The Salsa20 stream cipher
D. J. Bernstein

Thanks to:
University of Illinois at Chicago
NSF CCR–9983950
Alfred P. Sloan Foundation

Salsa20: additive stream cipher, expanding key and nonce into long stream of bytes to add to plaintext.

Key $k$: 16 or 32 bytes.
Same speed either way, simplifying hardware.

Nonce $n$: 8 bytes.
Can send $2^{64}$ messages under one key.

Stream Salsa20$_k(n)$: $2^{70}$ bytes for each message.
Salsa20: additive stream cipher, expanding key and nonce into long stream of bytes to add to plaintext.

Key $k$: 16 or 32 bytes.
Same speed either way, simplifying hardware.

Nonce $n$: 8 bytes.
Can send $2^{64}$ messages under one key.

Stream $\text{Salsa20}_k(n)$: $2^{70}$ bytes for each message.


Given message $m$ with nonce $n$:
Send $(n, c, \text{Poly1305}(s))$ where $(s, c) = \text{Salsa20}_k(n)$.

Very fast; short secret key; provably secure if Salsa20 is secure; better than encrypt-then-MAC.

Easily adapt to “AEAD,” i.e., allow unencrypted header.
Salsa20: additive stream cipher, expanding key and nonce into long stream of bytes to add to plaintext.

Key $k$: 16 or 32 bytes.  
Same speed either way, simplifying hardware.

Nonce $n$: 8 bytes.  
Can send $2^{64}$ messages under one key.

Stream $\text{Salsa20}_k(n)$:  
$2^{70}$ bytes for each message.


Given message $m$ with nonce $n$: Send $(n, c, \text{Poly1305}_r(c, s))$ where $(s, c) = \text{Salsa20}_k(n) \oplus (0, m)$.

Very fast; short secret key ($k, r$); provably secure if Salsa20 is secure; better than encrypt-then-MAC.

Easily adapt to “AEAD,” i.e., allow unencrypted header.
Salsa20: additive stream cipher, expanding key and nonce into long stream of bytes to add to plaintext.

Key: 16 or 32 bytes. Same speed either way, simplifying hardware.

Nonce: 8 bytes. Can send $2^{64}$ messages under one key.

Stream Salsa20: $2^{70}$ bytes for each message.


Given message $m$ with nonce $n$:
Send $(n, c, \text{Poly1305}_r(c, s))$ where $(s, c) = \text{Salsa20}_k(n) \oplus (0, m)$.

Very fast; short secret key $(k, r)$; provably secure if Salsa20 is secure; better than encrypt-then-MAC.

Easily adapt to “AEAD,” i.e., allow unencrypted header.

Let’s watch how Salsa20 generates block of 64 bytes from key $123416$, nonce $(255, 227, 1184, 2000)$. Notation: $\parallel$ means $1 + 2 + 16$.

Little-endian everywhere.

Key:

Nonce:

Given message $m$ with nonce $n$:
Send $(n, c, \text{Poly1305}_r(c, s))$ where $(s, c) = \text{Salsa20}_k(n) \oplus (0, m)$.

Very fast; short secret key $(k, r)$; provably secure if Salsa20 is secure; better than encrypt-then-MAC.

Easily adapt to “AEAD,” i.e., allow unencrypted header.

Let’s watch how Salsa20 generates block of 64 bytes from key $(1, 2, 3, \ldots, 16)$, nonce $(255, 227, 11, 84, 2, 0, 0, 0)$.

Notation: $\mathbf{1}$ means $1 + 2 + 16$. Little-endian everywhere.

Key:

Nonce:

Given message with nonce $n$:

$$\text{Send } (\text{Poly1305}(\ldots)) \text{ where } (\ldots) = \text{Salsa20}(0) \oplus (0, m).$$

Notation: $[\text{I}]$ means $1 + 2 + 16$.

Little-endian everywhere.

Key:

\begin{tabular}{cccc}
\hline
1 & 2 & 3 & 16 \\
\hline
\end{tabular}

Nonce:

\begin{tabular}{cccc}
\hline
255 & 227 & 11 & 84 \\
\hline
2 & 0 & 0 & 0 \\
\hline
\end{tabular}

Let’s watch how Salsa20 generates block of 64 bytes from key $(1, 2, 3, \ldots, 16)$, nonce $(255, 227, 11, 84, 2, 0, 0, 0)$.

