Ralph Ankele, BSc

Cryptanalysis of Symmetric Key Primitives

MASTER'S THESIS

to achieve the university degree of

Master of Science

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Florian Mendel

Institute for Applied Information Processing and Communication
Infeldgasse 16a, AUT-8010 Graz, Austria

Graz, March 2015

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the used sources. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

SIGNATURE INFORMATION	Signatory	Ralph Ankele
	Date/Time-UTC	2015-03-15T21:29:55+01:00
	Verification	Information about the verification of the electronic signature can be found at: https://www.signaturpruefung.gv.at
Note	This document is signed with a qualified electronic signature. According to § 4 art. 1 of the Signature Act it in principle is legally equivalent to a handwritten signature.	

Date Signature

Acknowlegements

This thesis is the result of the contribution of many people. First of all, I want to thank my advisor at DTU Denmark, Christian Rechberger, for his patience and for being a constant source of help and guidance. Many thanks also to my co-advisors, Stefan and Tyge, who always helped me if I got any questions. I also have to thank Tyge for his patience with my Austrian accent. Additionally, I have to thank my advisor at TU Graz, Florian Mendel.

Many thanks also to Lars Knudsen and Christian Rechberger for inviting me to Copenhagen and to join the crypto group during my master thesis. In this place, I want to thank the whole crypto group of DTU for the warm welcome and for several discussions during lunch. Many thanks to Andrey, Arnab, Christian, Elmar, Hoda, Hongbo, Lars, Martin, Mohamed, Stefan, Subhadeep, Takanori and Tyge. I also want to thank our secretary, Ann-Cathrin, for all the administrative stuff she has done for me.

None of this would have been possible without the support of my family during my studies. I also want to thank my twin brother Robin for his useful comments that improved this thesis.

Furthermore, I want to thank my better half, Rebeca for her continuing support during my studies and for cheering me up when something didn't work out as expected.

Finally, I want to thank my cat, Bailys, for the frequent distractions and for eating some parts of my papers.

Abstract

The analysis of a cryptographic algorithm is important for the security of newly proposed cryptographic algorithms. Symmetric key algorithms model several different symmetric primitives like encryption schemes, hash functions, pseudorandom number generators, message authentication codes or authenticated encryption schemes.

In this thesis, we analyze the security of such a symmetric cryptographic algorithm, called Spritz. Spritz was announced at CRYPTO 2014 as a replacement of the widely adapted stream cipher RC4. RC4 shows some weaknesses, but there are still no practical attacks. Many cryptographers have published improved versions of RC4. Spritz is a redesign of RC4 from Ronald L. Rivest, the designer of RC4, and Jacob Schuldt. The designers of Spritz argued the security level of Spritz according to statistical tests and extensive simulations. However, they have not published a detailed security analysis so far.

We provide the first cryptanalytic results on Spritz within this thesis. We investigated the security of Spritz against generic attacks, searched for weak key classes and implemented several statistic tests to search for biases in the keystream. As Spritz is sponge-like, it offers many primitives from sponge-constructions. In this context, we analyzed Spritz as a hash function and searched for collisions and applied pre-image attacks. Finally, we propose three different state recovery attacks on Spritz as a stream cipher, where our best attack recovers the initial state for Spritz with complexity of approximately 2^{1400}. Even tough this is faster than exhaustive searching through all possible states it is still far away from a practical attack.

Keywords: sponge-like, stream cipher, hash function, RC4, Spritz, cryptanalysis, collision, state recovery attack

Contents

List of Figures

List of Tables

List of algorithms

Notation

Notation and Terminology

N: N is the number of words in the Spritz permutation. The default value of N in Spritz is 256.

Mathematical Operations: All mathematical operations in RC4 as well as Spritz are always modulo N (e.g. "+" means addition modulo N).

State: The state Q_t at time t of RC4 and Spritz consist of the permutation S and the registers.

Registers: RC4 and Spritz use registers each holding one byte.

Permutation: Both, RC4 and Spritz hold a permutation S, which is a byte-array of length N consisting a permutation of $Z_N = 0, 1, \ldots, N-1$. All entries in the array are modulo N.

Nibbles: In Spritz, a byte consists of two half-byte nibbles. A byte b can be represented by two nibbles x, y, where $b = x \mid\mid y$.

List of mathematical Symbols

$a \oplus b$	exclusive-or (XOR)	$\lfloor a \rfloor$	floor function
a & b	logical-and (AND)	$\lceil a \rceil$	ceil function
$a \mid\mid b$	concatenation of two strings	$a!$	factorial
$log(a)$	logarithm function (base 10)	$ln(a)$	logarithm function (base e)

1 Introduction

1.1. Motivation

In this thesis, we analyze the security of symmetric key primitives. Symmetric cryptographic algorithms can be categorized into two classes: block ciphers and stream ciphers. These symmetric ciphers can then be used to model different symmetric primitives, like hash functions, pseudorandom number generators, message authentication codes or authenticated encryption schemes.

Recently, in August 2014, the RC4-like stream cipher and hash function Spritz [RS14] has been published. To use such a newly proposed cipher securely, it has to be analyzed in detail to claim important and distinct security bounds. In this thesis, we review Spritz and give a detailed analysis of the security of Spritz.

Spritz is the successor of the widely adopted stream cipher RC4, which was designed in 1987 and has been implemented in various software applications and protocols. Over the years it has been well analyzed and some

weaknesses have been shown. In 2013, AlFardan et. al. [ABP+13] published an attack on TLS when using RC4 that needed only 2^{24} connections, with the same key. To overcome these security issues, the designer of RC4, Ronald L. Rivest together with Jacob Schuldt, proposed a new improved version of RC4, called Spritz.

Our contribution in the cryptanalysis of Spritz contains many different aspects and we try to find tight security bounds that will endorse the security of the cipher. In our analysis, we looked at the distribution of the registers in Spritz as well as on various correlations between states and registers. We analyzed Spritz as a hash function regarding collision attacks. Thereby, we searched for collisions, where we introduce message differences by absorbing two slightly different messages and cancel the differences in the Absorb, Whip or Crush function of Spritz. Additionally, we applied pre-image attacks on Spritz. Furthermore, we analyzed Spritz as a stream cipher. We analyzed possible weak key classes, the cyclic behavior of Spritz and we introduced a state recovery attack on Spritz. The complexity of our best state recovery attack, for the default version of Spritz, is approximately 2^{1400} which is faster than exhaustive search.

1.2. Related Work

In the search for an improved variant of RC4, many ciphers have been published. In 2004, Souradyuti and Preneel [SP04] presented their variant, RC4A which improves the security of RC4. Also in 2004, Zoltak [Bar04] published his variant, VMPC. Maitra and Paul [SG08] published RC4+ in 2008. RC4+ improves the key schedule of RC4. In August 2014, Spritz [RS14] was proposed from Rivest and Schuldt, where the former one is also the designer of RC4.

Spritz was proposed recently at CRYPTO 2014 (rump session) as a drop-in replacement of RC4. However, it is too early to expect many published cryptanalytic results on Spritz. Nevertheless, there are a few implementations [spr14] in different programming languages available.

There have been published some statistical weaknesses in the comparison between Spritz and VMPC-R. Bartosz Zoltak, the designer of VMPC-R, published a statistical weakness in Spritz that shows a bias when observing the probability $Prob(output(x) = output(x + 2))$ for a simplified version of Spritz, that can distinguish the Spritz output from an ideal primitive after observing $2^{21.9}$ outputs.

1.3. Outline

This thesis is organized as follows. In Chapter 2, an overview of symmetric key primitives is given. We introduce symmetric cryptography and show how stream ciphers and sponge constructions are designed and how they work.

Chapter 3 lists analysis methods for symmetric key primitives. First, we give some preliminaries on the analysis of cryptographic ciphers. Afterwards, we outline some generic attacks that are applicable to any cipher. Furthermore, we discuss some statistical attacks. We continue with attacks against stream ciphers, like key reuse, cycles in the keystream and state recovery. We conclude Chapter 3 with attacks on hash functions.

We give a detailed description of Spritz in Chapter 4. First, we show some attacks against block ciphers in CBC mode and motivate the usage of RC4-like ciphers. We describe RC4, the predecessor of Spritz, and illustrate some attacks against it.

The main parts of this thesis are the cryptanalytic results on Spritz, given in Chapter 5. We start our analysis with simple distribution tests for the registers in Spritz up to various correlation tests between Spritz states and registers. We analyze Spritz in context as a hash function as well as a stream cipher. For hash function attacks we search for collisions between internal states that lead to the same output. In context of a stream cipher we observed the cyclic behavior of Spritz, we look at possible weak key classes and we propose a state recovery attack.

Finally, we conclude the thesis in Chapter 6. Additionally, we discuss direction of future work.

2

Symmetric Key Primitives

In this chapter, we first introduce symmetric cryptography. Next, we describe the working principle of stream ciphers. In the end we show the design of sponge constructions and discuss some applications of their usage.

2.1. Symmetric Cryptography

In symmetric cryptography, the same cryptographic key is used for both, encryption and decryption. The cryptographic key is a shared secret that must be known to the two or more parties that want to communicate securely over an untrusted channel. Compared to asymmetric cryptography, where each party has its own keys, symmetric cryptography remains with a drawback, the key distribution problem.

Symmetric cryptographic encryption schemes can be categorized in two classes, block ciphers and stream ciphers. Stream ciphers typically encrypt a single bit at a time, where they require a same length keystream as the

plaintext. Block ciphers operate on blocks of data, using the same key for each block. There are several common modes for block ciphers: electronic codebook mode (ECB), cipher block chaining mode (CBC), cipher feedback mode (CFB) and output feedback mode (OFB).

Symmetric ciphers can be used to construct many different cryptographic primitives like hash functions, message authentication codes, pseudorandom number generators and authenticated encryption schemes.

2.2. Stream Ciphers

A stream cipher is a cryptographic algorithm that combines the plaintext with a keystream using the exclusive-or (XOR) function. Stream ciphers are symmetric encryption algorithms that are inspired by the only known unconditionally secure crypto system, the one-time pad. The only drawback of the one-time pad is that it requires a keystream with the same length as the plaintext. The huge key length makes it nonpractical for key distribution. Hence, stream ciphers can resolve this issue by generating a pseudorandom keystream from a shorter secret key (seed). The keystream of a stream cipher must be random for a computationally bounded adversary. Stream ciphers can be categorized into two types: synchronous stream ciphers and self-synchronous stream ciphers.

Synchronous stream ciphers. The pseudorandom keystream of a synchronous stream cipher (illustrated in Figure 2.1) is independent of the plaintext and ciphertext. Synchronous stream ciphers are for example E0 [e001] and SEAL [RC94].

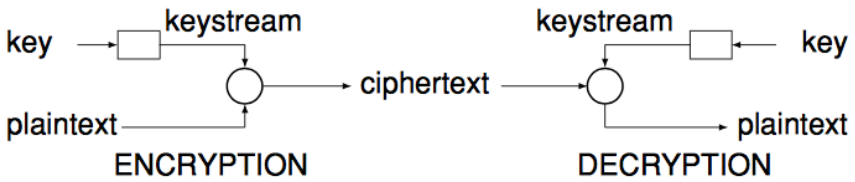

Figure 2.1.: A synchronous stream cipher is independent of the plaintext and ciphertext.

Self-synchronous stream ciphers. In a self-synchronous, or asynchronous, stream cipher (illustrated in Figure 2.2) the pseudorandom keystream depends on the secret key and on a fixed number of ciphertext bytes (that have already been produced). For instance, block ciphers in cipher feedback mode are self-synchronized stream ciphers.

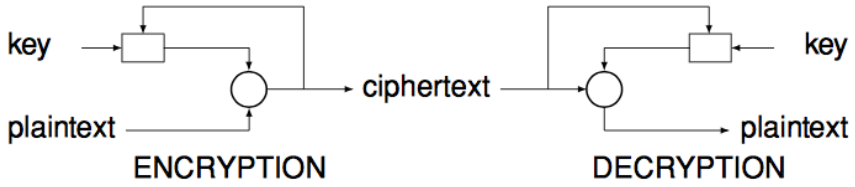

Figure 2.2.: A self-synchronous stream cipher depends on the secret key and ciphertext bytes.

2.2.1. Stream Cipher Designs

The design of stream ciphers can be categorized in software-oriented stream ciphers and hardware-oriented stream ciphers that are based on linear feedback shift registers.

The best-analyzed and most common software-based stream ciphers are RC4 and SEAL. There are some weaknesses in RC4, but it remains still secure if the beginning of the keystream is discarded [Miro2]. SEAL is a very fast stream cipher that is used for encrypting hard drives, but unfortunately it is patented.

Hardware-oriented stream ciphers are based on linear feedback shift registers (LFSR). Since linear feedback shift registers are easily analyzable and breakable, some schemes were proposed that increase the security. This can be achieved with nonlinear combinations of bits from the LFSR states or through combinations of several LFSR's. Another options are irregular clocking of the LFSR's. The most common hardware-oriented stream ciphers are A5/1 [a5187] and A5/2 [a5289] that are used in GSM mobile networks or E0 that is used in Bluetooth.

