Gift Cipher

3-cliqued

Charanesh Raj Pusarla
Tarun Singh
Swetha Reddy Bathala

Introduction

- A light weight Cryptography cipher, it received a lot of attention during the past decade
- Used majorly in IoT and wireless communication
- More energy efficient than many other ciphers
- There are 2 implementations of GIFT cipher. The gift-64 and gift-128. The difference is between the size of plaintext inputs that they accept.
- However both the implementations use 128-bit length keys

Cipher Specifications

- The cipher we tried to implement is the GIFT-64 cipher.
- This uses a 128-bit key and a 64-bit plaintext and gives a 64-bit ciphertext after encryption.
- The GIFT-64 that we implemented consists of 28 rounds, each of which consists of 3 steps: SubCells, PermBits, AddRoundKey.
- The bits of the data could be represented in 1D, 2D, etc. ways. For our application we have just used the 1D representation.
- We have used this cipher implementation to encryption and decryption of text, images, and audio files.

Some pre calculations and values...

- Here is the Sbox used:

```
Sbox = [0x1, 0xa, 0x4, 0xc, 0x6, 0xf, 0x3, 0x9, 0x2, 0xd, 0xb, 0x7, 0x5, 0x0, 0x8, 0xe]
```

- Permutation bits calculated using

```python
def perm_bit_calc_gift64(i):
    P_i = 4*(i//16) + 16*(((3*((i%16)//4) + i%4)%4)) + i%4
    return P_i
```

- This returns the following table:

```
[0, 17, 34, 51, 48, 1, 18, 35, 32, 49, 2, 19, 16, 33, 50, 3, 4, 21, 38, 55, 52, 5, 22, 39,
36, 53, 6, 23, 20, 37, 54, 7, 8, 25, 42, 59, 56, 9, 26, 43, 40, 57, 10, 27, 24, 41, 58, 11,
12, 29, 46, 63, 60, 13, 30, 47, 44, 61, 14, 31, 28, 45, 62, 15]
```

- Round constants used:

```
[0x01,0x03,0x07,0x0f,0x1f,0x3e,0x3d,0x3b,0x37,0x2f,0x1e,0x3c,0x39,0x33,0x27,0x0e,0x1d,0x3a,0x35
,0x2b,0x16,0x2c,0x18,0x30,0x21,0x02,0x05,0x0b]
```

Encryption Function

Input plaintext and Key

Round key generation

These 3 steps repeat for 28 rounds

```python
def encrypt_using_gift64(text, main_key):
    round_keys=generate_round_keys_from_main_key(main_key)
    text=hex_to_bin_str(text)
    #states1.append(text)
    for round_number in range(28):
        text=sub_cells(text)
        #states1.append(text)
        text=permutation_bits(text)
        #states1.append(text)
        text=add_round_key_and_round_constant(round_keys,text,round_number)
        #states1.append(text)
    return text
```

Outputs ciphertext

- This implementation takes in 64-bit plaintext and 128-bit key in hex formats

Decryption Function

Input ciphertext and Key

Round key generation

These 3 steps repeat for 28 rounds

```
def decrypt_using_gift64(text, main_key):
    round_keys=generate_round_keys_from_main_key(main_key)
    text=hex_to_bin_str(text)
    for round_number in range(27,-1,-1):
        #states2.append(text)
        text=add_round_key_and_round_constant(round_keys,text,round_number)
        #states2.append(text)
        text=rev_permutation_bits(text)
        #states2.append(text)
        text=rev_sub_cells(text)
    #states2.append(text)
    return text
```

Outputs plaintext

- This implementation takes in 64-bit ciphertext and 128-bit key in hex formats
- We can see that decryption function is almost the reverse of encryption function

The 3 sub-steps

1. SubCells:

```
def sub_cells(text):
    ans=[]
    for i in range(16):
        ans.append(hex(Sbox[int(text[4*i:4*i+4], 2)])[2:])
    for i in range(16):
        ans[i]=hex_to_bin_str(ans[i])
    return ''.join(ans)
```

2. PermBits:

```
def permutation_bits(text):
    text=text[::-1]
    new_text_permuted=[None]*len(text)
    for i in range(64):
        new_text_permuted[Permutation_bits[i]]=text[i]
    new_text_permuted=''.join(new_text_permuted)
    new_text_permuted=new_text_permuted[::-1]
    return new_text_permuted
```

3. AddRoundKey:

```
def add_round_key_and_round_constant(round_keys,text_in,round_number):

    round_key=round_keys[round_number]
    u, v = round_key[:16], round_key[16:]
    u=[int(x) for x in u]
    v=[int(x) for x in v]
    text=[int(x) for x in text_in]

    u.reverse()
    v.reverse()
    text.reverse()

    for j in range(16):
        text[4*j+1]^=u[j]
        text[4*j]^=v[j]

    round_constant=('00000000'+hex_to_bin_str(hex(round_constants[round_number])[2:]))[-6:]
    round_constant=[int(x) for x in round_constant]
    round_constant.reverse()
    text[63]^=1
    text[3]^=round_constant[0]
    text[7]^=round_constant[1]
    text[11]^=round_constant[2]
    text[15]^=round_constant[3]
    text[19]^=round_constant[4]
    text[23]^=round_constant[5]
    text.reverse()

    return ''.join(map(str, text))
```

How we used it for text encryption:

Since the gift64 can take in a fixed size input of 64 bits. So to encrypt variable size text files, here is what we did:

TextFile → binary data → break into 64-bit subparts → encrypt each part → combine the result

Similarly to decrypt the text file:

EncryptedText → break into 64-bit subparts → decrypt each part → text string → combine the result

- However, better methods may exist which are more secure than this but for the ease of implementation, we decided to go with this method.

