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 form using a constant permutation matrix. Due to the optimizations in modern devices (Vectorization, Cache & Parallelization), this is really fast. To acheive the confusion, the key is manipulated in 32-bit segments 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 bytes of the key. This high-speed (around 2x the speed of ChaCha20) algorithm is a revolution in cryptography as it makes use of modern hardware accelarations for permutations unlike older algorithms which offers zero non-uniform shuffling.

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, $x, y \in \mathbb{F}_{\text{|}}$ where $/F_p$ denotes the finite field with a prime field size. There are finite number of elements in F_p which means that $x, y < p$.

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

Implicitly denotes
$x + y = z \pmod 7$

So all addition operations require modulo p.
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

Permutating a vector refers to transposing the elements in a vector. It is a reversible operation. For example, if I have the vector: V

$$V = \begin{pmatrix} 0 & 1 & 2 & 3 \end{pmatrix}$$

and have the permutation vector P:

$$P = \begin{pmatrix} 3 & 2 & 1 & 0 \end{pmatrix}$$

then the resulting matrix would be

$$V' = \begin{pmatrix} 3 & 2 & 1 & 0 \end{pmatrix}$$

If we look at V_0 (**0**), according to P_0, it should be replaced with the 3rd value in P. so then $V'_0 = 3$. This doesn't mean that $V'_3 = 0$. This algorithm is demonstrated in dynamic_permutation section (5.5). . The permutation in this algorithm is used for the bytearray `key` and the 32-bit segmented `k`. In order to confuse (ARX) and diffuse (permutate), `k` and `key` needs to point to the same memory address. Most efficient way to do this, would be using `reinterpret_cast` in C++ or using `unions` in C. Since these are

low level approaches, endiannes of the system will affect the byte order. In a little-endian system, the data requires the following permutation before the transformation iterations:

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

This means that for little endian systems, the permutation matrix also needs to change. The little-endian permutation matrix deravation is given in the `test/perm.py`.

2 Security

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 NIST SP 800-22 - Statistical Test Suite - Official NIST Randomness Test

using the python tool from https://github.com/stevenang/randomness_testsuite, all 3 versions of the transformation algorithm have been testing using a fixed key (0-31) and

plaintext (0-31). Their values had a very obvious pattern, every time, the encryption index was incremented once per encryption. All 3 versions of the transformation function resulted in full-randomness for all 16-tests. This means that the slightest change in input provides a very good avalanche effect, and therefore has a proper avalanche effect.

2.2 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.

$$a = 115792089237316195423570985008687907853269984665640564039457584007913129639936_{10}$$
$$b = 115792088158918333131516597762172392628570465465856793992332884130307292657121_{10}$$
$$\texttt{let } \Delta = a - b$$
$$\Delta = 107839786229205438724651551522469951919978377004712469987760583698281 5_{10}$$

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`

2.2.1 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 $k_1 = 40065186743232052753462827452603425487 9$ for the root.
And for $b = 2^{256}$, we get $k_2 = 40065186929800117647231430640566502304 8$ for the root
So then $\Delta k = k_2 - k_1 = 18656806489376860318796307681 69 \approx 2^{101}$ 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.

2.2.2 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:
`transform(`key_1`) = 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 expression, and simultaniously, there are 17 combinations for y to satisfy the expression, so there is a total of 17 combinations.
But for $x + y = z$, there are 256 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.

$\therefore$ 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 operation): keysize is 32-bytes
so for each 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:

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

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:

$$n_\ell \approx\propto \frac{\sum_{i=1}^{\ell-1}}{\sum_{i=1}^{\ell-2}} \times k \qquad \texttt{where k is a multiplier:} k \in \mathbb{N}$$

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

$$n_\ell = (n_{\ell-1} \frac{\sum_{i=1}^{x}}{\sum_{i=1}^{x-1}} + 17) \times 3 - 17 \times 3((x+1) \mod 2)$$

where $x = \ell - 1$ and x should be incremented until correct answer is reached for $\ell \geq 6$ and $n_{\ell-1}$ 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:

$$\prod_{i=0}^{\ell-1} \frac{(x+i)}{1+i}$$

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:

maximum 32-bit number:

$$c = 2^{32} - 1 = \texttt{0xffffffff}$$

where $\ell = 8$ 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 $28718276287746698572837990721805747179039464327937453310303457477163745 28 \approx 2.9\times10^{72}$

