Full-Time-Pad
Symmetric Stream Cipher

Improved One-Time-Pad Encryption Scheme

Taha Canturk
kibnakanoto@protonmail.com

2024-05-20

Version 1.0

Abstract

`One-Time-Pad` Encrypion Scheme is a secure algorithm but there are 2 main security risks. One, a key cannot be reused. Two, plaintext length equals key length which is very inefficient when dealing with long plaintexts. These 2 security risks only exist due to a lack of confusion and diffusion per ciphertext. As denoted by Claude Shannon in the report he published in 1945, A Mathematical Theory of Cryptography, A secure cryptographic algorithm requires confusion and diffusion. The `Full-Time-Pad` symmetric stream cipher is developed based on the `One-Time-Pad` with solutions to the security risks while maintaining high speed computation. To achieve diffusion, the key is permutated in it's byte array form using a constant permutation matrix. To acheive the confusion, the key is manipulated in it's 32-bit integer representation using Modular **A**ddition in F_p, Bitwise **R**otations, and **X**or (**ARX**). The permutation guarantees that every time there is a manipulation, each 32-bit number is made up of a different byte order.

Contents

1 Introduction

1.1 Pre-requisite Terminology

Key	32-byte random array that's transformed, then hashed before XORed with the plaintext to encrypt
Symmetric	Same key is used for encryption and decryption
Stream	Plaintext is encrypted without seperating it into blocks
Plaintext	Plain data before encryption
Ciphertext	Encrypted plaintext
Cipher	Encryption algorithm. Plaintext is transformed into a ciphertext that can only be reversed with a key
Diffusion	plaintext/key is spread out in the ciphertext
Confusion	The ciphertext has no possible statistical analysis, or cryptoanalysis to determine the plaintext
Bit	0 or 1. Smallest discrete unit for computation
Byte	8-bit number
Galois Field	Finite Field where there are only limited number of numbers. Only prime galois fields (F_p) are used where size of the field is denoted by prime number p
Avalanche Effect	An aspect of diffusion. If smallest unit (1 bit) of data is changed, the ciphertext changes in an unrecognizable way.
Permutation	transposing (moving) elements of a vector without actually changing their values. A reversible operation

1.2 Applications

This encryption algorithm is fast yet reliable as the avalanche effect is strong and the confusion operations used have proved themselves in cryptography. If the generated key is potentially found due to a side-channel attack or any other method not relating to the specifics of this algorithm, then all plaintexts can be found. However this isn't a pressing issue as all encryption algorithms offer this flaw. In contrast, without this flaw, a new key would have to be generated per plaintext like in One-Time-Pad which is inefficient.

1.3 Key Generation

The 32-byte key should be generated using a cryptographically secure method, including but not limited to cryptographic random number generators and Elliptic Cryptography Diffie Hellman (ECDH) protocol with Hash-based Key Derivation Function (HKDF).

1.4 Prerequisite Mathematics

1.4.1 Galois Field Addition

In a prime galois field, where two variables (x, y) are added, $x, y \in \mathbb{F}_{\shortmid}$ where $/F_p$ denotes the finite field with a prime field size. There are finite number of elements in F_p which means that $x, y < p$.

In the expression:
$x + y = z$
$x, y, z \in \mathbb{F}_7$

and implicitly denotes
$x + y = z \pmod 7$

In the context of this encryption algorithm, it's used in the tranformation algorithm as one of three operations (**A**RX).

1.4.2 Vector Permutation

2 Security Vulnerabilities

In One-Time-Pad, key isn't reusable. Here is the proof:

`let` m_1, m_2 `be 2 plaintexts`
`let` k `be the key`
`let` $c_1 = m_1 \oplus k$
`let` $c_2 = m_2 \oplus k$
$c_1 \oplus c_2 = (m_1 \oplus k) \oplus (m_2 \oplus k)$
$c_1 \oplus c_2 = m_1 \oplus m_2$

