Salsa20 security

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

If the Salsa20 key k is a uniform random sequence of bytes, and the same nonce is never used for two different messages, then the Salsa20 encryption function is conjectured to produce ciphertexts that are indistinguishable from perfect ciphertexts, i.e., uniform random strings independent of the plaintexts.

At a lower level, the random function $n \mapsto \text{Salsa20}_k(n)$ from $\{0,1,\ldots,255\}^{16}$ to $\{0,1,\ldots,255\}^{64}$ is conjectured to be indistinguishable from uniform. This conjecture implies the first conjecture.

The word "indistinguishable" is implicitly parametrized by limits on cost. Section 2 makes those limits explicit, and in particular discusses the cost of a brute-force attack.

The remaining sections explain why these conjectures are plausible, i.e., why Salsa20 is difficult to break. The Salsa20 design is quite conservative, allowing more confidence in these conjectures than in the analogous conjectures for some other functions.

Side-channel attacks

Natural Salsa20 implementations take constant time on a huge variety of CPUs; here **constant** means input-independent. There is no incentive for the authors of Salsa20 software to use variable-time operations such as S-box lookups. Timing attacks against Salsa20 are therefore just as difficult as pure cryptanalysis of the Salsa20 outputs. The operations in Salsa20 are also among the easiest to protect against power attacks and other side-channel attacks.

2 The cost of an attack

Assume that the target's Salsa20 key k is a uniform random 32-byte sequence. How do we distinguish the Salsa20 ciphertexts from perfect ciphertexts?

The obvious attack is brute force. Consider a gigantic parallel machine with 2^{64} independent key-searching units, given a pair $(n, \text{Salsa20}_k(n))$ as input. One

* 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.

unit searches through 2^{192} keys in the time taken for 2^{192} Salsa20 hash-function evaluations; in the same amount of time, the entire machine operating in parallel searches through all 2^{256} keys, and is guaranteed to find the target's key. The machine has some chance of success after fewer computations: after the time for 2^{128} Salsa20 computations, it has a 2^{-64} chance of finding the target's key. See [1] for a much more detailed discussion of the power of a parallel brute-force search, including a search for multiple keys simultaneously.

The Salsa20 security conjecture is that one cannot simultaneously achieve a substantially better price, performance, and chance of success: there is no machine that

- costs substantially less than 2^{64} key-searching units,
- takes time substantially less than 2^{128} Salsa20 hash-function computations, and
- has chance substantially above 2^{-64} of distinguishing $n \mapsto \text{Salsa20}_k(n)$ from a uniform random function.

The word "substantially" makes room for minor speedups. For example, the following tradeoff often improves price-performance ratio: a key-search unit can perform a partial hash-function evaluation, saving time at the beginning and end of the computation, by taking advantage of the fact that a single-bit key change does not instantaneously affect all 512 bits inside Salsa20. This standard observation is consistent with the Salsa20 security conjecture.

The word "distinguishing" is defined in the standard way: the machine's chance of success is defined as the distance between its probability of saying "yes" given an oracle for $n \mapsto \text{Salsa20}_k(n)$ and its probability of saying "yes" given an oracle for a uniform random function. Note that attacks are allowed to perform many oracle queries; a brute-force attack works with just one oracle query, but additional queries are conjectured to provide no help to the attacker. Salsa20 users are allowed to continue using a single key for as many nonces and blocks as they want.

Half-size keys

The security conjecture for 16-byte Salsa20 keys chops each exponent in half: there is no machine that costs substantially less than 2^{32} key-searching units, takes time substantially less than 2^{64} Salsa20 hash-function computations, and has chance substantially above 2^{-32} of distinguishing $n \mapsto \text{Salsa20}_k(n)$ from a uniform random function.

It's not clear whether this conjectured 128-bit security level is adequate for applications. A brute-force search through 2^{96} keys would be extremely expensive but is not inconceivable, and a success probability of 2^{-32} is not negligible. Consequently, I recommend switching to 256-bit keys. This recommendation has nothing to do with the details of Salsa20; it is a general comment regarding the feasibility of searching through a noticeable fraction of a set of 2^{128} elements.

Uniform random function or uniform random injective function

The internal structure of Salsa20_k implies that, for each k, the function $n \mapsto \text{Salsa20}_k(n) - (0, n, 0)$ is injective. However, because the output has 64 bytes, this injectivity doesn't help an attacker distinguish Salsa20_k from a uniform random 16-byte-to-64-byte function, even if the attacker inspects all 2^{128} outputs. A uniform random 16-byte-to-64-byte function has collision probability only about 2^{-256}.

Salsa20 is different from typical block ciphers in this respect. A 16-byte block cipher is distinguishable from a uniform random 16-byte-to-16-byte function. A 16-byte block cipher is designed to be indistinguishable from a uniform random *permutation*, but this is not the property that most applications need.