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

2.3 Joux's Attack: Multicollisions in Iterated Hash Functions

This attack requires knowing a collision: `transform(`k_1`) = tranform(`k_2`)`. Given a secure hashing algorithm, this isn't possible. Therefore, the Joux's attack isn't possible. The only way to evaulate k_1 and k_2 would be using brute-force. That is mathematically proven in sections 2.2.1 and 2.2.2. This is also proven experimentally in `test/collision.cpp`. The values in `test/collision_report` show that over lots of iterations, getting the same hash is still impossible.

2.4 Related Key Attack

Related key attacks isn't possible considering that it requires that `transform(`k_1`)` is similiar to `transform(`k_2`)` where k_1 and k_2 have a predictible difference (incremented by encryption index). But this is only under the assumption that the `transform` function is an irreversible hashing algorithm.

2.5 Time Based Attacks

Start from 32 zeros for a key, then incremented each byte by 1 until there are 32 255s and test again. This is tested in `test/benchmark` in function `benchmark_hash_time_attack_v()`. This test passed. The keys with small numbers have the same speed as keys with large numbers (This was never in doubt), therefore algorithm is proven to be constant time.

```
----------TESTING SIDE CHANNEL ATTACKS----------
TESTING TRANSFORMATION VERSION 1.0:  PASSED (NO SIDE CHANNEL): data is consistent
TESTING TRANSFORMATION VERSION 1.1:  PASSED (NO SIDE CHANNEL): data is consistent
TESTING TRANSFORMATION VERSION 2.0:  PASSED (NO SIDE CHANNEL): data is consistent
```

Tested for time-based side-channel attacks by comparing the computation time for different key values, If computation time for smaller key (32 0s) approximately equals to computation time for (32 255s) key, then data is consistent. This test is to see if there was a side-channel vulnerability (hardware/software) where someone approximates the value of the key by timing the calculation speed.

2.6 Linear Cryptoanalysis

Linear Cryptoanalysis is used to find linear connections between ciphertext and plaintexts.
Given that running `test/significant_perm_byte -r` provides that all bytes of the key provide approximately the same collision resistance and shows that there is no pattern. This means that modifying a certain byte isn't only affected some bytes but all of them without a pattern, therefore there are no significant bytes. And all bytes of the key affect affect the result equally. This is mainly due to the use of sum of k_i in the transformation algorithm.

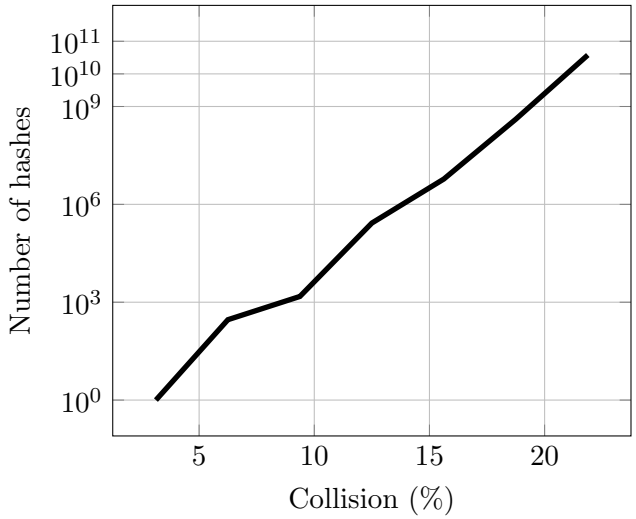
Here is data for SHA-256 hashing algortihm, what percentage of bytes can be equal, if this number gets to 100%, this means that there is a successful collision. i is the number of hashes that are generated to get n% equality.

This data fluctuates a lot and therefore doesn't actually have any visible pattern, The numbers being similiar to SHA-256 would be good news to suggest that the avalanche effect is real and that the math checks out for the birthday problem, it's not possible to find a collision.