Since the key is reused, the 2 ciphertext's XORed factor out the key since $k \oplus k = 0$
Using cryptoanalysis, the 2 plaintexts can be found.
For $c_1 \oplus c_2 = m_1 \oplus m_2$ to not hold true, for each encryption, the key needs to be different. If k is transformed each time so that it has an avalanche effect. Even with no confusion, it would still be secure since $k' \oplus k \neq 0$ where k' is transformed key.
But there is another concern,
What if the plaintext and ciphertext are known, then it is possible to find k so don't use k without transformation, since `plaintext` $\oplus$ `ciphertext` $=$ `key`. So for each plaintext, key needs to be transformed irreversibly and it also requires confusion since if k' is found,

k is still unknown but if k is found, then all instances of k'_n are known, which means that:

$k'_1 = hash(k+1)$ `where hash() is an irreversible transformation`
$k'_2 = hash(k+2)$
$c_1 \oplus c_2 = (m_1 \oplus k'_1) \oplus (m_2 \oplus k'_2)$
$c_1 \oplus c_2 \neq m_1 \oplus m_2$
$m_1 \oplus c_1 = k'_1$
$m_2 \oplus c_2 = k'_2$
k'_1, k'_2 `are calculated using an irreversible hashing algorithm`

$\therefore$ the `Full-Time-Pad` Cipher requires both diffusion and confusion

2.1 Brute-Force

Due to the use of a galois field. The total number of combinations per 256-bit key isn't $a = 2^{256}$, but rather $b = 4294967291^8$ where $p = 4294967291$ for arithmetic in F_p and there are 8 32-bit numbers in a 256-bit key.

$a = 115792089237316195423570985008687907853269984665640564039457584007913129639936_{10}$
$b = 115792088158918333131516597762172392628570465465856793992332884130307292657121_{10}$
`let` $\Delta = a - b$
$\Delta = 1078397862292054387246515515224699519199783770047124699877605836982815_{10}$

So the difference Δ is a somewhat large integer. The number of combinations with a galois field is lower than without a galois field ($b < a$). This isn't a big concern as their difference measured exponentially is only around $2^{\log_2 \Delta} \approx 2^{229}$ which means that their difference is around 2^{229}, this is a negligible difference as the difference between 2^{230} and 2^{229} is also huge.
$\therefore$`Using a galois field doesn't negatively impact number of cominations in terms of brute force as the total number of combinations when using a galois field vs not is a negligible amount`

2.1.1 Birthday Problem

The birthday problem is a paradox. It goes as follows: how many people are required so that there is more than 50% chance that at least 2 people have the same birthday.
The answer is an unexpected 23 people.
In the context of this encryption algorithm, it might be a concern, as number of key reused (with transformation) increase, the chances of finding the key increase:

`let` V_c `be the number of combinations per key without order and repetitions`
`let` k `be the number of keys needed for hash(key) to have a 50% chance to equal another hash(key)`
`let` V_t `be the number of combinations per key with order and repetitions`

$$V_c = \frac{b!}{(b-k)!} = \frac{4294967291^8!}{(4294967291^8 - k)!}$$
$$V_t = b^k = 4294967291^{8^k}$$
$$P(A) = \frac{V_c}{V_t}$$
$$P(A) = \frac{\frac{b!}{(b-k)!}}{b^k}$$
$$P(B) = 1 - P(A) = 50\%$$
$$P(A) = 1 - 50\%$$
$$1 - 50\% = \frac{\frac{b!}{(b-k)!}}{b^k}$$
$$\frac{1}{2}b^k = \frac{b!}{(b-k)!} \qquad \texttt{since } 50\% = \frac{1}{2}$$
$$\log_b \frac{1}{2}b^k = \log_b \frac{b!}{(b-k)!}$$
$$\log_b \frac{1}{2} + \log_b b^k = \log_b b! - \log_b (b-k)!$$
$$0 = \log_b b! - \log_b (b-k)! - \log_b \frac{1}{2} - k \qquad \texttt{since } \log_b b^k = k$$

According to Ramanujan's Approximation:

$$\log_b b! \approx \frac{b\ln b - b + \frac{\ln\left[\frac{1}{\pi^3}+b(1+4b(1+2b))\right]}{6} + \frac{ln\pi}{2}}{\ln b}$$

And

$$\log_b(b-k)! \approx \frac{(b-k)\ln(b-k) - (b-k) + \frac{\ln\left[\frac{1}{\pi^3}+(b-k)(1+4(b-k)(1+2(b-k)))\right]}{6} + \frac{\ln\pi}{2}}{\ln b}$$

