Extending the Salsa20 nonce

Daniel J. Bernstein *

Department of Computer Science (MC 152)
The University of Illinois at Chicago
Chicago, IL 60607–7053
djb@cr.yp.to

Abstract. This paper introduces the XSalsa20 stream cipher. XSalsa20 is based upon the Salsa20 stream cipher but has a much longer nonce: 192 bits instead of 64 bits. XSalsa20 has exactly the same streaming speed as Salsa20, and its extra nonce-setup cost is slightly smaller than the cost of generating one block of Salsa20 output. This paper proves that XSalsa20 is secure if Salsa20 is secure: any fast attack on XSalsa20 using q queries and succeeding with probability p can be converted into a fast attack on Salsa20 succeeding with probability at least $p/(q+1)$.

Keywords. Extended nonces, stream ciphers, high security, high speed, security proofs

1 Introduction

eSTREAM, the ECRYPT Stream Cipher Project, called for submissions of stream ciphers in November 2004. It received more than 30 proposals from 97 cryptographers in 19 countries, and over the subsequent years collected a total of 200 papers. The "final eSTREAM portfolio," containing four software stream ciphers and four hardware stream ciphers, was announced in April 2008. The portfolio was revised in September 2008 to eliminate a hardware stream cipher, F-FCSR v2, that had been broken.

eSTREAM focused on 80-bit keys for hardware and 128-bit keys for software, but three of the final four software ciphers also have high-security variants:

- Sosemanuk, 256-bit key, 128-bit nonce, 4.22 cycles/byte, 3.43 cycles/byte;
- Salsa20, 256-bit key, 64-bit nonce, 3.91 cycles/byte, 3.86 cycles/byte; and
- HC-256, 256-bit key, 256-bit nonce, 34.91 cycles/byte, 4.15 cycles/byte.

The two speed reports for each cipher here are for, respectively, 1536-byte packets and long streams on a 2GHz Intel Core i7 920X (106e5) named `ellprecisio`. These are two of the benchmarks reported by the eBACS project [20]; see the eBACS web site for many other measurements.

* Permanent ID of this document: `c4b172305ff16e1429a48d9434d50e8a`. Date of this document: 2011.02.04. This work was supported by the National Science Foundation under grant 0716498.

Salsa20, my own eSTREAM submission, offers high speed for long and short packets, but has a potential drawback: its nonce is limited to just 64 bits. Is a 64-bit nonce long enough for high-security applications?

There is a standard argument that a 64-bit nonce *is* long enough. Nonce security does not mean unpredictability; it means uniqueness. Applications can generate a nonce as a monotonic timestamp or simply a counter $1, 2, 3, \ldots$.

There is also a standard counterargument. Counters might sound simple but are sometimes mismanaged by applications, destroying security. Rather than blaming the application for this failure, we can append random bits to the nonce, adding protection that is likely to succeed even if the counter fails.

Contents of this paper

This paper introduces a new family of stream ciphers, XSalsa20.

XSalsa20 is, at first glance, quite similar to Salsa20: it is built from exactly the same operations, has exactly the same protections against side-channel attacks, has exactly the same streaming speed, supports 256-bit keys, and allows reduced-round variants such as XSalsa20/12. Note that the speed reports above are for full-round Salsa20/20, not Salsa20/12.

The advantage of XSalsa20 over Salsa20 is a longer nonce: 192 bits rather than 64 bits. The disadvantage is that nonce setup is less efficient—but the extra cost here is comparable to, and in fact slightly smaller than, the cost of generating a single Salsa20 output block.

XSalsa20 might at first appear to be an ad-hoc design, following standard principles but potentially vulnerable to new attacks. On the contrary! This paper proves that any fast successful attack on XSalsa20 can be converted into a fast successful attack on Salsa20. Confidence in the security of Salsa20 therefore implies confidence in the security of XSalsa20.

This paper is not meant to take a position in the dispute regarding the necessity of longer nonces. This paper does not claim any benefits for XSalsa20 in an application that already works with Salsa20's 64-bit nonces. What this paper shows is that—in case an application *does* want longer nonces—the Salsa20 nonce can be safely extended at surprisingly low cost.

Related work

There are several theoretical papers showing various ways that a "fixed-input-length pseudorandom function" can be used to build a "longer-fixed-input-length pseudorandom function" (or a "variable-input-length pseudorandom function"). Often these papers are expressed as constructions of message-authentication codes, but the underlying proofs show that the constructions are secure ciphers, i.e., that they are indistinguishable from uniform random functions.

Consider, for example, the "triple-length CBC MAC," which maps a 384-bit nonce (n_1, n_2, n_3) to a 128-bit output $E_k(E_k(E_k(n_1) \oplus n_2) \oplus n_3)$. Here E is a cipher mapping 128-bit blocks to 128-bit blocks; for example, E could be AES,

using a 256-bit key k. Bellare, Kilian, and Rogaway proved in [10, Theorem 3.1] that this 384-bit-to-128-bit function is a secure cipher if E is a secure cipher. More precisely: The insecurity of this function, against an attacker who sees q function outputs, is at most the insecurity of E plus $27q^2/2^{129}$. For simpler proofs and improved bounds see [43], [35], [14], and [12]. For variants and generalizations see [13], [23], [34], [29], [32], and [36].