%	**Full-Time-Pad -(i)**	**SHA-256 (i)**
3.125	1	2
6.25	291	6
9.375	1501	2648
12.5	267184	36595
15.625	5981439	6167198
18.75	408355721	530210645

Table 1: collision (%) VS. number of hashes

Collision (%) VS. Number of Hashes for Full-Time-Pad



2.7 Differential Cryptoanalysis

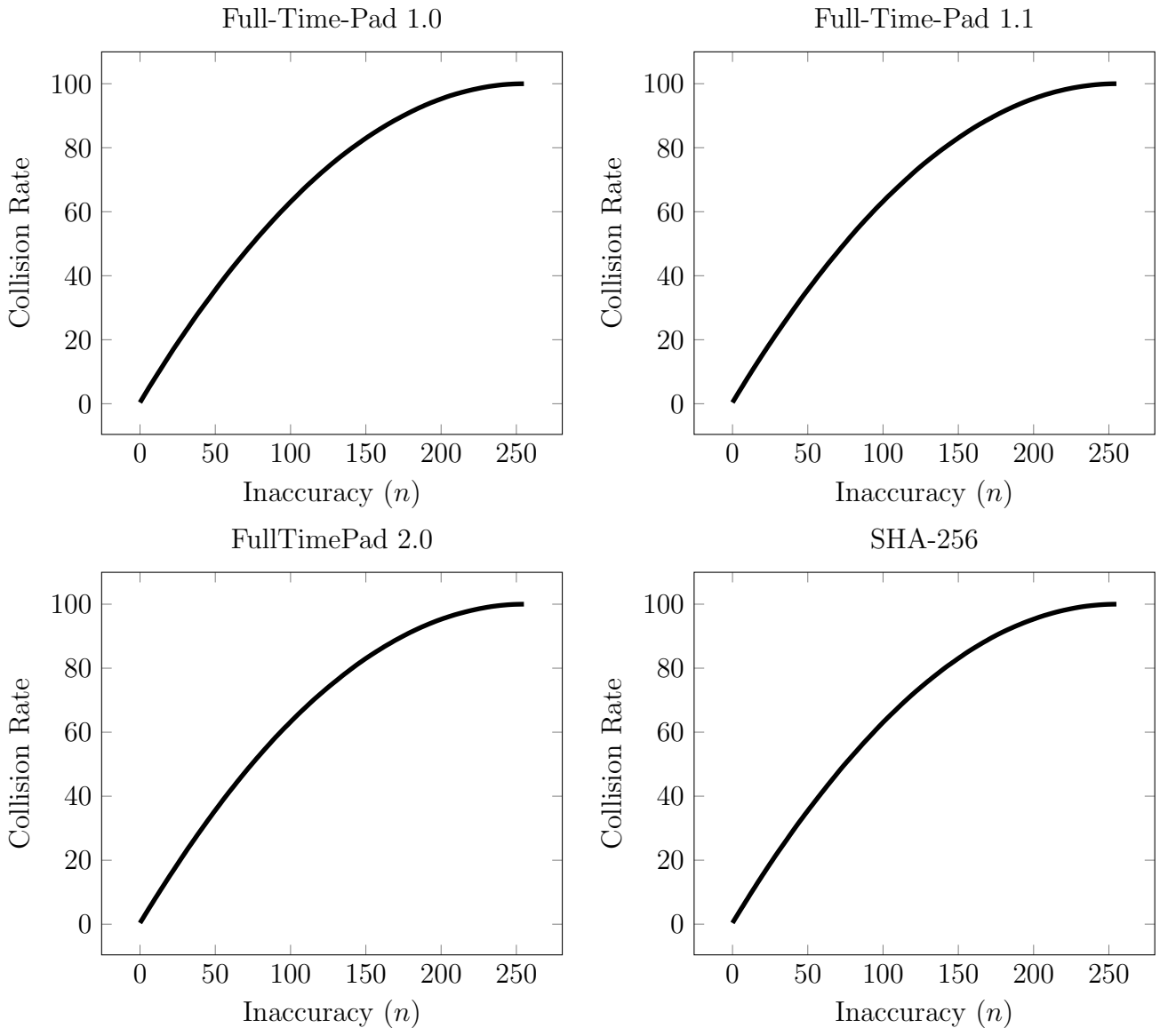
In `test/significant_perm_byte.cpp`, instead of comparing if values are equal, but rather approximately equal with accuracy of $\pm n$. This gave the same results are sha-256. And no specific bytes seemed to have followed a specific pattern. This means that there isn't a differential statistical pattern.

