SPRITZ—A SPONGY RC4-LIKE STREAM CIPHER AND HASH FUNCTION

Ronald L. Rivest[1] Jacob C. N. Schuldt[2]

[1]Vannevar Bush Professor of EECS
MIT CSAIL
Cambridge, MA 02139
rivest@mit.edu

[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

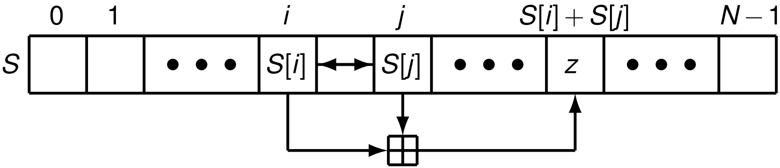
RC4

- Stream cipher RC4 designed by Rivest (1987).
- Widely used (50% of all TLS connections).
- Simple, fast.
- Works for any set of N “bytes”: $Z_N = \{0, 1, \ldots, N - 1\}$. (All math is mod N.) Default is $N = 256$.
- State consists of:
  - two mod-N “pointers” i and j
  - a permutation S of 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()

$i = i + 1$ // update state
$j = j + S[i]$
$\textsc{Swap}(S[i], S[j])$
$z = S[S[i] + S[j]]$ // generate output
return z



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 $\quad j = j + S[i] + K[i \bmod L]$
5 $\quad \textsc{Swap}(S[i], S[j])$
6 $i = j = 0$

- 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 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

SPRITZ

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.

SPRITZ-PRG

- 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

$$\underbrace{\texttt{iw+}}_{i},\ \underbrace{\texttt{kjiS+S+}}_{j},\ \underbrace{\texttt{ikjS++}}_{k},\ \underbrace{\texttt{jikz+S+S+S}}_{z}$$

RC4-PRG()

$i = i + 1$
$j = j + S[i]$

SWAP$(S[i], S[j])$
$z = S[S[i] + S[j]]$
return z

SPRITZ-PRG()

$i = i + w$
$j = k + S[j + S[i]]$
$k = i + k + S[j]$
SWAP$(S[i], S[j])$
$z = S[j + S[i + S[z + k]]]$
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 INITIALIZESTATE

- 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

INITIALIZESTATE(N)

```
S[0..N-1] = [0..N-1]
i = j = k = z = a = 0
w = 1
```

SQUEEZE to output r-byte array

SQUEEZE(r)

```
if a > 0                    // last operation was ABSORB
    SHUFFLE()
P = new array of size r
for v = 0 to r − 1
    P[v] = SPRITZ-PRG()
return P
```

Encryption

Encrypt(K, M)

Keysetup(K)
$C = M +$ Squeeze($M.length$)
return C

Keysetup(K)

InitializeState()
Absorb(K)

Spritz-KSA

- 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

- SHUFFLE effects a “random” one-way transformation on the current state.

SHUFFLE()

WHIP($2N$)
CRUSH()
WHIP($2N$)
CRUSH()
WHIP($2N$)
$a = 0$

WHIP

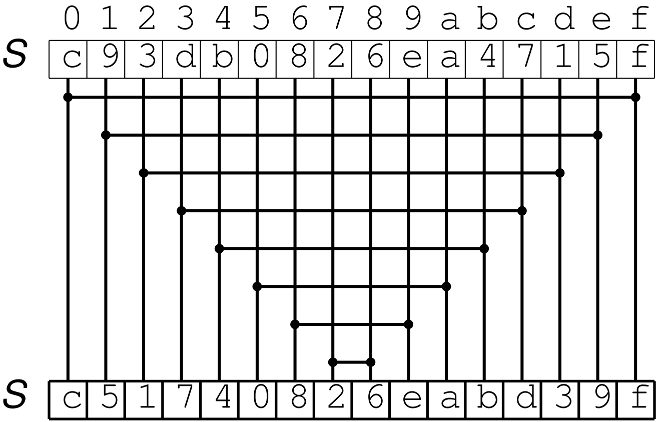
- Purpose of WHIP(r) is to “stir the pot” vigorously, by generating and ignoring r bytes of output, then increasing w by 2 (so w remains odd and relatively prime to 256.)