Unfortunately, these security proofs become meaningless as the number of queries approaches the square root of the number of inputs allowed by the original function: 2^{64} for AES, and even fewer for functions with smaller inputs. If $q = 2^{60}$—a huge but not inconceivable volume of data—then $27q^2/2^{129}$ is about 5%, an unacceptably large chance of success. The improved bounds are below 5% but are still unacceptably large. Telling cryptographic implementors that "the number of messages to be communicated in a session ... should not be allowed to approach $2^{n/2}$" (as in [21, page 20]) begs the question of what exactly the implementors are supposed to do when the users have more data to transmit—and the question of how users are supposed to gain confidence in the security of the resulting protocol. Switching session keys does not magically create immunity to cryptanalysis! As illustrated by the Albrecht–Paterson–Watson announcement [3] of a cryptographic flaw in `ssh`, one must analyze complete cryptographic protocols, not just pieces of those protocols.

There *are* some security proofs for constructions that can be viewed as switching session keys. Consider, for example, the "triple-length AES cascade," which maps a 384-bit nonce (n_1, n_2, n_3) to a 128-bit output $\mathrm{AES}_{\mathrm{AES}_{\mathrm{AES}_k(n_1)}(n_2)}(n_3)$. Bellare, Canetti, and Krawczyk showed in [8, Theorem 3.1] that this 384-bit-to-128-bit function is a secure cipher if AES is a secure cipher. More precisely: The insecurity of this function, against an attacker who sees q function outputs, is at most $3q$ times the insecurity of AES. This construction can be viewed as creating a first-level session key $k_1 = \mathrm{AES}_k(n_1)$ from the first part of the nonce, then creating a second-level session key $k_2 = \mathrm{AES}_{k_1}(n_2)$ from the second part of the nonce, then creating an output $\mathrm{AES}_{k_2}(n_3)$ from the third part of the nonce.

Unfortunately, the security level of this AES cascade is quantitatively unacceptable, even worse than the security level of CBC. Consider an attacker who collects 2^{43} cascade outputs $\mathrm{AES}_{\mathrm{AES}_{\mathrm{AES}_k(n)}(0)}(0)$ for various nonces of the form $(n, 0, 0)$. The attacker stores these outputs on a large machine consisting of 2^{43} tiny parallel search units. Each search unit cycles through 2^{43} possibilities for k_1, computes $\mathrm{AES}_{\mathrm{AES}_{k_1}(0)}(0)$, and compares the results to all of the collected outputs. (This can be done with negligible communication costs; see, e.g., [16].) The attacker then has a good chance of discovering an equation $k_1 = \mathrm{AES}_k(n)$, immediately revealing all of the cascade outputs for nonces of the form (n, n_2, n_3). This attack does not contradict the security guarantee in [8, Theorem 3.1]; recall that the insecurity of AES—which is approximately 2^{-42} against this attacker—is *multiplied* by the number of queries. The attack *does* disprove [7, "Theorem" 3.1], which omitted the factor q, as pointed out ten years later in [8].

The core problem in this AES cascade is the use of AES outputs—which are only 128 bits—as keys. High security demands larger keys. The cascade also has

a performance problem not present in CBC: the cascade requires an extra AES key setup for every new n_1 and for every new (n_1, n_2), whereas CBC continues using the same key. One could build a much more secure AES-based cascade by using, e.g., 256-bit keys of the form $(\mathrm{AES}_k(n, 0), \mathrm{AES}_k(n, 1))$, but this would create even larger speed problems. The literature does not seem to contain any serious attempts to build high-speed high-security cascades.

XSalsa20 can be viewed as a generalized cascade, taking advantage of the structure of Salsa20 to achieve high security at surprisingly high speed. The cascade has two levels, with exactly Salsa20 at the second level but with a slightly modified Salsa20 at the first level; the modification skips some of the operations in Salsa20 without sacrificing security. Relevant features of Salsa20 include free key setup, a 512-bit block size, randomly accessible output blocks, and a particularly simple key schedule.

2 Specification

This section defines the XSalsa20 family of stream ciphers. This section also defines "HSalsa20," an intermediate step towards XSalsa20. HSalsa20 is a helpful tool in the XSalsa20 security proof, can be used as a module in XSalsa20 implementations, and is potentially of independent interest. This section also discusses the relative speeds of Salsa20, HSalsa20, and XSalsa20.

Review of Salsa20

Salsa20/r produces a 512-bit output block starting from a 512-bit input block $(x_0, x_1, \ldots, x_{15})$ where

- $(x_0, x_5, x_{10}, x_{15})$ is the "Salsa20 constant"—the 128-bit string `0x61707865`, `0x3320646e`, `0x79622d32`, `0x6b206574`;
- $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14})$ is a 256-bit key,
- (x_6, x_7) is a 64-bit nonce, and
- (x_8, x_9) is a 64-bit block counter, the position of the 512-bit output block in the Salsa20 output stream.

Salsa20/r applies $r/2$ iterations of the "double-round" function defined in [15, Section 6], obtaining $(z_0, z_1, \ldots, z_{15}) = \mathrm{doubleround}^{r/2}(x_0, x_1, \ldots, x_{15})$. It then outputs $(x_0+z_0, x_1+z_1, \ldots, x_{15}+z_{15})$. Here $+$ is addition of 32-bit words modulo 2^{32}. All 32-bit words are viewed as strings in little-endian form.

Each round of Salsa20 uses 16 additions, 16 xors, and 16 rotations of 32-bit words; see [15, Sections 3–5]. Consequently Salsa20/r uses $16r + 16$ additions, $16r$ xors, and $16r$ rotations to generate an output block.

I originally recommended—and continue to recommend—Salsa20/20, with Salsa20/12 and Salsa20/8 as faster options for users who value speed more highly than confidence. Four attack papers by fourteen cryptanalysts ([25], [27], [42], and [5]) culminated in a 2^{184}-operation attack on Salsa20/7 and a 2^{251}-operation attack on Salsa20/8. The eSTREAM portfolio recommended Salsa20/12: "Eight

and twenty round versions were also considered during the eSTREAM process, but we feel that Salsa20/12 offers the best balance, combining a very nice performance profile with what appears to be a comfortable margin for security."