Results of collision rates are approximately equal given lower accuracy ($\pm n$). This suggests that there isn't any cryptoanalysis pattern. These numbers are average of 255 tries. Each try also yields similar results which means that there isn't a differential cryptoanalysis exploit regarding accuracy of comparision.

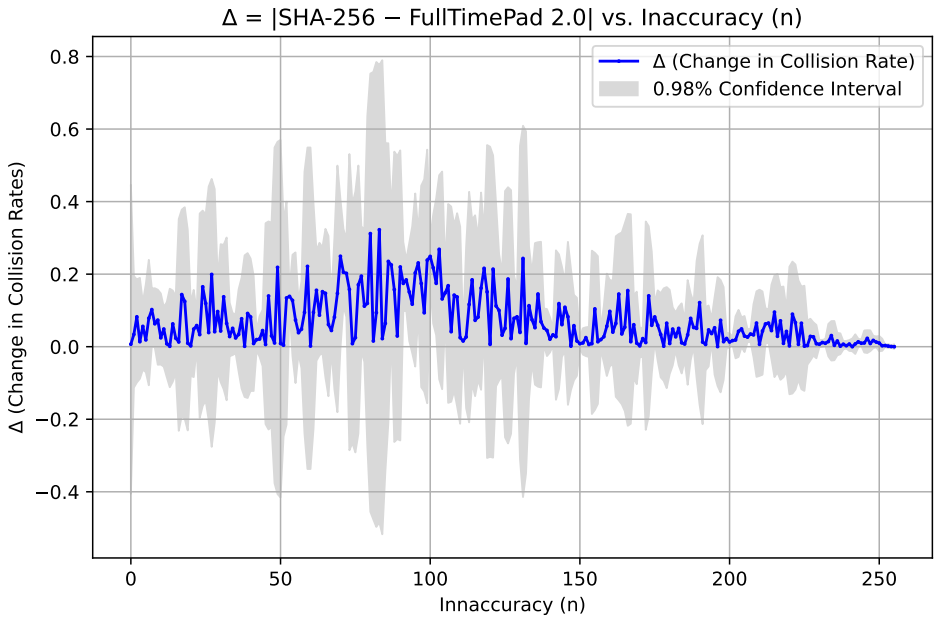
n	**Full-Time-Pad (%)**	**SHA-256 (%)**
0	0.3883	0.3923
5	4.2678	4.224
10	8.0342	8.0322
15	11.7421	11.7403
20	15.4626	15.4287
25	18.8071	18.8382

Table 2: Comparision Accuracy VS. Collision Rate

Figure 1: Collision Rate (%) vs. Inaccuracy (n)



These graphs demonstrates how lower accuracy of comparison results in higher collision rate (For both SHA-256 and Full-Time-Pad). The points for this graph is evaluated by running `test/significant_perm_byte -d -version_number`. The consistent non-volatile increase in the numbers are a good thing.



This graph shows the difference (Δ) between SHA-256 hashing algorithm and the `FullTimePad 2.0` which logically would be the most at risk version of `Full-Time-Pad` for security exploits. Since numbers on the y-axis are very small, this means that Δ is negligible and therefore there isn't a pattern.
The reason SHA-256 was chosen for this comparision is because this algorithm has proven itself to be secure.

2.8 Reverse Engineering the Transformation

this is demonstrated in the `test/reverse.cpp` code. The reversed key and key are very different.

2.9 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.

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

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

2.9.1 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.
Using Bent functions to generate a non-uniform permutation matrix is also possible.

In terms of efficiency, the permutation matrix should be a uniform permutation matrix. The current matrix is non-uniform, which might provide better diffusion (more random), it also creates the issue of efficiency. So in terms of computation speed, reducing the number of times the memory is accessed is required. To achieve this, I will implement an updated version with uniform matrix and perform benchmark tests, based on results of both avalanche effect and cryptoanalysis testing, I will keep/undo the tests.

