Vernam Two

The author’s email: dmilleville@Comcast.net

- This document is a PDF of the PowerPoint presentation that is to be presented where and when requested.
- It contains all information that can physically be included within this presentation document concerning this design.
- Other information can be furnished during a presentation that proves the methodology exists and produces what is claimed.
- This is a bona-fide modification/addition to an existing long-standing cryptographic algorithm combined, for the first time, with Algebraic law to produce a commercial version of a faster and more secure system than the AES.

Introducing a significant improvement over the current AES Standard

1. At least a 4-fold performance improvement as compared to the AES.
2. Ability to decrypt individual characters of plaintext without having to decrypt an entire block. When coupled with the performance improvement, this will vastly improve data searching throughput of sensitive protected databases.
3. No loss of security – mathematical proof is provided in this presentation.
4. No more 'Mode Of Operations' – No external data, counter, table or extra data stream needed for an unpredictably changing output – all data needed to decrypt the unpredictable encryption is encrypted along with the plaintext contained within the ciphertext file using the same encryption methodology.
5. Requires access to an approved Random Number Generator for the first block only.
6. The only 'mode' this design has, produces a virtually endless number of almost completely different ciphertext files, even if it repeatedly encrypts the same plaintext.
7. Can produce 10 billion+ different ciphertext files from any single plaintext input with no external data, count or stream needed.

What is the comparison of the AES to this proposed cipher design?

Point of Consideration	256-bit AES	Proposed cipher design
Input Key size	**256 bits**	**256 bits**
Time to encrypt a 15.8 Mbyte file	**62.8 seconds**	**12 seconds**
Security	**The ‘Standard’**	**Mathematical proof is provided that it is at least equal to The ‘Standard’**
Additional data and/or information needed for proper encryption or decryption to occur for most Modes Of Operation	**Provided/delivered external to the ciphertext, a possible security issue**	**No MOO, all data needed is encrypted within the ciphertext using the same encryption methodology**
When the user needs 1 or more characters from the ciphertext when searching for an SS or credit #, how much work is involved?	**The entire block has to be decrypted before access is provided for one character**	**Individual characters from the ciphertext can be decrypted without processing the entire block**

How can the speed increase with no loss of security?

- The AES relies on repeated mathematical processing of the entire block to provide the security required. This results in an average of 245 computer steps executed per character (Visual Basic version of the AES).
- The speed increase in this design is the result of using a combination of a well known cryptographic algorithm plus Algebraic law, involving only 2 steps per character as detailed in this presentation. Repetitious processing bogs down the process and allows for possible attacks.
- With significantly fewer steps to take per character, there is a very significant improvement in execution speed.

Key requirements and methodology for construction

- Key storage will be discussed later in this presentation.
- The AES's 'gkey' function was expanded to produce a base array of 2,097,184 (0 to 20001Fh) pseudo-random long words from the input 256-bit key.
- The 8,388,736 (0 to 80007Fh) byte main key this design uses is created by extracting 4 bytes from each base array long word.
- Two chain keys, 8,388,608 (0 to 7FFFFFh) long words each, are also created using the base array as the initializer and construction 'director'.
- The function of a chain key and the methodology used to construct this key is illustrated next.

What is the makeup and function of a ‘chain key’?

- The key array contains all numbers within a stated range, access chained into a single loop pseudo-randomly. An example of a chain key using 0 through 9:

  chn(0)=4, chn(4)=7, chn(7)=3, chn(3)=9, chn(9)=2,
  chn(2)=5, chn(5)=6, chn(6)=1, chn(1)=8, chn(8)=0

- The function of the key is to use all numbers only once within the effective range beginning anywhere when accessing all locations as above within the key array. In the above case, 0 through 9, in pseudo-random order.

What is the second ‘chain key’?

- The second chain key is the first key in the reverse chain direction. Here’s the ‘forward’ chain example from the previous slide:

  chn(0)=4, chn(4)=7, chn(7)=3, chn(3)=9, chn(9)=2,
  chn(2)=5, chn(5)=6, chn(6)=1, chn(1)=8, chn(8)=0

- Here is the same chain key in reverse:

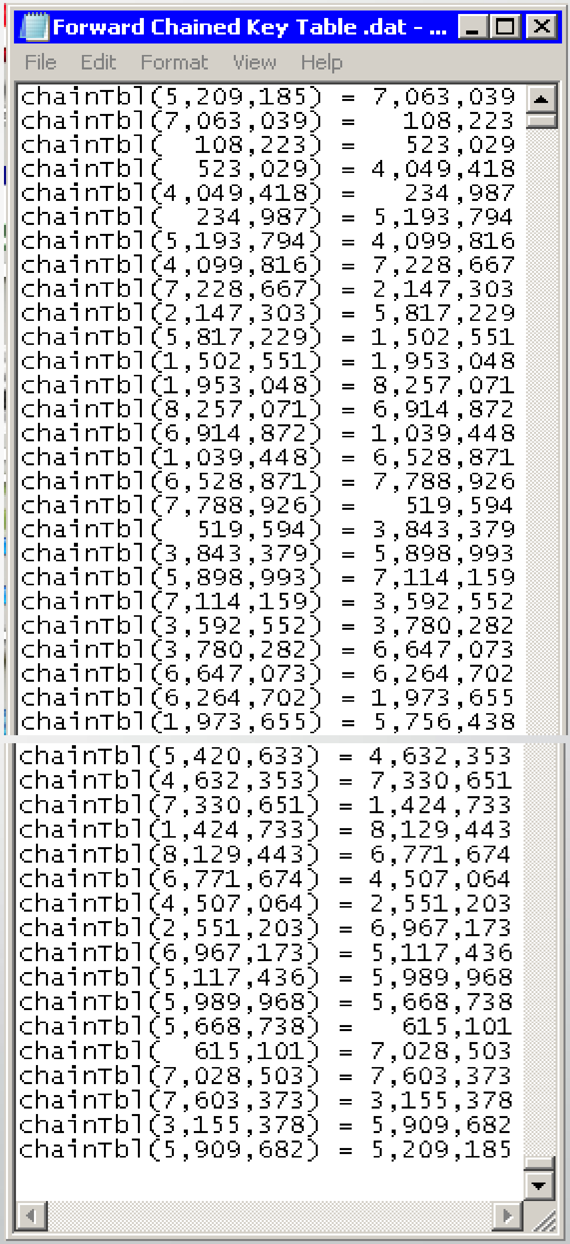
  chn(0)=8, chn(8)=1, chn(1)=6, chn(6)=5, chn(5)=2,
  chn(2)=9, chn(9)=3, chn(3)=7, chn(7)=4, chn(4)=0

What are the sizes of the ‘chain keys’ and how are they used in this design?

- Both of this cipher engine’s chain keys are 8,388,608 (0 to 7FFFFFh) long words.
- After 4 array pointers used in this methodology are randomly initialized using the PRNG for the first block only, these pointers are advanced for subsequent blocks using the first chain key to change their reference into the main key.
- Because the pointers use the chain key, a total of 8,388,608 sets of non-repeated pointers are created for up to that number of blocks. You will see why these pointers must not repeat later.
- The second chain key is used in the process to encrypt the starting pointers for the decrypt engine’s use.

An actual chain table

- Pictured on the right is a randomly selected start and end point of the 8+ million chain table used in the current demonstration application, illustrating how the chain is used, starting and ending at the randomly selected point in the key, address 5,209,185.
- The file pictured is 270+ Mbytes in size so this is why only the beginning and ending of the file are illustrated. Searching for the starting address 5,209,185 is found in only 2 places, the start and end as pictured. Notice the scroll bars show the segments shown are at the start and end.
- Searching for ANY other address results in only two adjacent lines containing the address searched. For example, searching for 6,914,872 occurs in only the two adjacent lines indicated in the entire file.

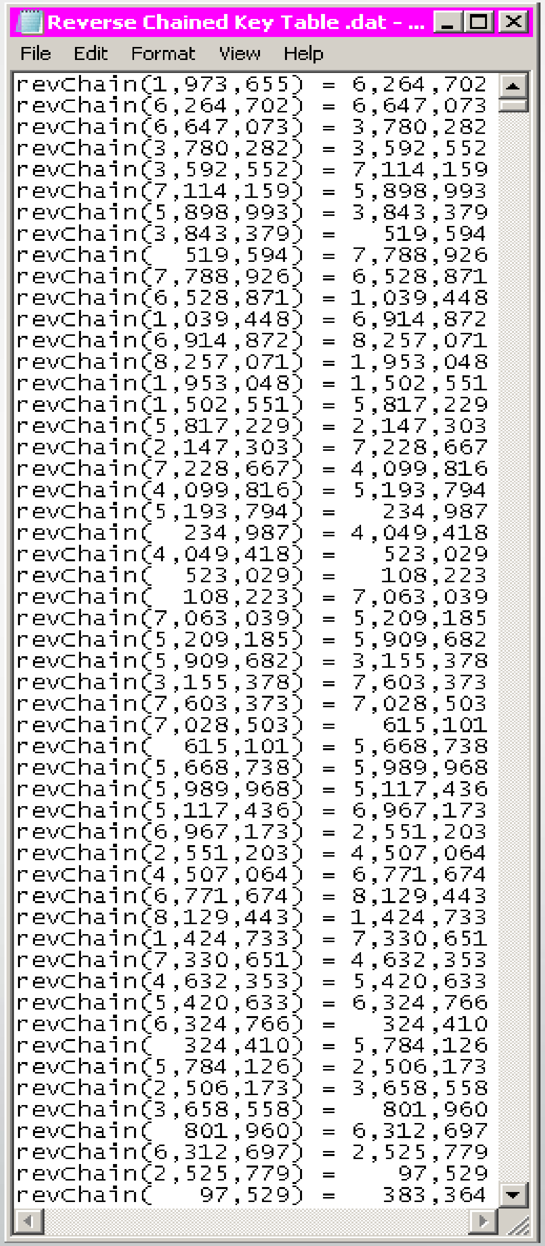


```
chainTbl(5,209,185) = 7,063,039
chainTbl(7,063,039) =   108,223
chainTbl(  108,223) =   523,029
chainTbl(  523,029) = 4,049,418
chainTbl(4,049,418) =   234,987
chainTbl(  234,987) = 5,193,794
chainTbl(5,193,794) = 4,099,816
chainTbl(4,099,816) = 7,228,667
chainTbl(7,228,667) = 2,147,303
chainTbl(2,147,303) = 5,817,229
chainTbl(5,817,229) = 1,502,551
chainTbl(1,502,551) = 1,953,048
chainTbl(1,953,048) = 8,257,071
chainTbl(8,257,071) = 6,914,872
chainTbl(6,914,872) = 1,039,448
chainTbl(1,039,448) = 6,528,871
chainTbl(6,528,871) = 7,788,926
chainTbl(7,788,926) =   519,594
chainTbl(  519,594) = 3,843,379
chainTbl(3,843,379) = 5,898,993
chainTbl(5,898,993) = 7,114,159
chainTbl(7,114,159) = 3,592,552
chainTbl(3,592,552) = 3,780,282
chainTbl(3,780,282) = 6,647,073
chainTbl(6,647,073) = 6,264,702
chainTbl(6,264,702) = 1,973,655
chainTbl(1,973,655) = 5,756,438
```

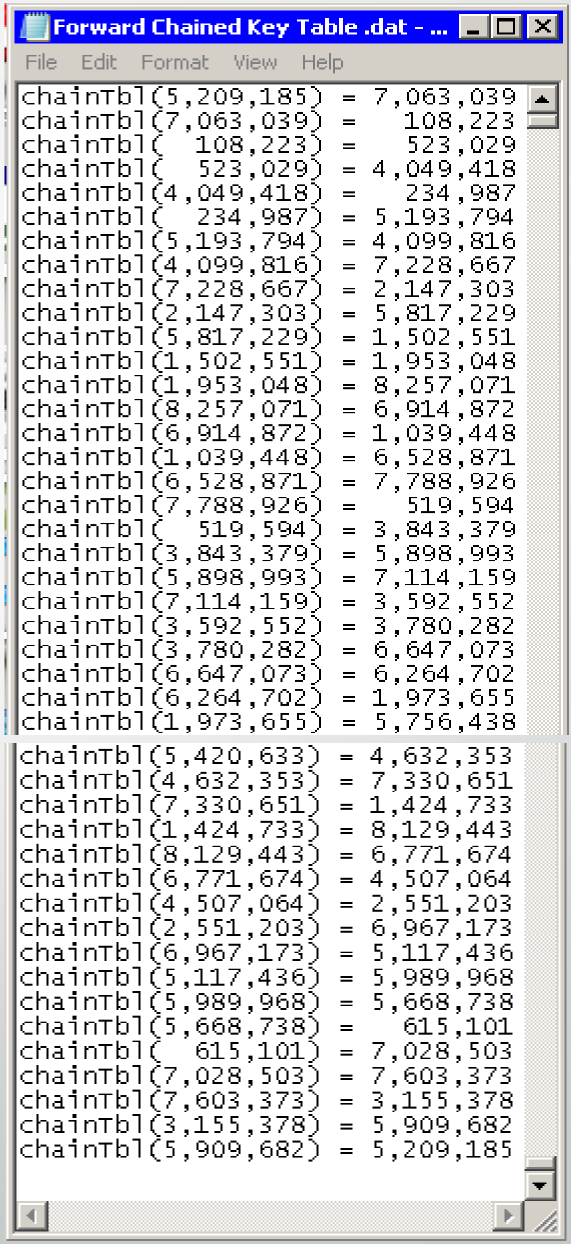
```
chainTbl(5,420,633) = 4,632,353
chainTbl(4,632,353) = 7,330,651
chainTbl(7,330,651) = 1,424,733
chainTbl(1,424,733) = 8,129,443
chainTbl(8,129,443) = 6,771,674
chainTbl(6,771,674) = 4,507,064
chainTbl(4,507,064) = 2,551,203
chainTbl(2,551,203) = 6,967,173
chainTbl(6,967,173) = 5,117,436
chainTbl(5,117,436) = 5,989,968
chainTbl(5,989,968) = 5,668,738
chainTbl(5,668,738) =   615,101
chainTbl(  615,101) = 7,028,503
chainTbl(7,028,503) = 7,603,373
chainTbl(7,603,373) = 3,155,378
chainTbl(3,155,378) = 5,909,682
chainTbl(5,909,682) = 5,209,185
```

A reverse chain table

- On the near right is a reverse chain table beginning at the last address on the top portion of the forward chain table, address 1,973,655.
- If you follow it down, it matches the reverse sequence of the forward table right through the ending.



```
revChain(1,973,655) = 6,264,702
revChain(6,264,702) = 6,647,073
revChain(6,647,073) = 3,780,282
revChain(3,780,282) = 3,592,552
revChain(3,592,552) = 7,114,159
revChain(7,114,159) = 5,898,993
revChain(5,898,993) = 3,843,379
revChain(3,843,379) =   519,594
revChain(  519,594) = 7,788,926
revChain(7,788,926) = 6,528,871
revChain(6,528,871) = 1,039,448
revChain(1,039,448) = 6,914,872
revChain(6,914,872) = 8,257,071
revChain(8,257,071) = 1,953,048
revChain(1,953,048) = 1,502,551
revChain(1,502,551) = 5,817,229
revChain(5,817,229) = 2,147,303
revChain(2,147,303) = 7,228,667
revChain(7,228,667) = 4,099,816
revChain(4,099,816) = 5,193,794
revChain(5,193,794) =   234,987
revChain(  234,987) = 4,049,418
revChain(4,049,418) =   523,029
revChain(  523,029) =   108,223
revChain(  108,223) = 7,063,039
revChain(7,063,039) = 5,209,185
revChain(5,209,185) = 5,909,682
revChain(5,909,682) = 3,155,378
revChain(3,155,378) = 7,603,373
revChain(7,603,373) = 7,028,503
revChain(7,028,503) =   615,101
revChain(  615,101) = 5,668,738
revChain(5,668,738) = 5,989,968
revChain(5,989,968) = 5,117,436
revChain(5,117,436) = 6,967,173
revChain(6,967,173) = 2,551,203
revChain(2,551,203) = 4,507,064
revChain(4,507,064) = 6,771,674
revChain(6,771,674) = 8,129,443
revChain(8,129,443) = 1,424,733
revChain(1,424,733) = 7,330,651
revChain(7,330,651) = 4,632,353
revChain(4,632,353) = 5,420,633
revChain(5,420,633) = 6,324,766
revChain(6,324,766) =   324,410
revChain(  324,410) = 5,784,126
revChain(5,784,126) = 2,506,173
revChain(2,506,173) = 3,658,558
revChain(3,658,558) =   801,960
revChain(  801,960) = 6,312,697
revChain(6,312,697) = 2,525,779
revChain(2,525,779) =    97,529
revChain(   97,529) =   383,364
```



```
chainTbl(5,209,185) = 7,063,039
chainTbl(7,063,039) =   108,223
chainTbl(  108,223) =   523,029
chainTbl(  523,029) = 4,049,418
chainTbl(4,049,418) =   234,987
chainTbl(  234,987) = 5,193,794
chainTbl(5,193,794) = 4,099,816
chainTbl(4,099,816) = 7,228,667
chainTbl(7,228,667) = 2,147,303
chainTbl(2,147,303) = 5,817,229
chainTbl(5,817,229) = 1,502,551
chainTbl(1,502,551) = 1,953,048
chainTbl(1,953,048) = 8,257,071
chainTbl(8,257,071) = 6,914,872
chainTbl(6,914,872) = 1,039,448
chainTbl(1,039,448) = 6,528,871
chainTbl(6,528,871) = 7,788,926
chainTbl(7,788,926) =   519,594
chainTbl(  519,594) = 3,843,379
chainTbl(3,843,379) = 5,898,993
chainTbl(5,898,993) = 7,114,159
chainTbl(7,114,159) = 3,592,552
chainTbl(3,592,552) = 3,780,282
chainTbl(3,780,282) = 6,647,073
chainTbl(6,647,073) = 6,264,702
chainTbl(6,264,702) = 1,973,655
chainTbl(1,973,655) = 5,756,438
```

```
chainTbl(5,420,633) = 4,632,353
chainTbl(4,632,353) = 7,330,651
chainTbl(7,330,651) = 1,424,733
chainTbl(1,424,733) = 8,129,443
chainTbl(8,129,443) = 6,771,674
chainTbl(6,771,674) = 4,507,064
chainTbl(4,507,064) = 2,551,203
chainTbl(2,551,203) = 6,967,173
chainTbl(6,967,173) = 5,117,436
chainTbl(5,117,436) = 5,989,968
chainTbl(5,989,968) = 5,668,738
chainTbl(5,668,738) =   615,101
chainTbl(  615,101) = 7,028,503
chainTbl(7,028,503) = 7,603,373
chainTbl(7,603,373) = 3,155,378
chainTbl(3,155,378) = 5,909,682
chainTbl(5,909,682) = 5,209,185
```

