vTilde=10
   i=4 j=1 gamma=4.0 vTilde=7
   i=5 j=3 gamma=2.0 vTilde=2
   v0=4
   V=[24L, 26L, 15L, 2L, 23L]
t=3
   a=40821990183399004352182122979476111349
   i=1 j=3 gamma=3.0 vTilde=7
   i=2 j=4 gamma=4.0 vTilde=4
   i=3 j=4 gamma=5.0 vTilde=10
   i=4 j=1 gamma=5.0 vTilde=9
   i=5 j=4 gamma=4.0 vTilde=4
   v0=7
   V=[8L, 18L, 25L, 8L, 16L]
t=4
   a=20064375677538914896342696153730078061
   i=1 j=2 gamma=1.0 vTilde=3
   i=2 j=1 gamma=3.0 vTilde=5
   i=3 j=5 gamma=4.0 vTilde=5
   i=4 j=3 gamma=2.0 vTilde=2
   i=5 j=4 gamma=4.0 vTilde=4
   v0=1
   V=[5L, 9L, 2L, 8L, 17L]
t=5
   a=268997158588285530637903546099270153390
   i=1 j=5 gamma=5.0 vTilde=11
   i=2 j=3 gamma=5.0 vTilde=14
   i=3 j=4 gamma=5.0 vTilde=15
   i=4 j=4 gamma=4.0 vTilde=4
   i=5 j=3 gamma=5.0 vTilde=11
   v0=19
   V=[22L, 23L, 8L, 27L, 21L]
t=6
   a=152231792563035368194837115496907499393
   i=1 j=3 gamma=5.0 vTilde=9
   i=2 j=1 gamma=5.0 vTilde=9
   i=3 j=3 gamma=5.0 vTilde=9
   i=4 j=4 gamma=5.0 vTilde=10
   i=5 j=1 gamma=3.0 vTilde=5
   v0=17
   V=[25L, 17L, 18L, 5L, 5L]

[[18L, 12L, 5L, 7L, 21L], [18L, 22L, 14L, 5L, 21L],
[23L, 24L, 13L, 14L, 8L], [20L, 9L, 1L, 2L, 4L],
[10L, 18L, 26L, 1L, 26L], [5L, 18L, 8L, 3L, 24L],
[4L, 5L, 0L, 12L, 0L]]
```

The matrix above contains our ciphertext blocks Y_t.

B.2.2 Decryption

We now call the decryption function.

```
>>>Y
```

```
[[18L, 12L, 5L, 7L, 21L], [18L, 22L, 14L, 5L, 21L],
[23L, 24L, 13L, 14L, 8L], [20L, 9L, 1L, 2L, 4L],
[10L, 18L, 26L, 1L, 26L], [5L, 18L, 8L, 3L, 24L],
[4L, 5L, 0L, 12L, 0L]]

>>>decrypt(Y,K,29,22,17)

a0=20
t=0
   a=20329511507878252388002799380484654579 6
   i=1 j=3 gamma=3.0 vTilde=7
   i=2 j=4 gamma=4.0 vTilde=4
   i=3 j=4 gamma=5.0 vTilde=10
   i=4 j=1 gamma=5.0 vTilde=9
   i=5 j=4 gamma=4.0 vTilde=4
   v0=7
   V=[8L, 18L, 25L, 8L, 16L]
t=1
   a=24599798093569944536093087011906445188 6
   i=1 j=1 gamma=5.0 vTilde=9
   i=2 j=4 gamma=5.0 vTilde=15
   i=3 j=3 gamma=5.0 vTilde=9
   i=4 j=4 gamma=2.0 vTilde=3
   i=5 j=2 gamma=5.0 vTilde=10
   v0=25
   V=[23L, 17L, 3L, 26L, 13L]
t=2
   a=32317694821620004336380923542464687450 9
   i=1 j=5 gamma=2.0 vTilde=2
   i=2 j=5 gamma=5.0 vTilde=8
   i=3 j=2 gamma=5.0 vTilde=10
   i=4 j=1 gamma=4.0 vTilde=7
   i=5 j=3 gamma=2.0 vTilde=2
   v0=4
   V=[24L, 26L, 15L, 2L, 23L]
t=3
   a=4082199018339900435218212297947611134 9
   i=1 j=3 gamma=3.0 vTilde=7
   i=2 j=4 gamma=4.0 vTilde=4
   i=3 j=4 gamma=5.0 vTilde=10
   i=4 j=1 gamma=5.0 vTilde=9
   i=5 j=4 gamma=4.0 vTilde=4
   v0=7
   V=[8L, 18L, 25L, 8L, 16L]
t=4
   a=2006437567753891489634269615373007806 1
   i=1 j=2 gamma=1.0 vTilde=3
   i=2 j=1 gamma=3.0 vTilde=5
   i=3 j=5 gamma=4.0 vTilde=5
   i=4 j=3 gamma=2.0 vTilde=2
   i=5 j=4 gamma=4.0 vTilde=4
   v0=1
   V=[5L, 9L, 2L, 8L, 17L]
```

```
t=5
   a=268997158588285530637903546099270153390
   i=1 j=5 gamma=5.0 vTilde=11
   i=2 j=3 gamma=5.0 vTilde=14
   i=3 j=4 gamma=5.0 vTilde=15
   i=4 j=4 gamma=4.0 vTilde=4
   i=5 j=3 gamma=5.0 vTilde=11
   v0=19
   V=[22L, 23L, 8L, 27L, 21L]
t=6
   a=152231792563035368194837115496907499393
   i=1 j=3 gamma=5.0 vTilde=9
   i=2 j=1 gamma=5.0 vTilde=9
   i=3 j=3 gamma=5.0 vTilde=9
   i=4 j=4 gamma=5.0 vTilde=10
   i=5 j=1 gamma=3.0 vTilde=5
   v0=17
   V=[25L, 17L, 18L, 5L, 5L]

[[14.0, 21.0, 4.0, 17.0, 26.0], [19.0, 7.0, 4.0, 26.0, 7.0],
[8.0, 11.0, 11.0, 26.0, -0.0], [13.0, 3.0, 26.0, 19.0, 7.0],
[17.0, 14.0, 20.0, 6.0, 7.0], [26.0, 19.0, 7.0, 4.0, 26.0],
[22.0, 14.0, 14.0, 3.0, 18.0]]
```

The decryption function has recovered all the plaintext blocks X_t.