The subsequent paper [38] claimed to "show that Salsa20 does not have 256-bit security." I responded in [19] that "the best 'attack' in the paper receives ciphertexts from 2^{191} users and finds a 256-bit key after time 2^{192} on a machine of size roughly 2^{192}" and that this is "vastly more expensive than the standard brute-force attacks discussed in the original Salsa20 documentation." There was no further comment from the authors of [38]. In any event, the claims of [38] are orthogonal to the topic of this paper. The single-key security of Salsa20 implies the single-key security of XSalsa20, which in turn implies the multiple-key security of XSalsa20; these are not tight reductions, but tightness is unnecessary when one starts with a 256-bit cipher.

Definition of HSalsa20

HSalsa20/r produces a 256-bit output block starting from a 512-bit input block $(x_0, x_1, \ldots, x_{15})$ where

- $(x_0, x_5, x_{10}, x_{15})$ is the Salsa20 constant,
- $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14})$ is a 256-bit key, and
- (x_6, x_7, x_8, x_9) is a 128-bit nonce.

HSalsa20/r applies $r/2$ iterations of the same "double-round" function, obtaining $(z_0, z_1, \ldots, z_{15}) = \text{doubleround}^{r/2}(x_0, x_1, \ldots, x_{15})$. HSalsa20 then outputs the 256-bit block $(z_0, z_5, z_{10}, z_{15}, z_6, z_7, z_8, z_9)$. The indices $0, 5, 10, 15, 6, 7, 8, 9$ here were not chosen arbitrarily; the choice is important for the security proof later in this paper.

Observe that HSalsa20 skips the final 16 additions in Salsa20, leaving only 128 additions in HSalsa20/8, 192 additions in HSalsa20/12, etc. HSalsa20 also eliminates the final 16 loads of $(x_0, x_1, \ldots, x_{15})$, and 8 of the final 16 stores. The overall savings are platform-dependent.

Note that, in Salsa20, the loads and additions of the key words $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14})$ are critical for security, since the double-round function is trivially invertible. One might think that the remaining loads and additions could be skipped without sacrificing security, achieving almost half of the HSalsa20 savings. However, this change would interfere with the vector loads and vector additions used in many high-speed Salsa20 implementations.

Definition of XSalsa20

XSalsa20/r produces a 512-bit output block starting from a 512-bit input block $(x_0, x_1, \ldots, x_{15})$ where

- $(x_0, x_5, x_{10}, x_{15})$ is the Salsa20 constant,
- $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14})$ is a 256-bit key, and

- (x_6, x_7, x_8, x_9) is the first 128 bits of a 192-bit nonce.

XSalsa20/r computes $(z_0, z_1, \ldots, z_{15}) = \text{doubleround}^{r/2}(x_0, x_1, \ldots, x_{15})$. It then builds a new 512-bit input block $(x'_0, x'_1, \ldots, x'_{15})$ where

- $(x'_0, x'_5, x'_{10}, x'_{15})$ is the Salsa20 constant,
- $(x'_1, x'_2, x'_3, x'_4, x'_{11}, x'_{12}, x'_{13}, x'_{14}) = (z_0, z_5, z_{10}, z_{15}, z_6, z_7, z_8, z_9)$,
- (x'_6, x'_7) is the last 64 bits of the 192-bit nonce, and
- (x'_8, x'_9) is a 64-bit block counter.

XSalsa20 then computes $(z'_0, z'_1, \ldots, z'_{15}) = \text{doubleround}^{r/2}(x'_0, x'_1, \ldots, x'_{15})$ and outputs the 512-bit block $(x'_0 + z'_0, x'_1 + z'_1, \ldots, x'_{15} + z'_{15})$. Overall XSalsa20 has the same shape as Salsa20, except for the much longer nonce: it produces a 512-bit output block given a 256-bit key, a 192-bit nonce, and a 64-bit block counter.

In other words, XSalsa20/r is a two-level generalized cascade, using the output of HSalsa20/r as a key for Salsa20/r. From an implementor's perspective, one can generate a stream of output blocks with XSalsa20/r by computing a single HSalsa20/r output block and then tail-calling an existing function to generate a stream of output blocks with Salsa20/r. This straightforward implementation strategy immediately produces the same streaming speeds for XSalsa20/r that have already been achieved for Salsa20/r. See [18] and [20] for surveys of those speeds. The overhead for XSalsa20/r, compared to Salsa20/r, is the HSalsa20/r computation, which as discussed above is slightly faster than computing a single Salsa20/r output block.

3 Security proof

This section proves that HSalsa20 and XSalsa20 are secure if Salsa20 is secure. Specifically, this section proves a new security theorem for generalized cascades, improving both qualitatively and quantitatively upon [8]; applies the theorem to XSalsa20, showing that XSalsa20 is secure if Salsa20 and HSalsa20 are both secure; and, finally, proves that HSalsa20 is secure if Salsa20 is secure.

This paper uses the standard "PRF" notion of cipher security: a cipher is secure if the cipher outputs for a uniform random secret key are indistinguishable from independent uniform random strings. This notion is well known to be a suitable foundation for simple, efficient, state-of-the-art cryptographic protocols. One can build protocols that rely on other notions of cipher security, such as resistance to various types of "related-key attacks"; however, there does not appear to be any literature claiming advantages of those protocols, so this paper focuses on the standard security notion.

The theorems and proofs in this section rely on the standard language of probability theory. In particular, this section assumes that the reader is familiar with the mathematician's definition of a random variable: a random element of a measurable space M is a measurable function from a fixed probability space Pr (intuitively, the set of all possible universes) to M. See [17, Appendix A] for

a three-page discussion of the standard language and its benefits. Advocates of other languages, such as the language of "games," might view this section as a challenge: can those languages be used to express the same theorems and proofs without sacrificing generality and without sacrificing simplicity?

