Fountain: A Lightweight Authenticated Cipher (v1)

Submitter: Bin Zhang
martin_zhangbin@hotmail.com, +86-13520126526
TCA, SKLCS, Institute of Software, Chinese Academy of Sciences
100190, Beijing, China

Contents

Introduction

Fountain v1 is a lightweight authenticated cipher with a 128-bit secret key and a 96-bit IV. It is oriented to be efficiently implemented in the constrained hardware environments and to have a reasonably good performance in software. Fountain v1 is designed bo be secure in the nonce-respecting setting. So far, no attack faster than 2^{112} has been identified in the single key model.

Chapter 1

Specification

The specification of Fountain v1 is given in this charpter.

1.1 Parameters

Fountain v1[1] is a stream cipher-based authenticated encryption primitive. It has three parameters: key length, nonce length and tag length. The parameter space is as follows. The key length is 16 bytes, the nonce length is 12 bytes and the tag length is 8-byte or 16-byte. From a 128-bit secret key K and a 96-bit $Nonce$, or initialization vector (IV), Fountain generates the keystream of length up to 2^{64} bits.

The inputs are a public message number $Nonce$, i.e., IV, and a secret key key, a plaintext $M = (m_0, m_1, \cdots, m_{ml-1})$ of ml bits, the associated data $A = (ad_0, ad_1, \cdots, ad_{al-1})$ of al bits. The length of M is up to 2^{64} bits, i.e., less than or equal to 2^{61} bytes. The length of A is up to $2^{50} - 1$ bytes. There is no secret message number, i.e., the secret message number is empty. Formally, the authenticated encryption procedure is

$$\texttt{Fountain\_AE}\,(K, IV, A, M) = (C, T)$$

The output of the authenticated encryption is (C, T), where C is the ciphertext of the plaintext M and T is the authenticated tag of 16-byte or 8-byte, which authenticates both A and M. The length of the ciphertext is exactly the same as the plaintext M. Thus, the number of bytes in M plus the tag length in bytes equals to the output length in bytes.

The verification and decryption procedure takes as input the same secret key K and the public IV, the associated data A, the ciphertext C and the received authenticated tag T, and outputs the plaintext M only if the verification of the tag is correct or $\perp$ when the verification of the tag fails. Formally, this procedure can be written as

$$\texttt{Fountain\_VD}\,(K, IV, A, C, T) = \{M, \perp\}$$

[1]We use Fountain to denote Fountain v1 hereafter.

1.2 Recommended Parameter Sets

Primary recommended parameter set of `Fountain v1`: 16-byte (128-bit) key, 12-byte (96-bit) nonce, 16-byte (128-bit) tag. Second recommended parameter set of `Fountain v1`: 16-byte (128-bit) key, 12-byte (96-bit) nonce, 8-byte (64-bit) tag.

1.3 Operations and Variables

The following operations and variables are used in the description.

- The bitwise logic AND is denoted by $\cdot$
- The bitwise exclusive OR is denoted by $\oplus$
- The bit or bit-string concatenation is denoted by $\|$
- The associated data is A, which will not be encrypted or decrypted
- One bit of the associated data is ad_i
- The bit length of the associated data is al with $0 \leq al < 8(2^{50} - 1)$
- The plaintext is M with one bit of plaintext as m_i
- The bit length of the plaintext is ml with $0 \leq ml < 2^{64}$
- The ciphertext is C with one bit of ciphertext as c_i
- The authenticated tag is T of length 128-bit or 64 bit
- $K = (k_{127}, k_{126}, \cdots, k_1, k_0)$, the 128-bit secret key used in Fountain, where k_i for $0 \leq i \leq 127$ are the binary values with k_0 being the least significant bit and k_{127} being the most significant bit
- $IV = (iv_{95}, iv_{94}, \cdots, iv_1, iv_0)$, the 96-bit initialization vector IV used in Fountain, where iv_i for $0 \leq i \leq 95$ are the binary values with iv_0 being the least significant bit and iv_{95} being the most significant bit
- d_i for $0 \leq i \leq 3$ are the 8-bit constants used in Fountain
- The sub-string $x_b \cdots x_{a+1} x_a$ of X is $X[b:a]$

1.4 Mode of Operation

The mode of operation of Fountain is depicted in Figure 1.1, where F is the state updating function that operate on an internal state of 256 bits, and f is the output function that takes 16 state bits to generate 1 keystream bit. For a more detailed description, please see the following sections.