Recall:

$$0 = \log_b b! - \log_b(b-k)! - \log_b \frac{1}{2} - k \qquad \texttt{isolate } \log_b(b-k)!$$

$$\log_b(b-k)! = \log_b b! - \log_b \frac{1}{2} - k$$

Combine both equations for $\log_b(b-k)!$:

$$\log_b b! - \log_b \frac{1}{2} - k \approx \frac{(b-k)\ln(b-k) - (b-k) + \frac{\ln\left[\frac{1}{\pi^3}+(b-k)(1+4(b-k)(1+2(b-k)))\right]}{6} + \frac{\ln\pi}{2}}{\ln b}$$

$$\frac{b\ln b - b + \frac{\ln\left[\frac{1}{\pi^3}+b(1+4b(1+2b))\right]}{6} + \frac{\ln\pi}{2}}{\ln b} - \log_b \frac{1}{2} - k \approx \frac{(b-k)\ln(b-k) - (b-k)}{\ln b} + \\ + \frac{\frac{\ln\left[\frac{1}{\pi^3}+(b-k)(1+4(b-k)(1+2(b-k)))\right]}{6} + \frac{\ln\pi}{2}}{\ln b}$$

$$\frac{b\log b - \cancel{b} + \frac{\ln\left[\frac{1}{\pi^3}+b(1+4b(1+2b))\right]}{6} + \cancel{\frac{ln\pi}{2}} - \ln b \log_b \frac{1}{2} - \ln bk}{\cancel{\ln b}} \approx \frac{(b-k)\ln(b-k) - \cancel{b} + k}{\cancel{\ln b}} + \\ + \frac{\frac{\ln\left[\frac{1}{\pi^3}+(b-k)(1+4(b-k)(1+2(b-k)))\right]}{6} + \cancel{\frac{\ln\pi}{2}}}{\cancel{\ln b}}$$

$$\texttt{let } C = b\ln b + \frac{\ln\left[\frac{1}{\pi^3}+b(1+4b(1+2b))\right]}{6} - \ln b\log_b\frac{1}{2} \approx (b-k)\ln(b-k) + k + \ln bk \\ + \frac{\ln\left[\frac{1}{\pi^3}+(b-k)(1+4(b-k)(1+2(b-k)))\right]}{6}$$

$$\texttt{let } f(k) = (b-k)\ln(b-k) + k + \ln bk + \frac{\ln\left[\frac{1}{\pi^3}+(b-k)(1+4(b-k)(1+2(b-k)))\right]}{6} - C = 0$$

$\therefore f(k)$ `can be used to evaluate how many keys it would take so that 2 hashes have a 50% chance of being equal.` $f(k)$ `can be evaluated using the secant algorithm`

After running `test/secant.py`, given the parameters:
Based on Wikipedia Article: Birthday Attack, we can approximate x_0 and x_1
$x_0 = \frac{1}{2} + \sqrt{\frac{1}{4} + 2 \times ln(2) \times b}$ (due to Approximation of number of people)
$x_1 = \sqrt{b}$ (due to square approximation)

error tolarance: $e = 1 \times 10^{-200}$
for $b = 4294967291^8$, we get $k_1 = 400651867432320527534628274526034254879$ for the root.
And for $b = 2^{256}$, we get $k_2 = 400651869298001176472314306405665023048$ for the root
So then $\Delta k = k_2 - k_1 = 1865680648937686031879630768169 \approx 2^{101}$ Since the difference between k_1 and k_2 is negligible (2^{101} isn't big considering the magnitude of b). We can conclude that using a galois field doesn't increase risk of birthday attacks which justifies the use of Galois fields to increase avalanche effect.