Diagonal entries are constants:

Other entries are key, nonce, block counter, key again.

Build $4 \times 4$ array of 4-byte words:

Key:

\begin{tabular}{cccc}
\hline
1 & 2 & 3 & 16 \\
\hline
\end{tabular}

Nonce:

\begin{tabular}{cccc}
\hline
255 & 227 & 11 & 84 \\
\hline
2 & 0 & 0 & 0 \\
\hline
\end{tabular}
Let's watch how Salsa20 generates block of 64 bytes from key \((1, 2, 3, \ldots, 16)\), nonce \((255, 227, 11, 84, 2, 0, 0, 0)\).

Notation: \(\square\) means \(1 + 2 + 16\).

Little-endian everywhere.

Key:

\[
\begin{array}{cccc}
\square & \square & \square & \square \\
\square & \square & \square & \square \\
\square & \square & \square & \square \\
\square & \square & \square & \square \\
\end{array}
\]

Nonce:

\[
\begin{array}{cccc}
\square & \square & \square & \square \\
\square & \square & \square & \square \\
\square & \square & \square & \square \\
\square & \square & \square & \square \\
\end{array}
\]

Build \(4 \times 4\) array of 4-byte words:

Diagonal entries are constants:

Optional entries are key; nonce; block counter; key again.
Let's watch how Salsa20 generates block of 64 bytes from key (1, 2, 3, 16), nonce (255, 227, 11, 84, 2, 0, 0, 0).

Notation: \(1 + 2 + 16\).

Little-endian everywhere.

Key:
.

Nonce:
.

Build 4 \(\times\) 4 array of 4-byte words:

- Diagonal entries are constants:
- Other entries are key; nonce; block counter; key again.

Modify one word using two others:

The modification is very simple:
- add two underlined words;
- rotate left by 7 bits;
- xor into next word down.

\[x[9] \oplus (x[1]+x[5]) \ll 7\]

Will do long series of these simple modifications, as in TEA.
Build 4 × 4 array of 4-byte words:

Diagonal entries are constants:

Other entries are key; nonce; block counter; key again.

Modify one word using two others:

The modification is very simple:
add two underlined words;
rotate left by 7 bits;
xor into next word down.

\[ x[9] \oplus (x[1] + x[5]) \ll 7 \]

Will do long series of these simple modifications, as in TEA.
Build 4

4 array of 4-byte words:
Diagonal entries are constants:
Other entries are key; nonce; block counter; key again.

Modify one word using two others:
The modification is very simple:
add two underlined words;
rotate left by 7 bits;
xor into next word down.

\[ x[9] \^= (x[1]+x[5]) \ll 7 \]

Will do long series of these simple modifications, as in TEA.

Modify other columns:
Columns wrap around from bottom to top.

\[ x[4] \^= (x[12]+x[0]) \ll 7 \]
\[ x[14] \^= (x[6]+x[10]) \ll 7 \]
\[ x[3] \^= (x[11]+x[15]) \ll 7 \]

Total: 4 modifications.
Modify one word using two others:
The modification is very simple:
add two underlined words;
rotate left by 7 bits;
xor into next word down.

\[ x[9] \oplus (x[1] + x[5]) \ll 7 \]

Will do long series of these simple modifications, as in TEA.

Modify other columns:
Columns wrap around from bottom to top.

\[ x[4] \oplus (x[12] + x[0]) \ll 7 \]
\[ x[14] \oplus (x[6] + x[10]) \ll 7 \]
\[ x[3] \oplus (x[11] + x[15]) \ll 7 \]

Total: 4 modifications.
Modify one word using two others:

The modification is very simple: add two underlined words; rotate left by 7 bits; xor into next word down.

\[ x[9] \oplus (x[1]+x[5]) \ll 7 \]

Will do long series of these simple modifications, as in TEA.

Modify other columns:

Columns wrap around from bottom to top.

\[ x[4] \oplus (x[12]+x[0]) \ll 7 \]
\[ x[14] \oplus (x[6]+x[10]) \ll 7 \]
\[ x[3] \oplus (x[11]+x[15]) \ll 7 \]

Total: 4 modifications.

