Spritz—a spongy RC4-like stream cipher and hash function

Ronald L. Rivest
MIT CSAIL
Cambridge, MA 02139
rivest@mit.edu

Jacob C. N. Schuldt
Research Institute for Secure Systems
AIST, Japan
jacob.schuldt@aist.go.jp

October 27, 2014

Abstract

This note reconsiders the design of the stream cipher RC4, and proposes an improved variant, which we call "Spritz" (since the output comes in fine drops rather than big blocks.)

Our work leverages the considerable cryptanalytic work done on the original RC4 and its proposed variants. It also uses simulations extensively to search for biases and to guide the selection of intermediate expressions.

We estimate that Spritz can produce output with about 24 cycles/byte of computation. Furthermore, our statistical tests suggest that about 2^{81} bytes of output are needed before one can reasonably distinguish Spritz output from random output; this is a marked improvement over RC4.

In addition, we formulate Spritz as a "sponge (or sponge-like) function,"[5], which can ABSORB new data at any time, and from which one can SQUEEZE pseudorandom output sequences of arbitrary length. Spritz can thus be easily adapted for use as a cryptographic hash function, an encryption algorithm, or a message-authentication code generator. (However, in hash-function mode, Spritz is rather slow.)

Keywords: RC4, Spritz, stream cipher, sponge function, Absorb, Squeeze, encryption, message authentication code, cryptographic hash function.

1 Introduction

The cryptographic stream cipher RC4 was designed by Ronald L. Rivest in 1987 for RSA Data Security; it then appeared in RSA's cryptographic library and was used first on a commercial basis in Lotus Notes.

RC4 was at first a trade secret, but the algorithm was reverse-engineered and published in 1994. Pseudocode for RC4 is given in Figure 1.

Since then, RC4 has been widely studied and has been adopted for use in TLS and other standards. It has been said that RC4 is (or was) the most widely used stream cipher in the world.

It is a maxim that cryptographic algorithms never get stronger over time, only weaker. RC4 is no exception. Studies of the algorithm have revealed that the key scheduling algorithm, the state update algorithm, and the output function all have statistical or other weaknesses. While RC4 is still usable (with care), the algorithm is now over 25 years old and deserves retirement and replacement by newer, stronger, alternatives.

This paper considers the space of "RC4-like" stream ciphers, and proposes one, which we call Spritz, that might make a suitable replacement. Spritz attempts to repair weak design decisions in RC4, while remaining true to its general design principles. We do not consider other stream-cipher proposals here, and expect that for many applications other word-oriented architectures may be a better choice than the byte-oriented RC4/Spritz style.

We thus consider the question, "What should the

```
KSA(K)
1  for i = 0 to N − 1
2        S[i] = i
3  j = 0
4  for i = 0 to N − 1
5        j = j + S[i] + K[i mod K.length]
6        Swap(S[i], S[j])
7  i = j = 0
```

```
PRG()
1  i = i + 1
2  j = j + S[i]
3  Swap(S[i], S[j])
4  z = S[S[i] + S[j]]
5  return z
```

Figure 1: Pseudocode for the original RC4. KSA is the RC4 key setup algorithm, which initializes i, j, and the permutation S based on a supplied key K. PRG is the pseudorandom generator—each call to PRG updates the state and produces one byte (modulo N) of output. All additions are modulo N.

original RC4 have looked like?"

Our proposal Spritz not only provides a "drop-in replacement" for RC4, with much improved security properties, but also provides a suite of new cryptographic functionalities based on its new "sponge-like" construction and interface.

(We reserve "RC4" to refer to the original RC4 as described in Figure 1. Our new proposal will be referred to as "Spritz.")

Today's crypto-designer has the advantage of faster processors: simulations and statistical analyses are much easier to perform. Intel's 386 (introduced in 1985) had a clock rate of 16MHz; today's processors are multi-core and run at 2GHz. We rely on extensive simulations of alternative designs to inform our design choices. (Indeed, our computations required about five "core-months" on modern CPU's!)

RC4 embodies the following architectural choices:

Large state space: The RC4 state is very large, compared to most previous stream cipher proposals.

Permutation of Z_N: The major portion of the state is a 256-byte array S that represents a permutation of $Z_N = \{0, 1, \ldots, N-1\}$; in the original RC4 we have $N = 256$, so RC4 is byte-oriented.

Fast update and output: The next-state function changes (swaps) only two bytes of the array S; the output function depends only on two registers i, j, and a few bytes of the array S.

Scalable (variable N): The RC4 cipher is well-defined for any value of $N > 2$, not just $N = 256$. This is helpful when analyzing RC4 variants. We call a value in $\{0, 1, \ldots, N-1\}$ a "byte" (or N-value) throughout.

Arbitrary key-length: The length L of the supplied key is arbitrary (up to 256 bytes in RC4). Each element of the key is a byte.

Non-invertible key setup: The key-setup algorithm is designed to be difficult to invert: it should be hard to derive the key, given the initial RC4 state.

Invertible update function: The next-state function is invertible. This provides some assurance that the period of the next-state function will be large.

Complex output function: The output function is a complex (yet simple to evaluate) function of the current state.

Given the cryptanalysis that has been performed on RC4 to date, the above design principles seem sound. Yet the particular design choices made for RC4 (such as the choice of key setup algorithm) seem to have been not as strong as one might like. In this paper, we revisit these particular design decisions,

```
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 2: Pseudocode for Spritz. The main interface routines are INITIALIZESTATE, ABSORB (and ABSORBSTOP), and SQUEEZE. The state consists of byte registers i, j, k, z, w, and a, and an array S containing a permutation of $\{0, 1, \ldots, N-1\}$. Section 2 illustrates the use of Spritz for encryption, decryption, hashing, MAC's, and authenticated encryption. Section 3 describes the above procedures in detail. These procedures have as an implicit argument the current state Q; components of Q (such as the permutation $Q.S$) are referred to by component name (e.g. S) rather than fully qualified names (as in $Q.S$) for brevity. This is the entire code, other than the definitions of LOW and HIGH in equations (1)–(2). When N is a power of 2, the last two lines of WHIP are equivalent to $w = w + 2$.

and suggest improved alternatives. Our Spritz proposal may serve as a "drop-in replacement" for RC4 with improved security; it also provides additional cryptographic capabilities, such as hashing, derived from its reformulation with a sponge-like construction.

2 Sponge functions

One objective of our RC4 redesign is to reformulate RC4 as a "sponge function." RC4 is a natural fit for adaptation to the sponge function paradigm, as it already has a large state space.

As proposed by Bertoni et al. in their seminal paper [5], a sponge function has a function ABSORB that absorbs variable-length input into the state, and a function SQUEEZE that produces variable-length output from the state. Both functions may change the state, using a single state-space permutation function f.

Many properties of the sponge function method are derived assuming that f is an arbitrary such permutation—that there are no properties of f that may be useful to an adversary, and that the adversary's best attacks are "generic" (they would work for any f).

In Spritz, the SHUFFLE procedure corresponds to the state-space permutation f of the sponge function. However, SHUFFLE is not a state-space permutation, but a many-to-one map (due to its invocations of CRUSH), so the correspondence is not precise. SHUFFLE does invoke procedure WHIP, however, which *is* such a state-space permutation. The sponge-like character of Spritz, based on SHUFFLE, WHIP, and CRUSH, is discussed further in Section 7.1.

Spritz does not fit the sponge function model exactly, for several reasons:

- the main component of the state is not a bit string, but rather a permutation S of $\{0, 1, \ldots, N-1\}$,
- the state changes involve swapping bytes of the permutation S, not exclusive-oring new input into the state, and
- the SHUFFLE primitive is not a permutation of the state space, but a many-to-one mapping,
- the production of output using SQUEEZE does not involve f (that is, SHUFFLE), but is based rather on a separate state-change operation (DRIP) that makes only local state changes, and
- Spritz includes a novel additional primitive, ABSORBSTOP, that has the effect of absorbing a special "stop-symbol" that is not part of the ordinary input alphabet.

Nonetheless, our work is inspired by that of Bertoni et al., and we consider the design of Spritz to be at least "sponge-like" or "spongy," if not strictly speaking a sponge function by their definition.

Bertoni et al. define the "capacity" c of a sponge function to be the amount of information in the state that is protected from changes when new input is absorbed, and that is only changed when f is applied.

The strength of a cryptographic function based on a sponge construction depends upon its capacity—for many security properties an adversary who wishes to break the property must find a collision within the state space, which requires time $O(2^{c/2})$ for a generic sponge construction [5], when the capacity c is measured in bits.

The capacity of Spritz (with $N = 256$) is at least 896 bits (112 bytes), since the last 112 bytes of S are untouched by ABSORB except through calls to SHUFFLE. The Spritz capacity is actually a bit larger, since we should include registers i, j, k, w, and z in our count; doing so yields a total Spritz capacity of $c = 936$ bits.

The capacity of Spritz exceeds that of the new SHA-3 standard hash function as adopted (Keccak) in all SHA-3 configurations except SHA-3 with 512-bit outputs (when the Keccak capacity is 1024 bits).

The following subsections illustrate the versatility of sponge functions, following Bertoni et al. [5]. The reader may also wish to consult the Spritz pseudocode in Figure 2 while reading these subsections, as our presentations have some aspects adapted for use with Spritz.

Our focus in this paper is on Spritz as a stream cipher. Although we formulate our proposal for Spritz

in the sponge function framework, additional evaluation is definitely needed regarding the use of Spritz in these additional sponge function modes.

2.1 AbsorbStop function

We extend the sponge function interface by introducing the function AbsorbStop(); calling AbsorbStop() is equivalent to absorbing a special "stop" symbol ("▌") that is outside the ordinary input alphabet. The intent is to provide a simple clean way to separate different inputs being absorbed. The name "stop" was chosen as it is reminiscent of the use of the word "STOP" for the full stop character in telegrams.

2.2 Encryption

Here is pseudocode illustrating the use of a sponge function for encryption and decryption.

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

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

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

Encrypt uses key-setup algorithm KeySetup to initialize the state and absorb the key K, then adds modulo N each byte of the message M with the corresponding byte of the output of Squeeze, yielding ciphertext C. Procedure Decrypt is identical, except for switching M and C, and replacing addition modulo N with subtraction modulo N. Note that the key, message, and ciphertext are all byte sequences (sequences of values modulo N).

Addition/subtraction modulo N are used instead of the more traditional exclusive-or, for consistency with the design goal that Spritz should work for all N, not just N that are powers of 2. Of course, when N *is* a power of 2, one could use exclusive-or rather than addition/subtraction. We call this variant Spritz-xor, but do not further discuss this variant in this paper.

To encrypt with an IV (initialization vector or nonce) IV, one should, after the key is input, call AbsorbStop procedure to separate the two fields, and then input the IV, as follows.

```
EncryptWithIV(K, IV, M)
1  KeySetup(K); AbsorbStop()
2  Absorb(IV)
3  C = M + Squeeze(M.length)
4  return C
```

2.3 Hash function

The following procedure Hash produces an r-byte hash of the input message (byte sequence) M.

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

Hash first absorbs the input message M (a byte sequence). Next, it call AbsorbStop to signal the end of the message M, and the beginning of the next input, which is the desired hash length, r.

For definiteness, assume that r is represented as a base-N integer, high-order byte first, with no leading zeros.

We achieve functional separation by including the value r as part of the input; we do not want a hash function producing 16-byte outputs to behave anything like its cousin that produces 32-byte outputs. Without including r as input, the 16-byte output would merely be a prefix of the 32-byte output.

Domain separation We next illustrate using a sponge function to provide "hashing with domain separation"—different functional behavior for different domains or applications.

Let J denote the domain or application name.

The DomHash procedure uses the new AbsorbStop procedure to cleanly separate the input of the domain name J from the input of the message M to be hashed; the hash length r is also provided as input, as in Hash, for providing functional separation based on output length.

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

2.4 Message Authentication Code (MAC)

The following pseudocode shows how one may implement a MAC (Message Authentication Code) easily using a sponge function.

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

2.5 Authenticated Encryption with Associated Data

The procedure AEAD(K, Z, H, M, r) takes as input a key K, a nonce Z (a value that will never be used again), a "header" H (this is the "associated data" that needs to be authenticated but not encrypted), and a message M (to be both encrypted and authenticated), and returns a two-part result consisting of the encryption of message M followed by an r-byte authentication tag (computed over K, Z, H, and M). The receipient may need to be sent Z and H if she doesn't otherwise know these values; we assume she knows K and r. See Bellare et al. [4] for more discussion of AEAD mode.