Generalized cascades

Theorem 3.1 generalizes the construction of XSalsa20 as follows:

- The set of 256-bit HSalsa20 keys is generalized to any finite set K_1.
- The set of 128-bit HSalsa20 inputs is generalized to any set I_1, not necessarily finite.
- The set of 256-bit HSalsa20 outputs—and of 256-bit Salsa20 keys—is generalized to any finite set K_2.
- HSalsa20 is generalized to any computable function $H : K_1 \times I_1 \to K_2$.
- The set of 128-bit Salsa20 inputs (each consisting of a 64-bit nonce and 64-bit block counter) is generalized to any set I_2, not necessarily finite.
- The set of 512-bit Salsa20 output blocks is generalized to any finite set L.
- Salsa20 is generalized to any computable function $S : K_2 \times I_2 \to L$.
- XSalsa20 is generalized to a function $X : K_1 \times I_1 \times I_2 \to L$, specifically the function $(k_1, i_1, i_2) \mapsto S(H(k_1, i_1), i_2)$.

Compared to this generalized cascade, the two-level cascade considered in [8] is the following special case: $I_1 = I_2 = \{0,1\}^b$; $K_1 = K_2 = L = \{0,1\}^k$; and $H = S$. The assumption $I_1 = I_2$ is compatible with XSalsa20, but the assumption $H = S$ does not allow the distinction between Salsa20 and HSalsa20, and the assumption $K_2 = L$ would force XSalsa20 to drop half of the output bits in each block.

[8] also considers three-level cascades, four-level cascades, etc. One can view an $(\ell + 1)$-level cascade as a generalized two-level cascade where the first level is an ℓ-level cascade. [8, Theorem 3.1], although described in [8] as "(almost) optimal," is quantitatively too weak to be used in this type of composition: it loses a factor of q at each level of the composition, where q is the number of attack queries. Theorem 3.1 below fixes this quantitative flaw.

Security proof for generalized cascades

Consider a generalized cascade $X = ((k_1, i_1, i_2) \mapsto S(H(k_1, i_1), i_2))$. The goal of an attack is to distinguish an oracle for $X(k_1)$, where k_1 is a uniform random element of K_1, from an oracle for a uniform random function from $I_1 \times I_2$ to L. Here $X(k_1)$ means $(i_1, i_2) \mapsto X(k_1, i_1, i_2)$; i.e., $(i_1, i_2) \mapsto S(H(k_1, i_1), i_2)$.

Starting from a fast successful attack against $X(k_1)$, the security proof constructs fast attacks against $H(k_1)$ and $S(k_2)$, where k_2 is a uniform random element of K_2, and shows that at least one of these attacks must be successful. Contrapositive: if H and S are both secure, then X must also be secure.

Specifically, given an algorithm A, define algorithms $A_0, A_1, A_2, \ldots$ as follows:

- The algorithm A_0, given an oracle $O : I_1 \to K_2$, runs the algorithm A with the oracle $(i_1, i_2) \mapsto S(O(i_1), i_2)$.
- For $j \geq 1$: The algorithm A_j, given an oracle $O : I_2 \to L$, generates (lazily) a uniform random function U from $I_1 \times I_2$ to L and, independently of U, a uniform random function V from I_1 to K_2. It runs the algorithm A with the following oracle: respond to (i_1, i_2) with $U(i_1, i_2)$ for the *first* $j - 1$ *distinct* query prefixes i_1 that appear, with $O(i_2)$ for the jth distinct query prefix, and with $S(V(i_1), i_2)$ for all other query prefixes.

Theorem 3.1 states that the A-distance from $X(k_1)$ to uniform—i.e., the gap $|\Pr[A(X(k_1)) = 1] - \Pr[A(U) = 1]|$—is at most the sum of the $A_0, A_1, \ldots, A_q$-distances from, respectively, $H(k_1), S(k_2), \ldots, S(k_2)$ to uniform.

Note that $A_0(H(k_1))$ is exactly the same computation as $A(X(k_1))$, so in particular it has the same speed and carries out the same number of oracle queries. The other algorithms A_j replace each $H(k_1, i_1)$ computation by generation of a uniform random $V(i_1)$; furthermore, as j increases, these algorithms progressively replace each $S(V(i_1), i_2)$ computation by generation of a uniform random $U(i_1, i_2)$.

Theorem 3.1. *Let K_1, I_1, K_2, I_2, L be sets, with K_1, K_2, L finite. Let H be a function from $K_1 \times I_1$ to K_2. Let S be a function from $K_2 \times I_2$ to L. Define X as the function $(k_1, i_1, i_2) \mapsto S(H(k_1, i_1), i_2)$ from $K_1 \times I_1 \times I_2$ to L. Let A be an algorithm that makes at most q oracle queries. Define $A_0, A_1, \ldots, A_q$ as above. Define δ_0 as the A_0-distance from $H(k_1)$ to uniform, where k_1 is a uniform random element of K_1. Define δ_j, for $j \in \{1, \ldots, q\}$, as the A_j-distance from $S(k_2)$ to uniform, where k_2 is a uniform random element of K_2. Then the A-distance from $X(k_1)$ to uniform, where k_1 is a uniform random element of K_1, is at most $\delta_0 + \delta_1 + \cdots + \delta_q$.*

As a corollary, if H has insecurity $\leq \epsilon$ against algorithms as fast as A_0, and S has insecurity $\leq \epsilon'$ against algorithms as fast as $A_1, \ldots, A_q$, then X has insecurity $\leq \epsilon + q\epsilon'$ against algorithms as fast as A. Note that this bound is affected less by the insecurity of H than by the insecurity of S. Designers might view this as a reason to choose H less conservatively than S, and theoreticians might try to prove security under weaker assumptions on H; on the other hand, it is not clear how much room is left to improve upon the speed of HSalsa20!