2.2.2. eSTREAM

The aim of the eSTREAM project [RB08] was to find a portfolio of stream ciphers that are well analyzed and can be used for widespread adoption. The portfolio has two profiles, one for hardware-based and one for software-based stream ciphers. The eSTREAM portfolio contains the following stream ciphers (see Table 2.1).

Profile 1 (Software)	Profile 2 (Hardware)
HC-128 [Wu08]	Grain v1 [HJM07]
Rabbit [BVP+03]	MICKEY 2.0 [BD08]
Salsa20/12 [Ber08]	Trivium [DCP08]
SOSEMANUK [BBC+08]	

Table 2.1.: eSTREAM

2.3. Sponge Constructions

A sponge construction [BDPVA14] is a mode of operation that maps a variable-length input to an arbitrary length output according to a transformation function $f : \{0,1\}^n \rightarrow \{0,1\}^n$. It is based on a fixed-length permutation and a padding rule for the input. Another similar construction is the duplex construction [BDPA11], which absorbs and squeezes in a different fashion. In the duplex construction, output is squeezed, immediately after one block is absorbed (with an application of the f function in between). The authenticated encryption mode of the duplex construction is called SpongeWrap [BDPVA14]. Both construction can model most cryptographic primitives including hash functions, message authentication codes, authenticated encryption, pseudorandom number generators and stream ciphers.

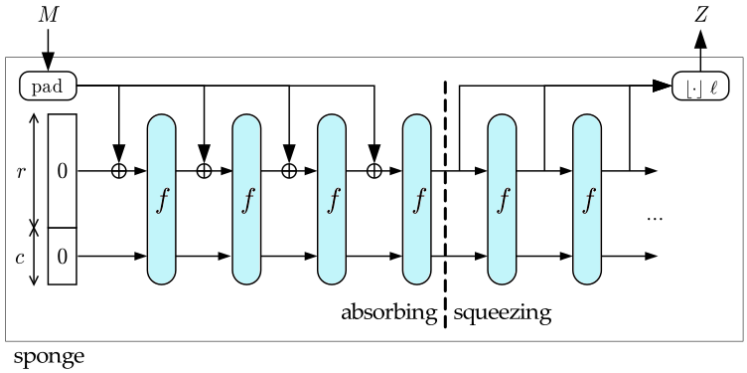

Figure 2.3.: Sponge construction

2.3.1. Construction

A sponge construction (as illustrated in Figure 2.3) is an iterated construction that builds on a permutation (or transformation) f that operates on a state S with size b. The state S is split into two parts, the bitrate r and the capacity c, that are concatenated together.

A sponge construction operates in two phases:

- **Absorbing phase:** First the input is padded and cut into r-bit blocks. Then the padded r-bit blocks are xored to the state alternating with an application of the transformation function f. After the whole input was absorbed, it changes to the squeezing phase.
- **Squeezing phase:** r-bit output blocks are returned alternating with an application of the transformation function f.

The capacity c is never directly linked to the input in the absorbing phase, neither it is directly outputted in the squeezing phase. The capacity parameter is therefore important for the security of a sponge function, while the bitrate defines the run time efficiency.

2.3.2. Applications

A sponge construction offers plenty use cases. It can model most cryptographic primitives like: stream ciphers, hash functions and authenticated encryption schemes.

Sponge as a stream cipher

A sponge construction can be easily used as a stream cipher (illustrated in Figure 2.4) by absorbing the key and an initialization vector. After that an arbitrary length keystream can be squeezed. For more details we refer to [BDPVA14].

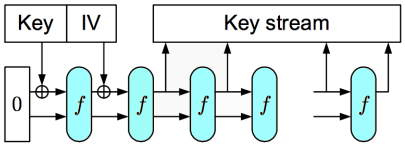

Figure 2.4.: Sponge as a stream cipher with key and initialization vector as input and the keystream as output.

Sponge as a hash function

To use a sponge construction as a hash function (illustrated in Figure 2.5), first the padded input message has to be absorbed. The hash value can be created by squeezing output bytes until the desired hash length is reached. For more details we refer to [BDPVA14].

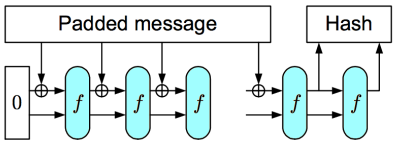

Figure 2.5.: Sponge as a hash function absorbs a variable-length input message and squeezes an arbitrary length hash value.

Sponge as a message authentication code

A sponge construction can be easily extended from a hash function to a message authentication code (as illustrated in Figure 2.6). As a message authentication code the sponge construction first absorbs the key, followed by the padded input message. Afterwards the message authentication code can be squeezed. For more details we refer to [BDPVA14].

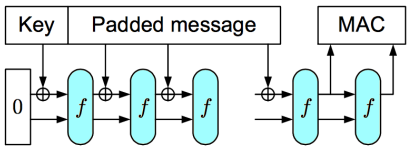

Figure 2.6.: Sponge as a message authentication code absorbs a key and the padded input message and squeezes the message authentication code.

3

Analysis of Symmetric Key Primitives

In this chapter, we describe some methods to cryptanalyze symmetric key primitives. We begin with generic attacks that are applicable to any primitive. Next, we describe statistical attacks, followed by attacks on stream ciphers. In the end, we discuss attacks on hash functions.

3.1. Preliminaries

3.1.1. Cryptanalysis

Cryptanalysis refers to the study of crypto systems, ciphers and ciphertexts to find hidden aspects or weaknesses without knowing the secret key. Cryptanalysis comes from the greek words, kryptós (i.e. ”hidden”) and analýein (i.e. ”to loosen”). The goal of crypt analyzing a cipher is to break the cipher,

find weaknesses or to proof the resistance of the cipher against attacks. There are many different techniques to analyze a cipher/crypto system, mostly depending on what information the cryptanalyst has. The methods to analyze a cipher reach from paper-and-pen methods to mathematically advanced computation with clusters and super computers.

3.1.2. Attack Model

Depending to what information an attacker has access, there exists numerous different attack models.

Chosen Ciphertext Attack. In a chosen ciphertext attack an encryption/decryption oracle is available to the attacker, with whom the attacker can encrypt/decrypt cipher texts of his choice. In this way the attacker can learn about the functionality of the cipher by observing plaintext and ciphertext pairs.

Chosen Plaintext Attack. The attacker can chose the plaintext that gets encrypted. Afterwards, the attacker can observe the resulting ciphertext. This mode is used for instance in differential cryptanalysis.

Known Plaintext Attack. The attacker has knowledge of a set of plaintext, ciphertext pairs. With this information the attacker tries to deduce the key to decrypt further ciphertexts.

Chiphertext-only Attack. The attacker knows several ciphertexts and tries to recover the plaintext or deduce the key. The attack succeeds if any information about the plaintext or the key could be gained.

3.1.3. Cryptographic Security

To measure the security or secrecy of a cipher we first need to define when a cipher is considered to be secure and when it is broken.

Unconditional Security or Perfect Secrecy. A cryptosystem offers perfect secrecy when it leaks no information about the plaintext by observing the ciphertext. It must not leak any information about the plaintext, even if the attacker has infinite computational power. Furthermore, this applies that the crypto system cannot be broken.

The only known cipher that offers perfect secrecy is the Vernam cipher (better known as one-time pad):

$$\text{ciphertext} = \text{plaintext} \oplus \text{keystream}$$

Perfect secrecy is only assured if the keystream is random and never used twice to encrypt a message.

Computational Security. A cryptosystem is called computational secure if the best-known attack requires 2^n operations, where 2^n is a very large number. In practice, an adversary (i.e. attacker) is computationally bound to limits of time, memory and data. Cryptographic schemes are designed to be secure against such adversaries. To calculate the security level for a cryptographic algorithm we first need to define how to measure those security level:

Defintion 1 (Big-O Notation):

$$\begin{aligned}\mathcal{O}(g(n)) = \{T(n) &: \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \\ &0 \geq f(n) \geq c \cdot g(n) \text{ for all } n \geq n_0\}\end{aligned}$$

Defintion 2 (polynomial time algorithm): An algorithm with input length n and for some constant c is called polynomial time algorithm, if its running time $f(n) = \mathcal{O}(n^c)$.

For a cryptosystem to be secure, the success probability for any polynomial time adversary to find a solution must be negligible. A cryptographic algorithm is called ”broken” if there exists an attack that is computationally faster than brute force. This does not automatically imply that such an attack is practical. Note that an attack is called practical, if it is possible to break the cipher with current computer technology.

3.2. Generic Attack Methods

In this section, we describe some general attack methods that can be applied to any cipher. Generic attacks treat ciphers as black boxes and do not exploit any weaknesses in a specific cipher. Therefore, they are applicable to any cipher. Furthermore, these attacks are used to define security level to ciphers regarding computational security.

3.2.1. Exhaustive Search

Exhaustive search is a brute force method, which involves searching the entire key space to find the correct key. This attack can be used when no weakness is known for the attacked cipher. The worst case complexity for exhaustive search is $\mathcal{O}(n) = 2^n$ for a key of n bits. To improve the search time an attacker can use heuristics that indicate where the solution is more probable. Ciphers should be designed to use a large enough key space to resist against brute force methods. Therefore, the bounds that are computable by current technologies can be used.

3.2.2. Time-Memory / Space-Time Tradeoff Attacks

In a space-time / time-memory attack an attacker reduces the execution time at the cost of memory or conversely. In cryptography an attacker can use these attacks to improve the time complexity by storing pre-computed results in lookup tables or by storing values in recursive algorithms.

3.2.3. Birthday Attack

The birthday attack is a mathematical attack that exploits the birthday paradox and pigeonhole principle. It can be used to find collisions with a complexity of $2^{n/2}$ for any function f with output size n. The attack can be used to attack the communication between two or more parties.

The birthday paradox tries to approximate the probability that in a set of n people, two of them have the same birthday. According to the pigeonhole principle, the probability is 50% with 23 people, 99.9% with 70 people and 100% with 367 people as illustrated in Figure 3.1.

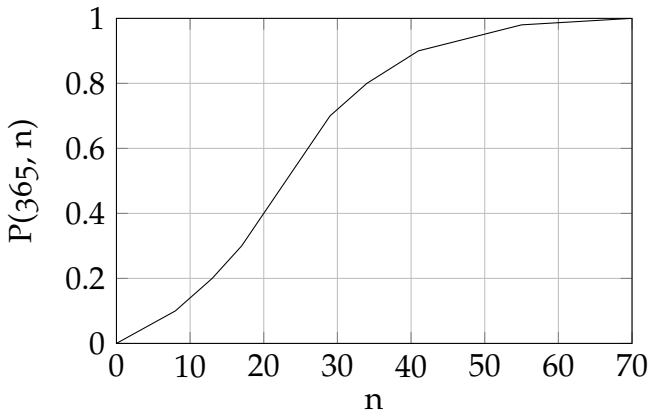

Figure 3.1.: Birthday paradox approximates the probability that in a set of n people, two of them have the same birthday.

This can be calculated by:

$$P(n,k) = 1 - \frac{n!}{(n-k)!n^k} \tag{3.1}$$

$$= 1 - [(1 - \frac{1}{n}) \cdot (1 - \frac{2}{n}) \cdot \cdots \cdot (1 - \frac{k-1}{n})] \tag{3.2}$$

If we assume that the following inequality holds:

$$(1 - x) \leq e^{-x} \tag{3.3}$$

then we have:

$$P(n,k) > 1 - [(e^{\frac{-1}{n}}) \cdot (e^{\frac{-2}{n}}) \cdot \cdots \cdot (e^{\frac{-(k-1)}{n}})] \tag{3.4}$$

$$> 1 - e^{\frac{-(k \cdot (k-1))}{2n}} \tag{3.5}$$

If we now want to know when the probability is higher than 50% we can set:

$$0.5 = 1 - e^{\frac{-(k \cdot (k-1))}{2n}} ln(2) = \frac{k \cdot (k-1)}{2n} \tag{3.6}$$

for large values of k we can approximate $k \cdot (k-1)$ by k^2:

$$k = \sqrt{2(ln(2))n} = 1,18 \cdot \sqrt{n} \tag{3.7}$$

The birthday paradox is a important factor for cryptanalyzing ciphers as the probability to find a collision is 50% with $2^{n/2}$ inputs.

3.2.4. Distinguisher

In a distinguishing attack an attacker observes the ciphertext and tries to find any pattern that distinguishes the encrypted output from the output of an ideal primitive. If such a pattern can be found the attacker can use it to recover the secret key. Ciphertext indistinguishability is a property of many modern ciphers.

3.2.5. Related Keys

Related key attacks were introduced independently by Knudsen [Knu92] and Biham [Bih93]. In a related key attack, an attacker uses two or more keys that are mathematically related and tries to observe a weakness in the cipher. Given keys K and $K^* \neq K$ an attacker choses a relationship f such that:

$$K^* = f(K)$$

The attacker knows the relationship of the keys, however the keys are unknown to the attacker. Using those related keys the attacker can try to observe the functionality of the cipher, or try to create collisions that lead to the same keystream, etc.