```
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_t,
   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 Squeeze(r)
```

The $N/4$-byte block size is chosen to equal the Spritz input block size for Absorb, for efficiency.

2.6 Deterministic Random Bit Generator (DRBG)

A sponge function is naturally a deterministic random-bit generator (DRBG), see NIST [20], or pseudo-random number generator (PRNG), as might be used for example in `\dev\random`. The sponge state is the "entropy pool." New random input can be included at any time using Absorb, and output may be extracted at any time using Squeeze.

3 Spritz specification

This section gives a precise specification of Spritz. See Figure 2 for Spritz pseudocode.

3.1 Notation and terminology

N: All values in Spritz are modulo-N (in $Z_N = \{0, 1, \ldots, N-1\}$). The default value of N is 256, so Spritz is byte-oriented. For convenience, we consistently refer to a value in Z_N as a "byte", or an "N-value."

Addition, etc.: The symbol "+" always means addition modulo N, and "−" always means subtraction modulo N.

Registers: Spritz uses a small number of registers, each holding one byte (N-value). RC4 used two registers: i and j. For Spritz we consider designs with six such registers: i, j, k, w, z, and a.

To the original registers i and j we add a new register k, with similar function.

Registers w, z, and a have special functions.

Register w is modified every time that Whip is called, but is always relatively prime to N; register i is incremented by w whenever it is changed.

Register z holds the last value produced by the output function.

Finally, we let a denote the number of nibbles (half-bytes) that have been absorbed since the start or the last Shuffle of the permutation S. Spritz is in "absorbing mode" if $a > 0$, and is in "squeezing mode" (that is, ready to squeeze output) if $a = 0$.

Permutation S: As with $RC4$, the array S of length N holds a permutation of $Z_N = \{0, 1, \ldots, N-1\}$. Subscripts for S are interpreted modulo N.

State: The state Q_t of Spritz at time t consists of the register values and S. We denote components of state Q as i, j, k, w, z, and a for the registers, and S for the S array; an element of array S is denoted with a subscript (e.g. $S[i]$).

Spritz thus has at most

$$\#(N) = N^6 N!$$

states.

In the pseudocode of Figure 2, the state Q is an implicit argument to all functions, and components of the state are for brevity written as i, j, $S[i]$ and so on rather than $Q.i$, $Q.j$, $Q.S[Q.i]\ldots$. Later, in Section 5, we may make the state explicit sometimes as a first argument, writing Absorb(Q, I) instead of Absorb(I), etc.

Key: The cryptographic key K is a length L array $K[0 \mathinner{..} L-1]$ where each element of the key is a byte (N-value). The length L of K may be any nonnegative integer (including zero).

We let $\mathcal{K}_L$ denote the set of all possible L-byte keys; the size of $\mathcal{K}_L$ is N^L. Abusing notation, we let $\mathcal{K}_L()$ denote a procedure that returns a randomly-chosen key from $\mathcal{K}_L$.

Nibbles: It is convenient to consider a byte as consisting of two half-byte "nibbles." Let $D = \lceil\sqrt{N}\rceil$, and let "nibble" mean a D-value (that is, a value in $\{0, 1, \ldots, D-1\}$).

A byte b can be represented as a pair (x, y) of nibbles, where $b = D \cdot x + y$. Here x is the high-order nibble and y is the low-order nibble. In radix-D notation, $b = xy$.

Let $\textsc{low}(b)$ and $\textsc{high}(b)$ denote the low and high order nibbles x and y, respectively, of a byte (N-value) b:

$$\textsc{low}(b) = b \bmod D \quad (1)$$

$$\textsc{high}(b) = \lfloor b/D \rfloor \quad (2)$$

For the default $N = 256$ we have $D = 16$; each eight-bit byte contains two four-bit (hexadecimal) nibbles, which can be easily computed using shifting and masking.

Other notation: Let $\lceil x \rceil$ and $\lfloor x \rfloor$ denote the ceiling and floor functions (the least integer not less than x, and the greatest integer not more than x, respectively).

If x and y are bytes (N-values), $x + y$ denotes their sum modulo N. If x and y are byte sequences of the same length, $x + y$ denotes the element-by-element sum modulo N. We similarly extend the notation $x - y$ (subtraction modulo N).

We let $\lg(x)$ denote the base-2 logarithm of x.

3.2 Spritz procedures

This section gives a more detailed description of the Spritz procedures given in Figure 2.

InitializeState: InitializeState sets Spritz to its standard initial state.

It sets the five registers i, j, k, z, and a to zero, and sets register w to 1. It also sets $S[0 \mathinner{..} N-1]$ to the identity permutation on $\{0, 1, \mathinner{..}, N-1\}$.

The registers have the following functions. Each time the function Drip is called, register i increases

by w modulo N, while registers j and k change pseudorandomly. Register a counts how many key nibbles have been absorbed in the current block (since initialization or the last call to SHUFFLE), and register z records the last output byte produced. Register w is always relatively prime to N—when N is a power of two this just means that w is odd—and w is updated every time WHIP is called.

Absorb: ABSORB takes as input a variable-length input sequence I (of N-values), and updates the Spritz state based on I. Here I might be a cryptographic key, or a portion of a message being hashed.

Spritz does not need to apply a "padding rule" to I to extend it out to a multiple of some desired block size, since input is implicitly padded out to a multiple of $N/4$ bytes with copies of the stop symbol "❚". (The stop symbol is perhaps best viewed as a "missing nibble"—calling ABSORBSTOP() increments register a that counts the number of absorbed nibbles, without actually absorbing any nibble.)

Spritz absorbs the input I in blocks of $\lfloor N/2 \rfloor$ nibbles each (low-order nibble of each byte first). After each block is absorbed SHUFFLE is executed.

For $N = 256$, an input block of $N/2 = 128$ four-bit nibbles contains 512 bits (64 bytes) of information, so each Spritz input block has a size (512 bits) equal to the typical block size of many existing hash functions.

Spritz may ABSORB additional input even after it has produced some output, since ABSORB merely updates the current state without re-initializing it. This corresponds to "duplex mode" in sponge function terminology (see Bertoni et al.[5]), where new input is provided (using ABSORB) alternately with new output being taken (using SQUEEZE).

For a given starting state ABSORB will map distinct inputs I to distinct states, up to the point where SHUFFLE is first called. Similarly, for any fixed input I, distinct starting permutations S yield distinct final permutations, up to the point where SHUFFLE is first called.

An input I may be supplied in pieces, each of non-negative length, using ABSORB on each piece. It doesn't matter how the input is divided into pieces, since ABSORB(X);ABSORB(Y) is equivalent to ABSORB(XY).

AbsorbByte: ABSORBBYTE updates the current Spritz state based on a given input byte (N-value) b. It splits the byte into two nibbles, and updates the state based on each nibble, low-order nibble first.

AbsorbNibble: See Figure 3. ABSORBNIBBLE first tests whether Spritz is "full" of absorbed data (that is, whether $a = \lfloor N/2 \rfloor$); if so, it calls SHUFFLE to mix in the absorbed data and reset a to 0. ABSORBNIBBLE then updates the state based on the value of the supplied nibble (D-value) x, by exchanging $S[a]$ and $S[\lfloor N/2 \rfloor + x]$. (Note that if $N > 2$ then $\lfloor N/2 \rfloor + D \leq N$, so the expression $S[\lfloor N/2 \rfloor + x]$ will not access beyond the end of S.)

AbsorbStop: ABSORBSTOP is the same as ABSORBNIBBLE, except that no swapping is done.

ABSORBSTOP() may be used to ensure that the input from a preceding ABSORB and that of a following ABSORB are cleanly separated. More precisely: in general ABSORB(X);ABSORB(Y) is fully equivalent to ABSORB(XY); putting a ABSORBSTOP() between the two calls to ABSORB ensures that this is *not* true. The call to ABSORBSTOP is equivalent to absorbing a special stop symbol "❚" that is outside the usual input alphabet.

The use of ABSORBSTOP in Spritz replaces the traditional use of "padding rules."

Shuffle: SHUFFLE whips, crushes, whips, crushes, and then whips again. Each whip "randomizes the state." Because CRUSH is called between each pair of calls to WHIP, the effects of CRUSH are not easily determined by manipulating the input, and any biases introduced by CRUSH are smoothed out before SHUFFLE returns. The parameter $2N$ on the size of each WHIP is chosen to produce a strong isolation of SHUFFLE inputs, outputs, and CRUSH inputs/outputs from each other.

Whip: WHIP(r) calls UPDATE a specified number r times. The Spritz system is "being whipped" (stirred vigorously) without producing output. The registers

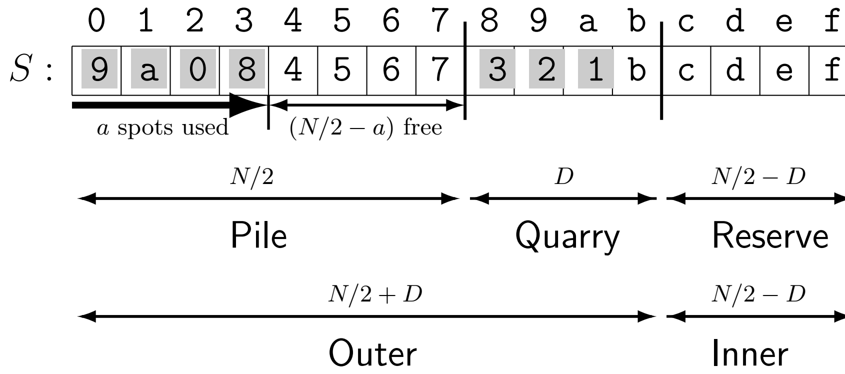

Figure 3: AbsorbNibble diagram. Here $N = 16$, $D = \lceil\sqrt{N}\rceil = 4$. The array S may be viewed as an $N/2$-element "*Pile*", a D-element "*Quarry*", and an $(N/2 - D)$-element "*Reserve*." In sponge function terminology, *Reserve* is the "inner state" *Inner* while the *Pile* and the *Quarry* form the "outer state" *Outer*. From the standard initial state, the four nibbles `1210` have just been absorbed, so that $a = 4$ and *Pile* elements $S[0..3]$ have been updated by exchanges with elements of the *Quarry* determined by the nibbles. Elements that have been exchanged are shown in gray. Absorbing nibble with value `3` next would exchange the *Pile* element $S[4] =$ `4` with the element from the *Quarry* at $S[N/2 + 3] =$ `b` and then increase a to 5. The *Reserve* (inner state) is never modified by AbsorbNibble; the capacity of Spritz is the size of *Reserve*, plus the number of bits in the registers.

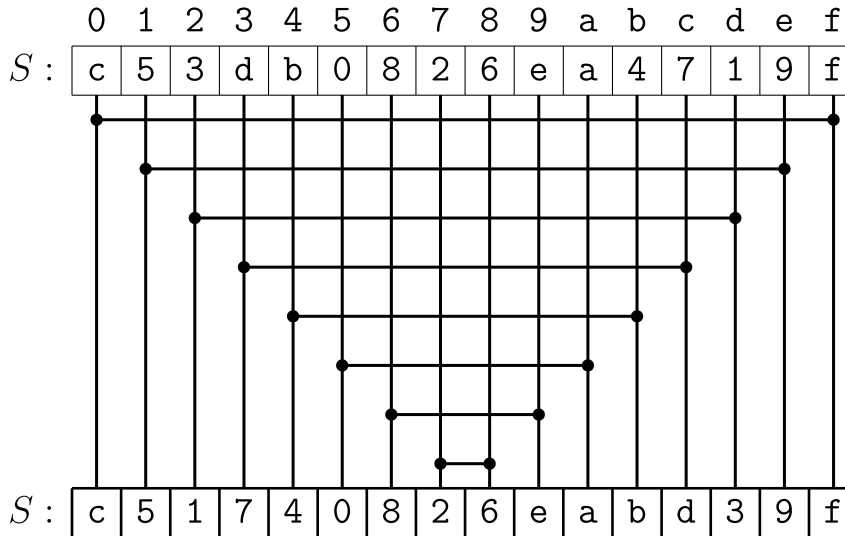

Figure 4: Crush diagram. The N elements of S are considered as $N/2$ pairs, and each pair is sorted into increasing order. The input is at the top, and the output at the bottom. Each horizontal line represents a two-element sorting operation. This operation provides "forward security" for the Shuffle operation.

and permutation S are given new values that is a complex function of their initial values, with larger values of r resulting in more complexity. The use of WHIP reflects a common recommendation for improving $RC4$. The name "WHIP" is intended to suggest the whipping of an egg or of cake batter.

Every WHIP call also updates w to the next larger value that is relatively prime to N. When N is a power of two, the last two lines of the WHIP code in Figure 2 are equivalent to $w = w + 2$. The new value for w is always relatively prime to N, so that the repeated execution of $i = i + w$ in the first line of UPDATE causes i to cycle between all values modulo N. (The cycle followed depends on w.)

Crush: See Figure 4. CRUSH provides a non-invertible transformation from states to states. CRUSH intentionally "loses information" about the current state. More precisely, it maps $2^{N/2}$ states to one, since each of $N/2$ pairs of compared values in S are sorted into increasing order. See Section 7.1.2.

The name "CRUSH" is intended to suggest the many-to-one nature of its mapping.

Squeeze: SQUEEZE is the main output function for Spritz. The name derives from the terminology of sponge functions (see Bertoni et al. [5])—think of squeezing water from a sponge. The input r says how many output bytes (N-values) to produce.

SQUEEZE begins (even if $r = 0$) by calling SHUFFLE if $a > 0$, thereby shuffling any unabsorbed input and putting Spritz into "squeeze mode" ($a = 0$).

SQUEEZE is otherwise equivalent to calling DRIP r times, and returning an array of the r outputs returned.

Drip: Procedure DRIP is the basic pseudorandom output routine designed in Sections 3.1–8.2; each call to DRIP first calls SHUFFLE if necessary (when $a > 0$) to ensure that Spritz is in "squeezing mode," updates the Spritz state using UPDATE, and produces one output byte using OUTPUT.

The test for $a > 0$ and call to SHUFFLE are placed both here, in DRIP, and in SQUEEZE, so that DRIP may be safely called directly by applications, ensuring that absorbed data is always SHUFFLE'd before any output is produced.

Update: UPDATE advances the system to the next state by adding w to i, giving j and k their next values, and swapping $S[i]$ and $S[j]$. Since w is relatively prime to N, the value of i cycles modulo N as repeated UPDATEs are performed.

Output: OUTPUT computes a single byte (N-value) to output, saves this value in register z, and returns this value.

This completes our description of the basic Spritz procedures.

4 Attacks on RC4

We now begin our discussion of the design of Spritz, starting with a brief review of some of the known weaknesses of and attacks on the original RC4.

We do not provide a comprehensive review of RC4 attacks, since our emphasis is on the design of Spritz, but the design of Spritz should be informed by the weaknesses in RC4. For excellent extended reviews of RC4 attacks, see Paul and Maitra [22] and Sen Gupta et al. [27]. Our sketch here follows the outline given in [27].

Weak keys and key recovery from state An ideal key-setup algorithm would take as input a key K and produce as output an RC4 state that is indistinguishable (to an adversary who does not know the key) from a state chosen uniformly at random from the set of all states. Unfortunately, the RC4 KSA did not come close to this goal.

Initial key-dependent biases In 1995, Roos [25] observed that for small v, $S[v]$ is highly correlated with the bytes of key K.

Key collisions One would hope that different keys would not produce the same state (a "key collision")—since this means the colliding keys produce the same output keystream. But Matsui [15]

produced such collisions for RC4; further work by Chen and Myaji [7] has produced short (22-byte) colliding key pairs.

Key recovery from internal state The problem of recovering the key from the internal state was first studied by Paul and Maitra [21], with further contributions made by Biham and Carmeli [6], Khazaei and Meier [11], Akgün et al. [1], and Basu et al. [3]. These results are still being improved, and one can reasonably conclude that RC4 provides little protection against key recovery from the internal state.

Key recovery from the keystream Fluhrer et al. [9] demonstrated that the approach used by WEP (the Wired Equivalent Privacy protocol, part of the 802.11 standard), in which a secret key is concantenated with a public key to form the working key, is insecure—they give a powerful related-key attack. See comments by RSA Labs [26], improvements to the attack by Klein [12], and the implementations of this and related attacks by Stubblefield et al. [31, 30], and by Tews et al. [32]. See also Maitra et al. [13]. Sepehrdad et al. [29] provide a practial attack on WPA (the WEP successor) that recovers the RC4 key using 2^{38} packets and complexity 2^{96}. Also see Sepehrdad's PhD thesis [28].

State recovery attacks A number of authors have work on the RC4 *state recovery problem*—determining the RC4 internal state from a known output sequence.

Maximov and Khovratovich[17] have perhaps the most efficient of the known state-recovery attacks, with a complexity of 2^{241} (both time and keystream) under reasonable assumptions.

Golić and Morgari [10] describe an interesting iterative probabilistic algorithm that can reconstruct an internal RC4 state from an output keystream of length 2^{41} using computation time 2^{689}.

Biases and Distinguishers The output biases of RC4 are of two types: *short-term*, and *long-term*. Short-term biases disappear once enough keystream bytes are produced (or discarded as part of the KSA), while long-term biases are persistent.

In 2001, Fluhrer, Mantin, and Shamir [9] published an analysis of RC4's Key Scheduling Algorithm (KSA), identifying a number of weak keys, for which knowledge of a small part of the secret key implies knowledge of many bits of the KSA output state. Fluhrer et al. [9] recommend discarding the first N output bytes (so that the corresponding state updates become part of KSA).

Paul and Maitra [22, p. 26] show that the RC4 KSA has "intrinsic" biases–changing the KSA update function for j will not remove such biases. However, increasing the number of KSA rounds will help.

In 2002, Mironov published an analysis [19] of RC4 focused on determining the number of initial output bytes that should be discarded in order for the resultant state to be indistinguishable from random, using a coupling argument; he recommends that at least 512 bytes be so discarded.

In 2001, Mantin and Shamir [14] published an distinguishing attack on RC4 based on the observation that the second keystream byte is biased towards zero: it occurs with probability about $2/N$, instead of the desired $1/N$.

In 2013, Alfardan et al. [2] gave plaintext attacks on RC4 based on extensive statistical analyses of single-byte biases in the RC4 output. Their work stimulated the work presented in this paper.

Our own statistical tests support the conclusion that the RC4 rule for updating j, ($j = j + S[i]$) was not a particularly good one.

5 Desired Security Properties

This section describes some desired security properties for Spritz, formalizing issues identified in the last section.

There are two security parameters for Spritz: the value N and the length L of the supplied cryptographic key (for those operations that have a key). In practice, we expect to have $N = 256$ and $L = 16$ or $L = 32$. In theory, we want security levels specified in terms of N and L.

We assume that $L \leq N/4$, so that the number of

pairs of keys is a negligible function of the number of Spritz states.

The capacity $c = c(N)$ of Spritz is also relevant for some security analyses (as it is for sponge functions); for Spritz we have $c(N) = (\lceil N/2 \rceil - D) \cdot \lg(N)$ bits where $D = \lceil \sqrt{N} \rceil$.

We introduce useful notations for Spritz states and the transformations of states induced by various Spritz procedures. In general, we use Q to denote some Spritz state, Q_0 to denote the state produced by InitialState(), and $Q_R()$ to denote a state picked uniformly at random.

