The Salsa20 family of stream ciphers

Daniel J. Bernstein *

Department of Mathematics, Statistics, and Computer Science (M/C 249)
The University of Illinois at Chicago
Chicago, IL 60607–7045
snuffle6@box.cr.yp.to

Abstract. Salsa20 is a family of 256-bit stream ciphers designed in 2005 and submitted to eSTREAM, the ECRYPT Stream Cipher Project. Salsa20 has progressed to the third round of eSTREAM without any changes. The 20-round stream cipher Salsa20/20 is consistently faster than AES and is recommended by the designer for typical cryptographic applications. The reduced-round ciphers Salsa20/12 and Salsa20/8 are among the fastest 256-bit stream ciphers available and are recommended for applications where speed is more important than confidence. The fastest known attacks use $\approx 2^{153}$ simple operations against Salsa20/7, $\approx 2^{249}$ simple operations against Salsa20/8, and $\approx 2^{255}$ simple operations against Salsa20/9, Salsa20/10, etc. In this paper, the Salsa20 designer presents Salsa20 and discusses the decisions made in the Salsa20 design.

1 Introduction

A sender and receiver share a short secret key. They use the secret key to encrypt a series of messages. A message could be short, just a few bytes, but it could be much longer, perhaps gigabytes. The series of messages could be short, just one message, but it could be much longer, perhaps billions of messages.

The sender and receiver encrypt messages using an **encryption function**: a function that produces the first ciphertext from the key and the first plaintext, that produces the second ciphertext from the key and the second plaintext, etc.

An encryption function has to be fast. Many senders have to encrypt large volumes of data in very little time using limited resources. Many receivers are faced with even larger volumes of data—not just the legitimate messages but also a flood of forgery attempts. A slow encryption function can satisfy some senders and receivers, but my focus is on encryption functions suitable for a wider range of applications.

An encryption function also has to be secure. Many users are facing, or at least think that they are facing, years of cryptanalytic computations by well-funded attackers equipped with millions of fast parallel processors. Some users

* Permanent ID of this document: 31364286077dcdff8e4509f9ff3139ad. Date of this document: 2007.12.25. This work was supported by the National Science Foundation under grants CCR–9983950 and ITR–0716498, and by the Alfred P. Sloan Foundation.

				Cycles/byte					
			Salsa20	Salsa20/8		Salsa20/12		Salsa20/20	
Arch	MHz	Machine	software	long	576	long	576	long	576
amd64	3000	Xeon 5160 (6f6)	amd64-xmm6	1.88	2.07	2.80	3.25	3.93	4.25
amd64	2137	Core 2 Duo (6f6)	amd64-xmm6	1.88	2.07	2.57	2.80	3.91	4.33
ppc32	533	PowerPC G4 7410	ppc-altivec	1.99	2.14	2.74	2.88	4.24	4.39
x86	2137	Core 2 Duo (6f6)	x86-xmm5	2.06	2.28	2.80	3.15	4.32	4.70
amd64	2000	Athlon 64 X2 (15,75,2)	amd64-3	3.47	3.65	4.86	5.04	7.64	7.84
ppc64	2000	PowerPC G5 970	ppc-altivec	3.28	3.48	4.83	4.87	7.82	8.04
amd64	2391	Opteron (f5a)	amd64-3	3.78	3.96	5.33	5.51	8.42	8.62
amd64	2192	Opteron (f58)	amd64-3	3.82	4.18	5.35	5.73	8.42	8.78
x86	2000	Athlon 64 X2 (15,75,2)	x86-1	4.50	4.78	6.27	6.55	9.80	10.07
x86	900	Athlon (622)	x86-athlon	4.61	4.84	6.44	6.65	10.04	10.24
ppc64	1452	POWER4	merged	6.83	7.00	8.35	8.51	11.29	11.47
hppa	1000	PA-RISC 8900	merged	5.82	5.97	7.68	7.85	11.39	11.56
amd64	3000	Pentium D (f64)	amd64-xmm6	5.38	5.87	7.19	7.84	10.69	11.73
x86	1300	Pentium M (695)	x86-xmm5	5.30	5.53	7.44	7.70	11.70	11.98
x86	3000	Xeon (f26)	x86-xmm5	5.30	5.86	7.41	8.21	11.64	12.55
x86	3200	Xeon (f25)	x86-xmm5	5.30	5.84	7.40	8.15	11.63	12.59
x86	2800	Xeon (f29)	x86-xmm5	5.33	5.95	7.44	8.20	11.67	12.65
x86	3000	Pentium 4 (f41)	x86-xmm5	5.76	6.92	8.12	9.33	11.84	13.40
x86	1400	Pentium III (6b1)	x86-mmx	6.37	6.79	8.88	9.29	13.88	14.29
sparc	1050	UltraSPARC IV	sparc	6.65	6.76	9.21	9.33	14.34	14.45
x86	3200	Pentium D (f47)	x86-athlon	7.13	7.66	9.90	10.31	15.29	15.94
ia64	1500	Itanium II	merged	8.49	8.87	12.42	12.62	18.07	18.27
ia64	1400	Itanium II	merged	8.28	8.65	12.56	12.76	18.21	18.40

Table 1.1. Salsa20 software speeds; measured by the official eSTREAM benchmarking framework; sorted by final column. "576" means single-core cycles/byte to encrypt a 576-byte packet; "long" means single-core cycles/byte to encrypt a long stream.

are satisfied with lower levels of security, but again my focus is on encryption functions suitable for a wider range of applications.