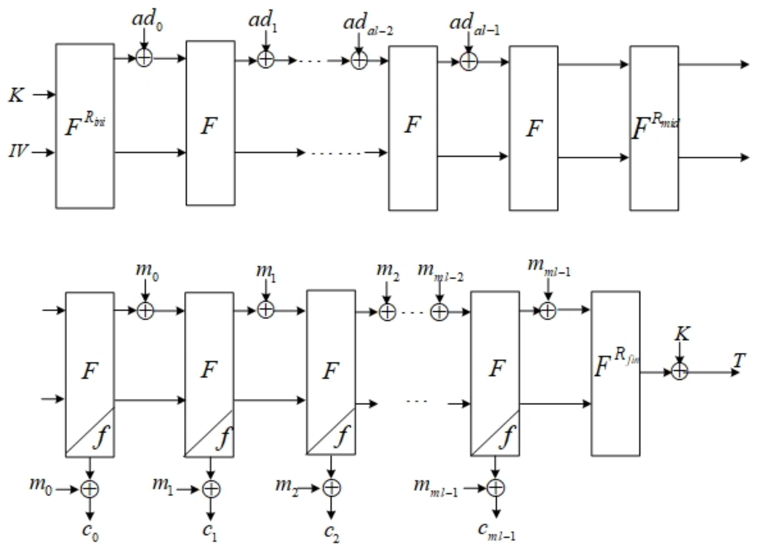

Figure 1.1: The authenticated encryption mode of Fountain v1

1.5 Description of Fountain

As depicted in Fig.1.2, there are 4 parts involved in the algorithm: 4 linear feedback shift registers (LFSR) defined over the finite field GF(2); a lightweight 4-bit to 4-bit S-box (SR), which extracts the contents of 4 LFSR cells to form the 4-bit input bits to a MDS matrix; a lightweight MDS matrix with the variable input patterns for different functionalities in AEAD; a filter function h to take the content of the current internal state to produce 1-bit keystream. Next, we will present Fountain's 4 components one-by-one.

1.5.1 The State of Fountain

Now we first look at the internal state, i.e., the 4 LFSRs in Fig.1.2. The primitive feedback polynomials of the 4 LFSRs are:

$$\begin{aligned}
&\text{LFSR1}: 1 + x^{12} + x^{25} + x^{31} + x^{64}\\
&\text{LFSR2}: 1 + x^{9} + x^{19} + x^{31} + x^{64}\\
&\text{LFSR3}: 1 + x^{14} + x^{20} + x^{31} + x^{64}\\
&\text{LFSR4}: 1 + x^{6} + x^{10} + x^{31} + x^{64}.
\end{aligned}$$

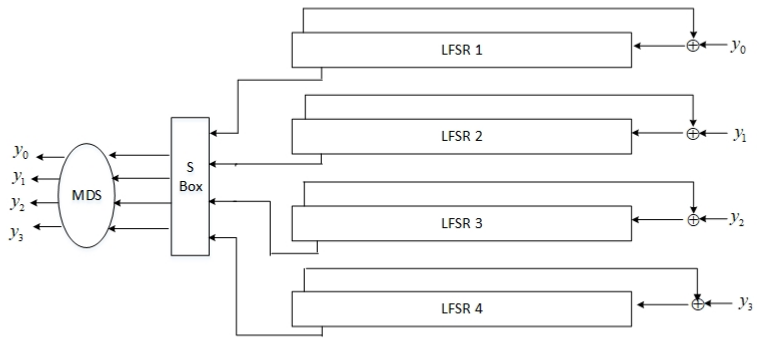

Figure 1.2: The 4 parallel LFSRs and S-box+MDS in Fountain

Table 1.1: The GIFT S-box in Fountain

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	1	A	4	C	6	F	3	9	2	D	B	7	5	0	8	E

For $i \geq 0$, the corresponding linear recursions are:

$$\begin{aligned}
\text{LFSR1}: \ & \alpha_{64+i} = \alpha_{31+i} \oplus \alpha_{25+i} \oplus \alpha_{12+i} \oplus \alpha_i \\
\text{LFSR2}: \ & \beta_{64+i} = \beta_{31+i} \oplus \beta_{19+i} \oplus \beta_{9+i} \oplus \beta_i \\
\text{LFSR3}: \ & \gamma_{64+i} = \gamma_{31+i} \oplus \gamma_{20+i} \oplus \gamma_{14+i} \oplus \gamma_i \\
\text{LFSR4}: \ & \zeta_{64+i} = \zeta_{31+i} \oplus \zeta_{10+i} \oplus \zeta_{6+i} \oplus \zeta_i \ .
\end{aligned}$$

Note that the 4 linear recursions (together with the following $h(\mathbf{x})$ and S-box+MDS) can be efficiently paralleled 32 times in harware/software implementations.