Modify each column again:

This time rotate by 9 bits.

\[ x[8] \oplus (x[0]+x[4]) \ll 9 \]
\[ x[13] \oplus (x[5]+x[9]) \ll 9 \]
\[ x[2] \oplus (x[10]+x[14]) \ll 9 \]
\[ x[7] \oplus (x[15]+x[3]) \ll 9 \]

Total: 8 modifications.
Modify other columns:

- Columns wrap around from bottom to top.

\[
x_4 \oplus= (x_{12}+x_0) \ll 7 \\
x_{14} \oplus= (x_6+x_{10}) \ll 7 \\
x_3 \oplus= (x_{11}+x_{15}) \ll 7
\]

Total: 4 modifications.

Modify each column again:

- This time rotate by 9 bits.

\[
x_8 \oplus= (x_0+x_4) \ll 9 \\
x_{13} \oplus= (x_5+x_9) \ll 9 \\
x_2 \oplus= (x_{10}+x_{14}) \ll 9 \\
x_7 \oplus= (x_{15}+x_3) \ll 9
\]

Total: 8 modifications.
Modify other columns:
Columns wrap around from bottom to top.

\[
x[4] ^= (x[12] + x[0]) <<< 7 \\
x[14] ^= (x[6] + x[10]) <<< 7 \\
\]

Total: 4 modifications.

Modify each column again:
This time rotate by 9 bits.

\[
x[8] ^= (x[0] + x[4]) <<< 9 \\
x[2] ^= (x[10] + x[14]) <<< 9 \\
\]

Total: 8 modifications.

Modify each column again:
This time rotate by 13 bits.

\[
x[12] ^= (x[4] + x[8]) <<< 13 \\
x[1] ^= (x[9] + x[13]) <<< 13 \\
x[6] ^= (x[14] + x[2]) <<< 13 \\
\]

Total: 12 modifications.
Modify each column again:

This time rotate by 9 bits.
\[
x[8] = (x[0] + x[4]) \ll 9 \\
x[13] = (x[5] + x[9]) \ll 9 \\
x[2] = (x[10] + x[14]) \ll 9 \\
x[7] = (x[15] + x[3]) \ll 9
\]
Total: 8 modifications.

Modify each column again:

This time rotate by 13 bits.
\[
x[12] = (x[4] + x[8]) \ll 13 \\
x[1] = (x[9] + x[13]) \ll 13 \\
x[6] = (x[14] + x[2]) \ll 13 \\
\]
Total: 12 modifications.
Modify each column again:
This time rotate by 9 bits.
\[
x[8] \leftarrow (x[0] + x[4]) \ll 9 \\
x[13] \leftarrow (x[5] + x[9]) \ll 9 \\
x[2] \leftarrow (x[10] + x[14]) \ll 9 \\
x[7] \leftarrow (x[15] + x[3]) \ll 9
\]
Total: 8 modifications.

Modify each column again:
This time rotate by 13 bits.
\[
x[12] \leftarrow (x[4] + x[8]) \ll 13 \\
x[1] \leftarrow (x[9] + x[13]) \ll 13 \\
x[6] \leftarrow (x[14] + x[2]) \ll 13 \\
x[11] \leftarrow (x[3] + x[7]) \ll 13
\]
Total: 12 modifications.

Modify each column again:
This time rotate by 18 bits.
\[
x[0] \leftarrow (x[8] + x[12]) \ll 18 \\
x[5] \leftarrow (x[13] + x[1]) \ll 18 \\
x[10] \leftarrow (x[2] + x[6]) \ll 18 \\
x[15] \leftarrow (x[7] + x[11]) \ll 18
\]
Total: 16 modifications.
Modify each column again:

This time rotate by 13 bits.

\[
x[12] ^= (x[4]+x[8]) <<< 13 \\
x[1] ^= (x[9]+x[13]) <<< 13 \\
x[6] ^= (x[14]+x[2]) <<< 13 \\
\]

Total: 12 modifications.

Modify each column again:

This time rotate by 18 bits.

\[
x[0] ^= (x[8]+x[12]) <<< 18 \\
x[5] ^= (x[13]+x[1]) <<< 18 \\
x[10] ^= (x[2]+x[6]) <<< 18 \\
\]

Total: 16 modifications.
Modify each column again:
This time rotate by 13 bits.

\[
x[12] \oplus= (x[4] + x[8]) \ll 13 \\
x[1] \oplus= (x[9] + x[13]) \ll 13 \\
x[6] \oplus= (x[14] + x[2]) \ll 13 \\
x[11] \oplus= (x[3] + x[7]) \ll 13
\]

Total: 12 modifications.