2.9.2 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.

2.9.3 Constant - F_p - Prime Galois Field Size

every addition operation is done in a prime galois field as every $k_i \in \mathbb{F}_{\text{|}}$ 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

1: **Output:** Biggest unsigned 32-bit prime number (p)
2: **Begin**
3: $p = 0$
4: **for** $i = \texttt{0xffffffff}$ to 0 **do**
5: **if** $2^{i-1} \mod i = 1$ **then**
6: $p = i$
7: break
8: **end if**
9: **end for**
10: **Return** p

2.9.4 Constant - r - Dynamic Rotation Constant

Cicadas only surface once every 17 years. Assuming that their predetor only surfaces once every 2 years, they will only meet each other once every 34 years. Since 17 is a prime number, they will barely ever meet. However, if they surfaced once every 18 years, while their predetor once every 2 years. Then every 18 years they would meet. This would also make it easier for the predetor to synchronize their emergence timing.
let x be the time interval for when the cicadas surface
let y be the time interval for when the predetors surface
We can conclude:
$GCD(x, y) = 1$, then they meet up every $x \times y$ years.
Whereas if $GCD(x, y) \neq 1$, then they meet up every x years (assuming $x > y$).

Bitwise rotation operation should also be done with either a prime number or a number where $GCD(x, y) = 1$ to maximize the collision resistance.

If not, then for example:
let w be 32-bit number
let $q = 8$ be rotation count
rotation can be represented by: $w = w \lll q$
If this rotation was repeated for 4 iterations, then the rotations would cancel out and rotated w (w') and w would be equal. Make sure that $GCD(q, 32) = 1$.

If there's only one rotation number, the likelihood of finding a pattern increases. Imagine a card dealer shuffling cards with two stacks, ensuring that one card from each stack is placed on top of the other in a consistent pattern. In this case, you could easily "unshuffle" the cards by simply separating them into two stacks, putting one card on one side and one card on the other. Using a single rotation like this is too simple, but if the cards were shuffled using a more complex method, the process becomes harder to reverse. Even though each method is reversible on its own, combining them makes it significantly more difficult to undo. The use of bitwise rotations on 32-bit segments combined with byte-level permutations creates excellent diffusion. The only potential improvement would be to ensure that all bytes are interlinked solely through the bitwise rotations and permutations, rather than relying on the sum, as seen in the current transformation algorithm.

Rotation numbers that are selected are represented by r:

$$r = \begin{pmatrix} 23 & 5 & 17 & 31 & 13 \end{pmatrix}$$

There are 5 numbers for r, this is because $16 \mod 5 \neq 0$, after 16 iterations in the transformation algorithm, all values of k aren't rotated the same number of times.

If you were to rotate 32-bit unsigned integer (x)

$$\sum_{i=0}^{4} r_i = 89$$

number of times, the net rotation would be 89 (a prime number). $x \ggg 89$ would produce a net rotation of $x \ggg 25$, since $89 \mod 32 = 25$. Knowing that $GCD(25, 32) = 1$, the minimum number of times where the expression $x \ggg 89$ is repeated till rotated x equals x is $89 \times 32 = 2848$. This means that only after 2848 iterations would rotated x equal x. Whereas if there was a singular number for r, the number of rotations for rotated x to equal x would be $r \times 32$ (e.g. $r = 23, 23 \times 32 = 736$).

$\therefore$ there needs to be multiple numbers for r.
It is designed so that $|r_i - r_{i-1}|$ results in a different magnitude each time so that the net rotation is a bigger number (more rotation means better diffusion/confusion).
For the deravation, the numbers started with prime numbers, all prime numbers from 0-32, but most were eliminated due to similiar values for $|r_i - r_{i-1}|$ results (e.g. $23 - 5 = 18$, $|5 - 17| = 12$ aren't too close). A good net rotation would be prime and without similiarities to net rotations.
In the transformation algorithm, There are bitwise right rotate ($\ggg$) and bitwise left rotate ($\lll$), this is done to properly mix the results.