By induction an ℓ-level generalized cascade has insecurity $\leq \epsilon_1 + q\epsilon_2 + q\epsilon_3 + \cdots + q\epsilon_\ell$ if each level n has insecurity $\leq \epsilon_n$. In particular, an ℓ-level non-generalized cascade has insecurity $\leq (1 + (\ell - 1)q)\epsilon$ if each level has insecurity $\leq \epsilon$.

For comparison, [8, Theorem 3.1] proves a weaker bound of $\ell q \epsilon$ for an ℓ-level cascade. As mentioned above, this bound is not suitable for composition, so the proof in [8] has to directly handle all ℓ, rather than focusing on the simplest case $\ell = 2$.

Proof. Define oracles $O_{-1}, O_0, O_1, \ldots, O_q$ as follows:

- O_{-1} is $X(k_1)$.

- O_j, for $j \geq 0$, generates a uniform random function $U' : I_1 \times I_2 \to L$ and a uniform random function $V' : I_1 \to K_2$ independent of U'. It then responds to (i_1, i_2) with $U'(i_1, i_2)$ for the first j distinct query prefixes i_1 that appear, and with $S(V'(i_1), i_2)$ for other query prefixes.

By hypothesis A performs at most q oracle queries, so $A(O_q) = A(U')$. Hence the A-distance from $X(k_1)$ to uniform is exactly the A-distance from O_{-1} to O_q. This is, in turn, is at most the sum of the A-distances from O_{-1} to O_0, from O_0 to O_1, from O_1 to O_2, and so on through O_q. The rest of the proof will show that these distances are exactly $\delta_0, \delta_1, \delta_2, \ldots, \delta_q$.

The oracle O_{-1} is $(i_1, i_2) \mapsto S(H(k_1, i_1), i_2)$ so $A(O_{-1}) = A_0(H(k_1))$. Similarly O_0 is $(i_1, i_2) \mapsto S(V'(i_1), i_2)$ so $A(O_0) = A_0(V')$. Hence the A-distance from O_{-1} to O_0 is exactly the A_0-distance from $H(k_1)$ to V'; i.e., exactly δ_0.

Now fix $j \in \{1, 2, \ldots, q\}$. Let k_2 be a uniform random element of K_2. By construction $A_j(S(k_2))$ runs A with an oracle that responds to (i_1, i_2) with $U(i_1, i_2)$ for the first $j - 1$ distinct query prefixes i_1, with $S(k_2, i_2)$ for the jth distinct query prefix, and with $S(V(i_1), i_2)$ for all other query prefixes. Here U is a uniform random function from $I_1 \times I_2$ to L; V is a uniform random function from I_1 to K_2; and k_2, U, V are independent.

Define $V'(i_1)$ as k_2 if i_1 is the jth distinct query prefix that occurred in this run, and as $V(i_1)$ otherwise. Also define $U' = U$. Then V' is a uniform random function from I_1 to K_2 (since the jth distinct query prefix is independent of k_2); U' is a uniform random function from $I_1 \times I_2$ to L; and U', V' are independent. Furthermore, the oracle responses were exactly $U'(i_1, i_2)$ for the first $j-1$ distinct query prefixes i_1 and $S(V'(i_1), i_2)$ for other query prefixes; i.e., the responses from O_{j-1}. Consequently $A_j(S(k_2)) = A(O_{j-1})$.

Similarly, let W be a uniform random function from I_2 to L. By construction $A_j(W)$ runs A with an oracle that responds to (i_1, i_2) with $U(i_1, i_2)$ for the first $j - 1$ distinct query prefixes i_1, with $W(i_2)$ for the jth distinct query prefix, and with $S(V(i_1), i_2)$ for all other query prefixes. As before U is a uniform random function from $I_1 \times I_2$ to L; V is a uniform random function from I_1 to K_2; and U, V, W are independent.

Define $U'(i_1, i_2)$ as $W(i_2)$ if i_1 is the jth distinct query prefix that occurred in this run, and as $U(i_1, i_2)$ otherwise. Also define $V' = V$. Then U' is a uniform random function from $I_1 \times I_2$ to L; V' is a uniform random function from I_1 to K_2; and U', V' are independent. Furthermore, the oracle responses were exactly $U'(i_1, i_2)$ for the first j distinct query prefixes i_1 and $S(V'(i_1), i_2)$ for other query prefixes; i.e., the responses from O_j. Consequently $A_j(W) = A(O_j)$.

Conclusion: The A-distance from O_{j-1} to O_j is exactly the A_j-distance from $S(k_2)$ to W; i.e., exactly δ_j. □

Notes on low-memory attacks

Among cryptanalysts there is an increasingly popular movement away from the simple but physically unrealistic idea that computations can instantaneously access unlimited amounts of RAM. In more sophisticated models of computation,

the cost of the algorithm A_j defined above may be swamped by the time and space required to manage an array of U values, an array of V values, and an array of the first j distinct query prefixes.

A standard way to eliminate the space for U and V is to replace random-number generation by pseudorandom-number generation. If there exists *any* fast cipher with security level above the cost of A then one can safely use that cipher as a replacement for U and V, after generating a single uniform random cipher key.

Eliminating the space for the array of query prefixes takes more work. One possible approach is to map I_1 pseudorandomly to $\{1, 2, \ldots, q\}$, and compare query prefixes to j under this map (hoping that j is not produced by two different query prefixes), rather than attempting to keep track of the actual time of appearance of each prefix. Another possible approach is to map I_1 to $\{0, 1\}$ and recursively analyze the two halves of the query prefixes.

