GMOCK-256

Constraints

We use a 32 bit key in order to maximize the keyspace of our algorithm. Our reliance on the SHA-256 hashing algorithm naturally lends itself to the use of a 256 bit block size.

Encryption Algorithm

Given initial key $\mathbf{\hat{K}}$, we first obtain XOR key $\mathbf{X}_0$ and permutation key $\mathbf{K}_0$ using our key scheduling algorithm:

1. Select 32 bit key $\mathbf{\hat{K}}$.
2. Take $SHA_{256}(\mathbf{\hat{K}})$ to find XOR Key $\mathbf{X}_0$.
3. Apply Mostyn reduction (see later section) to $\mathbf{X}_0$ to receive permutation key $\mathbf{K}_0$.

We then encrypt the first plaintext block $\mathbf{P}_0$ using our encryption scheme:

1. Using $\mathbf{K}_0$, apply the shuffle algorithm (see later section) to $\mathbf{P}_0$ to obtain scrambled plaintext block $\mathbf{\tilde{P}}_0$.
2. Generate the encrypted block $\mathbf{E}_0$ as $\mathbf{\tilde{P}}_0 \oplus \mathbf{X}_0$.

In order to encrypt plaintext block $\mathbf{P}_{i+1}$, take the XOR key $\mathbf{X}_i$ and compute $\mathbf{X}_{i+1}$ as $SHA_{256}(\mathbf{X}_i)$.

Decryption Algorithm

We first use the key scheduler defined in an earlier section to derive keys $\mathbf{X}_i$ and $\mathbf{K}_i$. We then decrypt encrypted message block $\mathbf{E}_i$ by first finding the scrambled plaintext via an XOR:

$$\mathbf{X}_i \oplus \mathbf{E}_i = \mathbf{\tilde{P}}_i$$

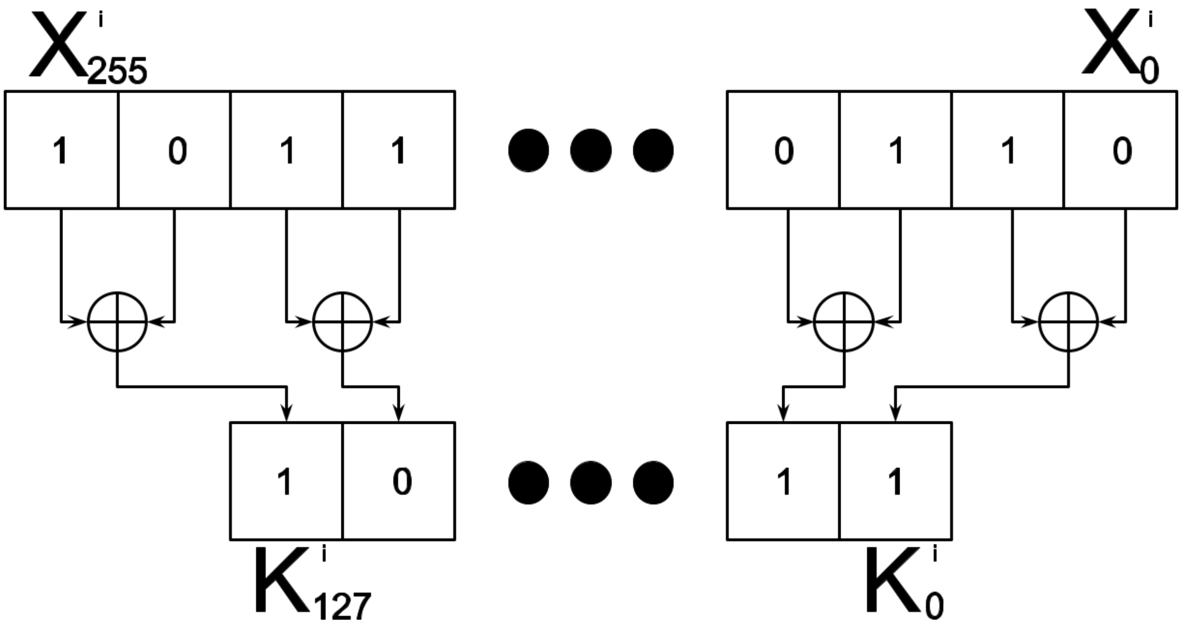
$\mathbf{\tilde{P}}_i$ can then be transformed to $\mathbf{P}_i$ by undoing the shuffle algorithm (see later section).