Constructing an 8 million long word chain key from only 2 million numbers

- The absolute value of a base array location is selected and the value Mod 8,388,608 (800000h) is used as a ‘start-load-at’ number.
- A source array of 8,388,608 (0 to 7FFFFFh) long words is loaded starting at position 0 loading the ‘start-load-at’ value and loading the locations with a round-robin incremented value to complete the load.
- Within the source array, every value from 0 to 8,388,607 inclusive is recorded only once.
- The build function then loops through the base array.

Constructing an 8 million long word chain key from only 2 million numbers

- If the absolute number in the source array within this loop at the base array pointer has not been used, it is transferred to the chain key array in the location ‘previous-value’.
- The number loaded becomes the new ‘previous value’ location, the number in the source array is flagged ‘used’.
- The location in the reverse chain key array is initialized by using the address as the data and the data as the address.
- Every time the loop completes using the base array, the source array is cleared of ‘used’ locations.

Constructing an 8 million long word chain key from only 2 million numbers

- The number of available values is used to Mod the value from the base array during the next loop through the source array.
- The base array is reused as many times as needed until the chain key array is fully constructed.
- When the chain key array has been completely loaded from the source array, the saved ‘starting-initial-value’, set at the start of construction, is transferred to the location indicated in ‘previous-value’ to close the chain, and the reverse chain key array is also closed using the reverse set of data and address.

Here’s the AES Visual Basic Encryption Code

- To calculate ‘Y(j)’, this code executes 70 steps.

```
For i = 1 To m_Nr - 1
    For j = 0 To m_Nb - 1
        m = j * 3
        Y(j) = m_ekey(k) Xor m_etable(X(j) And &HFF&) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m)), 8) And &HFF&), 8) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m + 1)), 16) And &HFF&), 16) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m + 2)), 24) And &HFF&), 24)
        k = k + 1
    Next
    t = X
    X = Y
    Y = t
Next
```

- If you would like to see proof of the 70 steps, it can be shown after this presentation.

Here's the AES Visual Basic Encryption Code

- The inner loop executes 8 times. 70 x 8 = 560 steps

```
For i = 1 To m_Nr - 1
    For j = 0 To m_Nb - 1
        m = j * 3
        Y(j) = m_ekey(k) Xor m_etable(X(j) And &HFF&) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m)), 8) And &HFF&), 8) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m + 1)), 16) And &HFF&), 16) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m + 2)), 24) And &HFF&), 24)
        k = k + 1
    Next
    t = X
    X = Y
    Y = t
Next
```

Here’s the AES Visual Basic Encryption Code

- The outer loop 13 times. 560 x 13 = 7,280 steps.

```
For i = 1 To m_Nr - 1
    For j = 0 To m_Nb - 1
        m = j * 3
        Y(j) = m_ekey(k) Xor m_etable(X(j) And &HFF&) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m)), 8) And &HFF&), 8) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m + 1)), 16) And &HFF&), 16) Xor _
            RotateLeft(m_etable(RShift(X(m_fi(m + 2)), 24) And &HFF&), 24)
        k = k + 1
    Next
    t = X
    X = Y
    Y = t
Next
```

Here’s the AES Visual Basic Encryption Code

- This 8-step loop executes once at the end of the encryption sequence for the block.
- 7,280 + (8 x 70)= 7,840

```
For j = 0 To m_Nb - 1
     m = j * 3
     Y(j) = m_ekey(k) Xor m_fbsub(X(j) And &HFF&) Xor _
          RotateLeft(m_fbsub(RShift(X(m_fi(m)), 8) And &HFF&), 8) Xor _
          RotateLeft(m_fbsub(RShift(X(m_fi(m + 1)), 16) And &HFF&), 16) Xor _
          RotateLeft(m_fbsub(RShift(X(m_fi(m + 2)), 24) And &HFF&), 24)
     k = k + 1
Next
```

The proposed cipher processes 128 characters per block

- AES takes 7,840 steps to encrypt 32 characters
- This cipher design encrypts 128 characters per block or 4 blocks of AES plaintext.
- 4 x 7,840 = 31,360 steps to encrypt 128 characters of plaintext for the AES.
- This is deliberately conservative as the single instructions in blue either side of the main instruction are not counted.

This cryptographic engine's Visual Basic code

- The 'key' is the 8,388,736 byte (0 to 80007Fh) key constructed by the gkey function.
- The 'Ptrx' pointers are initially randomly set between 0 and 8,366,607 inclusive by the PRNG during block 1 and modified by the chain key for each succeeding block.
- The 'str1' is the string holder that will contain the ciphertext or plaintext block characters.
- The 'str2' is the string holder that contains the plaintext or ciphertext block characters.

This cryptographic engine’s Visual Basic code

- This loop executes 2 steps for each of 128 characters:

```
For i = 0 To 127
    str1 = str1 + chr$(Asc(Mid$(str2, i + 1, 1)) Xor _
        key(Ptr1 + i) Xor key(Ptr2 + i) Xor _
        key(Ptr3 + i) Xor key(Ptr4 + i))
Next i
```

- Notice there are only table references, not functions called, to obtain the values to Xor together.
- How does this compare to the 31,360 steps (245 steps for each character) of the AES encryption for the same 128 plaintext characters?

What happens after the first block?

- After the first and subsequent blocks are processed and the engine is about to encrypt the next block, each pointer accesses the chain key. The pointers are all reset to different reference points within the main key.
- Even if only one pointer was changed by 1, the EKS would be almost entirely different – this can be demonstrated.
- Since all 4 pointers will change to constantly pseudo-different values, the EKS will be a non-repeating stream through the 8 million+ block size of the chain key.

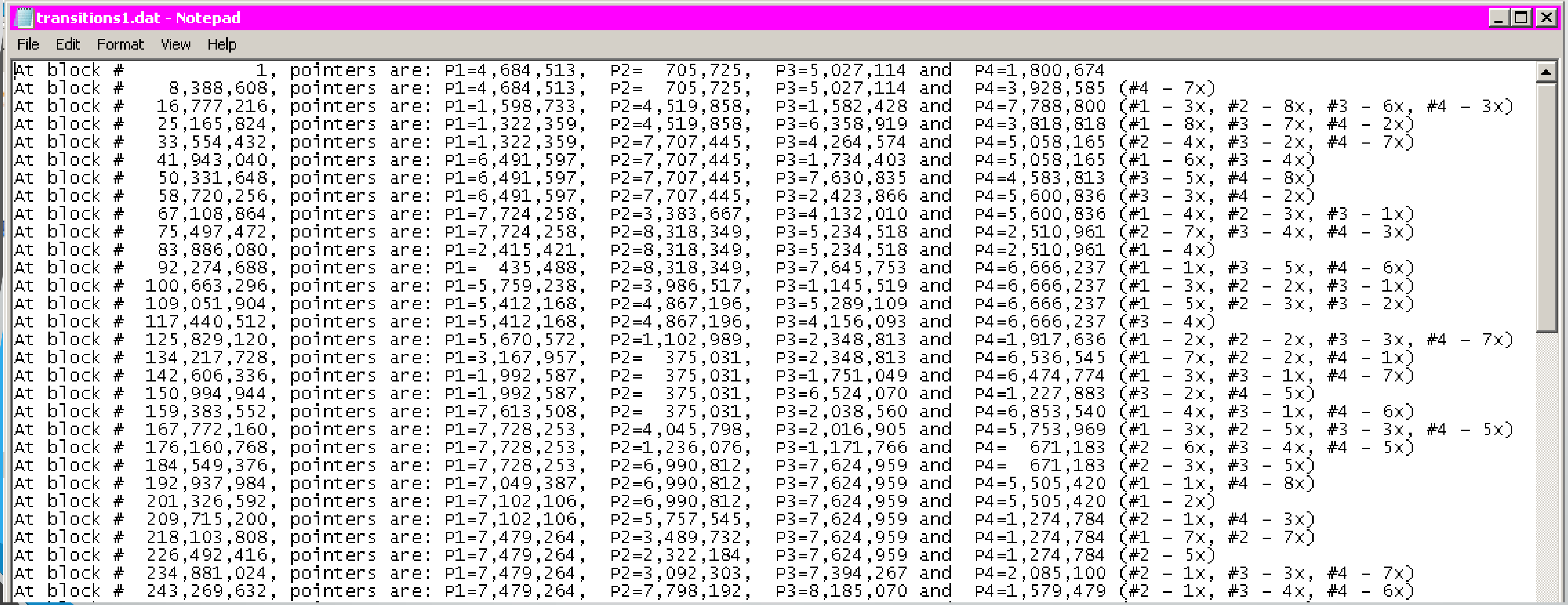
Does any attacker have any Possibility of reconstructing the entire key?

- Unlike most other ciphers, it is impossible to reconstruct the entire key if it were possible to determine the key streams used for one block.
- 4 streams of 128 bytes used per block = 512 bytes of the 8,388,736 byte key.
- Even if they could reconstruct the 512 bytes, they would have less than 0.007% of the entire 8,388,736 byte key, not to mention a critical failure of where those streams should be placed in the 8 Mbyte array.

What if the number of blocks exceeds 8,388,607 (1.73 Gbytes of plaintext)?

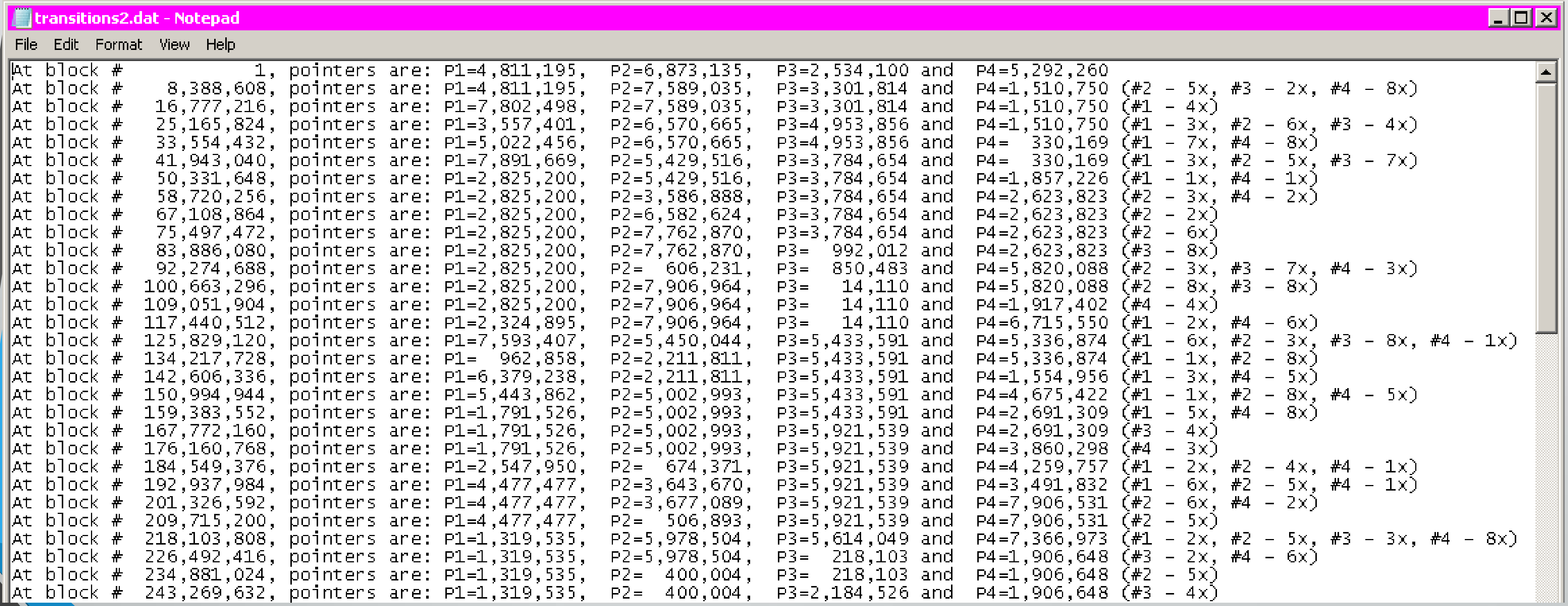
- The four pointers are Xor’ed together, result is then Mod 15.
- The result selects which set of 4 pointers, 1, 2, 3 or all 4, are to be additionally advanced, 15 possible combinations.
- For each pointer being additionally advanced, the location at the initial address of that pointer is Mod 8 + 1.
- Each pointer selected is then advanced using the chain key that number of times.
- For subsequent encryptions of large files, the set of pointers modified changes because the initial pointers are randomly set and may never be the same.
- On the next 2 slides are examples of advancements done.