Further, for $i \geq 0$, we denote the internal state of the 4 LFSRs and the shift operation as

$$\begin{aligned}
\text{LFSR1}: \ & (\alpha_i, \alpha_{i+1}, \cdots, \alpha_{i+63}) \leftarrow (\alpha_{i+1}, \alpha_{i+2}, \cdots, \alpha_{i+64}) \\
\text{LFSR2}: \ & (\beta_i, \beta_{i+1}, \cdots, \beta_{i+63}) \leftarrow (\beta_{i+1}, \beta_{i+2}, \cdots, \beta_{i+64}) \\
\text{LFSR3}: \ & (\gamma_i, \gamma_{i+1}, \cdots, \gamma_{i+63}) \leftarrow (\gamma_{i+1}, \gamma_{i+2}, \cdots, \gamma_{i+64}) \\
\text{LFSR4}: \ & (\zeta_i, \zeta_{i+1}, \cdots, \zeta_{i+63}) \leftarrow (\zeta_{i+1}, \zeta_{i+2}, \cdots, \zeta_{i+64}) \ .
\end{aligned}$$

Note that the variables of each LFSR internal state are indexed from 0.

1.5.2 The Variable Non-linear Feedbacks

The 4-bit to 4-bit S-box is defined as above in hexadecimal. This is the 4-bit S-box used in the GIFT block cipher [4], which is known to be lightweight and

Table 1.2: The integrated S-box when generating keystream

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	9	5	6	D	8	A	7	2	E	4	C	1	F	0	B	3

Table 1.3: The integrated S-box when processing the associated data

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	9	D	E	5	8	A	F	2	6	C	4	1	7	0	B	3

can be implemented in hardware in 16 GE. The MDS matrix B is defined over GF(2^2) as

$$B = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

where the finite field GF(2^2) is defined by the primitive polynomial $x^2 + x + 1$ over GF(2). This is a lightweight MDS matrix as well, which has the maximal differential branch number 3.

For $i \geq 0$, let $y_{i+3}y_{i+2}y_{i+1}y_i$ be the 4 output bits of the matrix B, where y_i is the least significant bit and y_{i+3} is the most significant bit, then we have

$$\begin{pmatrix} y_{i+3} \\ y_{i+2} \\ y_{i+1} \\ y_i \end{pmatrix} = B \circ \mathbf{x} = B \circ \begin{pmatrix} x_{i+3} \\ x_{i+2} \\ \hline x_{i+1} \\ x_i \end{pmatrix} = B \circ \mathrm{SR} \begin{pmatrix} \zeta_{i+1} \\ \gamma_{i+1} \\ \beta_{i+1} \\ \alpha_{i+1} \end{pmatrix}$$

where $\mathbf{x} = (x_{i+3}, x_{i+2}, x_{i+1}, x_i)$ is the 4 input bits to the matrix B with the concrete patterns discussed below and $\circ$ is the matrix-vector production. The 4 second least significant bits of the LFSRs are used as the 4 input bits to the S-box SR, i.e., α_{i+1} from LFSR1, β_{i+1} from LFSR2, γ_{i+1} from LFSR3 and ζ_{i+1} from LFSR4. On the output side of the S-box, we regard the two bits $x_{i+3}\|x_{i+2}$ as an element over the finite field GF(2^2) and $x_{i+1}\|x_i$ as another element over GF(2^2).

For the different functionalities in AEAD, we set different input patterns of the 4 input bits to the matrix B. Precisely, for the normal keystream generation, the input pattern is $\mathbf{x} = (x_{i+3}\|x_{i+2}, x_{i+1}\|x_i)$ with the $x_{i+1}\|x_i$ being the least significant element over GF(2^2) and $x_{i+3}\|x_{i+2}$ being the most significant one over GF(2^2). The integrated S-box is shown in Table 1.2.

When processing the associated data A, the input pattern to B is $\mathbf{x} = (x_{i+3}\|x_{i+1}, x_{i+2}\|x_i)$ with the two bits x_{i+2}, x_{i+1} swapping their positions. The corresponding integrated S-box is shown in Table 1.3.

For the finalization phase to produce the tag T, the input pattern is $\mathbf{x} = (x_i\|x_{i+3}, x_{i+2}\|x_{i+1})$, i.e., we make a 1-bit right rotation of the input pattern

Table 1.4: The integrated S-box in finalization and the tag generation

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	B	F	E	8	7	A	2	D	9	3	4	C	5	0	6	1

$(x_{i+3}, x_{i+2}, x_{i+1}, x_i)$ in the keystream generation phase. The corresponding integrated S-box is shown in Table 1.4.

The purpose of these different input patterns is to make the domain separation in AEAD, i.e., we use different state updating functions when dealing with different kinds of functionality.

