State-recovery analysis of Spritz

Ralph Ankele[1], Stefan Kölbl[2] and Christian Rechberger[2]

[1] Graz University of Technology, Graz, Austria
ralph.ankele@alumni.tugraz.at
[2] DTU Compute, Technical University of Denmark, Denmark
{stek,crec}@dtu.dk

Abstract. RC4 suffered from a range of plaintext-recovery attacks using statistical biases, which use substantial, albeit close-to-practical, amounts of known keystream in applications such as TLS or WEP/WPA. Spritz was recently proposed at the rump session of CRYPTO 2014 as a slower redesign of RC4 by Rivest and Schuldt, aiming at reducing the statistical biases that lead to these attacks on RC4.
Even more devastating than those plaintext-recovery attacks from large amounts of keystream would be state- or key-recovery attacks from small amounts of known keystream. For RC4, there is unsubstantiated evidence that they may exist, the situation for Spritz is however not clear, as resistance against such attacks was not a design goal.
In this paper, we provide the first cryptanalytic results on Spritz and introduce three different state recovery algorithms. Our first algorithm recovers an internal state, requiring only a short segment of keystream, with an approximated complexity of 2^{1400}, which is much faster than exhaustive search through all possible states, but is still far away from a practical attack. Furthermore, we introduce a second algorithm that uses a pattern in the keystream to reduce the number of guessed values in our state recovery algorithm. Our third algorithm uses a probabilistic approach by considering the permutation table as probability distribution.
All in all, rather than showing a weakness, our analysis supports the conjecture that compared to RC4, Spritz may also provide higher resistance against potentially devastating state-recovery attacks.
Keywords: Spritz, RC4, stream cipher, state recovery, cryptanalysis

1 Introduction

Spritz [1] is a stream cipher and hash function designed by Ronald L. Rivest and Jacob Schuldt. It is a complete redesign of the widely deployed stream cipher RC4. After the recent attacks on block ciphers in CBC mode in TLS, such as BEAST [2], Lucky 13 [3] and POODLE [4], the usage of RC4 was recommended. In 2013, AlFardan et. al. [5] published some new attacks on RC4 that require only 2^{24} TLS connections by exploiting new biases in RC4. RC4 has been well analyzed over the last years and some serious weaknesses have been found [6]. In practice it is often possible to mitigate such weaknesses by dropping the first n

bytes of the keystream. Nevertheless, the attacks on RC4 are getting very close to become practical and it is no longer recommended to use RC4. According to the community [7], some state organizations may have found a practical attack on RC4, which should further discourage the use of RC4 in any application. Recently, at CRYPTO 2014 (rump session), Ronald L. Rivest announced Spritz as a drop-in replacement of RC4.
The designers of Spritz claim their security bounds based on various statistical tests and extensive simulations, but did not publish a detailed security analysis in their proposal. In this paper, we want to amend this by providing the first cryptanalytic results on Spritz. If Spritz offers strong security bounds in various analyses, it could be used to replace RC4. Due to the big spectrum of applications and protocols in which RC4 is used, it is imperative though that a successor provides tight security bounds against all possible attack vectors.

Related Work. As Spritz was recently proposed as a drop-in replacement of RC4, no other cryptanalytic results are known to us.

There have been published some statistical weaknesses in the comparison between Spritz and VMPC-R [8]. Bartosz Zoltak [9], the designer of VMPC-R, published a statistical weakness in Spritz that shows a bias when observing the probability $\Pr(\text{output}(x) = \text{output}(x+2))$ for a simplified version of Spritz, that can distinguish the Spritz output from a random oracle after observing $2^{21.9}$ outputs.

Due to the different mode of operation and more complex structure of Spritz it is unclear to what extend previous state recovery attacks [10][11][12] on RC4 can be applied on Spritz. While those attacks typically have a very high complexity they might aid in comparing the security margins of RC4 and Spritz.

Contributions and Outline. In this paper, we provide the first results in the analysis of the recently announced stream cipher and hash function, Spritz. We first show some observations on Spritz like partial state rotations and that there exists a cycle in the keystream with length 6N, when no Swap is done. Using these observations, we implemented three different state recovery attacks on Spritz and compared our results with the state recovery on RC4. Spritz is significantly slower than RC4 ([1], Section 9, Performance analysis), and as it turns out also provides higher security against state recovery attacks than RC4 (see Table 1).

Our first state recovery algorithm uses a recursive backtracking algorithm and has a lower complexity than exhaustive search through all possible states. Furthermore, we provide two different state recovery algorithms, where the first one searches for a pattern in the keystream, which allows an adversary to easily recover values in the permutation according to the known register values in a short window. Our third algorithm uses a probabilistic approach by considering the permutation table as a probability distribution similar to the ideas proposed in [10].

The source code of our attacks is public available for independent verification [13].

The remainder of this paper is structured as follows. Section 2 gives a detailed description of Spritz, where its structure and properties are discussed. In Section 3 we give some general observations on Spritz. Our state recovery attacks are introduced in Section 4. We conclude in Section 5 and give an outlook of further improvements.

Table 1: Comparison of state recovery attacks between RC4 and Spritz.