Modify each column again:
This time rotate by 18 bits.

\[
x[0] \oplus= (x[8] + x[12]) \ll 18 \\
x[5] \oplus= (x[13] + x[1]) \ll 18 \\
x[10] \oplus= (x[2] + x[6]) \ll 18 \\
x[15] \oplus= (x[7] + x[11]) \ll 18
\]

Total: 16 modifications.

Modify rows by 7, 9, 13, 18:

Now every word has been modified twice.

Total: 32 modifications.

That’s 2 rounds of Salsa20.
Modify each column again:

\[
x[0] \oplus (x[8] + x[12]) \ll 18
\]
\[
x[5] \oplus (x[13] + x[1]) \ll 18
\]
\[
x[10] \oplus (x[2] + x[6]) \ll 18
\]
\[
x[15] \oplus (x[7] + x[11]) \ll 18
\]

Total: 16 modifications.

Now every word has been modified twice.

Total: 32 modifications.
That’s 2 rounds of Salsa20.
Modify each column again:

This time rotate by 18 bits.

\[ \begin{align*}
  x[0] &\quad ^{\oplus} (x[8] + x[12]) \lll 18 \\
  x[5] &\quad ^{\oplus} (x[13] + x[1]) \lll 18 \\
  x[10] &\quad ^{\oplus} (x[2] + x[6]) \lll 18 \\
\end{align*} \]

Total: 16 modifications.

Modify rows by 7, 9, 13, 18:

Now every word has been modified twice.

Total: 32 modifications.

That’s 2 rounds of Salsa20.

Repeat column modifications:

Now every word has been modified 3 times.

Total: 48 modifications.

That’s 3 rounds of Salsa20.
Modify rows by 7, 9, 13, 18:

Now every word
has been modified twice.
Total: 32 modifications.
That’s 2 rounds of Salsa20.

Repeat column modifications:

Now every word
has been modified 3 times.
Total: 48 modifications.
That’s 3 rounds of Salsa20.
Repeat column modifications:
Now every word has been modified 3 times.
Total: 48 modifications.
That’s 3 rounds of Salsa20.

Repeat row modifications:
Now every word has been modified 4 times.
Total: 64 modifications.
That’s 4 rounds of Salsa20.
Repeat column modifications:

Now every word
has been modified 3 times.
Total: 48 modifications.
That’s 3 rounds of Salsa20.

Repeat row modifications:

Now every word
has been modified 4 times.
Total: 64 modifications.
That’s 4 rounds of Salsa20.
Repeat column modifications:

Now every word has been modified 3 times.
Total: 48 modifications.
That’s 3 rounds of Salsa20.

Repeat row modifications:

Now every word has been modified 4 times.
Total: 64 modifications.
That’s 4 rounds of Salsa20.

Continue for 20 rounds total:
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows.

First block of Salsa20 output is final array plus original:
x[0]+z[0], ..., x[15]+z[15].

For subsequent blocks: Repeat with block counter 1, 2, etc.
Parallelizable. Very small state.
Repeat row modifications:

Now every word has been modified 4 times.
Total: 64 modifications.
That’s 4 rounds of Salsa20.

Continue for 20 rounds total:
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows.

First block of Salsa20 output is final array plus original array:
\[ x[0]+z[0], \ldots, x[15]+z[15]\].

For subsequent blocks: Repeat with block counter 1, 2, etc.
Parallelizable. Very small state.
Repeat row modifications:

Now every word has been modified 4 times.
Total: 64 modifications.
That's 4 rounds of Salsa20.

Continue for 20 rounds total:
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows.

First block of Salsa20 output is final array plus original array:
\( x[0] + z[0], \ldots, x[15] + z[15] \).

For subsequent blocks: Repeat with block counter 1, 2, etc.
Parallelizable. Very small state.

