Analysis and Improvements of the Full Spritz Stream Cipher

Banik, Subhadeep; Isobe, Takanori; Morii, Masakatu

2017

PAPER

Analysis and Improvements of the Full Spritz Stream Cipher*

Subhadeep BANIK[†], Takanori ISOBE[††a)], *Nonmembers*, and Masakatu MORII[†††], *Senior Member*

SUMMARY Spritz is a stream cipher proposed by Rivest and Schuldt at the rump session of CRYPTO 2014. It is intended to be a replacement of the popular RC4 stream cipher. In this paper we propose distinguishing attacks on the full Spritz, based on *a short-term bias* in the first two bytes of a keystream and *a long-term bias* in the first two bytes of every cycle of N keystream bytes, where N is the size of the internal permutation. Our attacks are able to distinguish a keystream of the *full* Spritz from a random sequence with samples of first two bytes produced by $2^{44.8}$ multiple key-IV pairs or $2^{60.8}$ keystream bytes produced by a single key-IV pair. These biases are also useful in the event of plaintext recovery in a broadcast attack. In the second part of the paper, we look at a state recovery attack on Spritz, in a special situation when the cipher enters a class of weak states. We determine the probability of encountering such a state, and demonstrate a state recovery algorithm that betters the 2^{1400} step algorithm of Ankele et al. at Latincrypt 2015. Finally we propose a simple fix that removes the bias in the first two keystream bytes. The countermeasure requires only one additional memory access and hence does not diminish software performance substantially, and in fact the loss in software speed is only around 1.5%.
key words: *RC4, Spritz, stream cipher, short-term bias, long-term bias, distinguishing attack, plaintext recovery attack, state recovery attack*

1. Introduction

RC4, designed by Rivest in 1987, is still one of most widely used stream ciphers in the world. It is adopted in many software applications and standard protocols such as SSL/TLS, WEP, Microsoft Lotus and Oracle secure SQL. After the disclosure of its algorithm in 1994, RC4 has attracted intensive cryptanalytic efforts over past 20 years. Finally, in 2013, practical plaintext recovery attacks on RC4 in SSL/TLS were proposed by AlFardan et al. [2] and Isobe et al. [12]. In response to these results, usage of RC4 has drastically decreased, especially in TLS, and major companies such as Google, Microsoft, and Mozilla announced that they will officially remove the RC4 from web browsers by early 2016.

At the same time, there has been extensive research in recent years to come up with RC4-like stream ciphers

that while marginally slower in software, would wipe out the known shortcomings of RC4. Many such ciphers like RC4A [20], NGG [17], GGHN [11], Quad-RC4 [19], RC4+ [13] and VMPC [27] have been proposed to fulfil this objective. However, all the aforementioned ciphers have had distinguishing attacks reported against them [5]–[7], [15], [21], [23], [24]. Spritz [22] is a stream cipher proposed by Rivest and Schuldt at the rump session of CRYPTO 2014. The authors intended Spritz to be a replacement for RC4, and hence the design for Spritz was chosen meticulously, with special attention given to the fact that known weaknesses of RC4 [14], [16] do not carry over. The authors automatically examined many thousands of candidates to obtain cryptographically secure update functions and an estimated 5 "core-months" of CPU time were used in the statistical experiments performed by them. Their experiments suggested that 2^{81} samples were required to distinguish the output of Spritz from random.

1.1 Description of Spritz

Spritz consists of a permutation S over the set $\{0, 1, 2, \ldots, N-1\}$ (default value of N is 256) and six pointers i, j, k, w, a, z, where i, j, k are index pointers, w gives the step distance for i, a is a nibble counter, and z stores the output byte. The design specifies a number of modules that are executed for producing a keystream as defined in Fig. 1. The authors specify a number of modes of operation using the Spritz structure like a stream cipher, hash function, MAC etc. In the stream cipher mode of operation the keystream is produced in the following manner. First the permutation is initialized using the INITIALIZESTATE(N) routine. The secret key K is then absorbed into the state using the ABSORB(K) module. Additionally, if an IV is to be used, then the ABSORBSTOP() module is invoked and the IV is absorbed by calling the ABSORB(IV) function. Thereafter, the SQUEEZE module is invoked to produce keystream bytes.

1.2 Previous Work

The only published work on cryptanalysis of Spritz is presented in [3]. The authors tackle the problem of state recovery using three different approaches. The best algorithm they propose theoretically recovers the internal permutation used in Spritz in 2^{1400} steps. Additionally, in [26], the author proposed a distinguisher for a scaled down version of Spritz ($N = 8$). It was observed that the event $Z_i = Z_{i+2}$ was bi-

INITIALIZESTATE(N)

1. $i = j = k = a = z = 0$, $w = 1$.
2. **for** $v \rightarrow 0$ to $N - 1$
   $S[v] = v$

ABSORB(I)

1. **for** $v \rightarrow 0$ to $I.length - 1$
   ABSORBBYTE($I[v]$)

ABSORBBYTE(b)

1. ABSORBNIBBLE($low(b)$)
2. ABSORBNIBBLE($high(b)$)

ABSORBNIBBLE(x)

1. **if** $a = \lfloor \frac{N}{2} \rfloor$
   SHUFFLE()
2. SWAP($S[a], S[\lfloor N/2 \rfloor + x]$)
3. $a = a + 1$

ABSORBSTOP()

1. **if** $a = \lfloor \frac{N}{2} \rfloor$
   SHUFFLE()
2. $a = a + 1$

SHUFFLE()

1. WHIP($2N$)
2. CRUSH()
3. WHIP($2N$)
4. CRUSH()
5. WHIP($2N$)
6. $a = 0$

WHIP(r)

1. **for** $v \rightarrow 0$ to $r - 1$
   UPDATE()
2. **do** $w = w + 1$
   until $gcd(w, N) = 1$

CRUSH()

1. **for** $v \rightarrow 0$ to $\lfloor N/2 \rfloor - 1$
   if $S[v] > S[N - 1 - v]$
   SWAP($S[v], S[N - 1 - v]$)

SQUEEZE(r)

1. **if** $a > 0$
   SHUFFLE()
2. $P = Array.New(r)$
3. **for** $v \rightarrow 0$ to $r - 1$
   $P[v] = $ DRIP()
4. **return** P

DRIP()

1. **if** $a > 0$
   SHUFFLE()
2. UPDATE()
3. **return** OUTPUT()

UPDATE()

1. $i = i + w$
2. $j = k + S[j + S[i]]$
3. $k = i + k + S[j]$
4. SWAP($S[i], S[j]$)

OUTPUT()

1. $z = S[j + S[i + S[z + k]]]$
2. **return** z

Fig. 1 Modules for Spritz. When N is a power of 2, the last two lines of WHIP are equivalent to $w = w + 2$.

ased. However, the bias was not theoretically proven and no analogous result for the full Spritz ($N = 256$) was proposed.