		Complexity		
Cipher	Permutation size	Time	Data	Reference
RC4	32	2^{53}	2^{32}	[10]
Spritz	32	2^{99}	2^{5}	*This work*
RC4	64	2^{132}	2^{64}	[10]
RC4	64	2^{60}	2^{60}	[11]
Spritz	64	2^{249}	2^{6}	*This work*
RC4	128	2^{324}	2^{128}	[10]
RC4	128	2^{113}	2^{112}	[11]
Spritz	128	2^{599}	2^{7}	*This work*
RC4	256	2^{779}	2^{256}	[10]
RC4	256	$2^{241\star}$	2^{240}	[11]
RC4	256	$2^{262\dagger}$	2^{211}	[12]
Spritz	256	2^{1400}	2^{8}	*This work*

2 Description of Spritz

Spritz [1] was proposed as an improved variant of the stream cipher RC4 and follows a similar design strategy. However, the update procedure is more complex and it uses a sponge-like construction.

Spritz builds upon a permutation and a few pointers (i.e. 8-bit registers). The UPDATE function of Spritz is the result of extensive simulations and statistical analysis. The designers created the UPDATE function by trying all possible candidates, constrained to six registers and a permutation of size N, and chose the one with the best security properties. The parameters were retrieved using a computing cluster with approximately five months of computation time, according to the designers [1].

Spritz uses a permutation $S = \{0, 1, \ldots, N-1\}$ with N elements. Additionally, it consists of six registers: i, j, k, z, w and a. Registers i, j and k are used as pointers to the permutation S in the UPDATE function. The value of register

* Under some realistic assumptions. See [11] Section 6. for details.
† With N/10 pre-assigned entries to the state recovery table. See [12] for details.

w is always relatively prime to N and is used to update register i. Updating register i by addition of register w causes it to cycle through all values modulo N. Register a denotes the number of nibbles that have been absorbed. Register z stores the last generated value of the output keystream. The state of Spritz consists of the six registers and the permutation S. Spritz has a maximum of $N! \cdot N^6 \approx 2^{1730}$ different states, for $N = 256$. The key in Spritz is an arbitrary length byte array $K[0, \ldots, L-1]$ where L denotes the length of the key. The default values for L are 16 and 32, resulting in key sizes of 128 and 256 bits, respectively.

The key schedule in Spritz is done by the ABSORB function, which takes an arbitrary length input of N-values as key. Each value is split into two half-nibbles, where register a counts the number of absorbed nibbles. For more details about the Spritz functions we refer to the Spritz proposal [1] and additionally give a short overview in Appendix B.

To produce an output word z_t the state of Spritz is updated as follows:

$$
\begin{aligned}
i_t &= i_{t-1} + w & (1)\\
j_t &= k_{t-1} + S_{t-1}[j_{t-1} + S_{t-1}[i_t]] & (2)\\
k_t &= i_t + k_{t-1} + S_{t-1}[j_t] & (3)\\
S_t[i_t] &= S_{t-1}[j_t],\ S_t[j_t] = S_{t-1}[i_t] & (4)\\
z_t &= S_t[j_t + S_t[i_t + S_t[z_{t-1} + k_t]]] & (5)
\end{aligned}
$$

where all operations are modulo N. In the next-state function, which is composed of Equations (1) to (5), which itself is a composition of the UPDATE and OUTPUT function, first the registers i, j and k are updated. Then the permutation S is swapped, at the index of registers i and j. In one update step, all words in the permutation remain the same except the swapped ones. In the end the output word z_i is generated.

3 Some Observations on Spritz

In our analysis of Spritz we made various observations, which might help to gain a better insight in Spritz. First, we show some partial state rotations that occur when Spritz is absorbing input of a specific form. Moreover, we analysed how Spritz works, if no swapping in the state update occurs and show that the output of Spritz quickly runs into a cycle, if the SWAP frequency is reduced to 0.

3.1 Partial State Rotations

We observed that the Spritz state partially rotates during ABSORB of particular messages. If we consider an internal state $S = \{s_0, s_1, \ldots, s_n\}$ and absorb a message block $M = \{0, 0, \ldots, 0\}$ where $|M| \leq N/4$ we can show that it holds that

$$S[i] = \begin{cases} s_{128}, & \text{if } i = 0 \\ s_{i-1}, & \text{if } 1 \leq i < 128 \\ s_0, & \text{if } i = 128 \\ s_i, & \text{if } 129 \leq i < 255 \end{cases}$$

Absorbing an input in Spritz increments the nibble counter a and the value of the permutation, at position of the nibble counter a is swapped with the value at $\lfloor N/2 \rfloor + x$, where x is the value of the absorbed nibble. If we absorb a message with the same content for each byte, Spritz results in a partial state rotation. These partial state rotations in Spritz are illustrated in Table 2.

Table 2: Partial state rotations in Spritz during Absorb.

Step a	$\lfloor N/2 \rfloor + x$	$S^{(i)}$										
0	128	$\textcircled{$s_{128}$}$	s_1	s_2	s_3	$\dots$	s_{127}	$\textcircled{$s_0$}$	s_{129}	$\dots$	s_{255}
1	128	s_{128}	$\textcircled{$s_0$}$	s_2	s_3	$\dots$	s_{127}	$\textcircled{$s_1$}$	s_{129}	$\dots$	s_{255}
2	128	s_{128}	s_0	$\textcircled{$s_1$}$	s_3	$\dots$	s_{127}	$\textcircled{$s_2$}$	s_{129}	$\dots$	s_{255}
3	128	s_{128}	s_0	s_1	$\textcircled{$s_2$}$	$\dots$	s_{127}	$\textcircled{$s_3$}$	s_{129}	$\dots$	s_{255}
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
127	128	s_{128}	s_0	s_1	s_1	$\dots$	$\textcircled{$s_{126}$}$	$\textcircled{$s_{127}$}$	s_{129}	$\dots$	s_{255}

Since there is a call to Shuffle after absorbing $N/2$ nibbles, these state rotations only occur when absorbing less than $N/4$ bytes, and disappear after Shuffle gets called. Moreover, due to the addition of the nibble x in $\lfloor N/2 \rfloor + x$ in Absorb, these state rotations are shifted by x if $x > 0$.