How we used it for image encryption:

- Handling image files was a little bit more difficult. We took images and extracted their binary data. The python open function can do that.
- Then we used python **base64 encoding** scheme to convert this binary data into string. Then this string is encrypted and decrypted as we saw before.
- Later, the decrypted string can be converted back into binary data by using **base64 decoding** which can be converted to .jpg image file.

.jpg image file → binary data → string → encrypt → decrypt → string → binary data → .jpg image file

Encrypting & Decrypting Audio? (Brownie Points?)

- Handling audio files was difficult. We used the Python **Wave** module to convert audio files of type **.wav into binary data.**
- Then we used python **base64 encoding** scheme to convert this binary data into string. Then this string is encrypted and decrypted as we saw before.
- Later, the decrypted string can be converted back into binary data by using **base64 decoding** which can be converted to .wav audio file.

.wav audio file → binary data → string → encrypt → decrypt → string → binary data → .wav audio file

- Note for converting binary data into .wav file we will also need **parameters** which are derived from the original audio. So these parameters will be stored in **another file** that will be required during decryption.

Cryptanalysis

- To find differential features, we utilized a method called Mixed Integer Linear Programming (MILP). As a result, for varying numbers of rounds, we determined the lower limits for the number of active Sboxes in both DC and LC
- It's worth mentioning that the optimal differential characteristic in most cases is the one with the fewest active Sboxes. Following differential characteristics contain much more active Sboxes than the original characteristic under the same input and output differences. As a result, the differential probability is close to the probability of the ideal differential characteristic.

Related Key Cryptanalysis

- GIFT-64 uses 32-bit round keys that are derived from the 128-bit main key
- For all the 128-bits of the main key to be involved, it will take at-least 128/32 = 4 rounds
- Thus, we could say it is possible that there may not be any active S-box from round 1 to round 4. So we need to begin with related key cryptanalysis-based calculations from round 5 onwards
- When we analyze the probabilities of differential features, we can see that the likelihood of the 12-round characteristic has a lower bound of 2. (-33). As a result, 28 rounds may not be especially safe against cryptanalysis of associated keys
- We could increase the number of rounds in our implementation to reduce the probability of of such attacks

Some Observations

The key state is merely XOR with half of the text at some predetermined locations, not with all of the bits, as can be seen here.Furthermore, the XOR occurs with the same bits every round. This improves efficiency as well.

```
def add_round_key_and_round_constant(round_keys,text_in,round_number):

    round_key=round_keys[round_number]
    u, v = round_key[:16], round_key[16:]
    u=[int(x) for x in u]
    v=[int(x) for x in v]
    text=[int(x) for x in text_in]

    u.reverse()
    v.reverse()
    text.reverse()

    for j in range(16):
        text[4*j+1]^=u[j]
        text[4*j]^=v[j]

    round_constant=('00000000'+hex_to_bin_str(hex(round_constants[round_number])[2:]))[-6:]
    round_constant=[int(x) for x in round_constant]
    round_constant.reverse()
    text[63]^=1
    text[3]^=round_constant[0]
    text[7]^=round_constant[1]
    text[11]^=round_constant[2]
    text[15]^=round_constant[3]
    text[19]^=round_constant[4]
    text[23]^=round_constant[5]
    text.reverse()

    return ''.join(map(str, text))
```

To remove any symmetry, each bit of the 6-bit round constant is XORed with a distinct nibble in the text.

Some Observations

The round key generation mechanism is built in such a manner that all 128 bits of the main key may be incorporated in the round keys in the shortest amount of time feasible.

```python
def generate_round_keys_from_main_key(main_key):
    round_keys=[]
    round_key=hex_to_bin_str(main_key)
    aa,bb='',''
    for _ in range(28):
        round_keys.append(round_key[-32:])
        a=round_key[-16:]
        b=round_key[-32:-16]
        aa=a[-12:]+a[:-12]
        bb=b[-2:]+b[:-2]
        round_key=bb+aa+round_key[:-32]
    return round_keys
```

Observations

- Our GIFT Sbox can be implemented with 4N+6X operations (smaller than the Sboxes in PRESENT and RECTANGLE),
- It has a maximum differential probability of 2−1.415 and linear bias of 2−2 , algebraic degree 3 and no fixed point.
- For the sub-optimal differential transitions with probability 2−1.415 , there are only 2 such transitions and the sum of Hamming weight of input and output differences is 4

	Type	Rounds	Time	Memory	Data	Source
GIFT-64	Integral	14	2^{96}	2^{63}	2^{63}	[2]
GIFT-64	MitM	15	2^{120}	2^{8}	2^{64}	[2]
GIFT-64	MitM	15	2^{112}	2^{16}	2^{64}	[14]
GIFT-64	Differential	19	2^{112}	2^{80}	2^{63}	Ours
GIFT-128	Differential	22	2^{114}	2^{53}	2^{114}	Ours

Conclusion

- The GIFT fixsliced representation enables software bitsliced solutions to be exceedingly efficient
- GIFT-64 outperforms all other 64-bit ciphers, except SPECK-64/128
- The performance of GIFT-128 is 1.6 times better than GIFT-64
- GIFT-128 implementations largely outperform the current AES-128 standard
- GIFTb-128 saves about 28% of cycles when compared to AES-128.

THANKING YOU