2.1.2 Collision Attack

Most denial of service attacks related to encryption algorithms are based on brute-force methods. To see if this algorithm has a potential collision attack:
$\texttt{transform}(key_1) = \texttt{transform}(key_2)$

For example: $x + y = 16$

$x, y \in \mathbb{Z}, 0 \leq x, y < 256$
there are 17-combinations for x to satisfy this equation, and simultaniously, there are 17 combinations for y to satisfy the equation, so a total of 17 combinations.
But for $x + y = z$, there are 257 combinations to try. if the result of an arithmetic operation

is known, there may be ways to get the same end-result with less combinations to brute-force. Knowing the value of z reduced the number of combinations by 15 times.
This means that the calculation done on 2.1.1 for the birthday problem would be irrelevant because there is a better algorithm than random brute forcing (to find collisions for `transform(key)`)

$\therefore$ if there is an operation that can provide the same output for a wide range of inputs, there can be a collision attack. Collision attacks can be used to derive the same transformed key using a different input key and decrypt the plaintext without actually having the original key.
In the context of this encryption algorithm (using addition as an example): keysize is 32-bytes so for byte n: $x_n + y_n = z_n$
$x, y, z \in \mathbb{Z}, 0 \leq x, y, z < 256$

Number of combinations can be represented by

$$\prod_{n=0}^{32-1} (z_n + 1)$$

so the number of combinations would be between a minimum of 32 combinations ($z_n = 0$ for all 32-bytes) up to a maximum of 2^{256} combinations ($z_n = 255$ for all 32-bytes) which can be brute forced for small z_n. So a simple addition is prone to collision attacks for $x_n + y_n = z_n$, where x, y are unknown. The use of galois field makes z_n even smaller. So even less combinations. Solution is to use operations that cannot be represented differently. e.g.

$$\sum_{i=0}^{z_n} x_n + y_n = z_n \Longrightarrow \sum_{i=0}^{z_n} (z_n - i) + (i) = z_n$$

solves for all possible x, y values for each z_n. An addition operation can be represented differently to solve for 2 unknowns, while a good mix of ARX operations cannot be reverse engineered. This is also the reason why pre-manipulating the key (using addition) before `transform()` isn't a good option. Since it provides a very obvious collision attack which makes it invalid even though pre-manipulation will provide a good avalanche effect for every single byte of the key (if 1-bit of any byte is changed, ciphertext changes completely).

So the final solution is to calculate sum of each 32-bit segment of the key (represented by k_i) in order to interlink them to make sure that every byte of the key offers the same avalanche effect:

$$\sum_{i=0}^{7} k_i$$

To test if this offers enough collision resistance: think of this problem as an example:
1. $x + y = 16$ `offers n=17 combinations`
2. $x + y + z = 16$ `offers n=153 combinations (determined experimentally)`
3. $x + y + z + v = 16$ `offers n=969 combinations (determined experimentally)`
$x, y, z, v \in \mathbb{Z}, 0 \leq x, y, z, v < 256$

So there has to be an equation or algorithm to summarize the relationship between number of variables (l) and the sum of the addition operation (16);

Knowing that equation **1.** is the simplest equation and it offers 17 combinations. Then if the rest of the equations are represented in 2-variable fashion. we can find number of combinations n:

For equation **2.**: There are 3 ways to represent as 2-variable equation

$$x + y \qquad x + z \qquad y + z$$

For equation **3.**: There are 6 ways to represent as 2-variable equation

$$x + y \qquad x + z \qquad x + v$$

$$y + z \qquad y + v \qquad z + v$$

The number of ways a multi-variable equation can be represented as a 2 variable equation can be summarized by the following:

$$\sum_{i=1}^{l-1} i$$

Using some number crunching and logic, I found that there is a direct corrolation between the number of combinations and the ratio between the current number of ways to represent as 2-variable equation over the previous number of ways to represent as a 2-variable equation:

$$n_l \propto \frac{\sum\limits_{i=1}^{l-1}}{\sum\limits_{i=1}^{l-2}}$$

Using more number crunching: I found the following recursive formula that finds the number of combinations that satisfies x+y+... = 16:

$$n_l = (n_{l-1} \frac{\sum\limits_{i=1}^{x}}{\sum\limits_{i=1}^{x-1}} + 17) \times 3 - 17 \times 3((x+1) \mod 2)$$

where $x = l - 1$ and x should be incremented until correct answer is reached for $l \geq 6$ and n_{l-1} is previous number of combinations. This formula doesn't translate to cases where the 2-variable equation doesn't have 17 combinations.