Mostyn Reduction

We obtain permutation key $\mathbf{K}_i$ as the Mostyn reduction of $\mathbf{X}_i$, defined as:

$$\mathbf{K}_{i,j} = \mathbf{X}_{i,2j} \oplus \mathbf{X}_{i,2j+1}$$

Where $\mathbf{A}_i \oplus \mathbf{B}_j$ is the bitwise XOR of bit i in **A** and bit j in **B**.



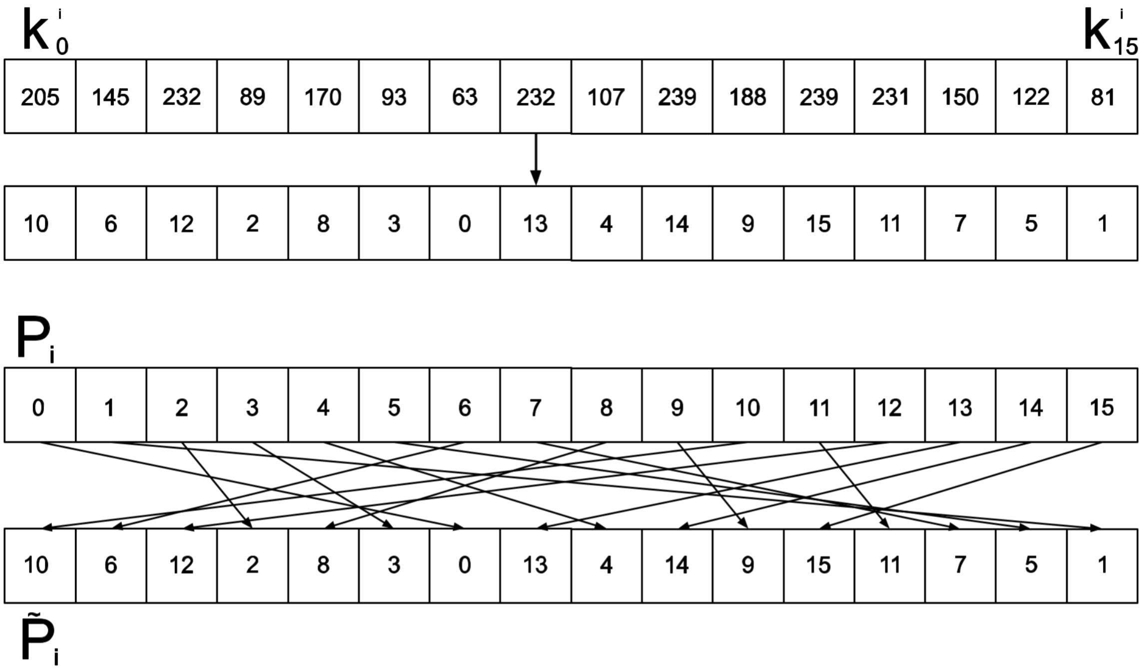
Shuffle Algorithm

Using the ith block's permutation key $\mathbf{K}_i$, we permute plaintext block $\mathbf{P}_i$ to $\tilde{\mathbf{P}}_i$ by first splitting $\mathbf{K}_i$ into eight 8-bit subkeys $\mathbf{k}_i0$, $\mathbf{k}_i1$, …$\mathbf{k}_i15$, which are interpreted as unsigned integers. We then take the 8 corresponding 16-bit chunks of plaintext $\mathbf{P}_i$ and stable sort[1] them in ascending order on their associated subkeys, yielding scrambled plaintext $\tilde{\mathbf{P}}_i$.

<u>Ex.</u>
$\mathbf{K}_i$ = 110011011001000111101000010110011010101001011101001111111110100001101011
$11101111101111001110111111100111100101100111101001010001_2$
⇒ ($\mathbf{k}_i0$, $\mathbf{k}_i1$, …$\mathbf{k}_i15$) = (205, 145, 232, 89, 170, 93, 63, 232, 107, 239, 188, 239, 231, 150, 122, 81)
which then permutes
$\mathbf{P}_i$ with labeled 16 bit blocks (0, 1, .. 15) to (10, 6, 12, 2, 8, 3, 0, 13, 4, 14, 9, 15, 11, 7, 5 , 1)

---

[1] That is, we sort in such a way that if $\mathbf{k}_i a = \mathbf{k}_i b$ and a < b, then $\mathbf{k}_i a$ will be sorted before $\mathbf{k}_i b$. See [here] for further reading regarding sort stability.



In order to unscramble $\tilde{\mathbf{P}}_i$ during decryption, label the sixteen 16 bit chunks of $\tilde{\mathbf{P}}_i$ as ($\mathbf{p}_i0$, $\mathbf{p}_i1$, … $\mathbf{p}_i15$) and reorder them to match the stable ordering of the permutation subkeys ($\mathbf{k}_i0$, $\mathbf{k}_i1$, …$\mathbf{k}_i15$).