Change in starting array for block 1:
Let's watch how this change affects subsequent rounds.
Changes shown here by xor.
Continue for 20 rounds total:
columns, rows, columns, rows, 
columns, rows, columns, rows, 
columns, rows, columns, rows, 
columns, rows, columns, rows, 
columns, rows, columns, rows.

First block of Salsa20 output is final array plus original array:
\[ x[0]+z[0], \ldots, x[15]+z[15]. \]

For subsequent blocks: Repeat with block counter 1, 2, etc.
Parallelizable. Very small state.

Change in starting array for block 1:

Let's watch how this change affects subsequent rounds.
Changes shown here by xor.
Continue for 20 rounds total:
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows,
columns, rows, columns, rows.

First block of Salsa20 output is
final array plus original array:
x[0] + z[0] + z[15].
For subsequent blocks: Repeat
with block counter 1, 2, etc.
Parallelizable. Very small state.

Let’s watch how this change affects subsequent rounds.
Changes shown here by xor.

Change in starting array for block 1:

Let’s watch how this change affects subsequent rounds.
Changes shown here by xor.

Change after one round:
Difference has propagated
to two other entries in the same column.
Depends on a few carries,
but still highly predictable.

Difference has propagated
to two other entries in the same column.
Depends on a few carries,
but still highly predictable.
Change in starting array for block 1:

Let’s watch how this change affects subsequent rounds.

Changes shown here by xor.

Change after one round:

Difference has propagated to two other entries in the same column.
Depends on a few carries, but still highly predictable.
Change in starting array for block 1:

Let's watch how this change affects subsequent rounds.
Changes shown here by xor.

Change after one round:
Difference has propagated to two other entries in the same column. Depends on a few carries, but still highly predictable.

Change after two rounds:
Difference has propagated across columns.
Change after one round:

Difference has propagated to two other entries in the same column. Depends on a few carries, but still highly predictable.

Change after two rounds:

Difference has propagated across columns.
Change after one round:
Difference has propagated to two other entries in the same column. Depends on a few carries, but still highly predictable.

Change after two rounds:
Difference has propagated across columns.

Change after three rounds:
Every word has been affected. A substantial fraction of bits are now active.
Change after two rounds:

Difference has propagated across columns.

Change after three rounds:

Every word has been affected.
A substantial fraction of bits are now active.
Change after two rounds:
Difference has propagated across columns.

Change after three rounds:
Every word has been affected.
A substantial fraction of bits are now active.

Change after four rounds:
Hundreds of active bits in every subsequent round.
Total > 4000 active bits interacting with carries in a random-looking way.
Change after three rounds:

Every word has been affected.

A substantial fraction of bits are now active.

Change after four rounds:

Hundreds of active bits in every subsequent round.

Total $> 4000$ active bits interacting with carries in a random-looking way.
Change after three rounds:
Every word has been affected.
A substantial fraction of bits
are now active.

Change after four rounds:
Hundreds of active bits
in every subsequent round.
Total > 4000 active bits
interacting with carries
in a random-looking way.

Surprise: Salsa20 is fast!
My current public-domain software:
26.75 Athlon cycles/round.
37.5 Pentium III cycles/round.
48 Pentium 4 f12 cycles/round.
33.75 Pentium M cycles/round.
24.5 PowerPC 7410 cycles/round.
33 PowerPC RS64 IV cycles/round.
40.5 UltraSPARC II cycles/round.
41 UltraSPARC III cycles/round.
Change after four rounds:

Hundreds of active bits
in every subsequent round.
Total > 4000 active bits
interacting with carries
in a random-looking way.

Surprise: Salsa20 is fast!

My current public-domain software:
26.75 Athlon cycles/round.
37.5 Pentium III cycles/round.
48 Pentium 4 f12 cycles/round.
33.75 Pentium M cycles/round.
24.5 PowerPC 7410 cycles/round.
33 PowerPC RS64 IV cycles/round.
40.5 UltraSPARC II cycles/round.
41 UltraSPARC III cycles/round.
Surprise: Salsa20 is fast!

My current public-domain software:
26.75 Athlon cycles/round.
37.5 Pentium III cycles/round.
48 Pentium 4 f12 cycles/round.
33.75 Pentium M cycles/round.
24.5 PowerPC 7410 cycles/round.
33 PowerPC RS64 IV cycles/round.
40.5 UltraSPARC II cycles/round.
41 UltraSPARC III cycles/round.