1.3 Our Contribution and Organization

In this paper, we first show *a short-term bias* which is present in the first two bytes of a keystream and *a long-term bias* which appears in the first two bytes of every cycle of N keystream bytes. We theoretically prove that these biases exist in a keystream of Spritz regardless of the value of N. Based on these biases, we propose distinguishing attacks on the full Spritz ($N = 256$). Our attacks are able to distinguish a keystream of the full Spritz from a random sequence with samples of first two bytes produced by $2^{44.8}$ multiple key-IV pairs or $2^{60.8}$ keystream bytes produced by a single key-IV pair. These biases are applicable to a plaintext recovery

Table 1   Summary of results on Spritz.

	Type of Attack	Complexity	Reference
1	Distinguishing attack on scaled down version ($N = 8$)	$2^{21.9}$ outputs	[26]
2	Distinguishing attack on full Spritz (multiple key-IV)	$2^{44.8}$ outputs	Section 2
3	Distinguishing attack on full Spritz (single key-IV)	$2^{60.8}$ outputs	Section 2
4	State recovery attack	2^{1400} steps	[3]
		2^{1247} steps	Section 3

attack in a broadcast setting and multi-session setting in SSL/TLS.

Thereafter we show that under certain conditions, Spritz enters a weak class of states, during which, the odd and even elements of the permutation are never swapped with each other. In this case, the sequence constructed with the last bit of every keystream byte becomes periodic with period equal to 4. We show that in such an event, a state recovery attack on Spritz is more efficient and improves upon the 2^{1400} step algorithm proposed in [3]. Table 1 shows the summary of our results.

Finally we propose a simple countermeasure that removes the bias in the first two keystream bytes produced by the cipher. In order to do so, we tweak the output function of the cipher in a manner that requires only one additional memory access and at most two additional assembly instructions on any standard CPU. As such the software speeds of the cipher after adding the countermeasure does not decrease significantly. Our findings indicate that for a CPU supporting the `x86_64` instruction set, the countermeasure decreases the speed by around 1.5%.

The paper is organized in the following manner. In Sect. 2, we will present the distinguisher on Spritz and study a few of its implications. In Sect. 3, we will present our state recovery attack on Spritz. In Sect. 4, we will outline the details of the countermeasure. Section 5 concludes the paper.

2.   Distinguishing Attacks on Spritz

Before we proceed to outline the details of the distinguisher, let us present a few observations on how the various index pointers are used when Spritz is operated in the stream cipher mode. Note that when Spritz is used in the stream cipher mode: the sequence of execution of modules is

A. ABSORB(K)
B. ABSORBSTOP(),
C. ABSORB(IV) (**optional, only if IV is used**)
D. SQUEEZE().

1. In the ABSORB(K) (and also ABSORB(IV)) phase, the internal permutation is swapped according to the nibble values of the key (IV). During this phase the index a is used only to keep track of the number of

nibbles currently absorbed in the permutation. After the ABSORB phase, the index a plays no further role in the SQUEEZE phase when the cipher starts producing keystream bytes.

2. The index w, which is used to increment the index i, is constant during the SQUEEZE phase. The value of this index does not depend on the secret key, and hence is not secret. Its value can be deduced from the length of the secret key and IV. If the length of key is limited to $\lfloor N/4 \rfloor$ bytes, and no IV is used, then the SHUFFLE procedure is executed only once. In that case, the value of w during the SQUEEZE phase is 7.

3. If the length of the Key is more than $\lfloor N/4 \rfloor$ bytes the value of w can be deduced by examining the number of times the SHUFFLE module has been called during the ABSORB phases. For example, if $N = 256$, and a Key of size 80 bytes, the SHUFFLE procedure gets called twice, at the end of the 64th byte and at the beginning of SQUEEZE. Each SHUFFLE call increases the value of w by 6 and so the value of w during the keystream generation is $1 + 6 + 6 = 13$.

4. The value of the index i at the beginning of the SQUEEZE phase is always 0, whatever be the size of the Key and IV used in the ABSORB phases. This is because whenever $\lfloor N/4 \rfloor$ bytes get absorbed, the value of the pointers i, j, k are altered by call to the SHUFFLE module. Each SHUFFLE module calls the WHIP($2N$) module thrice. Each WHIP module in turn updates i using the rule $i = i+w$ a total of $2N$ times. Whatever be the actual value of w, at the end of the any call to the WHIP module, the updated value of $i = 0 + 2wN \equiv 0 \bmod N$. And so the value of i remains 0 going in and out of the WHIP executions and hence also the SHUFFLE module.

5. The only indices that change during the SQUEEZE phase is i, j, k, z.

6. The sequence of updates during the SQUEEZE phase is therefore given as:

    a. $i = i + w$
    b. $j = k + S[j + S[i]]$
    c. $k = k + i + S[j]$
    d. SWAP ($S[i], S[j]$)
    e. **return** $z = S[j + S[i + S[z + k]]]$

Although the sequence of steps in Spritz is quite complicated, for the purpose of this paper it is sufficent to view ABSORB(K), ABSORBSTOP(), ABSORB(IV) as a black-box module that takes as input K, IV and outputs a random permutation S on $\{0, 1, 2, \ldots, N-1\}$ and byte indices i, j, k, w. The process is explained graphically in Fig. 2.
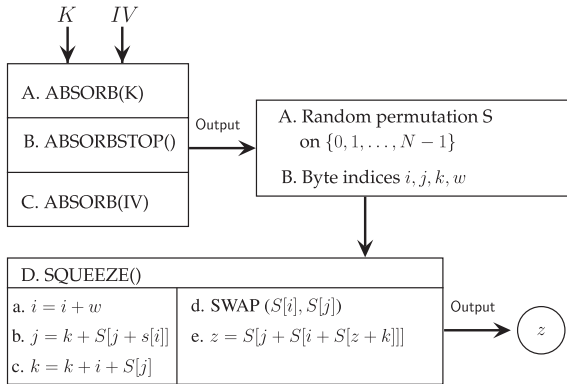
Fig. 2 Graphical representation of the SPRITZ procedure.