Simply put this equation couldn't be used accurately, it can only be an approximation. But upon further number crunching, I derived the following equation that satisfies all cases:

$$x \prod_{i=1}^{l-1} \frac{(x+i)}{1+i}$$

where x is the number of combinations for 2 variable equations. e.g. for $a+b = 16, x = 16+1 = 17$.

Using this equation for the context of this encryption algorithm:
Recall:

$$c = \sum_{i=1}^{l-1} i$$

where $l = 8$ since 8 32-bit segments to the 256-bit key then, $x = c + 1$

the total number of combinations according to the equation is between 1 and 2871827628774669857283799072180574717903946432793745331030345747716374528 $\approx 2.9 \times 10^{72}$

Which isn't possible to brute force provided key is random and not chosen to be a small value.

2.2 Reverse Engineering the Transformation

2.3 Collision-Resistance

Collision resistance has been proven in the birthday problem section and the collision attack section. The collision resistance is on average about as low as the numbers in sha256. and don't have any particular patterns as proven in `test/significant_perm_byte.cpp`. This test file shows the collision rate as any byte of the 32-byte key is modified by 1-bit.

$\texttt{collision\_rate} = \frac{\texttt{\#collisions}}{\texttt{32}}$

where `#collisions` is the number of bytes where $\texttt{previous key}_i$ and $\texttt{modified key}_i$ are equal

2.3.1 Different Permutation Matrices

tried permutation matrices that followed logic or randomness. But they didn't offer the proper diffusion and collision resistance required to make a secure algrorithm. The permutation matrix needs to be perfect so that the chances of collision (tested in `test/significant_perm_byte.cpp`) for every byte of the key should be around the same.

2.3.2 Number of Rounds

Number of rounds is determined by the number of rows in the permutation matrix. as using a permutation twice will cancel out.

2.3.3 Constant - F_p - Prime Galois Field Size

every addition operation is done in a prime galois field as every $k_i \in \mathbb{F}_{\shortmid}$ where k_i is the i'th index of the 32-bit segmented key $p = \texttt{0xfffffffb}$. p is the largest unsigned 32-bit prime number. It was evaluated using Fermat's Little Theorem:

Algorithm 1 Find the largest 32-bit prime number pseudo-code

Output: Biggest unsigned 32-bit prime number (p)
Begin
$p = 0$
for $i =$ `0xffffffff` to 0 **do**
    if $2^{i-1} \mod i = 1$ **then**
        $p = i$
        break
    end if
end for
Return p

2.3.4 Constant - r - Dynamic Rotation Constant

3 Hashing

The 'hashing' algorithm is used for manipulating the key each time with proper confusion and avalanche effect. The main reason why the key can be reused is due to the algorithm's strong avalanche effect as well as the lack of statistical patterns with collision rates as proven in the unit tests, `test/significant_perm_byte.cpp` which shows if some bytes have more or less collision reasistance.

3.1 Dynamic Matrix Permutation

Considering the use of `reinterpret_cast` for representing the bytearray (key) as 32-bit segments (for confusion operations), the endiannes of the computer will affect the matrix. the matrix given in this documentation is the big endian version as the algorithm is designed to be big-endian. The little endian version requires a new permutation matrix where the byte order every 32-bits is reversed. This is in the C++ implementation of `Full-Time-Pad` and the python Deravation program mentioned below.

3.1.1 Deravation

Python code is in the test/perm.py

Algorithm 2 Dynamic Permutation Matrix Deravation Pseudo-code

Input: an array of incrementing numbers (0-31) A
Output: Most Efficient Permutation Matrix V (16×32)
Begin
$P \leftarrow$ `copy of A`
for $k = 0$ to 4 **do**
  for $i = 0$ to 8 **do**
    $P_i \leftarrow A_{i\times 4}$
    $P_{i+8} \leftarrow A_{i\times 4+1}$
    $P_{i+16} \leftarrow A_{i\times 4+2}$
    $P_{i+24} \leftarrow A_{i\times 4+3}$
  end for
  $A \leftarrow$ `copy of P`
  $V.append(P)$
  $C \leftarrow$ `copy of P`
  for $m = 0$ to 3 **do**
    for $i = 0$ to 8 **do**
      for $n = 0$ to 4 **do**
        $P_{i\times 4+n} \leftarrow C_{(1+n+m) \bmod 4 \,+\, i\times 4}$
      end for
    end for
    $V.append(P)$
  end for
  $A \leftarrow$ `copy of P`
end for
Return V