3.2 Cycle in the Keystream with no Swap

Using Spritz as a stream cipher generates a pseudorandom keystream that can be added to a plaintext stream to obtain a ciphertext stream. As the keystream is only pseudorandom, this implies that after some time it repeats and runs in a cycle. Fluhrer et al. [6] studied this behaviour on RC4 and showed that RC4 without the Swap operation is useless as a keystream generator, as the keystream of RC4 becomes cyclic with a period of $2b$ (Note b in RC4 is 2^n, where n is the permutation's size). We observed that if no Swap is done in the next-state function and no input is absorbed, the output of Spritz runs in a cycle of length $6N$. If no Swap is done in RC4 and Spritz, the permutation S remains constant. The output z_n in RC4 depends on the still changing pointer registers

i and j, where in Spritz we have an additional register k and the previous value of the keystream z_{n-1} that leads to the additional overhead, which increases the output cycle length of Spritz to $6N$, if no Swap is done.

4 State Recovery Attacks

Recovering the internal state of Spritz would enable an attacker to trivially predict any further output of the key stream. Furthermore, an adversary can try to recover the internal state, after the call to Shuffle in the output functions Squeeze or Drip, by using an inverted next-state function and the knowledge of any internal state. Because of the call to Shuffle, it's not easily possible for an adversary to recover the initial state, since Crush in Shuffle aims to be non-invertible.

4.1 State Recovery with Backtracking

We introduce a recursive state recovery attack with backtracking and use a similar approach as Knudsen et al. [10] in their analysis of RC4. The state recovery attack needs only N output keystream bytes to successfully recover the internal state.

The idea of our recursive backtracking algorithm can be described as follows: We simulate the Update and Output function of Spritz as long as all input to these functions is known. If a value is unknown we simply guess it and proceed. In the state recovery attack, we can start at any point, but we assume that the initial registers are correctly known (either from a previous different attack, through some heuristics or by simply guessing them). If we absorb no input and start at the beginning, we know the initial values of the registers, but after one application of Shuffle (e.g. in Drip when any input was absorbed) we lose knowledge of the register values. In each step, we have to guess at most five unknown values (i.e. $S_{t-1}[i_t]$, $S_{t-1}[j_{t-1} + S_{t-1}[i_t]]$, $S_{t-1}[j_t]$, $S_t[z_{t-1} + k_t]$ and $S_t[i_t + S_t[z_{t-1} + k_t]]$) that we need to process to the next state.

For steps $t = 1 \ldots N$ we only proceed if $z'_t = z_t$, where z_t is the output word observed in the known output stream at step t, and z'_t is the output word from our simulation. In our tests we observed that the output word z'_t sometimes equals z_t even when we guessed the wrong entries in the partially recovered state. To avoid searching through dead branches in our search tree we impose several restrictions:

1. S is a permutation table, hence every element occurs only once. This reduces the number of possible values, which we have to guess, when a value is unknown.
2. If the known output word z'_t has been assigned to S during a previous guess, we can look if the index $j_t + S_t[i_t + S_t[z_{t-1} + k_t]]$ is equal to the position of the previous assigned value. If the indices are equal we can proceed to step $t+1$. If not, we have a contradiction and can cut off the branch in the search tree.

3. If the known output word z'_t has not been assigned to S during a previous guess, we can again look if there is already a value at index $j_t + S_t[i_t + S_t[z_{t-1} + k_t]]$. If there is already a value, we again have a contradiction and can cut off the branch in the search tree. If not, we can set z'_t at position of index $j_t + S_t[i_t + S_t[z_{t-1} + k_t]]$ and then proceed with step $t + 1$.