Security Analysis

Resistance to Brute Force Attacks

GMOCK-256 is not necessarily resistant to brute force attacks because its key length is only 32 bits, making the number of possible effective keys at most 2^{32}. However, the only way to increase our resistance to brute force attacks without expanding our key size is to increase the computational demands of the algorithm (perhaps by changing our hash function to the larger SHA_{2048}). We deem this to be an unacceptable tradeoff, as it negatively impacts the end user's experience.

Nonlinearity of Algorithm

SHA_{256} is a highly nonlinear hash function, and provides nearly all of our cipher's entropy. Any linear attack on our cipher would require two plaintext blocks to be encrypted using the same XOR key, which is unlikely due to the low rate of hash collision observed in SHA_{256}. Thus, our algorithm is, prima facie, secure against linear cryptanalysis.

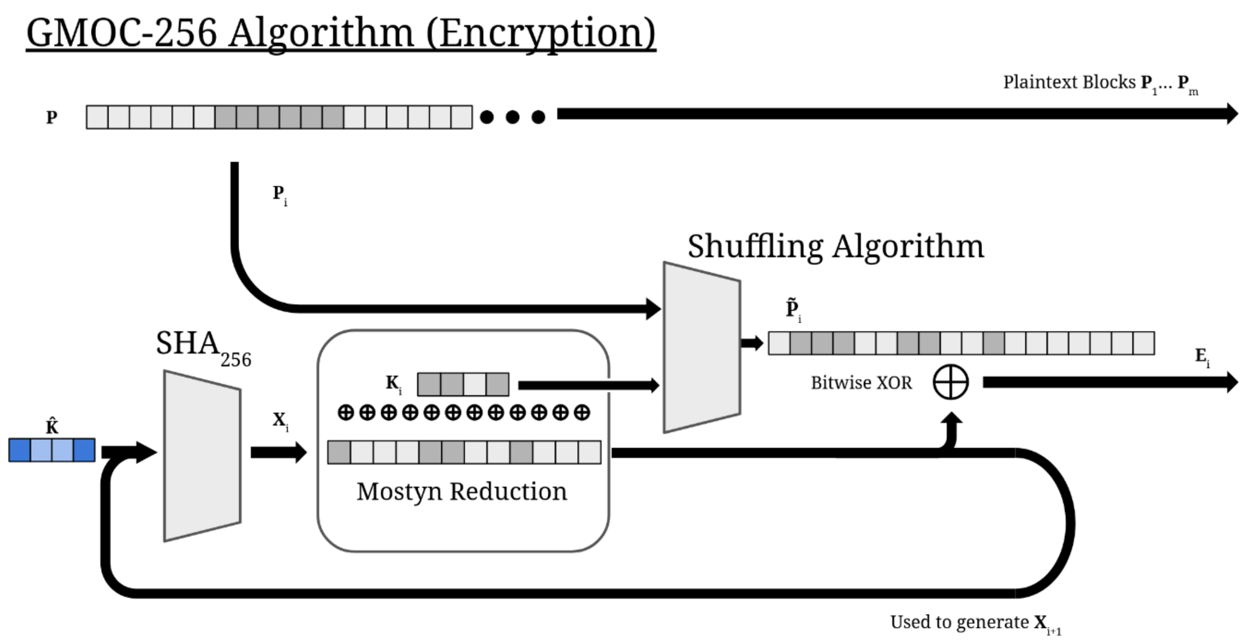
Diffusion of Algorithm

The use of a shuffle algorithm allows diffusion of the message, adding resistance against differential cryptanalysis.

Resistance to Other Cryptanalytic Methods

Because each XOR key in our algorithm is likely to be unique due to the low collision rate of SHA_{256}, we find the viability of known plaintext attacks on GMOCK-256 dubious.

Pseudocode and Diagrams




```
// Encryption
key = initial 32-bit key K̂
for each 256-bit block of plaintext
   xor_key = SHA256(key)
   permutation_key = mostyn_reduction(xor_key)
   shf_plaintext = stable_sort(plaintext_block, permutation_key)
   encrypted_block = shf_plaintext ^ xor_key
end loop
```

```
// Decryption
key = initial 32-bit key K̂
for each 256-bit block of cipher text
   xor_key = SHA256(key)
   permutation_key = mostyn_reduction(xor_key)
   shf_plaintext = cipher_block ^ xor_key
   plaintext_block = unsort(shf_plaintext, permutation_key)
end loop
```