We use abbreviations for the 12 Spritz procedures of Figure 2: a procedure's abbreviation consists of the first two letters of each of its words (which is the same as its first two letters, except for two-word names: AbBy for AbsorbByte), AbNi for AbsorbNibble, and AbSt for AbsorbStop.)

We denote the state resulting from applying procedure Xy to state Q as Q.Xy() (even in cases like Drip or Squeeze whose usual output convention is to return a byte or byte sequence; here our notation reflects the state transformation made).

The output produced by a given expression is denoted by underlining the expression, so that

$$\underline{Q_0.\text{Sq}(r).\text{Sq}(\text{s})}$$

denotes the $r + s$ byte output produced by running Squeeze(r) and then Squeeze(s) from the initial state, and

$$\underline{Q_0.\text{Ab}(M).\text{Ab}(r).\text{Sq}(r)}$$

denotes the r-byte Spritz Hash of message M (see Section 2.3).

Our notation allows Ab's (Absorb's) argument to be a *generalized input sequence* that may contain instances of the stop symbol "❙", with the natural interpretation, as in:

$$Q.\text{Ab}(K \,❙\, M) = Q.\text{Ab}(K).\text{AbSt}().\text{Ab}(M)$$

We call an ordinary input sequence a "*string*" if it contains only ordinary input characters (N-values); we call it a generalized input sequence or a "*chain*" if it may also contain stop symbols but does not end with a stop symbol. The definition of a chain is merely for notational convenience, as a real implementation would not have access to stop symbols, and would use the equivalent AbsorbStop interface.

For further notational conciseness, we introduce two useful notations.

Definition Let

$$Q(K) = Q.\text{Ab}(K).\text{Sq}(0)\ .$$

Thus, $Q_0(K)$ is the state of the Spritz system after key setup with key K and the system enters "squeezing mode".

Definition Let

$$Z(Q, r) = \underline{Q.\text{Sq}(r)}\ .$$

Thus, $Z(Q, r)$ is the output produced by starting in state Q and squeezing out r bytes. In particular,

$$Z(Q_0(K), r) = \underline{Q_0.\text{Ab}(K).\text{Sq}(r)}$$

is the r-byte Spritz string produced for key K.

Pseudorandom state from key It should be infeasible for an adversary to distinguish between the behavior of the procedure $Q_R()$, which produces a state generated uniformly at random, and the behavior of the procedure $Q_0(\mathcal{K}_L())$, which randomly picks an L-byte key K using the procedure $\mathcal{K}_L()$ and then returns $Q_0(K)$.

By "infeasible" we mean that an adversary that only uses enough computing resources sufficient to test an ϵ fraction of $\mathcal{K}_L$ (although it could use some other approach) will have an advantage over distinguishing by random guessing of at most ϵ.

Collision resistant key setup It should be infeasible for an adversary given a randomly chosen state $Q = Q_R()$ to find two distinct chains K and K' such that

$$Q(K) = Q(K').$$

That is, the adversary should have a negligible chance of finding a pair of chains that result in the same

state when when Spritz absorbs them and prepares for squeezing.

See [15, 7] for examples of such state collisions for RC4.

Here "infeasible" means that adversary that runs in time t has a chance $O(t^d/\#(N))$ of finding such a state collision, for some integer d. We allow $d > 2$ here in this definition, to account for the effective reduction in state-space caused by CRUSH.

(One would like this definition to work for $Q = Q_0$ as well. However, this variation doesn't actually work as a definition, since there exist programs which merely print K and K' in this case (collisions exist!). Perhaps it is enough to say that "in real life" one should never see an example of such a collision. See Rogaway [24] for an excellent discussion of these issues.)

One-wayness from key (pre-image resistance): It should infeasible for an adversary given $Q = Q_0(K)$ for a randomly chosen key $K = \mathcal{K}_L()$ to find a key $K' \in \mathcal{K}_L$ (possibly equal to K) such that $Q = Q_0(K')$.

That is, it should not be possible to determine an encryption key from the Spritz state produced from key setup.

Here "infeasible" means that an adversary does not have significantly more chance of success inverting the key-to-state mapping of Spritz than he would inverting a random 2^N-to-1 mapping ϕ from a set $\mathcal{S}$ of size $\#(N)$ to itself where only one in N^c elements of $\mathcal{S}$ are allowed as pre-images, and where c is the capacity of Spritz.

Output-collision-resistance: It should be infeasible for an adversary given a random state $Q = Q_R()$ and an integer input $r > 0$ to find two distinct chains K and K' such that

$$Z(Q(K), r) = Z(Q(K'), r) \ .$$

That is, an adversary should have a negligible chance of finding two chains for which Spritz produces the same r-byte output. Of course, this definition only makes sense if the running time of the adversary is constrained to be $o(N^{(r/2)})$, so we make this restriction.

Pseudorandomness: The SQUEEZE output should appear random (and uncorrelated with the key).

That is, $Z(Q_0(K), r)$ for a randomly chosen key $K \in \mathcal{K}_L$ should be indistinguishable to an adversary from an r-byte string chosen randomly from the set of all r-byte strings.

Resistance to related-key attacks: The Spritz SHUFFLE procedure is designed to prevent related-key attacks, as it is designed to "randomize" the state prior to any SQUEEZE.

We do not propose here a definition of security against related-key attacks.

Resistance to length-extension attack The security of the given MAC construction depends on the details of how ABSORB and SQUEEZE are implemented. For example, it is vulnerable to a *length-extension attack* [18, Sec. 9.64, p. 355] if one can determine $\text{MAC}(K, M', r)$ from $\text{MAC}(K, M, r)$, where M' is an extension of M (that is, M followed by additional characters).

Spritz is not vulnerable to a length-extension attack, since SHUFFLE is called before any output from SQUEEZE is produced, preventing the output from being used to determine the internal state just after M was absorbed.

6 Design Process

This section describes the design process and our rationale for various design decisions.

At a high level, the design splits into two parts: the design of the input process (ABSORB), and the design of the output process (SQUEEZE). The design of ABSORB is described in Section 7; the design of SQUEEZE is described in Section 8.

The design of the ABSORB is motivated by the sponge construction of Bertoni et al. [5]. Absorbing input only affects a portion of Spritz state; there is a reserve "capacity" that is untouched by the input.

The design of the SQUEEZE procedure first determined a set of seven robust candidates for the UPDATE procedure that updates registers i, j, and k. Then four strong candidates for OUTPUT to produce an output byte z from the registers and the permutation S were determined. This resulted in twenty-eight candidates for DRIP; statistical evaluation provided guidance on the final selection.

We first describe here our use of postfix notation for expressing alternatives, say a few words about our use of chi-squared tests for uniformity of distribution, and comment on the issue of "bad keys" and bad starting states.

Postfix notation We used the following postfix (reverse polish) notation for expressions. We have register names `i, j, k, w, z`; constants `1, 2, ...`; addition modulo N represented as `p` (binary operator); and `S` for array access (unary operator). (We used "`p`" for "+" so postfix expressions are usable as identifiers in programs.) For example:

$$\texttt{iSjSSpS} \equiv S[S[i] + S[S[j]]] \ .$$

Statistical tests Our statistical tests are mostly tests for uniformity of distribution: we arrange a set of trials, and categorize each trial's outcome as one of q possible outcomes, for some value q. We expect that the test (for a good cipher) will yield an empirical distribution that is uniform, or extremely close to uniform. Large deviations from uniformity are interpreted as conclusive evidence that the cipher design is flawed.

A simple test might look at just the distribution of Spritz output values, so $q = N$. A more elaborate test might examine pairs of successive output values, so $q = N^2$. Similar tests for correlations between various register values, various output values, and values in the S table form the heart of our statistical testing.

We measure deviation from uniformity using Pearson's chi-squared statistic:

$$\chi^2 = \sum_{u=0}^{q-1} (O_u - E_u)^2 / E_u \tag{3}$$

where u ranges over the q possible outcomes, O_u denotes the number of outcomes observed of type u, and E_u denotes the expected number of outcomes of type u (which is $E_u = T/q$ where T is the total number of trials). Our experiments are consistent with the guidance that E_u should always be at least five.

The number *df* of degrees of freedom in our chi-squared tests will typically be $N - 1$ when $q = N$, and will typically be $(N - 1)^2$ when $q = N^2$. The expected value of χ^2 with k degrees of freedom is k, and the standard deviation is $\sqrt{2k}$.

We consider a uniformity test to *fail* if its chi-squared statistic is greater than *four* standard deviations above its expected value.

Testing small-N versions The RC4/Spritz design nicely supports various values of N. Statistical tests for *small* N detects biases more easily. The literature on RC4 (e.g. Paul and Maitra's book [22]) shows that many biases show up as probabilities that are roughly $(1/N)^2$ away from their expected values. Since a deviation of size ϵ requires about $1/\epsilon^2$ samples to detect, testing designs with small values of N dramatically reduces the amount of testing required. We test designs with N as small as 16; a design that fails at $N = 16$ is deemed not suitable for use with larger N. That said, our final design has also been tested extensively at $N = 256$.

Short cycles and bad keys Many of our tests involve generating a pseudorandom sequence from a given starting state, and then computing a chi-squared statistic based on output values. However, we wish to be sensitive to the possibility of the existence of bad starting states that cause Spritz to enter short cycles or otherwise exhibit poor behavior. Therefore our experiments also test a large number of different randomly chosen starting states. We summarize the results by computing the chi-squared statistic corresponding to the combined distribution of all pseudorandom sequences generated for these starting states. In this way, the existence of bad starting states will be made readily apparent, as those starting states will make a large contribution to the chi-squared statistic of the combined distribution. (For many experiments, we additionally considered the maximum chi-squared statistic of the

individual starting states, but since the chi-squared statistic of the combined distribution better captures statistical abnormalities present across different starting states, we found this to be a better overall measure.)

See Paul and Maitra [22, Section 5.1] for further discussion of “Finney cycles” (a set of bad starting states for RC4).

7 Absorb design

The Absorb procedure accepts a sequence of input data, and effects a corresponding transformation of the state.

If the data sequence is not too long (at most $N/4$ bytes), then the transformation is one-to-one (collision-free). That is, it is not the case that there exists a starting state and two (short) inputs such that the effect of Absorb on the given starting state under the two inputs results in the same final state.

AbsorbNibble design Because of the sponge-like framework adopted for Spritz, a portion of the state (called the “*Reserve*” or the “capacity”) is left untouched by any Absorb operations.

Moreover, because S is a permutation and not just an array of bytes, Absorb must proceed by swapping elements of S.

To keep the elements of S chosen for swapping from ranging over all of S, we decompose an input byte b into its two nibbles x and y, and can then use x and y to compute indices for elements to swap. More precisely, the two elements swapped for an input nibble x are $S[a]$ (where a is the number of elements absorbed previously, since initialization or the last Shuffle), and $S[N/2 + x]$, and similarly for y.

These Absorb operations never modify the last 112 bytes of S; the “capacity” of Spritz is 896 bits (112 bytes) plus the size of the registers.

We note that an observer examining the state of S both before and after a nibble is absorbed can determine exactly the value of the nibble that was absorbed, since each AbsorbNibble changes exactly one value in *Pile*, and the value is determined by seeing which position in *Quarry* was changed.

AbsorbStop design AbsorbStop performs identically to AbsorbNibble, except that it performs no swapping. The counter a giving the number of absorbed nibbles is nonetheless advanced.

(We note for the record that an alternate, but much less efficient, way to implement AbsorbStop would be simply as a call to Shuffle.)

7.1 Shuffle procedure design

This section discusses the design of Shuffle.

Roughly speaking, Shuffle follows the oft-made suggestion that RC4 key-setup should be followed by discarding a certain number of output bytes (which is effectively what Whip does). However, Shuffle also includes two calls to Crush, which makes the Shuffle state-transformation many-to-one. Finally, we note that Shuffle is not only invoked whenever Spritz switches from “absorbing mode” to “squeezing mode”; it is also invoked after every $N/4$ bytes of input are absorbed.

7.1.1 Whip design

Calling Whip(r) procedure is intended to be effectively equivalent to calling Drip() a total of r times in a row, and throwing away the output. It “stirs the pot” without producing any output.

This near-equivalence can be seen by noting that Drip calls Update and then Output, whereas the Whip loop body has only a call to Update. The equivalence is not exact, since Output also modifies Spritz state: it modifies register z.

We note that Whip need not start with $i = 0$; it starts with whatever value i had previously.

The calls to Whip in Shuffle specify $r = 2N$: two passes over S are made. The rationale choosing $r = 2N$ are as follows:

- Mironov [19] recommends that RC4 should be modified to discard “at least 512 bytes”. This corresponds to $r = 2N$ when $N = 256$. His theory suggests that some biases may not disappear until at least the third or fourth pass.

- The construction of Spritz does not have any mixing going on before the call to SHUFFLE; so Mironov's recommendation might be interpreted as discarding at least 768 bytes in the context of of Spritz, since Spritz doesn't have the initial pass that the RC4 KSA does.

- SHUFFLE has *three* calls to WHIP, so the "effective argument" is really $r = 6N$. (The call to CRUSH between the two calls to WHIP may have some effect on the "effective r".)

- Our experiments (see Section 7.2 gives evidence that our choice of $r = 2N$ for each call provides strong mixing and removal of biases.

7.1.2 CRUSH design

CRUSH implements a many-to-one mapping of states to states, by considering the N elements of S as $N/2$ pairs, and sorting each pair into increasing order. Thus, CRUSH implements a $2^{N/2}$ to 1 mapping.

The rationale for including CRUSH in Spritz is to prevent an adversary who somehow learns a Spritz state (say by stealing a Spritz device while it is still in use) from "working backwards" to determine the key K. Once the adversary works backward to the CRUSH call, he is faced with trying to decide which of the $2^{N/2}$ predecessor states is the right one to continue working on.

CRUSH thus provides a "firewall" against this kind of backwards reasoning attack. (See Biham et al. [6] and Khazaei et al. [11] for such attacks on RC4.) CRUSH therefore provides some "forward security."

The "information loss" effected by CRUSH may be viewed from different perspectives, depending on the application and context in which Spritz is used.

In an encryption context (say with 16-byte keys), the number of reachable states (N^{16}) is small compared to the total number $\#(N)$ of states, and it would be very surprising if there exists two such reachable states that CRUSH caused to collide, assuming that WHIP acts pseudorandomly. One would expect, with very high probability, that each encryption key will produce a distinct pseudorandom output stream. (More precisely, this probability is negligible until the number of possible keys starts approaching the square root of the number of states divided by 2^{128}, and even with 80-byte (640-bit) keys we are nowhere near this threshold.)

The fact that the calls to CRUSH in SHUFFLE sit between calls to WHIP means that the state input to CRUSH has already been "randomized" (supporting the argument in the previous paragraph), and it means that the pattern in the output of CRUSH (that is, that each pair of elements in S is now in sorted order), is diffused and lost by the following WHIP.

In a hashing context, inputs may be long, so that state collisions are in any case unavoidable. The relevant question would be whether the use of CRUSH provides any leverage for an adversary to defeat the collision-resistance property desired for the hash function. Because of the WHIPs preceding the calls to CRUSH, and because the capacity of Spritz puts severe restrictions on the manipulations an advesary can make to the state by supplying input to be absorbed, we conjecture that CRUSH provides no effective help to an adversary trying to defeat hash function collision-resistance.

When Spritz is initialized with secret parameters (e.g., a key for encryption or for a MAC), one may be concerned about timing attacks, where an adversary might be able to gain information about the secret by observing the time taken to perform key setup. For example, if the Spritz state doesn't fit in the machine's cache, the running time may depend on the exact pattern of access to the entries of S. Similarly, the conditional branching in CRUSH may cause timing variations. With some machines, implementing CRUSH as follows will help, as both branches of the **if** statement will have equal running time.