3.1.2 Dynamic Permutation Matrix Values

$$
\left\{
\begin{matrix}
0 & 4 & 8 & 12 & 16 & 20 & 24 & 28 & 1 & 5 & 9 & 13 & 17 & 21 & 25 & 29 & 2 & 6 & 10 & 14 & 18 & 22 & 26 & 30 & 3 & 7 & 11 & 15 & 19 & 23 & 27 & 31 \\
4 & 8 & 12 & 0 & 20 & 24 & 28 & 16 & 5 & 9 & 13 & 1 & 21 & 25 & 29 & 17 & 6 & 10 & 14 & 2 & 22 & 26 & 30 & 18 & 7 & 11 & 15 & 3 & 23 & 27 & 31 & 19 \\
8 & 12 & 0 & 4 & 24 & 28 & 16 & 20 & 9 & 13 & 1 & 5 & 25 & 29 & 17 & 21 & 10 & 14 & 2 & 6 & 26 & 30 & 18 & 22 & 11 & 15 & 3 & 7 & 27 & 31 & 19 & 23 \\
12 & 0 & 4 & 8 & 28 & 16 & 20 & 24 & 13 & 1 & 5 & 9 & 29 & 17 & 21 & 25 & 14 & 2 & 6 & 10 & 30 & 18 & 22 & 26 & 15 & 3 & 7 & 11 & 31 & 19 & 23 & 27 \\
12 & 28 & 13 & 29 & 14 & 30 & 15 & 31 & 0 & 16 & 1 & 17 & 2 & 18 & 3 & 19 & 4 & 20 & 5 & 21 & 6 & 22 & 7 & 23 & 8 & 24 & 9 & 25 & 10 & 26 & 11 & 27 \\
28 & 13 & 29 & 12 & 30 & 15 & 31 & 14 & 16 & 1 & 17 & 0 & 18 & 3 & 19 & 2 & 20 & 5 & 21 & 4 & 22 & 7 & 23 & 6 & 24 & 9 & 25 & 8 & 26 & 11 & 27 & 10 \\
13 & 29 & 12 & 28 & 15 & 31 & 14 & 30 & 1 & 17 & 0 & 16 & 3 & 19 & 2 & 18 & 5 & 21 & 4 & 20 & 7 & 23 & 6 & 22 & 9 & 25 & 8 & 24 & 11 & 27 & 10 & 26 \\
29 & 12 & 28 & 13 & 31 & 14 & 30 & 15 & 17 & 0 & 16 & 1 & 19 & 2 & 18 & 3 & 21 & 4 & 20 & 5 & 23 & 6 & 22 & 7 & 25 & 8 & 24 & 9 & 27 & 10 & 26 & 11 \\
29 & 31 & 17 & 19 & 21 & 23 & 25 & 27 & 12 & 14 & 0 & 2 & 4 & 6 & 8 & 10 & 28 & 30 & 16 & 18 & 20 & 22 & 24 & 26 & 13 & 15 & 1 & 3 & 5 & 7 & 9 & 11 \\
31 & 17 & 19 & 29 & 23 & 25 & 27 & 21 & 14 & 0 & 2 & 12 & 6 & 8 & 10 & 4 & 30 & 16 & 18 & 28 & 22 & 24 & 26 & 20 & 15 & 1 & 3 & 13 & 7 & 9 & 11 & 5 \\
17 & 19 & 29 & 31 & 25 & 27 & 21 & 23 & 0 & 2 & 12 & 14 & 8 & 10 & 4 & 6 & 16 & 18 & 28 & 30 & 24 & 26 & 20 & 22 & 1 & 3 & 13 & 15 & 9 & 11 & 5 & 7 \\
19 & 29 & 31 & 17 & 27 & 21 & 23 & 25 & 2 & 12 & 14 & 0 & 10 & 4 & 6 & 8 & 18 & 28 & 30 & 16 & 26 & 20 & 22 & 24 & 3 & 13 & 15 & 1 & 11 & 5 & 7 & 9 \\
19 & 27 & 2 & 10 & 18 & 26 & 3 & 11 & 29 & 21 & 12 & 4 & 28 & 20 & 13 & 5 & 31 & 23 & 14 & 6 & 30 & 22 & 15 & 7 & 17 & 25 & 0 & 8 & 16 & 24 & 1 & 9 \\
27 & 2 & 10 & 19 & 26 & 3 & 11 & 18 & 21 & 12 & 4 & 29 & 20 & 13 & 5 & 28 & 23 & 14 & 6 & 31 & 22 & 15 & 7 & 30 & 25 & 0 & 8 & 17 & 24 & 1 & 9 & 16 \\
2 & 10 & 19 & 27 & 3 & 11 & 18 & 26 & 12 & 4 & 29 & 21 & 13 & 5 & 28 & 20 & 14 & 6 & 31 & 23 & 15 & 7 & 30 & 22 & 0 & 8 & 17 & 25 & 1 & 9 & 16 & 24 \\
10 & 19 & 27 & 2 & 11 & 18 & 26 & 3 & 4 & 29 & 21 & 12 & 5 & 28 & 20 & 13 & 6 & 31 & 23 & 14 & 7 & 30 & 22 & 15 & 8 & 17 & 25 & 0 & 9 & 16 & 24 & 1
\end{matrix}
\right\}
$$