An example of the pointer advancements:



```
At block #           1, pointers are: P1=4,684,513,  P2=  705,725,  P3=5,027,114 and  P4=1,800,674
At block #   8,388,608, pointers are: P1=4,684,513,  P2=  705,725,  P3=5,027,114 and  P4=3,928,585 (#4 - 7x)
At block #  16,777,216, pointers are: P1=1,598,733,  P2=4,519,858,  P3=1,582,428 and  P4=7,788,800 (#1 - 3x, #2 - 8x, #3 - 6x, #4 - 3x)
At block #  25,165,824, pointers are: P1=1,322,359,  P2=4,519,858,  P3=6,358,919 and  P4=3,818,818 (#1 - 8x, #3 - 7x, #4 - 2x)
At block #  33,554,432, pointers are: P1=1,322,359,  P2=7,707,445,  P3=4,264,574 and  P4=5,058,165 (#2 - 4x, #3 - 2x, #4 - 7x)
At block #  41,943,040, pointers are: P1=6,491,597,  P2=7,707,445,  P3=1,734,403 and  P4=5,058,165 (#1 - 6x, #3 - 4x)
At block #  50,331,648, pointers are: P1=6,491,597,  P2=7,707,445,  P3=7,630,835 and  P4=4,583,813 (#3 - 5x, #4 - 8x)
At block #  58,720,256, pointers are: P1=6,491,597,  P2=7,707,445,  P3=2,423,866 and  P4=5,600,836 (#3 - 3x, #4 - 2x)
At block #  67,108,864, pointers are: P1=7,724,258,  P2=3,383,667,  P3=4,132,010 and  P4=5,600,836 (#1 - 4x, #2 - 3x, #3 - 1x)
At block #  75,497,472, pointers are: P1=7,724,258,  P2=8,318,349,  P3=5,234,518 and  P4=2,510,961 (#2 - 7x, #3 - 4x, #4 - 3x)
At block #  83,886,080, pointers are: P1=2,415,421,  P2=8,318,349,  P3=5,234,518 and  P4=2,510,961 (#1 - 4x)
At block #  92,274,688, pointers are: P1=  435,488,  P2=8,318,349,  P3=7,645,753 and  P4=6,666,237 (#1 - 1x, #3 - 5x, #4 - 6x)
At block # 100,663,296, pointers are: P1=5,759,238,  P2=3,986,517,  P3=1,145,519 and  P4=6,666,237 (#1 - 3x, #2 - 2x, #3 - 1x)
At block # 109,051,904, pointers are: P1=5,412,168,  P2=4,867,196,  P3=5,289,109 and  P4=6,666,237 (#1 - 5x, #2 - 3x, #3 - 2x)
At block # 117,440,512, pointers are: P1=5,412,168,  P2=4,867,196,  P3=4,156,093 and  P4=6,666,237 (#3 - 4x)
At block # 125,829,120, pointers are: P1=5,670,572,  P2=1,102,989,  P3=2,348,813 and  P4=1,917,636 (#1 - 2x, #2 - 2x, #3 - 3x, #4 - 7x)
At block # 134,217,728, pointers are: P1=3,167,957,  P2=  375,031,  P3=2,348,813 and  P4=6,536,545 (#1 - 7x, #2 - 2x, #4 - 1x)
At block # 142,606,336, pointers are: P1=1,992,587,  P2=  375,031,  P3=1,751,049 and  P4=6,474,774 (#1 - 3x, #3 - 1x, #4 - 7x)
At block # 150,994,944, pointers are: P1=1,992,587,  P2=  375,031,  P3=6,524,070 and  P4=1,227,883 (#3 - 2x, #4 - 5x)
At block # 159,383,552, pointers are: P1=7,613,508,  P2=  375,031,  P3=2,038,560 and  P4=6,853,540 (#1 - 4x, #3 - 1x, #4 - 6x)
At block # 167,772,160, pointers are: P1=7,728,253,  P2=4,045,798,  P3=2,016,905 and  P4=5,753,969 (#1 - 3x, #2 - 5x, #3 - 3x, #4 - 5x)
At block # 176,160,768, pointers are: P1=7,728,253,  P2=1,236,076,  P3=1,171,766 and  P4=  671,183 (#2 - 6x, #3 - 4x, #4 - 5x)
At block # 184,549,376, pointers are: P1=7,728,253,  P2=6,990,812,  P3=7,624,959 and  P4=  671,183 (#2 - 3x, #3 - 5x)
At block # 192,937,984, pointers are: P1=7,049,387,  P2=6,990,812,  P3=7,624,959 and  P4=5,505,420 (#1 - 1x, #4 - 8x)
At block # 201,326,592, pointers are: P1=7,102,106,  P2=6,990,812,  P3=7,624,959 and  P4=5,505,420 (#1 - 2x)
At block # 209,715,200, pointers are: P1=7,102,106,  P2=5,757,545,  P3=7,624,959 and  P4=1,274,784 (#2 - 1x, #4 - 3x)
At block # 218,103,808, pointers are: P1=7,479,264,  P2=3,489,732,  P3=7,624,959 and  P4=1,274,784 (#1 - 7x, #2 - 7x)
At block # 226,492,416, pointers are: P1=7,479,264,  P2=2,322,184,  P3=7,624,959 and  P4=1,274,784 (#2 - 5x)
At block # 234,881,024, pointers are: P1=7,479,264,  P2=3,092,303,  P3=7,394,267 and  P4=2,085,100 (#2 - 1x, #3 - 3x, #4 - 7x)
At block # 243,269,632, pointers are: P1=7,479,264,  P2=7,798,192,  P3=8,185,070 and  P4=1,579,479 (#2 - 1x, #3 - 4x, #4 - 6x)
```

A second example of the pointer advancements:



```
At block #           1, pointers are: P1=4,811,195,  P2=6,873,135,  P3=2,534,100 and  P4=5,292,260
At block #   8,388,608, pointers are: P1=4,811,195,  P2=7,589,035,  P3=3,301,814 and  P4=1,510,750 (#2 - 5x, #3 - 2x, #4 - 8x)
At block #  16,777,216, pointers are: P1=7,802,498,  P2=7,589,035,  P3=3,301,814 and  P4=1,510,750 (#1 - 4x)
At block #  25,165,824, pointers are: P1=3,557,401,  P2=6,570,665,  P3=4,953,856 and  P4=1,510,750 (#1 - 3x, #2 - 6x, #3 - 4x)
At block #  33,554,432, pointers are: P1=5,022,456,  P2=6,570,665,  P3=4,953,856 and  P4=  330,169 (#1 - 7x, #4 - 8x)
At block #  41,943,040, pointers are: P1=7,891,669,  P2=5,429,516,  P3=3,784,654 and  P4=  330,169 (#1 - 3x, #2 - 5x, #3 - 7x)
At block #  50,331,648, pointers are: P1=2,825,200,  P2=5,429,516,  P3=3,784,654 and  P4=1,857,226 (#1 - 1x, #4 - 1x)
At block #  58,720,256, pointers are: P1=2,825,200,  P2=3,586,888,  P3=3,784,654 and  P4=2,623,823 (#2 - 3x, #4 - 2x)
At block #  67,108,864, pointers are: P1=2,825,200,  P2=6,582,624,  P3=3,784,654 and  P4=2,623,823 (#2 - 2x)
At block #  75,497,472, pointers are: P1=2,825,200,  P2=7,762,870,  P3=3,784,654 and  P4=2,623,823 (#2 - 6x)
At block #  83,886,080, pointers are: P1=2,825,200,  P2=7,762,870,  P3=  992,012 and  P4=2,623,823 (#3 - 8x)
At block #  92,274,688, pointers are: P1=2,825,200,  P2=  606,231,  P3=  850,483 and  P4=5,820,088 (#2 - 3x, #3 - 7x, #4 - 3x)
At block # 100,663,296, pointers are: P1=2,825,200,  P2=7,906,964,  P3=   14,110 and  P4=5,820,088 (#2 - 8x, #3 - 8x)
At block # 109,051,904, pointers are: P1=2,825,200,  P2=7,906,964,  P3=   14,110 and  P4=1,917,402 (#4 - 4x)
At block # 117,440,512, pointers are: P1=2,324,895,  P2=7,906,964,  P3=   14,110 and  P4=6,715,550 (#1 - 2x, #4 - 6x)
At block # 125,829,120, pointers are: P1=7,593,407,  P2=5,450,044,  P3=5,433,591 and  P4=5,336,874 (#1 - 6x, #2 - 3x, #3 - 8x, #4 - 1x)
At block # 134,217,728, pointers are: P1=  962,858,  P2=2,211,811,  P3=5,433,591 and  P4=5,336,874 (#1 - 1x, #2 - 8x)
At block # 142,606,336, pointers are: P1=6,379,238,  P2=2,211,811,  P3=5,433,591 and  P4=1,554,956 (#1 - 3x, #4 - 5x)
At block # 150,994,944, pointers are: P1=5,443,862,  P2=5,002,993,  P3=5,433,591 and  P4=4,675,422 (#1 - 1x, #2 - 8x, #4 - 5x)
At block # 159,383,552, pointers are: P1=1,791,526,  P2=5,002,993,  P3=5,433,591 and  P4=2,691,309 (#1 - 5x, #4 - 8x)
At block # 167,772,160, pointers are: P1=1,791,526,  P2=5,002,993,  P3=5,921,539 and  P4=2,691,309 (#3 - 4x)
At block # 176,160,768, pointers are: P1=1,791,526,  P2=5,002,993,  P3=5,921,539 and  P4=3,860,298 (#4 - 3x)
At block # 184,549,376, pointers are: P1=2,547,950,  P2=  674,371,  P3=5,921,539 and  P4=4,259,757 (#1 - 2x, #2 - 4x, #4 - 1x)
At block # 192,937,984, pointers are: P1=4,477,477,  P2=3,643,670,  P3=5,921,539 and  P4=3,491,832 (#1 - 6x, #2 - 5x, #4 - 1x)
At block # 201,326,592, pointers are: P1=4,477,477,  P2=3,677,089,  P3=5,921,539 and  P4=7,906,531 (#2 - 6x, #4 - 2x)
At block # 209,715,200, pointers are: P1=4,477,477,  P2=  506,893,  P3=5,921,539 and  P4=7,906,531 (#2 - 5x)
At block # 218,103,808, pointers are: P1=1,319,535,  P2=5,978,504,  P3=5,614,049 and  P4=7,366,973 (#1 - 2x, #2 - 5x, #3 - 3x, #4 - 8x)
At block # 226,492,416, pointers are: P1=1,319,535,  P2=5,978,504,  P3=  218,103 and  P4=1,906,648 (#3 - 2x, #4 - 6x)
At block # 234,881,024, pointers are: P1=1,319,535,  P2=  400,004,  P3=  218,103 and  P4=1,906,648 (#2 - 5x)
At block # 243,269,632, pointers are: P1=1,319,535,  P2=  400,004,  P3=2,184,526 and  P4=1,906,648 (#3 - 4x)
```

Two important questions to answer concerning this algorithm

- What does Algebraic law say about anyone being able to ever solve this one equation for the correct single values of the 4 unknowns?
- Does this provide adequate protection for the values within the fixed 8,388,736 byte key array ‘key’?

```
For i = 0 To 127
    ctx = ctx + chr$(Asc(Mid$(ptx, i + 1, 1)) Xor _
        key(Ptr1 + i) Xor key(Ptr2 + i) Xor _
        key(Ptr3 + i) Xor key(Ptr4 + i))
Next i
```

Two more important questions to answer concerning this algorithm

- Suppose the 4 table values were Xor'ed together and the result was loaded into temp, and this single location was Xor'ed with the plaintext ASCII number producing the ciphertext character.
- What decades-old cipher algorithm is the second expression?
- Does this provide protection at least equal to the AES in protecting the plaintext characters from discovery?

```
For i = 0 To 127
    temp = key(Ptr1 + i) Xor key(Ptr2 + i) Xor _
        key(Ptr3 + i) Xor key(Ptr4 + i)
    ctx = ctx + chr$(Asc(Mid$(ptx, i + 1, 1)) Xor temp)
Next i
```

One last question:

- What would be the mathematical process of obtaining the values of Ptr1 - Ptr4 used in this engine using only the plaintext and ciphertext ASCII characters that any attacker would use?
- Keep in mind that for each individual value in this equation, there are well over 32,000 locations within the 8,388,736 byte key with that same value. So, is it possible?

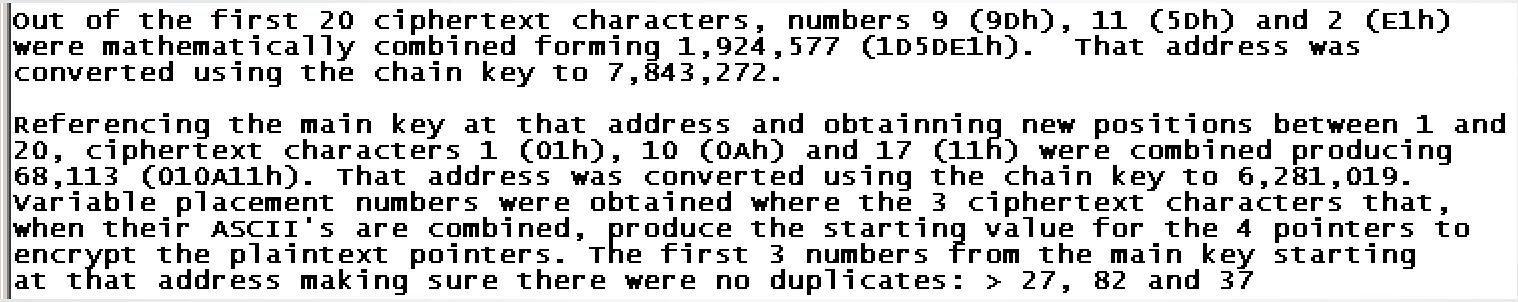
```
For i = 0 To 127
    ctx = ctx + chr$(Asc(Mid$(ptx, i + 1, 1)) Xor _
          key(Ptr1 + i) Xor key(Ptr2 + i) Xor _
          key(Ptr3 + i) Xor key(Ptr4 + i))Next i
Next i
```

Key table storage

- Since key changes will no longer be needed since there is no more concern about potential future breeches or key table theft during new key transport, key storage can be within the image itself.
- The image is secure within the computer chip, so if the key is there also, it too will be just as safe.
- The 32 bytes are individually stored throughout the source file in random locations.
- The key input function merely calls the 32 load subroutines and wherever they are within the image, they are put in the proper order in the 32-number key array.

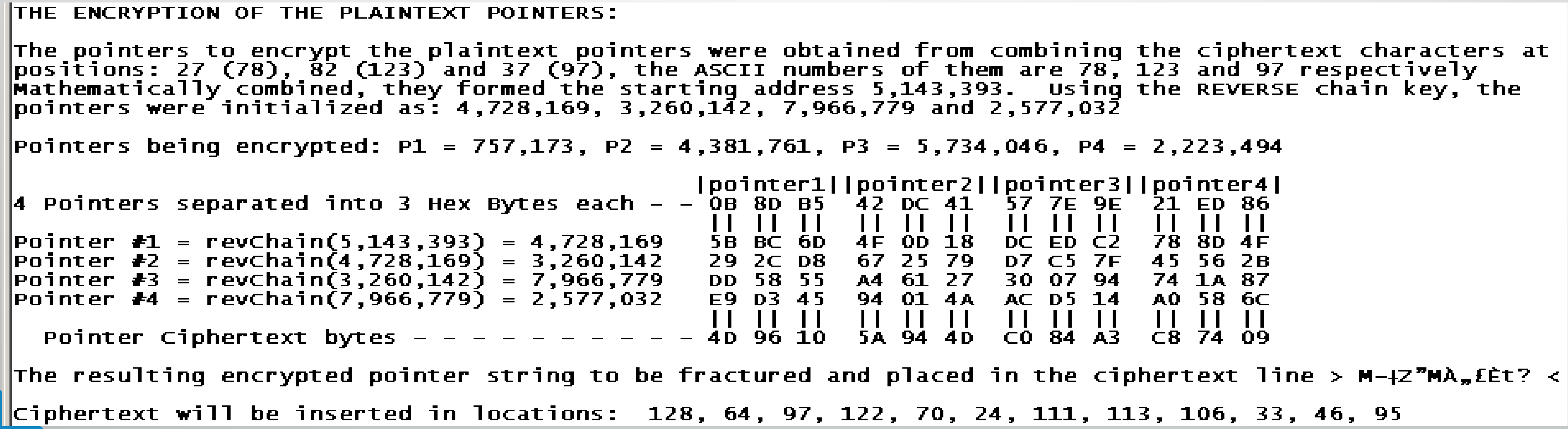
How are the main pointers encrypted and delivered to the decrypt cipher in the first block?