```
CRUSH()
1  for v = 0 to N/2 − 1
2      x = S[v]
3      y = S[N − 1 − v]
4      if x > y
5          S[v] = y
6          S[N − 1 − v] = x
7      else
8          S[v] = x
9          S[N − 1 − v] = y
```

The loss of timing information from CRUSH may actually not be a concern in most applications. One may consider a "leaky" version of Spritz that leaks $N/2$ bits from each call to CRUSH, leaking which branch of the **if** was taken in each loop iteration. Because each call to CRUSH is surrounded by two calls to WHIP, the information leaked is conjectured to be of little use to an adversary. The exception would be for the adversary who is "working backwards" to find the initial state, given the state at some later time; in this case the available timing information has removed the protection provided by CRUSH.

SHUFFLE actually calls CRUSH twice, so an adversary trying to work backwards through SHUFFLE is facing an (apparent) 2^N to one mapping.

The use of a many-to-one map like CRUSH means that Spritz provides a protection that is not provided by ordinary sponge constructions: if the *complete* state of the system is compromised (a possibility not contemplated in the sponge framework), then the presence of CRUSH means that the adversary can no longer easily "reason backwards" to infer the key.

We note that in the sponge function paper [5], in addition to random permutations, random transformations are considered. But the in-degree of a node in the range of such a random transformation is still a small constant, compared to the exact in-degree $2^{N/2}$ achieved by CRUSH. Therefore, Spritz is qualitatively rather different from the transformation-based sponge constructions previously studied.

7.2 SHUFFLE, WHIP, and CRUSH

We evaluated the quality of the mixing provided by SHUFFLE and WHIP by examining the correlations between the bytes of random keys that were absorbed, and the bytes of the subsequent SHUFFLE or WHIP output. We found that two WHIP passes did not eliminate all correlations, but that three passes did a very good job. SHUFFLE, which contains six WHIP passes, as well as two calls to CRUSH also did an excellent job of eliminating such correlations.

In addition to studying the correlations between inputs and outputs, we also studied the correlations between various pairs of output bytes in S, when random keys were absorbed and then the state was shuffled. SHUFFLE was again found to perform well at eliminating such correlations. (Details to be provided in expanded version of this paper.)

8 SQUEEZE design

We systematically and automatically generated a large number of SQUEEZE candidate designs, then filtered them for desired structural properties and lack of detectable statistical biases. Then the simplest remaining candidate was chosen as our proposal for Spritz.

A structured search We begin our description of how Spritz (specified above) was designed.

We consider only designs having the following general form. Here E_j, E_k, and E_z represent expressions (to be determined) of the registers i, j, k, z and the array S in the current state. This framework does not show the call to SHUFFLE (if $a > 0$) present in the final formulation, as it is not relevant to the statistical properties we are now considering. This formulation allows for additional flexibility not only in terms of the expressions used, but also in the use of the new register k, as well as allowing the previous output value z to be used in an expression.

DRIP()
1 UPDATE()
2 **return** OUTPUT()

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

OUTPUT()
1 $z = E_z$
2 **return** z

We thus consider only designs involving variant choices for E_j, E_k, and E_z.

We retain the RC4 design decision that register i cycles modulo N, in order to ensure faster mixing

of the state array S. The increment to i is, however, now a value w that need only be relatively prime to N to ensure that i cycles through all values modulo N.

Which such expressions E_j, E_k are good cryptographically?

Section 8.1 describes our approach to the design of the UPDATE procedure, while Section 8.2 describes our approach to the design of the OUTPUT procedure.

8.1 UPDATE procedure design

This section explains our design of the UPDATE procedure.

8.1.1 Another register

The "sponge-like" formulation of Spritz and its potential applications put a heavier burden on the UPDATE function design. For example, when Spritz is used as a hash function it should be infeasible to find hash function collisions.

We retain the RC4 design principle that register i should just cycle modulo N, so that each position of S may be swapped at least once in each round. But the increment w is now an arbitrary value relatively prime to N.

At a high level, the purpose of UPDATE is to compute a pseudorandom value j so that $S[i]$ and $S[j]$ may be swapped. The value of j should not only be pseudorandom, but hidden from an observer of the output, so that the observer may not track the evolution of S.

When Spritz is used as a hash function, however, it is important that the output depend on every portion of the input. The RC4 swapping strategy risks moving an element of S into an earlier position where it may not be touched for a while, and its value may not affect the evolution of S quickly enough.

We thus adopt the design principle that UPDATE should not only swap $S[i]$ with $S[j]$, but should also ensure that Spritz state evolution depends on those two values swapped.

One could try to achieve this goal by having the update rule for j depend upon both $S[i]$ and $S[j]$, via an update rule such as $j = S[i] + S[j]$. There are, however, *two* bytes of information in $S[i]$ and $S[j]$, and only one byte of information may be stored in j. Many pairs $S[i]$,$S[j]$ will have identical effects on j, resulting in the possibility of easily-found collisions.

We conclude that *a single hidden register j is insufficient for our purposes, and we add a new register k to the design of Spritz.*

8.1.2 Initial criteria

We list some simple syntactic criteria for E_j and E_k:

invertibility The expressions E_j should contain exactly one occurence of variable j, to ensure that the expression $j = E_j$ is invertible (reversible). Similarly for E_k: it should contain k.

must have S We only consider expressions that involve at least one application of S. We furthermore require that the update rule for j depends on $S[i]$, and similarly, that the update rule for k depends on $S[j]$.

no SS We only consider expressions that do not apply S twice in a row.

no doubling We do not consider expressions that add a value to itself.

no constants We do not use numerical constants.

length We do not consider expressions for j and k whose combined length (in postfix) exceeds 14.

We automatically generated all (E_j, E_k) candidate pairs satisfying the above criteria. This resulted in a total of 147 candidate pairs, the simplest having a combined length of 10 (see Appendix A for a complete list of the candidate pairs).

Based on this list of candidate pairs, we proceeded by systematically removing candidate pairs not providing desirable properties explained in the following sections.

8.1.3 "Swap-proof" candidates

We require more generally that the update rules should be "swap-proof": it should not be possible to swap two S values accessed during two successive

DRIP calls and leave the evolution of j and k unaffected. This more general condition helps keep an adversary from constructing collisions by swapping two values in S that are updated on consecutive calls to DRIP. Testing this condition was done by extensive simulations; noting if performing such swaps could leave the evoluation of j and k unaffected.

Among the 147 candidate pairs, 122 were found not to be "swap-proof" by the above described test, leaving just 25 candidate pairs.

8.1.4 Initial statistical testing

For the purpose of constructing a good output function E_z, it is desirable that the values of j and k obtained by evaluating the update functions E_j and E_k, are independent. To test the quality of candidate pairs with respect to independence of j and k, we considered the combined distribution of the variables i, j and k (which we will denote `ijk`) obtained by choosing a random starting state and successively evaluating UPDATE. As described in Section 6, we use Pearsons chi-squared statistic to evaluate the smoothness of the distribution, and consider a candidate pair to fail if the chi-squared statistic of the corresponding `ijk` distribution is greater than four times the standard deviation above its expected value.

In addition to `ijk`, we considered the distribution of two consecutive values of j and k. That is, we consider the combined distribution of the variable i, the variable j from the previous state (that is, before the previous call to UPDATE), and the variable j from the current state. We denote this distribution `ij1j` (here, "1" indicates that the value from the previous state of the preceding variable is considered). Likewise, we considered the distribution `ik1k`. Lastly, we consider the "mixed" distributions `ij1k` and `ik1j`.

For each candidate pair (E_J, E_k), we generated the above described distributions using $N = 16$, 2^{16} random starting states, and 2^{16} consecutive calls to DRIP for each starting state (that is, the distributions were generated using a total of 2^{32} calls to DRIP). Among the 25 candidate pairs, 18 failed the above described tests, leaving 7 candidate pairs.

id	E_j	E_k
3	$j = j + k + S[i]$	$k = S[k + S[j]]$
9	$j = k + S[j + S[i]]$	$k = S[j + k]$
34	$j = j + k + S[i]$	$k = k + S[i + S[j]]$
35	$j = j + k + S[i]$	$k = i + S[k + S[j]]$
36	$j = j + k + S[i]$	$k = S[i + k + S[j]]$
41	$j = j + S[k + S[i]]$	$k = i + k + S[j]$
49	$j = k + S[j + S[i]]$	$k = i + k + S[j]$

Figure 5: UPDATE candidate pairs (E_j, E_k) satisfying structural requirements and passing initial statistical tests.

8.1.5 The seven finalists for UPDATE

As a result of the above tests, we obtained the final list of (E_j, E_k) candidate pairs shown in Figure 5.

8.2 Four finalists for OUTPUT

If register k were not to be used in E_z, then one interesting contender here is perhaps the RC4 choice, $E_z = \texttt{iSjSpS} = S[S[i] + S[j]]$.

The resistance of Spritz to attack depends critically on the properties of E_z. The RC4 version has been around for quite a while now, and should be replaced for that reason alone.

In order to make cryptanalysis more difficult while maintaining efficiency, we decided on the following structural requirements for E_z:

- E_z should end (in postfix) in "S", but should not contain any subexpression of the form $S[S[\cdot]]$.
- E_z should contain both j and k, but should not contain any variable i,j,k, or z more than once.
- E_z should not contain any of $S[i]$, $S[j]$, $S[k]$, or $S[z]$ as subexpressions.
- We do not consider E_z expressions with a length exceeding 10 (in postfix notation).