Hence, the 4 feedback bits to the 4 LFSRs are

$$\begin{aligned}
\text{LFSR1}: \;& \alpha_{64+i} = \alpha_{31+i} \oplus \alpha_{25+i} \oplus \alpha_{12+i} \oplus \alpha_i \oplus y_i \\
\text{LFSR2}: \;& \beta_{64+i} = \beta_{31+i} \oplus \beta_{19+i} \oplus \beta_{9+i} \oplus \beta_i \oplus y_{i+1} \\
\text{LFSR3}: \;& \gamma_{64+i} = \gamma_{31+i} \oplus \gamma_{20+i} \oplus \gamma_{14+i} \oplus \gamma_i \oplus y_{i+2} \\
\text{LFSR4}: \;& \zeta_{64+i} = \zeta_{31+i} \oplus \zeta_{10+i} \oplus \zeta_{6+i} \oplus \zeta_i \oplus y_{i+3} \; .
\end{aligned}$$

1.5.3 The Output Function

The output function of Fountain $h(\mathbf{x})$ is defined as $h(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = x_0x_1 \oplus x_2x_3 \oplus x_4x_5 \oplus x_6x_7 \oplus x_0x_4x_8$, where $(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8) = (\zeta_{i+2}, \alpha_{i+5}, \beta_{i+4}, \gamma_{i+11}, \zeta_{i+23}, \gamma_{i+27}, \beta_{i+24}, \alpha_{i+29}, \zeta_{i+30})$. To generate the keystream bit z_i, there are 7 bits added to the output of the h function as

$$z_i = \alpha_{i+3} \oplus \alpha_{i+11} \oplus \beta_{i+20} \oplus \gamma_{i+5} \oplus \gamma_{i+16} \oplus \zeta_{i+7} \oplus \zeta_{i+29} \oplus h(\mathbf{x}),$$

at the time instant i. Note that this output function is also used in the Grain-128a stream cipher [2].

1.5.4 The Initialization Phase

The initialization of Fountain consists of first loading the key and IV into the state, and then running the cipher for 384 steps.

The Key/IV loading scheme is as follows.

$$\begin{aligned}
&\alpha[7:0] = K_0,\ \alpha[15:8] = IV_0,\ \alpha[23:16] = K_1,\ \alpha[31:24] = IV_1, \\
&\alpha[39:32] = K_2,\ \alpha[47:40] = IV_2,\ \alpha[55:48] = K_3,\ \alpha[63:56] = IV_3 \\
&\beta[7:0] = K_4,\ \beta[15:8] = IV_4,\ \beta[23:16] = K_5,\ \beta[31:24] = IV_5, \\
&\beta[39:32] = K_6,\ \beta[47:40] = IV_6,\ \beta[55:48] = K_7,\ \beta[63:56] = IV_7 \\
&\gamma[7:0] = K_8,\ \gamma[15:8] = IV_8,\ \gamma[23:16] = K_9,\ \gamma[31:24] = IV_9, \\
&\gamma[39:32] = K_{10},\ \gamma[47:40] = IV_{10},\ \gamma[55:48] = K_{11},\ \gamma[63:56] = IV_{11} \\
&\zeta[7:0] = K_{12},\ \zeta[15:8] = K_{13},\ \zeta[23:16] = d_0,\ \zeta[31:24] = K_{14}, \\
&\zeta[39:32] = K_{15},\ \zeta[47:40] = d_1,\ \zeta[55:48] = d_2,\ \zeta[63:56] = d_3,
\end{aligned}$$

where $K = K_0, \cdots, K_{15}$ is the 16 secret key bytes and $IV = IV_0, \cdots, IV_{11}$ is the 12 IV bytes, where K_0 and IV_0 are the least significant bytes, while K_{15} and IV_{15} are the most significant bytes.

The constants d_i for $0 \leq i \leq 3$ are defined as follows.

$$
\begin{aligned}
d_0 &= 0xff \\
d_1 &= 0x3f \\
d_2 &= 0x00 \\
d_3 &= 0x80.
\end{aligned}
$$

There are 384 rounds in the initialization phase in Fountain currently, which is shown as below.

1. Load the key, IV and constants into the LFSRs as specified above.
2. for $i = 0$ to 383 do

   compute the keystream bit z_i

   run the 4 LFSRs 1 step with z_i being feedback to the 4 LFSRs

Note that in the initialization phase, the keystream bit is used to update the internal state of Fountain.

1.5.5 Processing the Associated Data

After the initialization, the associated data A is used to update the state.

1. for $i = 0$ to $al - 1$ do.

   compute the keystream bit z_i

   run the 4 LFSRs 1 step with $z_i \oplus ad_i$ being feedback to the 4 LFSRs
2. for $i = 0$ to 63 do.

   compute the keystream bit z_i

   run the 4 LFSRs 1 step with z_i being feedback to the 4 LFSRs
3. $\beta[0] = \beta[0] \oplus 1$

Note that even when there is no associated data, we still need to run the cipher for 64 steps. When we process the associated data, the keystream bit is used to update the state according to the concrete input pattern to B when computing non-feedbacks, detailed in section 1.5.2. Then we xor 1 bit information to the LFSR2's cell $\beta[0]$ so as to separate the associated data from the plaintext/ciphertext.