For simplicity this paper analyzes only A_j, rather than these potentially less expensive variants.

Security proof for XSalsa20

Fix $r \in \{2, 4, 6, \ldots\}$. Theorem 3.2 states that XSalsa20/r is secure if Salsa20/r and HSalsa20/r are secure.

Define $\mathrm{HSalsa}_k(i)$, where k is a 256-bit string and i is a 128-bit string, as the 256-bit HSalsa20/r output block for key k and nonce i. Recall that this output is $(z_0, z_5, z_{10}, z_{15}, z_6, z_7, z_8, z_9)$ where $(x_0, x_5, x_{10}, x_{15})$ is the Salsa20 constant, $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14})$ is the key k, (x_6, x_7, x_8, x_9) is the input i, and $(z_0, z_1, \ldots, z_{15}) = \mathrm{doubleround}^{r/2}(x_0, x_1, \ldots, x_{15})$.

Define $\mathrm{Salsa}_k(i)$, where k is a 256-bit string and i is a 128-bit string, as the 512-bit Salsa20/r output block for key k, nonce equal to the first half of i, and block counter equal to the second half of i. Recall that this output is $(x_0 + z_0, x_1 + z_1, \ldots, x_{15} + z_{15})$ where $(x_0, x_5, x_{10}, x_{15})$ is the Salsa20 constant, $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14})$ is the key k, (x_6, x_7, x_8, x_9) is the input i, and $(z_0, z_1, \ldots, z_{15}) = \mathrm{doubleround}^{r/2}(x_0, x_1, \ldots, x_{15})$.

Define $\mathrm{XSalsa}_k(i)$, where k is a 256-bit string and i is a 256-bit string, as the 512-bit XSalsa20/r output block for key k, nonce equal to the first 192 bits of i, and block counter equal to the last 64 bits of i. Recall that this output is $(x'_0 + z'_0, x'_1 + z'_1, \ldots, x'_{15} + z'_{15})$ where

- $(x_0, x_5, x_{10}, x_{15})$ is the Salsa20 constant,
- $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14})$ is the key k,
- (x_6, x_7, x_8, x_9) is the first 128 bits of the input i,
- $(z_0, z_1, \ldots, z_{15}) = \mathrm{doubleround}^{r/2}(x_0, x_1, \ldots, x_{15})$,
- $(x'_0, x'_5, x'_{10}, x'_{15})$ is the Salsa20 constant again,
- $(x'_1, x'_2, x'_3, x'_4, x'_{11}, x'_{12}, x'_{13}, x'_{14}) = (z_0, z_5, z_{10}, z_{15}, z_6, z_7, z_8, z_9)$,
- (x'_6, x'_7, x'_8, x'_9) is the last 128 bits of the input i, and
- $(z'_0, z'_1, \ldots, z'_{15}) = \mathrm{doubleround}^{r/2}(x'_0, x'_1, \ldots, x'_{15})$.

Consequently $\mathrm{XSalsa}_k(i) = \mathrm{Salsa}_{\mathrm{HSalsa}_k(i_1)}(i_2)$ where i_1 is the first half of the input i and i_2 is the second half of the input i.

Theorem 3.2. *Let A be an algorithm that makes at most q oracle queries. Define $A_0, A_1, \ldots, A_q$ as above, where $K_1 = K_2 = \{0,1\}^{256}$, $I_1 = I_2 = \{0,1\}^{128}$, and $L = \{0,1\}^{512}$. Let k be a uniform random element of $\{0,1\}^{256}$. Define δ_0 as the A_0-distance from* HSalsa_k *to uniform. Define δ_j, for $j \in \{1, \ldots, q\}$, as the A_j-distance from* Salsa_k *to uniform. Then the A-distance from* XSalsa_k *to uniform is at most $\delta_0 + \delta_1 + \cdots + \delta_q$.*

Proof. Define $H : K_1 \times I_1 \to K_2$ as $(k, i) \mapsto \mathrm{HSalsa}_k(i)$. Define $S : K_2 \times I_2 \to L$ as $(k, i) \mapsto \mathrm{Salsa}_k(i)$. Define $X : K_1 \times I_1 \times I_2 \to L$ as $(k, i_1, i_2) \mapsto \mathrm{XSalsa}_k(i_1, i_2)$. Then $X(k, i_1, i_2) = S(H(k, i_1), i_2)$. The A-distance from $X(k)$ to uniform is at most $\delta_0 + \delta_1 + \cdots + \delta_q$ by Theorem 3.1. □

Security proof for HSalsa20

Theorem 3.3 states that HSalsa20/r is secure if Salsa20/r is secure. The theorem applies to any distribution of keys, and in particular to the uniform distribution considered in Theorem 3.2. Combining Theorem 3.3 with Theorem 3.2 shows that XSalsa20/r is secure if Salsa20/r is secure.

Theorem 3.3. *Let k be a random element of $\{0,1\}^{256}$. Let A be an algorithm. Define $Q : \{0,1\}^{128} \times \{0,1\}^{512} \to \{0,1\}^{256}$ by $Q(x_6, x_7, x_8, x_9, s_0, s_1, \ldots, s_{15}) = (s_0 - x_0, s_5 - x_5, s_{10} - x_{10}, s_{15} - x_{15}, s_6 - x_6, s_7 - x_7, s_8 - x_8, s_9 - x_9)$ where $(x_0, x_5, x_{10}, x_{15})$ is the Salsa20 constant. Define B as the algorithm that, given an oracle $O : \{0,1\}^{128} \to \{0,1\}^{512}$, runs A with the oracle $i \mapsto Q(i, O(i))$. Then the A-distance from* HSalsa_k *to uniform is the same as the B-distance from* Salsa_k *to uniform.*

As a corollary, if Salsa20 has insecurity $\le \epsilon$ against any algorithm as fast as B, then HSalsa20 has insecurity $\le \epsilon$ against any algorithm as fast as A. Note that B has almost exactly the same speed as A.

