Feistel Cipher with Hash Round Function

Cyril Dever
Edgewhere

November 21, 2019

Abstract

We define an obfuscation tool to secure data with an almost Format-Preserving Encryption process. By implementing a Feistel block cipher with a round function using any robust hashing function, it provides you with a one-way tool to both encrypt and decrypt the data.

I. Introduction

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

It's secure, yet very fast, and it comes with two handy features:

- Encryption and decryption both uses the same way to work, ie. the use of only one function is needed for both obfuscating and recovering data;
- The end result respects Format-Preserving Encryption (FPE), ie. the length of the output is the same as the one of the output.

II. The Algorithm

1. Formal Description

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

We use a balanced implementation, cutting the input data into two equal parts, processing them through our round function (see section 3), to finally concatenating the end results to form the final obfuscated ciphertext.

It is an almost Format-Preserving Encryption scheme; "almost" because it depends on the size of the input. If the latter is of even length, then the output will preserve its size; otherwise, we'd pad it at the start of the process (see section 2), making it longer by one character:

$$\begin{aligned} & y \leftarrow \mathfrak{F}(x) \\ & \Rightarrow |y| = \begin{cases} \textbf{if } x \bmod 2 = 0 \textbf{ then } |x| \\ \textbf{else } |x| + 1 \end{cases} \end{aligned} \tag{1}$$

Let us start with what we use as the basis for our own implementation: the formal description provided by Wikipedia [1] 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.

Algorithm 1: Standard Feistel cipher

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

2. Padding

We could have turned our cipher into a fully FPE-compliant system by forcing the input data to be of even length.

Instead, we kept a smoother approach by deciding we'd add the padding ourselves. That way, our users don't have to worry about this

[1] https://en.wikipedia.org/wiki/Feistel_cipher

step, only that the output might be one character longer than the input (as seen above).

But of course, should you provide data of even length (using your own padding system), then our cipher definitely follows a FPE scheme.

Algorithm 2 defines our left padding function $P()$. Let `PAD_CHAR` be a padding character[2].

```
Algorithm 2: Padding $P$
  Input: a message $m$, PAD_CHAR
  Output: the balanced message
  if $|m| \bmod 2 = 0$ then
      return $m$
  else
      return PAD_CHAR $||m$
```

Algorithm 3 shows its inverse, ie. the unpadding function.

```
Algorithm 3: Unpadding $P^{-1}$
  Input: a padded message $m$, PAD_CHAR
  Output: the unpadded message
  if $m[0] =$ PAD_CHAR then
      return $||_{i=1}^{|m|-1} m[i]$
  else
      return $m$
```

3. Hash Round Function

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

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

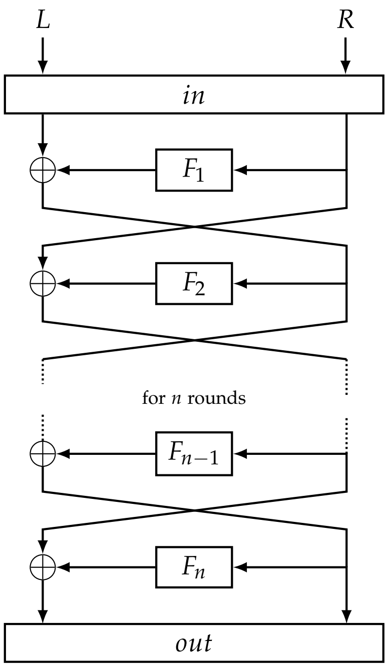

Figure 1: *Feistel block cipher* $\mathfrak{F}$

3.1 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.

$$\begin{aligned} S : \mathcal{K} \times \mathbb{N} &\to \mathcal{K} \\ (K, i) &\mapsto substr(K \ll i, 1) \,||\, K[0] \end{aligned} \tag{2}$$

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

3.2 Masking the new key

To enable the XOR part of the Feistel cipher, we have to apply a "masking" operation $\mu()$ on

[2] In our own implementation, we use the UTF-8 `U+0002` (start-of-text) character.

[3] In our first implementation, we use the `SHA-256` hash algorithm as $\mathfrak{h}()$ because it is both widely adopted (in particular natively in most browsers) and yet still very secure at the time of this writing.

the input key K to make it of length $l = |R|$:

$$\begin{aligned} \mu : \mathcal{K} \times \mathbb{N} &\to \mathcal{K} \\ (K, l) &\mapsto K' := \begin{cases} \text{if } |K| \geq l, \sum_{i=1}^{l} K[i] \\ \sum_{i=1}^{l} (K \times \lceil |K| \div l \rceil)[i] \end{cases} \end{aligned} \tag{3}$$

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.

3.3 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 4.

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

Algorithm 4: 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: $\texttt{a} \leftarrow 61, \texttt{b} \leftarrow 62 \Rightarrow \texttt{a} + \texttt{b} \leftarrow 123 \mapsto \texttt{b01111011}$.

3.4 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.

$$\begin{aligned} F : \omega \times \mathcal{K} \times \mathbb{N} &\to \omega \\ (R, K, i) &\mapsto \mathfrak{h}\left[A\left(R, \mu\left(S(K, i), |R|\right)\right)\right] \end{aligned} \tag{4}$$

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.

4. 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:

$$\begin{aligned} \mathfrak{F} : \omega \times \mathcal{K} \times \mathbb{N} &\to \omega \\ (m, K, N) &\mapsto \mathfrak{F}(P(m), K, N) \end{aligned} \tag{5}$$

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

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

III. Implementation

We created two different implementations for now: one in JavaScript[5] and one in Go [6].

On both environments, our latest tests show no significant impact with a 10 round 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 an ordinary PC).

[4] but we usually use at least 10 rounds

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

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

Contents

I Introduction **1**

II The Algorithm **1**

1 Formal Description 1

2 Padding 1

3 Hash Round Function 2

3.1 Shifting the key 2

3.2 Masking the new key . . 2

3.3 Adding parts 3

3.4 Wrapping it all up 3

4 Full cipher 3

III Implementation **3**

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.