1.5.6 Processing the Plaintext

After processing the associated data, at each step of the encryption, one plaintext bit m_i is used to update the state, and m_i is encrypted to c_i.

```
1. for $i = 0$ to $ml - 1$ do.
       compute the keystream bit $z_i$
       $c_i = z_i \oplus m_i$
       run the 4 LFSRs 1 step with $m_i$ being feedback to the 4 LFSRs
2. $\zeta[1] = \zeta[1] \oplus 1$
```

When we process the plaintext, the keystream bit is not used to update the state, but with the concrete input pattern to B when computing non-feedbacks, detailed in section 1.5.2. The cipher specification is changed so as to separate the processing of plaintext/ciphertext and the finalization.

1.5.7 Finalization and the Tag Generation

After processing all the plaintext bits, we generate the authentication tag T.

```
1. for $i = 0$ to 383 do.
       compute the keystream bit $z_i$
       run the 4 LFSRs 1 step with $z_i$ being feedback to the 4 LFSRs
```

The authentication tag T is the xored result of the secret key and the last 128 keystream bits generated from the newest updated internal state, i.e.,

$$T = (z_{fin+127} \| \cdots, z_{fin+1} \| z_{fin}) \oplus K.$$

For the 64-bit tag version, only the xored result of the least significant 64 bits of K and the last 64 keystream bits generated from the newest internal state is adopted as the tag value. Note that in the finalization phase, the state is updated according to the concrete input pattern to the matrix B when computing non-feedbacks, detailed in section 1.5.2. This is mainly used for the domain separation.

1.5.8 The Verification and Decryption

The verification and decryption procedures are very similar to the encryption and tag generation rountine. The finalization in the decryption process is the same as that in the encryption process. We emphasize that if the verification fails, the decrypted plaintext and the newly generated authentication tag should not be given as output.

Chapter 2

Security Goals

In Fountain, each (key, IV) pair is used to protect only one message. If verification fails, the new tag and the decrypted ciphertext should not be given as output.

Algorithm	Encryption	Authentication (128/64-bit tag)
`Fountain`	112-bit	128/64-bit

There is no secret message number in Fountain. The public message number is a nonce, i.e., the IV. The cipher does not promise any integrity or confidentiality if the legitimate key holder uses the same nonce (IV) to encrypt two different (plaintext, associated data) pairs under the same key.

The security claim of Fountain is the 112-bit security in the single key setting. For the forgery attacks on the authentication tag, the security level is the same as the tag size and the IV is not allowed to be re-used. If the tag verification failed, no output should be generated.

Chapter 3

Security Analysis

In this section, we will analyze the security of Fountain with respect to several attacks.

3.1 Period and Time/Memory/Data Tradeoffs

The 256-bit state of Fountain ensures that the period of the keystream is large enough for any practical applications, but the exact value of the keystream period of Fountain is difficult to predict in theory. The average period of the keystream is estimated to be larger than 2^{128}, if we assume that the invertible state updating function of Fountain is random. Besides, the 256-bit size internal state also eliminates the threat of the known form of the time/memory/data tradeoff attacks [5, 6, 15] with respect to 112-bit security, when taking into account the pre-computation/memory/time/data complexities.

3.2 Linear Distinguishing Attacks

Here we use the linear sequential circuit approximation (LSCA) method [14] to evaluate the strength of Fountain against linear distinguishing attacks.

Let the state transition matrix after decomposition the non-linear feedback functions of Fountain be A, then it is found that the algebraic degree of the minimal polynomial of A is $m = 256$, which is denoted by $\varphi(x) = \sum_{k=0}^{m} \phi_k x^k$, precisely

$$\begin{aligned}\varphi(x) = & x^{256} + x^{223} + x^{222} + x^{216} + x^{212} + x^{211} + x^{206} + x^{203} + \\ & x^{202} + x^{201} + x^{198} + x^{190} + x^{189} + x^{183} + x^{178} + \\ & x^{177} + x^{171} + x^{170} + x^{168} + x^{166} + x^{164} + x^{162} + \\ & x^{157} + x^{156} + x^{154} + x^{151} + x^{150} + x^{149} + x^{148} + \\ & x^{146} + x^{143} + x^{142} + x^{140} + x^{139} + x^{138} + x^{136} + \\ & x^{135} + x^{133} + x^{132} + x^{128} + x^{125} + x^{124} + x^{123} + \\ & x^{118} + x^{115} + x^{113} + x^{110} + x^{104} + x^{103} + x^{102} + \\ & x^{100} + x^{97} + x^{96} + x^{95} + x^{93} + x^{92} + x^{91} + x^{90} + \\ & x^{79} + x^{71} + x^{70} + x^{66} + x^{64} + x^{63} + x^{61} + x^{56} + \\ & x^{55} + x^{51} + x^{50} + x^{48} + x^{47} + x^{44} + x^{43} + x^{42} + \\ & x^{41} + x^{39} + x^{38} + x^{37} + x^{33} + x^{31} + x^{30} + x^{29} + \\ & x^{23} + x^{21} + x^{20} + x^{17} + x^{15} + x^{14} + x^{11} + x^{10} + \\ & x^{9} + x^{6} + 1,\end{aligned}$$

where the Hamming weight of the polynomial $\varphi(x)$ is 93. Based on this polynomial, there is no linear trial with a weight of less than 56 active non-linear operations found so far. Further, we have restricted the length of each keystream generated from a (key, IV) pair to be less than or equal to 2^{64} bits, thus we feel that Fountain is immune to the linear distinguishing attacks.

