On the Salsa20 Core Function

Julio Cesar Hernandez-Castro Juan M. E. Tapiador
Jean-Jacques Quisquater

Crypto Group DICE – Universite Catholique de Louvain
Computer Science Department – Carlos III University of Madrid

February 13, 2008

Abstract

- We point out some weaknesses in the Salsa20 core function
- These could be exploited to obtain up to 2^{31} collisions for its full (20 rounds) version
    - We find an invariant for its main building block, the **quarterround** function, then extended to the **rowround**, **columnround** and **doubleround** functions
    - We find a subset of size 2^{32} for which the Salsa20 core behaves exactly as the transformation $f(x) = 2x$
    - An attacker can take advantage of this for constructing 2^{31} collisions for any number of rounds
- We finally show another weakness in the form of a differential characteristic with probability one, that proves the Salsa20 core does not have 2^{nd} preimage resistance.

Salsa20 design

- Salsa20 is a design by D. Bernstein
- Nowadays mostly known because of its submission to the eSTREAM Project, where it passed to Phase 3 without major known attacks:

  > *"The core of Salsa20 is a hash function with 64-byte input and 64-byte output. The hash function is used in counter mode as a stream cipher: Salsa20 encrypts a 64-byte block of plaintext by hashing the key, nonce, and block number and xor'ing the result with the plaintext."*

- Note, however, that in spite of its name, the Salsa20 "hash" function was never really intended for hashing.
- Some interesting weaknesses over reduced-round versions have recently been pointed out

Salsa20 design

- Salsa20 represents quite an original and flexible design
- The author justifies the use of very simple operations (addition, xor, constant-distance rotation) and the lack of multiplication or S-boxes
- This helps to develop a very fast primitive that is also, by construction, immune to timing attacks.

The quarterround function

The main building block of Salsa20 is the **quarterround** function, defined as follows:
If $y = \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \end{pmatrix}$ then $\textbf{quarterround}(y) = \begin{pmatrix} z_0 & z_1 \\ z_2 & z_3 \end{pmatrix}$, where:

$$z_1 = y_1 \oplus ((y_0 + y_3) \lll 7) \tag{1}$$
$$z_2 = y_2 \oplus ((z_1 + y_0) \lll 9) \tag{2}$$
$$z_3 = y_3 \oplus ((z_2 + z_1) \lll 13) \tag{3}$$
$$z_0 = y_0 \oplus ((z_3 + z_2) \lll 18) \tag{4}$$

and $X \lll n$ is the rotation of the 32-bit word X to the left by n positions.

The quarterround function

Theorem 1. For any 32-bit value A, an input of the form $\begin{pmatrix} A & -A \\ A & -A \end{pmatrix}$ is left invariant by the **quarterround** function, where $-A$ represents the only 32-bit integer satisfying $A + (-A) = 0 \pmod{2^{32}}$.

Proof. Simply by substituting in the equations above, we obtain that every rotation operates over the null vector, so $z_i = y_i$ for every $i \in (0..3)$ $\square$

The rowround function

Similarly, the **rowround** function, defined below, suffers from the same problem:

$$\text{If } y = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix} \text{ then}$$

$$\textbf{rowround}(y) = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix} \text{ where:}$$

$$(z_0, z_1, z_2, z_3) = \textbf{quarterround}(y_0, y_1, y_2, y_3) \tag{5}$$

$$(z_5, z_6, z_7, z_4) = \textbf{quarterround}(y_5, y_6, y_7, y_4) \tag{6}$$

$$(z_{10}, z_{11}, z_8, z_9) = \textbf{quarterround}(y_{10}, y_{11}, y_8, y_9) \tag{7}$$

$$(z_{15}, z_{12}, z_{13}, z_{14}) = \textbf{quarterround}(y_{15}, y_{12}, y_{13}, y_{14}) \tag{8}$$

The rowround function

Theorem 2.: Any input of the form $\begin{pmatrix} A & -A & A & -A \\ B & -B & B & -B \\ C & -C & C & -C \\ D & -D & D & -D \end{pmatrix}$, for any 32-bit values A, B, C and D, is left invariant by the **rowround** transformation.

Proof. This trivially follows from the repeated application of Theorem 1 to the four equations in the definition of **rowround**. $\square$

The rowround function: Other possibilities

- **Remark.** It is important to note that any other rearrangement of the equations from its canonical form:

  $$(z_{4i}, z_{4i+1}, z_{4i+2}, z_{4i+3}) = \textbf{quarterround}(y_{4i}, y_{4i+1}, y_{4i+2}, y_{4i+3})$$

  will suffer from the same problem whenever the rearranging permutation keeps on alternating subindex oddness.
- This implies that, from the 2^{512} possible inputs, at least one easily characterizable subset of size 2^{128} remains invariant by **rowround**.

