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

Ronald L. Rivest\textsuperscript{1}  Jacob C. N. Schuldt\textsuperscript{2}

\textsuperscript{1}Vannevar Bush Professor of EECS
MIT CSAIL
Cambridge, MA 02139
rivest@mit.edu

\textsuperscript{2}Research Institute for Secure Systems
AIST, Japan
jacob.schuldt@aist.go.jp

CRYPTO DAY
October 24, 2014
Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion
Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion
Widely used (50% of all TLS connections).
Simple, fast.
Works for any set of $N$ “bytes”: $\mathbb{Z}_N = \{0, 1, \ldots, N - 1\}$.
(All math is mod $N$.) Default is $N = 256$.
State consists of:
- two mod-$N$ “pointers” $i$ and $j$
- a permutation $S$ of $\mathbb{Z}_N$
Key setup algorithm (KSA) initializes $S$ from secret key $K$
Pseudo-random generator (PRG) updates state and outputs pseudo-random byte; typically used as pseudo-one-time pad.
RC4-PRG

RC4-PRG()

1. $i = i + 1$  // update state
2. $j = j + S[i]$
3. $\text{SWAP}(S[i], S[j])$
4. $z = S[S[i] + S[j]]$  // generate output
5. return $z$

```
S  [•••]  S[i]  S[j]  [•••]  z  [•••]
```

0 1  $i$  $j$  $S[i] + S[j]$  $N - 1$
RC4-KSA

- input key $K$ is a sequence of $L$ bytes (mod $N$ values)

**RC4-KSA($K$)**

1. $S[0..N-1] = [0..N-1]$
2. $j = 0$
3. for $i = 0$ to $N - 1$
   4. $j = j + S[i] + K[i \mod L]$
   5. SWAP($S[i], S[j]$)
4. $i = j = 0$

- Common criticism is that loop of lines 3–5 is executed too few times; some recommend executing it $2N–4N$ times or more, or ignoring first $2N–4N$ outputs.
Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion
RC4 attacks

RC4 has numerous vulnerabilities and “soft spots” [see paper for citations]:

- Key-dependent biases of initial output
- Key collisions (producing same internal state)
- Key recovery possible from known internal state
- Related-key attacks (WEP)
- State recovery from known output (feasible?)
- Output biases; distinguishers
Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion
We started design after CRYPTO 2013. (Really after AlFarden, ..., and Schuldt. USENIX 2013)

**Design principles:**

- Drop-in replacement for RC4
- Retain “RC4 style” (e.g. state is a few registers plus a permutation $S$ of $\{0, 1, \ldots, N - 1\}$)
- Minimize statistical vulnerabilities
- Redo key-setup entirely
- Expand API to have “spongy” interface: can interleave “absorbing” input and “squeezing” out pseudo-random bytes.
Automatically examined many thousands of candidates

Expressions generated and represented by postfix expressions: \( ikjS++ \) means \( i + k + S[j] \)

Filtered by:

- **syntactic criterion** (e.g. invertible expressions containing \( S \) but no \( SS \)),
- **cryptographic criteria** (e.g. can not swap two values in \( S \) and leave evolution of \( j \) and \( k \) unaffected), and
- **statistical criteria** (very heavy testing of candidates for smaller values of \( N \). Approximately 12 “hyperthreaded core-years” of CPU time used. About \( 2^{53} \) Spritz outputs tested.)
Winner is #4933

\[
\begin{align*}
\text{i} \cdot \text{w} +, & \quad \text{k} \cdot \text{j} \cdot \text{i} \cdot \text{S} +, \\
& \quad \text{i} \cdot \text{k} \cdot \text{j} \cdot \text{S} +, \\
& \quad \text{j} \cdot \text{i} \cdot \text{k} \cdot \text{z} +, \\
& \quad \text{j} \cdot \text{i} \cdot \text{k} \cdot \text{z} +, \\
& \quad \text{j} \cdot \text{i} \cdot \text{k} \cdot \text{z} + \\
\end{align*}
\]

**RC4-PRG()**

1. \( i = i + 1 \)
2. \( j = j + S[i] \)
3. \( \text{SWAP}(S[i], S[j]) \)
4. \( z = S[S[i] + S[j]] \)
5. \text{return } z

**SPRITZ-PRG()**

1. \( i = i + w \)
2. \( j = k + S[j + S[i]] \)
3. \( k = i + k + S[j] \)
4. \( \text{SWAP}(S[i], S[j]) \)
5. \( z = S[j + S[i + S[z + k]]] \)
6. \text{return } z