WHIP(r)

for $v = 0$ **to** $r - 1$
    SPRITZ-PRG()   // output ignored
$w = w + 2$

- (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

ABSORB(K)

for $v = 0$ **to** $K.length - 1$
    ABSORBBYTE($K[v]$)

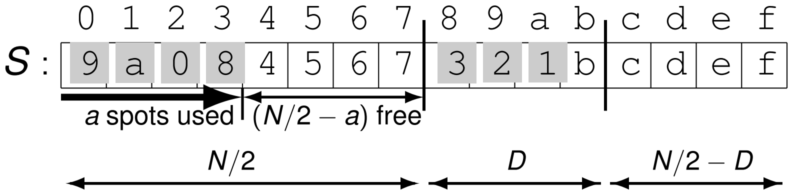
ABSORBBYTE(b)

ABSORBNIBBLE(LOW(b))
ABSORBNIBBLE(HIGH(b))

ABSORBNIBBLE(x)

if $a = \lfloor N/2 \rfloor$
    SHUFFLE()
SWAP($S[a], S[\lfloor N/2 \rfloor + x]$)
$a = a + 1$

AbsorbNibble



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  INITIALIZESTATE(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()

if $a = \lfloor N/2 \rfloor$
    Shuffle()
$a = a + 1$

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)

INITIALIZESTATE()
ABSORB(M); ABSORBSTOP()
ABSORB(r)
return SQUEEZE(r)

Spritz as a MAC

- MAC example with r-byte output.

  MAC(K, M, r)

  ```
  INITIALIZESTATE()
  ABSORB(K); ABSORBSTOP()
  ABSORB(M); ABSORBSTOP()
  ABSORB(r)
  return SQUEEZE(r)
  ```

Outline

RC4

RC4 attacks

Spritz

Security Analysis of Spritz

Performance

Conclusion

Statistical testing

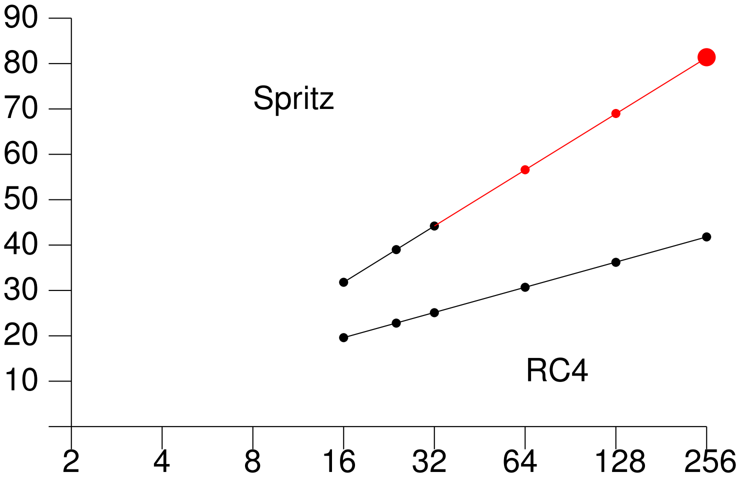
- Primary tool: chi-square testing for uniformity.
- Typical test: chi-square for uniformity of triple $(i, z1, z)$ (aka "`iz1z`") where *zs* is *z* delayed *s* steps. Table has N^3 entries for counts.
- Tests run include `jsj`, `iksk`, `izsz`, `ijsz`, and `iksz` for *s* up to *N*.
- Tested $N = 16$: 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

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

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.

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.
  - $\boxed{\text{Our tests suggest that } 2^{81} \text{ samples are required to distinguish SPRITZ-256 from random.}}$

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

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 of implementation, flexibility, and secure conservative design than its speed.

Outline

Conclusion

SPRITZ is a spongy stream cipher in the style of RC4; it shows excellent statistical properties and great flexibility for applications.

More...

Our paper on SPRITZ is here:

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

More security review needed; comments and analysis appreciated!

Thank you!