Slid Pairs in Salsa20 and Trivium

Deike Priemuth-Schmid and Alex Biryukov

FSTC, University of Luxembourg
6, rue Richard Coudenhove-Kalergi,
L-1359 Luxembourg
(deike.priemuth-schmid, alex.biryukov)@uni.lu

Abstract. The stream ciphers Salsa20 and Trivium are two of the finalists of the eSTREAM project which are in the final portfolio of new promising stream ciphers. In this paper we show that initialization and key-stream generation of these ciphers is *slidable*, i.e. one can find distinct (Key, IV) pairs that produce identical (or closely related) key-streams. There are 2^{256} and more then 2^{39} such pairs in Salsa20 and Trivium respectively. We write out and solve the non-linear equations which describe such related (Key, IV) pairs. This allows us to sample the space of such related pairs efficiently as well as detect such pairs in large portions of key-stream very efficiently. We show that Salsa20 does not have 256-bit security if one considers general birthday and related key distinguishing and key-recovery attacks.

Key words: Salsa20, Trivium, eSTREAM, stream ciphers, cryptanalysis

1 Introduction

In 2005 Bernstein [2] submitted the stream cipher Salsa20 to the eSTREAM-project [5]. Original Salsa20 has 20 rounds, later 8 and 12 rounds versions Salsa20/8 and Salsa20/12 were also proposed. The cipher Salsa20 uses the hash function Salsa20 in a counter mode. It has 512-bit state which is initialized by copying into it 128 or 256-bit key, 64-bit nonce and counter and 128-bit constant.

Previous attacks on Salsa used differential cryptanalysis exploiting a truncated differential over three or four rounds. The first attack was presented by Crowley [4] which could break the 5 round version of Salsa20 within claimed 3^{165} trials. Later a four round differential was exploited by Fischer et al. [6] to break 6 rounds in 2^{177} trials and by Tsnunoo et al. [11] to break 7 rounds in about 2^{190} trials. The currently best attack by Aumasson et al. [1] covers 8 round version of Salsa20 with estimated complexity of 2^{251}.

In 2005 De Cannière and Preneel [3] submitted the stream cipher Trivium to the eSTREAM-project [5]. Trivium has an internal state of 288

bits and uses an 80-bit key and an 80-bit initial value (IV). The interesting part of Trivium is the nonlinear update function of degree 2. In [10] Raddum presented and attacked simplified versions of Trivium called Bivium but the attack on Trivium had a complexity higher than the exhaustive key search. Bivium was completely broken by Maximov and Biryukov [8] and an attack on Trivium with complexity about 2^{100} was presented which showed that key-size of Trivium can not be increased just by loading longer keys into the state. In [9] McDonald et al. attacked Bivium using SatSolvers. Another approach that gained attention recently is to reduce the key setup of Trivium as done by Turan and Kara [12] and Vielhaber [13]. So far no attack faster than exhaustive key search was shown for Trivium.

In this paper we start with our investigation of Salsa20 followed by a description of the attacks. We show that the following observation holds: suppose that you are given two black boxes, one with Salsa20 and one with a random mapping. The attacker is allowed to chose a relation $\boldsymbol{\mathcal{F}}$ for a pair of inputs, after which a secret initial input x is chosen and a pair $(x, \boldsymbol{\mathcal{F}}(x))$ is encrypted either by Salsa20 or by a random mapping. We stress that only the relation $\boldsymbol{\mathcal{F}}$ is known to the attacker. The goal of the attacker is given a pair of ciphertexts to tell whether they were encrypted by Salsa20 or by a random mapping. To make the life of the attacker more difficult the pair may be hidden in a large collection of other ciphertexts. It is clear that for a truly random mapping no useful relation $\boldsymbol{\mathcal{F}}$ would exist and moreover there is no way of checking a large list except for checking all the pairs or doing a birthday attack. On the other hand Salsa20 can be easily distinguished from random in both scenarios if $\boldsymbol{\mathcal{F}}$ is a carefully selected function related to the round-structure of Salsa20. Moreover since in Salsa20 the initial state is initialized with the secret key, known nonce, counter and constant it is not only a distinguishing but also a complete key-recovery attack. Our attacks are independent of the number of rounds in Salsa and thus work for all the 3 versions of Salsa. We also show a general birthday attack on 256-bit key Salsa20 with complexity 2^{192} which can be further sped up twice using sliding observations.

In the second part of this paper we describe our results about Trivium which show a large related key-class (2^{39} out of 2^{80} keys) which produce identical key-streams up to a shift. We solve the resulting non-linear sliding equations using Magma and present several examples of such slid key-IV pairs. The interesting observation is that for shift of 111 clocks

24-key-bits do not appear in these equations and thus for a fixed IV there is a 2^{24} freedom of choice for the key that may have a sliding property.