Algorithm 1 State Recovery Algorithm with Backtracking

```
function RECOVERSTATE(t)
    $i_t \leftarrow i_{t-1} + w$
    if $S_{t-1}[i_t]$ is not assigned then
        Guess $S_{t-1}[i_t] \leftarrow v$                                    ▷ for 0 ≤ v < N
    end if
    if $S_{t-1}[j_{t-1} + S_{t-1}[i_t]]$ is not assigned then
        Guess $S_{t-1}[j_{t-1} + S_{t-1}[i_t]] \leftarrow v$                 ▷ for 0 ≤ v < N
    end if
    $j_t \leftarrow k_{t-1} + S_{t-1}[j_{t-1} + S_{t-1}[i_t]]$
    if $S_{t-1}[j_t]$ is not assigned then
        Guess $S_{t-1}[j_t] \leftarrow v$                                    ▷ for 0 ≤ v < N
    end if
    $k_t \leftarrow i_t + k_{t-1} + S_{t-1}[j_t]$
    $S_t[i_t] \leftarrow S_{t-1}[j_t]; S_t[j_t] \leftarrow S_{t-1}[i_t]$                ▷ Swap(S[i], S[j])
    if $S_t[z_{t-1} + k_t]$ is not assigned then
        Guess $S_t[z_{t-1} + k_t] \leftarrow v$                              ▷ for 0 ≤ v < N
    end if
    if $S_t[i_t + S_t[z_{t-1} + k_t]]$ is not assigned then
        Guess $S_t[i_t + S_t[z_{t-1} + k_t]] \leftarrow v$                   ▷ for 0 ≤ v < N
    end if
    $z'_t = S_t[j_t + S_t[i_t + S_t[z_{t-1} + k_t]]]$
    if $z'_t$ equals any word in S then
        if $S_t[j_t + S_t[i_t + S_t[z_{t-1} + k_t]]] \neq$ position of any word in S then
            contradiction
        else
            RECOVERSTATE(t + 1)
        end if
    else
        if $j_t + S_t[i_t + S_t[z_{t-1} + k_t]] \neq$ position of any word in S then
            $S_t[j_t + S_t[i_t + S_t[z_{t-1} + k_t]]] = z'_t$
            RECOVERSTATE(t + 1)
        else
            contradiction
        end if
    end if
end function
```

We implemented the state recovery algorithm in a recursive function RECOVERSTATE() (see Algorithm 1), where most branches end up in contradictions.

If we achieve to fill up the internal state table in one branch (at maximum of N steps) we can verify the solution by calculating repeatingly output bytes and comparing it with the ciphertext stream. Afterwards, we calculate the initial state by inverting the next-state function. Furthermore, we can speed up the search if we pre-assign the state recovery table with a few previously known values.

Complexity. The complexity is given by the number of steps performed, until a solution is found. In case of our state recovery attack on Spritz, the complexity is measured in the total number of assignments made for all entries in the initial table S_0. 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. Afterwards, we compute the complexity based on the number of known bytes x in the permutation S and the assigned probabilities.

The complexity of the state recovery attack on Spritz is

$$\sum_{i=1}^{5} c_i(x) = \frac{x}{N} \cdot c_{i+1}(x) + (1 - \frac{x}{N}) \cdot (N - x) \cdot c_{i+1}(x+1) \tag{6}$$

$$c_6(x) = \frac{x}{N} \cdot ((1 - \frac{x}{N}) \cdot 1 + c_1(x)) + (1 - \frac{x}{N}) \cdot (\frac{x}{N} \cdot 1 + c_1(x+1)) \tag{7}$$

In our state recovery attack we have to guess at least five values. The first part of Equation (6), which checks if the value is already in our partial recovered state table, will succeed on average with x/N. If it succeeds, we go to the next step without assigning a value. If not (second part of Equation (6)), we have to guess every possible value (which we do in this part of Equation (6): $(1-x/N)\cdot(N-x)$), and proceed until we run into a contradiction (see Equation (7)) or succeed.

The complexity of the last step of our state recovery algorithm (Equation (7)) can be calculated in a similar way, but we additionally have to consider the probability that we run into a contradiction. Therefore, in $c_6(x)$ we first check if z'_n is already in our partly recovered permutation table, which again succeeds with x/N and a contradiction occurs with a probability of $1-x/N$. In the second part, the output value z'_n is not in our partly recovered permutation table, with a probability of $1-x/N$. In this case, a contradiction can occur with probability x/N.

Another way of estimating the complexity is to experimentally observe it by counting how many assignments for all entries in the permutation S are made until we recover the initial state.

The results of our state recovery algorithm are shown in Table 3 where x gives the number of pre-assigned values in the state table. Furthermore, results for different N are given in Appendix C. The complexities are calculated using Equations (6) and (7). Additionally, we give the experimental complexity that we observed during testing our algorithm. The complexities are significantly lower than exhaustive search, but still far from practical for bigger N. Our algorithm becomes infeasible at $N = 32$, and is, therefore no threat to Spritz with $N = 256$. In our experiments we chose the number of pre-assigned values as low as possible to get a computable complexity. Moreover, we noted that there is a gap between the calculated values and our experimental observed values, which suggest that the attacks are even more powerful in practice than predicted by our complexity formulas.

Table 3: Calculated and experimental observed complexities for $N = 8, \ldots, 256$ using the backtracking algorithm, where x gives the number of pre-assigned values in the state table.

	calculated			experimental			
N	x	time comp.	$N!$	x	measured	time comp.	$(N-x)!$
8	0	$2^{13.7}$	$2^{15.2}$	0	$2^{11.7}$	$2^{13.7}$	$2^{15.2}$
16	0	$2^{44.3}$	$2^{44.2}$	5	$2^{18.5}$	$2^{22.7}$	$2^{25.3}$
32	0	$2^{99.8}$	$2^{117.6}$	19	$2^{21.3}$	$2^{28.5}$	$2^{32.5}$
64	0	$2^{249.0}$	2^{296}	47	$2^{20.8}$	$2^{42.2}$	$2^{48.3}$
128	0	$2^{599.4}$	$2^{716.1}$	109	$2^{18.7}$	2^{49}	$2^{56.8}$
256	0	2^{1400}	$2^{1683.9}$	-	-	-	-

4.2 Pattern Search and State Recovery

Our pattern search approach was inspired by the state recovery algorithm proposed by Maximov and Khovratovich [11]. In our algorithm, we applied a known plaintext attack, where we assumed that according to a pattern in the output keystream, all register values in a given window of length w_l are known. Therefore, we can easily derive a formula $S_{t-1}[j_t] = k_t - i_t - k_{t-1}$ from the update rule of register k. This allows us to compute some values of the permutation S without guessing additional values.