Proof. Compare the definitions of Salsa20 and HSalsa20 to see that if $i = (x_6, x_7, x_8, x_9)$ and $\mathrm{Salsa}_k(i) = (s_0, s_1, \ldots, s_{15})$ then $\mathrm{HSalsa}_k(i) = (s_0 - x_0, s_5 - x_5, s_{10} - x_{10}, s_{15} - x_{15}, s_6 - x_6, s_7 - x_7, s_8 - x_8, s_9 - x_9) = Q(i, \mathrm{Salsa}_k(i))$. Hence $B(\mathrm{Salsa}_k) = A(i \mapsto Q(i, \mathrm{Salsa}_k(i))) = A(\mathrm{HSalsa}_k(i))$.

Let U be a uniform random function from $\{0,1\}^{128}$ to $\{0,1\}^{512}$. Define $V(i) = Q(i, U(i))$. Then V is a uniform random function from $\{0,1\}^{128}$ to $\{0,1\}^{256}$. Furthermore $B(U) = A(V)$.

The B-distance from Salsa_k to U is therefore the same as the A-distance from $\mathrm{HSalsa}_k(i)$ to V. □

The definition of Q in Theorem 3.3 is designed to provide two critical properties of the function $Q(i)$:

- $Q(i)$ is a public computation of $\mathrm{HSalsa}_k(i)$ from $\mathrm{Salsa}_k(i)$.
- $Q(i)$ is a public computation of uniform random strings from uniform random strings.

The first property takes advantage of the choice of indices $0, 5, 10, 15, 6, 7, 8, 9$ in the definition of HSalsa20. The second property is not a mere technicality—for example, if HSalsa20 were redefined to output $(z_0, z_0, z_0, z_0, z_0, z_0, z_0, z_0)$, then it would still be publicly computable from the Salsa20 input and output without knowledge of the key, but of course it would not be secure!

References

[1] — (no editor), *37th annual symposium on foundations of computer science*, Institute of Electrical and Electronics Engineers, New York, 1996. ISBN 0–8186–7594–2. See [7].

[2] — (no editor), *30th IEEE symposium on security and privacy (S&P 2009), 17–20 May 2009, Oakland, California, USA*, Institute of Electrical and Electronics Engineers, New York, 2009. ISBN 978-0-7695-3633-0. See [3].

[3] Martin R. Albrecht, Kenneth G. Paterson, Gaven J. Watson, *Plaintext recovery attacks against SSH*, in [2] (2009), 16–26. URL: http://www.isg.rhul.ac.uk/~kp/SandPfinal.pdf. Citations in this document: §1.

[4] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, Christian Rechberger, *New features of Latin dances: analysis of Salsa, ChaCha, and Rumba* (2007); see also newer version [5]. URL: http://eprint.iacr.org/2007/472.

[5] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, Christian Rechberger, *New features of Latin dances: analysis of Salsa, ChaCha, and Rumba*, in [37] (2007), 470–488; see also older version [4]. Citations in this document: §2.

[6] 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 [27], [36].

[7] Mihir Bellare, Ran Canetti, Hugo Krawczyk, *Pseudorandom functions revisited: the cascade construction and its concrete security*, in [1] (1996), 514–523; see also newer version [8]. Citations in this document: §1.

[8] Mihir Bellare, Ran Canetti, Hugo Krawczyk, *Pseudorandom functions revisited: the cascade construction and its concrete security* (2005); see also older version [7]. URL: http://www-cse.ucsd.edu/~mihir/papers/cascade.html. Citations in this document: §1, §1, §1, §3, §3, §3, §3, §3, §3, §3.

[9] Mihir Bellare, Joe Kilian, Phillip Rogaway, *The security of cipher block chaining*, in [26] (1994), 341–358; see also newer version [10].

[10] Mihir Bellare, Joe Kilian, Phillip Rogaway, *The security of the cipher block chaining message authentication code*, Journal of Computer and System Sciences **61** (2000), 362–399; see also older version [9]. ISSN 0022–0000. URL: http://www-cse.ucsd.edu/~mihir/papers/cbc.html. Citations in this document: §1.

[11] Mihir Bellare, Krzysztof Pietrzak, Phillip Rogaway, *Improved security analyses for CBC MACs*, in [40] (2005), 527–545; see also newer version [12].

[12] Mihir Bellare, Krzysztof Pietrzak, Phillip Rogaway, *Improved security analyses for CBC MACs* (2005); see also older version [11]. URL: http://www-cse.ucsd.edu/~mihir/papers/cbc-improved.html. Citations in this document: §1.

[13] Daniel J. Bernstein, *How to stretch random functions: the security of protected counter sums*, Journal of Cryptology **12** (1999), 185–192. ISSN 0933–2790. URL: http://cr.yp.to/papers.html#stretch. Citations in this document: §1.

[14] Daniel J. Bernstein, *A short proof of the unpredictability of cipher block chaining* (2005). URL: http://cr.yp.to/papers.html#easycbc. Citations in this document: §1.

[15] Daniel J. Bernstein, *Salsa20 specification* (2005). URL: http://cr.yp.to/snuffle.html. Citations in this document: §2, §2.

[16] Daniel J. Bernstein, *Understanding brute force*, ECRYPT STVL Workshop on Symmetric Key Encryption (2005). URL: http://cr.yp.to/papers.html#bruteforce. Citations in this document: §1.