3.3 Differential Cryptanalysis

In order to investigate the immunity of Fountain against differential attacks, we introduce a single bit difference at each internal state position and try to trace the propagation of this difference. We gather the difference biases after several number of initialization rounds and try to distinguish it from the purely random case.

Our experiments so far showed that for the full 384 rounds of initialization, Fountain is non-distinguishable with the purely random case with respect to the single bit differential cryptanalysis.

3.4 Cube Attacks and Variants

Cube attacks, formally introduced by Dinur and Shamir [1, 12, 13, 19], is a generic key extraction technique exploiting the simple algebraic structure of some output bits after a reasonable size of cube summation. The success of cube attacks highly depends on the sparsity of the superpoly.

Our experiments so far showed that for the full 384 rounds of initialization, Fountain seems to be secure against the current forms of cube attacks.

3.5 Guess and Determine Attacks

In guess-and-determine attacks, the adversary usually guesses the content of some partial internal state, and then tries to derive the rest part of the internal state with the knowledge of the corresponding keystream segment. We have tried some simple form of guess-and-determine attacks, and have not found an attack that has a complexity less than 2^{112}.

3.6 Security of the Authenticated Mechanism

We have considered some simple forms of forgery attacks [3] against the finalization and tag generation phase, our experiments so far showed that for the full 384 rounds of finalization, Fountain seems to be secure against the simple forms of forgery attacks.

Chapter 4

Features

Fountain has the following useful features.

- New structure of stream ciphers. Fountain is the result of some efforts to parallelize the execution of the whole cipher. The challenge in this design approach is to achieve faster diffusion speed, and this problem is solved by using four parallel linear feedback shift registers with a common non-linear feedback source. Thus it is expensive to eliminate the difference in the internal state, and it is relatively easy to analyze the authentication security.
- One message bit is injected into the internal state in 4 places in each step. This feature benefits lightweight hardware implementation, and the control circuit in the hardware implementation can be greatly simplified.
- Fountain allows parallel computation. In Fountain, 32 steps can be computed in parallel. This parallel feature benefits high speed hardware and software implementation.
- Length information of associated data and plaintext/ciphertext is not needed in Fountain, i.e., Fountain does not need to check the length of message. This feature reduces further the cost of hardware implementation.
- Efficient in Hardware. Fountain has an internal state of 256-bit, which is smaller than that of Trivium [9] and each component is chosen to be lightweight.
- Efficient in Software In Fountain, 32 steps can be computed in parallel, so its software speed is reasonably fast.
- Fountain is more hardware efficient than AES-GCM (especially for constrained hardware resource and energy consumption).

Chapter 5

Performance

5.1 Hardware

Since Fountain has an internal state of 256 bits and each component (4 LFSRs + GIFT S-box + MDS + Grain-128a filter function) is also very efficient in hardware. Thus, it is expected that Fountain will have a reasonably good performance in hardware.

5.2 Software

Since Fountain can be parallelled 32 steps in computation, it is expected that its software performance after fully optimization will be reasonably good.

Chapter 6

Design Rationale

Fountain is designed to be efficient in the constrained hardware environments, and also efficient in software on some platforms.

In order to be efficient in hardware, we use a bit-based stream cipher for its well-known hardware efficiency (such as A5/1 [15], Grain [2] and Trivium [9]). In order to resist the traditional attacks (correlation attacks [7, 8, 16, 17, 18, 20] and algebraic attacks [10, 11]) on stream cipher, the state is updated in a non-linear way. We inject the message into the internal state so that we could obtain authentication security almost for free. The challenge is that in a bit-based sequential stream cipher based on nonlinear feedback registers, it is tremendously difficult to trace the differential propagation in the state, especially if we want to achieve high authentication security (such as 128-bit). Our design focus is to solve this problem so that the authentication security could be easily analyzed. Our solution is to use the 4 parallel linear feedback shift registers to ensure that once there is difference in the state, the number of difference bits in the state would be sufficiently large before the difference gets eliminated. When there are difference bits in the state, the linear feedbacks and the non-linear feedback function (S-box + MDS) introduces the difference noise into the state quickly to reduce the success rate of forgery attack. If an attacker intends to modify the ciphertext, the difference in the keystream bits would also affect the state through the decrypted plaintext bits. In order to further reduce the hardware complexity, Fountain does not check the message length in decryption and verification. In order to make the domain separation in AEAD, different state updating functions are adopted in the different functionalities. Separating the plaintext from the associated data means that an attacker cannot use part of the plaintext bits as associated data, and vice versa. Separating the encryption/decryption from the finalization means that an attacker cannot use part of the keystream as the authentication tag.

