State recovery of RC4 and Spritz Revisited

Martin Gábriš and Martin Stanek

Department of Computer Science
Comenius University
martin.gabris22@gmail.com, stanek@dcs.fmph.uniba.sk

Abstract

We provide an improved complexity analysis of backtracking-based state recovery attacks on RC4 and Spritz. Comparing new estimates with known results on Spritz, our analysis shows a significantly lower complexity estimate for simple state recovery attack as well as special state recovery attack. We validated the estimates by performing experiments for selected feasible parameters.

We also propose a prefix check optimization for simple state recovery attack on Spritz. We believe that the simple state recovery attack with this optimization and so-called "change order" optimization inspired by Knudsen et al. attack on RC4 constitutes currently the best state recovery attack on Spritz (when no special state is observed).

Keywords: RC4, Spritz, cryptanalysis, state recovery, complexity.

1 Introduction

RC4 is a well-known stream cipher with a simple design and implementation. It was widely used in various cryptographic schemes and protocols. Multiple cryptographic weaknesses were found in the design and the use of RC4, such as [1, 4, 7, 9, 10]. Hence it is advised to avoid RC4 in practice, for example there is even a RFC proposal to prohibit RC4 in TLS [11].

A stream cipher Spritz was proposed by Rivest and Schuldt as a replacement of RC4 [12]. Spritz retains some components of RC4, such as using a permutation as the main part of its internal state, while carefully modifying other parts in order to make the cipher resistant to known cryptanalytic attacks.

State recovery attacks are a class of cryptanalytic attacks that use known part of a keystream to find the internal state of the cipher. We analyze the complexity of some state recovery attacks on RC4 and Spritz. Our contribution can be summarized as follows:

- We provide an improved complexity estimate for state recovery attack on RC4 from [6]. The difference between the original and our analysis is rather small, e.g. 2^{779} vs. 2^{772} for permutation of length $N = 256$. However, our analysis justifies and quantifies the improvement obtained by using "change order" optimization proposed in [6].
- We provide improved complexity estimates for state recovery attack on Spritz from [2], and for "special-state" recovery attack from [3]. In these cases, the improved estimates are significantly better than the original estimates. For example, when $N = 256$, we get the estimate 2^{1122} instead of 2^{1575} for the simple state recovery attack and 2^{927} instead of 2^{1235} for special state recovery attack, in all cases without counting the expected complexity of guessing necessary internal state variables (additional factor $N^2 = 2^{16}$ for simple state recovery or $(N/2)^2 = 2^{14}$ for special state recovery).

– We propose a prefix check optimization for state recovery attack on Spritz. We believe that basic state recovery attack with a change order optimization inspired by [6] and the prefix check optimization constitutes currently the best state recovery attack on Spritz (when no special state is observed). The complexity of the attack with guessing necessary internal state variables, and without the prefix check optimization can be estimated $\approx 2^{1137}$.

We introduce the notation and state update functions of RC4 and Spritz in Section 2. In this section we also present the idea of basic state recovery attacks using backtracking. Section 3 introduces our approach to estimating the complexity of backtracking-based state recovery attacks and provides an analysis of state recovery attack by Knudsen et al. [6]. Section 4 focuses on Spritz. We show the improved analysis of basic state recovery for Spritz, proposed by Ankele et al. [2]. We also describe the change order optimization, and the prefix check optimization for basic state recovery attack. Finally, we revise the complexity estimate for recovering the special state of Spritz by Banik and Isobe [3].

2 Preliminaries

2.1 RC4 and Spritz – internal states and update functions

We describe the internal states of RC4 and Spritz, their update functions and how the output is produced. Both ciphers use a permutation S over the set $\mathbb{Z}_N = \{0, 1, \dots, N-1\}$, where the parameter N controls the size of the internal state. The most typical value of N is 256. RC4 uses two additional variables: $i, j \in \mathbb{Z}_N$. Spritz uses six additional variables: $i, j, k, a, w, z \in \mathbb{Z}_N$, although the variable a is not used in the update function.

Figure 1 shows the update functions. Let us note that all additions are computed modulo N. Since our work is focused solely on state recovery attacks, we do not present unrelated functions, such as key-scheduling for S initialization etc.

```
initialization of $i$, $j$:
$i \leftarrow 0$
$j \leftarrow 0$

update:
1.  $i \leftarrow i + 1$
2.  $j \leftarrow j + S[i]$
3.  swap($S[i]$,$S[j]$)
4.  $z \leftarrow S[S[i] + S[j]]$
5.  output $z$
```

RC4

```
initialization of $i$, $j$, $k$, $w$, $z$:
$i, w$ – dependent on the length of the key
$j, k$ – dependent on the key
$z \leftarrow 0$

update:
1.  $i \leftarrow i + w$
2.  $j \leftarrow k + S[j + S[i]]$
3.  $k \leftarrow i + k + S[j]$
4.  swap($S[i]$,$S[j]$)
5.  $z \leftarrow S[j + S[i + S[z + k]]]$
6.  output $z$
```

Spritz

Figure 1: State update functions of RC4 and Spritz

The initialization of internal variables in RC4 is very simple – both are set to 0. The situation is different for Spritz. The key scheduling makes j and k dependent on key. Variables i and w depend only on the key length after initialization. Moreover, w does not change in the update function and i is updated only by adding w, therefore we can assume they are always known to an attacker. Similarly, the attacker knows z since it is initialized to 0 and after that z is equal to a keystream value (the state recovery attacks assume the knowledge of the keystream). We denote the known keystream as a sequence $Z = \{z_t\}_{t \geq 0}$ in the description of the attacks.