Actual extraction #1 from the demonstration output application:



```
Out of the first 20 ciphertext characters, numbers 9 (9Dh), 11 (5Dh) and 2 (E1h)
were mathematically combined forming 1,924,577 (1D5DE1h).  That address was
converted using the chain key to 7,843,272.

Referencing the main key at that address and obtainning new positions between 1 and
20, ciphertext characters 1 (01h), 10 (0Ah) and 17 (11h) were combined producing
68,113 (010A11h). That address was converted using the chain key to 6,281,019.
Variable placement numbers were obtained where the 3 ciphertext characters that,
when their ASCII's are combined, produce the starting value for the 4 pointers to
encrypt the plaintext pointers. The first 3 numbers from the main key starting
at that address making sure there were no duplicates: > 27, 82 and 37
```

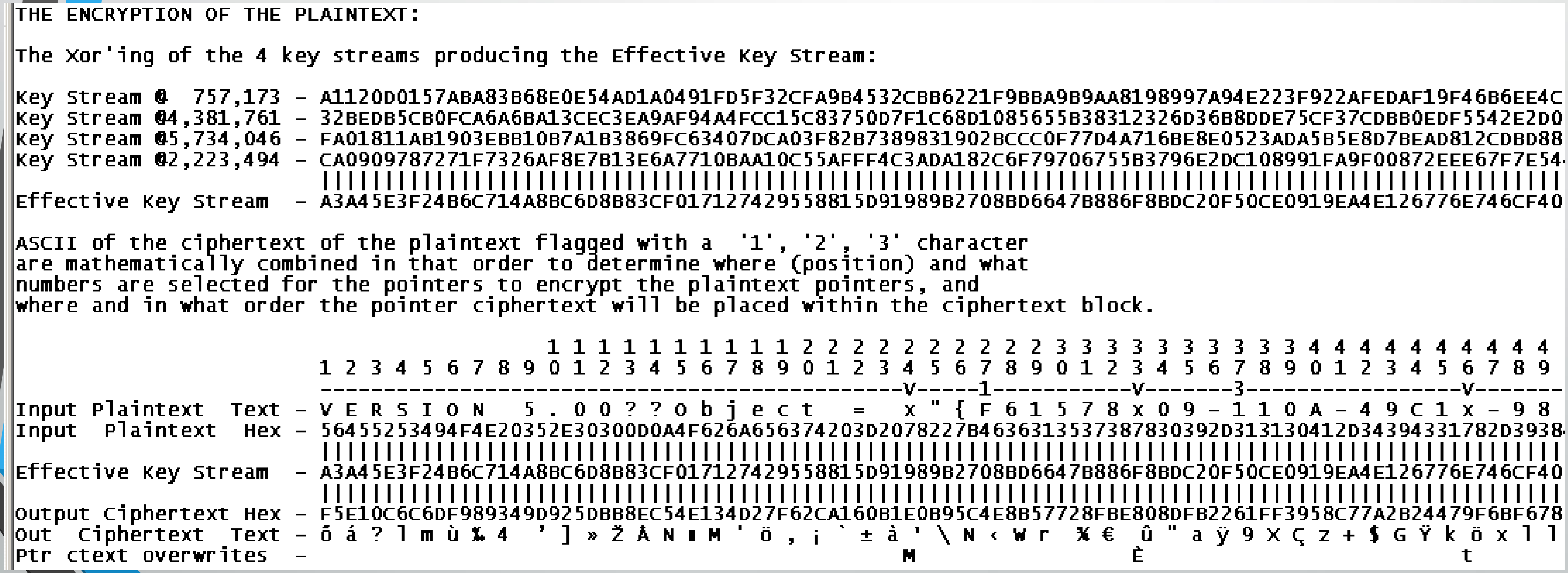
- These two sections are executed either side of the encrypt operation on the next slide, but shown together here because the top sequence obtains data the bottom sequence needs to execute



```
THE ENCRYPTION OF THE PLAINTEXT POINTERS:

The pointers to encrypt the plaintext pointers were obtained from combining the ciphertext characters at
positions: 27 (78), 82 (123) and 37 (97), the ASCII numbers of them are 78, 123 and 97 respectively
Mathematically combined, they formed the starting address 5,143,393.  Using the REVERSE chain key, the
pointers were initialized as: 4,728,169, 3,260,142, 7,966,779 and 2,577,032

Pointers being encrypted: P1 = 757,173, P2 = 4,381,761, P3 = 5,734,046, P4 = 2,223,494

                                             |pointer1||pointer2||pointer3||pointer4|
4 Pointers separated into 3 Hex Bytes each - - 0B 8D B5  42 DC 41  57 7E 9E  21 ED 86
                                               || || ||  || || ||  || || ||  || || ||
Pointer #1 = revChain(5,143,393) = 4,728,169   5B BC 6D  4F 0D 18  DC ED C2  78 8D 4F
Pointer #2 = revChain(4,728,169) = 3,260,142   29 2C D8  67 25 79  D7 C5 7F  45 56 2B
Pointer #3 = revChain(3,260,142) = 7,966,779   DD 58 55  A4 61 27  30 07 94  74 1A 87
Pointer #4 = revChain(7,966,779) = 2,577,032   E9 D3 45  94 01 4A  AC D5 14  A0 58 6C
                                               || || ||  || || ||  || || ||  || || ||
  Pointer Ciphertext bytes - - - - - - - - - - 4D 96 10  5A 94 4D  C0 84 A3  C8 74 09

The resulting encrypted pointer string to be fractured and placed in the ciphertext line > M–┼Z”MÀ„£Èt? <

Ciphertext will be inserted in locations:  128, 64, 97, 122, 70, 24, 111, 113, 106, 33, 46, 95
```

The plaintext encryption process

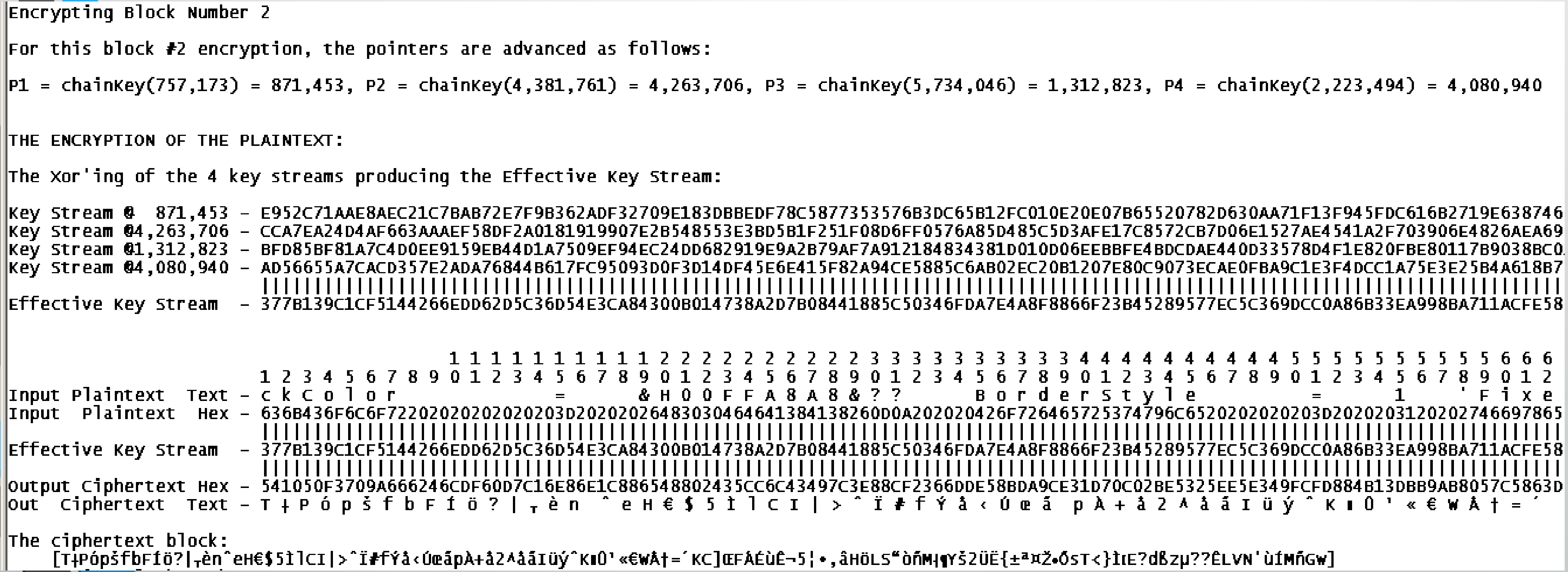
Actual extraction #1 from the demonstration output application:



```
THE ENCRYPTION OF THE PLAINTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @  757,173 - A1120D0157ABA83B68E0E54AD1A0491FD5F32CFA9B4532CBB6221F9BBA9B9AA8198997A94E223F922AFEDAF19F46B6EE4C
Key Stream @4,381,761 - 32BEDB5CB0FCA6A6BA13CEC3EA9AF94A4FCC15C83750D7F1C68D1085655B38312326D36B8DDE75CF37CDBB0EDF5542E2D0
Key Stream @5,734,046 - FA01811AB1903EBB10B7A1B3869FC63407DCA03F82B7389831902BCCC0F77D4A716BE8E0523ADA5B5E8D7BEAD812CDBD88
Key Stream @2,223,494 - CA0909787271F7326AF8E7B13E6A7710BAA10C55AFFF4C3ADA182C6F79706755B3796E2DC108991FA9F00872EEE67F7E54
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - A3A45E3F24B6C714A8BC6D8B83CF01712742955881 5D91989B2708BD6647B886F8BDC20F50CE0919EA4E126776E746CF40

ASCII of the ciphertext of the plaintext flagged with a  '1', '2', '3' character
are mathematically combined in that order to determine where (position) and what
numbers are selected for the pointers to encrypt the plaintext pointers, and
where and in what order the pointer ciphertext will be placed within the ciphertext block.

                                          1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4
                        1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
                        ----------------------------------------------V-----1-----------V---------3---------------V-------
Input Plaintext  Text - V E R S I O N   5 . 0 0 ? ? o b j e c t   =   x " { F 6 1 5 7 8 x 0 9 - 1 1 0 A - 4 9 C 1 x - 9 8
Input  Plaintext  Hex - 56455253494F4E20352E30300D0A4F626A656374203D2078227B46363135373878303 92D313130412D34394331782D3938
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - A3A45E3F24B6C714A8BC6D8B83CF01712742955881 5D91989B2708BD6647B886F8BDC20F50CE0919EA4E126776E746CF40
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output Ciphertext Hex - F5E10C6C6DF989349D925DBB8EC54E134D27F62CA160B1E0B95C4E8B57728FBE808DFB2261FF3958C77A2B24479F6BF678
Out  Ciphertext  Text - õ á ? l m ù ‰ 4   ’ ] » Ž Å N ■ M ' ö , ¡ ` ± à ' \ N ‹ W r   ¾ €   û " a ÿ 9 X Ç z + $ G Ÿ k ö x l l
Ptr ctext overwrites  -                                               M                 È                         t
```

Plaintext encryption After Block #1

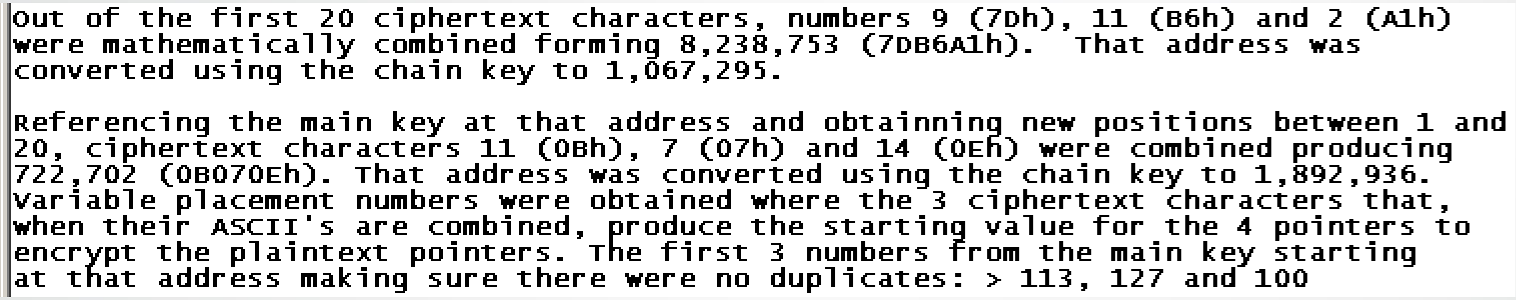
Actual extraction #1 from the demonstration output application, this functionality is repeated for all subsequent blocks:



```
Encrypting Block Number 2

For this block #2 encryption, the pointers are advanced as follows:

P1 = chainKey(757,173) = 871,453, P2 = chainKey(4,381,761) = 4,263,706, P3 = chainKey(5,734,046) = 1,312,823, P4 = chainKey(2,223,494) = 4,080,940


THE ENCRYPTION OF THE PLAINTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @  871,453 - E952C71AAE8AEC21C7BAB72E7F9B362ADF32709E183DBBEDF78C5877353576B3DC65B12FC010E20E07B65520782D630AA71F13F945FDC616B2719E638746
Key Stream @4,263,706 - CCA7EA24D4AF663AAAEF58DF2A0181919907E2B548553E3BD5B1F251F08D6FF0576A85D485C5D3AFE17C8572CB7D06E1527AE4541A2F703906E4826AEA69
Key Stream @1,312,823 - BFD85BF81A7C4D0EE9159EB44D1A7509EF94EC24DD682919E9A2B79AF7A9121848343810010D06EEBBFE4BDCDAE440D33578D4F1E820FBE80117B9038BC0
Key Stream @4,080,940 - AD56655A7CACD357E2ADA76844B617FC95093D0F3D14DF45E6E415F82A94CE5885C6AB02EC20B1207E80C9073ECAE0FBA9C1E3F4DCC1A75E3E25B4A618B7
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 377B139C1CF5144266EDD62D5C36D54E3CA84300B014738A2D7B08441885C50346FDA7E4A8F8866F23B45289577EC5C369DCC0A86B33EA998BA711ACFE58


                                          1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 6 6 6
                        1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
Input Plaintext  Text - c k C o l o r               =       & H 0 0 F F A 8 A 8 & ? ?       B o r d e r S t y l e           =       1     ' F i x e
Input  Plaintext  Hex - 636B436F6C6F7220202020202020203D20202026483030464641384138260D0A202020426F72646572537479 6C65202020202020203D2020203120202746697865
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 377B139C1CF5144266EDD62D5C36D54E3CA84300B014738A2D7B08441885C50346FDA7E4A8F8866F23B45289577EC5C369DCC0A86B33EA998BA711ACFE58
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output Ciphertext Hex - 541050F3709A666246CDF60D7C16E86E1C886548802435CC6C43497C3E88CF2366DDE58BDA9CE31D70C02BE5325EE5E349FCFD884B13DBB9AB8057C5863D
Out  Ciphertext  Text - T ┼ P ó p š f b F í ö ? | ┬ è n   ˆ e H € $ 5 Ì l C I | > ˆ Ï # f Ý å ‹ Ú œ ã   p À + å 2 ^ å ã I ü ý ˆ K ▮ Û ' « € W Å † = ´

The ciphertext block:
    [T┼PópšfbFíö?|┬ènˆeH€$5ÌlCI|>ˆÏ#fÝå‹ÚœãpÀ+å2^åãIüýˆK▮Û'«€WÅ†=´KC]ŒFÅÉÙÊ¬5¦•,âHÖLS“òñM┤¶YŠ2ÜË{±ª¤Ž•ŐsT<}ÌıE?dßzµ??ÊLVN'ùÍMñGw]
```

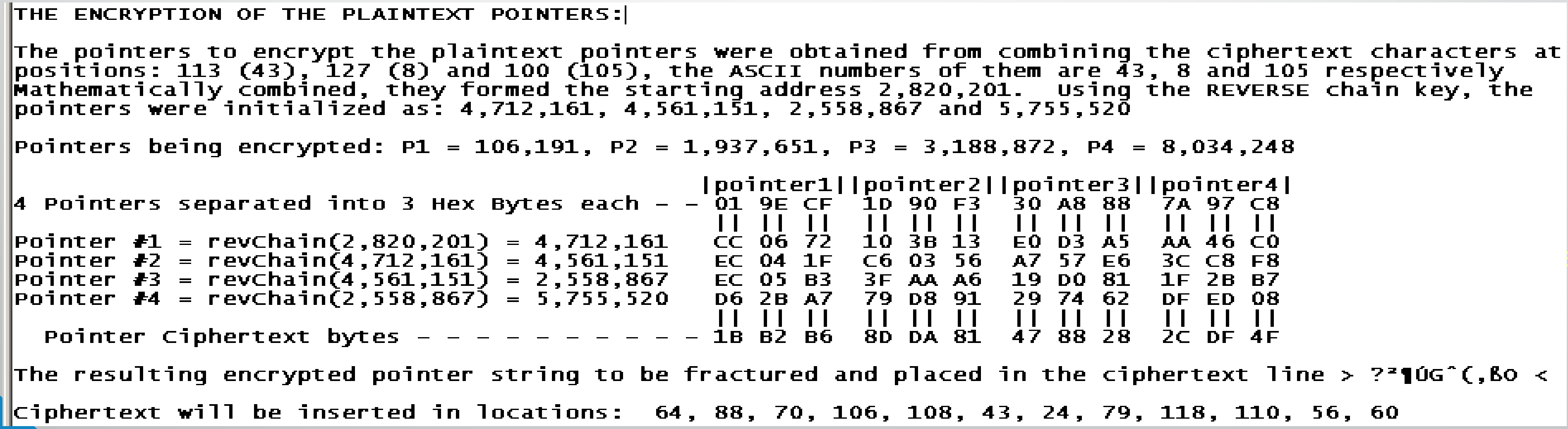
How are the main pointers encrypted and delivered to the decrypt cipher in the first block?