With the known entries in the permutation S and the knowledge of the register values during the window of length w_l, we can now apply our state recovery algorithm. It iteratively tries to recover an internal state. As long as we are inside our window, the register values are known and we can easily compute new values. If we lose knowledge of register j or k, outside of our window, we have to guess new permutation entries to proceed in our state recovery algorithm.
Assume at step t in a window, with length w_l, of the keystream z all the values $j_t, j_{t+1}, \ldots, j_{t+w_l}$ and $k_t, k_{t+1}, \ldots, k_{t+w_l}$ are known. Then we can according to

$$S_{t-1}[j_t] = k_t - i_t - k_{t-1}$$

calculate w_l entries for $S_{t-1}[j_t]$, which after SWAP become $S_t[i_t]$. Unfortunately, due to the uniform distribution of register j, the values are randomly distributed through our state recovery table. Nevertheless, we only have to guess three unknown values:

$$S_{t-1}[i_t],\ S_{t-1}[j_{t-1} + S_{t-1}[i_t]],\ S_{t-1}[j_t]$$

instead of five, as in our previous state recovery attack with backtracking (see Section 4.1). Our state recovery algorithm can be described as shown in Algorithm 2.

Algorithm 2 State Recovery Algorithm with Pattern Search

```
As long as registers $i$, $j$ and $k$ are known:

1. Calculate $S_{t-1}[j_t] = k_t - i_t - k_{t-1}$
2. Swap $S_t[i_t] \leftarrow S_{t-1}[j_t]; S_t[j_t] \leftarrow S_{t-1}[i_t]$
3. Check if $S_t[z_{t-1} + k_t]$ is already known
   3.1. If true $\rightarrow$ check if $S_t[i_t + s_t[z_{t-1} + k_t]]$ is already known
      3.1.1. If true $\rightarrow$ check if at index $j_t + S_t[i_t + s_t[z_{t-1} + k_t]]$ is already a value
         3.1.1.1. If true $\rightarrow$ compare if it is the same value as $z_t$
         3.1.1.2. If false $\rightarrow$ set $z_t$ at index $j_t + S_t[i_t + s_t[z_{t-1} + k_t]]$

If registers $j$ and $k$ are no longer known:

4. Guess $S_{t-1}[i_t]$
5. Guess $S_{t-1}[j_{t-1} + S_{t-1}[i_t]]$
   5.1. Calculate $j_t = k_{t-1} + S_{t-1}[j_{t-1} + S_{t-1}[i_t]]$
6. Guess $S_{t-1}[j_t]$
   6.1. Calculate $k_t = i_t + k_{t-1} + S_{t-1}[j_t]$
7. Proceed with step 1
```

A contradiction can occur in steps $1 \dots 3$ if the newly calculated value is already in another cell, in the state table. Moreover, a contradiction can occur, if there already exists a value, at the index of the current computed value, and the new computed value differs from the already existing value.

Our state recovery algorithm assumes that for a given window of length w_l all registers are known. This assumption is based on a pattern in the keystream that lets us determine the register values with high probability. Therefore, we need two definitions to describe these patterns in more detail (these definitions were defined by Maximov and Khovratovich [11] and are adjusted to Spritz):

Definition 1. *A* d-order pattern *is a 5-tuple defined as*

$$P_d = \{i, j, k, I, V\}, \qquad i, j, k \in \mathbb{Z}_N, I, V \in \mathbb{Z}_N^d$$

At step t the internal state of Spritz is compliant with pattern P_d if $i_t = i$, $j_t = j$, $k_t = k$ and d entries of permutation S contain their values in vector V and their corresponding indices in vector I.

Definition 2. *A pattern P_w is called* w-generative*, if for any internal state that is compliant to P_w, in the next w_l steps, all registers are known and let us derive w_l formulas, of the form $S_{t-1}[j_t] = k_t - i_t - k_{t-1}$.*

It is obvious that such patterns exist in the keystream of Spritz. However, for efficiency of our state recovery algorithm, we need to find patterns with a high d-order and patterns that are high w-generative, so that we initially know the values for the registers in a large window. We implemented a simple pattern search algorithm (described in Algorithm 3), that first fixes $i = 0$, and then tries all N values for each j and k before increasing i. If for a combination i, j and k the first index, value pair from the vectors I and V fits, we have found a pattern with d-order = 1. Our algorithm requires a keystream of length 2^N, where the search of such a pattern is a pre-computation stage of our attack.

Algorithm 3 Pattern Search

```
function searchPattern(i, j, k, V, I)
    for n = 0 to $2^N$ do
        for $\{i, j, k\}$ = 0 to N do
            if $i_t = i$ and $j_t = j$ and $k_t = k$ then
                if keystream at position t equals $V_t$ and $I_t$ then
                    while keystream equals V and I do
                        $P \leftarrow V$ and $I$
                        $P \leftarrow i, j, k$
                    end while
                end if
            end if
        end for
        StorePattern(P)
    end for
end function
```

We implemented the state recovery algorithm and tested it for various sizes of N. If a contradiction occurs, we reset the responsible values in our state recovery table and continue with the recovery. In our tests we observed that even with some pre-assigned values, contradictions occurs too often, whereby we delete more information from our state recovery table, than we fill up with our state recovery attack. If we do not delete all responsible values when we reach a contradiction, we observed that either the complexity gets too high or we get too many wrong values in our state recovery table.