3.2.6. Weak Keys

A weak key is a specific key, that when used in a cipher, makes the cipher behave in an unexpected way. If for a set of weak keys, the cipher is much weaker, then this set is called a class of weak keys. Weak keys are an important factor for the security of ciphers and modern ciphers should be designed to avoid such keys. There are some ciphers that are vulnerable to weak keys (e.g. RC_4 [FMS01], DES [MS87], IDEA [Haw98]).

To find weak keys an attacker can do a so-called membership test for the class. We can assume that for a key space of n bits, there can be 2^n possible keys if an attacker tries to find the key using exhaustive search. If there exist a class of weak keys of size 2^w that can be found within a membership test with complexity of 2^m. Then if $m < w$ the class is a weak key class with a complexity of less than exhaustive search. This can lead to attacks on the cipher with faster attacks then exhaustive search for keys in the weak key class.

3.3. Statistical Attacks

Statistical attacks exploits weaknesses in the underlying cryptosystem. Therefore, many mathematical tests can be used like distribution tests, pairs of letter tests, correlation tests and plenty other statistical tests.

A popular example for a statistical attack is frequency analysis, where an attacker analyzes the frequency of each letter and then maps the most frequent letter in the observed ciphertext to the most common letter in the language of the plaintext.

3.3.1. Pearson's Chi-Squared Test

Pearson's chi-squared test is a statistical test that evaluates the likelihood of a difference between two sets. It can be used for two types of comparison: tests of goodness of fit and tests of independence. In the goodness of fit test the deviation from uniformity of a distribution is measured.

In cryptography, the test can be used to find statistical abnormalities and biases in encrypted data that show weaknesses in a cipher.

The chi-squared statistic can be calculated by:

$$\chi^2 = \sum_{u=0}^{q-1} \frac{(O_u - E_u)^2}{E_u}$$

given E_u as:

$$E_u = \frac{T}{q}$$

χ^2	...	chi squared statistic
O_u	...	number of outcomes observed of type u
E_u	...	expected number of outcomes of type u (should be at least 5)
T	...	total number of trials

The expected value for the chi-squared test is the number of degrees of freedom. The number of degrees of freedom, $\#df$, is equal to the number of elements in the test set, subtracted by the reduction in degrees of freedom (which depends on the distribution, e.g. uniform distribution = 1). The standard deviation can be calculated as $\sqrt{2 \cdot \#df}$.

A test fails if the chi-squared statistic is larger than four times the standard deviation above its expected value.

3.3.2. Correlations

Pearson's correlation coefficient is used to measure the dependence of two datasets. In cryptography, an attacker can correlate the plaintext with the ciphertext to find weaknesses in the cipher (e.g. weak keys). It can also be used to measure the similarity of encrypted data to find related keys.

Pearson's correlation coefficient r_{xy} can be calculated by:

$$r_{xy} = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{(n-1)s_x s_y} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}$$

The correlation coefficient r_{xy} varies between -1 and 1 whereby -1 shows a strong negative correlation and 1 shows a strong positive correlation. If r_{xy} is 0 this indicates that there is no correlation between the two measured datasets.

3.4. Stream Cipher Attacks

Stream ciphers (as described in Section 2.2) normally build upon the one-time pad, where the plaintext is combined with a keystream regarding the exclusive-or function (XOR). The security of stream ciphers depends on a few constraints. The keystream, which is xored to the plaintext, has to be random, and should never used twice.

3.4.1. Reuse Key

In a stream cipher the same key should never be used more than once, to encrypt two or more messages. If we encrypt two messages m_1 and m_2, both encrypted with the same key K. The stream cipher produces a keystream C with the same length as the messages, which can be XORed to each message. The encrypted messages are:

$$c_1 = m_1 \oplus C$$

$$c_2 = m_2 \oplus C$$

If an adversary now intercepts both encrypted messages c_1 and c_2 he can compute:

$$c_1 \oplus c_2 = (m_1 \oplus C) \oplus (m_2 \oplus C) = (m_1 \oplus m_2) \oplus (C \oplus C) = m_1 \oplus m_2$$

because the XOR function is commutative (i.e. x $\oplus$ y = y $\oplus$ x) and self-inverse (x $\oplus$ x = 0). If the two intercepted messages are of different length, an adversary can only recover information, until the end of the shorter message.

3.4.2. Cycles in the Keystream

The generation of random values is quite difficult. Stream ciphers are pseudorandom number generators (PRNG) that generate a keystream. After a while the pseudorandom number generator returns the same values again and runs in a cycle. The size of such a cycle depends on the internal state of the stream cipher, as well as the algorithm itself. The internal state should be chosen large enough that small cycles do not occur. However, this alone does not guarantee that there are no short cycles in the keystream.

A well known example of cycles were discovered by Hal Finnley [Fin94], in 1994, on RC_4.

3.4.3. State Recovery

In a state recovery attack, an attacker tries to recover the internal state of a cipher. To recover an internal state the attacker takes advantage of known values for any variable of the cipher and the known output keystream. Additionally, an attacker may assume that some parts of the internal state are known (i.e. with an previous other attack, according to some biases in the output keystream or with the usage of some heuristics).

If the next-state function (UPDATE) is easily invertible (which is the case for example in RC4), an attacker can recover the initial state if any internal state is known. Furthermore, with the knowledge of an internal state, the attacker can generate further output keystream bytes if the attacker is in possession of an encryption oracle. If a state recovery attack succeeds, the cipher can be broken complete. Most ciphers use a large internal state to prevent against state recovery attacks, so that e.g. exhaustive search or time/memory tradeoff attacks of an internal state are infeasible.

3.4.4. Key Recovery

In a key recovery attack, an attacker tries to recover the cryptographic key of a crypto system. One has to differ between two types, key recovery from the keystream and key recovery from an internal state. The first one uses the information from the known output keystream values to recover the key. The second one builds upon a state recovery attack that recovers an internal state, or from a known internal state. To make exhaustive key-search over the key space infeasible, modern ciphers chose a large enough key space (e.g. key space with $>$ 96-bit and larger).

3.5. Hash Function Attacks

A cryptographic hash function maps an arbitrary input message to a fixed-length hash value. For a secure hash function it should be computational infeasible to find two messages that map to the same hash value. As a hash

function is defined as a one-way function, it should also be infeasible to find the input from a given hash value.

A cryptographic hash function must have several properties to be secure: collision resistance, pre-image resistance and second pre-image resistance.

3.5.1. Collision Attack

In a collision attack, the attacker tries to find two messages that result in the same hash value, i.e. a collision occurs. Mathematically, a collision attack can be described by choosing a message m_1 and a second message m_2 that result in the same hash value, i.e. hash(m_1) = hash(m_2) but $m_1 \neq m_2$. According to the birthday attack collisions can occur with complexity of $2^{n/2}$. Figure 3.2 illustrates a hash collision attack.

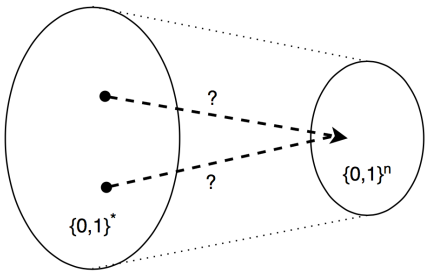

Figure 3.2.: A hash collision attack, i.e. hash(m_1) = hash(m_2) but $m_1 \neq m_2$.

3.5.2. Pre-image Attack

In a pre-image attack, the attacker tries to find a message for a specific hash value, i.e. the pre-image of the hash value. It should be infeasible to find a message, that is hashed to the given value, i.e. given hash value y = hash(x), find pre-image x. Pre-image resistance is given up to complexity of 2^n. Figure 3.3 shows a pre-image attack.

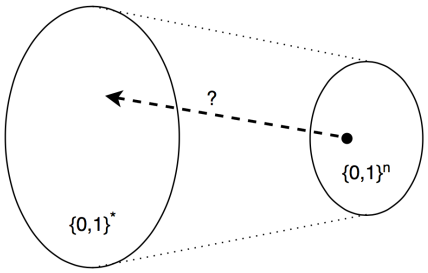

Figure 3.3.: A pre-image attack, i.e. given hash value y = hash(x), find pre-image x.

3.5.3. Second Pre-image Attack

In a second pre-image attack, the attacker tries to find a second message m_2 that results in the same hash value, than a given message m_1, i.e. find m_2 for a given message m_1 such that hash(m_1) = hash(m_2) but $m_1 \neq m_2$. Second pre-image resistance is given up to complexity of 2^n. Figure 3.4 illustrates a second pre-image attack.

If a hash function is collision resistant to a given bound, this implies it is also second pre-image resistant to that bound. However, this is not valid for pre-image resistance.

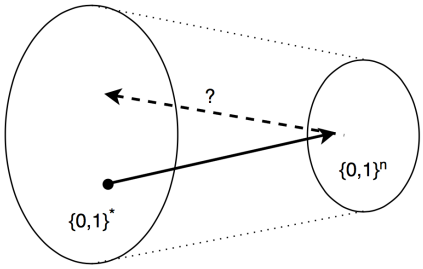

Figure 3.4.: Second pre-image attack, i.e. find m_2 for a given message m_1 so that hash(m_1) = hash(m_2) but $m_1 \neq m_2$.

4

Spritz: A RC4-like Stream Cipher and Hash Function

In this chapter, we give an introduction to Spritz. We motivate the design of Spritz with some flaws in the cipher block chaining (CBC) mode, which leads to several destructive attacks on SSL/TLS. As result of these attacks RC4 was highly recommended. Next, we give an overview of RC4, which is the predecessor of Spritz. Afterwards, we highlight the current cryptanalytic results on RC4 that show the need of an improved version of RC4. Finally, we give a detailed description of Spritz.

4.1. Motivation

In the last few years, there have been some devastating attacks against SSL/TLS. Most of them exploited flaws in the block cipher mode, cipher block chaining mode, others aimed at implementation errors.

BEAST. In 2011, Thai Duong and Juliano Rizzo [DR11] demonstrated an attack called BEAST (Browser Exploit against SSL/TLS) exploiting a long known vulnerability in the cipher block chaining mode of block ciphers. The vulnerability was discovered in 2002 by Phillip Rogaway [Rog02], but no practical attack has previously been published. Since RC4 as a stream cipher is not vulnerable to BEAST, it was recommended for use in TLS in spite of its shortcomings.

Lucky 13. In February 2013, Nadhem J. AlFardan and Kenneth G. Paterson [AFP13] published an attack called Lucky 13 that allows a man-in-the-middle attack when the cipher block chaining mode in TLS is used. The attack uses a timing side channel of the TLS error messages in combination with the authentication part of TLS. Using RC4 can mitigate the Lucky 13 attack.

POODLE. Recently, in September 2014, Bodo Moeller et. al. [MDK14] published the POODLE attack (Padding Oracle On Downgraded Legacy Encryption). POODLE exploits the fallback of client software and attacks SSL v3.0. It therefore uses the absence of a specification of the padding bytes in SSL v3.0 in combination with the CBC mode. In December 2014, Adam Langley [Lan14] improved the attack such that the fallback to SSL v3.0 is not needed anymore.

All those attacks aimed at block ciphers in CBC mode. To overcome these attacks, the usage of the stream cipher RC4 in SSL/TLS was recommended. Unfortunately, RC4 has some security flaws, which lead to the point that RC4 is weaker than previously thought and should not be used either. The design of RC4 is widely analyzed and gives a good starting point for a new algorithm. In the progress of finding a successor of RC4 many improved designs have been proposed. Recently, the designer of RC4, Rivest in cooperation with Schuldt proposed a new version, called Spritz.

4.2. Description of RC4

The RC4 stream cipher was designed by Ronald L. Rivest in 1987. It is a software-oriented stream cipher, which is extremely fast and has a simple design. In contrast to other keystream generators, which consist of linear feedback shift registers (LFSR's), RC4 is a software based stream cipher that is based on a permutation of 2^n n-bit words (the default value for n = 8, so that RC4 operates on bytes) and two 8-bit registers. The design of RC4 was for a long time kept as a trade secret, until it was anonymously leaked to the cypherpunks mailing list [Ano94] in 1994. RC4 is one of the most widely deployed stream ciphers and is implemented in various software applications and standardized protocols like SSL/TLS. Moreover, it is used in wireless networks within the WEP and WPA protocols and in commercial products like Microsoft Lotus, Oracle Secure SQL, Microsoft Windows, and many others.

The internal state of RC4 consists of a permutation table $S = S[l]_{l=0}^{2^n-1}$ of 2^n n-bit words and two n-bit pointers i and j (the default value for $n = 8$). The cipher is initialized with a variable length key K (with size typically between $40 \ldots 256$ bit) during the key-scheduling algorithm (KSA). The internal state size is given by $log_2(2^n!) + 2n$ that is calculated by the possible number of states for the permutation S and the two pointers i and j. After the KSA is finished, keystream can be generated by calling the pseudo-random generation algorithm (PRGA).