We generated all E_z expression satisfying the above structural requirements, which resulted in a list of 33 E_z candidates. These candidates were each given an

id number from 1 to 33 (see appendix A for the full list of candidates).

The number of E_z candidates was further reduced by the following heuristic arguments.

- If two E_z expressions are identical except that one is using the variable i and the other the variable z, we eliminate the former. While an attacker might know the value of both i and z in some attack scenarios, the value of z is considered to have more uncertainty from the attacker's viewpoint, and hence we consider the expression using z to be superior.

- All expressions using the subexpression $i+z$ are eliminated. Since i is considered to be known to the attacker, identical expressions using just z are preferred due to their simplicity and better efficiency.

- Expressions using variables j or k in the outer layer (e.g. $S[j+S[i+S[k+z]]]$) are preferred to candidates using the variables i or z. This is perceived to increase uncertainty, from the attacker's viewpoint, of the index in the state S which is used to produce the final keystream byte.

- The variables j and k are designed to act as independent variables, and expressions in which j and k are swapped are heuristically considered to be equivalent. Of these expressions, we consider only the ones having the variable j in the outer layer. (This heuristic decision might have been made the other way with similar end results.)

- Lastly, since the value of the variables i and z are potentially known to the attacker, expression which are identical except that these variables are swapped, are heuristically considered to be equivalent. Of these expressions, we prefer the ones where z appear in the innermost layer to increase the uncertainty of the initially used index of the state S.

Applying the above heuristic elimination rules to the initial list of 33 candidates for OUTPUT, we obtained the four candidates shown in Figure 6.

id	E_z
1	$z=S[j+k]$
6	$z=S[j+S[k+z]]$
22	$z=S[S[i+j]+S[k+z]]$
33	$z=S[j+S[i+S[k+z]]$

Figure 6: OUTPUT candidates E_z satisfying structural requirements and passing heuristic elimination of inferior candidates.

8.3 Final DRIP candidates

The combination of seven candidates for (E_j, E_k) and four candidates for E_z makes for twenty-eight DRIP candidates. We give a DRIP candidate an id of $xxyy$ if xx is the (E_j, E_k) (UPDATE) id number and yy is the E_z (OUTPUT) id number; so the final twenty-eight candidates are numbered:

$$301, 306, \ldots, 901, 906, \ldots, 4922, 4933\ .$$

8.3.1 Further statistical testing

We performed additional statistical analyses to see which of the twenty-eight candidates could be eliminated, and to select the final Spritz winning DRIP combination.

To evaluate the candidates, we considered an extended set of tests. More specifically, using the notation from Section 8.1.4, we considered chi-squared statistic of the distributions

$$\texttt{ijQj}, \texttt{ikQk}, \texttt{izQz}$$

for $\texttt{Q}=1,\ldots,N$, and the distributions

$$\texttt{ijQk}, \texttt{ijQz}, \texttt{ikQz}$$

for $\texttt{Q}=0,\ldots,N$. (Note that for distributions not involving the variable z (that is, `ijQj`, `ikQk`, and `ijQk`), the candidates with the same update expressions (E_j, E_k) will behave identically.) Since an adversary will not have direct access to the internal variables j and k, but might be able to learn z (e.g. via partially known plaintext attack), we place greater

UPDATE id	χ^2 bias of ij2j (#σ from uniform)
3	16178
9	1000
34	16927
35	16184
36	16207
41	1657
49	997

Figure 7: Biases in the chi-square statistic of the distribution ij2j for the considered UPDATE candidates.

emphasis on the distribution involving the variable z than distributions only involving variables i, j and k.

For $N = 16$, 2^{16} random starting states, and 2^{16} calls to DRIP for starting state (that is, 2^{32} calls to DRIP in total), 13 of the 28 candidates showed a bias of more than four times the standard deviation in distributions involving the variable z. More specifically, all candidates using OUTPUT candidate 1, and all candidates using OUTPUT candidate 6, with the exception of candidate 4106, showed a bias in the chi-squared statistic of either the distribution ij1z or the distribution iz2z.

While the remaining 15 candidates did not show any biases for the distributions containing the variable z, biases in the distribution of the internal variables i, j, and k were detectable. In particular, all the considered UPDATE candidates showed the strongest bias in the ij2j distribution. This bias was significantly stronger for UPDATE candidates 3, 34, 35, and 36, while candidates 9 and 49 showed the least significant biases, as can be seen from the table shown in Figure 7. Based on this, we reduced the list of considered OUTPUT candidates to 922, 933, 4922, and 4933.

To further evaluate the last four OUTPUT candidates, we considered the above described distributions involving the variable z for $N = 16, 32, 64$ and for 2^{18} random starting states and 2^{18} calls to DRIP for each starting state. This revealed the detectable biases in the iz2z and iz3z distributions for $N = 16$,

DRIP id	χ^2 bias (#σ from uniform)	
	iz2z	iz3z
922	17.63	14.68
933	5.89	19.96
4922	19.89	14.79
4933	< 4	17.88

Figure 8: Biases in the chi-square statistic of the distributions iz2z and iz3z for the considered combinations of UPDATE and OUTPUT candidates.

as can be seen in Figure 8 For the considered distributions, no biases were detected for $N = 32$ and $N = 64$.

While DRIP candidates 933 and 4933 seem to have similar characteristics, we chose 4933 as the wining candidate.

8.3.2 Extrapolation of biases

The statistical biases in the output of our winning candidate are really very faint (*much* fainter than those of RC4), and we were unable to measure them directly for $N = 256$. However, the biases were detectable for smaller values of N. This section explains how we extrapolated the biases observed for $N = 16$, 24, and 32 to estimate the statistical biases of Spritz for $N = 256$.

As observed above, the winning candidate 4933 has detectable biases in the keystream for $N = 16$. In particular, of the considered distributions, iz3z contains the strongest biases. In the following, we estimate the strength of these biases for $N = 256$.

Evaluation of the original RC4 suggests that the relative strength of biases in this type of construction can be described as

$$\epsilon = c \cdot N^{-d}$$

where the parameters (c, d) are dependent on the particular bias considered. Assuming that the biases in Spritz are all of the above form, that is, assuming that the expected number of occurrences of an iz3z triple can be described as

$$TN^{-3}(1 \pm c \cdot N^{-d}),$$

N	#keystream bytes	χ^2 bias of `iz3z` (#σ from uniform)	$\log_2(\epsilon)$
16	2^{46}	19173.04	-12.6367
24	2^{48}	492.79	-15.8389
32	2^{51}	118.20	-18.0575

Figure 9: Biases measured in the `iz3z` distribution of final Spritz candidate.

where T denotes the number of generated keystream bytes (that is, calls to DRIP), the expected value of the chi-square statistic for the `iz3z` distribution can be estimated as

$$\mathrm{E}[\chi^2] = N^3 + T \cdot \left(c \cdot N^{-d}\right)^2 \quad (4)$$

(See appendix C for details.) Note that this estimate assumes that all biases in the considered distribution can be captured using a single set of parameters (c, d).

Based on the above, we estimated the value of the parameters (c, d) for the `iz3z` distribution of the winning candidate 4933. More precisely, for $N = 16, 24, 32$, we obtained estimates of the chi-square statistic by considering distributions generated by 2^{44}, 2^{47}, and 2^{51} total calls to DRIP, respectively. This yielded the results shown in Figure 9. Fitting the measurements in Figure 9 to the expression $\epsilon = c \cdot N^{-d}$ for the relative bias using the method of least squares, we obtained the following estimate for (c, d):

$$c = 589.5107 \qquad d = 5.4600$$

Finally, using the obtained parameters (c, d), we extrapolated the relative bias of the chi-square statistic for the `iz3z` distribution to $N = 256$, which yielded $\epsilon = 2^{-34.4768}$.

Based on the biases in the `iz3z` distribution, it is possible to mount a distinguishing attack against Spritz for $N = 256$. In particular, an adversary capable of observing sufficiently many keystream bytes such that the expected chi-square statistic for Spritz will deviate with one standard deviation from that of a truly random sequence of bytes, will be able to obtain a constant advantage (winning about 0.69146% of the time) by basing his guess on the value of the chi-square statistic of the observed distribution (that is, guessing "Spritz" if the chi-square statistic of the observed distribution deviates with more than one standard deviation from the expected value for a truly random sequence, and guessing "random" otherwise).

The above estimated strength of the biases in the `iz3z` distribution allow us to estimate the complexity of such an attack. For a truly random sequence of bytes, the standard deviation of the chi-square statistic is given by $\sigma = \sqrt{2(N^3 - 1)}$. Combining this with Equation 4 and the above estimated value of ϵ, yields that approximately 2^{81} drips are required to mount the above described distinguishing attack against Spritz based on the biases in the `iz3z` distribution.

8.3.3 Individual biases

While the chi-square statistic captures how biased the entire `iz3z` distribution is, it does not provide the strength or position of the individual biases in the distribution (that is, the biases of the individual `iz3z` tuples). In Figure 10 we show an overview of the strongest individual biases detected in the `iz3z` distribution of Spritz. The measured relative strength of the biases is based on the data generated to estimate the bias of the chi-squared statistic of the entire `iz3z` distribution. Note that since the variable i is assumed to be known to an attacker, the measured relative bias is with respect to a fixed value of i i.e. the measured bias was computed based on the distribution of `iz3z` tuples with the same value of i.

8.3.4 Comparison with RC4

Given the similarities between Spritz and the original RC4, comparing the strength of the known biases of the two ciphers seems interesting. In the following we will consider RC4 biases appearing when the state of the original RC4 is initialized with a truly random state. (Note that the key schedule of the original RC4 do not achieve this, and as a result, if the original key schedule is used, strong biases will appear in the initial part of the keystream.)

iz3z tuple	Estimated rel. bias ($\log_2(\epsilon)$)		
	$N = 16$	$N = 24$	$N = 32$
$(0, 0, 0)$	-11.3098	-13.2348	-14.6683
$(0, 1, 0)$	-11.3962	-12.6918	-14.0146
$(0, N-1, N-1)$	-11.1694	-13.3496	-14.9024
$(1, 1, 0)$	-11.2904	-13.3961	-14.7268
$(1, 2, 1)$	-10.7283	-12.5944	-13.8832