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.
To maximize the effects of confusion using the current logic, 4 out of 8 values k (32-bit segmented key) should be modified per iteration (half of key). So each byte of the key is transformed a total of 8 times ($\frac{\texttt{16iterations}}{8 \div 4}$). since $16 \mod 4 = 0$, all bytes are transformed the same number of times which would also tarnish the possibility of a statistical exploit that could target less transformed bytes of the key.
Half of the key should be modified per iteration because classical cryptographic algorithms (non-quantum) rely on computational complexity for their security. So more transformation the better. But the most minimum work to get a secure transformed key is enough,
therefore it's the goal. this was determined experimentally; testing collision resistance of each byte upon modifying the transformation algorithm.

I would represent the transformation algorithm with a game of cards. Imagine that 8 cards are randomly picked in a pile of 2^{256} cards, each card is cut up into 4 equal pieces, the cut up pieces are shuffled using a permutation algorithm while also replaced with other cut up values from the pile of cards and simultaniously, some pieces are cut up to 2 pieces with magnitude of a prime number (e.g. 5mm, other pieces being 32-5mm if card is 32mm). If this process repeated for 16 iterations, could you recover the original cards? Probably not.

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.
This generates a non-uniform permutation matrix. Which is a "key" concept in uniform randomization of the transposition of bytes.

3.1.1 Deravation

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**
                $P_{i \times 4+n} \leftarrow C_{(1+n+m) \mod 4 + i \times 4}$
            end for
        end for
        $V.append(P)$
    end for
    $A \leftarrow$ `copy of P`
end for
Return V

Python code is in the test/perm.py

3.1.2 Dynamic Permutation Matrix Values

$$\left\{\begin{array}{cccccccccccccccccccccccccccccccc}
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{array}\right\}$$

3.2 Confusion - ARX

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

3.2.1 A - Modular Addition

For the modular addition, a constant array of 8 values is used (`A`):

$$\texttt{encryption index} \gg 32$$
$$\texttt{encryption index \& 0xffffffff}$$
$$\texttt{0x119f904f}$$
$$\texttt{0x73d44db5}$$
$$\texttt{0x3918fa83}$$
$$\texttt{0x5546b403}$$
$$\texttt{0x216c46df}$$
$$\texttt{0x64997dfd}$$

`A` is derived as follows:
Start with the message: "**cipher**" (6 digit message with 6 uneiqe characters)
Take every ascii encoding of each character of the message and calculate to the 32nd power and take the first 32-bits:
Example:
"c": 99 in ascii encoding
In python: hex(99^{32})[:10] = `0x0x119f904f`
The word **cipher** and $\mathbf{A}_i$ is arbitrary. In other words, it's "nothing up my sleeve numbers". This is an initial array. In the transformation algorithm, every $\mathbf{A}_i$ is modified twice (in 16 iterations). This provides the confusion. The reason why the first 2 values are the 64-bit encryption index is because this was the most efficient way to diffuse in the encryption index.

3.2.2 R - Bitwise Rotation

"Using the values determined for r (as discussed in Section 2.9.4), rotate one value to the right, another to the left, and then the third to the right again. The reasoning behind alternating the rotation directions goes back to the card analogy. Imagine two stacks of cards. One on the left and the other on the right. To shuffle them efficiently, a logical approach would be to alternately take a card from the left stack and place a card from the right stack on top. For a "randomized" shuffle, the best strategy would be to take nth card from alternating sides, where n is a prime number, to minimize collisions."

3.2.3 X - XOR

Some of the values are xored because xor operator offers 256 combinations per byte unlike unsigned addition which mostly offers less (section 2.2.2). This will bring more possabilites which in return limits brute-forcing attacks. It is avoided when there is a good possibility of xoring the same 2 values ($x \oplus x$), this is why the sum of k_i is used rather then xor.

4 Performance

In terms of performance, the transformation version 1.0 is quite slow compared to ChaCha20 and AES. The main reason for the poor performance is the permutation. Version 2.0 of the transformation algorithm could use further optimizations in later versions of the `Full-Time-Pad` cipher which is the fastest yet secure version of the algorithm. Not all versions value speed,

some are purely focused on security. Sometimes when dealing with things like passwords, I would recommend using slower algorithms to reduce brute-forcing possibilities.