There is a conflict between these desiderata. One can reasonably conjecture, for example, that every function that encrypts data in 0.5 Core-2 cycles/byte is breakable. One can also conjecture that *almost every* function that encrypts data in 5 Core-2 cycles/byte is breakable. On the other hand, several unbroken submissions to eSTREAM, the ECRYPT Stream Cipher Project, encrypt data in fewer than 5 Core-2 cycles/byte.

In particular, my 20-round stream cipher Salsa20/20 encrypts data in 3.93 Core-2 cycles/byte. (For comparison: Matsui and Nakajima recently reported 9.2 Core-2 cycles/byte for 10-round AES using a pre-expanded 128-bit key. See [18].) The fastest known attack against Salsa20/20 is a 256-bit brute-force search. I recommend Salsa20/20 for encryption in typical cryptographic applications.

Reduced-round ciphers in the Salsa20 family are attractive options for users who value speed more highly than confidence. The 12-round stream cipher Salsa20/12 encrypts data in 2.80 Core-2 cycles/byte; the fastest known attack

against Salsa20/12 is a 256-bit brute-force search. The 8-round stream cipher Salsa20/8 encrypts data in 1.88 Core-2 cycles/byte; as discussed in Section 5, papers by several cryptanalysts have culminated in an attack against Salsa20/8 taking "only" 2^{249} operations, but this is far beyond any computation that will be carried out in the foreseeable future. Perhaps better attacks will be developed, but competing ciphers at similar speeds seem to be much more easily broken!

I hadn't heard of the Core 2 when I designed Salsa20. I was aiming for high speed on a wide variety of platforms; I don't find it surprising that Salsa20 is able to take advantage of a new platform. Table 1.1 shows Salsa20's software speeds on various CPUs.

This paper defines Salsa20 and explains the decisions that I made in the Salsa20 design. Section 2 discusses the selection of low-level operations used in Salsa20—a deliberately limited set, in particular with no S-boxes. Section 3 discusses the high-level data flow in Salsa20—again quite limited, in particular with no communication across blocks aside from a simple block counter. Section 4 discusses the middle-level structure of Salsa20. Section 5 reviews known attacks on Salsa20.

2 Low level: Which operations are used?

2.1 What does Salsa20 do?

The Salsa20 encryption function is a long chain of three simple operations on 32-bit words:

- 32-bit addition, producing the sum $a + b \bmod 2^{32}$ of two 32-bit words a, b;
- 32-bit exclusive-or, producing the xor $a \oplus b$ of two 32-bit words a, b; and
- constant-distance 32-bit rotation, producing the rotation $a \lll b$ of a 32-bit word a by b bits to the left, where b is constant.

On occasion I encounter the superstitious notion that these operations are "too simple." In fact, these operations can easily simulate any circuit, and are therefore capable of reaching the same security level as any other selection of operations. The real question for the cipher designer is whether a different mix of operations could achieve the same security level *at higher speed*.

2.2 Should there be integer multiplications?

Some popular CPUs can quickly compute $xy \bmod 2^{64}$, given x, y. Some ciphers are designed to take advantage of this operation. Sometimes one of x, y is a constant; sometimes x, y are both variables.

The basic argument for integer multiplication is that the output bits are complicated functions of the input bits, mixing the inputs more thoroughly than a few simple integer operations.

The basic counterargument is that integer multiplication takes several cycles on the fastest CPUs, and many more cycles on other CPUs. For comparison, a

comparably complex series of simple integer operations is always reasonably fast. Multiplication might be slightly faster on some CPUs but it is not *consistently* fast.

I do like the amount of mixing provided by multiplication, and I'm impressed with the fast multiplication circuits included (generally for non-cryptographic reasons) in many CPUs, but the potential speed benefits don't seem big enough to outweigh the massive speed penalty on other CPUs. Similar comments apply to 64-bit additions, to 32-bit multiplications, and to variable-distance ("data-dependent") rotations.

A further argument against integer multiplication is that it increases the risk of timing leaks. What really matters is not the speed of integer multiplication, but the speed of *constant-time* integer multiplication, which is often much slower.