2 Slid Pairs in Salsa20

2.1 Brief Description of Salsa20

The Salsa20 encryption function uses the Salsa20 hash function in a counter mode. The internal state of Salsa20 is a 4×4 - matrix of 32-bit words. A vector (y_0, y_1, y_2, y_3) of four words is transformed into (z_0, z_1, z_2, z_3) by calculating[1]

$$\begin{aligned} z_1 &= y_1 \oplus ((y_0 + y_3) \lll 7), \\ z_2 &= y_2 \oplus ((z_1 + y_0) \lll 9), \\ z_3 &= y_3 \oplus ((z_2 + z_1) \lll 13), \\ z_0 &= y_0 \oplus ((z_3 + z_2) \lll 18). \end{aligned}$$

This nonlinear operation is called **quarterround** and it is the basic part of the **columnround** where it is applied to columns (y_0, y_4, y_8, y_{12}), (y_5, y_9, y_{13}, y_1), $(y_{10}, y_{14}, y_2, y_6)$ and $(y_{15}, y_3, y_7, y_{11})$ as well as of the **rowround** which transforms rows (y_0, y_1, y_2, y_3), (y_4, y_5, y_6, y_7), $(y_8, y_9, y_{10}, y_{11})$ and $(y_{12}, y_{13}, y_{14}, y_{15})$. A so called **doubleround** consists of a columnround followed by a rowround. The doubleround function of Salsa20 is repeated 10 times. If Y denotes the matrix a key-stream block is defined by

$$Z = Y + \text{doubleround}^{10}(Y) \ .$$

In this feedforward the symbol "+" denotes the addition modulo 2^{32}.

One columnround as well as one rowround has 4 quarterrounds which means 48 word operations in total. Salsa20 consists of 10 doublerounds which gives altogether 960 word operations. The feedforward at the end of Salsa20 has 16 word operations which concludes 976 word operations in total for one encryption.

The cipher takes as input a 256-bit key $(k_0, \ldots, k_7)$, a 64-bit nonce (n_0, n_1) and a 64-bit counter (c_0, c_1). A 128-bit key version of Salsa20 copies the 128-bit key twice. In this paper we mainly concentrate on the 256-bit key version. The remaining four words are set to fixed publicly known constants, denoted with $\sigma_0, \sigma_1, \sigma_2$ and σ_3.

[1] Here the symbol "+" denotes the addition modulo 2^{32}, the other two symbols work at the level of the bits with "$\oplus$" as XOR-addition and "$\lll$" as a shift of bits.

2.2 Slid Pairs

The structure of a doubleround can be rewritten as columnround then a matrix transposition another columnround followed by a second transposition. Now the 10 doublerounds can be transferred into 20 columnrounds each followed by a transposition.

We define $\mathcal{F}$ to be a function which consists of a columnround followed by a transposition. If we have 2 triples (key1, nonce1, counter1) and (key2, nonce2, counter2) so that

$$\begin{aligned}&\mathcal{F}\,[\,1^{\text{st}}\text{ starting state (key1, nonce1, counter1)}]\\ &=\ 2^{\text{nd}}\text{ starting state (key2, nonce2, counter2)}\end{aligned}$$

then this property holds for each point during the round computation of Salsa20 especially for the end of the round computation. Pay attention that the feedforward at the end of Salsa20 destroys this property. We call such a pair of a 1^{st} and 2^{nd} starting state a slid pair. The relation of a slid pair is shown in Fig. 1.

In a starting state four words are constants and 12 words can be chosen freely which leads to a total amount of 2^{384} possible starting states. If we want that a starting state after applying function $\mathcal{F}$ results in a 2^{nd} starting state we obtain four wordwise equations. This means we can choose eight words of the 1^{st} starting state freely whereas the other four words are determined by the equations as well as the words for the 2^{nd} starting state. This leads to a total amount of 2^{256} possible slid pairs.

For the 128-bit key version no such slid pair exists due to the additional constrains of four fewer words freedom in the 1^{st} starting state and four more wordwise equations in the 2^{nd} starting state.

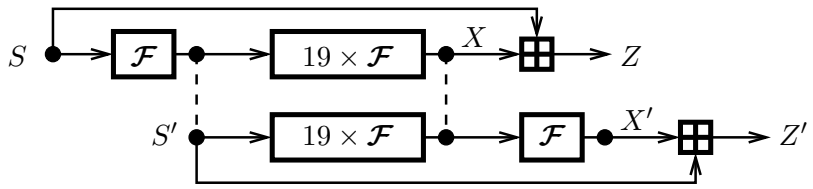

Fig. 1. relation of a slid pair

With function $\boldsymbol{\mathcal{F}}$ we get two equations $S' = \boldsymbol{\mathcal{F}}(S)$ and $X' = \boldsymbol{\mathcal{F}}(X)$. The words for these matrices we denote as

$$S = \begin{pmatrix} \sigma_0 & k_0 & k_1 & k_2 \\ k_3 & \sigma_1 & n_0 & n_1 \\ c_0 & c_1 & \sigma_2 & k_4 \\ k_5 & k_6 & k_7 & \sigma_3 \end{pmatrix} \quad X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} \quad Z = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix}$$