2.2 State recovery using backtracking

The basic idea of the state recovery attacks is to use backtracking. First, we guess internal state variables that are unknown. Starting with an unknown permutation S we proceed through the update function. Each time we need some value $S[x]$ that is unknown we make a guess – we try all possible assignments to $S[x]$ from values not already used in S (since S is a permutation). Moreover, we use the known keystream to check for contradictions – the update function must produce the correct output values. Another contradictions are detected using the fact that S cannot contain any value twice. Using the contradictions we cut some branches in the backtracking early. Therefore, we expect and indeed obtain lower complexity of this state search algorithm than trying all possible permutations (i.e. a brute-force approach).

Various optimizations of the basic state recovery are possible, depending on details and properties of particular update function. Some of them are discussed in the following sections.

The complexity of the state recovery attacks is calculated by counting the expected number of assignments (guesses) that are made in the backtracking. Usually, we introduce variables that count the expected number of assignments starting in a particular place of the algorithm. We form equations relating the variables and reflecting the logic of the algorithm. The complexity of the attack is calculated by solving these equations.

Remark. Let us note that the complexity estimates calculate the entire backtracking, from the first guess to the last assignment, all paths that do not lead to a contradiction are explored. There is no way to detect the correct state and "interrupt" complexity equations. In practice we stop the backtracking when the correct state (permutation) is found. Our experiments show that this reduces the complexity by half (on average).

Remark. Recall that the real attack must guess other variables as well. This is not the case for RC4, assuming that we do a state recovery of the initial state (where $i = j = 0$). For Spritz we must guess the variables j and k.

3 RC4

We start with an analysis of the basic state recovery attack on RC4 as outlined in Section 2.2. Let us stress that our aim is not to show the most efficient state recovery attack on RC4 but to explain how the complexity of an attack is estimated. Similar approach will be used in our analysis of Spritz in Section 4. For more efficient state recovery attacks on RC4, see for example [5, 8].

3.1 Basic state recovery – simple backtracking

The backtracking follows iterations of the update function and known keystream. Starting with completely undefined S it gradually tries to fill those values that are needed to proceed with the update function. The guesses are made, exploring all possibilities while detecting and avoiding contradictions, i.e. no value from $\mathbb{Z}_N$ can appear twice in S, and the output should be consistent with the keystream. In each iteration of the update function the backtracking needs the values $S[i]$, $S[j]$, and $S[S[i] + S[j]]$ (in this order). Similarly to [6] we use variables $c_1(x)$, $c_2(x)$, and $c_3(x)$, for $x \in \mathbb{Z}_N$, denoting the expected number of assignments performed by the backtracking when S contains exactly x known (not necessary correct) values and the algorithm starts:

$c_1(x)$	at the beginning of the backtracking, i.e. before $S[i]$ is needed;
$c_2(x)$	just before the second value, $S[j]$, is needed;
$c_3(x)$	just before the third value, $S[S[i] + S[j]]$, is needed.

The overall expected complexity of the algorithm is $c_1(0)$. Trivial boundary conditions are values for $x = N$, when the permutation S is complete, and no assignment is needed, hence $c_1(N) = c_2(N) = c_3(N) = 0$. If $c_1(x + 1)$, $c_2(x + 1)$, and $c_3(x + 1)$ are known, we can compute $c_1(x)$, $c_2(x)$, and $c_3(x)$ using the following set of linear equations:

$$c_1(x) = (x/N) \cdot c_2(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_2(x + 1))$$
$$c_2(x) = (x/N) \cdot c_3(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_3(x + 1))$$
$$c_3(x) = (x/N) \cdot (1/N \cdot c_1(x)) + (1 - x/N)^2 \cdot (1 + c_1(x + 1))$$

Let us explain the equations. Counting assignments in $c_1(x)$ can be split into two cases – either $S[i]$ is known (this happens with probability x/N) or unknown (with probability $1 - x/N$). In the first case no assignment is needed and we can proceed further, thus counting $c_2(x)$ with the same number of known values in S. In the second case we try all $N - x$ unused values as $S[i]$, i.e. for each value we perform one assignment and proceed further with increased number of known values $(1 + c_2(x + 1))$. The equation for $c_2(x)$ can be explained similarly.

Counting assignments in $c_3(x)$ can be split again into two cases – either $S[S[i]+S[j]]$ is known (probability x/N) or unknown (probability $1-x/N$). In the first case we proceed with next iteration of the update function (hence counting $c_1(x)$ assignments) if and only if the $S[S[i] + S[j]]$ is equal to the current keystream value z_t (probability $1/N$). A contradiction happens if $S[S[i] + S[j]]$ is not equal to z_t – then no further assignments are made, i.e. this situation does not contribute to the equation. In the second case, $S[S[i] + S[j]]$ is unknown, we perform one assignment $S[S[i] + S[j]] \leftarrow z_t$ and proceed with next iteration of the update function (hence counting $c_1(x + 1)$ assignments). However, this happens if and only if z_t is not already in S at some other place (probability $1 - x/N$). Otherwise we found a contradiction.