Chapter 7

Test Vectors

Some test vectors of Fountain are provided in this chapter.

```
===========================================
Length of plaintext: 1 bytes
Length of associated data: 0 bytes
The key is: 00000000000000000000000000000000
The iv is: 000000000000000000000000
The plaintext is: 01
The associated data is
The ciphertext is: 7c
The tag is: 9837767ba440b723aee10b981d60b28e
The verification is successful in decryption
The decrypted plaintext is: 01
=============================================
Length of plaintext: 0 bytes
Length of associated data: 1 bytes
The key is: 00000000000000000000000000000000
The iv is: 000000000000000000000000
The plaintext is:
The associated data is 01
The ciphertext is:
The tag is: 31ded1e44ebf34dce767f9b0bbd55807
The verification is successful in decryption
The decrypted plaintext is:
===========================================
Length of plaintext: 1 bytes
Length of associated data: 1 bytes
The key is: 01000000000000000000000000000000
The iv is: 000000000000000000000000
The plaintext is: 00
The associated data is 00
The ciphertext is: 4e
The tag is: 78214d49298fa1ff38680b0a11a3530e
The verification is successful in decryption
The decrypted plaintext is: 00
============================================
Length of plaintext: 1 bytes
Length of associated data: 1 bytes
The key is: 00000000000000000000000000000000
The iv is: 010000000000000000000000
The plaintext is: 00
The associated data is 00
The ciphertext is: 98
The tag is: 38ab1135ee8e7771e7a241d325241e12
The verification is successful in decryption
The decrypted plaintext is: 00
============================================
Length of plaintext: 16 bytes
Length of associated data: 16 bytes
The key is: 01010101010101010101010101010101
The iv is: 010101010101010101010101
The plaintext is: 01010101010101010101010101010101
The associated data is 01010101010101010101010101010101
The ciphertext is: 3c73ebaca6d38599cbeab1e667229c61
The tag is: bd501e8bea8415dd1e8e7d026d938467
The verification is successful in decryption
The decrypted plaintext is: 01010101010101010101010101010101
===========================================
Length of plaintext: 16 bytes
Length of associated data: 16 bytes
The key is: 000102030405060708090a0b0c0d0e0f
The iv is: 000306090c0f1215181b1e21
The plaintext is: 01010101010101010101010101010101
The associated data is 01010101010101010101010101010101
The ciphertext is: 4d68e915da0d61f11eee119551056923
The tag is: e11db9d0e64739d93ddd949398ede405
The verification is successful in decryption
The decrypted plaintext is: 01010101010101010101010101010101
===========================================
Length of plaintext: 73 bytes
Length of associated data: 43 bytes
The key is: 000102030405060708090a0b0c0d0e0f
The iv is: 000306090c0f1215181b1e21
The plaintext is:
00070e151c232a31383f464d545b6269
70777e858c939aa1a8afb6bdc4cbd2d9
e0e7eef5fc030a11181f262d343b4249
50575e656c737a81888f969da4abb2b9
```

`c0c7ced5dce3eaf1f8`
The associated data is
`00050a0f14191e23282d32373c41464b`
`50555a5f64696e73787d82878c91969b`
`a0a5aaafb4b9bec3c8cdd2`
The ciphertext is:
`f72260e796a3f7179b1dc3a345fa7f40`
`bf588f52bd529f487fa841ed36c8a1a3`
`89021e0653c0303c44656ec5c128a060`
`4f92d2eaf47e1fac755ec5267586e503`
`0dcdc7705fcea00551`
The tag is: `63e37ce690ed6371d6a3f35430e96d27`
The verification is successful in decryption
The decrypted plaintext is:
`00070e151c232a31383f464d545b6269`
`70777e858c939aa1a8afb6bdc4cbd2d9`
`e0e7eef5fc030a11181f262d343b4249`
`50575e656c737a81888f969da4abb2b9`
`c0c7ced5dce3eaf1f8`

Bibliography

[1] Aumasson, J.-P., Dinur, I., Meier, W., Shamir, A.: Cube testers and key recovery attacks on reduced-round MD6 and Trivium, *Fast Software Encryption-FSE'2009*, LNCS vol. 5665, Springer, Heidelberg,(2009), pp. 1-22.