$$S' = \begin{pmatrix} \sigma_0 & k'_0 & k'_1 & k'_2 \\ k'_3 & \sigma_1 & n'_0 & n'_1 \\ c'_0 & c'_1 & \sigma_2 & k'_4 \\ k'_5 & k'_6 & k'_7 & \sigma_3 \end{pmatrix} \quad X' = \begin{pmatrix} x'_0 & x'_1 & x'_2 & x'_3 \\ x'_4 & x'_5 & x'_6 & x'_7 \\ x'_8 & x'_9 & x'_{10} & x'_{11} \\ x'_{12} & x'_{13} & x'_{14} & x'_{15} \end{pmatrix} \quad Z' = \begin{pmatrix} z'_0 & z'_1 & z'_2 & z'_3 \\ z'_4 & z'_5 & z'_6 & z'_7 \\ z'_8 & z'_9 & z'_{10} & z'_{11} \\ z'_{12} & z'_{13} & z'_{14} & z'_{15} \end{pmatrix}.$$

The set up of the system of equations for a whole Salsa20 computation is too complicated but the equations for the computation of $\boldsymbol{\mathcal{F}}$ are very clear. For a complete description of the equations see the appendix A.1. Both systems of equations coming from the relation $\boldsymbol{\mathcal{F}}$ are related in that way that, if one knows part of nonce and / or counter one can decide which system to use to solve the equations. Due to the eight words freedom we have in a 1^{st} or 2^{nd} starting state there are some relations in the 12 non-fixed words. For the 2^{nd} starting state these relations are very clear as they deal only with words

$$0 = k'_2 + k'_1, \quad 0 = k'_3 + n'_1, \quad 0 = c'_1 + c'_0 \quad \text{and} \quad 0 = k'_7 + k'_6 \ , \qquad (1)$$

whereas for the 1^{st} starting state these relations depend on the bits and thus are more complicated. Sliding by the function $\boldsymbol{\mathcal{F}}$ is applicable to any version of Salsa20/r where r is even. For r odd there would be no transposition at the end of the round computation, equations are a bit different, though still solvable.

2.3 Sliding State Recovery Attack on the Davies-Meyer Mode

In this subsection we consider a general state-recovery slide attack on a Davies-Meyer construction. We demonstrate it on an example of Davies-Meyer feedforward used with the iterative permutation from Salsa20. The feedforward breaks the sliding property and makes slide attack more complicated to mount. We consider the following scenario:

1. The oracle chooses a secret 512-bit state S (here we assume that there is no restriction of 128-bit diagonal constants and the full 512 bits can be chosen at random).
2. The oracle computes $\boldsymbol{\mathcal{F}}(S) = S'$.

3. The oracle computes Salsa20(S), Salsa20(S') and gives them to the attacker.
4. The goal of the attacker is to recover the secret state S.

Due to the weak diffusion of $\boldsymbol{\mathcal{F}}$ the attacker can write separate systems of equations for each column of S. With the four guesses of 2^{64} steps each the attacker can completely recover the 512-bit secret state S. The attacker will have to solve equations of the type:

$$z'_1 = [s_4 \oplus ((s_0 + s_{12}) \lll 7)] + [(z_4 - s_4) \oplus ((z_0 - s_0 + z_{12} - s_{12}) \lll 7)] \ .$$

For a detailed description see Appendix A.2. This shows that Salsa20 without the diagonal constants is easily distinguishable from a random function, for which a similar task would require about 2^{511} steps.

The addition of the diagonal constants reduces the flexibility of the oracle in a choice of the initial states to 2^{256} but the attack works even better:

1. The oracle chooses a starting state S' with the key k', nonce n' and counter c' satisfying equations (1). The attacker does not know this state.
2. The oracle applies $\boldsymbol{\mathcal{F}}^{-1}(S')$ to compute the related key k, nonce n and counter c.
3. The oracle computes Salsa20(S), Salsa20(S') and gives them to the attacker.
4. The goal of the attacker is to recover the secret state S.

The knowledge of the diagonal makes the previous attack even faster and allows the full 384-bit (256-bit entropy) state recovery with complexity of $4 \cdot 2^{32}$. If the attacker chooses the nonce and the counter n', c' (160-bits of entropy) then the complexity drops to $2 \cdot 2^{32}$.

Furthermore if nonce and counter n, c are known (128-bits of entropy left). The state can be recovered immediately (with or without knowing counter c' and nonce n') as is shown in the following section.

Table 1 shows the time complexities for the described attacks, memory complexity is negligible.

2.4 Related Key Key-Recovery Attack on Salsa20

Let us assume that we know two ciphertexts and the corresponding nonces and counters. We do not know both keys but we know that both starting states differ by function $\boldsymbol{\mathcal{F}}$ which gives the relation shown in Fig. 2 and

Table 1. Time complexities for state-recovery attacks

known words of the starting states	sliding on Salsa20	random oracle
nothing	2^{66}	2^{511}
only diagonal	2^{34}	2^{255}
diagonal, nonce and counter n', c'	2^{33}	2^{159}
diagonal, nonce and counter n, c	$O(1)$	2^{127}
diagonal, nonce and counter n, c and n', c'	$O(1)$	2^{63}

that the 2nd starting state conforms to the initial state format of Salsa20 (i.e. has proper 128-bit constant on the diagonal). In this subsection we show that this information is sufficient to completely recover the two related 256-bit secret keys of Salsa20 with $O(1)$ complexity.