4.3 Probabilistic State Recovery

Knudsen et al. [10] proposed a probabilistic state recovery approach on RC4, that has been further improved by Golic and Morgari [12]. Our probabilistic state recovery algorithm follows a similar strategy and can be described as follows.

The initial state of the permutation S in Spritz depends on the secret key that is absorbed and is therefore, unknown to an attacker. We assume that all $N!$ possible states for the initial state are equally likely, i.e. the a priori probability distribution is uniform for the initial state. From the observation of the output keystream we gain information and can calculate an a posteriori probability distribution for the permutation S. After some steps the calculation for S should converge and we can recover the internal state.

In our probabilistic state recovery algorithm, we represent the information about the registers j, k and the permutation S by means of probability distributions. In each step, we calculate the a posteriori probability distribution of S_{dist}, j_{dist} and k_{dist}. We observe the keystream z and with the update rule for $z = S[j + S[i + S[z + k]]]$ and the Bayes rule we update the probability distributions. The distribution S_{dist} is represented as a $N \cdot N$ matrix (see Equation (8)), which represents the conditional probabilities of a given register and the associated entries in the permutation where the registers maps (e.g. $S[i][S_{t-1}[j_t]] = \Pr(S_{t-1}[i_t] \mid j)$).

$$S_{dist} = \begin{pmatrix} \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \end{pmatrix} \tag{8}$$

We implemented this state recovery algorithm (see Algorithm 4) in a function PROBABILISTICRECOVERSTATE() that runs for a given amount of steps, where for each step it updates the probabilities of our S_{dist}, j_{dist} and k_{dist} distributions. If the algorithm converges, it stops and we can invert the next-state function of Spritz to recover the initial state. The convergence criteria in this case is that in the S_{dist} distribution each value is either zero or one. Based on that we can map our S_{dist} distribution to our state recovery table.

Algorithm 4 Probabilistic State Recovery Algorithm

```
function PROBABILISTICRECOVERSTATE()
    INITIALIZESTATE()
    ABSORB(random key)
    $z$ ←DRIP()                                        ▷ Store output keystream
    $\{S_{dist}, j_{dist}, k_{dist}\}$ ← INITIALIZEPROBABILITYDISTRIBUTIONS()
    for step i = 1 to steps do
        $\{S_{dist}, j_{dist}, k_{dist}\}$ ← UPDATEPROBABILITYDISTRIBUTIONS()
        if $\{S_{dist}, j_{dist}, k_{dist}\}$ converges then
            RECOVERINITIALSTATE()
        end if
    end for
end function
```

We tested our probabilistic state recovery algorithm with different sizes for N and with pre-assigned values (the distribution table was accordingly adjusted) for the state table, but even for small sizes of N our algorithm did not converge after a few steps and the complexity becomes infeasible.

5 Conclusion

We have introduced three different state recovery algorithms for Spritz. The algorithms try to recover the initial state of Spritz by reconstructing any internal state. Our best state recovery algorithm has a complexity of 2^{1400} and requires only a small amount of known keystream. Nevertheless, the complexity is far from practical and is no real threat for Spritz using the default parameters. The consequences of a practical state recovery attack on any cipher would break the cipher. An adversary would be able to produce further output bytes, without knowing the secret key and therefore leverage the security provided by the secret key.

Furthermore, we implemented two additional state recovery algorithms based on pattern search and a probabilistic approach. However, the state recovery algorithm with the pattern search did not converge, because of too many contradictions and the probabilistic state recovery algorithm had a very high complexity, after very few steps.

As cryptanalytic results only get better over time we list here a few ideas for possible improvements. All three state recovery algorithms can be improved by heuristics that let an attacker pre-assign values into the state recovery table. Another possible improvement can be the combination of some of the state recovery algorithms, like combining the backtracking and probabilistic approach or only partially recover an internal state. It is our hope that our results will encourage others for further research on Spritz, to get a better understanding of its security margin.

References

1. Rivest, R.L., Schuldt, J.C.N.: Spritz—a spongy RC4-like stream cipher and hash function. http://people.csail.mit.edu/rivest/pubs/RS14.pdf (2014)
2. Duong, T., Rizzo, J.: Here come the ⊕ Ninjas. BEAST attack (2011)
3. Al Fardan, N., Paterson, K.: Lucky Thirteen: Breaking the TLS and DTLS Record Protocols. In: Security and Privacy (SP), 2013 IEEE Symposium on. (2013) 526–540
4. Moeller, B., Duong, T., Kotowicz, K.: This POODLE Bites: Exploiting The SSL 3.0 Fallback. https://www.openssl.org/~bodo/ssl-poodle.pdf (2014)
5. AlFardan, N.J., Bernstein, D.J., Paterson, K.G., Poettering, B., Schuldt, J.C.N.: On the Security of RC4 in TLS and WPA. http://www.isg.rhul.ac.uk/tls/RC4biases.pdf (2013)
6. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Selected Areas in Cryptography. Lecture Notes in Computer Science (2001)
7. Schneier, B.: The NSA Is Breaking Most Encryption on the Internet. https://www.schneier.com/blog/archives/2013/09/the_nsa_is_brea.html (2013)
8. Bartosz, Z.: VMPC One-Way Function and Stream Cipher. In: Lecture Notes in Computer Science, Springer (2004)
9. Bartosz, Z.: Statistical weakness in Spritz against VMPC-R: in search for the RC4 replacement. Cryptology ePrint Archive, Report 2014/985 (2014)
10. Knudsen, L.R., Meier, W., Preneel, B., Rijmen, V., Verdoolaege, S.: Analysis Methods for (Alleged) RC4. In: Advances in Cryptology - ASIACRYPT '98. Volume 1514 of Lecture Notes in Computer Science., Springer (1998) 327–341
11. Maximov, A., Khovratovich, D.: New State Recovery Attack on RC4. In: Advances in Cryptology – CRYPTO 2008. Lecture Notes in Computer Science, Springer (2008)
12. Golic, J., Morgari, G.: Iterative Probabilistic Reconstruction of RC4 Internal States. Cryptology ePrint Archive, Report 2008/348, http://eprint.iacr.org/2008/348 (2008)
13. Ankele, R.: https://github.com/ralphankele/Spritz (2015)