Benchmark tests show the following results:

```
----------TESTING SPEED----------
TESTING TRANSFORMATION VERSION 1.0:  Computation Time for 1000000 key transformations:
0.641983 seconds
Hashrate:  1557674 keys per second

TESTING TRANSFORMATION VERSION 1.1:  Computation Time for 1000000 key transformations:
0.526699 seconds
Hashrate:  1898618 keys per second

TESTING TRANSFORMATION VERSION 2.0:  Computation Time for 1000000 key transformations:
0.059143 seconds Hashrate:  19178011 keys per second
```

On the same device (12th Gen Intel(R) Core(TM) i7-12700H, DDR5 RAM). The ChaCha20 benchmark tests yield the following results:

```
Time taken to encrypt 1000000 32-byte messages:  0.160442 seconds
Encryption rate:  6232781 keys per second
```

This means that `Full-Time-Pad` transformation version 2.0 is more than 3 times faster than ChaCha20. However, ChaCha20 has 64-byte keystreams while my algorithm has 32-byte. So Mine is 3x faster for encrypting unieqe plaintexts under 32 bytes (For long plaintexts, 1.5x). These tests are under the assumption that there are full optimizations possible (modern hardware).

One must also consider that my algorithm could be redesigned to generate 64-byte blocks faster than ChaCha20 as longer keystream doesn't mean a linear increase in time. This is a potential upgrade in future version of this algorithm.

Also, for Versions 1.0 & 1.1, The poor performance is mainly due to the poor implementation. Implementation could be improved, yet I don't think speed would be affected too significantly.

5 Cipher

5.1 Transformation - Version 1.0 - Prioritize Complexity

Algorithm 3 The key transformation operation per encryption

Input: 32-byte bytearray key (`key`), `64-bit encryption_index`
Output: Transformed 32-byte bytearray key (k)
Begin
// `keysize` = 32, length of input key is 32-bytes
uint32_t *k ← endian_8_to_32_arr(`key`)
for $i = 0$ to 16 **do**
    index ← $i \ll 2$
    i1mod ← index $\mod 8$
    i2mod ← index + 1) $\mod 8$
    i3mod ← index + 2) $\mod 8$
    i4mod ← index + 3) $\mod 8$
    imod8 ← $i \mod 8$
    imod9 ← $(i + 1) \mod 8$
    rmod ← $i \mod 5$

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

    // Permutate the bytearray key
    `key` ← dynamic_permutation(`key`, i);
    increment `encryption_index` by 1 for each plaintext
end for
Return `key`

5.2 Transformation - Version 1.1 - Prioritize Speed & Complexity

This version of the transformation offers more speed less complexity. This is preferred to the previous option since the other has more unnecessary operations that add complexity but isn't really needed, the previous option is better suited for when speed doesn't matter as much as security, or when slower algorithms are appreciated.

Algorithm 4 The key transformation operation per encryption

Input: 32-byte bytearray key (`key`), `64-bit encryption_index`
Output: Transformed 32-byte bytearray key (k)
Begin
// `keysize` = 32, length of input key is 32-bytes
uint32_t * k ← endian_8_to_32_arr(`key`)
for $i = 0$ to 16 **do**
    index ← $i \ll 2$
    i1mod ← index $\mod 8$
    i2mod ← index + 1) $\mod 8$
    i3mod ← index + 2) $\mod 8$
    i4mod ← index + 3) $\mod 8$
    imod8 ← $i \mod 8$
    imod9 ← $(i + 1) \mod 8$
    rmod ← $i \mod 5$

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

    // Permutate the bytearray key
    `key` ← dynamic_permutation(`key`, i);
end for
increment `encryption_index` by 1 for each plaintext
Return `key`

5.3 Transformation - Version 2.0 - Prioritize Speed

This version of the transformation prioritizes speed. This is preferred to the previous option if complexity doesn't matter as much. This is still secure, but not as secure as Version 1.0. This version is the updated Version 1.1, the main difference is removing the galois field as this still provides a good confusion and making the algorithm have only 10 rotations (Determined experimentally).