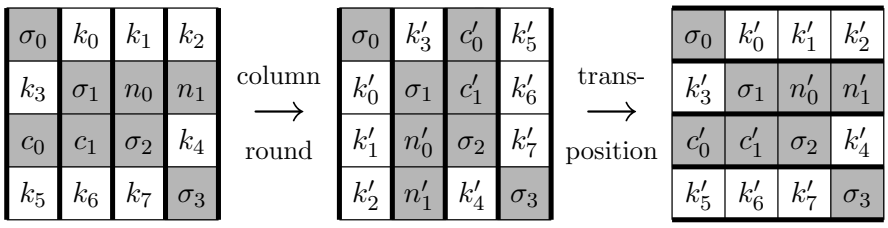

Fig. 2. relation of the 1st and 2nd starting state

With the knowledge of both nonces and counters (indicated as grey squares in Fig. 2) we are able to recover four words of the first unknown key and two words of the second unknown key (marked by bold letters) by solving the equations[2]

$$\begin{aligned} \boldsymbol{k'_3} &= -n'_1 & \boldsymbol{k_0} &= k'_3 \oplus ((n'_1 + n'_0) \lll 13) \\ \boldsymbol{k_6} &= n'_1 \oplus ((n'_0 + \sigma_1) \lll 9) & \boldsymbol{k'_4} &= -c'_0 + ((c'_1 \oplus n_0) \ggg 13) \\ \boldsymbol{k_1} &= c'_0 \oplus ((k'_4 + \sigma_2) \lll 9) & \boldsymbol{k_7} &= k'_4 \oplus ((\sigma_2 + n_0) \lll 7) \ . \end{aligned}$$

At this point we still have not used the similar equations from the bottom of the round computation. If part of the nonce / counter is unknown these equations can be used to check our guesses.

[2] The complete System of equations is shown in appendix A.1 here we only give the rearranged equations.

Then two similar systems of equations are left to get the other 10 key words. For the first one we solve the equation

$$z'_2 = [(z_8 - c_0) \oplus ((z'_1 - \boldsymbol{k'_0} + z_0 - \sigma_0) \lll 9)] + [c_0 \oplus ((\boldsymbol{k'_0} + \sigma_0) \lll 9)] \ , \quad (2)$$

whereas some words of the ciphertexts are used and only k'_0 is unknown. With the solution of k'_0 we get four more words

$$\begin{array}{ll} \boldsymbol{k'_1} = c_0 \oplus ((k'_0 + \sigma_0) \lll 9) & \boldsymbol{k'_2} = -k'_1 \\ \boldsymbol{k_5} = k'_2 \oplus ((k'_1 + k'_0) \lll 13) & \boldsymbol{k_3} = k'_0 \oplus ((\sigma_0 + k_5) \lll 7) \ . \end{array}$$

For the second system we solve a similar equation to get k'_5

$$z'_{13} = [(z_7 - n_1) \oplus ((z'_{12} - \boldsymbol{k'_5} + z_{15} - \sigma_3) \lll 9)] + [n_1 \oplus ((\boldsymbol{k'_5} + \sigma_3) \lll 9)] \ . \quad (3)$$

With the solution of k'_5 we get the last four words

$$\begin{array}{ll} \boldsymbol{k'_6} = n_1 \oplus ((k'_5 + \sigma_3) \lll 9) & \boldsymbol{k'_7} = -k'_6 \\ \boldsymbol{k_4} = k'_7 \oplus ((k'_6 + k'_5) \lll 13) & \boldsymbol{k_2} = k'_5 \oplus ((\sigma_3 + k_4) \lll 7) \ . \end{array}$$

Equation (2) (similarly (3)) can be solved just by checking all 2^{32} possibilities for k'_0 (respectively k'_5 for (3)), however it can be done much more efficiently with less then 2^8 steps by guessing k'_0 (or k'_5) gradually and checking the bitwise equations. We solved 16 equations, but used 26 of the 64 equations in the appendix A.1 to get these 16 equations. The complexity to get all the 16 key words is approximately 56 word operations which is much faster than a single Salsa20 encryption (976 word operations). For both keys we compute Salsa20 and compare the calculated ciphertexts to the given ones. If they match we have found the right keys.