A Pseudocode of the Spritz functions

```
InitializeState(N)
i := j := k := z := a := 0
w := 1
for v = 0 to N − 1
    S[v] := v

Absorb(I)
for v = 0 to I.length − 1
    AbsorbByte(I[v])

AbsorbByte(b)
AbsorbNibble(Low(b))
AbsorbNibble(High(b))

AbsorbNibble(x)
if a = ⌊N/2⌋
    Shuffle()
Swap(S[a], S[⌊N/2⌋ + x])
a := a + 1

AbsorbStop()
if a = ⌊N/2⌋
    Shuffle()
a := a + 1

Shuffle()
Whip(2N)
Crush()
Whip(2N)
Crush()
Whip(2N)
a := 0

Whip(r)
for v = 0 to r − 1
    Update()
do w := w + 1
until Gcd(w, N) = 1

Crush()
for v = 0 to ⌊N/2⌋ − 1
    if S[v] > S[N − 1 − v]
        Swap(S[v], S[N − 1 − v])

Squeeze(r)
if a > 0
    Shuffle()
P := Array.New(r)
for v = 0 to r − 1
    P[v] = Drip()
return P

Drip()
if a > 0
    Shuffle()
Update()
return Output()

Update()
i := i + w
j := k + S[j + S[i]]
k := i + k + S[j]
Swap(S[i], S[j])

Output()
z := S[j + S[i + S[z + k]]]
return z
```

Fig. 1: Pseudocode of Spritz. All additions are modulo N. When N is a power of 2, the last two lines in Whip are equivalent to $w = w + 2$.

B Detailed Description of the Spritz functions

InitializeState. This function initializes Spritz and sets the registers i, j, k, z and a to zero and register w to one. The permutation S is initialized with the identity permutation.

Absorb. The Absorb function in Spritz absorbs arbitrary length input and updates the Spritz state accordingly. The input is split into bytes and absorbed with the AbsorbByte function. This function again splits the byte in two nibbles, while the lower nibble is absorbed first. For each nibble that is absorbed the register a is increased. If $a > \lfloor N/2 \rfloor$ is reached, Spritz is "full" (i.e. the capacity is reached) and Shuffle gets called. The AbsorbStop function is used in Spritz as padding function. AbsorbStop calls Shuffle if $a > \lfloor N/2 \rfloor$ and increments a by one. This is equivalent as absorbing a special stop symbol "▮" that is outside of the input alphabet.

Shuffle. Shuffle randomizes the Spritz state. To achieve good randomization three applications of Whip and two applications of Crush are performed alternatively. Whip calls Update $2N$ times without producing any output, in order to randomly update the registers i, j and k as well as the permutation S. Additionally, Whip updates register w every time it gets called to the next value relative prime to N. Crush maps $2^{N/2}$ states to one state, and looses information intentionally, which makes Crush a non-invertible transformation. In more detail, Crush compares iteratively the beginning to the end, and sorts the state ascending.

Squeeze, Drip. Squeeze and Drip are the output functions of Spritz. First, register $a > 0$ is checked, and if necessary Shuffle is called which puts Spritz in "squeezing mode" (i.e. $a = 0$). Afterwards Squeeze, calls Drip r times and returns the output in an array. Drip uses Update and Output to produce a new output byte.

Update, Output. Update is the next-state function of Spritz. In Update the registers i, j and k are updated and $S[i]$ and $S[j]$ are swapped. Output produces a single byte output by nested lookups in the permutation S mixed with the registers i, j, k and also feedback from the last produced output value.

C Detailed Results of State Recovery Attacks

In this section, the results of our state recovery attacks are highlighted. We implemented three different state recovery algorithms. Our best one uses backtracking and cuts off branches that result in a contradiction. We measured the complexity experimentally, which is given in the next tables as $exp.complexity$. Additionally, we calculated the complexity which is given as $calc.complexity$. The parameter x denotes the number of pre-assigned values in our state recovery table. The last column shows the number of possible values in the permutation S given by $N!$.

C.1 Results of the state recovery attack with backtracking

In Table 4 and Table 6 we applied our state recovery attack with random input, which leads to random initial states that we have to recover.

C.2 Results of the state recovery attack with backtracking and no input

We can show that the performance of our state recovery attack is much higher if no input is absorbed (i.e. the initial state is the identity permutation) which is highlighted in Table 7 to Table 8.