Example: On the Motorola PowerPC 7450 (G4e), a fairly common general-purpose CPU, the `mull` multiplication instruction usually takes 2 cycles (with 4-cycle latency), but it takes only 1 cycle (with 3-cycle latency) if "the 15 msbs of the B operand are either all set or all cleared." See [1, page 6.45]. The same is true for the 8641D, the newest CPU in the same family. It is possible to eliminate the timing leak on these CPUs by, e.g., using the floating-point multiplier, but moving data back and forth to floating-point registers costs CPU cycles, not to mention extra programming effort.

2.3 Should there be S-box lookups?

An **S-box lookup** is an array lookup using an input-dependent index. Most ciphers are designed to take advantage of this operation. For example, typical high-speed AES software has several 1024-byte S-boxes, each of which converts 8-bit inputs to 32-bit outputs.

The basic argument for S-boxes is that a single table lookup can mangle its input quite thoroughly—more thoroughly than a chain of a few simple integer operations taking the same amount of time.

The basic counterargument is that a simple integer operation takes one or two 32-bit inputs rather than one 8-bit input, so it effectively mangles several 8-bit inputs at once. It is not obvious that a series of S-box lookups—even with rather large S-boxes, as in AES, increasing L1 cache pressure on large CPUs and forcing different implementation techniques for small CPUs—is faster than a comparably complex series of integer operations.

A further argument against S-box lookups is that, on most platforms, they are vulnerable to timing attacks. NIST's statement to the contrary in [19, Section 3.6.2] (table lookup is "not vulnerable to timing attacks") is erroneous. It is extremely difficult to work around this problem without sacrificing a tremendous amount of speed. See my paper [5] for much more information on this topic, including an example of successful remote extraction of a complete AES key.

For me, the timing-attack problem is decisive. For any particular security level, I'm not sure whether adding S-box lookups would gain speed, but I'm sure that adding *constant-time* S-box lookups would *not* gain speed.

Salsa20 is certainly not the first cipher without S-boxes. The Tiny Encryption Algorithm, published by Wheeler and Needham in [23], is a classic example of a reduced-instruction-set cipher: it is a long chain of 32-bit shifts, 32-bit xors, and 32-bit additions. IDEA, published by Lai, Massey, and Murphy in [17], is even older and almost as simple: it is a long chain of 16-bit additions, 16-bit xors, and multiplications modulo $2^{16} + 1$.

2.4 Should there be fewer rotations?

Rotations account for about 1/3 of the integer operations in Salsa20. If rotations are simulated by shift-shift-xor (as they are on the UltraSPARC and with XMM instructions) then they account for about 1/2 of the integer operations in Salsa20. Replacing some of the rotations with a comparable number of additions might achieve comparable diffusion in less time.

The reader may be wondering why I used rotations rather than shifts. The basic argument for rotations is that one xor of a rotated quantity provides as much diffusion as two xors of shifted quantities. There does not appear to be a counterargument. Rotate-xor is faster than shift-shift-xor-xor on many CPUs and is never slower.

3 High level: How do blocks interact?

3.1 What does Salsa20 do?

Salsa20 expands a 256-bit key and a 64-bit nonce (unique message number) into a 2^{70}-byte stream. It encrypts a b-byte plaintext by xor'ing the plaintext with the first b bytes of the stream and discarding the rest of the stream. It decrypts a b-byte ciphertext by xor'ing the ciphertext with the first b bytes of the stream. There is no feedback from the plaintext or ciphertext into the stream.

Salsa20 generates the stream in 64-byte (512-bit) blocks. Each block is an independent hash of the key, the nonce, and a 64-bit block number; there is no chaining from one block to the next. The Salsa20 output stream can therefore be accessed randomly, and any number of blocks can be computed in parallel.

There are no hidden preprocessing costs in Salsa20. In particular, Salsa20 does not preprocess the key before generating a block; each block uses the key directly as input. Salsa20 also does not preprocess the nonce before generating a block; each block uses the nonce directly as input.

3.2 Should encryption and decryption be different?

The most common model of a stream cipher is that each ciphertext block is the xor of the plaintext block and the stream block at the same position. Each stream block is determined by its position, the nonce, the key, and the previous blocks of plaintext—equivalently, the previous blocks of ciphertext. Salsa20 follows this model, as does any block cipher in counter mode, OFB mode, CFB mode, et al.

Some ciphers mangle plaintext in a more complicated way. Consider, for example, AES in CBC mode: the nth plaintext block p_n is converted into the nth ciphertext block c_n by the formula $c_n = \mathrm{AES}_k(c_{n-1} \oplus p_n)$.

The popularity of CBC appears to be a historical accident. I have found very few people arguing for CBC over counter mode, and none of the arguments are even marginally convincing. On occasion I encounter the superstitious notion that encryption by xor is "too simple"; but a one-time pad (in conjunction with, for example, a Gilbert/MacWilliams/Sloane authenticator) provably achieves perfect secrecy (and any desired level of integrity), so there is obviously nothing wrong with xor.