2.5 A Generalized Related Key Attack on Salsa20

Suppose we are given a (possibly large) list of ciphertexts with the corresponding nonces and counters and we are told that in this list the slid pair is hidden. The question is, can we find slid pairs in a large list of ciphertexts efficiently? As we saw in the previous section, given such slid ciphertext pair it is easy to compute both keys. The task is made more difficult by the feedforward of Salsa20, which destroys the sliding relationship. Nevertheless in this section we show that given a list of ciphertexts of size $O(2^l)$ it is possible to detect a slid pair with memory and time complexity of just $O(2^l)$[3]. The naive approach which would require to

[3] Sorting is done via Bucket sort so we save the logarithmic factor l in complexity.

check for each possible pair the equations from function $\boldsymbol{\mathcal{F}}$ will have complexity $O(2^{2l})$ which is too expensive. Our idea is to reduce the amount of potential pairs by sorting them by eight precomputed words, so that only elements where these eight words match have the possibility to yield a slid pair. After decreasing the number of possible pairs in that way we can check the remaining pairs using additional constraints coming from the sliding equations.

For the sorting we use Bucket sort because each word has only 2^{32} possibilities. The number of words we sort by is equal to the number of runs of Bucket sort.

We have a set M of ciphertexts with corresponding nonces and counters. Each ciphertext can be either a 1^{st} starting state or a 2^{nd} starting state to regard this the set is stored twice first under M_1 to check for possible 1^{st} starting states and second under M_2 to check for possible 2^{nd} starting states.

Step 1: **Sort the first list**
For each element in set M_1 undo the feedforward for the four words on the diagonal and $x_9 = z_9 - c_1$. Then sort M_1 by the specified eight words $x_0, x_5, x_{10}, x_{15}, x_9$ and c_1, z_1, z_{13}.

Step 2: **Sort the second list**[4]
Select only elements of M_2 that satisfy equation $0 = c'_0 + c'_1$ since only such an entry can be a 2^{nd} starting state. For each element undo the feedforward for the four words on the diagonal and $x'_6, \ldots, x'_9$ because nonces and counters are known. Then compute for each element the words marked in bold in the equations

$$
\begin{aligned}
\boldsymbol{x_0} &= x'_0 \oplus ((z'_2 + z'_3) \lll 18) & \boldsymbol{k'_3} &= -n'_1 \\
\boldsymbol{x_5} &= x'_5 \oplus ((z'_4 + n'_1 + x'_7) \lll 18) & \boldsymbol{x_{10}} &= x'_{10} \oplus ((x'_9 + x'_8) \lll 18) \\
\boldsymbol{x_{15}} &= x'_{15} \oplus ((z'_{13} + z'_{14}) \lll 18) & \boldsymbol{x_1} &= (z'_4 + n'_1) \oplus ((x'_7 + x'_6) \lll 13) \\
\boldsymbol{x_9} &= x'_6 \oplus ((x_5 + x_1) \lll 7) & \boldsymbol{k_0} &= k'_3 \oplus ((n'_1 + n'_0) \lll 13) \\
\boldsymbol{c_1} &= n'_0 \oplus ((\sigma_1 + k_0) \lll 7) & \boldsymbol{z_1} &= x_1 + k_0 \\
\boldsymbol{k_6} &= n'_1 \oplus ((n'_0 + \sigma_1) \lll 9) & \boldsymbol{z_{13}} &= (k_6 + x'_7) \oplus ((x'_6 + x_5) \lll 9) \ .
\end{aligned}
$$

[4] If the number of rounds of Salsa is odd then such simple sorting would not be possible, since Salsa equations are easier to solve in reverse direction. In our approach we know 2 words at the input and 3 words at the output of the columnround which is easier to solve than the opposite (3 words at the input vs. 2 at the output). Nevertheless the system is still solvable.

During this computation we calculate three key words k_0, k_6 and k'_3. Sort the set M_2 by the calculated eight words $x_0, x_5, x_{10}, x_{15}, x_9$ and c_1, z_1, z_{13} for the potential 1^{st} starting states.

Step 3: **Check each possible pair**
Cross check all the possible pairs that match in the eight words and thus satisfy the 256-bit filtering. For the conforming pairs we can continue the check, using the following equations. If a test condition is wrong this pair can not be a slid pair. For each pair undo for the ciphertext of the 1^{st} starting state the feedforward for the word $x_6 = z_6 - n_0$. Then compute the bold variables and check the conditions for the three equations

1. comp.	$\boldsymbol{k'_4} = -c'_0 + ((n_0 \oplus c'_1) \ggg 13)$	$\boldsymbol{x'_{11}} = -x'_8 + ((x_6 \oplus x'_9) \ggg 13)$
check	$z'_{11} = x'_{11} + k'_4$	
2. comp.	$\boldsymbol{k_1} = c'_0 \oplus ((k'_4 + \sigma_2) \lll 9)$	$\boldsymbol{x_2} = x'_8 \oplus ((x'_{11} + x_{10}) \lll 9)$
check	$z_2 = x_2 + k_1$	
3. comp.	$\boldsymbol{k_7} = k'_4 \oplus ((\sigma_2 + n_0) \lll 7)$	$\boldsymbol{x_{14}} = x'_{11} \oplus ((x_{10} + x_6) \lll 7)$
check	$z_{14} = x_{14} + k_7$.	

During this computation we calculate the three key words k_1, k_7 and k'_5.