Table 4: Results for state recovery attack with backtracking for $N = 8$.

x	exp. complexity	calc. complexity	$(N-x)!$
07	2^{0}	2^{0}	2^{0}
06	$2^{1.7}$	2^{1}	2^{1}
05	$2^{2.7}$	$2^{2.6}$	$2^{2.6}$
04	$2^{4.6}$	$2^{4.6}$	$2^{4.6}$
03	$2^{6.2}$	$2^{6.9}$	$2^{6.9}$
02	$2^{8.1}$	$2^{9.5}$	$2^{9.5}$
01	$2^{9.0}$	$2^{11.3}$	$2^{12.3}$
00	$2^{11.7}$	$2^{13.7}$	$2^{15.3}$

Table 5: Results for state recovery attack with backtracking for $N = 16$.

x	exp. comp.	calc. comp.	$(N-x)!$
15	2^{0}	2^{0}	2^{0}
14	$2^{1.6}$	2^{1}	2^{1}
13	$2^{3.0}$	$2^{2.6}$	$2^{2.6}$
12	$2^{3.8}$	$2^{4.6}$	$2^{4.6}$
11	$2^{4.4}$	$2^{6.9}$	$2^{6.9}$
10	$2^{6.5}$	$2^{9.5}$	$2^{9.5}$
09	$2^{6.6}$	$2^{11.3}$	$2^{12.3}$
08	$2^{7.0}$	$2^{13.7}$	$2^{15.3}$
07	$2^{7.1}$	$2^{16.5}$	$2^{18.5}$
06	$2^{12.0}$	$2^{19.5}$	$2^{21.8}$
05	$2^{18.5}$	$2^{22.7}$	$2^{25.3}$
04	-	$2^{26.0}$	$2^{28.9}$
03	-	$2^{28.5}$	$2^{32.6}$
02	-	$2^{31.5}$	$2^{36.4}$
01	-	$2^{34.9}$	$2^{40.3}$
00	-	$2^{38.4}$	$2^{44.3}$

Table 6: Results for state recovery attack with backtracking for $N = 32$.

x	exp. comp.	calc. comp.	$(N-x)!$
31	2^{0}	2^{0}	2^{0}
30	$2^{1.5}$	2^{1}	2^{1}
29	$2^{2.3}$	$2^{2.6}$	$2^{2.6}$
28	$2^{3.5}$	$2^{4.6}$	$2^{4.6}$
27	$2^{5.4}$	$2^{6.9}$	$2^{6.9}$
26	$2^{9.4}$	$2^{9.5}$	$2^{9.5}$
25	$2^{15.3}$	$2^{11.3}$	$2^{12.3}$
24	$2^{15.9}$	$2^{13.7}$	$2^{15.3}$
23	$2^{18.9}$	$2^{16.5}$	$2^{18.5}$
22	$2^{19.3}$	$2^{19.5}$	$2^{21.8}$
21	$2^{19.7}$	$2^{22.7}$	$2^{25.3}$
20	$2^{20.1}$	$2^{26.0}$	$2^{28.9}$
19	$2^{21.3}$	$2^{28.5}$	$2^{32.6}$
18	-	$2^{31.5}$	$2^{36.4}$

Table 7: Results for state recovery attack with backtracking for $N = 8$ and no input (i.e. identity permutation as initial state).

x	exp. comp.	calc. comp.	$(N-x)!$
07	2^{0}	2^{0}	2^{0}
06	2^{1}	2^{1}	2^{1}
05	$2^{1.6}$	$2^{2.6}$	$2^{2.6}$
04	2^{2}	$2^{4.6}$	$2^{4.6}$
03	$2^{2.4}$	$2^{6.9}$	$2^{6.9}$
02	$2^{2.6}$	$2^{9.5}$	$2^{9.5}$
01	$2^{2.9}$	$2^{11.3}$	$2^{12.3}$
00	-	$2^{13.7}$	$2^{15.3}$

Table 8: Results for state recovery attack with backtracking for $N = 16$ and no input (i.e. identity permutation as initial state).

x	exp. comp.	calc. comp.	$(N-x)!$
15	2^{0}	2^{0}	2^{0}
14	$2^{1.6}$	2^{1}	2^{1}
13	2^{2}	$2^{2.6}$	$2^{2.6}$
12	$2^{3.6}$	$2^{4.6}$	$2^{4.6}$
11	$2^{4.1}$	$2^{6.9}$	$2^{6.9}$
10	$2^{5.3}$	$2^{9.5}$	$2^{9.5}$
09	$2^{6.3}$	$2^{11.3}$	$2^{12.3}$
08	$2^{7.6}$	$2^{13.7}$	$2^{15.3}$
07	$2^{11.9}$	$2^{16.5}$	$2^{18.5}$
06	$2^{14.45}$	$2^{19.5}$	$2^{21.8}$
05	$2^{14.45}$	$2^{22.7}$	$2^{25.3}$
04	$2^{14.45}$	$2^{26.0}$	$2^{28.9}$
03	$2^{14.45}$	$2^{28.5}$	$2^{32.6}$
02	$2^{14.45}$	$2^{31.5}$	$2^{36.4}$
01	$2^{14.45}$	$2^{34.9}$	$2^{40.3}$
00	-	$2^{38.4}$	$2^{44.3}$