There are several clear arguments against CBC. One disadvantage of CBC is that it requires different code for encryption and decryption, increasing costs in many contexts. Another disadvantage of CBC is that the extra communication from the cryptanalyst into the cipher state is a security threat; regaining the original level of confidence means adding rounds, taking additional time.

There is a security proof for CBC. How, then, can I claim that CBC is less secure than counter mode? One answer is that CBC's security guarantee assumes that the block cipher outputs *for attacker-controlled inputs* are indistinguishable from uniform, whereas counter mode applies the block cipher to *highly restricted* inputs, with many input bits forced to be 0. There are many examples in the literature of block ciphers for which CBC has been broken but counter mode is unbroken.

3.3 Should the stream depend on the plaintext?

A more restricted model of a stream cipher is that ciphertext is plaintext xor stream, where the stream is determined by the nonce and the key. The plaintext and ciphertext do not affect the stream. Salsa20 follows this model, as does any block cipher in counter mode.

Some stream ciphers violate this model: they produce a stream that depends on the plaintext. One example is Helix, published in [13] by Ferguson, Whiting, Schneier, Kelsey, Lucks, and Kohno. The tweaked cipher Phelix was submitted to eSTREAM by Whiting, Schneier, Lucks, and Muller.

The basic argument for incorporating plaintext into the stream (specifically, incorporating plaintext blocks into subsequent blocks of the stream) is that this allows message authentication "for free." After encrypting the plaintext, one can generate a constant number of additional stream blocks and output those blocks as an authenticator of the plaintext.

One counterargument is that "free" is a wild exaggeration. Incorporating the plaintext into the stream takes time for every block, and generating an authenticator takes time for every message.

Another counterargument is that the incorporation of plaintext, being extra communication from the cryptanalyst into the cipher state, is a security threat. Regaining the original level of confidence means adding rounds, which takes additional time for every block.

Another counterargument is that state-of-the-art 128-bit authenticators can be computed in just a few cycles per byte. This may exceed the cost of "free" authentication for *legitimate* packets, but it is much less expensive than "free" authentication for *forged* packets, because it skips the cost of decryption.

For me, the cost of rejecting forged packets is decisive. Consider a denial-of-CPU-service attack in which an attacker floods a CPU with forged packets through a large network. In this situation, a traditional authenticator, such as Poly1305 from [4], is capable of handling a substantially larger flood than a "free" authenticator. See [9] for a new strategy to compute authenticators at even higher speeds.

The idea of incorporating plaintext into the stream clearly deserves further study for users who value authenticated-encryption performance more highly than forgery-rejection performance. In [8] I reported speed measurements for many authenticated-encryption methods; Phelix provided impressive speeds for authenticated encryption and verified decryption. Phelix was later eliminated from eSTREAM for reasons I consider frivolous, namely an "attack" against users who have trouble counting $1, 2, 3, \ldots$; I have no idea why this "attack" should eliminate an attractive option for users who *are* able to count $1, 2, 3, \ldots$.

3.4 Should there be more state?

Salsa20 carries minimal state between blocks. Each block of the stream is a separate hash of the key, the nonce, and the block counter.

Most stream ciphers use a larger state, reusing portions of the first-block computation as input to the second-block computation, reusing portions of the second-block computation as input to the third-block computation, etc.

The argument for a larger state is that one does not need as many cipher rounds to achieve the same conjectured security level. Copying state across blocks seems to provide just as much mixing as the first few cipher rounds. A larger state therefore saves some time after the first block.

One counterargument is that a larger state reduces the number of communication channels that can be handled simultaneously by limited hardware. Ciphers that chain between blocks typically use 64 or more bytes for each channel. With Salsa20, each channel uses just 32 bytes for a key (less if several channels share a key), at most 8 bytes for a nonce, and at most 8 bytes for a block counter.

Another counterargument is that reuse forces serialization. Chaining between blocks prohibits random access to the stream (unless the stream is precomputed and saved, consuming memory). Chaining between blocks means that one cannot take advantage of extra hardware to reduce the latency of computing a long stream.

For me, the serialization problem is decisive. Inability to exploit parallelism is often a disaster. A few extra rounds are often undesirable but are never a disaster.

Case study (due to Wei Dai): As discussed in Section 4, there are 4 parallel 32-bit operations in each step of computing a Salsa20 block. The Core 2 CPU has more parallelism than this: it can carry out (in each core) 12 parallel 32-bit

arithmetic operations. Fortunately, thanks to the lack of chaining, there are 16 parallel 32-bit operations in each step of computing 4 consecutive Salsa20 blocks.

3.5 Should blocks be larger than 64 bytes?

Salsa20 hashes its key, nonce, and block counter into a 64-byte block. Similar structures could easily produce a larger block.

The basic argument for a larger block size, say 256 bytes, is that one does not need as many cipher rounds to achieve the same conjectured security level. Using a larger block size, like copying state across blocks, seems to provide just as much mixing as the first few cipher rounds. A larger state therefore saves time.