For the rest of the pairs we have two more similar systems of equations to check. For the first one we solve (2) and if there is no solution for k'_0 this pair can not be a slid pair. Otherwise we use the k'_0 to compute four more key words while we check two more conditions

1. comp.	$\boldsymbol{k'_1} = c_0 \oplus ((k'_0 + \sigma_0) \lll 9)$	$\boldsymbol{k'_2} = -k'_1$
	$\boldsymbol{k_5} = k'_2 \oplus ((k'_1 + k'_0) \lll 13)$	$\boldsymbol{x'_1} = z'_1 - k'_0$
	$\boldsymbol{x_{12}} = (z'_3 - k'_2) \oplus ((z'_2 - k'_1 + x'_1) \lll 13)$	
check	$z_{12} = x_{12} + k_5$	
2. comp.	$\boldsymbol{k_3} = k'_0 \oplus ((\sigma_0 + k_5) \lll 7)$	$\boldsymbol{x_4} = x'_1 \oplus ((x_0 + x_{12}) \lll 7)$
check	$z_4 = x_4 + k_3$.	

For the second system we similarly solve (3) and again if there is no solution for k'_5 this pair can not be a slid pair. Otherwise we use the k'_5 to compute the rest of the key words while we check two more conditions

1. comp.	$\boldsymbol{k'_6} = n_1 \oplus ((k'_5 + \sigma_3) \lll 9)$	$\boldsymbol{k'_7} = -k'_6$
	$\boldsymbol{k_4} = k'_7 \oplus ((k'_6 + k'_5) \lll 13)$	$\boldsymbol{x'_{12}} = z'_{12} - k'_5$
	$\boldsymbol{x_{11}} = (z'_{14} - k'_7) \oplus ((z'_{13} - k'_6 + x'_{12}) \lll 13)$	
check	$z_{11} = x_{11} + k_4$	
2. comp.	$\boldsymbol{k_2} = k'_5 \oplus ((\sigma_3 + k_4) \lll 7)$	$\boldsymbol{x_3} = x'_{12} \oplus ((x_{15} + x_{11}) \lll 7)$
check	$z_3 = x_3 + k_2$.	

For the checking of the potential slid pairs we have 9 extra test conditions. We would expect only 7 conditions but due to the different arithmetic operations the dependencies of the equations are not clear. In total we have at least filtering power of 32×7 bits. Thus we expect that only the correct slid pairs survive this check. The remaining pairs are the correct slid pairs for which we completely know both keys.

Complexity. Assume we are given a list of 2^l ciphertexts with corresponding nonces and counters. Instead of storing the list twice we use two kinds of pointers, one kind for the potential 1^{st} starting states and the other one for the potential 2^{nd} starting states. For the pointers we need $l/32 \times 2^l$ words of memory. A summary for the complexity of different lists is given in Table 2. The larger the list of the random states in which our target is hidden – the larger would be the complexity of the attack. However the time complexity of the attack grows only linearly with the size of the list. The memory is given in words and the time in Salsa encryptions.

Table 2. Complexities for different list seizes

list size	memory	time
2^{128}	28×2^{128}	2^{122}
2^{192}	32×2^{192}	2^{186}
2^{256}	36×2^{256}	2^{250}

2.6 Time-Memory Tradeoff Attacks on Salsa

Salsa20 has 2^{384} possible starting states. We notice that the square root of 2^{384} is less than the keyspace size for keys longer than 192-bits. Thus a trivial birthday attack on 256-bit key Salsa20 would proceed as follows:

During the preprocessing stage generate a list of 2^{192} randomly chosen starting states and run for each of them Salsa20 to get a sample of ciphertexts. Afterwards we sort this list by the ciphertexts. During the on-line stage we capture 2^{192} ciphertexts for which we want to find the keys. We do not have to store these ciphertexts and can check each of them immediately for a match with the sorted array of precomputed ciphertexts. If we have a match we retrieve the corresponding key from our table. Of course due to very high memory complexity this attack can be only viewed as a certificational weakness.

If we can choose the nonce or the counter there exist only 2^{320} different starting states reducing the attack to precomputation and memory complexity of 2^{160} and if we can choose both – the state space drops to 2^{256} and the attack complexity drops to 2^{128}. Similar reasoning for 128-bit key Salsa20 would yield attack with 2^{64} complexity. Thus it is crucial for the security of Salsa20 that nonces are chosen at random for each new key and the counter is not stuck at some fixed value (like 0, for example).

The complexities are summarized in Table 3 where R stands for a complete run of Salsa20 and M for a matrix of Salsa (16 words).