Solving the sets of equations in sequence, from $x = N - 1$ down to $x = 0$, yields the expected complexity of this simple backtracking $c_1(0) \approx 2^{775}$ for $N = 256$. Results for some other values of N are presented in Table 1.

Remark. Estimating the probability of known value for some $S[\cdot]$ as x/N assumes random distribution of index values or random distribution of known values in S. This approach is used for complexity estimates in [6] for RC4, and for Spritz in [2, 3]. However, the experiments show that the probabilities are skewed. On the other hand, the complexity estimates derived this way are very close to the experimental results. Therefore, we use this probability estimate for the rest of the paper.

3.2 Change order optimization

The optimization to basic state recovery proposed by Knudsen et al. [6] changes the order in which different values in the backtracking are used. Observe that if z_t is already somewhere in S and $S[i]$ is known, then $S[j]$ can be computed directly: $S[k] = z_t \Rightarrow S[j] = k - S[i]$. Let us describe how the "guessing steps" of the backtracking work, and simultaneously derive corresponding complexity equations:

1. Start with $S[i]$: if known (probability x/N when x values in S are already known) proceed further, otherwise (probability $1 - x/N$) assign a value to $S[i]$ and proceed further for all $N - x$ values.
   This step is the same as in the simple backtracking, hence the equation for $c_1(x)$ is the same as well:
   $$c_1(x) = (x/N) \cdot c_2(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_2(x + 1))$$

2. If z_t is in S (probability x/N), let $z_t = S[k]$, for some $k \in \mathbb{Z}_N$. Let us denote $l = k - S[i]$, and look at $S[j]$:

(1)	$S[j]$ is unknown & l is not in S	assign $S[j] \leftarrow l$ and start next iteration;
(2)	$S[j]$ is unknown & l is in S	contradiction;
(3)	$S[j]$ is known & $S[j] = l$	start next iteration of the update function;
(4)	$S[j]$ is known & $S[j] \neq l$	contradiction.

   Otherwise, z_t is not in S (probability $1 - x/N$), proceed further.
   We obtain the following complexity equation, annotated for clarity:

$$c_2(x) = \underbrace{(x/N)}_{z_t \in S} \cdot \Big(\underbrace{(1 - x/N)^2 \cdot (1 + c_1(x+1))}_{(1)} + \underbrace{(x/N) \cdot (1/N) \cdot c_1(x)}_{(3)} \Big) + \underbrace{(1 - x/N)}_{z_t \notin S} \cdot c_3(x)$$

3. In this step we know that z_t is not in S. We have two cases: either $S[j]$ is unknown (probability $1 - x/N$) or known (probability x/N). In the first case we try all $N - x$ possible values for $S[j]$. We have two separate sub-cases, according to what value is assigned: (1) z_t, and (2) other values:

   (1) if $S[S[i] + S[j]] = z_t$ (probability $1/N$): start next iteration
       else: contradiction
   (2) if $S[S[i] + S[j]]$ is unknown (probability $1 - (x+1)/N$):
           assign $S[S[i] + S[j]] \leftarrow z_t$ and start next iteration
       else: contradiction

   In the second case, when $S[j]$ is known, we check that the position to place z_t, i.e. $S[S[i] + S[j]]$ is not occupied by some other value (probability $1 - x/N$). If the position is free, we assign the value z_t and proceed to the next iteration of the update function.
   Annotated complexity equation:

$$\begin{aligned} c_3(x) = {} & \underbrace{(1 - x/N)}_{S[j] \text{ unknown}} \cdot \Big(\underbrace{1 + (1/N) \cdot c_1(x+1)}_{(1)} + \underbrace{(N - x - 1) \cdot (1 - (x+1)/N) \cdot (1 + c_1(x+2))}_{(2)} \Big) \\ & + \underbrace{(x/N)}_{S[j] \text{ known}} \cdot \underbrace{(1 - x/N)}_{S[S[i]+S[j]] \text{ unknown}} \cdot (1 + c_1(x+1)) \end{aligned}$$

The boundary conditions are trivial: $c_1(x) = c_2(x) = c_3(x) = 0$, for $x \geq N$. Using this change in order of guessed variables we can save few assignments in the backtracking. Indeed, calculating the complexity by solving above equations yields $c_1(0) \approx 2^{772}$ for $N = 256$. Table 1 shows results for some other values of N.

The complexity equations provided in [6], also calculating the expected number of assignments, are different from ours. Unfortunately, they yield unsatisfactory results:

– Complexity 2^{779} for $N = 256$ is worse than the complexity of simple backtracking presented in the previous section (similarly for $N = 128$ etc.). Hence the analysis from [6] does not justify any improvement by using the change order optimization for realistic N values.