$(1, 2, N-1)$	-10.9844	-13.2360	-14.5829
$(N-1, 0, 1)$	-10.3501	-12.4090	-13.8885

Figure 10: Overview of the tuples in the iz3z distribution of Spritz with the strongest biases. The estimated relative biases for $N = 16, 24, 32$ are based on distributions generated by 2^{46}, 2^{48}, and 2^{51} keystream bytes, respectively.

N	#keystream bytes	χ^2 of iz1z (#σ from uniform)	$\log_2(\epsilon)$
16	2^{38}	350789	-6.5400
24	2^{38}	36888	-7.7259
32	2^{38}	7422	-8.5712
64	2^{38}	168	-10.5505
128	2^{43}	115	-12.5763
256	2^{45}	12	-14.4473

Figure 11: Biases in the iz1z distribution of the original RC4.

Like for Spritz, we obtained estimates of the chi-square statistic for RC4 for distributions iz1z, iz2z, and iz3z, and for $N = 16, 24, 32$. While biases in all the distributions were detected, the strongest biases appear in iz1z (see Appendix B).

For comparison purposes, we measured the biases in the chi-square statistic of the iz1z distribution of RC4 for $N = 16, 24, 32, 64, 128, 256$. The results are shown in Figure 11. Fitting the relative biases shown in the figure to the expression $\epsilon = c \cdot N^{-d}$ using the method of least squares, we obtained the following values for the parameters (c, d) for RC4:

$$c = 2.9691 \qquad d = 2.0275$$

Based on this, we can directly compare the strength of the biases in the iz1z distribution of RC4

N	$\log_2$(#keystream bytes)	
	RC4 (iz1z)	Spritz (iz3z)
16	19.5799	31.7734
24	22.8294	39.0387
32	25.1350	44.1934
64	30.6900	56.6135
128	36.2450	69.0335
256	41.8000	81.4535

Figure 12: Comparison of the expected number of samples required for RC4 and Spritz, respectively, to reach a distribution with a chi-square statistic deviating with one standard deviation from the expected chi-square statistic of a uniform distribution.

to the strength of the biases in the iz3z distribution of Spritz (note that no biases were detected in the iz1z distribution of Spritz for $N = 16$ and 2^{38} calls in total to DRIP). The table in Figure 12 shows the expected number of keystream bytes required before the value of the chi-square statistic of the RC4 iz1z distribution and the Spritz iz3z distribution deviate with one standard deviation from the expected chi-square statistic of a uniform distribution. As described above, a distinguisher with constant success probability approximately 0.69 can be constructed assuming this number of keystream bytes can be obtained.

9 Performance analysis

To evaluate the performance of Spritz, we measured the keystream generation rate when generating long sequences of keystream bytes, the encryption rate when absorbing a key followed by encryption of a short message, and lastly, the rate at which larger amounts of data can be absorbed, which is relevant if Spritz were to be used to authenticate or hash data. All measurements were done on a Macbook Air (1.8GHz Core i5). We emphasize that for these measurements, a simple and unoptimized implementation of Spritz was used.

Squeeze We measured the following keystream generation rates for (unoptimized) Squeeze:

- 94,689,168 bytes / second
- 11 nanoseconds / byte
- 24 cycles / byte

For comparison, using the eSTREAM [8] (optimized) implementations of Salsa20, RC4, and AES-CTR, we measured keystream generation rates of 296MB/s, 293MB/s, and 152MB/s, respectively.

Encryption of short packets To measure the impact of the key setup time on encryption, we measured the time taken to setup a 16 byte key followed by encryption of a 512 bytes message. This yielded the following encryption rate:

- 32,165,588 bytes / second
- 31 nanoseconds / byte
- 71 cycles / byte

Performing the same measurements for Salsa20, RC4, and AES-CTR, yielded the encryption rates 268MB/s, 142MB/s, 146MB/s, respectively.

Absorb Lastly, we measured the processing rate of Absorb when large amounts of data is given as input:

- 5,615,850 bytes / second
- 179 nanoseconds / byte
- 408 cycles / byte

While our timings are for quite unoptimized code, it does seem that the speed of Spritz as a hash function is very much slower than Keccak (around 11 c/b) or SHA-256 (around 14 c/b). The virtues of Spritz as a hash function are more its simplicity of implementation and conservative design than its speed.

10 Other RC4 variants

For a recent comprehensive treatment of RC4 and its variants, see Paul and Maitra [22]. We note a few other RC4 variant proposals here for the record.

B. Zoltak [33] presented an RC4 variant called VMPC, which uses composition of permutations to define new keystream output and shuffle operations. The variant RC4A was designed by Paul and Preneel [23], it uses *two* permutations S two generate the keystream. Maximov [16] studies the security of these ciphers and provides some distinguishing attacks.

11 Open Problems

There a number of natural cryptanalytic problems for Spritz:

- Devise a distinguishing attack for Spritz.
- Determine the complexity of inverting Shuffle, given a target for the last $N/2 - D$ bytes of S.
- Determine the complexity of finding an "inner collision" for Spritz: two messages that, when absorbed, yield the same inner state.

12 Conclusion

We have presented an updated design (Spritz) for the RC4 stream cipher, adapting it to have a sponge-like interface. The code remains relatively simple, given this expanded interface.

We have also performed extensive simulations to look for detectable statistical biases, and selected a candidate with no biases detected in our experiments.

Further cryptanalytic review is recommended before this cipher is used in critical applications.

Acknowledgments

The first author gratefully acknowledges support from his Vannevar Bush Professorship. We also thank MIT's Computer and Artificial Laboratory for supporting our computation needs.

Note on Intellectual Property (IP)

We do not know of any patent or other restrictions on the use of the ideas proposed here. We have not filed for any such patents (and will not). The authors place into the public domain any and all rights they have on the use of the ideas, methods or systems proposed here. As far as we are concerned, others may freely make, use, sell, offer for sale, import, embed, modify, or improve the ideas, methods, or systems described in this note. There is no need to contact us or ask our permission in order to do so. We ask only that this paper be cited as appropriate.

References

[1] Mete Akgün, Pınar Kavak, and Hüseyin Demirci. New results on the key scheduling algorithm of RC4. In D.R. Chowdhury, V. Vijmen, and A. Das, editors, *Proc. Indocrypt 2008*, volume 5365 of *Lecture Notes in Computer Science*, pages 40–52. Springer, 2008.

[2] Nadhem J. AlFardan, Daniel J. Bernstein, Kenneth G. Paterson, Bertram Poettering, and Jacob C. N. Schuldt. On the security of RC4 in TLS. In *Proc. 22nd USENIX Security Symposium*, Aug. 2013. `https://www.usenix.org/conference/usenixsecurity13/security-rc4-tls`.

[3] Riddhipratim Basu, Subhamoy Maitra, Goutam Paul, and Tanmoy Talukdar. On some sequences of the secret pseudo-random index j in RC4 key scheduling. In *Proc. AAECC 2009*, volume 5527 of *Lecture Notes in Computer Science*, pages 137–148. Springer, 2009.

[4] M. Bellare, P. Rogaway, and D. Wagner. EAX: A conventional authenticated-encryption mode (rev. 2003/9/9). Cryptology ePrint Archive, Report 2003/069, April 13, 2003. `eprint.iacr.org/2003/069`.

[5] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic sponge functions. `http://sponge.noekeon.org/`, January 14, 2011. Version 0.1.

[6] Eli Biham and Yaniv Carmeli. Efficient reconstruction of RC4 keys from internal states. In K. Nyberg, editor, *Proc. Fast Software Encryption*, volume 5086 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2008.

[7] Jiageng Chen and Atsuko Miyaji. How to find short RC4 colliding key pairs. In *Proc. of the 14th international conference on Information security*, ISC'11, pages 32–46, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] eSTREAM: the ECRYPT Stream Cipher Project. `http://www.ecrypt.eu.org/stream/`.

[9] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In Serge Vaudenay and Amr M. Youssef, editors, *Selected Areas in Cryptography*, volume 2259 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[10] J. D. Golić and G. Morgari. Iterative probabilistic reconstruction of RC4 internal state. IACR eprint Archive, 2008. Report 2008/348. Available at `http://eprint.iacr.org/2008/348`.

[11] Shahram Khazaei and Willi Meier. On reconstruction of RC4 keys from internal states. In J. Calmet, W. Geiselmann, and J. Muller-Quade, editors, *Proc. Beth Festschrift*, volume 5393 of *Lecture Notes in Computer Science*, pages 179–189. Springer, 2008.

[12] Andreas Klein. Attacks on the RC4 stream cipher. *Designs, Codes and Cryptography*, 48(3):269–286, 2008.

[13] Subhamoy Maitra, Goutam Paul, Santanu Sarkar, Michael Lehmann, and Willi Meier. New results on generalization of roos-type biases and related keystreams of RC4. In A. Youssef, A. Nitaj, and A. Hassanien, editors, *Proc. AFRICACRYPT 2013*, volume 7918 of *Lecture Notes in Computer Science*, pages 222–239. Springer, 2013.

[14] Itsik Mantin and Adi Shamir. A practical attack on broadcast RC4. In Mitsuru Matsui, editor,

FSE, volume 2355 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2001.

[15] Mitsuru Matsui. Key collisions of the RC4 stream cipher. In O. Dunkelman, editor, *Proc. Fast Software Encryption*, volume 5665 of *Lecture Notes in Computer Science*, pages 38–50. Springer, 2009.

[16] Alexander Maximov. Two linear distinguishing attacks on VMPC and RC4A and weakness of RC4 family of stream ciphers (corrected). In H. Gilbert and H. Handschuh, editors, *Proc. Fast Software Encryption 2005*, volume 3557 of *Lecture Notes in Computer Science*, pages 342–358. Springer, 2005.

[17] Alexander Maximov and Dmitry Khovratovich. New state recovery attack on RC4. In D. Wagner, editor, *Proc. CRYPTO 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 297–315. Springer, 2008.

[18] Alfred Menezes, Paul van Oorschot, and Scott Vanstone. *Handbook of Applied Cryptography.* CRC, 1997.

[19] Ilya Mironov. (Not so) random shuffles of RC4. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 304–319. Springer, 2002.

[20] NIST. Recommendation for random number generation using deterministic random bit generators. Technical Report SP 800-90 A Rev. 1 (Draft), NIST, 2013. `http://csrc.nist.gov/publications/drafts/800-90/draft_sp800_90a_rev1.pdf`.

[21] Goutam Paul and Subhamoy Maitra. Permutation after RC4 key scheduling reveals the secret key. In C. Adams, A. Miri, and M. Wiener, editors, *Proc. SAC 2007*, volume 4876 of *Lecture Notes in Computer Science*, pages 360–377. Springer, 2007.

[22] Goutam Paul and Subhamoy Maitra. *RC4 Stream Cipher and Its Variants.* CRC Press, 2012.

[23] S. Paul and B. Preneel. A new weakness in the RC4 keystream generator and an approach to improve the security of the cipher. In *Proc. FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 2004.