The columnround function

The same happens with the **columnround** function, which is defined below:

If $y = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$ then

$\textbf{columnround}(y) = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix}$ where:

$$(z_0, z_4, z_8, z_{12}) = \textbf{quarterround}(y_0, y_4, y_8, y_{12}) \tag{9}$$

$$(z_5, z_9, z_{13}, z_1) = \textbf{quarterround}(y_5, y_9, y_{13}, y_1) \tag{10}$$

$$(z_{10}, z_{14}, z_2, z_6) = \textbf{quarterround}(y_{10}, y_{14}, y_2, y_6) \tag{11}$$

$$(z_{15}, z_3, z_7, z_{11}) = \textbf{quarterround}(y_{15}, y_3, y_7, y_{11}) \tag{12}$$

The columnround function

Theorem 3.: Any input of the form $\begin{pmatrix} A & -B & C & -D \\ -A & B & -C & D \\ A & -B & C & -D \\ -A & B & -C & D \end{pmatrix}$,
for any 32-bit values A, B, C and D, is left invariant by the **columnround** transformation.
Proof. This follows directly from the repeated application of Theorem 1, and can be seen as a dual of Theorem 2 $\square$

The doubleround function

Theorem 4.: Any input of the form $\begin{pmatrix} A & -A & A & -A \\ -A & A & -A & A \\ A & -A & A & -A \\ -A & A & -A & A \end{pmatrix}$ for any 32-bit value A, is left invariant by the **doubleround** transformation.

Proof. This is quite obvious. The point is that, due to the arrangement of the indexes in the **columnround** and the **rowround** function, we cannot have as free a hand. Here we are forced to make $B = -A$, $C = A$, and $D = -A$.
As **doubleround** is defined as the composition of a **columnround** and a **rowround** operation:

$$\mathbf{doubleround}(x) = \mathbf{rowround}(\mathbf{columnround}(x)) \qquad (13)$$

a common fixed point should be also a fixed point of its composition. □

Collision finding

Theorem 5.: For any input of the form $\begin{pmatrix} A & -A & A & -A \\ -A & A & -A & A \\ A & -A & A & -A \\ -A & A & -A & A \end{pmatrix}$

and for any 32-bit value A, the Salsa20 core function behaves as a linear transformation of the form $f(x) = 2x$, and this happens independently of the number of rounds.

Proof. As the Salsa20 "hash" is defined as:

$$Salsa20(x) = x + \textbf{doubleround}^{10}(x) \tag{14}$$

and every input of the said form is an invariant (fixed point) for the **doubleround** function, then:

$$Salsa20(x) = x + \textbf{doubleround}^{10}(x) = x + x = 2x \tag{15}$$

(And this happens independently of the number of rounds)

Collision finding

- The previous result is of great use in collision finding. All what is left now is to find two different nontrivial inputs, x and x', of the said form such that:

$$x \neq x' \quad \text{but} \quad 2x = 2x' \tag{16}$$

- Fortunately, this is possible thanks to modular magic, i.e. the fact that all operations in Salsa20 are performed mod 2^{32}.
- Let us assume that X is a 32-bit integer such that $X < 2^{31}$. Then, we define $X' = X + 2^{31}$. The interesting point here is that, even though $X \neq X'$, $2X = 2X' \pmod{2^{32}}$.

Collision finding

Theorem 6.: Any pair of inputs $\begin{pmatrix} Z & -Z & Z & -Z \\ -Z & Z & -Z & Z \\ Z & -Z & Z & -Z \\ -Z & Z & -Z & Z \end{pmatrix}$ and $\begin{pmatrix} Z' & -Z' & Z' & -Z' \\ -Z' & Z' & -Z' & Z' \\ Z' & -Z' & Z' & -Z' \\ -Z' & Z' & -Z' & Z' \end{pmatrix}$, such that $Z < 2^{31}$ and $Z' = Z + 2^{31}$, generate a collision for any number of rounds of the Salsa20 "hash" function, producing $\begin{pmatrix} 2Z & -2Z & 2Z & -2Z \\ -2Z & 2Z & -2Z & 2Z \\ 2Z & -2Z & 2Z & -2Z \\ -2Z & 2Z & -2Z & 2Z \end{pmatrix}$ as a common hash value.

Collision finding

Proof. This follows directly from the last observations and definitions. Substitution of the proposed input values into the formulæ for the Salsa20 “hash” will confirm this hypothesis. □

Remark: Theorem 6 implies that there are at least (these conditions are sufficient but probably not necessary) 2^{31} input pairs that generate a collision in the output, proving that indeed Salsa20 is not to be used as-is as a hash function.

As an example, two of these pairs are provided in the Appendix.

Collision finding: *A*-states

- Let us call inputs of the said form *A-states*.
- Then, the Salsa20 output of any *A-state* is also an *A-state* (where, in this case, A is even).
- It could be interesting to check whether these states could be reached at any intermediate step during a computation beginning with a *non-A state*. This would have important security implications.
- However, it could be easily shown that this is not the case, so any state leading to an *A-state* should be an *A-state* itself.