Actual extraction #2 from the demonstration output application:



```
Out of the first 20 ciphertext characters, numbers 9 (7Dh), 11 (B6h) and 2 (A1h)
were mathematically combined forming 8,238,753 (7DB6A1h).  That address was
converted using the chain key to 1,067,295.

Referencing the main key at that address and obtainning new positions between 1 and
20, ciphertext characters 11 (0Bh), 7 (07h) and 14 (0Eh) were combined producing
722,702 (0B070Eh). That address was converted using the chain key to 1,892,936.
Variable placement numbers were obtained where the 3 ciphertext characters that,
when their ASCII's are combined, produce the starting value for the 4 pointers to
encrypt the plaintext pointers. The first 3 numbers from the main key starting
at that address making sure there were no duplicates: > 113, 127 and 100
```

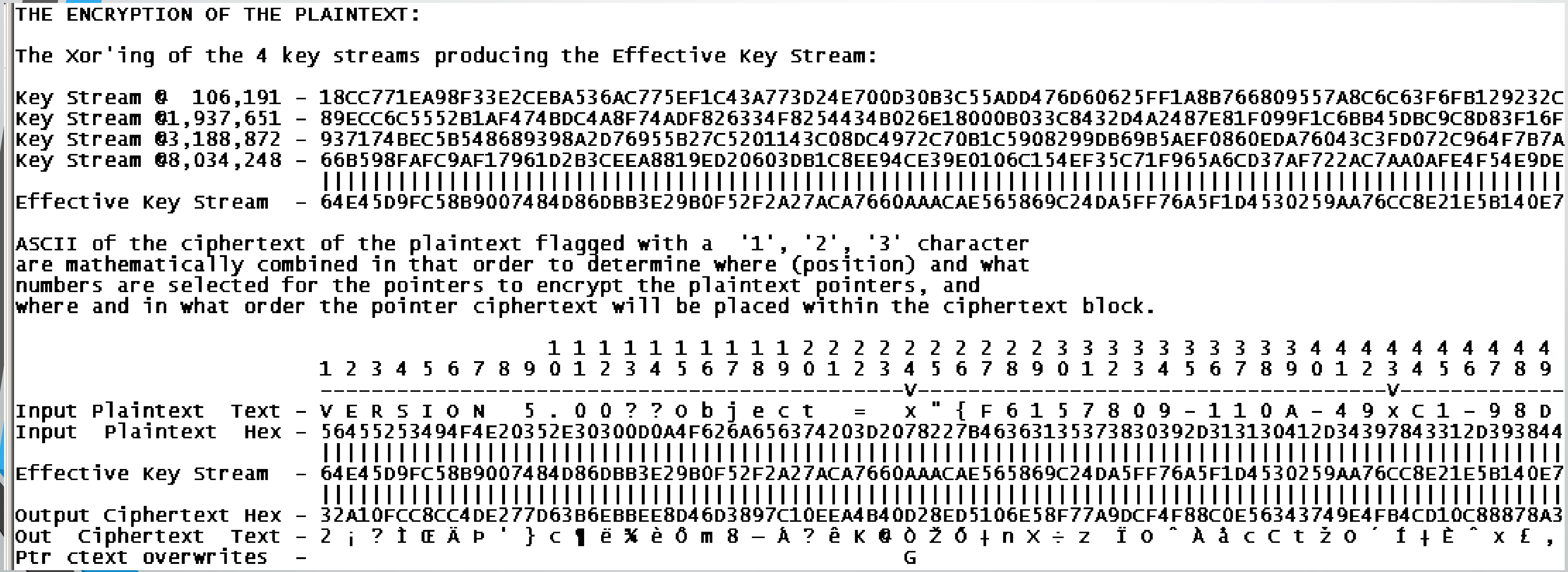
- These two sections are executed either side of the encrypt operation on the next slide, but shown together here because the top sequence obtains data the bottom sequence needs to execute



```
THE ENCRYPTION OF THE PLAINTEXT POINTERS:|

The pointers to encrypt the plaintext pointers were obtained from combining the ciphertext characters at
positions: 113 (43), 127 (8) and 100 (105), the ASCII numbers of them are 43, 8 and 105 respectively
Mathematically combined, they formed the starting address 2,820,201.  Using the REVERSE chain key, the
pointers were initialized as: 4,712,161, 4,561,151, 2,558,867 and 5,755,520

Pointers being encrypted: P1 = 106,191, P2 = 1,937,651, P3 = 3,188,872, P4 = 8,034,248

                                              |pointer1||pointer2||pointer3||pointer4|
4 Pointers separated into 3 Hex Bytes each - - 01 9E CF  1D 90 F3  30 A8 88  7A 97 C8
                                               || || ||  || || ||  || || ||  || || ||
Pointer #1 = revChain(2,820,201) = 4,712,161   CC 06 72  10 3B 13  E0 D3 A5  AA 46 C0
Pointer #2 = revChain(4,712,161) = 4,561,151   EC 04 1F  C6 03 56  A7 57 E6  3C C8 F8
Pointer #3 = revChain(4,561,151) = 2,558,867   EC 05 B3  3F AA A6  19 D0 81  1F 2B B7
Pointer #4 = revChain(2,558,867) = 5,755,520   D6 2B A7  79 D8 91  29 74 62  DF ED 08
                                               || || ||  || || ||  || || ||  || || ||
  Pointer Ciphertext bytes - - - - - - - - - - 1B B2 B6  8D DA 81  47 88 28  2C DF 4F

The resulting encrypted pointer string to be fractured and placed in the ciphertext line > ?²¶ÚGˆ(,ßO <

Ciphertext will be inserted in locations:  64, 88, 70, 106, 108, 43, 24, 79, 118, 110, 56, 60
```

The plaintext encryption process

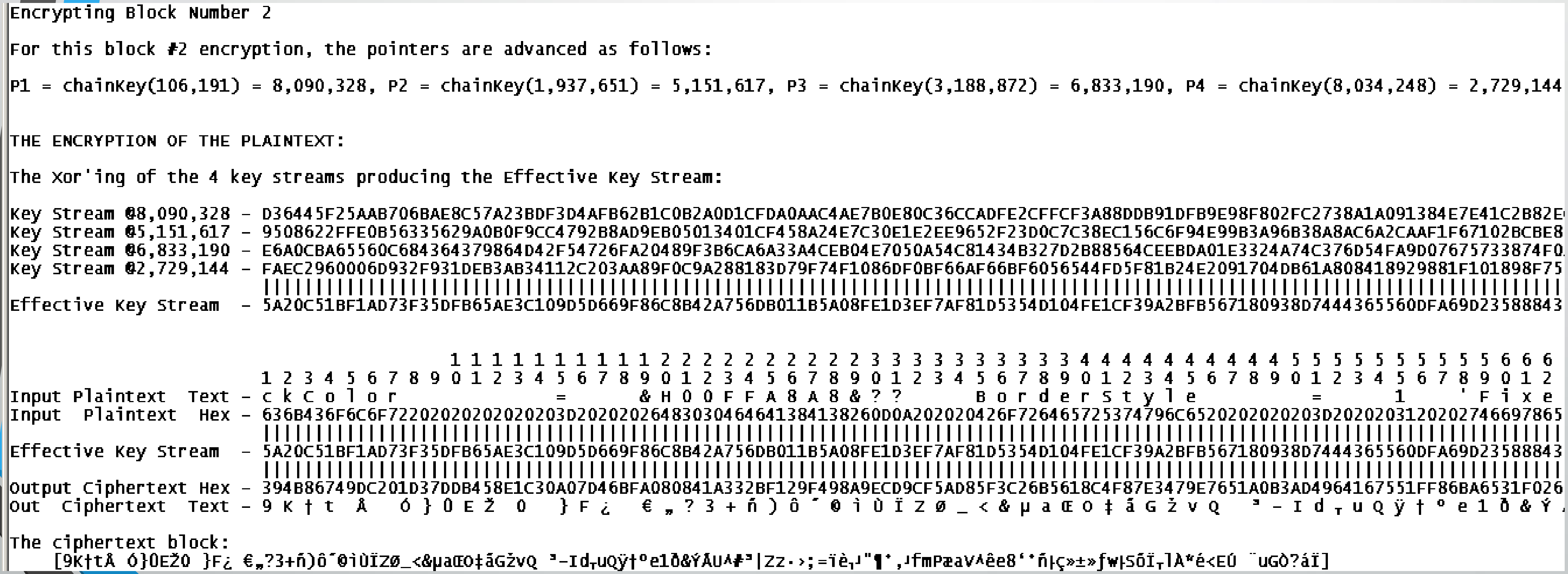
Actual extraction #2 from the demonstration output application:



```
THE ENCRYPTION OF THE PLAINTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @  106,191 - 18CC771EA98F33E2CEBA536AC775EF1C43A773D24E700D30B3C55ADD476D60625FF1A8B766809557A8C6C63F6FB129232C
Key Stream @1,937,651 - 89ECC6C5552B1AF474BDC4A8F74ADF826334F8254434B026E18000B033C8432D4A2487E81F099F1C6BB45DBC9C8D83F16F
Key Stream @3,188,872 - 937174BEC5B548689398A2D76955B27C5201143C08DC4972C70B1C5908299DB69B5AEF0860EDA76043C3FD072C964F7B7A
Key Stream @8,034,248 - 66B598FAFC9AF17961D2B3CEEA8819ED20603DB1C8EE94CE39E0106C154EF35C71F965A6CD37AF722AC7AA0AFE4F54E9DE
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 64E45D9FC58B9007484D86DBB3E29B0F52F2A27ACA7660AAACAE565869C24DA5FF76A5F1D4530259AA76CC8E21E5B140E7

ASCII of the ciphertext of the plaintext flagged with a  '1', '2', '3' character
are mathematically combined in that order to determine where (position) and what
numbers are selected for the pointers to encrypt the plaintext pointers, and
where and in what order the pointer ciphertext will be placed within the ciphertext block.

                                          1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4
                        1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
                        ----------------------------------------------V-------------------------------------V-------------
Input Plaintext  Text - V E R S I O N   5 . 0 0 ? ? o b j e c t   =   x " { F 6 1 5 7 8 0 9 - 1 1 0 A - 4 9 x C 1 - 9 8 D
Input  Plaintext  Hex - 5645525349 4F4E20352E30300D0A4F626A656374203D2078227B46363135373830392D313130412D343978433132D393844
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 64E45D9FC58B9007484D86DBB3E29B0F52F2A27ACA7660AAACAE565869C24DA5FF76A5F1D4530259AA76CC8E21E5B140E7
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output Ciphertext Hex - 32A10FCC8CC4DE277D63B6EBBEE8D46D3897C10EEA4B40D28ED5106E58F77A9DCF4F88C0E56343749E4FB4CD10C88878A3
Out  Ciphertext  Text - 2 ¡ ? Ì Œ Ä Þ ' } c ¶ ë ¾ è Ô m 8 – À ? ê K @ Ò Ž Õ † n X ÷ z Ï O ˆ À å c C t ž o ´ Í † È ˆ x £ ,
Ptr ctext overwrites  -                                               G
```

Plaintext encryption After Block #1

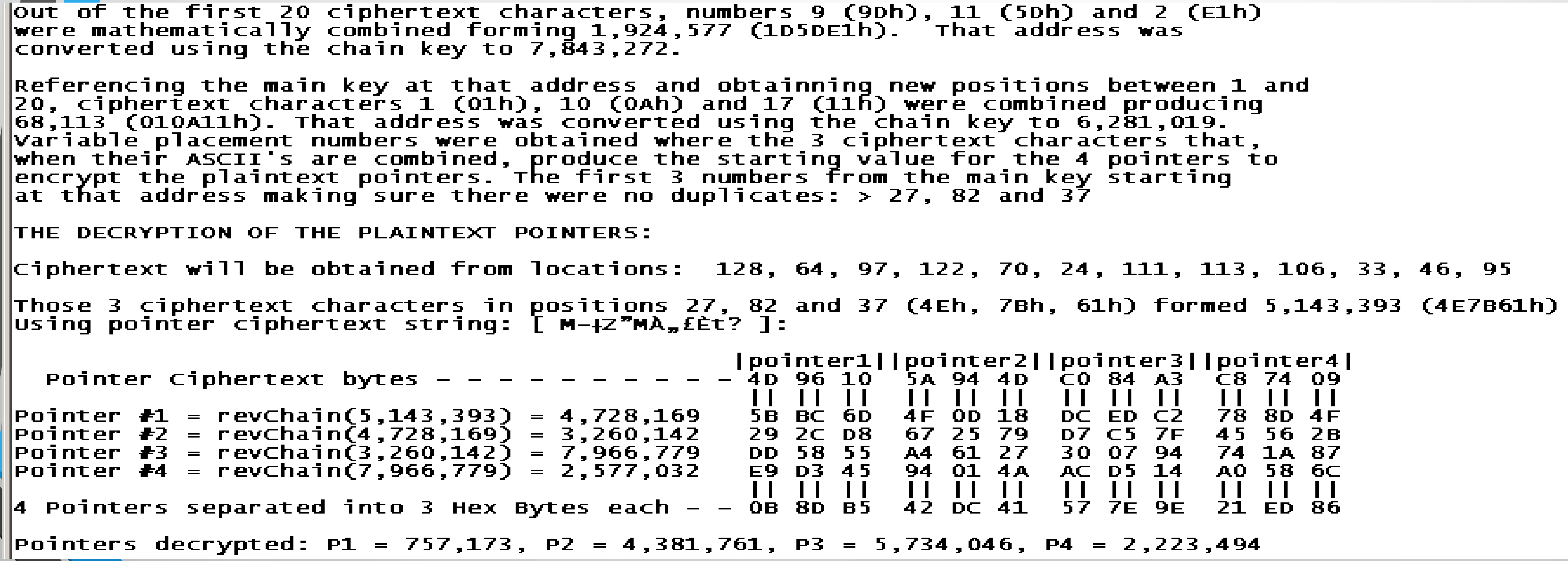
Actual extraction #2 from the demonstration output application, this functionality is repeated for all subsequent blocks:



```
Encrypting Block Number 2

For this block #2 encryption, the pointers are advanced as follows:

P1 = chainKey(106,191) = 8,090,328, P2 = chainKey(1,937,651) = 5,151,617, P3 = chainKey(3,188,872) = 6,833,190, P4 = chainKey(8,034,248) = 2,729,144


THE ENCRYPTION OF THE PLAINTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @8,090,328 - D36445F25AAB706BAE8C57A23BDF3D4AFB62B1C0B2A0D1CFDA0AAC4AE7B0E80C36CCADFE2CFFCF3A88DDB91DFB9E98F802FC2738A1A091384E7E41C2B82E
Key Stream @5,151,617 - 9508622FFE0B56335629A0B0F9CC4792B8AD9EB05013401CF458A24E7C30E1E2EE9652F23D0C7C38EC156C6F94E99B3A96B38A8AC6A2CAAF1F67102BCBE8
Key Stream @6,833,190 - E6A0CBA65560C684364379864D42F54726FA20489F3B6CA6A33A4CEB04E7050A54C81434B327D2B88564CEEBDA01E3324A74C376D54FA9D07675733874F0
Key Stream @2,729,144 - FAEC2960006D932F931DEB3AB34112C203AA89F0C9A288183D79F74F1086DF0BF66AF66BF6056544FD5F81B24E2091704DB61A808418929881F101898F75
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 5A20C51BF1AD73F35DFB65AE3C109D5D669F86C8B42A756DB011B5A08FE1D3EF7AF81D5354D104FE1CF39A2BFB567180938D744436556 0DFA69D23588843


                                          1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 5 6 6 6
                        1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
Input Plaintext  Text - c k C o l o r               =       & H 0 0 F F A 8 A 8 & ? ?       B o r d e r S t y l e           =       1     ' F i x e
Input  Plaintext  Hex - 636B436F6C6F72202020202020203D202020264830304646413841382 60D0A202020426F72646572537479 6C652020202020203D2020203120202746697865
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 5A20C51BF1AD73F35DFB65AE3C109D5D669F86C8B42A756DB011B5A08FE1D3EF7AF81D5354D104FE1CF39A2BFB567180938D744436556 0DFA69D23588843
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output Ciphertext Hex - 394B86749DC201D37DDB458E1C30A07D46BFA080841A332BF129F498A9ECD9CF5AD85F3C26B5618C4F87E3479E7651A0B3AD4964167551FF86BA6531F026
Out  Ciphertext  Text - 9 K † t   Å   Ó } Û E Ž   0   } F ¿   € „ ? 3 + ñ ) ô ˜ © ì Ù Ï Z Ø _ < & µ a Œ O ‡ ã G ž v Q   ³ - I d ┬ u Q ÿ † º e 1 ð & Ý ,

The ciphertext block:
    [9K†tÅ Ó}ÛEŽ0 }F¿ €„?3+ñ)ô˜©ìÙÏZØ_<&µaŒO‡ãGžvQ ³-Id┬uQÿ†ºe1ð&ÝÅU^#³|Zz·›;=ïè┐┘"¶'‚┘fmPæaV^êe8''ñ╞ç»±»ƒw╞SõÏ┬lÀ*é<EÚ ¨uGÒ?áÏ]
```

How are the main pointers obtained by the decrypt cipher?