Multiply by 20 for 20 rounds, divide by 64 for cycles/byte
but rounds aren’t everything.
I still need to optimize for block counting, xor, etc.; combine with Poly1305; do comprehensive benchmarks.

But it’s clear that Salsa20 will be at least as fast as AES, sometimes much faster, depending on the CPU.
Surprise: Salsa20 is fast!

My current public-domain software:

- 26.75 Athlon cycles/round.
- 37.5 Pentium III cycles/round.
- 48 Pentium 4 f12 cycles/round.
- 33.75 Pentium M cycles/round.
- 24.5 PowerPC 7410 cycles/round.
- 33 PowerPC RS64 IV cycles/round.
- 40.5 UltraSPARC II cycles/round.
- 41 UltraSPARC III cycles/round.

Multiply by 20 for 20 rounds, divide by 64 for cycles/byte

... but rounds aren't everything.

I still need to optimize code for block counting, xor, etc.; combine with Poly1305; do comprehensive benchmarks.

But it's clear that Salsa20 will be at least as fast as AES, sometimes much faster, depending on the CPU.
Surprise: Salsa20 is fast!

My current public-domain software:

- 26.75 Athlon cycles/round.
- 37.5 Pentium III cycles/round.
- 48.5 Pentium 4 cycles/round.
- 33.75 Pentium M cycles/round.
- 24.5 PowerPC 7410 cycles/round.
- 33 PowerPC RS64 IV cycles/round.
- 40.5 UltraSPARC II cycles/round.
- 41 UltraSPARC III cycles/round.

Multiply by 20 for 20 rounds, divide by 64 for cycles/byte... but rounds aren't everything.

I still need to optimize code for block counting, xor, etc.; combine with Poly1305; do comprehensive benchmarks.

But it's clear that Salsa20 will be at least as fast as AES, sometimes much faster, depending on the CPU.

Here AES has 16-byte key; slower with 32-byte key. Salsa20 has no such slowdown.

AES becomes even slower if key is not pre-expanded. Salsa20 has no precomputation.

AES has serious timing leaks: see http://cr.yp.to/papers.html#cachetiming for successful AES key extraction. Constant-time AES is very slow. Salsa20 has no timing leaks.
Multiply by 20 for 20 rounds, divide by 64 for cycles/byte
... but rounds aren’t everything.

I still need to optimize code for block counting, xor, etc.; combine with Poly1305; do comprehensive benchmarks.

But it’s clear that Salsa20 will be at least as fast as AES, sometimes much faster, depending on the CPU.

Here AES has 16-byte key; slower with 32-byte key. Salsa20 has no such slowdown.

AES becomes even slower if key is not pre-expanded. Salsa20 has no precomputation.

AES has serious timing leaks: see http://cr.yp.to/papers.html#cachetiming for successful AES key extraction. Constant-time AES is very slow. Salsa20 has no timing leaks.
Multiply by 20 for 20 rounds, divide by 64 for cycles/byte. But rounds aren’t everything.

I still need to optimize code for block counting, xor, etc.; combine with Poly1305; do comprehensive benchmarks.

But it’s clear that Salsa20 will be at least as fast as AES, sometimes much faster, depending on the CPU.

Here AES has 16-byte key; slower with 32-byte key. Salsa20 has no such slowdown.

AES becomes even slower if key is not pre-expanded. Salsa20 has no precomputation.

AES has serious timing leaks: see http://cr.yp.to/papers.html#cachetiming for successful AES key extraction. Constant-time AES is very slow. Salsa20 has no timing leaks.

I offer $1000 prize for the public Salsa20 cryptanalysis that I consider most interesting. Awarded at the end of 2005.
Send URLs of your papers to snuffle@box.cr.yp.to.
Here AES has 16-byte key; slower with 32-byte key. Salsa20 has no such slowdown.

AES becomes even slower if key is not pre-expanded. Salsa20 has no precomputation.

AES has serious timing leaks: see http://cr.yp.to/papers.html#cachetiming for successful AES key extraction. Constant-time AES is very slow. Salsa20 has no timing leaks.

I offer $1000 prize for the public Salsa20 cryptanalysis that I consider most interesting. Awarded at the end of 2005. Send URLs of your papers to snuffle@box.cr.yp.to.