Algorithm 5 The key transformation operation per encryption

Input: 32-byte bytearray key (`key`), `64-bit encryption_index`
Output: Transformed 32-byte bytearray key (k)
Begin
// `keysize` = 32, length of input key is 32-bytes
uint32_t *k ← endian_8_to_32_arr(`key`)
for $i = 0$ to 10 **do**
    index ← $i \ll 2$
    i1mod ← index mod 8
    i2mod ← index + 1) mod 8
    i3mod ← index + 2) mod 8
    i4mod ← index + 3) mod 8
    imod8 ← i mod 8
    imod9 ← $(i + 1)$ mod 8
    rmod ← i mod 5

    $k_{\text{i1mod}} \leftarrow k_{\text{i1mod}} + A_{\text{imod8}} + k_{\text{i1mod}} \ggg r_{\text{rmod}}$
    sum ← $\sum_{j=0}^{8} k_j$
    $A_{\text{imod9}} \leftarrow A_{\text{imod9}} \oplus \text{sum}$
    $k_{\text{i2mod}} \leftarrow k_{\text{i2mod}} + A_{\text{imod9}} + k_{\text{i2mod}} \lll r_{\text{rmod}}$
    $A_{\text{imod8}} \leftarrow A_{\text{imod8}} \oplus k_{\text{i2mod}}$
    $k_{\text{i3mod}} \leftarrow A_{\text{imod8}} \oplus k_{\text{i3mod}}$
    $k_{\text{i4mod}} \leftarrow A_{\text{imod8}} \oplus k_{\text{i4mod}}$

    // Permutate the bytearray key only 2 times, don't permute before last iteration
    if i mod 4 = 0 and $i \neq 8$ **then**
        `key` ← dynamic_permutation(`key`, i);
    end if
end for
increment `encryption_index` by 1 for each plaintext
Return `key`

5.4 dynamic_permutation

Algorithm 6 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

5.5 endian_8_to_32_arr

Algorithm 7 The endian_8_to_32_arr() in the transformation function

Input: 32-byte bytearray (B)
Output: 32-byte bytearray (B) permutated to flip endiannes if necesarry
if big endian system **then**
    for $j = 0$ to 32 **do**
        swap B_i with B_{i+3}
        swap B_{i+1} with B_{i+2}
    end for
end if
Return reinterpret_cast <uint32_t* >(B)

5.6 Key Reuse

In order to reuse the key, there is an encryption index which is a 64-bit nonce that is incremented per encryption in order to keep the transformed key unique. The encryption index is infused in the encryption algorithm in an efficient manner, the first 2 values is the encryption index. It's the first 2, not the last 2 because indexing in the transformation algorithm uses the first 2 indexes of **A** so this would provide more iterations to diffuse in the encryption index. The encryption index could also be randomly generated each time. If it's incremented it's necessary to make sure that the same encryption index is incremented when the plaintext is long (not just incremented per message but per 32-byte segment of every message sent.)

5.7 Long Plaintexts

When there are long plaintexts used, the encryption index should be incremented by 1 the same way as key reuse. It might be logical to randomly generate the encryption index per plaintext so that you don't have to keep track of incrementing the encryption index for all plaintexts but rather per plaintext. This isn't an issue in the algorithm itself but rather a side-channel exploit if you don't update encryption index. My implementation of the algorithm doesn't impose this problem as long as you don't use the same key for 2 `FullTimePad` Objects.

5.8 Encryption/Decryption operation

Encryption operation is the same as decryption

Algorithm 8 Encrypt/Decrypt transform Function

Input: plaintext (p), length (ℓ), 32-byte original key (k), encryption index/nonce (`encryption_index`)
Output: ciphertext (c)
for each 32-byte segment of plaintext **do**
    // create a copy of unmodified k, and transform it using an incrementing unieqe `encryption_index`
    // Then, encrypt using plaintext xor transformed key = ciphertext
    copy = copy(k)
    $c \leftarrow c \,||\, p \oplus$ transformation(copy, `encryption_index`)
end for

// For the remainder:
$c \leftarrow c \,||\, p \oplus$ transformation(copy, `encryption_index`)
Return c