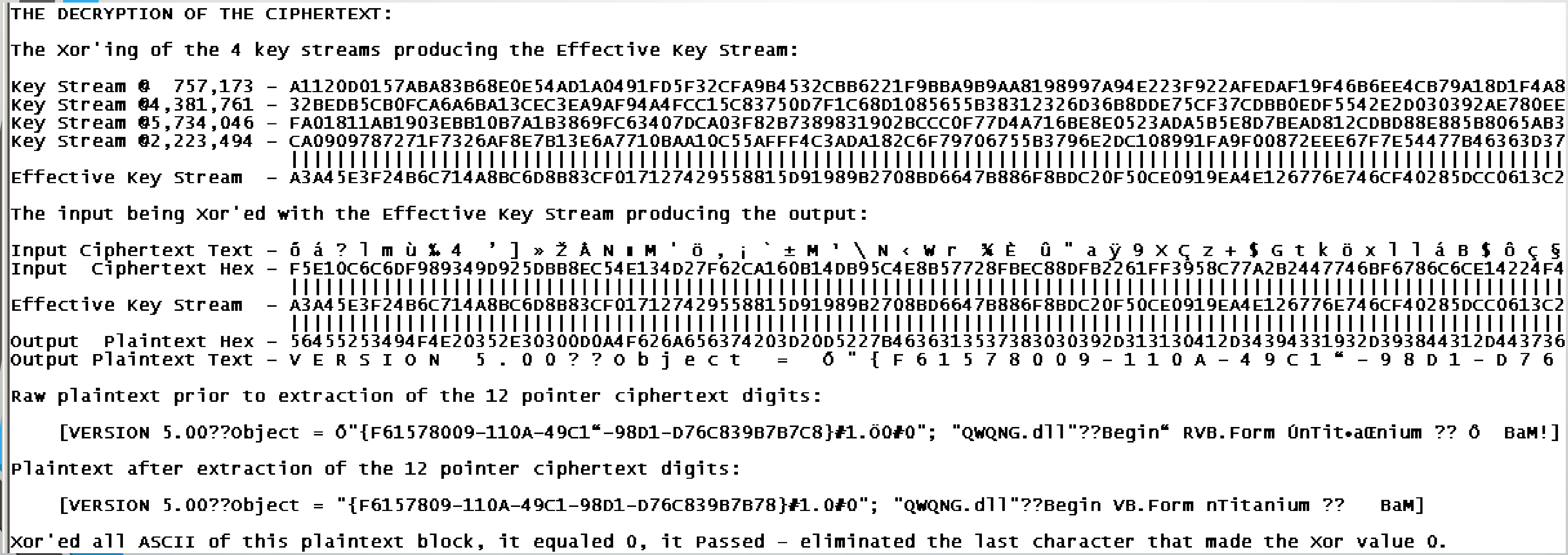
Actual extraction #1 from the demonstration output application:



```
Out of the first 20 ciphertext characters, numbers 9 (9Dh), 11 (5Dh) and 2 (E1h)
were mathematically combined forming 1,924,577 (1D5DE1h).  That address was
converted using the chain key to 7,843,272.

Referencing the main key at that address and obtainning new positions between 1 and
20, ciphertext characters 1 (01h), 10 (0Ah) and 17 (11h) were combined producing
68,113 (010A11h). That address was converted using the chain key to 6,281,019.
Variable placement numbers were obtained where the 3 ciphertext characters that,
when their ASCII's are combined, produce the starting value for the 4 pointers to
encrypt the plaintext pointers. The first 3 numbers from the main key starting
at that address making sure there were no duplicates: > 27, 82 and 37

THE DECRYPTION OF THE PLAINTEXT POINTERS:

Ciphertext will be obtained from locations:  128, 64, 97, 122, 70, 24, 111, 113, 106, 33, 46, 95

Those 3 ciphertext characters in positions 27, 82 and 37 (4Eh, 7Bh, 61h) formed 5,143,393 (4E7B61h)
Using pointer ciphertext string: [ M–†Z”MÀ„£Èt? ]:

                                                    |pointer1||pointer2||pointer3||pointer4|
  Pointer Ciphertext bytes - - - - - - - - - - - - 4D 96 10  5A 94 4D  C0 84 A3  C8 74 09
                                                   || || ||  || || ||  || || ||  || || ||
Pointer #1 = revChain(5,143,393) = 4,728,169      5B BC 6D  4F 0D 18  DC ED C2  78 8D 4F
Pointer #2 = revChain(4,728,169) = 3,260,142      29 2C D8  67 25 79  D7 C5 7F  45 56 2B
Pointer #3 = revChain(3,260,142) = 7,966,779      DD 58 55  A4 61 27  30 07 94  74 1A 87
Pointer #4 = revChain(7,966,779) = 2,577,032      E9 D3 45  94 01 4A  AC D5 14  A0 58 6C
                                                   || || ||  || || ||  || || ||  || || ||
4 Pointers separated into 3 Hex Bytes each - -    0B 8D B5  42 DC 41  57 7E 9E  21 ED 86

Pointers decrypted: P1 = 757,173, P2 = 4,381,761, P3 = 5,734,046, P4 = 2,223,494
```

The plaintext decryption process

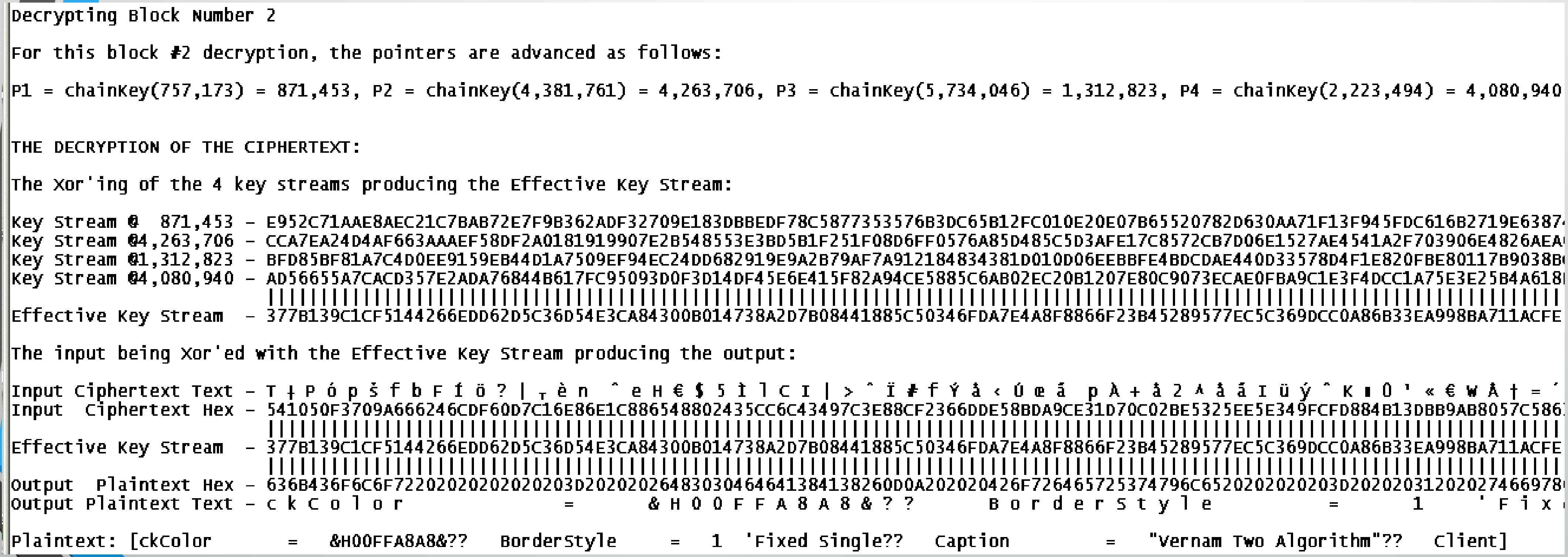
Actual extraction #1 from the demonstration output application:



```
THE DECRYPTION OF THE CIPHERTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @  757,173 - A1120D0157ABA83B68E0E54AD1A0491FD5F32CFA9B4532CBB6221F9BBA9B9AA8198997A94E223F922AFEDAF19F46B6EE4CB79A18D1F4A8
Key Stream @4,381,761 - 32BEDB5CB0FCA6A6BA13CEC3EA9AF94A4FCC15C83750D7F1C68D1085655B38312326D36B8DDE75CF37CDBB0EDF5542E2D030392AE780EE
Key Stream @5,734,046 - FA01811AB1903EBB10B7A1B3869FC63407DCA03F82B7389831902BCCC0F77D4A716BE8E0523ADA5B5E8D7BEAD812CDBD88E885B8065AB3
Key Stream @2,223,494 - CA0909787271F7326AF8E7B13E6A7710BAA10C55AFFF4C3ADA182C6F79706755B3796E2DC108991FA9F00872EEE67F7E54477B46363D37
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - A3A45E3F24B6C714A8BC6D8B83CF0171274295588 15D91989B2708BD6647B886F8BDC20F50CE0919EA4E126776E746CF40285DCC0613C2

The input being Xor'ed with the Effective Key Stream producing the output:

Input Ciphertext Text - õ á ? l m ù ‰ 4   ’ ] » Ž Å N ▮ M ' ö , ¡ ` ± M ' \ N ‹ W r   ¾ È   û " a ÿ 9 X Ç z + $ G t k ö x l l á B $ ô ç §
Input  Ciphertext Hex - F5E10C6C6DF989349D925DBB8EC54E134D27F62CA160B14DB95C4E8B57728FBEC88DFB2261FF3958C77A2B2447746BF6786C6CE14224F4
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - A3A45E3F24B6C714A8BC6D8B83CF01712742955881 5D91989B2708BD6647B886F8BDC20F50CE0919EA4E126776E746CF40285DCC0613C2
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output  Plaintext Hex - 56455253494F4E20352E30300D0A4F626A656374203D20D5227B46363135373830303 92D313130412D343943319 32D39384431 2D443736
Output Plaintext Text - V E R S I O N   5 . 0 0 ? ? O b j e c t   =   Õ " { F 6 1 5 7 8 0 0 9 - 1 1 0 A - 4 9 C 1 “ - 9 8 D 1 - D 7 6

Raw plaintext prior to extraction of the 12 pointer ciphertext digits:

    [VERSION 5.00??Object = Õ"{F61578009-110A-49C1“-98D1-D76C839B7B7C8}#1.Ö0#0"; "QWQNG.dll"??Begin“ RVB.Form ÚnTit•aŒnium ?? Õ   BaM!]

Plaintext after extraction of the 12 pointer ciphertext digits:

    [VERSION 5.00??Object = "{F6157809-110A-49C1-98D1-D76C839B7B78}#1.0#0"; "QWQNG.dll"??Begin VB.Form nTitanium ??   BaM]

Xor'ed all ASCII of this plaintext block, it equaled 0, it Passed - eliminated the last character that made the Xor value 0.
```

Plaintext decryption After Block #1

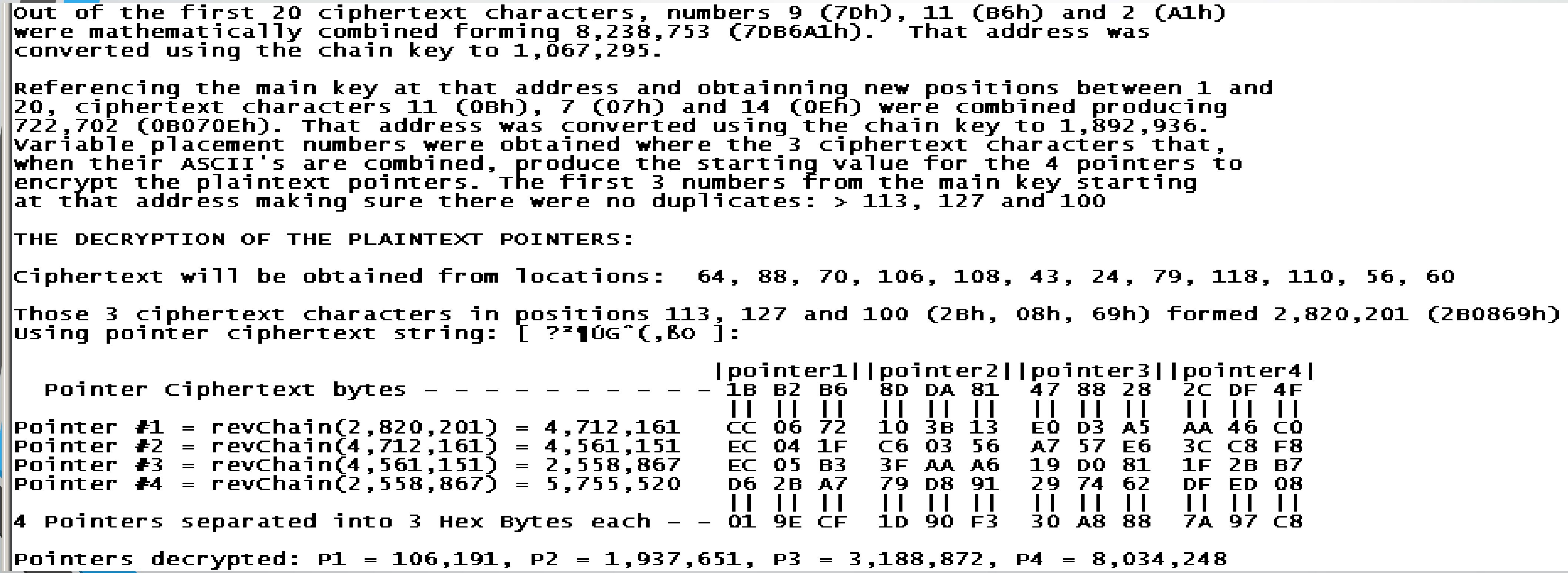
Actual extraction #1 from the demonstration output application, this functionality is repeated for all subsequent blocks:



```
Decrypting Block Number 2

For this block #2 decryption, the pointers are advanced as follows:

P1 = chainKey(757,173) = 871,453, P2 = chainKey(4,381,761) = 4,263,706, P3 = chainKey(5,734,046) = 1,312,823, P4 = chainKey(2,223,494) = 4,080,940


THE DECRYPTION OF THE CIPHERTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @  871,453 - E952C71AAE8AEC21C7BAB72E7F9B362ADF32709E183DBBEDF78C5877353576B3DC65B12FC010E20E07B65520782D630AA71F13F945FDC616B2719E6387
Key Stream @4,263,706 - CCA7EA24D4AF663AAAEF58DF2A0181919907E2B548553E3BD5B1F251F08D6FF0576A85D485C5D3AFE17C8572CB7D06E1527AE4541A2F703906E4826AEA
Key Stream @1,312,823 - BFD85BF81A7C4D0EE9159EB44D1A7509EF94EC24DD682919E9A2B79AF7A912184834381D010D06EEBBFE4BDCDAE440D33578D4F1E820FBE80117B9038B
Key Stream @4,080,940 - AD56655A7CACD357E2ADA76844B617FC95093D0F3D14DF45E6E415F82A94CE5885C6AB02EC20B1207E80C9073ECAE0FBA9C1E3F4DCC1A75E3E25B4A618
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 377B139C1CF5144266EDD62D5C36D54E3CA84300B014738A2D7B08441885C50346FDA7E4A8F8866F23B45289577EC5C369DCC0A86B33EA998BA711ACFE

The input being Xor'ed with the Effective Key Stream producing the output:

Input Ciphertext Text - T ┼ P ó p š f b F ſ ö ? | ┬ è n  ˆ e H € $ 5 İ l C I | > ˆ Ï ≠ f Ý å ‹ Ú œ ã  p À + å 2 ^ å ã I ü ý ˆ K ▮ Ô ' « € W Å † = ´
Input  Ciphertext Hex - 541050F3709A666246CDF60D7C16E86E1C886548802435CC6C43497C3E88CF2366DDE58BDA9CE31D70C02BE5325EE5E349FCFD884B13DBB9AB8057C586
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 377B139C1CF5144266EDD62D5C36D54E3CA84300B014738A2D7B08441885C50346FDA7E4A8F8866F23B45289577EC5C369DCC0A86B33EA998BA711ACFE
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output  Plaintext Hex - 636B436F6C6F72202020202020203D20202026483030464641384138260D0A202020426F726465725374796C6520202020203D20202031202027466978
Output Plaintext Text - c k C o l o r              =       & H 0 0 F F A 8 A 8 & ? ?       B o r d e r S t y l e            =       1       ' F i x

Plaintext: [ckColor       =   &H00FFA8A8&??   BorderStyle     =   1  'Fixed Single??   Caption          =   "Vernam Two Algorithm"??   Client]
```

How are the main pointers obtained by the decrypt cipher?