– Let $N = 4$ (similar argument works for other values, e.g. $N = 8$ or $N = 16$). Let us compute the expected number of assignments when the algorithm starts with S with only a single unknown value, i.e. 3 out of 4 values in S are already assigned (randomly). Using equations from [6] we get 0.136. On the other hand, the backtracking starts with $S[i]$, i.e. we make 0.25 assignments right in the first step. Hence the expected number of assignments is at least 0.25 (note that there might be more assignment to add when $S[i]$ is known and we proceed further). In contrast, our equations yield $c_1(3) \approx 0.528$ for simple backtracking and $c_1(3) \approx 0.410$ for change order optimization.

	Theory			Experiments	
N	Simple	Change order	[6]	Simple	Change order
8	9.47	8.56	8.65	9.69	9.32
12	15.38	14.28	14.56	15.72	15.19
16	22.02	20.77	21.25	22.53	21.90
64	132.28	130.33	132.65	–	–
128	322.85	320.55	324.76	–	–
256	775.03	772.39	779.68	–	–

Note 1: The numbers represent “lg” values of the complexity.

Note 2: Last two columns show average value from 1000 experiments.

Table 1: RC4 – complexity of selected state recovery algorithms

Our analyses show that the change order optimization really speed-up the backtracking. The speed-up factor increases from approximately 2 for small N to approximately 5 for $N = 128$ or 6 for $N = 256$. Also, there is a rather small difference between our results and results obtained using equations from [6]. Let us remind that much better state recovery attacks exist, for example [5, 8].

Remark. In order to directly compare experimental results to theoretical estimates, we performed the experiments as follows: the backtracking starts with empty permutation, it does not stop after the correct state is found and continues counting guesses until every non-contradicting assignment is tried. Our experiments show that stopping immediately when the correct state is recovered reduces (on average) the complexity by half (i.e. the “lg” value decreases approximately by 1).

Improvement of change order optimization can also be observed in experimental data, see Table 1. Differences between theoretical estimates and corresponding experimental values are caused by the assumption of random distribution of index values in complexity equations, as noted in Section 3.1.

4 Spritz

4.1 Basic state recovery – simple backtracking

Basic state recovery attack on Spritz is similar to attack on RC4 described in Section 3.1. The main difference is more positions in S that must be known or guessed in order to proceed with the update function.

First, we assume the knowledge of all internal variables of the state (guessed or calculated, see Section 2) – the backtracking recovers the permutation S, starting with empty S. In each iteration we need the values $S[i]$, $S[j+S[i]]$, $S[j]$, $S[z+k]$, $S[i+S[z+k]]$ and $S[j+S[i+S[z+k]]]$ (in this order). The contradictions are detected and prevented as usual – by assigning values not already in S, and by checking with the keystream values. We introduce variables $c_1(x), \dots, c_6(x)$, for $x \in \mathbb{Z}_N$, denoting the expected number of assignments performed by the backtracking when S contains exactly x known (not necessary correct) values and the algorithm starts: at the beginning of the update function ($c_1(x)$), or just before second ($c_2(x)$), third ($c_3(x)$), …, and sixth ($c_6(x)$) value from S is needed.

The overall expected complexity of the algorithm is $c_1(0)$. Boundary conditions are again $c_i(N) = 0$, for $i = 1, \dots, 6$. The set of equations that allow to find values $c_1(x), \dots, c_6(x)$ if $c_1(x+1)$,

$\dots, c_6(x+1)$ are known can be constructed analogously to the simple backtracking of RC4:

$$c_i(x) = (x/N) \cdot c_{i+1}(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_{i+1}(x+1)), \quad \text{for } i = 1, \dots, 5$$
$$c_6(x) = (x/N) \cdot (1/N \cdot c_1(x)) + (1 - x/N)^2 \cdot (1 + c_1(x+1))$$

The expected complexity of the simple backtracking to recover the state of Spritz is obtained by solving the sets of equations in sequence, for $x = N-1, \dots, 0$. Result for $N = 256$ is $c_1(0) \approx 2^{1122}$, some other results are presented in Table 2.

Simple backtracking was described and analyzed in [2]. However, the analysis exhibits the following problems:

– The complexity presented in the paper is inconsistent with the complexity equations. For example, solving equations from [2] for $N = 256$ yield complexity $\approx 2^{1575}$, while the complexity presented there is 2^{1400}.

– Similarly to the problem identified in Section 3.2, we can deduce that the complexity equations from [2] are incorrect. For example, set $N = 4$ and $x = 3$ (again, similar argument works for other values N). Using equations from [2] we get 0.1083. On the other hand, assuming random state S with 3 known values, we have $c_1(3) \geq 0.25$ since the backtracking makes expected 0.25 assignments in the first step by assigning $S[i]$. In contrast, our equations yield $c_1(3) \approx 0.81$.

4.2 Improvements for simple backtracking

In this section we present two improvements to the simple backtracking: change order of the last two steps in the backtracking, similar to one described in Section 3.2 for RC4, and additional verification (contradiction detection) with the prefix of keystream.