The basic counterargument is that a larger block size also loses time. On most CPUs, the communication cost of sweeping through a 256-byte block is a bottleneck; CPUs are designed for computations that don't involve so much data.

Another way that a larger block size loses time is by increasing the overhead for inconvenient message sizes. Expanding a 300-byte message to 512 bytes is much more expensive than expanding it to 320 bytes.

3.6 Should keys be smaller than 256 bits?

The original eSTREAM call for submissions asked for 128-bit software ciphers and 80-bit hardware ciphers. Salsa20 is a 256-bit cipher; it allows smaller keys as options, but I recommend 256-bit keys.

Larger keys are more expensive than smaller keys, especially in hardware. Are they necessary for security?

The basic argument for 128-bit keys is that they will never be found by a brute-force attack. If checking about 2^{20} keys per second requires a CPU costing about 2^6 euros, then searching 2^{128} keys in a year will cost an inconceivable 2^{89} euros.

The basic counterargument is that 128-bit keys *will* be found by a brute-force attack. Here are three reasons that 2^{89} euro-years is a wild exaggeration, even without any improvements in computer technology:

- The attacker can succeed in far fewer than 2^{128} computations. He reaches success probability p after just $2^{128}p$ computations.
- More importantly, each key-checking circuit costs far less than 2^6 euros, at least in bulk: 2^{10} or more key-checking circuits can fit into a single chip, effectively reducing the attacker's costs by a factor of 2^{10}.
- Even more importantly, if the attacker simultaneously attacks (say) 2^{40} keys, he can effectively reduce his costs by a factor of 2^{40}.

One can counter the third cost reduction by putting extra randomness into nonces, but putting the same extra randomness into keys is less expensive. See [7] for a much more detailed discussion of these issues.

I predict that future cryptographers will settle on 256-bit keys as providing a comfortable security level. They will regard 80-bit keys as a silly historical mistake, and 128-bit keys as uncomfortably risky.

4 Medium level: How is a block generated?

4.1 What does Salsa20 do?

The goal of the Salsa20 core, as discussed in Section 3, is to produce a 64-byte block given a key, nonce, and block counter. The tools available to the Salsa20 core, as discussed in Section 2, are addition, xor, and constant-distance rotation of 32-bit words.

The Salsa20 core builds an array of 16 words containing the constant word `0x61707865`, the first 4 key words, the constant word `0x3320646e`, the 2 nonce words, the 2 block-counter words, the constant word `0x79622d32`, the remaining 4 key words, and the constant word `0x6b206574`. Strings are always interpreted in little-endian form. (Most current CPUs take extra time for big-endian accesses, while big-endian CPUs generally have good support for little-endian accesses.)

For example, here is the starting array for key $(1, 2, 3, 4, 5, \ldots, 32)$, nonce $(3, 1, 4, 1, 5, 9, 2, 6)$, and block 7:

`0x61707865`, `0x04030201`, `0x08070605`, `0x0c0b0a09`,
`0x100f0e0d`, `0x3320646e`, `0x01040103`, `0x06020905`,
`0x00000007`, `0x00000000`, `0x79622d32`, `0x14131211`,
`0x18171615`, `0x1c1b1a19`, `0x201f1e1d`, `0x6b206574`.

The diagonal constants are the same for every block, every nonce, and every 32-byte key. As an extra (non-recommended) option, Salsa20 can use a 16-byte key, repeated to form a 32-byte key; in this case the diagonal constants change to `0x61707865`, `0x3120646e`, `0x79622d36`, `0x6b206574`. Salsa20 can also use a 10-byte key, zero-padded to form a 16-byte key; in this case the diagonal constants change to `0x61707865`, `0x3120646e`, `0x79622d30`, `0x6b206574`.

Salsa20 now modifies each below-diagonal word as follows: add the diagonal and above-diagonal words, rotate left by 7 bits, and xor into the below-diagonal words. The result is the following array:

`0x61707865`, `0x04030201`, `0x08070605`, `0x95b0c8b6`,
`0xd3c83331`, `0x3320646e`, `0x01040103`, `0x06020905`,
`0x00000007`, `0x91b3379b`, `0x79622d32`, `0x14131211`,
`0x18171615`, `0x1c1b1a19`, `0x130804a0`, `0x6b206574`.

The underlined words were added, and the next word was modified.

Salsa20 then modifies each below-below-diagonal word as follows: add the diagonal and below-diagonal words, rotate left by 9 bits, and xor into the below-

below-diagonal words. The result is the following array:

0x61707865, 0x04030201, 0xdc64a31d, 0x95b0c8b6,
0xd3c83331, 0x3320646e, 0x01040103, 0xa45e5d04,
0x71572c6d, 0x91b3379b, 0x79622d32, 0x14131211,
0x18171615, 0xbb230990, 0x130804a0, 0x6b206574.