3 A Salsa20 example

Consider computing the first block of the Salsa20 stream with nonce 0 and key $(1, 2, 3, \ldots, 32)$. The Salsa20 hash function starts with the following 16 4-byte words, viewed as a 4×4 array:

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

The four words `0x61707865`, `0x3320646e`, `0x79622d32`, `0x6b206574` along the diagonal are Salsa20 constants used for every block, every nonce, and every 32-byte key. The all-zero words in the second row are the nonce. The all-zero words in the third row are the block counter. The remaining words are the key.

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`, `0x00000000`, `0x00000000`,
`0x00000000`, `0x91b3379b`, `0x79622d32`, `0x14131211`,
`0x18171615`, `0x1c1b1a19`, `0x91098721`, `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`, `0xdf6fa011`, `0x95b0c8b6`,
`0xd3c83331`, `0x3320646e`, `0x00000000`, `0xa25c5401`,
`0x71572c6a`, `0x91b3379b`, `0x79622d32`, `0x14131211`,
`0x18171615`, `0xbb230990`, `0x91098721`, `0x6b206574`.

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

0x61707865, 0xcc266b9b, 0xdf6fa011, 0x95b0c8b6,
0xd3c83331, 0x3320646e, 0x24e64e0f, 0xa25c5401,
0x71572c6a, 0x91b3379b, 0x79622d32, 0xb785f510,
0xf3e47eb6, 0xbb230990, 0x91098721, 0x6b206574.

Salsa20 finally modifies the diagonal entries, this time rotating left by 18 bits:

0xcdf1ec8b, 0xcc266b9b, 0xdf6fa011, 0x95b0c8b6,
0xd3c83331, 0xe78e794b, 0x24e64e0f, 0xa25c5401,
0x71572c6a, 0x91b3379b, 0xc1e23c65, 0xb785f510,
0xf3e47eb6, 0xbb230990, 0x91098721, 0x4f6502fd.

Salsa20 now modifies the rows in the same way, with the same rotation counts, again finishing with the diagonal entries:

0x11714935, 0x1d7ccb2a, 0x0200cbc7, 0x276eeb59,
0x28201ed6, 0xd09a67f5, 0x8fb07052, 0xdf8f6eef,
0x35341014, 0x111a81d5, 0x8647255f, 0x7d3ff539,
0xc4a171c6, 0xb7ca8fb8, 0x112648ac, 0x2ef6213e.

That's the end of two rounds. Salsa20 continues for a total of 20 rounds, modifying each word 20 times, producing the following array:

0xaa2db012, 0xd6ed6aa1, 0x53690720, 0xcfb88afc,
0x69a8a1d4, 0xee2f46d2, 0x77a47a7d, 0x369cd56b,
0xe83a3e0e, 0x6320d74c, 0xc5876267, 0xb3d8f55b,
0x16fb6058, 0x555ea6b2, 0xb33ae6fb, 0x10a7b0f9.

After these 20 rounds, Salsa20 adds the final 4×4 array to the original array to obtain its 64-byte output block:

0x0b9e2877, 0xdaf06ca2, 0x5b700d25, 0xdbc39505,
0x79b7afe1, 0x214fab40, 0x77a47a7d, 0x369cd56b,
0xe83a3e0e, 0x6320d74c, 0x3ee98f99, 0xc7ec076c,
0x2f12766d, 0x7179c0cb, 0xd35a0518, 0x7bc8166d.

4 Notes on the diagonal constants

Each Salsa20 column round affects each column in the same way starting from the diagonal. Each Salsa20 row round affects each row in the same way starting from the diagonal. Consequently, shifting the entire Salsa20 hash-function input array along the diagonal has exactly the same effect on the output.

The Salsa20 expansion function eliminates this shift structure by limiting the attacker's control over the hash-function input. In particular, the input diagonal is always `0x61707865`, `0x3320646e`, `0x79622d32`, `0x6b206574`, which is different from all its nontrivial shifts. In other words, two distinct arrays with this diagonal are always in distinct orbits under the shift group.

Similarly, the Salsa20 hash-function operations are *almost* compatible with rotation of each input word by, say, 10 bits. Rotation changes the effect of carries that cross the rotation boundary, but it is consistent with all other carries, and with the Salsa20 operations other than addition.

The Salsa20 expansion function also eliminates this rotation structure. The input diagonal is different from all its nontrivial shifts *and* all its nontrivial rotations *and* all nontrivial shifts of its nontrivial rotations. In other words, two distinct arrays with this diagonal are always in distinct orbits under the shift/rotate group.

5 An example of diffusion

Consider computing the *second* block of the Salsa20 stream with nonce 0 and key $(1, 2, 3, \ldots, 32)$. Rather than displaying the arrays produced by the second-block computation, this section displays the xor between those arrays and the corresponding first-block arrays, to emphasize the "active" bits—the bits where the computations differ.