4.2.1 Change order optimization

We can apply the optimization described in Section 3.2 to Spritz. Instead of guessing (if needed) values $S[i]$, $S[j + S[i]]$, $S[j]$, $S[z + k]$, $S[i + S[z + k]]$, in this order, and then verifying consistency with the keystream value, we can do the following: we check whether z_t is in S before guessing $S[i + S[z + k]]$. If $z_t = S[m]$, for some $m \in \mathbb{Z}_N$, then $S[i + S[z + k]]$ must be $m - j$. We check for contradictions and proceed to the next iteration of the update function. If z_t is not in S, we guess $S[i + S[z + k]]$, verify consistency, assign z_t to the right place in S, and proceed to the next iteration.

We derive our complexity equations that reflect the backtracking steps with outlined optimization:

1. We start with checking first four (potentially) unknown values $S[i]$, $S[j + S[i]]$, $S[j]$, $S[z + k]$ and guessing them if needed. If the particular value is known (probability x/N as x values are already assigned in S), proceed further. Otherwise (probability $1 - x/N$) assign one by one all $N - x$ suitable values and proceed further to the next step.

   These four steps are the same as in the simple backtracking:

$$c_i(x) = (x/N) \cdot c_{i+1}(x) + (1 - x/N) \cdot (N - x) \cdot (1 + c_{i+1}(x+1)), \quad \text{for } i = 1, \dots, 4.$$

2. If z_t is in S (probability x/N), let $z_t = S[m]$, for some $m \in \mathbb{Z}_N$. Let us denote $l = m - j$ and $r = S[i + S[z + k]]$, i.e. r can be either unknown or some value from $\mathbb{Z}_N$. Notice that

$S[z+k]$ is already known (e.g. guessed), so the index $i + S[z+k]$ is known. There are four separate cases:

(1)	r is unknown & l is not in S	assign $S[i + S[z+k]] \leftarrow l$ and start next iteration;
(2)	r is unknown & l is in S	contradiction;
(3)	r is known & $r = l$	start next iteration of the update function;
(4)	r is known & $r \neq l$	contradiction.

Otherwise, z_t is not in S (probability $1 - x/N$), and we proceed further.
Putting all cases together we get:

$$c_5(x) = \underbrace{(x/N)}_{z_t \in S} \cdot \Big(\underbrace{(1 - x/N)^2 \cdot (1 + c_1(x+1))}_{(1)} + \underbrace{(x/N) \cdot (1/N) \cdot c_1(x)}_{(3)} \Big) + \underbrace{(1 - x/N)}_{z_t \notin S} \cdot c_6(x)$$

3. In this step we know that z_t is not in S. Again, let us denote $r = S[i + S[z+k]]$. The value r is either unknown (probability $1 - x/N$) or known (probability x/N). In the first case we try all $N - x$ possible values for $S[i + S[z+k]]$. For this case we have two separate sub-cases, according to what value is assigned: (1) z_t, and (2) other values:

   (1) if $S[j + S[i + S[z+k]]] = z_t$ (probability $1/N$): start next iteration
   else: contradiction
   (2) if $S[j + S[i + S[z+k]]]$ is unknown (probability $1 - (x+1)/N$):
   assign z_t there and start next iteration
   else: contradiction

   In the second case, when r is known, we check that the position to place z_t, i.e. $S[j + S[i + S[z+k]]]$ is unknown (probability $1 - x/N$) so we can put z_t there, and proceed to next iteration of the update function.
   Annotated complexity equation:

$$c_6(x) = \underbrace{(1 - x/N)}_{r \text{ unknown}} \cdot \Big(\underbrace{1 + (1/N) \cdot c_1(x+1)}_{(1)} + \underbrace{(N - x - 1) \cdot (1 - (x+1)/N) \cdot (1 + c_1(x+2))}_{(2)} \Big) + \underbrace{(x/N)}_{r \text{ known}} \cdot \underbrace{(1 - x/N)}_{S[j+S[i+S[z+k]]] \text{ unknown}} \cdot (1 + c_1(x+1))$$

Remark. Notice that the equations for $c_5(x)$ and $c_6(x)$ are the same as equations for $c_2(x)$ and $c_3(x)$ in the change order optimization for RC4, see Section 3.2.

The boundary conditions are again $c_i(x) = 0$ for $i \in \{1, \dots, 6\}$, and $x \geq N$. Overall complexity $c_1(0)$ can be computed in similar fashion as the complexity of previous backtracking algorithms – by solving the sets of equations. The change order optimization, as expected, reduces the complexity of the state recovery backtracking. For example, $c_1(0) \approx 2^{1121}$ for $N = 256$, which is marginally better than the complexity of the simple backtracking. Table 2 shows $\lg(c_1(0))$ values for some selected parameters N and x.

4.2.2 Prefix check optimization

The idea of the prefix check optimization is to start state recovery in a shifted position of the known keystream and using the skipped prefix to check for possible contradictions. Let us assume we know keystream $Z = \{z_t\}_{t \geq 0}$ produced from some starting state S_0. All previously described state recovery backtracking algorithms aim to recover S_0. Let S_p be the permutation after p updates, for some $p \geq 0$. The value i is known in any moment, i.e. also after p updates, since i is updated by adding w (which is known and fixed value while iterating the update function). Let us

		Theory			Experiments		
N	x	Simple	[2]	Change order	Simple	Change order	All optim.
8	0	13.02	13.40	12.54	13.05	12.64	11.98
10	0	17.34	19.36	16.87	17.40	16.97	16.13
16	5	16.76	22.49	16.14	16.81	16.23	14.95
64	48	13.65	40.19	12.67	13.59	12.61	9.65
128	114	6.07	32.82	5.13	5.97	5.09	2.88
256	240	4.47	40.24	3.64	4.51	3.62	1.94
64	0	195.81	272.03	194.83	–	–	–
128	0	473.34	664.35	472.18	–	–	–
256	0	1122.34	1575.43	1121.01	–	–	–

Note 1: The numbers represent “lg” values of the complexity.