- About 50% longer
- Uses new register \( k \) as well RC4 registers \( i, j \); output register \( z \) also used in feedback. Register \( w \) always relatively prime to \( N \).
Start **SPRITZ** with **INITIALIZE**<strong>STATE**

- State variable $S$ initialized to identity permutation
- “Pointer” variables $i$, $j$, $k$, initialized to 0.
- “Last output” variable $z$ initialized to 0
- “Number of nibbles absorbed” variable $a$ set to 0
- “Step size” variable $w$ initialized to 1

**INITIALIZE**<strong>STATE**($N$)

1. $S[0..N-1] = [0..N-1]$
2. $i = j = k = z = a = 0$
3. $w = 1$
**SQUEEZE to output \( r \)-byte array**

\[
\text{SQUEEZE}(r)
\]

1. \textbf{if} \( a > 0 \) \hspace{1cm} // \text{last operation was ABSORB}
2. \textbf{SHUFFLE}()
3. \( P = \) new array of size \( r \)
4. \textbf{for} \( v = 0 \) \textbf{to} \( r - 1 \)
5. \( P[v] = \text{SPRITZ-PRG}() \)
6. \textbf{return} \( P \)
Encryption

\[ \text{ENCRYPT}(K, M) \]

1. \text{KEYSETUP}(K)
2. \( C = M + \text{SQUEEZE}(M.\text{length}) \)
3. \text{return } C

\text{KEYSETUP}(K)

1. \text{INITIALIZESTATE}()
2. \text{ABSORB}(K)
ABSORB takes an arbitrary sequence $K$ of bytes as input.
Absorbs each byte by absorbing its two four-bit "nibbles".
After each 512 bits of input, or when output is desired, SHUFFLE procedure called to "stir the pot" (WHIP) and to "provide forward security (CRUSH).
Variable $a$ is number of nibbles absorbed since last SHUFFLE
SHUFFLE effects a “random” one-way transformation on the current state.

\[
\text{SHUFFLE()}
\]

1. \texttt{WHIP}(2N)
2. \texttt{CRUSH}()
3. \texttt{WHIP}(2N)
4. \texttt{CRUSH}()
5. \texttt{WHIP}(2N)
6. \( a = 0 \)
Purpose of WHIP$(r)$ is to “stir the pot” vigorously, by generating and ignoring $r$ bytes of output, then increasing $w$ by 2 (so $w$ remains odd and relatively prime to 256.)

```plaintext
WHIP\( (r) \)
1  for \( v = 0 \) to \( r - 1 \)
2   SPRITZ-PRG()     // output ignored
3  \( w = w + 2 \)

(If $N$ is not a power of 2, WHIP increases $w$ to the next value that is relatively prime to $N$.)
```
**Crush for forward security**

The elements of $S$ are considered as $N/2$ pairs; each is sorted into increasing order. The input is at the top; the output at the bottom. Horizontal lines represent two-element sorting operations. **Crush** provides “forward security” for **Shuffle**.
Key-Setup (or general input) with ABSORB

\texttt{ABSORB}(K)

\begin{verbatim}
1  for \( v = 0 \) to \( K.length - 1 \)
2    \texttt{ABSORBBYTE}(K[v])
\end{verbatim}

\texttt{ABSORBBYTE}(b)

\begin{verbatim}
1  \texttt{ABSORBNIBBLE}(\texttt{LOW}(b))
2  \texttt{ABSORBNIBBLE}(\texttt{HIGH}(b))
\end{verbatim}

\texttt{ABSORBNIBBLE}(x)

\begin{verbatim}
1  if \( a = \lfloor N/2 \rfloor \)
2    \texttt{SHUFFLE}()
3    \texttt{SWAP}(S[a], S[\lfloor N/2 \rfloor + x])
4    \( a = a + 1 \)
\end{verbatim}
Nibble sequence 1,2,1,0 has just been absorbed. When the $a$-th nibble $x$ is absorbed, $S[a]$ is exchanged with $S[N/2 + x]$; note that $0 \leq x < D$, where $D = \sqrt{N}$. ABSORB never touches the last $N/2 - D$ elements of $S$, greatly limiting how adversarial input can affect $S$. 
SPRITZ is spongy!