4.2.1. KSA: Key-scheduling algorithm

The key-scheduling algorithm initializes the permutation S with the identity permutation and sets both pointers to 0. Afterwards j is updated over the size of the permutation S, where it is influenced from its previous value, the permutation S and the key K. The permutation S is updated in each step by swapping one value with another one. The pointer i runs over all possible values from $0 \ldots 256$ so that each element in S is swapped at least once. Listing 4.1 provides pseudocode of the KSA.

```
1 for i = 0 to 255
2   S[i] := i
3 end for
4 j := 0
5 for i = 0 to 255
6   j := (j + S[i] + K[i mod K.length]) mod 256
7   SWAP(S[i], S[j])
8 end for
9 i := j := 0
```

Algorithm 4.1: KSA is the key-scheduling algorithm, which initializes i, j and the permutation S based on key K

4.2.2. PRGA: Pseudorandom generation algorithm

The pseudorandom generation algorithm updates the internal state of RC4 for each step. The pointer i is incremented by 1 each step, where j is incremented by the value of permutation S at position i. Afterwards, the values of the permutation S are swapped at the updated pointers i and j. Finally, at each step a keystream byte is generated from the value in permutation S at the index of the sum $S[i] + S[j]$. All operations in RC4 are done modulo 2^n. Algorithm 4.2 provides pseudocode of the PRGA.

```
1 i := (i + 1) mod 256
2 j := (j + S[i]) mod 256
3 SWAP(S[i], S[i])
4 Z = S[(S[i] + S[j]) mod 256]
5 output Z
```

Algorithm 4.2: PRGA is the pseudo-random generation algorithm, where at each step the state is updated and one output byte is produced

4.3. Cryptanalytic Results on RC4

Since RC4 was designed in 1987 it has become a widely used popular stream cipher. It has been well analyzed over the years, but until now no

practical attacks have been found. Despite some flaws in the first iterations of the output stream, RC4 is still secure, if the first r output bytes are discarded [Mir02].

4.3.1. Biases and Distinguishers

Most of the cryptanalysis on RC4 focuses on biases found in the key stream or distinguishers that differentiate the pseudorandom output of RC4 from a random oracle. The first results were published by Roos [Roo95], in 1995, who discovered that the permutation S is highly correlated with the key. Fluhrer, Mantin and Shamir [FMS01] published weak keys and recommended to discard the first N output bytes of RC4. In 2001, Mantin and Shamir [MS01] found a bias in the second output key stream byte which occurs with probability $2/N$ instead of $1/N$. This leads to a distinguishing attack. In 2013, AlFardan et. al. [ABP+13] published new biases against RC4 which lead to an attack on TLS, with only 2^{24} connections/sessions to reliable recover plaintext bytes.

4.3.2. Key Collisions

In 2009, Matsui [Mat09] found colliding key pairs for 24-byte keys. Chen and Myaji [CM11] improved the attack and found colliding key pairs for shorter keys (22-byte keys). These keys can be used to create collisions in the internal state of RC4.

4.3.3. Key Recovery

For key recovery, one has to distinguish between key recovery from an internal state and key recovery from the key stream. Key recovery from an internal state was first published by Paul and Maitra [PM07]. Further improvements were published by Biham and Carmeli [BC08] and Khazei and Meier [KM08].

Key recovery from the key stream is mostly done on WEP or WPA. First results were done by Fluhrer et. al. [FMS01]. The attack was improved by Klein [Kle08]. Sepehrdad et. al. [SVV11] published their results in WPA with a complexity of 2^{96}.

4.3.4. State Recovery

There are several papers on determining the internal state from a known output key stream. The first one is from Knudsen et. al. [KMP+98] which can recover the initial state of RC4 with complexity of 2^{779}. Maximov and Khovratovich [MK08] have published an improved algorithm which uses patterns in the key stream and leads to a complexity of 2^{241} under plausible assumptions. Another probabilistic algorithm was published by Golic and Morgari [GM08] with a complexity of 2^{689}. The complexity of each those attacks is far beyond practical complexities. Therefore, it is no real threat to RC4. Nevertheless, a practical state recovery attack would allow an adversary to generate further output without knowing the secret key, or to recover the initial state of RC4 (which can be used for a key recovery attack).

4.4. Description of Spritz

Spritz [RS14] was recently presented at the rump session of CRYPTO 2014. It was designed by Rivest and Schuldt. The former one was also the designer of RC4. Spritz was proposed as an improved variant of the stream cipher RC4 and follows similar design principles. Additionally, Spritz is formulated as a sponge-like function, which offers the typical sponge functionalities. Spritz can be used as stream cipher, hash function, message authentication code or for authenticated encryption.

Spritz is build upon RC4 and fixes many weaknesses. It was designed as drop-in replacement of RC4 and should keep the simple design. Therefore, Spritz as well as RC4 builds upon a permutation and a few pointers (i.e. 8-bit registers). Spritz is the result of extensive simulations and statistical

analysis. The designers created the Spritz functions by basically generating all possible candidates, constrained to six registers and a permutation of size N, and chose the ones with the best security properties. The design was done in a huge computer cluster and with about five months of computation time, according to the designers [RS14].

4.4.1. Spritz Spezification

Spritz offers several functions as described in Figure 4.1. Moreover, it is build upon a sponge function as proposed by Bertoni et. al. [BDPVA14]. As it is not a sponge function by definition it should be considered as sponge-like or spongy.

Permutation. Spritz uses a permutation $S = \{0, 1, \ldots, N-1\}$ with N elements.

Registers. Spritz uses 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, similar as in RC4. Register w is always relative prime to N and is used to update register i. The update of register i causes it to cycle between all values modulo N. Register a denotes the number of nibbles that have been absorbed. Register z stores the last generated output key stream value.

State. The state of Spritz consists of the six registers and the permutation S. Spritz has a maximum of:

$$N! \cdot N^6$$

possible states. For the default value of $N = 256$ this leads to $log_2(N! \cdot N^6) \approx 2^{1730}$ states.

Key. The key in Spritz is an arbitrary length byte array K[0, ..., L-1] where L denotes the length of the key. The default values for L are $\langle 16,\ 32 \rangle$, resulting in key sizes of 128 and 256 bits, respectively.

Capacity. As Spritz is sponge-like the capacity of Spritz is relevant for the security analysis. The capacity of Spritz is defined as:

$$c(N) = (\lceil N/2 \rceil - D) \cdot log(N), \text{ where } D = \lceil \sqrt{N} \rceil$$

(for $N = 256$ the capacity yields to $c(N)$ = 936 bits).

4.4.2. Sponge Functions

A detailed description is given in Figure 4.1.

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 (i.e. half bytes = 4-bit), 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.

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

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

AbsorbByte(b)
1  AbsorbNibble(Low(b))
2  AbsorbNibble(High(b))

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

AbsorbStop()
1  if a = ⌊N/2⌋
2     Shuffle()
3  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 = 0 to r − 1
2     Update()
3  do w := w + 1
4  until Gcd(w, N) = 1

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

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

Drip()
1  if a > 0
2     Shuffle()
3  Update()
4  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
```

Figure 4.1.: Pseudocode of Spritz. All additions are modulo N. When N is a power of 2, the last two lines in Whip are equivalent to $w = w + 2$.

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 (Figure 4.2 illustrates Crush).

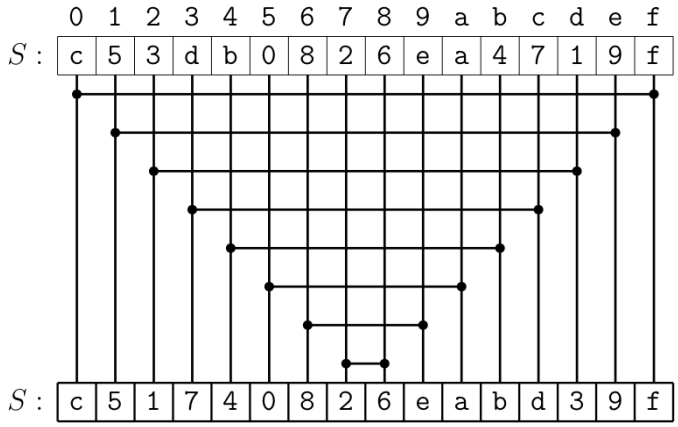

Figure 4.2.: Crush compares pairs from the beginning to the end and sorts ascending

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.

4.4.3. Applications

As Spritz is inspired by the sponge construction, it can be used as stream cipher, hash function, message authentication code (MAC) or also for authenticated encryption with associated data (AEAD).

Stream cipher. When Spritz is used as stream cipher the KeySetup is called which absorbs the key. Afterwards, a key stream is generated as long as the message that should be encrypted. Then each byte of the message is added to the corresponding byte of the key stream, which yields to the ciphertext. Algorithm 4.3 to 4.5 illustrates Spritz as a stream cipher.

Encrypt(K, M)

```
1 KeySetup(K)
2 C = M + Squeeze(M.length)
3 return C
```

Algorithm 4.3: Encrypt function

Decrypt(K, C)

```
1 KeySetup(K)
2 M = C − Squeeze(M.length)
3 return M
```

Algorithm 4.4: Decrypt function

KeySetup(K)

```
1 InitializeState()
2 Absorb(K)
```

Algorithm 4.5: KeySetup function

Hash function. Spritz can also be used as hash function, which can produce hash values of arbitrary length. The hash function first absorbs the message. Next, it calls AbsorbStop, which can be seen as message padding. Then an integer r with the desired hash length is absorbed. The hash length is absorbed so that e.g. a 16-byte hash is not a prefix of a e.g. 32-byte hash. Algorithm 4.6 illustrates pseudocode of the hash function.

Hash(M,r)

```
1 InitializeState()
2 Absorb(M); AbsorbStop()
3 Absorb(r)
4 return Squeeze(r)
```

Algorithm 4.6: Hash function

Message Authentication Code (MAC). The following pseudo code (see Algorithm 4.7) illustrates how Spritz can be used as Message Authentication Code. First, the key is absorbed, afterwards it works as a hash function.

MAC(K,M,r)

```
1 InitializeState()
2 Absorb(K); AbsorbStop()
3 Absorb(M); AbsorbStop()
4 Absorb(r)
5 return Squeeze(r)
```

Algorithm 4.7: MAC function

Authenticated Encryption with Associated Data (AEAD). Spritz can be used as authenticated encryption function with associated data. It takes as inputs the key K, a nonce Z, a header H (which is the associated data) and a message M. The function returns the ciphertext as well as an r-byte authentication tag τ.

AEAD(K,Z,H,M,r)

