Dillon Li

Project 2:
CSS Cryptosystem

Function Overview

EncryptCSS.m : This function encrypts a 40 bit stream (or 5 character word) using another 40 bit key (can also be a 5 character word). Done using bitXOR.

Inputs:

Bitstream – The 5 character word to be encrypted

Key – The 5 character (40 bit) key used for encryption. Converting from ASCII directly sometimes gives 7 bits, so I padded to make each character 8. This key will also be necessary for decryption.

Outputs:

Output_bitstream – This is the encrypted bitstream.

Output_plaintext – What the encrypted bitstream looks like as text (typically nonsensical)

Orig_bitstream – What the unencrypted word looks like as a 40 bit stream

DecryptCSS.m: This function decrypts an encrypted 40 bit stream using the same key it was encrypted with. Done using bitXOR.

Inputs:

Bitstream – The 40 bit encrypted stream. This is the output, 'output_bitstream' received from EncryptCSS.m

Key – The same key used to encrypt the original 40 bit stream (5 character word)

Outputs:

Orig_stream – The original 40 bit stream (stored in orig_bitstream output of EncryptCSS for comparison)

Orig_text – The original 5 character word that was encrypted.

Note: This function does not take in text for the bitstream since it is usually nonsensical. It only works with encrypted 40 bit streams.

CSSBitstream.m: This function is used to generate the pseudo-random 40 bit stream that will be XOR'd with input streams for encryption and decryption.

Input:

Passcode – Your 40 bit (5 character) key.

Output:

Bytestream – The pseudo-random bit stream.

Process Overview

As detailed in how CSS is performed, five keystream bytes are generated using the provided key by splitting the key into two linear feedback shift registers and running the results through a full adder. The actual process details an 8-bit full adder in which the carries will go into each new byte. This is the same as five 8-bit adders chained together, so in my implementation, I used a 40-bit adder for simplicity.

CSSBitstream is where most of the logic takes place. This function generates the keystream used for encryption and decryption, and the encryption and decryption functions simply take the output of this function and perform bitwise XOR operations. The principle behind encryption and decryption is as follows:

Encryption: Orig_bitstream XOR keystream = encrypted_stream

Decryption: encrypted_stream XOR keystream = Orig_bitstream

As for the implementation of the keystream generation itself, two polynomials are used to determine the next most significant bit to be placed into each LSFR after the most significant bit is harvested. Due to the nature of the polynomials and the way in which I stored the bit strings, the indices and the polynomial exponents are not the same. I had to account for this when performing XOR operations to determine what the next bit would be after shifting (first polynomial XORs the 17^{th} bit with something, but in my implementation, the 17^{th}, or most significant bit, is actual in index 1).

A carry of 0 is initialized, along with a variable containing space for 40 bits that is left blank to be filled as adding occurs. In the case of 0 or 1 sums, the adder places that number in the current index and the carry is 0. In the case of 2, sum is 0 and carry is 1. In the case of 3, both sum and carry are 1.

In this method, as LFSRs harvest and shift, and then harvested bits are added, the keystream is generated.

Snapshots demonstrating encryption/decryption:

Encryption:

```
Trial>> [bits text stream] = EncryptCSS('Anime', 'crypt')

bits =

    '01100011100000011011100110010010101100011'


text =

    'c ¹'³'


stream =

    '01000001011011100110100101101101011001 01'
```

Decryption:

```
Trial>> [orig_stream orig_text] = DecryptCSS(bits, 'crypt')

orig_stream =

    '01000001011011100110100101101101011001 01'


orig_text =

    'Anime'
```

Keystream (function is called in EncryptCSS and DecryptCSS):

```
Trial>> CSSBitstream('crypt')

ans =

    '00100010111011111101000011111111111010110'
```

Some Analysis/Attack

The randomness of the keystream is somewhat dependent on the polynomials used. The ones selected seem to do a decent job of it. In this example, the encrypted bitstream has a somewhat similar number of 1's and 0's to the original one, meaning there is no easily discernable pattern.

Breaking this cipher, however, is not actually that difficult. If some prior information is known concerning one or two of the bytes, this information could be used to decipher the keystream fairly easily by testing via cycling through the remaining bytes.

Even without prior information, the key is only 40 bits long. This is within feasible range of a brute force attack. By trying every possible key, and checking for feasible information, it would easy to figure out the key.