Note 2: Last three columns show average value from 1000 experiments.

Note 3: The last column shows results for both optimizations (Change order and Prefix check).

Table 2: Spritz – complexity of selected state recovery algorithms

assume we also know, e.g. by guessing, the values j and k after p updates. Notice that any state recovery attack must guess these values even if it starts with S_0.

We start the state recovery backtracking with offset p, aiming to recover S_p. Observe that Spritz's update function can be easily reverted if the state in known. Moreover, we can revert any current state in the backtracking (partially-filled with x known elements) to position p, since all necessary values from S were already assigned. The reverted state will contain exactly the same x known elements, possibly rearranged because of swap operation in the update function. Then we try to revert this state further and check if produced output is z_{p-1}, z_{p-2} etc. If the output is not consistent with the keystream, we know that some assignments in the backtracking are incorrect (contradiction is detected), and we can cut the current branch in the backtracking. Let us note that reverting state from position p may need values in permutation that are still unknown – in that case we cannot proceed with our check and no contradiction is detected, so the backtracking continues as usual.

Remark. We don't have to stop the prefix check when there is a need for an unknown value in computation of z (Figure 1, line 5 of the update function). As this is merely a consistency check and further reverting of state does not depend on this value (correct value of z is known from prefix), we can skip this consistency check and continue with reverting, hoping there will be a contradiction with other prefix value.

Remark. When checking for consistency with prefix, in situation $S[j + S[i + S[z + k]]]$ is unknown and current value z_t from keystream prefix is not in S, we can assign $S[j + S[i + S[z + k]]] \leftarrow z_t$, as this is the only way the current state can be consistent, thus increasing the chance that a contradiction will be found further down in the prefix check.

We do not estimate the complexity of this optimization by equations since our experimental data show only a moderate improvement, in a magnitude similar to the change order optimization.

The experimental results are presented in Table 2. We performed our experiments as close to the situation of complexity estimates as possible, so the backtracking does not stop after correct state is found. Moreover, when starting with preselected elements, positions and values of these elements are chosen at random, as the theoretical estimates assume uniform random positions and values for partially filled permutations. Hence it is highly probable that these instances do not have a solution, because the keystream is independent of partially known starting state. The value p for our experiments was chosen $p = 8$, although most of prefix checks ended within the first three values from the prefix.

Table 2 shows that the prefix check optimization yields better results when more values in S are preselected.

4.3 Special state recovery

Banik and Isobe [3] observed several interesting properties of Spritz regarding a "special" state. A state is called special if the following conditions are satisfied:

1. $S[t] \equiv 0 \pmod 2$, for all odd $t \in \mathbb{Z}_N$, and
2. $S[t] \equiv 1 \pmod 2$, for all even $t \in \mathbb{Z}_N$, and
3. $j \equiv 0 \pmod 2$, and $k \equiv 0 \pmod 2$.

Assuming even N, it can be shown that starting with the special state has following properties:

1. The state after four iterations of the update function is again a special state.
2. The parity of values in S does not change – in every iteration even and odd indexed positions in S will continue to hold odd and even values respectively.
3. The keystream is periodic with the period 4: $z_t \equiv z_{t+4} \pmod 2$, for all t.

The last property allows easy identification of the special state. Although the probability of the special state is small (e.g. approximately $2^{-253.7}$ for $N = 256$), it facilitates a more efficient backtracking if observed. The improved backtracking uses the second property to reduce the number of assignments that are tried in particular step of the update function. The backtracking must consider the parity of indexes used in the update function to access S – these parities depend on the offset with respect to the length of the cycle (period 4) (period 4), see Table 3.

	Offsets			
Index	$v = 0$	$v = 1$	$v = 2$	$v = 3$
$i \leftarrow i + w$ *	1	0	1	0
$j + S[i]$ *	0	0	0	0
$j \leftarrow k + S[j + S[i]]$ *	1	0	1	0
$k \leftarrow k + i + S[j]$	1	0	1	0
$z + k$ *	1	0	0	1
$i + S[z + k]$ *	1	1	0	0
$j + S[i + S[z + k]]$ *	1	0	0	1
$z \leftarrow S[j + S[i + S[z + k]]]$	0	1	1	0

Note 1: Values marked with '*' are used for trying assignments (guessing) in the backtracking algorithm.

Note 2: We start numbering offsets with 0 instead of 1.

Table 3: special state – parities of indexes through 4 iterations [3]

Remark. Let us note that the backtracking with improvements from the previous section is still the best general option for state recovery of Spritz, i.e. when no special state is detected.