```
1  InitializeState()
2  Absorb(K); AbsorbStop()
3  Absorb(Z); AbsorbStop()
4  Absorb(H); AbsorbStop()
5  Divide M into blocks M_1, M_2, ..., M_n, each N/4 bytes long
       except possibly the last.
6  for i = 1 to t
7    output C_i = M_i + Squeeze(M_i.length)
8    Absorb(C_i)
9  AbsorbStop()
10 Absorb(r)
11 output τ = Squeeze(r)
```

Algorithm 4.8: AEAD function

4.4.4. Performance

The designers of Spritz give some insights in the performance of their algorithm in the proposal.

Table 4.1 compares Spritz to its predecessor RC4 as well as to the eSTREAM candidate Salsa20 and AES-CTR. In the first column (Squeeze) the amount of output data is measured, in cycles per byte. The second column (encryption of short packages) measures the key setup time by encrypting a 512-byte message with a 16-byte key.

Table 4.1.: Performance of Spritz as a stream cipher compared to other stream ciphers

Primitive	Squeeze	Encryption of short packages
Spritz	19 c/b	56 c/b
RC4	6 c/b	13 c/b
Salsa20	6 c/b	7 c/b
AES-CTR	12 c/b	12 c/b

Table 4.2 shows the performance of Spritz as a hash function. Spritz is very slow compared to Keccak, or SHA-256. The designers claim that the goal of

Spritz should be the simplicity and easy implementable design and not its performance.

Table 4.2.: Performance of Spritz as a hash function compared to other hash functions

Primitive	Absorb
Spritz	408 c/b
Keccak	11 c/b
SHA-256	14 c/b

5
Cryptanalysis of Spritz

In this chapter, we first motivate the need and the purpose of an analysis of Spritz. We begin with our results from generic attacks on Spritz. Afterwards, we continue with our results on statistical attacks, such as distribution tests and correlation tests. Moreover, we discuss our results on the analysis of Spritz when used as a hash function. We conclude this section with the results on the analysis of Spritz, when used as a stream cipher, such as the cycle structure of Spritz and state recovery attacks.

5.1. Motivation

Spritz is a recently proposed stream cipher and hash function, as a drop-in replacement for the widely used stream cipher RC4. In 2013 [ABP+13], some new attacks were published on RC4 that require only 2^{24} connections. The authors of Spritz, Rivest and Schuldt, claimed their security bounds based on various statistical tests, but did not published a detailed security analysis

of Spritz, in their proposal. 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 important that a successor provides tight security bounds against all possible attack vectors.

5.2. Generic Attacks

In this section, we discuss some generic attacks on Spritz. First, we searched for weak key classes in Spritz. For this we used some common weak key patterns and observed how the output varies if these weak key patterns are applied. Next, we investigate how the state randomization function Shuffle changes the number of possible states. The number of possible states is interesting for example in collision attacks. Furthermore, we studied if there are any anomalies with the sign of the permutation. We conclude this section by studying state rotation of Spritz, which can be used to find cycles in Spritz. This can be used for state recovery attacks and again also to find two messages that lead to an internal collision.

5.2.1. Weak Keys

A typical test, when analyzing a new cipher, is to look if there are any weak key classes. If there are any weak key classes, then we may find also a vulnerability in the key schedule. This can be used in further attacks. In Spritz the key length is 128 to 256 bits, so the key space is too large to try all keys to find weak key classes. Instead, we generate a huge amount of keys, with specific patterns, and applied our tests on those keys. Moreover, we tried various correlations between keys and the output to see if there are any weaknesses.

We applied several different tests for weak keys. Some of the tests are simple tests, where we absorbed a key with an increasing byte order and looked if the output has the same byte order. We also tested keys that are prefixes of other keys, where one byte in the end of the second key is different. Additionally, we tested some weak key patterns like keys

with all $0x00$, $0xFF$, $0x55$ and others. These patterns are used for the whole key (e.g. key = $\{0xFF, \ldots, 0xFF\}$). We also rotated such a pattern through a key with all constant values (e.g. key = $\{0xFF, 0x00, \ldots\} \rightarrow key = \{0x00, 0xFF, 0x00, \ldots\}$) or we applied various patterns alternating through the key (e.g. key = $\{0xFF, 0x55, 0xFE, 0xEF, \ldots\}$).

Our results are listed in Table 5.1.

Table 5.1.: Results on weak key tests

Test	N	Input	Results
k_1 is a prefix of k_2	8 16 32 64 128 256	2^{20} 16/17-byte key pairs where k_1 is a prefix of k_2	32825 correlated outputs 8182 correlated outputs no correlated outputs no correlated outputs no correlated outputs no correlated outputs
increasing input increasing output	8 ... 256	2^{20} increasing random length keys	no correlated outputs
decreasing input decreasing output	8 ... 256	2^{20} decreasing random length keys	no correlated outputs
weak key patterns (whole key)	8 ... 256	keys from 16 to 32 byte length	no correlated outputs
weak key patterns (alternating)	8 ... 256	keys from 16 to 32 byte length	no correlated outputs
weak key patterns (rotational)	8 ... 256	keys from 16 to 32 byte length	no correlated outputs

5.2.2. Analysis of Spritz States

The internal state Q_t in Spritz at time t consists of the six registers i, j, k, w, z and a as well as the permutation S. Therefore, the maximum number of states in Spritz is

$$\#states = \underbrace{N^6}_{registers} \cdot \underbrace{N!}_{permutation}$$

For $N = 256$ this leads to $N^6 \cdot N! \approx 2^{1730}$ states. While absorbing input and updating the permutation S does not effect the number of states, Crush with its many-to-one mapping does effect the number of states. Crush provides a non-invertible transformation where it maps $2^{N/2}$ states to one. Therefore, we looked in our tests how much information is lost after applying Crush. After Crush, Whip gets called, which again increases the number of possible states due to the different values in registers i, j and k and the call to Update in Whip.

In our tests, we additionally looked at the registers of Spritz. To find internal collisions we need to find two inputs that somehow lead to the same internal state. If we achieve to find two inputs that lead to the same permutation S, we therefore also have to check if the registers i, j, k, z, w and a are the same.

Our results show that if we start with different random permutations then we can observe that for all N, the registers stay nearly the same after one application of Whip (Note that Crush did not change the registers). If we additionally look when the whole state (i.e. registers and permutation) stays the same, we can observe that for $N = 8$ there are about 266 equal states after Whip and about 1716 equal states after Crush. However, for larger $N > 8$, we could not observe any equal states. The different tests are summarized in Table 5.2.

5.2.3. Sign of Permutation

In 2002, Mironov [Miro2] published an anomaly in the RC4 key schedule whereby it is possible to correctly guess the sign of the RC4 permutation S

Table 5.2.: Results on state tests

Test	N	Input	Results	
equal registers (i, j, k, z, w and a)	8	2^{20} random initial states	$2^{20.000}$ equal registers	
	16		$2^{20.000}$ equal registers	
	32		$2^{19.999}$ equal registers	
	64		$2^{19.994}$ equal registers	
	128		$2^{19.977}$ equal registers	
	256		$2^{19.907}$ equal registers	
			after Whip	after Crush
equal states (permutation + registers)	8	2^{16} random initial states	266 equal states	1716 equal states
	16		no equal states	no equal states
	32		no equal states	no equal states
	64		no equal states	no equal states
	128		no equal states	no equal states
	256		no equal states	no equal states

with probability 56%. Since Spritz also uses a permutation, whose entries are swapped in the next state function of Spritz, we applied the same analysis.

The sign of a permutation π, which is represented as a product of non-trivial transpositions (i.e. $\pi = (a_1 b_1)(a_2 b_2) \dots (a_m b_m)$), is defined as:

$$sign(\pi) = (-1)^m = \begin{cases} +1 & \text{if } 2 \mid m \\ -1, & \text{otherwise.} \end{cases}$$

Equal to RC4 the permutation S in Spritz is initialized with the identity permutation. Therefore, after InitializeState the sign (S) = +1, which is called in the beginning of Spritz. As the sign of the permutation is defined over the parity of the permutation it can also be defined over the number of transpositions in the decomposition (i.e. m) of the permutation S.

Each Swap in each iteration changes the sign of the permutation, unless the pointers i and j are the same. If we assume that $i = j$ with probability $1/N$ we have an advantage in guessing the correct sign over random guessing. We can calculate the probability of the sign being odd or even, which is

$(1-1/n)^n \approx e^{-1}$. The distribution of the two possible values for sign (S) was calculated by [Miro2]:

$$\begin{aligned}\Pr[sign(S) = (-1)^N] &= \left(1-\frac{1}{N}\right)^N + \binom{N}{2}\left(1-\frac{1}{N}\right)^{N-2}\frac{1}{N^2} + \cdots + \frac{1}{N^N} \\ &= e^{-1}\left(1+\frac{1}{2!}+\frac{1}{4!}+\ldots\right) \xrightarrow[N\to\infty]{} \frac{1}{2}(1+e^{-2})\end{aligned}$$

$$\Pr[sign(S) = (-1)^{N-1}] = 1 - \Pr[sign(S) = (-1)^N] \xrightarrow[N\to\infty]{} \frac{1}{2}(1-e^{-2})$$

which results in an advantage of $1/2 \cdot e^{-2} \approx 6{,}7\%$ for guessing the sign correctly, over random guessing.

It should be noted that this is only valid under the assumption that N is large (i.e. the default value for N in Spritz is 256, which is large enough) and we also rely on the assumption that $i = j$ with probability $1/N$.

The attack on Spritz has some additional restrictions. If we absorb more than $N/4$ bytes, Shuffle is executed which contains a call to Crush. As Crush performs a unknown number of swaps due to its ascending ordering of the permutation S, we loose information about the number of swaps. Therefore, it is not possible to guess the correct sign of the permutation anymore. Additionally, our experiments have shown that for $N = 256$ the registers i and j are never the same. Hence, it is unfortunately not possible to use this attack on Spritz, because an attacker only has an advantage of guessing the sign of the permutation correctly, if the registers i and j are at least one time the same.

5.2.4. Partial State Rotations

In 2008, Indesteege and Preenel [IP08] published a collision attack on RC4-Hash, which is based on RC4. The attack exploits some fixed points and partial state rotations in the compression function.

Spritz also seems to be vulnerable to partial state rotations during Absorb. If we consider an internal state $S = \{s_0, s_1, \ldots, s_n\}$ and absorb a message block $M = \{m, m, \ldots, m\}$ with $m = (0x00)^n$ we can show that

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

This partial state rotations in Spritz are illustrated in Table 5.3.

Table 5.3.: Partial state rotations in Spritz during Absorb

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

Since there is a call to Shuffle after absorbing $N/2$ Nibbles these state rotations just 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 $x > 0$.

In the attack on RC4-Hash these state rotations can be used to create fixed points by applying 255 iterations of the compression function, which leads to the same state (i.e. initial state). A collision can then be found for RC4-Hash

by using two messages like.

$$M = P||M^{0,1}||\overbrace{M^{1,1}||\ldots||M^{1,1}}^{255}$$
$$M' = P||\overbrace{M^{0,0}||\ldots||M^{0,0}}^{255}M^{0,1}$$

The attack on RC4-Hash is illustrated in Figure 5.1.

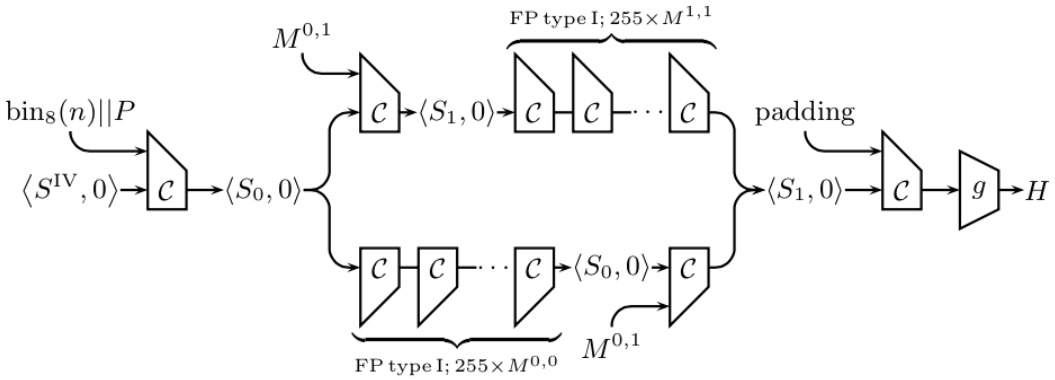

Figure 5.1.: Collision attack on RC4-Hash [IPo8].

We applied the same attack on Spritz, but due to the execution of SHUFFLE after absorbing $N/4$ bytes the attack is not possible in Spritz. In Spritz we are not able to create the two different messages that lead to the same internal state, since the state rotations only hold for the first $N/2$ bytes and then the whole state is mixed up in SHUFFLE.

5.3. Statistical Attacks

The designers of Spritz invested a huge amount of computing power and time to find the Spritz design. After their tests they chose the design of Spritz according to the functions that have the best choices for the update function used in Spritz. They applied various statistical tests, including distribution tests and correlation tests.

We additionally applied some statistical tests on Spritz to verify the results of the designers and for further testing. Our tests can be categorized in three classes. First, we applied some distribution tests, where we measured if the distribution of the registers or output values differs from a uniform distribution. Second, we looked at equal values in the output keystream with a fixed distance between the output bytes. Third, we applied various correlation tests between registers and permutations.

5.3.1. Distribution Tests

If Spritz behaves like a ideal primitive, its output values are completely random and distributed uniformly where each value occurs with probability of $1/N$. The registers in Spritz should also be uniformly distributed.

In our distribution tests we used Pearson's chi-squared test (described in Section 3.3.1) to measure whether the distribution deviates from a uniform distribution. We applied several tests on the output keystream of Spritz. This included tests, where we did not absorb any input, so SHUFFLE did not get called, or we absorbed some input so that SHUFFLE could randomize the state. Additionally, we looked at the distribution of successive output values as well as the individual values. In our simple output tests we squeezed a large amount of output and counted the occurrences of each value. In the test with the separate distribution we absorbed random input and just dripped one byte each time, which we used to measure the distribution of the output values.

Furthermore, we studied the distribution of the different Spritz registers. Register i is increased by w every time UPDATE gets called, and results in a perfect uniform distribution. Registers j and k should be pseudorandom. Our tests show that both register j and k also have a uniform distribution.

We also looked at some more advanced tests, where we observed the combined distribution of some registers. The designers or Spritz mentioned a bias in the $iz3z$ distribution of Spritz, which can be used in a distinguishing attack where 2^{81} calls to DRIP are required. In our tests we also observed some biases in the ijk, $iz2z$ and $iz3z$ distribution. This biases may be used in some more advanced attacks. Due to the large amount of plaintext that

are needed to show the biases, they may not be useful for practical attacks, but they give some new insights in the security of Spritz.

The results of our distribution tests can be found in Appendix A in Table A.1 to A.3.

5.3.2. Distant-Equality Tests

In December 2014, Zoltak [Bar14], published a statistical weakness of Spritz in the comparison to VMPC-R [Bar04], another RC4 like cipher, similar to Spritz. The weakness occurs in a distant-equality test, where the output stream of Spritz is observed and two words of the keystream are equal with distance k. This can be denoted by:

$$z_i = z_{i+k} \text{ for } k \in \{1, 2, \ldots, 8\}$$

where z_i denotes the output byte at position i and k is the distance between two output bytes.

Spritz shows a statistical weakness if the distance between two output bytes is two. In our tests we verified the results published by Zoltak. A bias occurs after observing 2^{26} samples for $N = 8$ and after 2^{40} samples for $N = 16$. More details of the result are shown in Table 5.4.

Table 5.4.: Results on distant-equality tests

Test	N	Iterations	Results
distant-equality	8	2^{26}	bias ($\approx$ 13 standard deviation)
	16	2^{40}	bias ($\approx$ 7 standard deviation)

5.3.3. Correlation Tests

In our correlation tests, we use Pearson's correlation coefficient (described in Section 3.3.2) to measure the dependence between two datasets.

We applied various correlation tests in our analysis. Therefore, we analyzed the correlation between the initial state and the internal state after absorbing

$N/4$ random inputs, which leads to a high correlation for $N > 128$. We also analyzed the behavior of Whip and Crush, which shows that after Whip there are no correlations, but after Crush the correlation increases again. When using Shuffle (i.e. two applications of Crush and three applications of Whip) there are no correlations compared to the initial state. Furthermore, we observed the correlation between random input and the corresponding output keystream.

The results of our correlation tests are highlighted in Table A.4 in Appendix A. The correlations results may be used in some more advanced attacks. Nevertheless, they give some new insights in Spritz.

5.4. Hash Attacks

As Spritz is designed as sponge-like it is possible to use all different application areas that a sponge construction offers. This enables us to use Spritz as a hash function. However, we need to consider some attacks on hash functions, i.e. collision, pre-image and second pre-image attacks.

5.4.1. Collision Attack

In a collision attack, an attacker tries to find two different messages m_1 and m_2 that lead to the same hash value $hash(m_1) = hash(m_2)$. Due to the birthday paradox (described in Section 3.2.3) it is always possible to find collisions after testing $2^{n/2}$ inputs.

To successfully create collisions we need to find two messages that create the same state and then lead to the same output. In this case, we came up with three different approaches. In our first approach, we try to cancel the differences directly during Absorb. Our second approach aims to cancel differences in the Update function. The third approach is to cancel the differences in Crush.

Collisions in Absorb. In our first approach, we try to cancel the differences between the messages directly when absorbing the message during the Swap operation in AbsorbNibble. In our attacks we aim to cancel differences introduced by message differences directly in the first Swap in AbsorbNibble. In AbsorbNibble the nibble count register a is swapped with the input nibble added to $N/2$ like Swap(S[a], S[$\lfloor N/2 \rfloor + x$]). Furthermore, we tried related key attacks where we used keys that are nearly the same, just with some slightly changed values and looked if the differences are canceled in Spritz.

Table 5.5.: Results on collision attacks in Absorb

Test	N	Inputs	Results
Collision attack in Absorb	$8 \ldots 256$	$256 \cdot \frac{N}{2}$ ascending inputs	no collisions found

Collisions in Whip. Our second approach aim to cancel differences introduced by different messages during one of the first calls to Update in Whip. If we absorb a message with just a few differences the Spritz state does not differ too much. We then can try to cancel the differences by swapping the different values. A similar approach is to reach a state, where for one message register i and register j are the same so that a call to Swap does not change the permutation. When for the second message now a swap is done and the same state is reached, we have found a collision. The attack to find colliding inputs that cancel during Whip is illustrated in Algorithm 5.1.

The results of our attack are summarized in Table 5.6. We performed the attack for $N = 8 \ldots 256$ for an ascending number of calls to Whip starting from 0 to $2N$. In our attack we absorb random input and look if we find a collision during one of the first calls to Update in Whip. Unfortunately, we have not found any collisions that occur in Whip.

Table 5.6.: Results on collision attacks in Whip

Test	N	Inputs	Results
Collision attack in Whip	$8 \ldots 256$	2^{24} random inputs	no collisions found

Algorithm 5.1 Collision attack in Whip

