Salsa20 design

Daniel J. Bernstein *

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

1 Introduction

Fifteen years ago, the United States government was trying to stop publication of new cryptographic ideas—but it had made an exception for cryptographic hash functions, such as Ralph Merkle's new Snefru.

This struck me as silly. I introduced Snuffle to point out that one can easily use a strong cryptographic hash function to efficiently encrypt data.

Snuffle 2005, formally designated the "Salsa20 encryption function," is the latest expression of my thoughts along these lines. It uses a strong cryptographic hash function, namely the "Salsa20 hash function," to efficiently encrypt data.

This approach raises two obvious questions. First, why did I choose this particular hash function? Second, now that the United States government seems to have abandoned its asinine policies, why am I continuing to use a hash function to encrypt data?

This document discusses the Salsa20 design, explaining why this particular encryption function is better than other encryption functions. Section 2 discusses the choice of low-level operations. Section 3 discusses the choice of high-level encryption structure. Section 4 discusses the combination of low-level operations into a hash function that fits the high-level encryption structure.

2 Operations

Any reasonable selection of operations can easily simulate any circuit, and is therefore capable of reaching the same security level as any other selection of operations—but perhaps not at the same speed.

I generally opted for a long chain of simple operations, rather than a shorter chain of complicated operations. Specifically, the Salsa20 hash function is a long chain of three simple operations: 32-bit addition (breaking linearity over $\mathbf{Z}/2$), 32-bit xor (breaking linearity over $\mathbf{Z}/2^{32}$), and constant-distance 32-bit rotation (diffusing changes from high bits to low bits).

The argument for complicated operations is that they provide a remarkable amount of mixing at reasonable speed on many CPUs, and thus achieve any

* The author was supported by the National Science Foundation under grant CCR–9983950, and by the Alfred P. Sloan Foundation. Date of this document: 2005.04.27.

desired security level more quickly than simple operations on those CPUs. The counterargument is that the potential speedup is fairly small, and is accompanied by huge slowdowns on other CPUs. The simple operations are *consistently* fast.

This section discusses the choice of operations in more detail.

Should there be integer multiplications?

Many popular CPUs can quickly compute $xy \bmod 2^{32}$, 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 typical CPUs, and many more cycles on some 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 variable-distance ("data-dependent") rotations, and to 64-bit additions.

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]. It is possible to eliminate the timing leak on this CPU 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.

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 [9, 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 [2] 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 [11], 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 [8], is even older and almost as simple: it is a long chain of 16-bit additions, 16-bit xors, and multiplications modulo $2^{16} + 1$.

Should there be fewer rotations?

Rotations account for about one third of the integer operations in Salsa20, and more on the UltraSPARC. Replacing some of the rotations with a comparable number of additions might achieve comparable diffusion in fewer rounds.

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. Yes, the UltraSPARC has to simulate rotations with shifts, but it can still xor a rotated quantity just as quickly as performing two xors of shifted quantities. Meanwhile, on many other CPUs, rotation saves time.

3 Encryption

Salsa20 is, at first glance, a traditional stream cipher; at second glance, a hash function in counter mode. This section discusses these design choices.

A continuing theme in this discussion is that, conjecturally, we gain security by reducing the amount of communication from the cryptanalyst to the internal cipher state. For example, a typical differential attack on a block cipher becomes impossible if most bits of the input block are always set to 0. As another example, Wang's recent construction of two-block MD5 collisions relied crucially on the cryptanalyst's ability to control many bits of MD5 input.

Extra communication is not a disaster: we can, conjecturally, gain the same security by adding extra cipher rounds. But extra rounds take time.

Should encryption and decryption be different?

The most common model of a stream cipher is that ciphertext is plaintext xor stream. Each byte of the stream is determined by its position, the nonce (the unique message number), the key, and the previous bytes of plaintext—equivalently, the previous bytes 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 a Wegman-Carter MAC) 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.

Should the stream be independent of 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.[1] 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 the model: the stream depends on the plaintext. An example is Helix, published by Ferguson, Whiting, Schneier, Kelsey, Lucks, and Kohno in [6].