The analysis of special state recovery algorithm by Banik and Isobe [3] has several problems:

- The published complexity of the backtracking $\approx 2^{1233}$ for $N = 256$, not counting guesses for state variables j, k, is worse than the complexity $\approx 2^{1122}$ of the simple backtracking from Section 4.1. Hence the result shows no improvement at all.
- Our evaluation of equations stated in [3] yields different results than published. For example, the published complexity of the backtracking (without guessing registers j and k) is $\approx 2^{1233}$ for $N = 256$. On the other hand, the complexity obtained by our evaluation of the equations is $\approx 2^{1235}$ for $N = 256$. Similarly, also some other values from Table 4 are visually different from those presented in [3, Fig. 4].
- The boundary conditions $c_i(N/2 - 1) = 1$, for details see [3], are incorrect – the expected number of assignments with one odd/even value missing is strictly less than one, since there is nonzero probability we do not need the missing value and find a contradiction with the keystream.

Corrected analysis. We define variables $c_u^v(x,y)$, for $u \in \{1,2,\dots,6\}$, $v \in \mathbb{Z}_4$, and $x,y \in \mathbb{Z}_{N/2+1}$. The variable $c_u^v(x,y)$ denotes the expected number of assignments in the following situation:

x	number of values in S known on even indexed positions;
y	number of values in S known on odd indexed positions;
u	starting backtracking before the u-th assignment;
v	the offset in the special state cycle.

Obviously, the complexity of the special state recovery is $c_1^0(0,0)$. Trivial boundary conditions: $c_u^v(N/2,N/2) = 0$, and $c_u^v(x,N/2+1) = c_u^v(N/2+1,y) = 0$ for all $u \in \{1,2,\dots,6\}$, $v \in \mathbb{Z}_4$, and $x,y \in \mathbb{Z}_{N/2+1}$. The equations must reflect parity of the index to S, see Table 3. For brevity, let us denote $N' = N/2$. The equations use similar reasoning as the equations for simple backtracking.

Case $u = 1$ (guessing $S[i]$, parity of the index: $1,0,1,0$):

$$c_1^v(x,y) = y/N' \cdot c_2^v(x,y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_2^v(x,y+1)), \text{ for } v \in \{0,2\}$$
$$c_1^v(x,y) = x/N' \cdot c_2^v(x,y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_2^v(x+1,y)), \text{ for } v \in \{1,3\}$$

Case $u = 2$ (guessing $S[j + S[i]]$, parity of the index: $0,0,0,0$):

$$c_2^v(x,y) = x/N' \cdot c_3^v(x,y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_3^v(x+1,y)), \text{ for } v \in Z_4$$

Case $u = 3$ (guessing $S[j]$, parity of the index: $1,0,1,0$):

$$c_3^v(x,y) = y/N' \cdot c_4^v(x,y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_4^v(x,y+1)), \text{ for } v \in \{0,2\}$$
$$c_3^v(x,y) = x/N' \cdot c_4^v(x,y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_4^v(x+1,y)), \text{ for } v \in \{1,3\}$$

Case $u = 4$ (guessing $S[z + k]$, parity of the index: $1,0,0,1$):

$$c_4^v(x,y) = y/N' \cdot c_5^v(x,y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_5^v(x,y+1)), \text{ for } v \in \{0,3\}$$
$$c_4^v(x,y) = x/N' \cdot c_5^v(x,y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_5^v(x+1,y)), \text{ for } v \in \{1,2\}$$

Case $u = 5$ (guessing $S[i + S[z + k]]$, parity of the index: $1, 1, 0, 0$):

$$c_5^v(x,y) = y/N' \cdot c_6^v(x,y) + (1 - y/N') \cdot (N' - y) \cdot (1 + c_6^v(x,y+1)), \text{ for } v \in \{0,1\}$$
$$c_5^v(x,y) = x/N' \cdot c_6^v(x,y) + (1 - x/N') \cdot (N' - x) \cdot (1 + c_6^v(x+1,y)), \text{ for } v \in \{2,3\}$$

Case $u = 6$ (guessing $S[j + S[i + S[z + k]]]$, checking keystream and contradictions, parity of the index: $1, 0, 0, 1$):

$$c_6^0(x,y) = y/N' \cdot 1/N' \cdot c_1^1(x,y) + (1 - y/N')^2 \cdot (1 + c_1^1(x,y+1))$$
$$c_6^1(x,y) = x/N' \cdot 1/N' \cdot c_1^2(x,y) + (1 - x/N')^2 \cdot (1 + c_1^2(x+1,y))$$
$$c_6^2(x,y) = x/N' \cdot 1/N' \cdot c_1^3(x,y) + (1 - x/N')^2 \cdot (1 + c_1^3(x+1,y))$$
$$c_6^3(x,y) = y/N' \cdot 1/N' \cdot c_1^0(x,y) + (1 - y/N')^2 \cdot (1 + c_1^0(x,y+1))$$

Remark. The resulting system of linear equations is rather large, for example we get almost 400 000 equations for $N = 256$. Instead of solving them at once, we can solve each set of 24 equations for 24 variables $c_u^v(x,y)$ (where x and y are fixed) separately. Carefully chosen order of parameters (x,y), i.e. order in which the equations are solved, guarantees that values $c_u^v(x+1,y)$ and $c_u^v(x,y+1)$ are already computed.