2.1 Bias in First Two Output Bytes of a Keystream

We first prove that the first two output bytes produced by the Spritz stream cipher are biased towards the tuple $(-w, -w)$. For example, if $N = 256$, and if a 64-byte key is used, then $w = 7$, and then the first 2 bytes are biased towards the value $(249, 249)$.

Theorem 1: The first two output bytes Z_1 and Z_2 produced by the Spritz stream cipher are biased towards $(-w, -w)$. The probability of this event is given by $\Pr[Z_1 = Z_2 = -w] = \frac{1}{N^2} + \frac{3}{N^4}$.

Proof 1: We outline three mutually exclusive events **I, II** and **III**, each of which occurs with probability $\frac{1}{N^4}$, that guarantees that the first two output bytes produced by the cipher are both equal to $-w$. Each of the three events are denoted by the states of the permutation and the values of the index pointers before the beginning of the SQUEEZE phase.

I. $S[w] = -w$, $S[2w] = 0$, $k = 0$, $S[j - w] = 2w$
II. $k = 2w$, $S[j + S[w]] = -2w$, $S[2w] = w$, $S[0] = -w$
III. $k + S[j - w] = 2w$, $k + S[2w] = 0$, $S[w - k] = 0$, $S[w] = -w$

For example, when **I** occurs in the first round we have the following changes : 1. $i \leftarrow i + w = w$
2. $j \leftarrow 0 + S[j + S[w]] = S[j - w] = 2w$
3. $k \leftarrow k + i + S[j] = 0 + w + S[2w] = 0 + w + 0 = w$
4. $S[w] \leftarrow 0$, $S[2w] \leftarrow -w$ after SWAP
5. $z \leftarrow S[j + S[i + S[z + k]]] = S[2w + S[w + S[w]]] = S[2w + S[w]] = S[2w] = -w$

Similarly in the second round we have the following changes:
1. $i \leftarrow i + w = 2w$,
2. $j \leftarrow w + S[2w + S[2w]] = w + S[w] = w$
3. $k \leftarrow k + i + S[j] = w + 2w + S[w] = 3w + 0 = 3w$
4. $S[w] \leftarrow -w$, $S[2w] \leftarrow 0$ after SWAP
5. $z \leftarrow S[w + S[2w + S[3w - w]]] = S[w + S[2w + S[2w]]] = S[w] = -w$

We get similar results when we analyze **II** and **III**. Let us

now denote by E the union of the events **I, II** and **III**. We have $\Pr[\mathsf{E}] = \frac{3}{N^4}$, and $\Pr[Z_1 = Z_2 = -w|\mathsf{E}] = 1$. We assume that when E does not occur $\Pr[Z_1 = Z_2 = -w|\mathsf{E}^c] = \frac{1}{N^2}$, and is more or less uniformly random. We were able to verify the assumption by running computer simulations. Therefore by Bayes theorem, we have:

$$\Pr[Z_1 = Z_2 = -w] = \Pr[Z_1 = Z_2 = -w|\mathsf{E}] \cdot \Pr[\mathsf{E}] +$$
$$\Pr[Z_1 = Z_2 = -w|\mathsf{E}^c] \cdot \Pr[\mathsf{E}^c]$$
$$= 1 \cdot \frac{3}{N^4} + \frac{1}{N^2} \cdot \left[1 - \frac{3}{N^4}\right]$$
$$\approx \frac{1}{N^2} + \frac{3}{N^4}$$
□

2.1.1 Experimental Results:

By performing extensive computer simulations with **(a)** one billion random keys, and **(b)** a fixed key with one billion random IVs, the probability $\Pr[Z_1 = Z_2 = -w]$ was found to be around $\frac{1}{N^2} + \frac{2.9}{N^4}$ for $N = 16$ and $N = 32$. In Fig. 3, we plot $\left[\Pr[(Z_1, Z_2) = x] - \frac{1}{N^2}\right] \cdot N^4$ for all values of x when $N = 16$ and 32 with $w = 7$. The x-axis is marked as $NZ_1 + Z_2$. We can see a sharp peak at the x-axis mark corresponding to $(-7, -7)$ (i.e. $9 * 16 + 9 = 153$ for $N = 16$ and $25 * 32 + 25 = 825$ for $N = 32$). The plot is not uniform and there seems to be some bias for other values of x too, but the most significant bias exists at the point corresponding to $(-w, -w)$.

2.2 Distinguishing Attack with Multiple Key-IV Pairs Based on a Short-Term Bias

We now state the following theorem from [14], which outlines the number of output samples required to distinguish two distributions X and Y.

Theorem 2: (Mantin-Shamir [14]) Let X, Y be distributions, and suppose that the event e happens in X with probability p and in Y with probability $p(1 + q)$. Then for small p and q, $O\left(\frac{1}{pq^2}\right)$ samples suffice to distinguish X from Y with a constant probability of success.

Let X be the probability distribution of Z_1 and Z_2 in an ideal random stream, and let Y be the probability distribution of Z_1 and Z_2 in streams produced by Spritz for randomly chosen keys. Let the event e denote $Z_1 = Z_2 = -w$, which occurs with probability of $\frac{1}{N^2}$ in X and $\frac{1}{N^2} + \frac{3}{N^4} = \frac{1}{N^2} \cdot \left(1 + \frac{3}{N^2}\right)$ in Y. By using the Theorem 2 with $p = \frac{1}{N^2}$ and $q = \frac{3}{N^2}$, we can conclude that we need about $\frac{1}{pq^2} = \frac{N^6}{9} \approx 2^{44.8}$ output samples to reliably distinguish the two distributions.