3.1.3 Other Options

3.2 Confusion - ARX

For the confusion aspect of the key transformation algorithm, ARX operations are used.

3.2.1 A - Modular Addition

3.2.2 R - Bitwise Rotation

3.2.3 X - XOR

4 Cipher

4.1 Transformation

4.2 dynamic_permutation

4.3 Encryption Index

4.4 Long Plaintexts

Algorithm 3 The key transformation operation per encryption

Input: 32-byte bytearray key (`key`)
Output: Transformed 32-byte bytearray key (k)
Begin
// `keysize` = 32, length of input key is 32-bytes
// Vector used for dynamic permutation, dynamically permutated key placeholder with length of keysize.
// Use stack memory
uint32_t *k ← endian_8_to_32_arr(`key`)
for $i = 0$ to 16 **do**
    int index ← $i \ll 2$
    int i1mod ← $\mathit{index} \mod 8$
    int i2mod ← $\mathit{index} + 1) \mod 8$
    int i3mod ← $\mathit{index} + 2) \mod 8$
    int i4mod ← $\mathit{index} + 3) \mod 8$
    int imod8 ← $i \mod 8$
    int imod9 ← $(i + 1) \mod 8$
    int rmod ← $i \mod 5$

    // No unwanted overflow, so convert to uint64_t
    $k_{i1mod} \leftarrow (\text{uint64\_t})k_{i1mod} + A_{imod8} + k_{i1mod} \ggg r_{rmod} \pmod p$
    uint32_t sum ← $\sum_{j=0}^{8} k_j \pmod p$
    $A_{imod9} \leftarrow A_{imod9} \oplus \text{sum}$
    $k_{i2mod} \leftarrow (\text{uint64\_t})k_{i2mod} + A_{imod9} + k_{i2mod} \lll r_{rmod} \pmod p$
    $A_{imod8} \leftarrow A_{imod8} \oplus (\text{uint64\_t})k_{i2mod} + k_{i1mod} \ggg r_{(i+1) \mod 5} \pmod p$
    $k_{i3mod} \leftarrow (\text{uint64\_t})A_{imod8} \oplus k_{i3mod} + A_{imod9} \oplus k_{i4mod} \pmod p$
    $k_{i4mod} \leftarrow (\text{uint64\_t})A_{imod8} \oplus k_{i4mod} + A_{imod9} \oplus k_{i3mod} \pmod p$

    // Permutate the bytearray key
    `key` ← dynamic_permutation(`key`, i);
end for
Return `key`

Algorithm 4 The dynamic_permutation() in the transformation function

Input: 32-byte bytearray (B), permutation matrix (V), iteration index (i)
Output: 32-byte permutated bytearray (B)
let P be placeholder vector declared with length of 32-bytes
for $j = 0$ to 32 **do**
    $P_j = B_{V_{i_j}}$
end for
Return `P`