The basic argument for incorporating plaintext into the stream (specifically, incorporating plaintext bytes into subsequent bytes of the stream) is that this allows message authentication "for free." After encrypting the plaintext, one can

[1] Ciphers with these two properties are often called "additive stream ciphers." This terminology is somewhat misleading. "Additive" is a sensible name for the first property, namely ciphertext being plaintext xor stream; but it doesn't suggest the second property, namely stream being independent of plaintext.

generate a constant number of additional stream bytes and output them 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 Wegman-Carter MACs—in particular, Poly1305, defined in [3]—take only a few cycles per byte. Even if this exceeds the cost of "free" authentication for *legitimate* packets, 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 Wegman-Carter MAC is capable of handling a substantially larger flood than a "free" authenticator.

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.

Many 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.

The counterargument is that a larger state loses time in some contexts. Reuse forces serialization: one cannot take advantage of extra hardware to reduce the latency of encrypting or decrypting long messages. Furthermore, large states reduce the number of messages that can be processed simultaneously on limited hardware.

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.

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, 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.

4 Hashing

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

The Salsa20 hash function combines the tools as follows. It starts with 16 words: the 32-byte key, the 8-byte nonce, the 8-byte block counter, and 16 bytes of constants. It then performs 320 invertible word modifications in a particular pattern, where each modification xors into one word a rotated sum of two other words. Finally, it adds the resulting 16 words to the original 16 words.

This section discusses several choices made in this structure.

Should the hash use a block cipher?

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. Invertibility 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.

Should there be more code?

The Salsa20 code is very short: it is a loop of identical double-rounds, where each double-round modifies each word twice. It can be made even shorter—a loop of identical rounds, where each round modifies each word once—at the expense of a simple transposition after each round.

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 [4], 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.

None of these comments are meant to discourage research into MARS-style ciphers. Perhaps it really is possible to obtain higher speed without sacrificing confidence.

Should there be faster diffusion among words?

Salsa20 views its 16 words as a 4×4 array. During the first round, there is no communication between columns; each column has its own chain of 12 serial operations modifying the words in that column. During the second round, there is no communication between rows; each row has its own chain of 12 serial 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. For example, my `salsa20_word_pii` software, which has the current speed record for the Pentium III, relies on the ability to operate locally within 4 words for a little while.

Should there be modifications other than add-rotate-xor?

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.

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 `salsa20_word_p4` software relies on the fact that each quarter-round uses the same sequence of distances.

Should there be fewer rounds?

I'm comfortable with the 20 rounds of Salsa20 as being far beyond what I'm able to break. Perhaps it will turn out that, after more extensive attempts at cryptanalysis, the community is comfortable with a smaller number of rounds; I can imagine using a smaller number of rounds for the sake of speed. On the other hand, Salsa20 will still have its place as a conservative design that is fast enough for practically all applications.

Presumably 16-byte keys can get away with fewer rounds than 32-byte keys. But this type of variability creates two real-world problems: first, it complicates hardware implementations; second, it seems to tempt users to reduce key sizes even in situations where the cost savings is insignificant.

As above, none of these comments are meant to discourage research into higher-speed stream ciphers. Perhaps it is possible to obtain higher speed without sacrificing confidence.

References

1. —, *MPC7450 RISC microprocessor family reference manual*, Freescale Semiconductor, 2005. URL: `http://www.freescale.com/files/32bit/doc/ref_manual/MPC7450UM.pdf`.
2. Daniel J. Bernstein, *Cache-timing attacks on AES* (2005). URL: `http://cr.yp.to/papers.html#cachetiming`. ID `cd9faae9bd5308c440df50fc26a517b4`.
3. Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in Proceedings of Fast Software Encryption 2005, to appear (2005). URL: `http://cr.yp.to/papers.html#poly1305`. ID `0018d9551b5546d97c340e0dd8cb5750`.

4. 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`.
5. 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.
6. Niels Ferguson, Doug Whiting, Bruce Schneier, John Kelsey, Stefan Lucks, Tadayoshi Kohno, *Helix: fast encryption and authentication in a single cryptographic primitive*, in [7] (2003), 330–346. URL: `http://www.macfergus.com/helix/`.
7. 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.
8. Xuejia Lai, James L. Massey, Sean Murphy, *Markov ciphers and differential cryptanalysis*, in [5] (1991), 17–38.
9. 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`.
10. 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.
11. David J. Wheeler, Roger M. Needham, *TEA, a tiny encryption algorithm*, in [10] (1995), 363–366.