Therefore, we can mount a distinguishing attack with multiple key-IV pairs, if output samples of Z_1 and Z_2 produced by $2^{44.8}$ distinct key-IV pairs are available. In the single key setting, it requires samples of first two bytes Z_1 and Z_2 generated by $2^{44.8}$ different IVs.
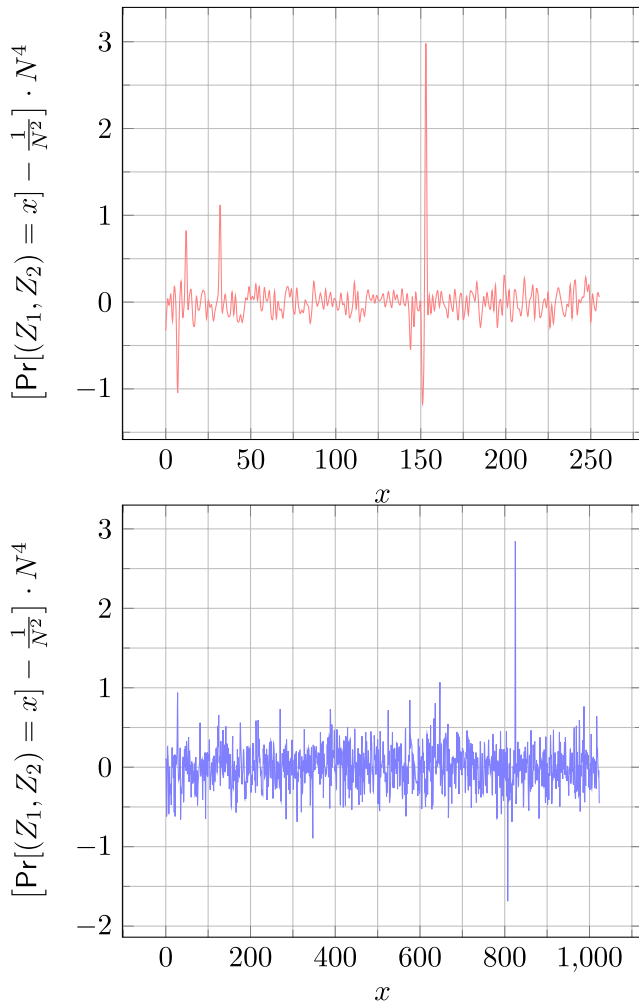
Fig. 3 $\left[\Pr[(Z_1, Z_2) = x] - \frac{1}{N^2}\right] \cdot N^4$ (for $N = 16, 32$).

2.3 Distinguishing Attack with a Single Key-IV Pair Based on a Long-Term Bias

The distinguishing attack on Spritz described in Theorem 1 requires that i and z are both zero at the beginning of the SQUEEZE phase. In general, during the production of a single stream of keystream bytes from any key or key/IV pair i and z are not both zero at the beginning of each round. This is why although the result in Theorem 1, holds for distinguishing the first 2 output bytes produced by multiple key/IV pairs, the same result can not be translated for a single keystream byte sequence using the event $Z_t = Z_{t+1} = -w$.

However i becomes 0 after every N rounds, and so in order to distinguish a single sequence of keystream bytes, one could look at the event $Z_{mN+1} = Z_{mN+2} = -w$ (for all integers $m \geq 0$) i.e. the first two of every **cycle** of N keystream bytes. However we still need $Z_{mN} = 0$ for the initial conditions of the distinguisher to be fulfilled and so we should really look at the event $\Pr[Z_{mN+1} = Z_{mN+2} = -w|Z_{mN} = 0]$. For the reasons outlined in Theorem 1, we

also have

$$\Pr[Z_{mN+1} = Z_{mN+2} = -w|Z_{mN} = 0] = \frac{1}{N^2} + \frac{3}{N^4}$$

where the probability this time is calculated over several integral values of m. Note that we will need $T = O(\frac{N^6}{9}) \approx 2^{44.8}$ samples to reliably distinguish the stream. However for this we need $T \cdot N$ **cycles** of keystream bytes (as $Z_{mN} = 0$ will on average occur once every N **cycles**) and hence $T \cdot N^2 = O(\frac{N^8}{9}) \approx 2^{60.8}$ keystream bytes. The distinguishing attack was verified for 100 random keys for $N = 16, 32$.

2.4 Plaintext Recovery Attacks in the Broadcast Setting

These short- and long-term biases are also used for plaintext recovery attacks in the broadcast setting where the same plaintext is encrypted with different keys or/and IV in the same manner of previous attacks [2], [12], [14], [18]. Note that the broadcast setting is converted into the multi-session setting where the target plaintext block are repeatedly sent in the same position in the plaintexts in multiple SSL/TLS sessions. According to Theorem 2, given $\frac{1}{pq^2}$ ciphertexts, we can distinguish the distribution of correct candidates of plaintext bytes (the biased distribution) from the distribution of wrong candidates of plaintext bytes (a random distribution) with a constant probability. It can be considered as the lower bound of the required number of ciphertexts for recovering biased bytes of a plaintext in this setting as mentioned in [14]. Recent statistical methods to detect a correct plaintext e.g. likelihood calculations of techniques [2], [25] and Bayesian analysis [10] might help to reduce the required number of ciphertexts when mounting an actual attack.

3. State Recovery Attack on Spritz

We first look at a class of special states of the Spritz stream cipher that occurs just before the beginning of the SQUEEZE phase.

Definition 1: Define a Spritz state as the 3-tuple (S, j, k) just at the beginning of the SQUEEZE phase. A Spritz state is called a SPECIAL state if all the following conditions hold simultaneously.

1. $S[t] \equiv 0 \bmod 2$, if $t \equiv 1 \bmod 2$,
2. $S[t] \equiv 1 \bmod 2$, if $t \equiv 0 \bmod 2$,
3. $j \equiv 0 \bmod 2$ and $k \equiv 0 \bmod 2$

In other words a SPECIAL state occurs when all the even indexed positions of the S array hold odd values, all the odd indexed positions hold even values and additionally j and k are even. We will now show that if the state at the beginning of the SQUEEZE phase is a SPECIAL state, then the sequence $Z_t \bmod 2$, $t = 0, 1, 2, 3, \ldots$ is periodic with period equal to 4.

Lemma 1: If the state at the beginning of the SQUEEZE phase is a SPECIAL state then the following hold (assuming

Table 2 The modulo 2 values of the various indices through 4 iterations. The ones marked with * are used in the State recovery process in Algorithm 1.

#	Index	$t = 1$	$t = 2$	$t = 3$	$t = 4$
1	$i = i + w$*	1	0	1	0
2	$j + S[i]$*	0	0	0	0
3	$j = k + S[j + S[i]]$*	1	0	1	0
4	$k = k + i + S[j]$	1	0	1	0
5	$z + k$*	1	0	0	1
6	$i + S[z + k]$*	1	1	0	0
7	$j + S[i + S[z + k]]$*	1	0	0	1
8	$z = S[j + S[i + S[z + k]]]$	0	1	1	0

N is even):

a) The state after every four iterations is a SPECIAL state.

b) In every iteration, the updated values of i and j are equal modulo 2. Hence no SWAP between odd and even values occur. And so, even and odd indexed positions of the S array will continue to hold odd and even values respectively.

c) $Z_t \equiv Z_{t+4} \mod 2$, for all values of t.