The Salsa20 hash function starts with a 4×4 input array whose only difference from the first block is the different block counter, as shown by the following xor:

`0x00000000`, `0x00000000`, `0x00000000`, `0x00000000`,
`0x00000000`, `0x00000000`, `0x00000000`, `0x00000000`,
`0x00000001`, `0x00000000`, `0x00000000`, `0x00000000`,
`0x00000000`, `0x00000000`, `0x00000000`, `0x00000000`.

By the end of the first round, the difference has propagated to two other entries in the same column:

`0x80040001`, `0x00000000`, `0x00000000`, `0x00000000`,
`0x00000000`, `0x00000000`, `0x00000000`, `0x00000000`,
`0x00000001`, `0x00000000`, `0x00000000`, `0x00000000`,
`0x0000e000`, `0x00000000`, `0x00000000`, `0x00000000`.

At this point there are still just a few active bits. The difference depends on a few carries but is still highly predictable.

The second round then propagates the difference across columns:

`0xedc5e0a9`, `0x020000c0`, `0x381f830c`, `0x304888dc`,
`0x00000000`, `0x00000000`, `0x00000000`, `0x00000000`,
`0x00000001`, `0x00006000`, `0x800c0001`, `0x00000000`,
`0x0000e000`, `0x01c00000`, `0x040000d8`, `0x01200f00`.

By the end of the third round, every word has been affected:

0x39545d5e, 0x0cc160d8, 0x301fb030, 0xa05208dc,
0xa240cc8b, 0x24e0120c, 0x2a030dc7, 0xabeeb94e,
0x39ea409b, 0x0000000f, 0xcf3bb828, 0x1c205f6d,
0xc6612ba5, 0x01c06a00, 0x02000018, 0x6745c36b.

A substantial fraction of the bits are now active, although two words still have stretches of bits that were not (and were unlikely to be) active.

By the end of the fourth round, those last two stretches of inactivity have been eliminated:

0xf5eebb6a, 0x79a3e194, 0x52e3644f, 0x28fc33dd,
0xcbfe2c2e, 0xa0ce9f57, 0xfa23cf02, 0x2f549d35,
0x2b1af315, 0x7af4976b, 0xa100a15f, 0x86f420f1,
0x2900cc14, 0x8dcbf124, 0x90611242, 0x61fdabbe.

That's just 4 out of the 20 rounds in Salsa20. In every subsequent round, there are hundreds of active bits, for a total of more than 4000 active bits. Each of those 4000 active bits interacts with carries in a random-looking way, producing random-looking differences, not shown here.

6 Differential attacks

The idea of a differential attack is that some "small" differences in input states have a perceptible chance of producing "small" differences after the first step of the computation, the second step of the computation, etc.

Suppose that there is a "small" difference $n \oplus n'$ that has a perceptible chance of producing a "small" state difference after several rounds of Salsa20. In other words: suppose that, for all the pairs (n, n') having that difference, and for many keys k, there is a "small" difference after several rounds of Salsa20. Then it should be possible to find at least *one example* of a qualifying (n, n', k)—but I have found no such examples, and I see no reason to believe that any such examples exist.

Salsa20 is quite different in this respect from ciphers such as AES where the input size is as large as the state size. AES has 16-byte inputs, 16-byte outputs, and (at least) 16-byte keys; there are 2^{384} choices of (n, n', k), so presumably there are more than 2^{128} choices in which both of the 128-bit quantities $n' \oplus n$ and $\text{AES}_k(n') \oplus \text{AES}_k(n)$ are "small." Salsa20 has 16-byte inputs, 64-byte outputs, and 32-byte keys; there are 2^{512} choices of (n, n', k), so there is no a-priori reason to believe that any of the choices have the 128-bit quantity $n' \oplus n$ and the 512-bit quantity $\text{Salsa20}_k(n') \oplus \text{Salsa20}_k(n)$ both being "small."

I'm not saying that there are differential attacks on AES. I'm saying that differential attacks are inherently impossible against a broad class of functions;

that Salsa20 appears to be in that class; and that one cannot reasonably expect AES to be in that class.

The difficulty in constructing differentials should be clear to the reader from the example in Section 5. Even with control over k, it does not appear to be possible to keep a difference constrained within a small number of bits. The first two rounds

- convert a small change to x_6 into large changes in x_5, x_8, x_9, x_{10} and smaller changes in $x_0, x_2, x_3, x_4, x_6, x_7, x_{11}, x_{12}, x_{13}, x_{14}, x_{15}$;
- convert a small change to x_7 into medium-size changes in x_{13}, x_{14}, x_{15} and smaller changes in $x_4, x_5, x_7, x_8, x_9, x_{10}, x_{11}, x_{12}, x_{13}$;
- convert a small change to x_8 into medium-size changes in x_0, x_2, x_3 and smaller changes in $x_1, x_8, x_9, x_{10}, x_{12}, x_{13}, x_{14}, x_{15}$; and
- convert a small change to x_9 into large changes in x_0, x_4, x_5, x_7 and smaller changes in $x_1, x_2, x_3, x_6, x_8, x_9, x_{10}, x_{11}, x_{13}, x_{14}, x_{15}$.