```
function CollisionAttackWhip()
    for #ofWhips = 0 to 2N do
        for i = 0 to 2^10 do
            InitializeState()
            Absorb(random input)
            Whip(#ofWhips)
            S_AfterWhip ← getPermutation()
            for j = 0 to 2^10 do
                InitializeState()
                Absorb(random input)
                Whip(#ofWhips)
                if S_AfterWhip == getPermutation() then
                    return "Collision found!"
                end if
            end for
        end for
    end for
end function
```

Collisions in Crush. In our third approach, we try to cancel differences between the two messages m_1 and m_2 during the ascending ordering in Crush. Our goal is to find two messages that are nearly the same after one application of Whip, where only a few positions in the permutation change so that these differences are canceled during the ascending sorting in Crush.

For example if we have two messages, where for the first message, Crush did not need to sort anything and for the second message, only the first and the last entry of the permutation are in different order, but otherwise the state is the same. If in the second message the first entry in the permutation is larger than the last entry then we have to sort just for one time and than reach the same state as with the first message. Afterwards the Spritz state is the same for both messages and we have a collision.

Our collision attack in Crush is described in Algorithm 5.2.

Algorithm 5.2 Collision attack in Crush

```
function CollisionAttackCrush()
    for m_1 = 0 to 256 · (N/2) do
        InitializeState()
        Absorb(m_1)
        Whip(2N)
        Crush()
        S_AfterCrush ← getPermutation()
        for m_2 = 0 to 256 · (N/2) do
            InitializeState()
            Absorb(m_2)
            Whip(2N)
            Crush()
            if S_AfterCrush == getPermutation() then
                return "Collision found!"
            end if
        end for
    end for
end function
```

We performed our attack for $N = 8 \dots 256$ but did not find any collisions that are canceled out in Crush. Our results for the different N are given in Table 5.7.

Table 5.7.: Results on collision attacks in Crush

Test	N	Inputs	Results
Collision attack in Crush	8 ... 256	$256 \cdot \frac{N}{2}$ ascending inputs	0 collisions found

5.4.2. Pre-image Attack

In a pre-image attack an attacker tries to determine the message, for a given hash value, that results in the desired hash value.

In our analysis, we performed a pre-image attack combined with a collision attack on Spritz, where we generated all 2^n possible messages m_1 for a given length and calculated until the beginning of Crush. Then we used a modified version of Crush, called descendingCrush, which compares the permutation S like Crush, but sorts only one time in descending order. After that we started our pre-image attack, where we used inverseWhip (i.e. this function reverses the Whip function), to calculate back to the state after the message has been absorbed. Next, we generate all possible messages m_2 up to $2^n/4$ (i.e. after $N/4$ messages Shuffle gets called, which contains the non-invertible function Crush). We applied inverseAbsorb for all messages m_2 and checked if the permutation equals the identity permutation. If we now got the identity permutation then we got a valid input, which leads to a pre-image in Crush (i.e. we just have to sort one more time in Crush). If we do not reach the identity permutation, it is not possible to generate the state with a valid input. The attack is highlighted in Algorithm 5.3.

Algorithm 5.3 Pre-image attack

```
function Pre-ImageAttack()
    for m_1 = 0 to 2^n do
        Absorb()
        Whip(2N)
        descendingCrush()
        inverseWhip(2N)
        for m_2 = 0 to 2^n/4 do
            inverseAbsorb()
            if permutation == identity permutation then
                return "Pre-image found!"
            end if
        end for
    end for
end function
```

We started our attack for small $N \leq 32$ but unfortunately, did not find the identity permutation for any of the inputs. Therefore, it is not possible with any of the tested valid inputs to reach such a state that leads to a pre-image in Crush. Additionally, we calculated the correlation between the identity

permutation and the output permutations after our pre-image attack to observe if we may achieve some near collisions. For $N = 16$ we found some high correlated permutations in ascending order, but they still diverged from the identity permutation.

5.4.3. Length-extension

A length-extension attack misuses a message authentication code (MAC) by applying additional input data and breaches the integrity of the MAC function. A vulnerable hash function outputs its internal state as the message digest. An attacker can then reconstruct the internal state according to the message digest and extend the message with additional data.

Spritz can easily be extended from a hash function to a message authentication code, by absorbing a key K before absorbing the message M. The authors of Spritz claim that Spritz is secure against length-extension attacks, since they call the state randomization function, SHUFFLE, before any output in SQUEEZE/DRIP is produced. As a consequence each input that has been absorbed, will be variously mixed before it will be squeezed. In this case it is not possible to use the output to determine the internal state of Spritz after absorbing message M.

5.5. Stream Cipher Attacks

Spritz is a redesign of the stream cipher RC4 with intention to replace RC4 due to better security and more application areas. Its main focus is still as a stream cipher and therefore, we have to consider various stream cipher attacks.

In our analysis, we studied the cyclic behavior of Spritz and searched for ”bad” starting states, which lead to small cycles in Spritz. Furthermore, we looked at state recovery attacks and describe several different state recovery attacks on reduced versions of Spritz.

5.5.1. Cycles

Spritz as a stream cipher generates a pseudorandom output keystream that is byte wise xored to a plaintext resulting in a ciphertext. As the keystream is pseudorandom, this implies that after some time the keystream repeats and runs in a cycle. If we want to use Spritz securely, it is important that this cycle length is very high for any N. We can distinguish between two possible sources where a cycle can be produced. The first one is the UPDATE function, which continuously updates the registers i, j and k and the permutation S. The second possibility to create a cycle is in DRIP, where additionally to UPDATE the OUTPUT function is called. The UPTDATE function changes the z register that is also used as feedback by updating itself. In the search for cycles in Spritz we implemented a multithreaded algorithm that is described in Algorithm 5.4.

The algorithm first generates all possible permutations for a given N, such that we can search for cycles for any given starting state. Next, we break down the whole list of possible permutations and with all threads we search for cycles. Afterwards, we start the FINDCYCLES function for each thread. In the FINDCYCLES function we initialize the permutation S with one of the generated permutations and execute the DRIP function until we reach our starting state again (i.e. a cycle occurs). When we found a cycle, we check if it is the current smallest one, and then iterate over the remaining permutations. The results of our cycle search algorithm are given in Table 5.8, where the expected cycle length should be $N! \cdot N^6$.

Table 5.8.: Results on the Cycle Search Algorithm

Test	N	Permutations	Smallest Cycle
Smallest Cycle	6	6!	12
	8	8!	208
	10	10!	$> 2^{20}$

We searched for cycles for $N = 6 \ldots 10$, but we where computationally limited after $N > 10$.

Furthermore, we analyzed the cyclic behavior of Spritz if we reduce the frequency of swaps during UPDATE. Therefore, we reduced the frequency

Algorithm 5.4 Multithreaded Cycle Search Algorithm