Proof 2: Note that i and z are 0 at the beginning of the SQUEEZE phase and so both are even to begin with. If N is even, the design of the WHIP module ensures that the value of w is odd, whatever be the length of key/IV. Thereafter, all the above claims can be verified by running four iterations of the UPDATE function. We summarize the modulo 2 values of the various indices over 4 iterations in Table 2. Note that the updated values of i, j in each round is either both odd or both even, which means that the odd and even values are never swapped during the SQUEEZE phase. At the end of round 4, i, j, k, z become even again and so the modulo values of the above indices will repeat every 4 cycles. And therefore, the sequence of the modulo 2 values of the keystream byte z becomes periodic with period 4: $0, 1, 1, 0, \quad 0, 1, 1, 0, \quad 0, 1, 1, 0 \ldots$

□

3.1 Probability of a SPECIAL State

Combinatorially, it is easy to see that the total number of SPECIAL states is $\left(\frac{N}{2}\right)^2 \cdot \left[\left(\frac{N}{2}\right)!\right]^2$. Therefore, if carry out the key/IV Setup operation with different keys/ single key and different IVs, then the probability that the state at the beginning of the SQUEEZE state is SPECIAL is given by

$$\rho = \frac{\left(\frac{N}{2}\right)^2 \cdot \left[\left(\frac{N}{2}\right)!\right]^2}{N^2 \cdot (N!)}$$

For $N = 256$, $\rho \approx 2^{-253.7}$. So if one employs an IV of length more than 254 bits, it is likely that a SPECIAL state will be encountered in ρ^{-1} attempts. Using this, a state recovery attack can be mounted in a Multiple IV mode as follows:

1. For a fixed key, and Multiple IVs collect keystream of around $10 * N$ bytes and inspect the sequence $Z_t \mod 2$.
2. If the sequence is $0, 1, 1, 0, \quad 0, 1, 1, 0, \quad 0, 1, 1, 0 \ldots$ i.e.

periodic with period 4, then the attacker can conclude with high probability that he has encountered a SPECIAL state and he proceeds according to Algorithm 1.

3. The above technique is likely to succeed once in ρ^{-1} attempts.

3.2 State Recovery of SPECIAL States

Once the attacker is sure that he has encountered a SPECIAL state, he has the task of recovering a much simpler state and he proceeds in the same manner as in [3, Algorithm 1]. However, there are a few differences as given in Algorithm 1.

The algorithm can be summarized in the following words: In each round, the attacker guesses the value of some of the elements of the internal permutation to determine the value of all the five indices required in the state update operation, each time making sure that odd indices get even values and vice versa. He then inspects the keystream byte produced in the round and tries to determine if the intermediate guessed permutation is consistent with the keystream byte observed. The attacker computes the index $d = j + S[i + S[z + k]]$ with the guessed values of the permutation and then performs the Verification step: Depending on the comparison between $S[d]$ and the current keystream byte Z_r he makes the following transitions:

If $S[d]$=NULL and $Z_r \notin S \rightarrow$ Assign $S[d] = Z_r$, Go to next round $r + 1$

If $S[d]$=NULL and $Z_r \in S \rightarrow$ Contradiction!! Try another assignment

If $S[d] \neq$ NULL and $Z_r \neq S[d] \rightarrow$ Contradiction!! Try another assignment

If $S[d] \neq$ NULL and $Z_r = S[d] \rightarrow$ Go to next round $r + 1$

3.3 Complexity of the Algorithm

The complexity is given by the number of guesses or assignments made, until a solution is found. As in [3], we compute the complexity by splitting the algorithm in several cases $c_i(x)$ to which we assign probabilities according to the occurrence of each case. Note that we can view the above internal state recovery algorithm, as two modules each working to recover exactly one half of the elements of the permutation. This is true since, the odd and the even indices never swap among each other. Let us denote by $\mathcal{T}_1, \mathcal{T}_2$ as the average number of assignments that would made in recovering the odd/even indexed elements of the permutation, if they were operating independent of the other. Since for every assignment in $\mathcal{T}_1$ we would need $\mathcal{T}_2$ assignments to verify the correctness of the solution, the total complexity of our algorithm is $\mathcal{T} = \mathcal{T}_1 \cdot \mathcal{T}_2$.

To estimate $\mathcal{T}_1$, we have to note the parity of the odd indices assigned in every cycle. We already know that the parity of all the indices will repeat after every 4 rounds, so observing the first 4 cycles is sufficient. As per Algorithm 1, the five indices that are used in the assignment process

---

Algorithm 1: State Recovery Algorithm for SPE-CIAL states

Input: Keystream bytes Z_t for $t = 0$ to $10 * N$;
Output: Permutation S at the beginning of SQUEEZE stage;

$S[t] \leftarrow$ NULL for $t = 0$ to $N - 1$;
Run $StateRecovery(S, i, j, k, 0)$;

$StateRecovery(S, i, j, k, r)$;
$i_{next} \leftarrow i + w$;
if $S[i_{next}] = $ NULL $\wedge u_1$ *is not in* $S \wedge u_1 \not\equiv i_{next} \bmod 2$ **then**
   | Assign $S[i_{next}] \leftarrow u_1$ /* for $u_1 \leftarrow 0$ to $N - 1$ */
end

$a = j + S[i_{next}]$;
if $S[a] = $ NULL $\wedge u_2$ *is not in* $S \wedge u_2 \not\equiv a \bmod 2$ **then**
   | Assign $S[a] \leftarrow u_2$ /* for $u_2 \leftarrow 0$ to $N - 1$ */
end

$j_{next} \leftarrow j + S[a]$;
if $S[j_{next}] = $ NULL $\wedge u_3$ *is not in* $S \wedge u_3 \not\equiv j_{next} \bmod 2$ **then**
   | Assign $S[j_{next}] \leftarrow u_3$ /* for $u_3 \leftarrow 0$ to $N - 1$ */
end

$k_{next} \leftarrow k + i_{next} + S[j_{next}]$;
SWAP $(S[i_{next}], S[j_{next}])$;
$b \leftarrow Z_{r-1} + k_{next}$;
if $S[b] = $ NULL $\wedge u_4$ *is not in* $S \wedge u_4 \not\equiv b \bmod 2$ **then**
   | Assign $S[b] \leftarrow u_4$ /* for $u_4 \leftarrow 0$ to $N - 1$ */
end

$c \leftarrow i_{next} + S[b]$;
if $S[c] = $ NULL $\wedge u_5$ *is not in* $S \wedge u_5 \not\equiv c \bmod 2$ **then**
   | Assign $S[c] \leftarrow u_5$ /* for $u_5 \leftarrow 0$ to $N - 1$ */
end