[2] Agren M., Hell, M., Johansson T., and Meier, W.: Grain-128a: a new version of Grain-128 with optional authentication. *International Journal of Wireless and Mobile Computing (IJWMC)* vol. 5, No. 1, pp. 48-59. (2011)

[3] Agren, M., Hell, M., Johansson, T.: On hardware-oriented message authentication with applications towards RFID, in Proceedings of the 2011 Workshop on Lightweight Security and Privacy: Devices, Protocols, and Applications, E. Savas, A. A. Selcuk, and U. Uludag, Eds., pp. 26-33. (2011).

[4] Banik, S., Pandey, S. K. , Peyrin T., Sasaki Y., Sim S. M., Todo Y.: GIFT: A Small Present - Towards Reaching the Limit of Lightweight Encryption, *Cryptographic Hardware and Embedded Systems–CHES'2017*, LNCS vol. 10529, Springer, Heidelberg,(2017), pp. 321-345.

[5] Biryukov, A., Shamir, A.: Cryptanalytic Time/Memory/Data tradeoffs for stream ciphers, *Advances in Cryptology-ASIACRYPT'2000*, LNCS vol. 1976, Springer, Heidelberg,(2000), pp. 1-13.

[6] Biryukov, A., Shamir, A., Wagner, D.: Real time cryptanalysis of A5/1 on a PC. *Fast Software Encryption–FSE 2000*, LNCS, vol. 1978, pp. 1-18, Springer, Heidelberg, 2001.

[7] Canteaut A. and Trabbia. M., Improved fast correlation attacks using parity-check equations of weight 4 and 5. In Preneel B. (eds), *Advances in Cryptology–EUROCRYPT 2000*, LNCS vol. 1807, pp. 573–588, Springer Berlin Heidelberg, 2000.

[8] Chose P., Joux A. and Mitton M., Fast correlation attacks: an algorithmic point of view. In Knudsen L. R. (eds), *Advances in Cryptology–EUROCRYPT 2002*. LNCS vol. 2332, Springer Berlin Heidelberg, pp. 209-221, 2002.

[9] Christophe De Cannière, Preneel, B: Trivium, *New Stream Cipher Designs–The eSTREAM Finalists: 2008*, LNCS vol. 4986, Springer, Heidelberg,(2008), pp. 244-266.

[10] Courtois N. T., Weier. W., Algebraic attacks on stream ciphers with linear feedback, In Biham E. (eds), *Advances in Cryptology–EUROCRYPT'2003*, LNCS vol.2656, Springer-Verlag, pp. 345–359, 2003.

[11] Courtois N. T., Fast algebraic attacks on stream ciphers with linear feedback, In Boneh D. (eds), *Advances in Cryptology–CRYPTO'2003*, LNCS vol.2729, Springer-Verlag, pp. 176–194, 2003.

[12] Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials, *Advances in Cryptology-EUROCRYPT'2009*, LNCS vol. 5479, Springer, Heidelberg,(2009), pp. 278-299.

[13] Dinur, I., Shamir, A.: Breaking Grain-128 with dynamic cube attacks, *Fast Software Encryption-FSE'2011*, LNCS vol. 6733, Springer, Heidelberg,(2011), pp. 167-187.

[14] Golić Jovan Dj.: Correlation properties of a general binary combiner with memory, *Journal of Cryptology*, vol.9, Springer-Verlag. pp. 111-126, (1996).

[15] Golić Jovan Dj.: Cryptanalysis of alleged A5 stream cipher, *Advances in Cryptology-EUROCRYPT'1997*, LNCS vol.1233, Springer-Verlag. pp. 239-255, (1997).

[16] Willi, M., Staffelbach, O.: Fast correlation attacks on certain stream ciphers. *Journal of Cryptology*, 1(3): 159-176, 1989.

[17] Johansson T. and Jönsson F., Improved fast correlation attacks on stream ciphers via convolutional codes, In Stern J. (eds), editor, *Advances in Cryptology–EUROCRYPT'99*, LNCS vol. 1592, pp. 347–362, Springer Berlin / Heidelberg, 1999.

[18] Johansson T. and Jönsson F., Fast correlation attacks through reconstruction of linear polynomials, In Bellare M. (eds), *Advances in Cryptology–CRYPTO 2000*, LNCS vol. 1880, pp. 300-315, 2000.

[19] Todo Y., Structural Evaluation by Generalzed Integral Property, In Oswald E. and Fischlin M. (eds), *Advances in Cryptology–EUROCRYPT'2015*, LNCS vol. 9056, pp. 287-314, 2015.

[20] Zhang B., Xu C., and Meier W., Fast correlation attacks over extension fields, Large-unit linear approximation and Cryptanalysis of SNOW 2.0, In Gennaro H. and Robshaw M. (eds), *Advances in Cryptology–CRYPTO'2015*, LNCS vol. 9215, pp. 643-662, 2015.