Format-Preserving Encryption using Hash Functions in a Feistel Cipher

Cyril Dever
Edgewhere

March 27, 2021

Abstract

We define an obfuscation tool to secure data with a true Format-Preserving Encryption process. By implementing a Feistel block cipher with a round function using any robust hashing function, it provides you with a tool to obfuscate strings in a safe, fast and secure way featuring the convenience of format-preserving. It might be used to create digital confidential redacted documents.

I. Introduction

Provided you need a robust obfuscation tool for protecting your data more than an actual encryption cipher, meet our newest implementation of the well-known Feistel network algorithm.

It is secure yet very fast and comes with the very handy feature of Format-Preserving Encryption (FPE), ie. it takes a string and returns a string, and the length of the output is the same as the one of the output. The latter is said "readable" in the sense that it uses a possibly readable character set (getting rid of any weird control character). Our implementation uses a specific extract of the first 512-characters UTF-8 table.

When used within a document you wish to anonymize for instance, it preserves the general form of the document, only obfuscating the needed data. For example, the company name in "Edgewhere is a technology service company" would output as "Ki(#q|r5* is a technology service company".

But further development could even allow a full anonymization by changing each and every occurrence to a different output, making it impossible to link the pseudonyms to their underlying value. It becomes the digital version of redacting classified information[1].

[1]Please ask for my `redacted` library available in different language implementations for that purpose

II. The Algorithm

1. Formal Description

We herein define $\mathfrak{F}$ our own implementation of a Feistel block cipher[1].

We use an unbalanced implementation, cutting the input data into two parts, eventually preparing them before processing them through our round function (see section 2), to finally concatenating the end results to form the final obfuscated ciphertext.

It is a fully Format-Preserving Encryption scheme.

Let us start with what we use as the basis for our own implementation: the formal description provided by Wikipedia [2] for a Feistel block cipher is as described in Algorithm 1.

Let $N = n + 1$ be the number of rounds, $K_0, K_1, ..., K_n$ the keys associated with each round and $F : \omega \times \mathcal{K} \mapsto \omega$ a function of the $(words \times keys)$ space to the *words* space.

2. Hash Round Function

Figure 1 provides a graphical representation of our cipher $\mathfrak{F}$ in its entirety.

Our implementation takes its robustness by not actually using one different key per round, but rather a well-tested hash function[3] $\mathfrak{h}()$ and

[2]`https://en.wikipedia.org/wiki/Feistel_cipher`

[3]In our latest FPE implementation, we offer to choose between four well-known and well-tested hashing algorithms: `Blake2b`, `Keccak`, `SHA-256` or `SHA-3`, all in their 256-bits versions.

Algorithm 1: Standard Feistel cipher

Input: a message m
Output: the ciphertext c
let the encrypted word in step i be $m_i := L_i \mathbin{+\!\!+} R_i$ with $m_0 := L_0 \mathbin{+\!\!+} R_0$ as the unciphered message;
for $i \leftarrow 0$ **to** n **by** 1 **do**
    $L_{i+1} \leftarrow R_i$;
    $R_{i+1} \leftarrow L_i \oplus F(L_i, K_i)$;
$m_N := L_{n+1} \mathbin{+\!\!+} R_{n+1}$;
return m_N

a single key K in its round function F.

The round function F thus consists in taking the right side R at each round and apply to it two operations:

- Shift K by the number of round;
- Add R to the masked key K' (of the shifted K);
- Hash the result R' through $\mathfrak{h}()$.

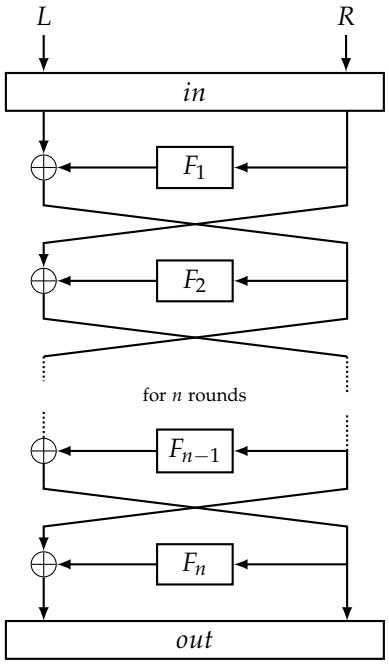

Figure 1: *Feistel block cipher* $\mathfrak{F}$

2.1 Shifting the key

To shift the passed key K by one character at each round, we use the shifting function $S()$.