```
function SearchForCycles()
    for i = 0 to N! do
        P ← GeneratePemutations(N)
    end for
    for t = 0 to #ofThreads do
        thread ← P[t]
        thread → findCycles()
    end for
end function
function FindCycles()
    for p = 0 to P_thread do
        InitializeState()
        S ← P_thread[p]
        while currentState() ≠ startingState do
            Drip()
        end while
        if CheckIfSmallestCycle() then
            return "Smallest Cycle Found!"
        end if
    end for
end function
```

from swapping every time Update gets called down to zero swaps. We observed that if we do not swap in Update and no input is absorbed, Spritz runs in a cycle of length $6N$ for every N. Table 5.9 illustrates an cycle of $6N$ for $N = 16$.

5.5.2. State Recovery

In a state recovery attack an attacker searches (parts of) the internal state of the attacked cipher. If the next-state function (Figure 5.1) of the attacked cipher is easily invertible (as it is the case in Spritz), an attacker can then recover the initial state of the cipher after the secret key was absorbed. Moreover, an attacker can produce further output words without knowing

Table 5.9.: Cycle for $N = 16$ in the output sequence with no swaps in UPDATE where after $6N$ the output runs in a cycle.

4	4	5	14	15	1	15	1	4	7	10	14	6	10	15	12
1	7	9	15	6	13	4	12	8	0	9	10	3	13	3	13
8	3	14	10	10	6	3	8	5	3	13	11	10	9	8	8
12	12	13	6	7	9	7	9	12	15	2	6	14	2	7	4
9	15	1	7	14	5	12	4	0	8	1	2	11	5	11	5
0	11	6	2	2	14	11	0	13	11	5	3	2	1	0	0
4	4	5	14	15	1	15	1	4	7	10	14	6	10	15	12
1	7	9	15	6	13	4	12	8	0	9	10	3	13	3	13
8	3	14	10	10	6	3	8	5	3	13	11	10	9	8	8
12	12	13	6	7	9	7	9	12	15	2	6	14	2	7	4
9	15	1	7	14	5	12	4	0	8	1	2	11	5	11	5
0	11	6	2	2	14	11	0	13	11	5	3	2	1	0	0

the secret key, if the attacker can recover an internal state. To increase the security against state recovery attacks many ciphers use a large internal state, so that exhaustive search is infeasible. The number of possible states in Spritz is $N! \cdot N^6$ which leads to 2^{1730} possible states for $N = 256$, which is computationally infeasible.

Figure 5.1.: Next-state function of Spritz

$$i_t = i_{t-1} + w \tag{5.1}$$

$$j_t = k_{t-1} + S_{t-1}[j_{t-1} + S_{t-1}[i_t]] \tag{5.2}$$

$$k_t = i_t + k_{t-1} + S_{t-1}[j_t] \tag{5.3}$$

$$S_t[i_t] = S_{t-1}[j_t], S_t[j_t] = S_{t-1}[i_t] \tag{5.4}$$

$$z_t = S_t[j_t + S_t[i_t + S_t[z_{t-1} + k_t]]] \tag{5.5}$$

In this section, we propose three different state recovery algorithms. Our best algorithm recovers the initial state with complexity of $\approx 2^{1400}$, which is faster than exhaustive search through all possible initial states. The first algorithm implements a recursive backtracking approach to recover the internal state. The second approach searches for a pattern in the keystream

that allows us to easily recover the values for the Spritz registers i, j and k for a given window length. In the third approach we implemented a probabilistic algorithm to recover the internal state.

State Recovery with Backtracking

In our recursive state recovery algorithm with the backtracking, we used a similar approach as Knudsen et. al. [KMP+98] in their analysis of RC4. The state recovery attack needs only N output keystream bytes to successfully recover the internal state. To recover the initial state from a given internal state we can easily invert the Spritz next-state function, UPDATE, and calculate backwards to the state until the SHUFFLE function was applied in DRIP/SQUEEZE. Note that CRUSH in SHUFFLE cannot be inverted.

The idea of our recursive backtracking algorithm can be described as follows. We simulate the UPDATE/OUTPUT function of Spritz as long as all values to proceed are known. If a value is unknown we simply guess it and proceed. In our state recovery attack we can start at any point, but we assume that the initial registers are correctly known (either to a previous 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 loose 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 the calculated output word $z'_t = z_t$ where z_t is the output word we observe from the known output stream at step t, and z'_t is the output word according to our simulation of the next-state function. In our tests we observed that the output word z'_t sometimes equals z_t even with wrong guessed entries in the partly recovered state. There are several restrictions, which we can use to cut off branches in our search tree that could not lead to a correct solution anymore.

1. S is a permutation table where every element can only occur 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 indexes 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 $t + 1$.

We implemented the state recovery algorithm in a recursive function RECOVERSTATE() (see Algorithm 5.5), where most branches end up by contradictions. If in one branch we achieved to fill up the internal state table (at maximum of N steps) we verified the correct internal state by calculating the next-state a few more times and comparing the output words. Afterwards, we calculated 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.

To determine the efficiency of our attack we have to determine the complexity of the attack. The complexity is measured as total number of operations that are necessary to perform 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 can calculate the complexity by splitting the algorithm in several cases $c_i(a)$ to which we assign probabilities according to the occurrence of each case. Afterwards, we can compute the complexity based on the number of known bytes a in the permutation S and the assigned probabilities.

Algorithm 5.5 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 o $\leq$ 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 o $\leq$ 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 o $\leq$ 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 o $\leq$ 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 o $\leq$ 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

The complexity of the state recovery attack on Spritz can be calculated by using equation (5.6) & (5.7):

$$\sum_{i=1}^{5} c_i(a) = \frac{a}{2^n} \cdot c_{i+1}(a) + (1 - \frac{a}{2^n}) \cdot (2^n - a) \cdot c_{i+1}(a+1) \tag{5.6}$$

$$c_6(a) = \frac{a}{2^n} \cdot ((1 - \frac{a}{2^n}) \cdot 1 + c_1(a)) + (1 - \frac{a}{2^n}) \cdot (\frac{a}{2^n} \cdot 1 + c_1(a+1)) \tag{5.7}$$

Another option to estimate the complexity is to experimentally observe it by counting how many assignments for all entries in the permutation S are made until the initial state can be reconstructed.

The results of our state recovery algorithm are shown in Table 5.10 where a gives the number of pre-assigned values in the state table. Furthermore, results for different N are given in Appendix B. The complexities are calculated using Equations (5.6) and (5.7). Additionally, we give the complexity that we observed during experimental tests of our algorithm. The complexities are slightly faster then exhaustive searching through all possible initial states. Our algorithm becomes infeasible at $N = 32$, and is, therefore no threat to Spritz with $N = 256$.

Table 5.10.: Approximated complexity for $N = 32 \ldots 256$

N	a	calc. complexity	N!
32	0	$2^{99.8}$	$2^{117.6}$
64	0	$2^{249.0}$	$2^{296.0}$
128	0	$2^{599.4}$	$2^{716.1}$
256	0	$\approx 2^{1400}$	$2^{1683.9}$

Pattern Search and State Recovery

Our pattern search approach was inspired by the state recovery algorithm proposed by Maximov and Khovratovich [MK08]. 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 us 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 5.5.2). Our state recovery algorithm can be described as shown in Algorithm 5.6.

A contradiction can occur in steps $1 \ldots 3$ 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 compute 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 let us determine the register values with a high probability.

Algorithm 5.6 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

Therefore, we need two definitions to describe these patterns in more detail (these definitions were defined by Maximov and Khovratovich [MK08] and are adjusted to Spritz):

Definition 1 (d-order pattern). A d-order pattern is a tuple

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

where I and V are two vectors. 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 indexes in vector I.

Definition 2 (w-generative pattern). A pattern P_w is called w-generative if for any internal state that is compliant to P_w 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 5.7), 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^{256}, where the search of such a pattern is a pre-computation stage of our attack.

Algorithm 5.7 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 ← V and I
                        P ← i, j, k
                    end while
                end if
            end if
        end for
        StorePattern(P)
    end for
end function
```

We implemented our 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, we too often reach to a contradiction, whereby we delete more information from our state recovery table, than we can fill it up with our state recovery attack. If we handle contradictions in a different way, we observed that either the complexity gets too high or we get to much wrong values in our state recovery table.

Probabilistic State Recovery

Knudsen et. al. [KMP+98] proposed an probabilistic state recovery approach on RC4, that has been further improved by Golic and Morgari [GM08]. 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 equalky probable, which means that 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 register 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} distribution. 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 in a $N \cdot N$ matrix (see Equation 5.8), which represents the conditional probabilities of a given register and the associated entries in the permutation where the registers maps (e.g. $S[i][S_{t-1}[j_t]] = \Pr(S_{t-1}[i_t] \mid \mathrm{j})$).

$$S = \begin{pmatrix} \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \\ \frac{1}{N} & \frac{1}{N} & \frac{1}{N} & \frac{1}{N} \end{pmatrix} \tag{5.8}$$

We implemented the state recovery algorithm (see Algorithm 5.8) 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 5.8 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 unfortunately our algorithm either did not converge or some of the entries in the table were incorrect.

5.6. Summary

Our contribution to the analysis of Spritz consists of a large amount of tests and attacks. First, we applied some generic attacks, such as the search for weak key classes and state rotations. Next, we applied some statistical attacks, where we found some biases. Unfortunately, the biases occur only if a large amount of plaintext is used, such that they can not be used in practical attacks. Nevertheless, we gained new insights in Spritz. Furthermore, we searched for collisions and pre-image attacks, when Spritz is used as a hash function. We also introduced three state recovery attacks, when Spritz is used as a stream cipher. Our best state recovery attack has a complexity of approximately 2^{1400}, which is faster than exhaustive search of all possible states, but still far from a practical attack.

6
Conclusions

In this thesis, we analyzed the security of symmetric key primitives. In more detail, we focused on the recently proposed stream cipher and hash function called Spritz. Spritz was presented at the rump session of CRYPTO 2014 by the designer of RC4, Ronald L. Rivest together with Jacob Schuldt. After years of analysis there are still no practical attacks on RC4. Nevertheless, RC4 is well analyzed and some weaknesses has been published. Spritz is a complete redesign of RC4 introducing new security features and new cryptographic functionalities due to its sponge-like design. Spritz was designed as a drop-in replacement of RC4.

However, a detailed analysis is needed before Spritz can be used in practical applications. In the proposal of Spritz the designers claimed the security bounds according to many statistical tests and extensively simulations. This thesis provides some of the first cryptanalytic results on Spritz. In our analysis, we examined Spritz according to various different cryptanalytic methods. These include generic attacks, weak keys, state rotations as well as statistical attacks with distribution tests and various correlations. Furthermore, we studied the security of Spritz, when used as a hash function. In

our analysis, we searched for collisions in the Spritz hash function but Spritz seems to be secure against collision attacks. Furthermore, we investigated pre-image attacks combined with collision attacks but unfortunately, we could not construct any hash values that can be used in a pre-image attack to find the corresponding message.

As Spritz should replace the stream cipher RC4, our main focus was considered on Spritz as a stream cipher. In the context of stream cipher attacks we analyzed the cyclic behavior of Spritz and proposed three different approaches for state recovery attacks on Spritz. In the context of Spritz as a stream cipher we observed that the output of Spritz results in a cycle of $6N$ if no input is absorbed and no SWAP is done in the next-state function, UPDATE. Moreover, we propose three different state recovery algorithms where our best algorithm can recover the initial state with complexity of $\approx 2^{1400}$, which is faster then exhaustive searching through all possible states.

As attacks on cryptographic algorithms never get weaker, we will see more advanced attacks on Spritz in the near future. There are many points that can be improved in the analysis of Spritz. In the search for biases and distinguishers an attacker with a huge computer cluster can do more advanced statistical tests with much more inputs. For collision and pre-image attacks other analysis may show different aspects and other potentials for attacks. Furthermore, the complexity of the state recovery attacks may be reduced by optimizing the state recovery attacks or through combination of the state recovery attacks with some heuristics that reduce the number of branches in the search trees. Another open point is to determine the complexity of inverting the state randomization function SHUFFLE related to CRUSH.

It is important to continue the analysis of Spritz to get a good view on the security margins of Spritz. Even though this paper provides the first external analysis, it is only a first step towards increasing the confidentiality in the security of Spritz.

Bibliography

[a5187] A5/1 gsm standard, 1987.

[a5289] A5/2 gsm standard, 1989.

[ABP+13] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of rc4 in tls and wpa. `http://www.isg.rhul.ac.uk/tls/RC4biases.pdf`, July 2013.

[AFP13] N.J. Al Fardan and K.G. Paterson. Lucky thirteen: Breaking the tls and dtls record protocols. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 526–540, May 2013.

[Ano94] Anonymous. Rc4 source code. `http://cypherpunks.venona.com/date/1994/09/msg00304.html`, September 1994.

[Bar04] Zoltak Bartosz. Vmpc one-way function and stream cipher. In *Lecture Notes in Computer Science*. Springer, 2004.

[Bar14] Zoltak Bartosz. Statistical weakness in spritz against vmpc-r: in search for the rc4 replacement. Cryptology ePrint Archive, Report 2014/985 `http://eprint.iacr.org/`, 2014.

[BBC+08] Côme Berbain, Olivier Billet, Anne Canteaut, Nicolas Courtois, Henri Gilbert, Louis Goubin, Aline Gouget, Louis Granboulan, Cédric Lauradoux, Marine Minier, Thomas Pornin, and Hervé Sibert. Sosemanuk, a fast software-oriented stream cipher. In *New Stream Cipher Designs*, Lecture Notes in Computer Science, 2008.

[BC08] Eli Biham and Yaniv Carmeli. Efficient reconstruction of rc4 keys from internal states. In *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[BD08] Steve Babbage and Matthew Dodd. The mickey stream ciphers. In *New Stream Cipher Designs*, Lecture Notes in Computer Science, 2008.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: single-pass authenticated encryption and other applications. Cryptology ePrint Archive, Report 2011/499 `http://eprint.iacr.org/`, 2011.

[BDPVA14] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic sponge functions. `http://sponge.noekeon.org/`, January 2014.

[Ber08] Daniel J. Bernstein. The salsa20 family of stream ciphers. In *New Stream Cipher Designs*, Lecture Notes in Computer Science, 2008.

[Bih93] Eli Biham. New types of cryptoanalytic attacks using related keys (extended abstract). In *Advances in Cryptology - EUROCRYPT '93*, volume 765 of *Lecture Notes in Computer Science*, pages 398–409. Springer, 1993.

[BVP+03] Martin Boesgaard, Mette Vesterager, Thomas Pedersen, Jesper Christiansen, and Ove Scavenius. Rabbit: A new high-performance stream cipher. In *Fast Software Encryption*, Lecture Notes in Computer Science, 2003.

[CM11] Jiageng Chen and Atsuko Miyaji. How to find short rc4 colliding key pairs. In *Information Security*, volume 7001 of *Lecture Notes in Computer Science*, pages 32–46. Springer, 2011.

[DCP08] Christophe De Cannière and Bart Preneel. Trivium. In *New Stream Cipher Designs*, Lecture Notes in Computer Science, 2008.

[DR11] Thai Duong and Juliano Rizzo. Here come the $\oplus$ ninjas. BEAST attack, September 2011.

[e001] Bluetooth^{TM}. Bluetooth Specifications, Version 1.1, 2001.

[Fin94] Hal Finnley. An rc4 cycle that can't happen. `https://groups.google.com/d/msg/sci.crypt/JsO3xEATGFA/nelEG6RoUxcJ`, September 1994.

[FMS01] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In *Selected Areas in Cryptography*, Lecture Notes in Computer Science, pages 1–24, 2001.

[GM08] Jovan Golic and Guglielmo Morgari. Iterative probabilistic reconstruction of rc4 internal states. Cryptology ePrint Archive, Report 2008/348, `http://eprint.iacr.org/2008/348`, 2008.

[Haw98] Philip Hawkes. Differential-linear weak key classes of idea. In *Advances in Cryptology — EUROCRYPT'98*, volume 1403 of *Lecture Notes in Computer Science*, pages 112–126. Springer, 1998.