[17] Daniel J. Bernstein, *Stronger security bounds for Wegman–Carter–Shoup authenticators*, in [24] (2005), 164–180. URL: http://cr.yp.to/papers.html#securitywcs. Citations in this document: §3.

[18] Daniel J. Bernstein, *The Salsa20 family of stream ciphers*, in [39] (2008). URL: http://cr.yp.to/papers.html#salsafamily. Citations in this document: §2.

[19] Daniel J. Bernstein, *Response to 'Slid pairs in Salsa20 and Trivium'* (2008). URL: http://cr.yp.to/snuffle.html. Citations in this document: §2.

[20] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems*, accessed 4 February 2011 (2011). URL: http://bench.cr.yp.to. Citations in this document: §1, §2.

[21] John Black, Shai Halevi, Alejandro Hevia, Hugo Krawczyk, Ted Krovetz, Phillip Rogaway, *UMAC: message authentication code using universal hashing*, RFC 4418 (2006). URL: http://www.ietf.org/rfc/rfc4418.txt. Citations in this document: §1.

[22] John Black, Phillip Rogaway, *A block-cipher mode of operation for parallelizable message authentication*, in [33] (2002), 384–397; see also newer version [23].

[23] John Black, Phillip Rogaway, *A block-cipher mode of operation for parallelizable message authentication* (2002); see also older version [22]. URL: http://www.cs.ucdavis.edu/~rogaway/papers/index.html. Citations in this document: §1.

[24] Ronald Cramer (editor), *Advances in Cryptology—EUROCRYPT 2005, 24th annual international conference on the theory and applications of cryptographic techniques, Aarhus, Denmark, May 22–26, 2005, proceedings*, Lecture Notes in Computer Science, 3494, Springer, 2005. ISBN 3-540-25910-4. See [17].

[25] 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: §2.

[26] Yvo Desmedt (editor), *Advances in cryptology—CRYPTO '94*, Lecture Notes in Computer Science, 839, Springer-Verlag, Berlin, 1994. See [9].

[27] Simon Fischer, Willi Meier, Côme Berbain, Jean-François Biasse, Matthew J. B. Robshaw, *Non-randomness in eSTREAM candidates Salsa20 and TSC-4*, in [6] (2006), 2–16. Citations in this document: §2.

[28] Shai Halevi, Tal Rabin (editors), *Theory of cryptography, third theory of cryptography conference, TCC 2006, New York, NY, USA, March 4-7, 2006, proceedings*, Lecture Notes in Computer Science, 3876, Springer, 2006. ISBN 3-540-32731-2. See [32].

[29] Tetsu Iwata, Kaoru Kurosawa, *OMAC: one-key CBC MAC*, in [30] (2003), 129–153. URL: http://www.nuee.nagoya-u.ac.jp/labs/tiwata/omac/omac.html. Citations in this document: §1.

[30] 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, 2003. ISBN 3-540-20449-0. See [29].

[31] Marc Joye (editor), *Topics in cryptology—CT-RSA 2003, the cryptographers' track at the RSA conference 2003, San Francisco, CA, USA, April 13-17, 2003, proceedings*, Lecture Notes in Computer Science, 2612, Springer, 2003. ISBN 3-540-00847-0. See [34].

[32] Charanjit S. Jutla, *PRF domain extension using DAGs*, in [28] (2006), 561–580. URL: http://eprint.iacr.org/2005/092. Citations in this document: §1.

[33] Lars R. Knudsen (editor), *Advances in cryptology—EUROCRYPT 2002, international conference on the theory and applications of cryptographic techniques, Amsterdam, the Netherlands, April 28–May 2, 2002, proceedings*, Lecture Notes in Computer Science, 2332, Springer, 2002. ISBN ISBN 3-540-43553-0. See [22], [35].

[34] Kaoru Kurosawa, Tetsu Iwata, *TMAC: two-key CBC MAC*, in [31] (2003), 33–49. URL: http://eprint.iacr.org/2002/092/. Citations in this document: §1.

[35] Ueli Maurer, *Indistinguishability of random systems*, in [33] (2002), 110–132. URL: http://www.iacr.org/archive/eurocrypt2002/eurocrypt2002.html. Citations in this document: §1.

[36] Mridul Nandi, *A simple and unified method of proving indistinguishability*, in [6] (2006), 317–334. Citations in this document: §1.

[37] Kaisa Nyberg (editor), *Fast software encryption, 15th international workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, revised selected papers* (2008). ISBN 978-3-540-71038-7. See [5].

[38] Deike Priemuth-Schmid, Alex Biryukov, *Slid pairs in Salsa20 and Trivium* (2008). URL: http://eprint.iacr.org/2008/405. Citations in this document: §2, §2, §2.

[39] Matthew Robshaw, Olivier Billet (editors), *New stream cipher designs*, Lecture Notes in Computer Science, 4986, Springer, 2008. ISBN 978-3-540-68350-6. See [18].

[40] Victor Shoup (editor), *Advances in cryptology—CRYPTO 2005: 25th annual international cryptology conference, Santa Barbara, California, USA, August 14–18, 2005, proceedings*, Lecture Notes in Computer Science, 3621, Springer, 2005. ISBN 3-540-28114-2. See [11].

[41] Douglas R. Stinson, Stafford E. Tavares (editors), *Selected areas in cryptography, 7th annual international workshop, SAC 2000, Waterloo, Ontario, Canada, August 14–15, 2000, proceedings*, Lecture Notes in Computer Science, 2012, Springer, 2001. ISBN 3-540-42069-X. See [43].

[42] 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: §2.

[43] Serge Vaudenay, *Decorrelation over infinite domains: the encrypted CBC-MAC case*, in [41] (2001), 189–201. Citations in this document: §1.