$d \leftarrow j_{next} + S[c]$;
if $S[d]$ *is* NULL $\wedge Z_r$ *is not in* S **then**
   | Assign $S[d] \leftarrow Z_r$;
   | $StateRecovery(S, i_{next}, j_{next}, k_{next}, r + 1)$;
end
if $S[d]$ *is* NULL $\wedge Z_r$ *is in* S **then**
   | Contradiction /*Try another assignment */;
end
if $S[d]$ *is not* NULL $\wedge S[d] \neq Z_r$ **then**
   | Contradiction /*Try another assignment */;
end
if $S[d]$ *is not* NULL $\wedge S[d] = Z_r$ **then**
   | $StateRecovery(S, i_{next}, j_{next}, k_{next}, r + 1)$;
end

---

are $i_{next}, a, j_{next}, b, c$, and the index used in the verification process is d. It is easy to see that these correspond to $i, j + S[i], j, z + k, i + S[z + k]$ and $j + S[i + S[z + k]]$ respectively. A quick look at Table 2, tells us four of the assignment indices and the only verification index are odd in the first round. Thereafter the second and third rounds have one and two assignment indices odd. The fourth round has one assignment and one verification index odd. This means that there are four assignments followed by a verification, which is followed by another cycle of four assignments and a verification. Therefore in total we have 10 stages of assignment/verification. Let $c_i[x]$ ($1 \leq i \leq 10$) denote the average

complexity associated with each stage, assuming that x elements of the $N/2$ odd-indexed positions are already filled, then we have for $i \in [1, 10] \setminus \{5, 10\}$:

$$c_i[x] = \frac{2x}{N} \cdot c_{i+1}[x] + (1 - \frac{2x}{N}) \cdot (\frac{N}{2} - x) \cdot c_{i+1}[x + 1],$$

and for $i = 5, 10$:

$$c_i[x] = (\frac{2x}{N})^2 \cdot c_{i+1}[x] + (1 - \frac{2x}{N})^2 \cdot c_{i+1}[x + 1],$$

where c_{11} denotes c_1. In the above equation, when $i \in [1, 10] \setminus \{5, 10\}$, it denotes an assignment phase, when $i = 5, 10$, it denotes a verification phase. During an assignment, if x elements are already present in the permutation, then with probability $\frac{x}{N/2}$, the index to be assigned would be already filled, and in this case the algorithm would move on to stage $i + 1$ without assignment. Alternatively with probability $1 - \frac{x}{N/2}$, the index is empty and there are exactly $\frac{N}{2} - x$ ways to assign it, after which it moves to stage $i + 1$. During verification stage the analysis is as follows:

a. With probability $\frac{x}{N/2}$, the verification index d is already filled.

b. Therefore with probability $\frac{x}{N/2} \cdot (1 - \frac{x}{N/2})$, the index is already filled by a value other than Z_r. In this case the path is terminated.

c. With probability $(\frac{x}{N/2})^2$ the index is filled with Z_r and the algorithm moves to the next phase.

d. With probability $(1 - \frac{x}{N/2})$ the verification index d is empty.

e. Therefore with probability $(1 - \frac{x}{N/2}) \cdot (\frac{x}{N/2})$ it happens that Z_r exists in some other index of the permutation. In this case too the path is terminated.

f. With probability $(1 - \frac{x}{N/2})^2$, Z_r is not present in the permutation, and so after assigning $S[d] \leftarrow Z_r$ it moves to the next stage.

The complexity $\mathcal{T}_1$ can be estimated as $c_1[0]$, with the boundary conditions $c_i[\frac{N}{2} - 1] = 1$. The above recurrence can be solved by a dynamic programming approach to find an estimate for $c_1[0]$. A similar recurrence relation can be deduced for estimating $\mathcal{T}_2$ by keeping track of the even valued assignment/verification indices. We write the recurrence relation below for the benefit of the reader. For $i \in [1, 14] \setminus \{6, 10\}$:

$$c_i[x] = \frac{x}{N/2} \cdot c_{i+1}[x] + (1 - \frac{x}{N/2}) \cdot (\frac{N}{2} - x) \cdot c_{i+1}[x + 1],$$

and for $i = 6, 10$:

$$c_i[x] = (\frac{x}{N/2})^2 \cdot c_{i+1}[x] + (1 - \frac{x}{N/2})^2 \cdot c_{i+1}[x + 1],$$

where c_{15} denotes c_1.

3.3.1 Experimental Results

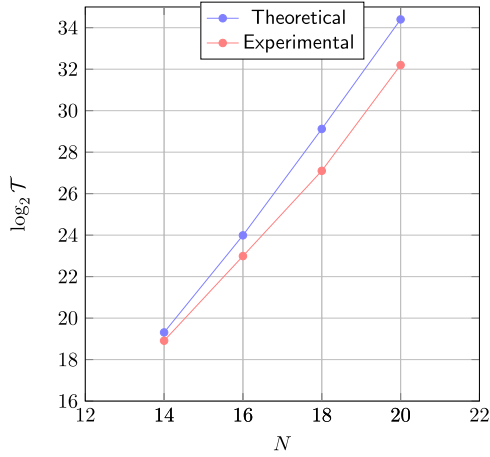We performed the state recovery for $N = 14, 16, 18, 20$ for

Fig. 4 Experimental and theoretical estimates of $\log_2 \mathcal{T}$.

100 random permutations. The algorithm was always able to recover the permutation. In Fig. 4, we plot the base 2 logarithm of the theoretical estimate $\mathcal{T}$ with the base 2 logarithm of the experimentally obtained average number of steps, for different even values of N. We can see that the theoretical value always overestimates the experimentally obtained complexity. For $N = 256$, the theoretical estimate for $\mathcal{T} \approx 2^{1233}$. And so the estimated complexity of state recovery is given as $\mathcal{T} \cdot \left(\frac{N}{2}\right)^2 \approx 2^{1247}$ (taking into account the additional complexity of guessing the values of j, k at the beginning of the SQUEEZE phase). So the total complexity consists of ρ^{-1} encryptions plus $\mathcal{T} \cdot \left(\frac{N}{2}\right)^2$ assignments which again comes to approximately 2^{1247}.

4. Countermeasures

Since the first two output bytes are biased to the publicly known value of $-w$, one might think that the problem would be solved by not making the value of w dependent only on the length of the Secret Key. This could be achieved if the ABSORBBYTE phase were tweaked to make the value w dependent on the Key. Such a move would be risky on two counts

- The bias in the distribution of (Z_1, Z_2) would still exist but towards some value of $-w$ which is not publicly known.
- The attacker can find the value of w by observing a peak in the distribution of (Z_1, Z_2) and thereby reveal some non trivial information about the Secret Key.