Table 4 shows results of our theoretical analysis. It compares them with results computed using equations in the original paper [3]. The table also shows some experimental data obtained for selected parameters N, x, and y. We can estimate the complexity of special state recovery as 2^{941}, for $N = 256$, if the complexity of guessing j and k is included (i.e. $(N/2)^2 = 2^{14}$).

			Theory		
N	x	y	Our analysis	[3]	Experiments
16	0	0	23.02	24.80	23.08
18	0	0	27.02	29.80	27.14
20	0	0	31.19	35.10	31.28 *
64	22	22	13.99	34.50	14.07
128	50	50	15.19	59.32	15.26
256	110	110	12.37	87.60	12.13
64	0	0	152.62	194.11	–
128	0	0	381.03	500.05	–
256	0	0	927.00	1235.32	–

Note 1: The numbers represent "lg" values of the complexity.
Note 2: Last column shows average value from 1000 experiments, except for the value with * (100 experiments average).

Table 4: Spritz – complexity of special state recovery

5 Conclusion

We analyzed the complexity of backtracking-based state recovery attacks on RC4 and Spritz. Our estimates are significantly better than known results on Spritz. We also adapted the change order optimization and proposed the prefix check optimization for simple state recovery attack on Spritz.

All computations were performed by SageMath[1]. In order to allow anyone to verify or experiment with the complexity equations, we provide the source code used for solving them[2] as well as the source code used for our experiments[3].

There are some interesting open problems for further research. The prefix check optimization deserves a theoretic analysis and a more precise estimate of how it improves the complexity of the state recovery backtracking. Another problem is to better estimate the probabilities that some position in the internal permutation holds an already known value (or is still unknown). These probabilities are used in the complexity equations, and therefore they influence the estimated complexity of an attack (although the experiments show only a marginal impact on the overall complexity).

Acknowledgment

Supported by VEGA 1/0967/16.

References

[1] AlFardan N., Bernstein D., Paterson K., Poettering B., Schuldt J.: On the Security of RC4 in TLS and WPA, 22nd USENIX Security Symposium, pp. 305–320, 2013.
[https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/alFardan]

[2] Ankele R., Kölbl S., Rechberger C.: State-recovery analysis of Spritz, Progress in Cryptology – LATINCRYPT 2015, Springer, LNCS 9230, pp. 204–221, 2015.
Cryptology ePrint Archive, Report 2015/828, [https://eprint.iacr.org/2015/828]

[3] Banik S., Isobe T.: Cryptanalysis of the Full Spritz Stream Cipher, Fast Software Encryption 2016, [to appear].
Cryptology ePrint Archive, Report 2016/092, [https://eprint.iacr.org/2016/092]

[4] Fluhrer S., Mantin I., Shamir A.: Weaknesses in the Key Scheduling Algorithm of RC4, Selected Areas of Cryptography: SAC 2001, Springer, LNCS 2259, pp. 1–24, 2001.

[5] Golić J., Morgari G.: Iterative Probabilistic Reconstruction of RC4 Internal States, Cryptology ePrint Archive, Report 2008/348, 2008. [https://eprint.iacr.org/2008/348]

[6] Knudsen L.R., Meier W., Preneel B., Rijmen V., Verdoolaege S.: Analysis Methods for (Alleged) RC4, Advances in Cryptology – ASIACRYPT '98, LNCS 1514, Springer, pp. 327–341, 1998.
[https://securewww.esat.kuleuven.be/cosic/publications/article-68.pdf]

[7] Mantin I., Shamir A.: A Practical Attack on Broadcast RC4, Fast Software Encryption: FSE 2001, Springer, LNCS 2355, pp. 152–164, 2002.

[8] Maximov A., Khovratovich D.: New State Recovery Attack on RC4, Advances in Cryptology – CRYPTO 2008, LNCS 5157, Springer, pp. 297–316, 2008.
Cryptology ePrint Archive, Report 2008/017, [https://eprint.iacr.org/2008/017]

[9] Ohigashi T., Isobe T., Watanabe Y., Morii M.: How to Recover Any Byte of Plaintext on RC4, In Selected Areas in Cryptography 2013, LNCS, Vol. 8282, pp. 155–173, 2013.

[1] Sage Mathematics Software (Version 7.0), 2016, [http://www.sagemath.org].
[2] https://github.com/mgabris/state-recovery-equations
[3] https://github.com/mgabris/state-recovery-backtrack

[10] Paul G., Maitra S.: Permutation after RC4 Key Scheduling Reveals the Secret Key, Selected Areas of Cryptography: SAC 2007, Springer, LNCS 4876, pp. 360–377, 2007.

[11] Popov A.: Prohibiting RC4 Cipher Suites, RFC 7465, DOI 10.17487/RFC7465, February 2015. [https://www.rfc-editor.org/rfc/rfc7465.txt]

[12] Rivest R., Schuldt J.: Spritz – a spongy RC4-like stream cipher and hash function, 2014. [https://people.csail.mit.edu/rivest/pubs/RS14.pdf]