Table 3. Complexities for the Birthday attack

attack	precomputation	memory	time	captured ciphert.
chosen nonce and counter	$R \times 2^{128}$	$2M \times 2^{128}$	2^{128}	2^{128}
chosen nonce or counter	$R \times 2^{160}$	$2M \times 2^{160}$	2^{160}	2^{160}
general	$R \times 2^{192}$	$2M \times 2^{192}$	2^{192}	2^{192}
using sliding property	$R \times 2^{192}$	$2.5M \times 2^{192}$	2^{192}	2^{191}

Improved Birthday Using the Sliding Property. We can use the sliding property to increase the efficiency of the birthday attack twice (which can be translated into reduction of memory, time or increase of success probability of the birthday attack). During the preprocessing stage we generate a sample of 2^{nd} starting states by using (1) and choose the remaining eight words at random. We compute the corresponding ciphertexts for these states as well as the eight specified words for the corresponding 1^{st} starting states mentioned in section 2.5. We use two kinds of pointers to sort this generated list by the ciphertexts for the 2^{nd} starting states and by the eight words for the corresponding 1^{st} starting state. We capture ciphertext from the key-stream where we also know the nonce and the counter and check if it is matching a 2^{nd} starting state from our list (direct birthday) or is a correct 1^{st} starting state for one of the states from our collection (indirect birthday). In both cases we learn the key for this ciphertext.

3 Slid Pairs for Trivium

3.1 Brief Description of Trivium

The designers introduced the stream cipher Trivium with a state size of 288 bits. This internal state can be split into three registers. The first register which we call A has length 93, the second one called B has length 84 and the last register named C has 111 bits. Trivium uses an 80-bit key, an 80-bit initial value and a nonlinear update function of degree 2.

Update and Key-Stream Production. The internal state is denoted in the following way

$$\text{A: } (s_1, s_2, \ldots, s_{93}) \quad \text{B: } (s_{94}, s_{95}, \ldots, s_{177}) \quad \text{C: } (s_{178}, s_{279}, \ldots, s_{288}) \ .$$

The nonlinear update function uses 15 bits of the internal state to compute three new bits each for one register and the key-stream bit z_i is calculated by adding only 6 of these 15 bits together. In the following pseudo-code all computations are over GF(2).

$$\begin{aligned}
t_1 &\longleftarrow s_{66} + s_{93} \\
t_2 &\longleftarrow s_{162} + s_{177} \\
t_3 &\longleftarrow s_{243} + s_{288} \\
z_i &\longleftarrow t_1 + t_2 + t_3 \\
t_1 &\longleftarrow t_1 + s_{91} \cdot s_{92} + s_{171} \\
t_2 &\longleftarrow t_2 + s_{175} \cdot s_{176} + s_{264} \\
t_3 &\longleftarrow t_3 + s_{286} \cdot s_{287} + s_{69} \\
\text{A: } & (s_1, s_2, ..., s_{93}) \longleftarrow (t_3, s_1, \ldots, s_{92}) \\
\text{B: } & (s_{94}, s_{95}, \ldots, s_{177}) \longleftarrow (t_1, s_{94}, \ldots, s_{176}) \\
\text{C: } & (s_{178}, s_{279}, \ldots, s_{288}) \longleftarrow (t_2, s_{178}, \ldots, s_{287})
\end{aligned}$$

Key and IV Setup. In register A the key is loaded and in register B the IV. All remaining positions in the three registers are set to zero except for the last three bits in register C which are set to one

$$\begin{aligned}
\text{A: } & (s_1, s_2, ..., s_{93}) \longleftarrow (K_{80}, \ldots, K_1, 0, \ldots, 0) \\
\text{B: } & (s_{94}, s_{95}, \ldots, s_{177}) \longleftarrow (IV_{80}, \ldots, IV_1, 0, \ldots, 0) \\
\text{C: } & (s_{178}, s_{279}, \ldots, s_{288}) \longleftarrow (0, \ldots, 0, 1, 1, 1) \ .
\end{aligned}$$

In this paper we refer to this state with key, IV and 128 fixed positions as starting state. After the registers are initialized in the described way the cipher is clocked 4×288 times using the update function without producing any key-stream bits. This will finish the key setup. Now each following clock will produce a key-stream bit.

3.2 Slid Pairs

We start with the observation made by Jin Hong on the eSTREAM forum [7], that it is possible to produce sliding states in Trivium. We searched for pairs of key and IV which produce another starting state after a few clocks. If we have a key and IV pair (K_1, IV_1) which produce another starting state with a key and IV pair (K_2, IV_2), the created key-stream by (K_2, IV_2) will be the same as the one created by (K_1, IV_1) except for a shift of some bits. The number of shifted bits is the same as the number of clocks which are needed to get from the first starting state to the second one. We call such a pair of two key and IV pairs a slid pair and denote this with $[(K_1, IV_1), (K_2, IV_2), c]$ whereas c stands for the number of clocks-shifts.

Due to the special structure of the third register with 108 zeros and the last three ones the first possibility of a second starting state to occur is after 111 clocks. Each following clock gives the chance for a second starting state. Two examples for slid pairs are given in the appendix B.

3.3 Systems of Equations

We describe the second starting state as polynomial equations in the 80 key and 80 IV variables of the first key and IV. The 128 fixed positions in a starting state yield a system of equations with 160 variables and 128 equations. We have more variables than equations which gives us freedom in 32 variables. To solve these systems we used the F4 algorithm implemented in the computer algebra system Magma. A solution of such a system of equations gives us the first pair of key and IV of a slid pair. To get the second key we use the system of equations which describes the key in the second starting state and get the solution by replacing the variables with the known values of the first key and IV pair. In the same way we use the system of equations which describes the IV in the second starting state to compute the second IV. One could also just clock Trivium c times starting from (K_1, IV_1) to get (K_2, IV_2).