When we analyze the proofs for the distinguishers in Theorem 1 we observe that the events **I, II, III** that lead to the bias have one characteristic trait in common. In all the three events subsequent updated values of the indices i, j, k get stuck at values which are either positive or negative multiples of w. So one possible solution towards solving this problem could be involving some other location of the array that is guaranteed not to be a multiple of w. The value of i
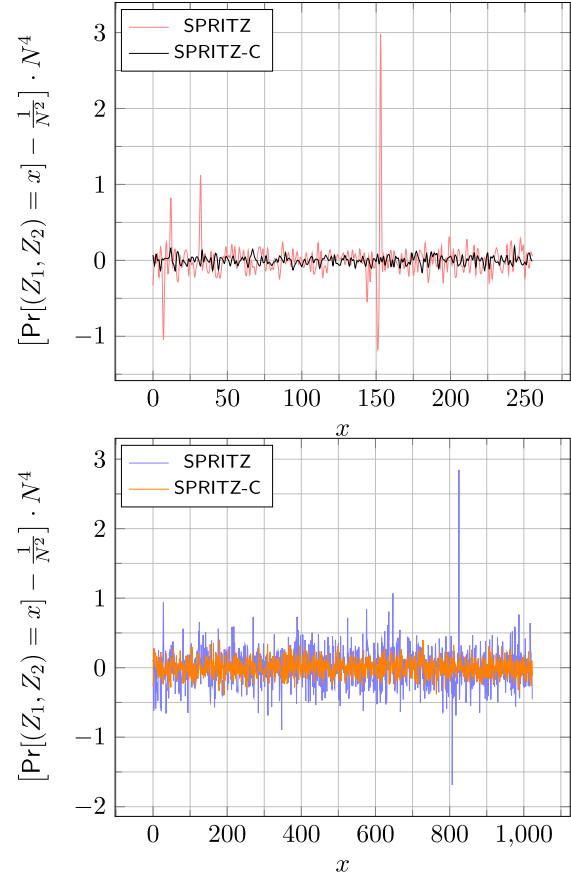

Fig. 5 Plot of $\left[\Pr[(Z_1, Z_2) = x] - \frac{1}{N^2}\right] \cdot N^4$ after applying countermeasure (for $N = 16, 32$).

in the first two rounds of the of the DRIP phase is $w, 2w$, and so one could utilize the array index $N - 1 - i$ (which can be calculated in most CPUs as the bitwise complement of i) to compute the final keystream bit. So we propose to tweak the output function to

$$z = S[j + S[i + S[z + k]]] \oplus S[N - 1 - i]$$

Let us denote the modified cipher by SPRITZ-C. When the output function is so tweaked we observe that the bias in the first two bytes disappears as seen in Fig. 5.

4.1 Implementation in `x86_64` CPU

Implementing the above tweak in any CPU supporting the `x86_64` instruction set is surprisingly simple and requires only three additional assembly level instructions.

```
notb    %bl
movzbl  (%rdi,%rbx), %ebx
xorb    %bl,%al
```

In the above code snippet the register `%rbx, %rax, %rdi` hold the indices i, z and the memory address of the first element of the S array respectively. The instruction `notb %bl`

Table 3 Tabulation of software speeds (CPB stands for cycles per byte, all speeds in MBps).

#	Cipher	PROFILE A		PROFILE B	
		Speed	CPB	Speed	CPB
1	RC4	605.8	3.93	792.3	3.01
2	Public SPRITZ Implementation [1]	36.7	64.97	12.5	190.73
3	Optimized SPRITZ	139.7	17.07	66.8	35.69
4	SPRITZ-C	137.8	17.30	66.7	35.74

performs the bitwise complement of the least significant 8 bits of `%rbx`. Then the instruction `movzbl (%rdi,%rbx), %ebx` moves the value $S[N-1-i]$ from the memory to `%rbx` and clears its MSBs. Finally `xorb %bl,%al` does an 8-bit xor between the least significant bytes of `%rbx,%rax` (i.e. z and $S[N-1-i]$) and stores the result in `%rax`.

Due this small overhead that requires only one additional memory access, the loss of software speed while implementing this countermeasure is not significant. To do a fair comparison, we obtained a publicly available implementation of Spritz available at [1]. We then went ahead and implemented an optimized version of SPRITZ using the `x86_64` instruction set. Finally we implemented SPRITZ-C to investigate the amount by which we loose software speed. The simulations were carried out on an Intel Core i7-6500U CPU, running at 2.5 GHz on Ubuntu v14.04. For each of the ciphers we investigate the software speeds under two test cases:

PROFILE A: This covers the events when a large number of data packets are being encrypted. More specifically, we generated around 16 KB of keystream for several randomly chosen Keys and measured the average keystream generation rates.

PROFILE B: This covers the events when short data packets are being encrypted. This essentially measures the impact of the Key setup on encryption speed. We used several randomly chosen 16 byte Keys and encrypted messages of length 512 bytes.

Table 3 tabulates the performance results for our implementations and compares the results with an optimized RC4 implementation. As expected, RC4 outperforms all implementations of SPRITZ in terms of speed. Our optimized implementation of SPRITZ is only around 4 times slower than RC4 in PROFILE A. Also the addition of the countermeasure, decreases the software speed in PROFILE A by only around 2 MBps which is around 1.5% decrease with respect to the original SPRITZ cipher, whereas there does not appear to be any significant loss in speed in PROFILE B.

5. Conclusion

In this paper, we analyzed the security of the stream cipher Spritz. We first proposed distinguishing attacks based on the short-term and the long-term biases in the keystream of Spritz. The distinguisher can be used both for distinguishing keystreams produced by multiple key-IVs and for distinguishing a keystream produced by a single key-IV pair. In the second half of the paper we looked at the state recovery attack on Spritz (in the multiple IV setting), in the situation when the cipher has entered a special class of SPECIAL states. We calculated the probability of such an event happening, and went on to outline an algorithm to recover the internal permutation. Our estimates suggest that in this case we need approximately 2^{1247} assignments to recover the internal state which is an improvement on the 2^{1400} step algorithm proposed in [3]. Finally we propose a simple countermeasure that removes the bias in the keystream bytes produced by SPRITZ, which uses only one additional memory access and thus does not decrease the software speed significantly.

References

[1] Araduino Spritz Cipher Library. Available at https://hackaday.io/project/8244-arduino-spritz-cipher-library

[2] N.J. AlFardan, D.J. Bernstein, K.G. Paterson, B. Poettering, and J.C. N. Schuldt, "On the security of RC4 in TLS and WPA," Proc. 22nd USENIX Conference on Security, pp.305–320, 2013.