[24] Philip Rogaway. Constructing cryptographic definitions. Available at `http://www.cs.ucdavis.edu/~rogaway/papers/iscisc.pdf`, 2011. Essay to accompany ISCISC 2011 invited talk.

[25] Andrew Roos. A class of weak keys in the RC4 stream cipher. Posted to sci.crypt, 1995.

[26] RSA Laboratories . RSA Security response to weaknesses in key scheduling algorithm of RC4. `http://www.emc.com/emc-plus/rsa-labs/historical/rsa-security-response-weaknesses-algorithm-rc4.htm` , 2001.

[27] Sourav Sengupta, Subhamoy Maitra, Goutam Paul, and Santanu Sarkar. RC4: (Non-) random words from (non-)random permutations. *IACR Cryptology ePrint Archive*, 2011:448, 2011.

[28] Pouyan Sepehrdad. *Statistical and Algebraic Cryptanalysis of Lightweight and Ultra-Lightweight Symmetric Primitives.* PhD thesis, École Polytechnique Fédérale de Lausanne (Suisse), 2012. Thèse No. 5415. Available at `http://lasecwww.epfl.ch/~sepehrdad/Pouyan_Sepehrdad_PhD_Thesis.pdf`.

[29] Pouyan Sepehrdad, Serge Vaudenay, and Martin Vuagnoux. Statistical attack on RC4 – distinguishing WPA. In Kenneth G. Paterson, editor, *EUROCRYPT 2011*, volume 6632 of *Lecture Notes in Computer Science*, pages 343–363. Springer, 2011.

[30] Adam Stubblefield, John Ioannidis, and Aviel D. Rubin. A key recovery attack on the 802.11b wired equivalent privacy protocol (wep). *ACM Trans. Inf. Syst. Secur.*, 7(2):319–332, May 2004.

[31] Adam Stubblefield, John Ioannidis, and Avield D. Rubin. Using the Fluhrer, Mantin, and Shamir attack to break WEP. Technical report, AT&T, 2001. AT&T Labs Technical Report TD-4ZCPZZ.

[32] Erik Tews, Ralf-Philipp Weinmann, and Andrei Pyshkin. Breaking 104 bit WEP in less than 60 seconds. In Sehun Kim, Moti Yung, and Hyung-Woo Lee, editors, *WISA 2007*, volume 4867 of *Lecture Notes in Computer Science*, pages 188–202. Springer, 2007.

[33] B. Zoltak. One-way function and stream cipher. In *Proc. FSE 2004*, volume 3017 of *Lecture Notes in Computer Science*, pages 210–225. Springer, 2004.

A UPDATE and OUTPUT candidates

The 147 UPDATE candidate pairs (E_j, E_k) satisfying the syntactic criteria listed in Section 8.1, are listed in Figure 13 and 14. The 33 UPDATE candidates satisfying the syntactic criteria described in Section 8.2, are listed in Figure 15

B RC4 `iz1z`, `iz2z`, and `iz3z` distributions

The table below shows the biases of the `iz1z`, `iz2z`, and `iz3z` distributions of the RC4 keystream generated based on 2^{19} random starting states, and 2^{19} generated keystream bytes for each starting state. Note that while all the distributions are biased, the strongest biases appear in `iz1z`. (The biases are measured in standard deviations from the expected chi-square statistic of a uniform distribution.)

N	#keystream bytes	χ^2 bias (#σ from uniform)		
		`iz1z`	`iz2z`	`iz3z`
16	2^{38}	350789	62462	14091
24	2^{38}	36888	3626	608
32	2^{38}	7422	552	132

C Chi-square statistic of a biased distribution

In this section, we will justify the estimate of the chi-squared statistic of a biased distribution which is used in Section 8.3.2.

Recall that the chi-squared statistic for a three-variable distribution (e.g. `iz3z`) is given by:

$$\chi^2 = \sum_{u=0}^{N^3-1} (O_u - T/N^3)^2 \cdot N^3/T$$

where u enumerates the possible tuples in the distribution, O_u denotes the observed number of occurrences of tuple u, and T denotes the total number of samples. Hence, we have

$$\begin{aligned} \mathrm{E}[T\cdot\chi^2/N^3] \\ &= \mathrm{E}[\sum_{u=0}^{N^3-1} (O_u - T/N^3)^2] \\ &= \mathrm{E}[\sum_{u=0}^{N^3-1} (O_u - \mathrm{E}[O_u])^2 + (\mathrm{E}[O_u] - T/N^3)^2] \\ &= \mathrm{E}[\sum_{u=0}^{N^3-1} \mathrm{Var}[O_u] + (\mathrm{E}[O_u] - T/N^3)^2] \qquad (5) \end{aligned}$$

We now assume that the expected value of the number of occurrences of the tuples is biased with the relative bias ϵ, that is, $\mathrm{E}[O_u] = T(N^{-3} \pm \epsilon)$. We furthermore assume that half of the tuples will have a negative bias (i.e. $T(N^{-3} - \epsilon)$), and that the other half will have a positive bias (i.e. $T(N^{-3} + \epsilon)$).

Observe that O_u is a binomial variable with mean $T \cdot p$ and variance $T \cdot p(1-p)$, where $p = N^{-3} \pm \epsilon$. Hence, we have $\mathrm{E}[O_u] = T \cdot N^{-3} \pm T \cdot \epsilon$ and that

$$(\mathrm{E}[O_u] - T/N^3)^2 = (T \cdot \epsilon)^2 \qquad (6)$$

Furthermore,

$$\begin{aligned} \mathrm{Var}[O_u] &= T(N^{-3} \pm \epsilon)(1 - N^{-3} \pm \epsilon) \\ &= T(1/N^3 \pm 2 \cdot \epsilon \cdot 1/N^3 \pm \epsilon - \epsilon^2 - 1/N^6) \end{aligned}$$

Applying the assumption that half of the tuples have a negative bias and the other half a positive bias, we

id	E_j	E_k
1	$j+k+S[i]$	$S[j]+k$
2	$j+k+S[i]$	$S[k]+S[j]$
3	$j+k+S[i]$	$S[S[j]+k]$
4	$S[j+k+S[i]]$	$S[j]+k$
5	$S[S[i]+k]+j$	$S[j]+k$
6	$S[j+k]+S[i]$	$S[j]+k$
7	$S[k]+j+S[i]$	$S[j]+k$
8	$S[j]+k+S[i]$	$S[j]+k$
9	$S[S[i]+j]+k$	$S[j]+k$
10	$j+k+S[i]$	$S[j]+k+i$
11	$j+k+S[i]$	$S[S[k]+S[j]]$
12	$S[j+k+S[i]]$	$S[k]+S[j]$
13	$S[j+k+S[i]]$	$S[S[j]+k]$
14	$S[S[i]+k]+j$	$S[k]+S[j]$
15	$S[S[i]+k]+j$	$S[S[j]+k]$
16	$S[j+k]+S[i]$	$S[k]+S[j]$
17	$S[j+k]+S[i]$	$S[S[j]+k]$
18	$S[k]+j+S[i]$	$S[k]+S[j]$
19	$S[k]+j+S[i]$	$S[S[j]+k]$
20	$S[j]+k+S[i]$	$S[k]+S[j]$
21	$S[j]+k+S[i]$	$S[S[j]+k]$
22	$S[S[i]+j]+k$	$S[k]+S[j]$
23	$S[S[i]+j]+k$	$S[S[j]+k]$
24	$S[S[k]+j]+S[i]$	$S[j]+k$
25	$S[S[j]+k+S[i]]$	$S[j]+k$
26	$S[S[j]+S[i]]+k$	$S[j]+k$
27	$S[S[j+k]+S[i]]$	$S[j]+k$
28	$S[S[j]+k]+S[i]$	$S[j]+k$
29	$S[S[S[i]+k]+j]$	$S[j]+k$
30	$S[S[k]+j+S[i]]$	$S[j]+k$
31	$S[S[k]+S[i]]+j$	$S[j]+k$
32	$S[j]+S[k]+S[i]$	$S[j]+k$
33	$S[S[S[i]+j]+k]$	$S[j]+k$
34	$j+k+S[i]$	$S[S[j]+i]+k$
35	$j+k+S[i]$	$S[S[j]+k]+i$
36	$j+k+S[i]$	$S[S[j]+k+i]$
37	$j+k+S[i]$	$S[k]+S[j]+i$
38	$j+k+S[i]$	$S[j]+k+S[i]$
39	$S[j+k+S[i]]$	$S[j]+k+i$
40	$S[j+k+S[i]]$	$S[S[k]+S[j]]$
41	$S[S[i]+k]+j$	$S[j]+k+i$
42	$S[S[i]+k]+j$	$S[S[k]+S[j]]$
43	$S[j+k]+S[i]$	$S[j]+k+i$
44	$S[j+k]+S[i]$	$S[S[k]+S[j]]$
45	$S[k]+j+S[i]$	$S[j]+k+i$
46	$S[k]+j+S[i]$	$S[S[k]+S[j]]$
47	$S[j]+k+S[i]$	$S[j]+k+i$
48	$S[j]+k+S[i]$	$S[S[k]+S[j]]$
49	$S[S[i]+j]+k$	$S[j]+k+i$
50	$S[S[i]+j]+k$	$S[S[k]+S[j]]$
51	$S[S[k]+j]+S[i]$	$S[k]+S[j]$
52	$S[S[k]+j]+S[i]$	$S[S[j]+k]$
53	$S[S[j]+k+S[i]]$	$S[k]+S[j]$
54	$S[S[j]+k+S[i]]$	$S[S[j]+k]$
55	$S[S[j]+S[i]]+k$	$S[k]+S[j]$
56	$S[S[j]+S[i]]+k$	$S[S[j]+k]$
57	$S[S[j+k]+S[i]]$	$S[k]+S[j]$
58	$S[S[j+k]+S[i]]$	$S[S[j]+k]$
59	$S[S[j]+k]+S[i]$	$S[k]+S[j]$
60	$S[S[j]+k]+S[i]$	$S[S[j]+k]$
61	$S[S[S[i]+k]+j]$	$S[k]+S[j]$
62	$S[S[S[i]+k]+j]$	$S[S[j]+k]$
63	$S[S[k]+j+S[i]]$	$S[k]+S[j]$
64	$S[S[k]+j+S[i]]$	$S[S[j]+k]$
65	$S[S[k]+S[i]]+j$	$S[k]+S[j]$
66	$S[S[k]+S[i]]+j$	$S[S[j]+k]$
67	$S[j]+S[k]+S[i]$	$S[k]+S[j]$
68	$S[j]+S[k]+S[i]$	$S[S[j]+k]$
69	$S[S[S[i]+j]+k]$	$S[k]+S[j]$
70	$S[S[S[i]+j]+k]$	$S[S[j]+k]$
71	$S[S[S[j]+S[i]]+k]$	$S[j]+k$
72	$S[S[j]+S[k]]+S[i]$	$S[j]+k$
73	$S[S[j]+S[k]+S[i]]$	$S[j]+k$
74	$S[S[S[k]+j]+S[i]]$	$S[j]+k$
75	$S[S[S[k]+S[i]]+j]$	$S[j]+k$
76	$S[S[S[j]+k]+S[i]]$	$S[j]+k$
77	$j+k+S[i]$	$S[S[S[j]+i]+k]$
78	$j+k+S[i]$	$S[S[j]+S[i]]+k$
79	$j+k+S[i]$	$S[S[j]+k]+S[i]$
80	$j+k+S[i]$	$S[k]+S[j]+S[i]$
81	$j+k+S[i]$	$S[S[j]+k+S[i]]$
82	$j+k+S[i]$	$S[S[k]+S[j]]+i$
83	$j+k+S[i]$	$S[S[k]+S[j]+i]$
84	$j+k+S[i]$	$S[S[S[j]+k]+i]$
85	$S[j+k+S[i]]$	$S[S[j]+i]+k$
86	$S[j+k+S[i]]$	$S[S[j]+k]+i$
87	$S[j+k+S[i]]$	$S[S[j]+k+i]$
88	$S[j+k+S[i]]$	$S[k]+S[j]+i$
89	$S[j+k+S[i]]$	$S[j]+k+S[i]$
90	$S[S[i]+k]+j$	$S[S[j]+i]+k$

Figure 13: UPDATE candidate expressions (E_j, E_k), 1 to 90.

id	E_j	E_k
91	$S[S[i]+k]+j$	$S[S[j]+k]+i$
92	$S[S[i]+k]+j$	$S[S[j]+k+i]$
93	$S[S[i]+k]+j$	$S[k]+S[j]+i$
94	$S[S[i]+k]+j$	$S[j]+k+S[i]$
95	$S[j+k]+S[i]$	$S[S[j]+i]+k$
96	$S[j+k]+S[i]$	$S[S[j]+k]+i$
97	$S[j+k]+S[i]$	$S[S[j]+k+i]$
98	$S[j+k]+S[i]$	$S[k]+S[j]+i$
99	$S[j+k]+S[i]$	$S[j]+k+S[i]$
100	$S[k]+j+S[i]$	$S[S[j]+i]+k$
101	$S[k]+j+S[i]$	$S[S[j]+k]+i$
102	$S[k]+j+S[i]$	$S[S[j]+k+i]$
103	$S[k]+j+S[i]$	$S[k]+S[j]+i$
104	$S[k]+j+S[i]$	$S[j]+k+S[i]$
105	$S[j]+k+S[i]$	$S[S[j]+i]+k$
106	$S[j]+k+S[i]$	$S[S[j]+k]+i$
107	$S[j]+k+S[i]$	$S[S[j]+k+i]$
108	$S[j]+k+S[i]$	$S[k]+S[j]+i$
109	$S[j]+k+S[i]$	$S[j]+k+S[i]$
110	$S[S[i]+j]+k$	$S[S[j]+i]+k$
111	$S[S[i]+j]+k$	$S[S[j]+k]+i$
112	$S[S[i]+j]+k$	$S[S[j]+k+i]$
113	$S[S[i]+j]+k$	$S[k]+S[j]+i$
114	$S[S[i]+j]+k$	$S[j]+k+S[i]$
115	$S[S[k]+j]+S[i]$	$S[j]+k+i$
116	$S[S[k]+j]+S[i]$	$S[S[k]+S[j]]$
117	$S[S[j]+k+S[i]]$	$S[j]+k+i$
118	$S[S[j]+k+S[i]]$	$S[S[k]+S[j]]$
119	$S[S[j]+S[i]]+k$	$S[j]+k+i$
120	$S[S[j]+S[i]]+k$	$S[S[k]+S[j]]$
121	$S[S[j+k]+S[i]]$	$S[j]+k+i$
122	$S[S[j+k]+S[i]]$	$S[S[k]+S[j]]$
123	$S[S[j]+k]+S[i]$	$S[j]+k+i$
124	$S[S[j]+k]+S[i]$	$S[S[k]+S[j]]$
125	$S[S[S[i]+k]+j]$	$S[j]+k+i$
126	$S[S[S[i]+k]+j]$	$S[S[k]+S[j]]$
127	$S[S[k]+j+S[i]]$	$S[j]+k+i$
128	$S[S[k]+j+S[i]]$	$S[S[k]+S[j]]$
129	$S[S[k]+S[i]]+j$	$S[j]+k+i$
130	$S[S[k]+S[i]]+j$	$S[S[k]+S[j]]$
131	$S[j]+S[k]+S[i]$	$S[j]+k+i$
132	$S[j]+S[k]+S[i]$	$S[S[k]+S[j]]$
133	$S[S[S[i]+j]+k]$	$S[j]+k+i$
134	$S[S[S[i]+j]+k]$	$S[S[k]+S[j]]$
135	$S[S[S[j]+S[i]]+k]$	$S[k]+S[j]$
136	$S[S[S[j]+S[i]]+k]$	$S[S[j]+k]$
137	$S[S[j]+S[k]]+S[i]$	$S[k]+S[j]$
138	$S[S[j]+S[k]]+S[i]$	$S[S[j]+k]$
139	$S[S[j]+S[k]+S[i]]$	$S[k]+S[j]$
140	$S[S[j]+S[k]+S[i]]$	$S[S[j]+k]$
141	$S[S[S[k]+j]+S[i]]$	$S[k]+S[j]$
142	$S[S[S[k]+j]+S[i]]$	$S[S[j]+k]$
143	$S[S[S[k]+S[i]]+j]$	$S[k]+S[j]$
144	$S[S[S[k]+S[i]]+j]$	$S[S[j]+k]$
145	$S[S[S[j]+k]+S[i]]$	$S[k]+S[j]$
146	$S[S[S[j]+k]+S[i]]$	$S[S[j]+k]$
147	$S[S[S[j]+S[k]]+S[i]]$	$S[j]+k$

Figure 14: UPDATE candidate expressions (E_j, E_k), 91 to 147.

id	E_z
1	$S[k+j]$
2	$S[z+k+j]$
3	$S[k+j+i]$
4	$S[S[z+j]+k]$
5	$S[S[k+j]+z]$
6	$S[S[z+k]+j]$
7	$S[S[j+i]+k]$
8	$S[S[k+j]+i]$
9	$S[S[k+i]+j]$
10	$S[z+k+j+i]$
11	$S[S[z+j]+k+i]$
12	$S[S[j+i]+z+k]$
13	$S[S[k+j]+z+i]$
14	$S[S[z+k]+j+i]$
15	$S[S[k+i]+z+j]$
16	$S[S[z+j+i]+k]$
17	$S[S[z+k+j]+i]$
18	$S[S[k+j+i]+z]$
19	$S[S[z+k+i]+j]$
20	$S[S[z+i]+k+j]$
21	$S[S[S[k+i]+j]+z]$
22	$S[S[z+k]+S[j+i]]$
23	$S[S[S[k+i]+z]+j]$
24	$S[S[S[z+j]+i]+k]$
25	$S[S[S[z+j]+k]+i]$
26	$S[S[S[j+i]+z]+k]$
27	$S[S[S[j+i]+k]+z]$
28	$S[S[S[z+i]+j]+k]$
29	$S[S[S[k+j]+z]+i]$
30	$S[S[S[k+j]+i]+z]$
31	$S[S[S[z+i]+k]+j]$
32	$S[S[S[z+k]+j]+i]$
33	$S[S[S[z+k]+i]+j]$

Figure 15: OUTPUT candidate expressions E_z

obtain

$$\sum_{u=0}^{N^3-1} \mathrm{Var}[O_u] = T(N^{-3} - \epsilon^2 - N^{-6}) \qquad (7)$$

Inserting (6) and (7) into (5) and taking into account that the sum in (5) is over N^3 elements, yields that

$$\mathrm{E}[T \cdot \chi^2/N^3] = N^3(T \cdot N^{-3} - T\epsilon^2 - T/N^6 + T^2\epsilon^2),$$

and from this we obtain

$$\mathrm{E}[\chi^2] = N^3 + T \cdot N^6 \cdot \epsilon^2$$

which is the estimate used in Section 8.3.2.

D Alternate Squeeze

To follow the sponge paradigm more exactly, we would want to extract output from the outer portion of the current state, without touching the inner state. We present one approach to doing this. It has the advantage of having provably unbiased outputs (for randomly-chosen states), but it is quite slow. It would be of interest perhaps only for hashing or key derivation, where the outputs are short.

We try a natural approach of inverting AbsorbNibble, extracting one nibble at a time and modifying the state. The input r is the number of output bytes desired.