Salsa20 continues down each column, rotating left by 13 bits:

0x61707865, 0xcc266b9b, 0xdc64a31d, 0x95b0c8b6,
0xd3c83331, 0x3320646e, 0x95f3bcee, 0xa45e5d04,
0x71572c6d, 0x91b3379b, 0x79622d32, 0xf0a45550,
0xf3e4deb6, 0xbb230990, 0x130804a0, 0x6b206574.

Salsa20 then modifies the diagonal words, this time rotating left by 18 bits:

0x4dfdec95, 0xcc266b9b, 0xdc64a31d, 0x95b0c8b6,
0xd3c83331, 0xe78e794b, 0x95f3bcee, 0xa45e5d04,
0x71572c6d, 0x91b3379b, 0xf94fe453, 0xf0a45550,
0xf3e4deb6, 0xbb230990, 0x130804a0, 0xa272317e.

Salsa20 finally transposes the array:

0x4dfdec95, 0xd3c83331, 0x71572c6d, 0xf3e4deb6,
0xcc266b9b, 0xe78e794b, 0x91b3379b, 0xbb230990,
0xdc64a31d, 0x95f3bcee, 0xf94fe453, 0x130804a0,
0x95b0c8b6, 0xa45e5d04, 0xf0a45550, 0xa272317e.

That's the end of one round.

In the second round, Salsa20 performs exactly the same modifications, with the same rotation counts, again starting with the below-diagonal words and finishing with the diagonal words, and finally transposing the array:

0xba2409b1, 0x1b7cce6a, 0x29115dcf, 0x5037e027,
0x37b75378, 0x348d94c8, 0x3ea582b3, 0xc3a9a148,
0x825bfcb9, 0x226ae9eb, 0x63dd7748, 0x7129a215,
0x4effd1ec, 0x5f25dc72, 0xa6c3d164, 0x152a26d8.

That's the end of two rounds. Note that implementors can eliminate the transposes and perform the second round on rows instead of columns.

Salsa20/r continues for a total of r rounds, modifying each word r times. For example, Salsa20/20 produces the following array:

0x58318d3e, 0x0292df4f, 0xa28d8215, 0xa1aca723,
0x697a34c7, 0xf2f00ba8, 0x63e9b0a1, 0x27250e3a,
0xb1c7f1f3, 0x62066edc, 0x66d3ccf1, 0xb0365cf3,
0x091ad09e, 0x64f0c40f, 0xd60d95ea, 0x00be78c9.

After these r rounds, Salsa20 adds the final 4×4 array to the original array to obtain its 64-byte output block. For example, here is the 64-byte output block for Salsa20/20:

0xb9a205a3, 0x0695e150, 0xaa94881a, 0xadb7b12c,
0x798942d4, 0x26107016, 0x64edb1a4, 0x2d27173f,
0xb1c7f1fa, 0x62066edc, 0xe035fa23, 0xc4496f04,
0x2131e6b3, 0x810bde28, 0xf62cb407, 0x6bdede3d.

4.2 Should key words and nonce words be separated?

Salsa20 puts its key k and its nonce/counter n into a single array. It uses the k words to modify the k words, the k words to modify the n words, the n words to modify the n words, and the n words to modify the k words. After a few rounds there is no reasonable distinction between the k parts of the array and the n parts of the array. Both the k words and the n words are used as output. The final addition prevents the cryptanalyst from inverting the computation.

For comparison, a "block cipher" uses the k words to modify the k words, the k words to modify the n words, and the n words to modify the n words; but it *never* uses the n words to modify the k words. The k words are kept separate from the n words through the entire computation. Only the n words are used as output. The omission of k prevents the cryptanalyst from inverting the computation.

The basic argument for a block cipher—for keeping the k words independent of the n words—is that, for fixed k, it is easy to make a block cipher be an invertible function of n. But this feature seems to be of purely historical interest. Invertible stream generation is certainly not necessary for encryption.

The basic disadvantage of a block cipher is that the k words consume valuable communication resources. A 64-byte block cipher with a 32-byte key would need to repeatedly sweep through 96 bytes of memory (plus a few bytes of temporary storage) for its 64 bytes of output; in contrast, Salsa20 repeatedly sweeps through just 64 bytes of memory (plus a few bytes of temporary storage) for its 64 bytes of output.

I also see each use of a k word as a missed opportunity to spread changes through the n words. The time wasted is not very large—in AES, for example, 80% of the table lookups and most of the xor inputs are n-dependent—and can be reduced by precomputation in contexts where the cost of memory is unnoticeable; but dropping the barrier between k and n achieves the same conjectured security level at higher speed.

4.3 Should there be more code?

Salsa20 can be implemented as a loop of identical rounds, where each round modifies each word once and then transposes the result. Or it can be implemented

as a loop of identical double-rounds, where each double-round modifies each word twice, without any transposition. Either way, the Salsa20 code is very short.

Some ciphers have more code: e.g., using different structures for the first and last rounds, or even using different code in every round. MARS, published by Burwick et al. in [10], has about one third of its operations in initial and final rounds that look quite different from the remaining rounds.