Let $substr(x, start)$ be a function that keeps the substring of the passed x from $start$ to end.

$$\begin{aligned} S : \mathcal{K} \times \mathbb{N} &\to \mathcal{K} \\ (K, i) &\mapsto substr(K \ll i, 1) \mathbin{+\!\!+} K[0] \end{aligned} \tag{1}$$

That way, we build a "new" key from K for $\|K\|$ rounds, adding security to our round function.

2.2 Masking the new key

To enable the XOR part of the Feistel cipher, we have to apply a "masking" operation $\mu()$ on the input key K to make it of length $l = \|R\|$:

$$\begin{aligned} \mu : \mathcal{K} \times \mathbb{N} &\to \mathcal{K} \\ (K, l) &\mapsto K' := \begin{cases} \text{if } \|K\| \geq l, \sum_{i=1}^{l} K[i] \\ \sum_{i=1}^{l} (K \times \left\lceil \frac{\|K\|}{l} \right\rceil)[i] \end{cases} \end{aligned} \tag{2}$$

If the key K is too long, the masking function $\mu()$ eventually cuts it, ie. only keeping the l-th first bytes. And if it is too short, it multiplies it the needed number of times and cut the concatenation to the desired length l.

2.3 Adding parts

At each round, we add the masked key K' with the right part of the previous round R through the function $A()$ described in Algorithm 2.

Let *charcode* be the UTF-8 character code of the concerned byte.

Algorithm 2: Addition function A

Input: R, $K' \leftarrow \mu(K, \|R\|)$
Output: R'
initialize $R' \leftarrow \emptyset$ of length $\|R\|$;
foreach *charcode* $i \in R$ *and* $i \in K'$ **do**
    $R'[i] := R[i] + K'[i]$;
return R'

For example, the addition of `a` with `b` gives: `a` $\leftarrow 61$, `b` $\leftarrow 62 \Rightarrow$ `a` $+$ `b` $\leftarrow 123 \mapsto$ `b01111011`.

2.4 Wrapping it all up

We define the final round function F at round i as the hash of the previous addition, the result we XOR with the left part L of the previous round to form the new basis for the next round where L and the output of F are switched.

$$\begin{aligned} F : \omega \times \mathcal{K} \times \mathbb{N} &\to \omega \\ (R, K, i) &\mapsto \mathfrak{h}\left[A\left(R, \mu\Big(S(K,i), \|R\|\Big)\right)\right] \end{aligned} \tag{3}$$

To be able to respect FPE, a neutral byte may be added to either the output of F or the left part before applying the XOR operation.

F is applied at every round. And our implementation eventually unpads the result of $\mathfrak{F}$ by adding a final $P^{-1}()$ step at the end.

3. Full cipher

The last parameter of the whole cipher $\mathfrak{F}$ is the number of rounds N. Note that it has been proved [2] that, for such an implementation of the Feistel block cipher, four rounds of permutations are enough to make it "strong"[4].

We finally define the full cipher $\mathfrak{F}$ that respects Figure 1 with $F_i = F(R, K, i)$ at round i as follows:

$$\begin{aligned} \mathfrak{F} : \omega \times \mathcal{K} \times \mathbb{N} &\to \omega \\ (m, K, N) &\mapsto \mathfrak{F}(P(m), K, N) \end{aligned} \tag{4}$$

Recall. One of the main advantage of using this Feistel block cipher construction is that encryption and decryption are similar:

$$out = \mathfrak{F}(in, K_\gamma, n) \iff in = \mathfrak{F}(out, K_\gamma, n)$$

4. Decipher

Unlike the classic operation of a Feistel network, respecting FPE forces us to apply different operations for ciphering and deciphering. In particular, in case the input is of odd length, and the number of rounds used is also odd, we need to change a little the way the split between left and right parts is initially made when deciphering.

Whereas, the split function generally applied keeps the smallest part to the left side (when the input is of odd length), the first split must make the the right part the smallest before the first round. This is done to ensure the data provided to the round function when deciphering is the same as what was the final parts when ciphering.

III. Implementation

We enriched our initial two different implementations with the new fully FPE cipher: the one in JavaScript[5] and the one in Go[6], and we also created a specific `redacted` library to push the concept to its full achievement.

On both environments, our latest tests show no significant impact with even a 255 round FPE cipher (a few dozens of nanoseconds at most). The results are in fact mostly impacted by the speed of the used hash function on the machine it is run (and obviously a little slower in the browser of a commodity desktop computer).

References

[1] Horst Feistel. *Cryptography and Computer Privacy*, Scientific American, 1973.

[2] Michael Luby, Charles Rackoff. *How to Construct Pseudorandom Permutations from Pseudorandom Functions*, SIAM Journal on Computing, 1988.

[4] but we usually use at least 10 rounds

[5] `https://npmjs.org/package/feistel-ceipher`

[6] `https://github.com/cyrildever/feistel`