Small combinations of these changes do not cancel many active bits. As far as I can tell, for every key, *every* input pair has more active bits after two rounds, and has thousands of active bits overall, as in Section 5. Those thousands of active bits have thousands of random-looking interactions with carries.

Other notions of small differences—using $-$ instead of $\oplus$, for example, or ignoring some bits—don't seem to help. Higher-order differential attacks don't seem to help. "Slide" differentials, in which one compares an input array to (e.g.) the 2-round array for another input, are hopeless for the same basic reason.

7 Algebraic attacks

The idea of an algebraic attack is to come up with a "small" set of equations satisfied by input states, output states, and (unknown) intermediate states, and then solve the equations—or, for a distinguisher, see whether the equations have a solution.

More generally, one might come up with equations that are *usually* satisfied, or *sometimes* satisfied, or satisfied noticeably more often for the cipher than for independent uniform random input and output bits. This broader perspective includes differential attacks, linear attacks, etc.

General-purpose equation-solving methods, notably Buchberger's algorithm for computing Groebner bases, are remarkably powerful. Clegg, Edmonds, and Impagliazzo in [2] proved—for a comparable problem, namely finding proofs in propositional logic—that a Groebner-basis computation can quickly solve any problem that can be quickly solved by various ad-hoc proof-finding techniques. Even better, the Groebner-basis computation can quickly solve other problems that cannot be quickly solved by the ad-hoc techniques. It would be interesting to see analogous theorems regarding various ad-hoc cryptanalytic techniques.

Fortunately, there does not seem to exist any "small" set of equations for the state bits in Salsa20. Each of the 320 32-bit additions in the Salsa20 computation

requires dozens of quadratic equations, producing a substantially larger system of equations than are required to describe, for example, the bits in AES.

Groebner-basis techniques for solving the AES-bit equations are, by the most optimistic estimates, slightly faster than brute-force search for a 256-bit key, but they use vastly more memory and thus have a much worse price-performance ratio. Algebraic attacks against Salsa20 appear to be even more difficult.

8 Other notions of security

Weak-key attacks

Suppose that there's a special set of 2^{200} keys that are surprisingly easy to recognize: they're found by a machine with comparable cost to 2^{56} key-searching units running for only as long as 2^{120} Salsa20 hash-function computations, rather than the obvious $2^{200}/2^{56} = 2^{144}$.

That machine, when applied to a uniform random Salsa20 key, would have success probability $2^{200}/2^{256} = 2^{-56}$. This machine—being 2^8 times faster, 2^8 times less expensive, and 2^8 times more likely to succeed than the machine described in Section 2—would violate the Salsa20 security conjecture.

In other words, there is no need to make a separate conjecture regarding weak keys.

This type of attack seems highly implausible for Salsa20. The Salsa20 key is mangled along with the input in an extremely complicated way. Any key differences rapidly spread through the entire Salsa20 state for the same reason that input differences do.

Equivalent-key attacks

Suppose that there's an easily searched set S of 2^{176} keys where each key $k \in S$ transforms inputs in the same way as $2^{24} - 1$ other keys.

A machine with comparable cost to 2^{56} key-searching units, running for only as long as 2^{120} Salsa20 hash-function computations, searching through that set of 2^{176} keys, would actually be a distinguisher for 2^{200} keys, and would have success probability $2^{200}/2^{256} = 2^{-56}$. This machine would violate the Salsa20 security conjecture.

In other words, there is no need to make a separate conjecture regarding equivalent keys.

This type of attack, like a weak-key attack, seems highly implausible for Salsa20.

Related-key attacks

The standard solutions to all the standard cryptographic problems—encryption, authentication, etc.—are protocols that do not allow related-key attacks on the underlying primitives. I see no evidence that we can save time by violating this condition. The reader might guess that Salsa20 is highly resistant to related-key attacks; but I simply don't care.

References

1. Daniel J. Bernstein, *Understanding brute force* (2005). URL: `http://cr.yp.to/papers.html#bruteforce`. ID `73e92f5b71793b498288efe81fe55dee`.
2. Matthew Clegg, Jeffery Edmonds, Russell Impagliazzo, *Using the Groebner basis algorithm to find proofs of unsatisfiability* (1996). URL: `http://www.cs.yorku.ca/~jeff/research/`.