SPRITZ is also a (modified) sponge function, and usable as a hash function:

1. **INITIALIZE STATE**(N)
2. **ABSORB**(“abc”) — ACCEPT INPUT PIECEMEAL.
3. **ABSORB**(“def”)
4. **SQUEEZE**(32) — OUTPUT 32 BYTE HASH.
5. **ABSORB**(“ghi”) — KEEP GOING...
6. **SQUEEZE**(1000)

- Large state space (like KECCAK), but also has built-in protection against inference of key from knowledge of internal state (which KECCAK does not).
- (But very much slower than Keccak...)
**ABSORBSTOP** rather than padding

- **ABSORBSTOP** absorbs an “out-of-alphabet” symbol; makes for easier interfaces than padding rules.
- All **ABSORBSTOP** does is increase $a$ (the number of absorbed nibbles) by one, without actually absorbing a nibble.

```
ABSORBSTOP()
1 if a = \lfloor N/2 \rfloor
2 SHUFFLE()
3 a = a + 1
```
Spritz as a hash function

- Note that we include output length $r$ in the hash input, so $r$-byte hash outputs are not just a prefix of $r'$-byte hash outputs for $r < r'$; these act as distinct hash functions.

```
HASH(M, r)
1  INITIALIZESTATE()
2  ABSORB(M); ABSORBSTOP()
3  ABSORB(r)
4  return SQUEEZE(r)
```
Spritz as a MAC

MAC example with $r$-byte output.

$$\text{MAC}(K, M, r)$$

1. $\text{INITIALIZE\,STATE()}$
2. $\text{ABSORB}(K); \text{ABSORB\,STOP()}$
3. $\text{ABSORB}(M); \text{ABSORB\,STOP()}$
4. $\text{ABSORB}(r)$
5. $\text{return SQUEEZE}(r)$
Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion
Statistical testing

- Primary tool: chi-square testing for uniformity.
- Typical test: chi-square for uniformity of triple \((i, z1, z)\) (aka \(iz1z\)) where \(zs\) is \(z\) delayed \(s\) steps. Table has \(N^3\) entries for counts.
- Tests run include \(jsj, iksk, izsz, ijsz,\) and \(iksz\) for \(s\) up to \(N\).
- Tested \(N = 16\): no biases for \(2^{32}\) outputs; for \(2^{36}\) outputs biases detected (strongest \(iz3z\)).
- Chi-square biases modelled as \(cN^{-d}\); good model for all RC4-like designs; can fit curves to estimate \(c\) and \(d\) as function of \(N\).
- Measured biases for \(N = 16, 24, 32\), extrapolate to \(N = 64, 128, 256\).
Biases measured and extrapolated

The expected number of outputs required for RC4 and Spritz to reach a distribution with a chi-square deviating by one standard deviation from the expected chi-square statistic of a uniform distribution, for the best distinguisher in each case.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$\log_2(#\text{keystream bytes})$</th>
<th>RC4 ($\text{i z1 z}$)</th>
<th>Spritz ($\text{i z3 z}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>19.5799</td>
<td>31.7734</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>22.8294</td>
<td>39.0387</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>25.1350</td>
<td>44.1934</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>30.6900</td>
<td>56.6135</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>36.2450</td>
<td>69.0335</td>
<td></td>
</tr>
<tr>
<td>256</td>
<td>41.8000</td>
<td>81.4535</td>
<td></td>
</tr>
</tbody>
</table>
Graph

log$_2$ of outputs required versus $N$
Much better statistics!

- Spritz statistical biases are much fainter than for RC4.
- For $N = 256$:
  - Can distinguish RC4-256 from random with only $2^{41}$ samples.
  - Our tests suggest that $2^{81}$ samples are required to distinguish SPRITZ-256 from random.
Other security properties

Design of Spritz should also make the following hard:
- inferring state from observed output
- inferring key from known state
- related-key attacks
- finding collision for Spritz as hash function
Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion
Performance

- Squeeze output at 94MB/sec (24 cycles/byte) (RC4 is 293MB/sec).
- Absorb data at 5MB/sec (408 cycles/byte) (Keccak is 11 cycles/byte)

The virtues of Spritz are more its simplicity of implementation, flexibility, and secure conservative design than its speed.
Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion
Conclusion

**SPRITZ** is a spongy stream cipher in the style of RC4; it shows excellent statistical properties and great flexibility for applications.
Our paper on SPRITZ is here:

people.csail.mit.edu/rivest/pubs.html#RS14

More security review needed; comments and analysis appreciated!

Thank you!