[HJM07] Martin Hell, Thomas Johansson, and Willi Meier. Grain, a stream cipher for constrained environments. *International Journal on Mobile Communication and Computing*, 2007.

[IP08] Sebastian Indesteege and Bart Preneel. Collisions for rc4-hash. In *Information Security*, Lecture Notes in Computer Science, pages 355–366. Springer, 2008.

[Kle08] Andreas Klein. Attacks on the rc4 stream cipher. *Des. Codes Cryptography*, 48(3):269–286, 2008.

[KM08] Shahram Khazaei and Willi Meier. On reconstruction of rc4 keys from internal states. In *Mathematical Methods in Computer Science*, Lecture Notes in Computer Science, pages 179–189. Springer, 2008.

[KMP+98] Lars R. Knudsen, Willi Meier, Bart Preneel, Vincent Rijmen, and Sven Verdoolaege. Analysis methods for (alleged) rc4. In *Advances in Cryptology - ASIACRYPT '98*, volume 1514 of *Lecture Notes in Computer Science*, pages 327–341. Springer, 1998.

[Knu92] Lars R. Knudsen. Cryptanalysis of loki91. In *Advances in Cryptology - AUSCRYPT '92*, volume 718, pages 196–208, 1992.

[Lan14] Adam Langley. The poodle bites again. `https://www.imperialviolet.org/2014/12/08/poodleagain.html`, December 2014.

[Mat09] Mitsuru Matsui. Key collisions of the rc4 stream cipher. In *Fast Software Encryption*, volume 5665 of *Lecture Notes in Computer Science*, pages 38–50. Springer, 2009.

[MDK14] Bodo Moeller, Thai Duong, and Krzysztof Kotowicz. This poodle bites: Exploiting the ssl 3.0 fallback. `https://www.openssl.org/~bodo/ssl-poodle.pdf`, September 2014.

[Mir02] Ilya Mironov. (not so) random shuffles of rc4. In *Advances in Cryptology — CRYPTO 2002*, Lecture Notes in Computer Science, pages 304–319. Springer, 2002.

[MK08] Alexander Maximov and Dmitry Khovratovich. New state recovery attack on rc4. Cryptology ePrint Archive, Report 2008/017, `http://eprint.iacr.org/2008/017`, 2008.

[MS87] Judy H. Moore and Gustavus J. Simmons. Cycle structure of the des with weak and semi-weak keys. In *Advances in Cryptology — CRYPTO' 86*, Lecture Notes in Computer Science, pages 9–32. Springer, 1987.

[MS01] Itsik Mantin and Adi Shamir. A practical attack on broadcast rc4. In *Fast Software Encryption*, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2001.

[PM07] Goutam Paul and Subhamoy Maitra. Permutation after rc4 key scheduling reveals the secret key. In *Selected Areas in Cryptography*, volume 4876 of *Lecture Notes in Computer Science*, pages 360–377, 2007.

[RB08] Matthew J. B. Robshaw and Olivier Billet, editors. *New Stream Cipher Designs - The eSTREAM Finalists*. Lecture Notes in Computer Science. Springer, 2008.

[RC94] Phillip Rogaway and Don Coppersmith. A software-optimized encryption algorithm. In Springer, editor, *Fast Software Encryption*, volume 809 of *Lecture Notes in Computer Science*, 1994.

[Rog02] Phillip Rogaway. Security of cbc ciphersuites in ssl/tls: Problems and countermeasures. `https://www.openssl.org/~bodo/tls-cbc.txt`, 2002.

[Roo95] Andrew Roos. A class of weak keys in the rc4 stream cipher. `https://groups.google.com/forum/#!topic/sci.crypt/FeVX7yiztXU`, 1995.

[RS14] Ronald L. Rivest and Jacob C. N. Schuldt. Spritz—a spongy rc4-like stream cipher and hash function. `http://people.csail.mit.edu/rivest/pubs/RS14.pdf`, August 2014.

[SG08] Maitra Subhamoy and Paul Goutam. Analysis of rc4 and proposal of additional layers for better security margin. In *Lecture Notes in Computer Science*. Springer, 2008.

[SP04] Paul Souradyuti and Bart Preneel. A new weakness in the rc4 keystream generator and an approach to improve the security of the cipher. In *Lecture Notes in Computer Science*. Springer, 2004.

[spr14] Spritz source code. `https://www.npmjs.com/package/spritzjs`, 2014.

[SVV11] Pouyan Sepehrdad, Serge Vaudenay, and Martin Vuagnoux. Statistical attack on rc4 distinguishing wpa. In *Advances in Cryptology*, EUROCRYPT'11, pages 343–363. Springer, 2011.

[Wu08] Hongjun Wu. The stream cipher hc-128. In *New Stream Cipher Designs*. Springer, 2008.

A

Results of Statistic Tests

In this Chapter the results of our statistical tests are summarized. In our analysis we performed various different statistical tests. These include distribution tests, where we observed if the registers, the permutation or the output values follow a uniform distribution. Additionally, we applied some correlation tests between internal states and the initial state or between random input and the output of Spritz.

The results are given in Table A.1 to Table A.4.

A.1. Results of register distribution tests

Table A.1 shows the results of our tests of the different registers in Spritz. Therefore we tested for various sizes of $N = 8 \dots 256$ with about 2^{20} iterations. In our tests we revealed no significant bias that is larger than four times the standard deviation of its expected value. All registers therefore provide a uniform distribution of their values.

Table A.1.: Results of register distribution tests

Test	N	Iterations	Results
register i	8 ... 256	2^{20}	no bias
register j	8 ... 256	2^{20}	no bias
register k	8 ... 256	2^{20}	no bias
register z	8 ... 256	2^{20}	no bias

A.2. Results of output distribution tests

Table A.2 shows the results of our tests of the output values in Spritz. In our analysis we tested Spritz for different $N = 8 \ldots 256$ and up to 2^{20} iterations to find biases in the output key stream. In our main test we squeezed 2^{20} output values and then observed if there are some anomalies in the key stream. We found a bias for $N = 8$ when we absorb so much input that Shuffle gets called during Absorb. In our second test (with the separate distribution) we Drip for 2^{20} times and directly observe the first output byte. Therefore we again found a bias for $N = 8$ which is about 12 times the standard deviation.

Table A.2.: Results of output distribution tests

Test	N	Iterations	Results
output values (no input)	8 ... 256	2^{20}	no bias
output values ($\frac{N}{4}$ input)	8 ... 256	2^{20}	no bias
output values ($\frac{N}{4}+1$ input, Shuffle called)	8	2^{12}	bias ($\approx$ 14 standard deviation)
	16 ... 256	2^{20}	no bias
successive output values	8 ... 256	2^{20}	no bias
output values (separate distribution)	8	2^{13}	bias ($\approx$ 12 standard deviation)
	16 ... 256	2^{20}	no bias

A.3. Results of combined registers distribution tests

Table A.3 shows the results of our tests for combined register distributions. In these tests we observed the deviation from a uniform distribution if we combine several registers. We found biases for distributions *ijk*, *iz*2*z* and *iz*3*z* for all $N = 8 \ldots 256$.

Table A.3.: Results of combined registers distribution tests

Test	N	Inputs	Results
combined distribution of registers *ijk*	8	2^9	bias ($\approx \frac{1}{2}\cdot$ standard deviation)
	16	2^{12}	bias ($\approx 7\cdot$ standard deviation)
	32	2^{15}	bias ($\approx 19\cdot$ standard deviation)
	64	2^{18}	bias ($\approx 24\cdot$ standard deviation)
	128	2^{22}	bias ($\approx 49\cdot$ standard deviation)
	256	2^{25}	bias ($\approx 65\cdot$ standard deviation)
combined distribution of registers *iz*2*z*	8	2^9	bias ($\approx 4{,}7 \cdot 10^6\cdot$ standard deviation)
	16	2^{12}	bias ($\approx 805 \cdot 10^6\cdot$ standard deviation)
	32	2^{16}	bias ($\approx 113 \cdot 10^{11}\cdot$ standard deviation)
combined distribution of registers *iz*3*z*	8	2^9	bias ($\approx 4{,}6 \cdot 10^6\cdot$ standard deviation)
	16	2^{12}	bias ($\approx 798 \cdot 10^6\cdot$ standard deviation)
	32	2^{16}	bias ($\approx 113 \cdot 10^{11}\cdot$ standard deviation)

A.4. Results of correlation tests

Table A.4 shows the results of our correlation tests. We could observe that for large $N = 128, 256$ after Absorb the correlation between the internal state and the initial state is high. After an application of Shuffle, the correlation is nearly zero. We also absorbed the correlation between random input and Spritz output. For 2^{20} random inputs we found about 20 correlating outputs, which can be seen as normal according to the high number of inputs.

Table A.4.: Results of correlation tests

Test	N	Inputs	Results
correlation after Absorb	8 ... 64	$\frac{N}{4}$ random input	no correlation
	128 ... 256		high correlation ($\approx 0,95$)
correlation after Whip	8 ... 256	no input	no correlation
correlation after Whip, Crush	8 ... 256	no input	correlation ($\approx 0,5$)
correlation after Shuffle	8 ... 256	no input	no correlation
correlation random input / output	8	2^{20} 10 byte random inputs	23 correlated pairs
	16		17 correlated pairs
	32		25 correlated pairs
	64		26 correlated pairs
	128		17 correlated pairs
	256		21 correlated pairs

B

Results of State Recovery Attacks

In this Chapter 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 experimental which is given in the next tables as *exp.complexity*. Additionally, we calculated the complexity which is given as *calc.complexity*. The parameter k means 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 $N!$.

In Table B.1 and Table B.2 we applied our state recovery attack with random input, which leads to random initial states that we have to recover. 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 B.3 to Table B.5.

B.1. Results of the state recovery attack with backtracking

Table B.1.: Results for state recovery attack with backtracking for N = 8

k	exp. complexity	calc. complexity	(N-k)!
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 B.2.: Results for state recovery attack with backtracking for N = 16

k	exp. complexity	calc. complexity	(N-k)!
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	$2^{7,5}$	$2^{13,7}$	$2^{15,3}$
07	$2^{21,0}$	$2^{16,5}$	$2^{18,5}$
06	-	$2^{19,5}$	$2^{21,8}$
05	-	$2^{22,7}$	$2^{25,3}$
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	-	$2^{38,4}$	$2^{44,3}$

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

Table B.3.: Results for state recovery attack with backtracking for N = 8 and no input (i.e. identity permutation as initial state)

k	exp. complexity	calc. complexity	(N-k)!
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 B.4.: Results for state recovery attack with backtracking for N = 16 and no input (i.e. identity permutation as initial state)

k	exp. complexity	calc. complexity	(N-k)!
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	-	$2^{38,4}$	$2^{44,3}$

Table B.5.: Results for state recovery attack with backtracking for N = 32 and no input (i.e. identity permutation as initial state)

k	exp. complexity	calc. complexity	(N-k)!
31	2^{0}	2^{0}	2^{0}
30	$2^{1,6}$	2^{1}	2^{1}
29	$2^{2,6}$	$2^{2,6}$	$2^{2,6}$
28	$2^{3,0}$	$2^{4,6}$	$2^{4,6}$
27	$2^{3,6}$	$2^{6,9}$	$2^{6,9}$
26	$2^{3,8}$	$2^{9,5}$	$2^{9,5}$
25	$2^{4,5}$	$2^{11,3}$	$2^{12,3}$
24	$2^{5,3}$	$2^{13,7}$	$2^{15,3}$
23	$2^{5,3}$	$2^{16,5}$	$2^{18,5}$
22	$2^{5,7}$	$2^{19,5}$	$2^{21,8}$
21	$2^{6,5}$	$2^{22,7}$	$2^{25,3}$
20	$2^{6,5}$	$2^{26,0}$	$2^{28,9}$
19	$2^{7,1}$	$2^{28,5}$	$2^{32,6}$
18	$2^{7,1}$	$2^{31,5}$	$2^{36,4}$
17	$2^{7,1}$	$2^{34,9}$	$2^{40,3}$
16	$2^{18,7}$	$2^{38,4}$	$2^{44,3}$
15	$2^{18,7}$	$2^{42,1}$	$2^{48,3}$
14	-	$2^{46,0}$	$2^{52,5}$
13	-	$2^{48,9}$	$2^{56,7}$
12	-	$2^{52,4}$	$2^{61,0}$
11	-	$2^{56,1}$	$2^{65,4}$
10	-	$2^{60,0}$	$2^{69,9}$
09	-	$2^{64,1}$	$2^{74,4}$
08	-	$2^{68,2}$	$2^{79,0}$
07	-	$2^{71,5}$	$2^{83,6}$
06	-	$2^{75,3}$	$2^{88,3}$
05	-	$2^{79,3}$	$2^{93,1}$
04	-	$2^{83,5}$	$2^{97,9}$
03	-	$2^{87,8}$	$2^{102,8}$
02	-	$2^{92,3}$	$2^{107,7}$
01	-	$2^{95,8}$	$2^{112,6}$
00	-	$2^{99,8}$	$2^{117,6}$