[3] R. Ankele, S. Kölbl, and C. Rechberger, "State-recovery analysis of Spritz," LatinCrypt 2015, LNCS, vol.9230, pp.204–221, 2015.

[4] S. Banik and T. Isobe, "Cryptanalysis of the full Spritz stream cipher," Fast Software Encryption, LNCS, vol.9783, pp.63–77, 2016.

[5] S. Banik, S. Sarkar, and R. Kacker, "Security analysis of RC4+ stream cipher," INDOCRYPT 2013, LNCS, vol.8250, pp.297–307, 2013.

[6] S. Banik and S. Jha, "How not to combine RC4 states," SPACE 2015, LNCS, vol.9354, pp.95–112, 2015.

[7] S. Banik and S. Jha, "Some security results of the RC4+ stream cipher," Security and Communication Networks, vol.8, no.18, pp.4061–4072, 2015.

[8] E. Biham, L. Granboulan, and P.Q. Nguyen, "Impossible fault analysis of RC4 and differential fault analysis of RC4," FSE 2005, LNCS, vol.3557, pp.359–367, 2005.

[9] H. Finney, An RC4 Cycle That Can't Happen, Posting to sci.crypt, Sept. 1994.

[10] C. Garman, K.G. Paterson, and T. van der Merwe," "Attacks only get better: Password recovery attacks against RC4 in TLS," Proc. 24th USENIX Conference on Security, pp.113–128, 2015.

[11] G. Gong, K.C. Gupta, M. Hell, and Y. Nawaz, "Towards a general RC4-like keystream generator," CISC 2005, LNCS, vol.3822, pp.162–174, 2005.

[12] T. Isobe, T. Ohigashi, Y. Watanabe, and M. Morii, "Full plaintext recovery attack on broadcast RC4," FSE 2013, LNCS, vol.8424, pp.179–202, 2014.

[13] S. Maitra and G. Paul, "Analysis of RC4 and proposal of additional layers for better security margin," INDOCRYPT 2008, LNCS, vol.5365, pp.27–39, 2008.

[14] I. Mantin and A. Shamir, "A practical attack on broadcast RC4," FSE 2001, LNCS, vol.2355, pp.152–164, 2001.

[15] A. Maximov, "Two linear distinguishing attacks on VMPC and RC4A and weakness of RC4 family of stream ciphers," FSE 2005, LNCS, vol.3557, pp.342–358, 2005.

[16] A. Maximov and D. Khovratovich, "New state recovery attack on RC4," CRYPTO 2008, LNCS, vol.5157, pp.297–316, 2008.

[17] Y. Nawaz, K.C. Gupta, and Guang Gong, "A 32-bit RC4-like keystream generator," IACR Cryptology ePrint Archive 2005/175, 2005.

[18] T. Ohigashi, T. Isobe, Y. Watanabe, and M. Morii, "How to recover any byte of plaintext on RC4," Selected Areas in Cryptography 2013, LNCS, vol.8282, pp.155–173, 2013.

[19] G. Paul, S. Maitra, and A. Chattopadhyay, "Quad-RC4: Merging

four RC4 states towards a 32-bit stream cipher," IACR Cryptology ePrint Archive 2013/572, 2013.

[20] S. Paul and B. Preneel, "A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher," FSE 2004, LNCS, vol.3017, pp.245–259, 2004.

[21] S. Paul and B. Preneel, "On the (In)security of stream ciphers based on arrays and modular addition," ASIACRYPT 2006, LNCS, vol.4284, pp.69–83, 2006.

[22] R. Rivest and J. Schuldt, "Spritz - a spongy RC4-like stream cipher and hash function," Available at https://people.csail.mit.edu/rivest/pubs/RS14.pdf

[23] Y. Tsunoo, T. Saito, H. Kubo, M. Shigeri, T. Suzaki, and T. Kawabata, The Most Efficient Distinguishing Attack on VMPC and RC4A, In SKEW 2005. Available at http://www.ecrypt.eu.org/stream/papers.html

[24] Y. Tsunoo, T. Saito, H. Kubo, and T. Suzaki, "A distinguishing attack on a fast software-implemented RC4-like stream cipher," IEEE Trans. Inf. Theory, vol.53, no.9, pp.3250–3255, 2007.

[25] M. Vanhoef and F. Piessens, "All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS," Proc. 24th USENIX Conference on Security, pp.97–112, 2015.

[26] B. Zoltak, "Statistical weakness in Spritz against VMPC-R: in search for the RC4 replacement," Available at http://eprint.iacr.org/2014/985.pdf

[27] B. Zoltak, "VMPC one-way function and stream cipher," FSE 2004, LNCS, vol.3017, pp.210–225, 2004.

Masakatu Morii received the B.E. degree in electrical engineering and the M.E. degree in electronics engineering from Saga University, Saga, Japan, and the D.E. degree in communication engineering from Osaka University, Osaka, Japan, in 1983, 1985, and 1989, respectively. From 1989 to 1990 he was an Instructor in the Department of Electronics and Information Science, Kyoto Institute of Technology, Japan. From 1990 to 1995 he was an Associate Professor at the Department of Computer Science, Faculty of Engineering at Ehime University, Japan. From 1995 to 2005 he was a Professor at the Department of Intelligent Systems and Information Science, Faculty of Engineering at the University of Tokushima, Japan. Since 2005, he has been a Professor at the Department of Electrical and Electronics Engineering, Faculty of Engineering at Kobe University, Japan. His research interests are in error correcting codes, cryptography, discrete mathematics and computer networks and information security. He is a member of the IEEE.

Subhadeep Banik recieved B.Tech in Electronics and Electrical Communications Engineering and M. Tech in Automation and Computer Vision from the Indian Institute of Technology Kharagpur in 2005. He recieved Ph.D. in Compter Science from the Indian Statistical Institute, Kolkata in 2014. He is currently a research scientist in Nanyang Technological University, Singapore. His current research interests include cryptography and digital design.

Takanori Isobe received the B.E., M.E., and Ph.D. degrees from Kobe University, Japan, in 2006, 2008, and 2013, respectively. He joined the Sony Corporation in 2008. His current research interests include information security and cryptography. He received the SCIS Paper Award and SCIS 2013 Innovation Paper Award from ISEC group of IEICE in 2008 and 2014, respectively. He also received the Best Paper Award from IEICE in 2015. He received the FSE 2011 Best Paper Award from IACR in 2011.