Actual extraction #2 from the demonstration output application:



```
Out of the first 20 ciphertext characters, numbers 9 (7Dh), 11 (B6h) and 2 (A1h)
were mathematically combined forming 8,238,753 (7DB6A1h).  That address was
converted using the chain key to 1,067,295.

Referencing the main key at that address and obtainning new positions between 1 and
20, ciphertext characters 11 (0Bh), 7 (07h) and 14 (0Eh) were combined producing
722,702 (0B070Eh). That address was converted using the chain key to 1,892,936.
Variable placement numbers were obtained where the 3 ciphertext characters that,
when their ASCII's are combined, produce the starting value for the 4 pointers to
encrypt the plaintext pointers. The first 3 numbers from the main key starting
at that address making sure there were no duplicates: > 113, 127 and 100

THE DECRYPTION OF THE PLAINTEXT POINTERS:

Ciphertext will be obtained from locations:  64, 88, 70, 106, 108, 43, 24, 79, 118, 110, 56, 60

Those 3 ciphertext characters in positions 113, 127 and 100 (2Bh, 08h, 69h) formed 2,820,201 (2B0869h)
Using pointer ciphertext string: [ ?²¶ÚG^(,ßo ]:

                                              |pointer1||pointer2||pointer3||pointer4|
  Pointer Ciphertext bytes - - - - - - - - - - 1B B2 B6   8D DA 81   47 88 28   2C DF 4F
                                               || || ||   || || ||   || || ||   || || ||
Pointer #1 = revChain(2,820,201) = 4,712,161   CC 06 72   10 3B 13   E0 D3 A5   AA 46 C0
Pointer #2 = revChain(4,712,161) = 4,561,151   EC 04 1F   C6 03 56   A7 57 E6   3C C8 F8
Pointer #3 = revChain(4,561,151) = 2,558,867   EC 05 B3   3F AA A6   19 D0 81   1F 2B B7
Pointer #4 = revChain(2,558,867) = 5,755,520   D6 2B A7   79 D8 91   29 74 62   DF ED 08
                                               || || ||   || || ||   || || ||   || || ||
4 Pointers separated into 3 Hex Bytes each - - 01 9E CF   1D 90 F3   30 A8 88   7A 97 C8

Pointers decrypted: P1 = 106,191, P2 = 1,937,651, P3 = 3,188,872, P4 = 8,034,248
```

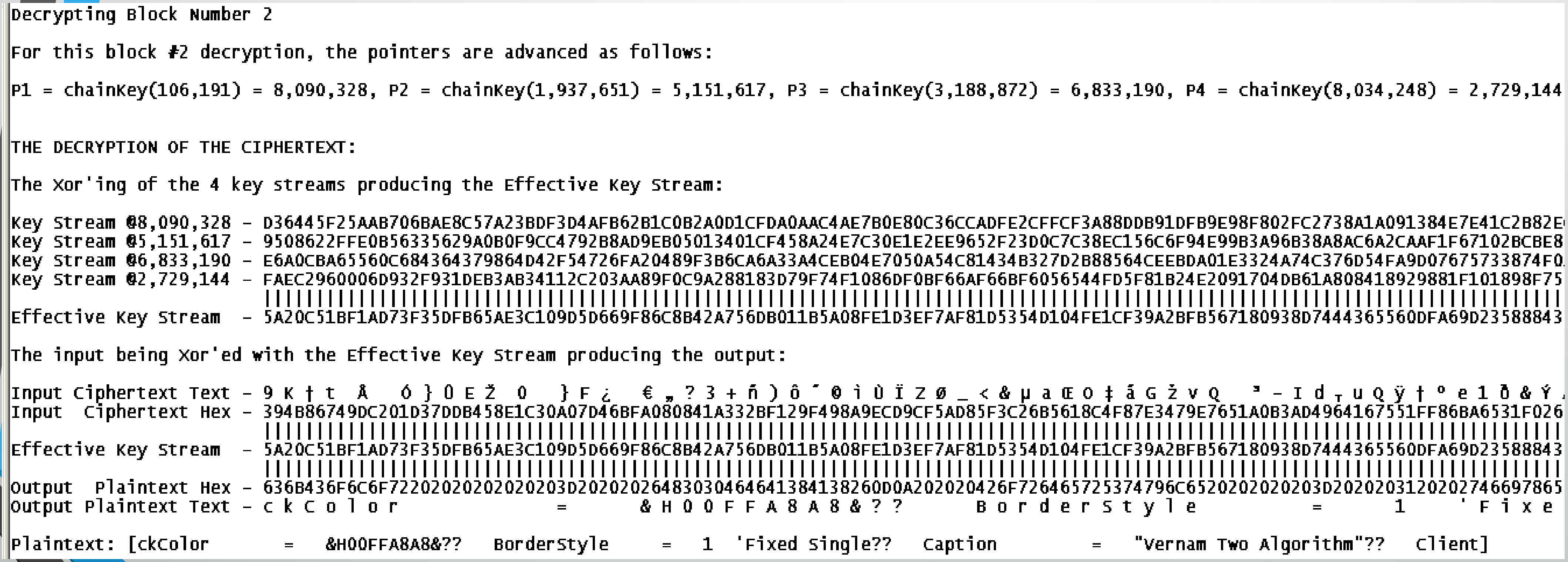
The plaintext decryption process

Actual extraction #2 from the demonstration output application:

```
THE DECRYPTION OF THE CIPHERTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @  106,191 - 18CC771EA98F33E2CEBA536AC775EF1C43A773D24E700D30B3C55ADD476D60625FF1A8B766809557A8C6C63F6FB129232C80BA770B10B7
Key Stream @1,937,651 - 89ECC6C5552B1AF474BDC4A8F74ADF826334F8254434B026E18000B033C8432D4A2487E81F099F1C6BB45DBC9C8D83F16F25ACEB5C171C
Key Stream @3,188,872 - 937174BEC5B548689398A2D76955B27C5201143C08DC4972C70B1C5908299DB69B5AEF0860EDA76043C3FD072C964F7B7A975B477E4B12
Key Stream @8,034,248 - 66B598FAFC9AF17961D2B3CEEA8819ED20603DB1C8EE94CE39E0106C154EF35C71F965A6CD37AF722AC7AA0AFE4F54E9DE2F6F581AC3FD
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 64E45D9FC58B9007484D86DBB3E29B0F52F2A27ACA7660AAACAE565869C24DA5FF76A5F1D4530259AA76CC8E21E5B140E71D2283338F44

The input being Xor'ed with the Effective Key Stream producing the output:

Input Ciphertext Text - 2 ¡ ? Ì Œ Ä Þ ´ } c ¶ ë ¾ è Ô m 8 – Á ? ê K @ G Ž Õ ‡ n X ÷ z  Ï O ˆ À å c C t Ž O  Í ‡ È ˆ x £ , ? Ç ┘ ' • ß L
Input  Ciphertext Hex - 32A10FCC8CC4DE277D63B6EBBEE8D46D3897C10EEA4B40478ED5106E58F77A9DCF4F88C0E56343749E4F81CD10C88878A32C0FC704B907
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 64E45D9FC58B9007484D86DBB3E29B0F52F2A27ACA7660AAACAE565869C24DA5FF76A5F1D4530259AA76CC8E21E5B140E71D2283338F44
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output  Plaintext Hex - 56455253494F4E20352E30300D0A4F626A656374203D20ED227B46363135373830392D313130412D34394D43312D393844312D44373643
Output Plaintext Text - V E R S I O N   5 . 0 0 ? ? O b j e c t   =   í " { F 6 1 5 7 8 0 9 - 1 1 0 A - 4 9 M C 1 - 9 8 D 1 - D 7 6 C

Raw plaintext prior to extraction of the 12 pointer ciphertext digits:

    [VERSION 5.00??Object = í"{F6157809-110A-49MC1-98D1-D76Cê839ÓB7Bƒ78}#1‡.0#0"; "ùQWQNG.dl•l"??Begin VB.FormU Én?Titaniu™m ??   BaM]

Plaintext after extraction of the 12 pointer ciphertext digits:

    [VERSION 5.00??Object = "{F6157809-110A-49C1-98D1-D76C839B7B78}#1.0#0"; "QWQNG.dll"??Begin VB.Form nTitanium ??   BaM]

Xor'ed all ASCII of this plaintext block, it equaled 0, it Passed - eliminated the last character that made the Xor value 0.
```

Plaintext decryption After Block #1

Actual extraction #2 from the demonstration output application, this functionality is repeated for all subsequent blocks:



```
Decrypting Block Number 2

For this block #2 decryption, the pointers are advanced as follows:

P1 = chainKey(106,191) = 8,090,328, P2 = chainKey(1,937,651) = 5,151,617, P3 = chainKey(3,188,872) = 6,833,190, P4 = chainKey(8,034,248) = 2,729,144


THE DECRYPTION OF THE CIPHERTEXT:

The Xor'ing of the 4 key streams producing the Effective Key Stream:

Key Stream @8,090,328 - D36445F25AAB706BAE8C57A23BDF3D4AFB62B1C0B2A0D1CFDA0AAC4AE7B0E80C36CCADFE2CFFCF3A88DDB91DFB9E98F802FC2738A1A091384E7E41C2B82E
Key Stream @5,151,617 - 9508622FFE0B56335629A0B0F9CC4792B8AD9EB05013401CF458A24E7C30E1E2EE9652F23D0C7C38EC156C6F94E99B3A96B38A8AC6A2CAAF1F67102BCBE8
Key Stream @6,833,190 - E6A0CBA65560C684364379864D42F54726FA20489F3B6CA6A33A4CEB04E7050A54C81434B327D2B88564CEEBDA01E3324A74C376D54FA9D07675733874F0
Key Stream @2,729,144 - FAEC2960006D932F931DEB3AB34112C203AA89F0C9A288183D79F74F1086DF0BF66AF66BF6056544FD5F81B24E2091704DB61A808418929881F101898F75
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 5A20C51BF1AD73F35DFB65AE3C109D5D669F86C8B42A756DB011B5A08FE1D3EF7AF81D5354D104FE1CF39A2BFB567180938D7444365560DFA69D23588843

The input being Xor'ed with the Effective Key Stream producing the output:

Input Ciphertext Text - 9 K † t   Å   Ó } Û E Ž   0   } F ¿   € „ ? 3 + ñ ) ô ˜ © ì Ù Ï Z Ø _ < & µ a Œ O ‡ ã G ž v Q   ³ - I d ┬ u Q ÿ † º e 1 ð & Ý
Input  Ciphertext Hex - 394B86749DC201D37DDB458E1C30A07D46BFA080841A332BF129F498A9ECD9CF5AD85F3C26B5618C4F87E3479E7651A0B3AD4964167551FF86BA6531F026
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Effective Key Stream  - 5A20C51BF1AD73F35DFB65AE3C109D5D669F86C8B42A756DB011B5A08FE1D3EF7AF81D5354D104FE1CF39A2BFB567180938D7444365560DFA69D23588843
                        ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Output  Plaintext Hex - 636B436F6C6F72202020202020203D2020202648303046464138413826 0D0A202020426F726465725374796C65202020202020 3D20202031202027466978 65
Output Plaintext Text - c k C o l o r               =       & H 0 0 F F A 8 A 8 & ? ?       B o r d e r S t y l e           =       1     ' F i x e

Plaintext: [ckColor        =    &H00FFA8A8&??   BorderStyle     =   1  'Fixed Single??   Caption          =   "Vernam Two Algorithm"??   Client]
```

What about a non-repeating key to Xor with the plaintext, required for the Vernam algorithm?

- 4 pseudo-randomly selected key streams from the fixed 8,388,608 byte key are Xor'ed together. The pointers are changed for each block using the chain key array, producing up to 8,388,608 non-repeating Effective Key Streams.
- There are 6 sets of 65,536 files to prove this methodology produces non-repeating key streams ready for examination.
- The EKS streams were sorted so that the contents of each file contains streams with the same first 4 hex digits.
- After this presentation, proof is available that any of the 100 million entries in any of these files was produced with this single 8 Mbyte key.

What about a non-repeating key to Xor with the plaintext, required for the Vernam algorithm?

- This test created 6 sets of 65,536 files for 100 million blocks of Effective Key Streams, needed to encrypt 12.8 Gigabytes of plaintext, created using the 8,388,608 byte key and chain keys.
- With each pointer having a possibility between 0 and 8,388,607 inclusive, there are $8{,}388{,}608^4 = 4.951 \times 10^{27}$ sets of non-repeating Effective Key Streams of virtually any size that could be produced.
- These 6 sets of 65,536 files were produced using only six of the possible 4.951×10^{27} sets of 4 starting pointers. This should indicate how many possible strings of 100 million non-repetitive Effective Key Streams this methodology could produce, satisfying the requirement for the Vernam algorithm.

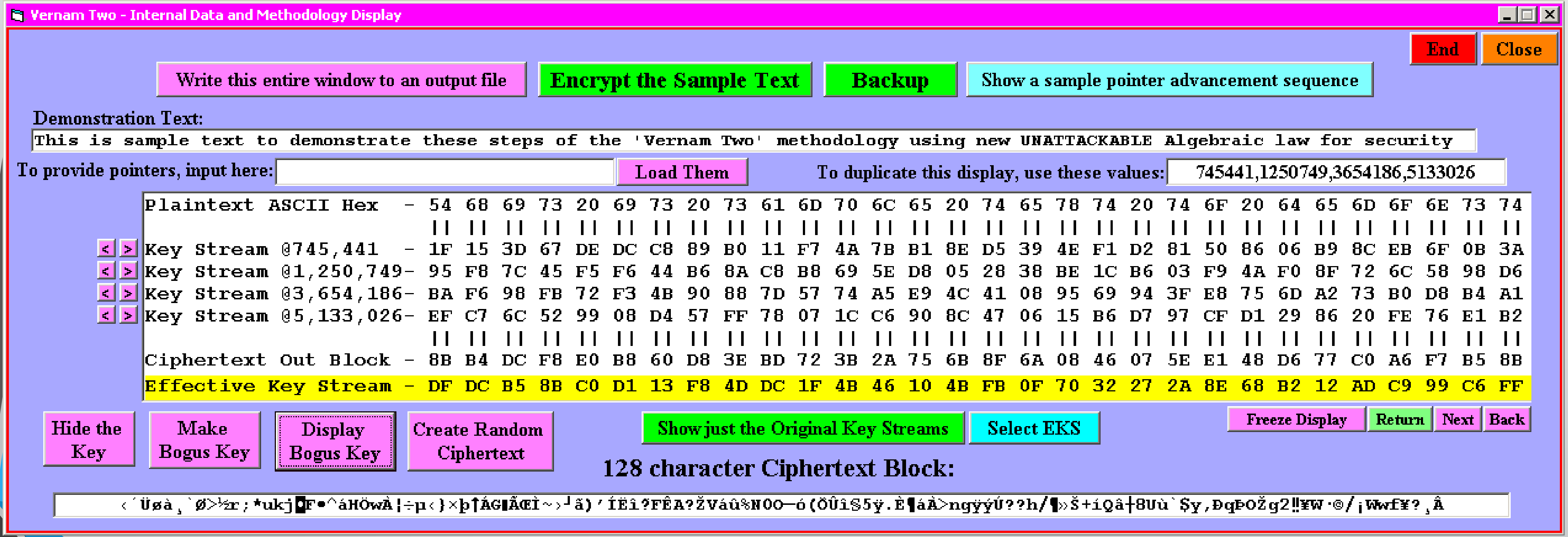
What about a non-repeating key to Xor with the plaintext, required for a Vernam algorithm?

- Even if a potential attacker could find two ciphertext files with blocks that have the same Effective Key Stream (EKS), ***Algebraic law prohibits the correct determination of the content of the 4 key streams used to create that EKS***.
- The app that produced these files used the first 25 Effective Key Stream hex numbers creating the 50-digit strings, plus the pointers used, recorded in each of the 65,536 files, about 140 Kbytes for each file.
- At the end of creating the files, it then opened each file and compared each EKS with every other EKS within the file.

What about a non-repeating key to Xor with the plaintext, required for a Vernam algorithm?

- It did not find any duplicates in any of the 65,536 files in any of the 6 example sets.
- Each example produced 256 files in each of 256 subdirectories. 128 subdirectories are available on each of 2 data DVD’s for each example for your examination, along with the app to prove they are correct.
- They will be shown and demonstrated later in this presentation.
- A test showed that no duplicates were encountered after 2 Billion blocks.