Collision finding: *A*-states

- This property has an interesting similitude with *Finney-states* for RC4 and could be useful in mounting an impossible fault analysis for the Salsa20 stream cipher, as *Finney-states* were of key importance on the impossible fault cryptanalysis of RC4.
- *A-states*, on the other hand, have the interesting advantage over *Finney-states* that their influence over the output is immediately recognized, so they can be detected in an even simpler way.
- However, is much less likely to reach an *A-state* by simply injecting random faults, as the set of conditions that should hold is larger than for the RC4 case.

Collision finding

- Once we have shown that the Salsa20 core function is not collision resistant, we focus on its security against 2^{nd} preimage attacks.
- Next result reveals that 2^{nd} preimage attacks are not only possible but even easy.
    - This property was presented informally before by Robshaw and later by Wagner [Message in the sci.crypt newsgroup on September 26th, 2005]

2^{nd} preimage attack

Theorem 7.: Every pair of inputs A, B with a difference of
$A - B = A \bigoplus B =$

$$\begin{pmatrix} \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} \\ \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} \\ \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} \\ \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} & \mathrm{0x80000000} \end{pmatrix}$$

will produce the same output over any number of Salsa20 rounds.

2nd preimage attack

Proof. This depends on a couple of interesting observations.

- The first one is that *addition* behaves as *xor* over the most significant bit (that changed by adding 0x80000000).
    - So the result in each of the four additions on **quarterround** is the same when both its inputs are altered by adding 2^{31} (differences cancel out mod 2^{32}).
- The second one is that in the **quarterround** function, all partial results $z_0, \ldots z_3$ are computed after an odd number (three in this case) of *addition/xor* operations.
    - This has as a result that **quarterround** conserves the input difference, and so it does **rowround**, **columnround** and **doubleround**.
    - As in the last stage of the Salsa20 core function the input is added to the output, cancelling out input differences. $\square$

2^{nd} preimage attack

- Theorem 6 could be now seen as a particular instance of this result (because $2 * 0x80000000 = 0x00000000$).
  - This result has some points in common with that on the existence of equivalent keys for TEA made by Kelsey et al., and also with the exact truncated differential found by Crowley.
  - A direct consequence of this result is that the effective key/input space of the Salsa20 core is reduced by half, so there is a speed up by a factor of 2 in any exhaustive key/input search attack.
  - This also means that $Salsa20(x) = y$ has solution for no more than (at most) half of the possible y's (Dunkelman)

Conclusions

- Although the Salsa20 “hash” function was never intended for cryptographic hashing
  - some previous results showed that finding a good differential for the core function was not as hard as might have been expected.
- Even though its author acknowledges that the Salsa20 core is not collision-free
  - no work has so far focused on finding and characterizing these collisions
- In this paper we explicitly show that there is a relevant amount (2^{31}) of easily characterizable collisions, together with an undesirable linear behavior over a large subset of the input space
- In a sense, Theorem 6 is a generalization of Robshaw’s previous observation

Conclusions: Implications to the Salsa20 stream cipher

- Since the stream cipher uses four diagonal constants to limit the attacker's control over the input (thus making unreachable the differences needed for a collision), these results have no straightforward implications on its security.
- However, these undesirable structural properties might be useful to mount an impossible fault attack for the stream cipher. Particularly, what we have called A-states could play a role analogous to Finney states for RC4, in way similar to that presented by Biham et al. at FSE'05.
- We consider this as an interesting direction for future research.

Conclusions: Possible improvements

- That being said, we still consider that Salsa20 design is very innovative and well-motivated. Further work along the same guidelines should be encouraged.
- Particularly, we believe that a new, perhaps more complex and time consuming definition of the **quarterround** function should lead to a primitive that would not be vulnerable to any of these weaknesses and could, in fact, provide a high-level security algorithm.
- This will, obviously, be more computationally expensive, but there may exist an interesting trade-off between incrementing the complexity of the **quarterround** function and decreasing the total number of rounds.

Conclusions: Possible improvements

- The use of the add-rotate-xor chain at every stage of the **quarterround** function considerably eases the extension of these bad properties to any number of rounds.
- Although the author justified this approach because of performance reasons, we believe that alternating this structure with xor-rotate-add and making all output words depending on all input words will present the cryptanalyst with a much more difficult task.
- This should be the subject of further study
  - ChaCha?
  - Xalxa?

Conclusions: Possible improvements

- On the other hand, in the light of our results we can also conclude that the inclusion of the diagonal constants is absolutely mandatory.
- An additional idea derived from our results is that less diagonal constants might suffice for stopping these kinds of undesirable structural properties
- with a significant efficiency improvement
  - that can vary from a 16% (from processing 384 bits to 448 bits in the same amount of time, that is, using only two diagonal constants)
  - up to a 33% (in the extreme case of fixing the most significant bit of two diagonal 32-bit values)

Thanks!

Any Questions?