The basic argument for using two different kinds of rounds is the idea that attacks will have some extra difficulty passing through the switch from one kind to another. This extra difficulty would allow the cipher to reach the same security level with fewer rounds.

The basic counterargument is that extra code is expensive in many contexts. It increases pressure on a CPU's L1 cache, for example, and it increases the minimum size of a hardware implementation.

Even if larger code were free, I wouldn't feel comfortable reducing the number of rounds. The cryptanalytic literature contains a huge number of examples of how extra rounds increase security; it's much less clear how much benefit is obtained from switching round types.

4.4 Should there be faster diffusion among words?

During the first round of Salsa20, there is no communication between words in different columns; each column has its own chain of 12 operations modifying the words in that column. During the second round, there is no communication between words that were in different rows; each (transposed) row has its own chain of 12 operations modifying the words in that row. Et cetera.

There are pairs (i, j) such that a change in word i has no opportunity to affect word j until the third round. A different communication structure would allow much faster diffusion of changes through all 16 words. On the other hand, it doesn't appear to be possible to achieve much faster diffusion of changes through all 512 *bits*.

The current communication structure has speed benefits on CPUs that do not have many fast registers. For example, my software for the Pentium III relies on the ability to operate locally within 4 words for a little while.

4.5 Should there be modifications other than xor-a-rotated-sum?

There are many plausible ways to modify each word in a column using other words in the same column. I settled on "xor a rotated sum" as bouncing back and forth between incompatible structures on the critical path. I chose "xor a rotated sum" over "add a rotated xor" for simple performance reasons: the x86 architecture has a three-operand addition (LEA) but not a three-operand xor.

4.6 Should there be other rotation distances?

I chose the Salsa20 rotation distances $7, 11, 13, 18$ as doing a good job of spreading every low-weight change across bit positions within a few rounds. The exact choice of distances doesn't seem very important.

My software uses SIMD vector operations for the Pentium 4, the Core 2, et al. These operations rely on the fact that each column uses the same sequence of distances.

5 Cryptanalysis

This section briefly reviews the history of third-party cryptanalysis of Salsa20.

2005.05 [6]: I presented Salsa20 at the ECRYPT Symmetric-Key Encryption Workshop in Aarhus. I offered a $1000 prize for the most interesting Salsa20 cryptanalysis made public that year.

2005.10 [11]: Crowley posted a 2^{165}-operation attack on Salsa20/5. Crowley received the $1000 prize and presented his attack at the 2006.02 ECRYPT State of the Art of Stream Ciphers workshop in Leuven. The attack works forwards from a small known input difference to a biased bit 3 rounds later, and works 2 rounds backwards from an output after guessing 160 relevant key bits.

2006.12 [14]: Fischer, Meier, Berbain, Biasse, and Robshaw reported a 2^{177}-operation attack on Salsa20/6 (and a much faster attack on Salsa20/5, clearly breaking Salsa20/5) at the Indocrypt conference in Calcutta. The attack works forwards from a small known input difference to a biased bit 4 rounds later, and works 2 rounds backwards from an output after guessing 160 relevant key bits.

2007.01 [22]: Tsunoo, Saito, Kubo, Suzaki, and Nakashima reported a 2^{184}-operation attack on Salsa20/7 (and a much faster attack on Salsa20/6, clearly breaking Salsa20/6) at the ECRYPT State of the Art of Stream Ciphers workshop in Bochum. The attack works forwards from a small known input difference to a biased bit 4 rounds later, and works 3 rounds backwards from an output after guessing 171 highly relevant key bits.

2007.12 [2]: Aumasson, Fischer, Khazaei, Meier, and Rechberger reported a 2^{249}-operation attack on Salsa20/8 and a 2^{153}-operation attack on Salsa20/7. The Salsa20/8 attack works forwards from a small known input difference to a biased bit 4 rounds later, and works 4 rounds backwards from an output after guessing 228 extremely relevant key bits.

References

1. — (no editor), *MPC7450 RISC microprocessor family reference manual*, Freescale Semiconductor, 2005. URL: `http://www.freescale.com/files/32bit/doc/ref_manual/MPC7450UM.pdf`. Citations in this document: §2.2.
2. Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, Christian Rechberger, *New features of Latin dances: analysis of Salsa, ChaCha, and Rumba* (2007). URL: `http://eprint.iacr.org/2007/472`. Citations in this document: §5.
3. Rana Barua, Tanja Lange (editors), *Progress in Cryptology—INDOCRYPT 2006, 7th International Conference on Cryptology in India, Kolkata, India, December 11–13, 2006, Proceedings*, Lecture Notes in Computer Science, 4329, Springer, 2006. ISBN 3–540–49767–6. See [14].

4. Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in [15] (2005), 32–49. URL: http://cr.yp.to/papers.html#poly1305. ID 0018d9551b5546d97c340e0dd8cb5750. Citations in this document: §3.3.
5. Daniel J. Bernstein, *Cache-timing attacks on AES* (2005). URL: http://cr.yp.to/papers.html#cachetiming. ID cd9faae9bd5308c440df50fc26a517b4. Citations in this document: §2.3.
6. Daniel J. Bernstein, *The Salsa20 stream cipher*, slides of talk at ECRYPT STVL Workshop on Symmetric Key Encryption (2005). URL: http://cr.yp.to/talks.html#2005.05.26. Citations in this document: §5.
7. Daniel J. Bernstein, *Understanding brute force*, in Workshop Record of ECRYPT STVL Workshop on Symmetric Key Encryption, eSTREAM report 2005/036 (2005). URL: http://cr.yp.to/papers.html#bruteforce. ID 73e92f5b71793b498288efe81fe55dee. Citations in this document: §3.6.
8. Daniel J. Bernstein, *Cycle counts for authenticated encryption*, in Workshop Record of SASC 2007: The State of the Art of Stream Ciphers, eSTREAM report 2007/015 (2007). URL: http://cr.yp.to/papers.html#aecycles. ID be6b4df07eb1ae67aba9338991b78388. Citations in this document: §3.3.
9. Daniel J. Bernstein, *Polynomial evaluation and message authentication* (2007). URL: http://cr.yp.to/papers.html#pema. ID b1ef3f2d385a926123e1517392e20f8c. Citations in this document: §3.3.
10. Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, Nevenko Zunic, *MARS: a candidate cipher for AES* (1999). URL: www.research.ibm.com/security/mars.pdf. Citations in this document: §4.3.
11. Paul Crowley, *Truncated differential cryptanalysis of five rounds of Salsa20*, in Workshop Record of SASC 2006: Stream Ciphers Revisted, eSTREAM technical report 2005/073 (2005). URL: http://www.ecrypt.eu.org/stream/papers.html. Citations in this document: §5.
12. Donald W. Davies, *Advances in cryptology—EUROCRYPT '91: proceedings of the workshop on the theory and application of cryptographic techniques held in Brighton, April 8–11, 1991*, Lecture Notes in Computer Science, 547, Springer-Verlag, Berlin, 1991. ISBN 3–540–54620–0. MR 94b:94003. See [17].
13. Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, Tadayoshi Kohno, *Helix: fast encryption and authentication in a single cryptographic primitive*, in [16] (2003), 330–346. URL: http://www.macfergus.com/helix/. Citations in this document: §3.3.
14. Simon Fischer, Willi Meier, Côme Berbain, Jean-François Biasse, Matthew J. B. Robshaw, *Non-randomness in eSTREAM candidates Salsa20 and TSC-4*, in [3] (2006), 2–16. Citations in this document: §5.
15. Henri Gilbert, Helena Handschuh (editors), *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, Lecture Notes in Computer Science, 3557, Springer, 2005. ISBN 3–540–26541–4. See [4].
16. Thomas Johansson (editor), *Fast software encryption: 10th international workshop, FSE 2003, Lund, Sweden, February 24–26, 2003, revised papers*, Lecture Notes in Computer Science, 2887, Springer-Verlag, Berlin, 2003. ISBN 3–540–20449–0. See [13].
17. Xuejia Lai, James L. Massey, Sean Murphy, *Markov ciphers and differential cryptanalysis*, in [12] (1991), 17–38. Citations in this document: §2.3.

18. Mitsuru Matsui, Junko Nakajima, *On the power of bitslice implementation on Intel Core2 Processor*, in [20] (2007), 121–134. Citations in this document: §1.
19. James Nechvatal, Elaine Barker, Lawrence Bassham, William Burr, Morris Dworkin, James Foti, Edward Roback, *Report on the development of the Advanced Encryption Standard (AES)*, Journal of Research of the National Institute of Standards and Technology **106** (2001). URL: `http://nvl.nist.gov/pub/nistpubs/jres/106/3/cnt106-3.htm`. Citations in this document: §2.3.
20. Pascal Paillier, Ingrid Verbauwhede (editors), *Cryptographic Hardware and Embedded Systems: CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, Lecture Notes in Computer Science, 4727, Springer, 2007. ISBN 978–3–540–74734–5. See [18].
21. Bart Preneel (editor), *Fast software encryption: second international workshop, Leuven, Belgium, 14–16 December 1994, proceedings*, Lecture Notes in Computer Science, 1008, Springer-Verlag, Berlin, 1995. ISBN 3–540–60590–8. See [23].
22. Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, Hiroki Nakashima, *Differential cryptanalysis of Salsa20/8*, in Workshop Record of SASC 2007: The State of the Art of Stream Ciphers, eSTREAM report 2007/010 (2007). URL: `http://www.ecrypt.eu.org/stream/papers.html`. Citations in this document: §5.
23. David J. Wheeler, Roger M. Needham, *TEA, a tiny encryption algorithm*, in [21] (1995), 363–366. Citations in this document: §2.3.