```
AltSqueeze(r)
1  if a > 0
2        Shuffle()
3  P = Array.New(r)
4  b = 0
5  for v = 0 to r − 1
6        if w = N/4
7              Shuffle()
8              b = 0
9        y = index of min value in S[N/2..N/2 + D − 1]
10       Swap(S[2b], S[y])
11       x = index of min value in S[N/2..N/2 + D − 1]
12       Swap(S[2b + 1], S[x])
13       P[v] = D · (x − N/2) + (y − N/2)
14       b = b + 1
15 return P
```

Determining the nibble values x and y is somewhat expensive (D steps each). As written, AltSqueeze assumes that $N = D^2$ and that N is even (default $N = 256$ is OK). Furthermore, AltSqueeze doesn't support piecemeal output (it should only be called once), and shouldn't be used if Squeeze is also being used.

Theorem 1 *If* Shuffle *produces a random permutation of S at each call, then each output string of r bytes from* AltSqueeze *is equally likely.*

Proof The first y is equally likely to be any nibble. and after $S[y]$ is swapped into position $S[2w]$, all permutations of $S[2b+1...N/2+D-1]$ are equally likely, so the rest proceeds similarly by induction. ∎

One could use heaps to compute x and y, but this would probably be slower, since D is small (e.g. 16).

It should be noted that AltSqueeze isn't exactly inverting AbsorbNibble, since it presumes that the S yielded at the end is lexicographically minimal among those reachable by its procedure. Indeed, you can't really tell what the input nibble was, unless you also know what the start state was.

E Test vectors

This appendix provides several test vectors for Spritz.

Basic Spritz output:

Input	Output
ABC	77 9a 8e 01 f9 e9 cb c0 ...
spam	f0 60 9a 1d f1 43 ce bf ...
arcfour	1a fa 8b 5e e3 37 db c7 ...

Spritz hash function outputs (for 32-byte hashes):

Input	Output
ABC	02 8f a2 b4 8b 93 4a 18 ...
spam	ac bb a0 81 3f 30 0d 3a ...
arcfour	ff 8c f2 68 09 4c 87 b9 ...