Demonstration Application in Demo mode, actual encryption example



Demonstration Text:
This is sample text to demonstrate these steps of the 'Vernam Two' methodology using new UNATTACKABLE Algebraic law for security

To provide pointers, input here:

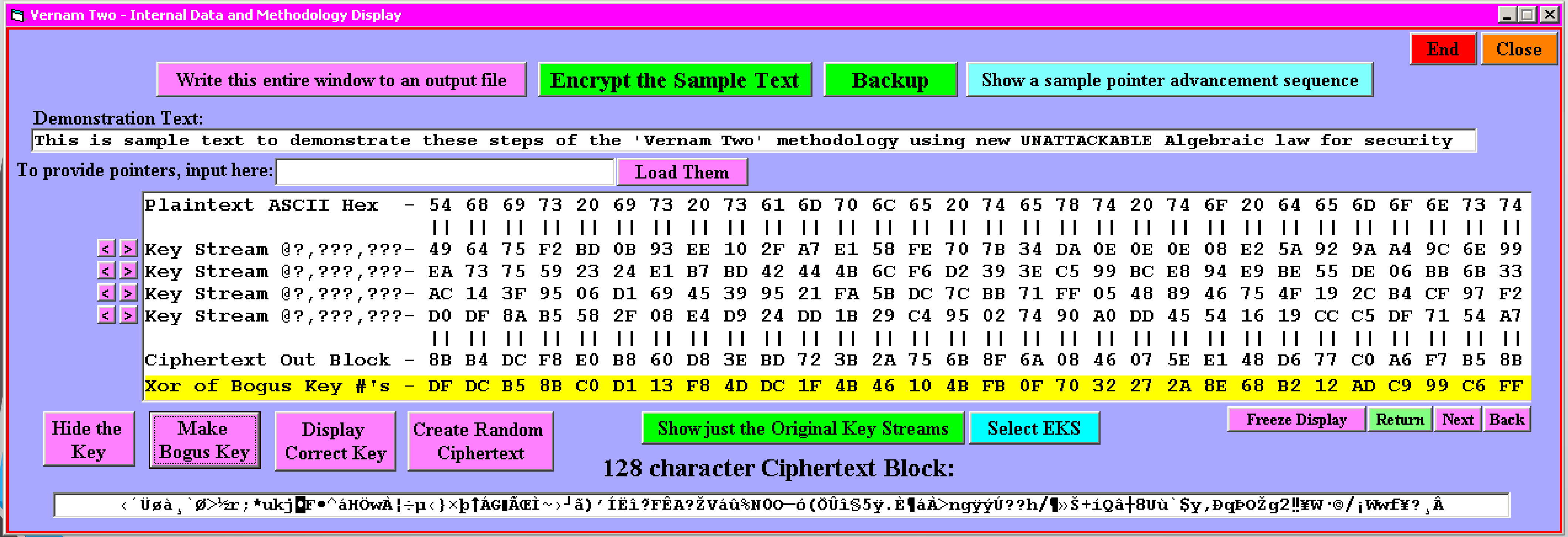
To duplicate this display, use these values: 745441,1250749,3654186,5133026

```
Plaintext ASCII Hex   - 54 68 69 73 20 69 73 20 73 61 6D 70 6C 65 20 74 65 78 74 20 74 6F 20 64 65 6D 6F 6E 73 74
                        || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Key Stream @745,441   - 1F 15 3D 67 DE DC C8 89 B0 11 F7 4A 7B B1 8E D5 39 4E F1 D2 81 50 86 06 B9 8C EB 6F 0B 3A
Key Stream @1,250,749- 95 F8 7C 45 F5 F6 44 B6 8A C8 B8 69 5E D8 05 28 38 BE 1C B6 03 F9 4A F0 8F 72 6C 58 98 D6
Key Stream @3,654,186- BA F6 98 FB 72 F3 4B 90 88 7D 57 74 A5 E9 4C 41 08 95 69 94 3F E8 75 6D A2 73 B0 D8 B4 A1
Key Stream @5,133,026- EF C7 6C 52 99 08 D4 57 FF 78 07 1C C6 90 8C 47 06 15 B6 D7 97 CF D1 29 86 20 FE 76 E1 B2
                        || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Ciphertext Out Block  - 8B B4 DC F8 E0 B8 60 D8 3E BD 72 3B 2A 75 6B 8F 6A 08 46 07 5E E1 48 D6 77 C0 A6 F7 B5 8B
Effective Key Stream  - DF DC B5 8B C0 D1 13 F8 4D DC 1F 4B 46 10 4B FB 0F 70 32 27 2A 8E 68 B2 12 AD C9 99 C6 FF
```

128 character Ciphertext Block:

```
‹´Üøà¸`Ø>½r;*ukj■F•^áHÖwÀ¦÷µ‹}×þ↑ÁG▮ÃŒÌ~›┘ã)ʹÍËî?FÊA?ŽVáû%NOO—ó(ÕÛî§5ÿ.È¶áÀ>ngÿýÚ??h/¶»Š+íQâ┼8Uù`$y,ÐqÞOŽg2‼¥W·©/¡Wwf¥?¸Â
```

Demonstration Application in Demo mode, bogus key encryption example



Demonstration Text:

This is sample text to demonstrate these steps of the 'Vernam Two' methodology using new UNATTACKABLE Algebraic law for security

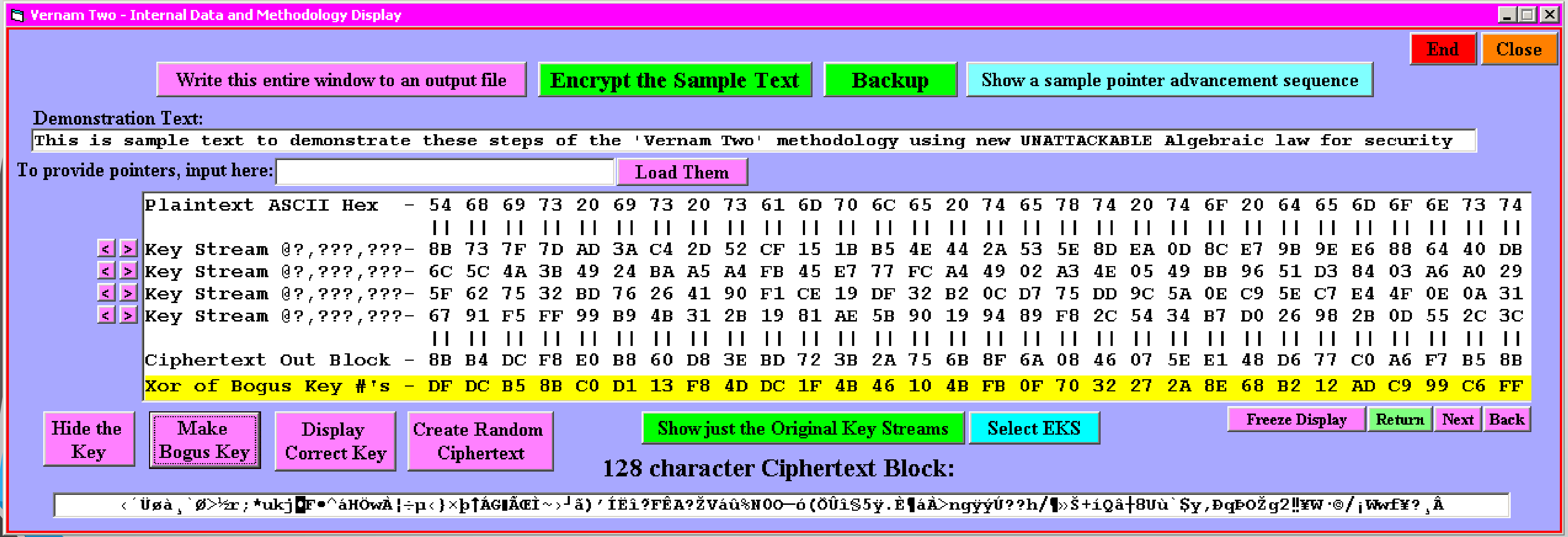
To provide pointers, input here:

```
Plaintext ASCII Hex  - 54 68 69 73 20 69 73 20 73 61 6D 70 6C 65 20 74 65 78 74 20 74 6F 20 64 65 6D 6F 6E 73 74
                       || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Key Stream @?,???,???- 49 64 75 F2 BD 0B 93 EE 10 2F A7 E1 58 FE 70 7B 34 DA 0E 0E 0E 08 E2 5A 92 9A A4 9C 6E 99
Key Stream @?,???,???- EA 73 75 59 23 24 E1 B7 BD 42 44 4B 6C F6 D2 39 3E C5 99 BC E8 94 E9 BE 55 DE 06 BB 6B 33
Key Stream @?,???,???- AC 14 3F 95 06 D1 69 45 39 95 21 FA 5B DC 7C BB 71 FF 05 48 89 46 75 4F 19 2C B4 CF 97 F2
Key Stream @?,???,???- D0 DF 8A B5 58 2F 08 E4 D9 24 DD 1B 29 C4 95 02 74 90 A0 DD 45 54 16 19 CC C5 DF 71 54 A7
                       || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Ciphertext Out Block - 8B B4 DC F8 E0 B8 60 D8 3E BD 72 3B 2A 75 6B 8F 6A 08 46 07 5E E1 48 D6 77 C0 A6 F7 B5 8B
Xor of Bogus Key #'s - DF DC B5 8B C0 D1 13 F8 4D DC 1F 4B 46 10 4B FB 0F 70 32 27 2A 8E 68 B2 12 AD C9 99 C6 FF
```

128 character Ciphertext Block:

```
‹´Üøà¸`Ø>½r;*ukj█F•^áHÖwÀ¦÷µ‹}×þ†ÁG▮ÅŒÌ~›┘ã)′ÍËî?FÊA?ŽVáû%NOO—ó(ÕÛî§5ÿ.È¶áÀ>ngÿýÚ??h/¶»Š+íQâ┼8Uù`$y,ÐqÞOŽg2‼¥W·©/¡Wwf¥?¸Â
```

Demonstration Application in Demo mode, bogus key encryption example



Demonstration Text:

This is sample text to demonstrate these steps of the 'Vernam Two' methodology using new UNATTACKABLE Algebraic law for security

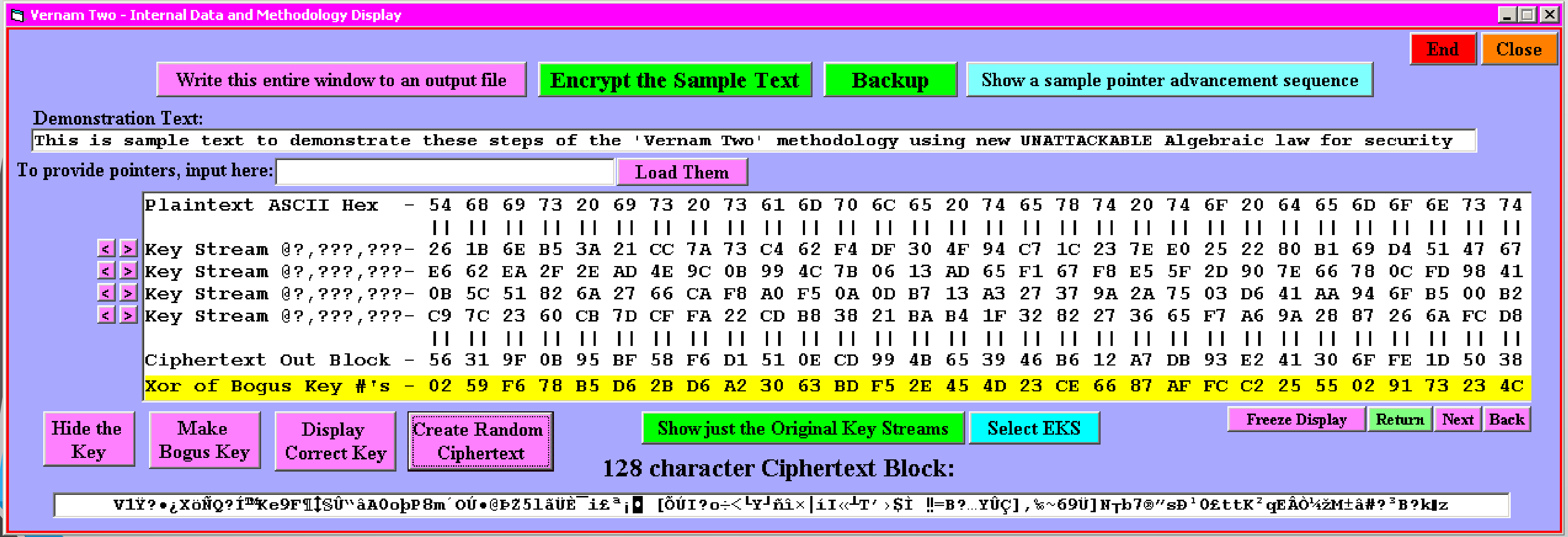
To provide pointers, input here:

```
Plaintext ASCII Hex  - 54 68 69 73 20 69 73 20 73 61 6D 70 6C 65 20 74 65 78 74 20 74 6F 20 64 65 6D 6F 6E 73 74
                       || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Key Stream @?,???,???- 8B 73 7F 7D AD 3A C4 2D 52 CF 15 1B B5 4E 44 2A 53 5E 8D EA 0D 8C E7 9B 9E E6 88 64 40 DB
Key Stream @?,???,???- 6C 5C 4A 3B 49 24 BA A5 A4 FB 45 E7 77 FC A4 49 02 A3 4E 05 49 BB 96 51 D3 84 03 A6 A0 29
Key Stream @?,???,???- 5F 62 75 32 BD 76 26 41 90 F1 CE 19 DF 32 B2 0C D7 75 DD 9C 5A 0E C9 5E C7 E4 4F 0E 0A 31
Key Stream @?,???,???- 67 91 F5 FF 99 B9 4B 31 2B 19 81 AE 5B 90 19 94 89 F8 2C 54 34 B7 D0 26 98 2B 0D 55 2C 3C
                       || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Ciphertext Out Block - 8B B4 DC F8 E0 B8 60 D8 3E BD 72 3B 2A 75 6B 8F 6A 08 46 07 5E E1 48 D6 77 C0 A6 F7 B5 8B
Xor of Bogus Key #'s - DF DC B5 8B C0 D1 13 F8 4D DC 1F 4B 46 10 4B FB 0F 70 32 27 2A 8E 68 B2 12 AD C9 99 C6 FF
```

128 character Ciphertext Block:

```
‹´Üøà¸`Ø>½r;*ukj█F•^áHÖwÀ¦÷µ‹}×þ†ÁG▌ÃŒÌ~›┘ã)′ÍËî?FÊA?ŽVáû%NOO—ó(ÕÛî§5ÿ.È¶áÀ>ngÿýÚ??h/¶»Š+iQâ┼8Uù`$y,ÐqÞOŽg2‼¥W·©/¡Wwf¥?¸Â
```

Demonstration Application in Demo mode, bogus ciphertext encryption example



Demonstration Text:

This is sample text to demonstrate these steps of the 'Vernam Two' methodology using new UNATTACKABLE Algebraic law for security

To provide pointers, input here:

```
Plaintext ASCII Hex  - 54 68 69 73 20 69 73 20 73 61 6D 70 6C 65 20 74 65 78 74 20 74 6F 20 64 65 6D 6F 6E 73 74
                       || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Key Stream @?,???,???- 26 1B 6E B5 3A 21 CC 7A 73 C4 62 F4 DF 30 4F 94 C7 1C 23 7E E0 25 22 80 B1 69 D4 51 47 67
Key Stream @?,???,???- E6 62 EA 2F 2E AD 4E 9C 0B 99 4C 7B 06 13 AD 65 F1 67 F8 E5 5F 2D 90 7E 66 78 0C FD 98 41
Key Stream @?,???,???- 0B 5C 51 82 6A 27 66 CA F8 A0 F5 0A 0D B7 13 A3 27 37 9A 2A 75 03 D6 41 AA 94 6F B5 00 B2
Key Stream @?,???,???- C9 7C 23 60 CB 7D CF FA 22 CD B8 38 21 BA B4 1F 32 82 27 36 65 F7 A6 9A 28 87 26 6A FC D8
                       || || || || || || || || || || || || || || || || || || || || || || || || || || || || || ||
Ciphertext Out Block - 56 31 9F 0B 95 BF 58 F6 D1 51 0E CD 99 4B 65 39 46 B6 12 A7 DB 93 E2 41 30 6F FE 1D 50 38
Xor of Bogus Key #'s - 02 59 F6 78 B5 D6 2B D6 A2 30 63 BD F5 2E 45 4D 23 CE 66 87 AF FC C2 25 55 02 91 73 23 4C
```

128 character Ciphertext Block:

```
V1Ÿ?•¿XöÑQ?Í™Ke9F¶‡§Û`'âA0oþP8m´OÚ•@ÞZ51ãÜÈ¯i£ª¡◘ [ÕÚI?o÷<└Y┘ñî×|íI«┴T′›$Ì ‼=B?…YÛÇ],%~69Ü]N┬b7®″sÐ¹0£ttK²qEÂÒ¼žM±â#?³B?k▌z
```

Is this a cryptography first?

- The first 3 bogus key streams were randomly created and the 4th stream was calculated to provide the needed results.
- Therefore, in any vertical column, 3 of the numbers can have any value from 0 to 255. There are 256^{3} = 16,777,216 possible sets of 4 key stream numbers for each column.
- With 128 columns per block, there are $16{,}777{,}216^{128}$ possible key streams that will result in the plaintext to ciphertext conversion. Attackers have no single key goal.
- How many keys will correctly translate an AES ciphertext to the plaintext? Do attackers have a one key goal to reach?

What should be the conclusions of this new design?

- The fixed key is protected from discovery by Algebraic law.
- The plaintext is protected from discovery by the Vernam Algorithm.
- The values of the pointers are protected by simple mathematics.
- There is no mathematical process available that could ever distinguish the plaintext ciphertext from the pointer ciphertext because both processes use the same methodology of encryption.
- The pointer ciphertext characters are pseudo-randomly mixed together with the plaintext ciphertext characters in different positions in different orders in different ciphertext files.
- The two processes (plaintext and pointer processing) use different sets of 4 pseudo-randomly set pointers.