Some Facts about these Systems. The system of equations for the first instance which appears after 111 clocks has only 136 variables because the last 24 bits of the key do not occur in this system. Furthermore 16 bits are given a priori due to the 13 zeros in register A and 3 ones in register C. The degree of the monomials in the equations raised from 1 to 3. Magma was able to solve this system without guessing any further variables. Due to the missing of the 24 key bits in the equations these

bits can be chosen arbitrarily. This leads us to 2^{24} different keys for one IV in the first key and IV pair of a slid pair.

Table 4 collects some facts for the systems we were able to solve with Magma to get a Gröbner basis and the values for the key and IV for the first starting state.

Table 4. Some facts for the systems of equations

clock-shift c	111	112	113	114	115
variables in equations	136	137	138	139	140
last key bits not in the equation	24	23	22	21	20
a priori given bits	16	15	14	13	13
guess bits to solve[5]	0	4	6	8	10
computing time magma (days)	2.5	2.5	10	32.5	64

The higher the clock-shift will be the more complicated the systems of equations will get. For each clock-shift another system of equations is needed but for every step of c most of the equations are the same or related. Due to the length of register C which defines the occurrence of a second starting state we have at least 111 clock-shifts. Thus we have minimum $111 \times 2^{32} \approx 2^{39}$ slid pairs, just within a shift of 221 bits of each other. There are much more slid pairs for longer shifts, but the equations would be much more complicated.

Nonexistence of Special Slid Pairs. We searched for slid pairs with additional constraints. The first type applies when the keys in the two key and IV pairs are the same for any clock-shift c: $([(K, IV_1), (K, IV_2)], c)$ and the second type applies when both times the same IV is used for any clock-shift c: $([(K_1, IV), (K_2, IV)], c)$. In both cases the fixed second key or IV leads to 80 additional equations which account for the occurrence of all 80 key or IV bits. Thus we have overdefined systems with 208 equations and 160 variables. For both types the systems are not likely to be solvable for any reasonably small amount of shift. As a result of the 48 extra equations the chance for such system to have a solution is about 2^{-48}. We computed that for the first 31 instances (clock-shifts 111 up to 142) these systems have no solution.

[4] We guessed these bits to get a solution from Magma in a reasonable amount of time.

4 Conclusion

In this paper we have described sliding properties of Salsa20 and Trivium which lead to distinguishing, key recovery and related-key attacks on these ciphers. We also show that Salsa20 does not offer 256-bit security due to a simple birthday attack on its 384-bit state. Since the likelihood of falling in our related key classes by chance is relatively low (2^{256} out of 2^{384} for Salsa20, 2^{39} out of 2^{80} for Trivium) these attacks do not threaten most of the real-life usage of these ciphers. However designer of protocols which would use these primitives should be definitely aware of these non-randomness properties, which can be exploited in certain scenarios.

References

1. J.-P. Aumasson, S. Fischer, S. Khazaei, W. Meier and C. Rechberger. *New Features of Latin Dances: Analysis of Salsa, ChaCha and Rumba* To appear in Proceedings of Fast Software Encryption 2008 (FSE 2008) , LNCS, Lausanne, Switzerland, February 10-13, 2008. Full version as IACR eprint, http://eprint.iacr.org/2007/472
2. D. J. Bernstein. *Salsa20.* eSTREAM, ECRYPT Stream Cipher Project, Report 2005/025, 2005. http://www.ecrypt.eu.org/stream/
3. C. De Cannière and B. Preneel. *TRIVIUM - a stream cipher construction inspired by block cipher design principles.* eSTREAM, ECRYPT Stream Cipher Project, Report 2005/030, 2005. http://www.ecrypt.eu.org/stream/
4. P. Crowley. *Truncated differential cryptanalysis of five rounds of Salsa20.* SASC 2006 - Stream Ciphers Revisited, 2006.
5. eSTREAM: The ECRYPT Stream Cipher Project, http://www.ecrypt.eu.org/stream/
6. S. Fischer, W. Meier, C. Berbain, J.-F. Biasse, M. J. B. Robshaw. *Non-randomness in eSTREAM Candidates Salsa20 and TSC-4.* INDOCRYPT, volume 4329 of LNCS, pages 2-16. Springer, 2006.
7. Jin Hong. Discussion Forum. *certain pairs of key-IV pairs for Trivium*, created September 13, 2005 05:11PM. http://www.ecrypt.eu.org/stream/phorum/read.php?1,152
8. A. Maximov and A. Biryukov. *Two Trivial Attacks on Trivium.* SASC 2007 - The State of the Art of Stream Ciphers, 2007.
9. C. McDonald, C. Charnes and J. Pieprzyk. *Attacking Bivium with MiniSat.* eSTREAM, ECRYPT Stream Cipher Project, Report 2007/040, 2007. http://www.ecrypt.eu.org/stream/
10. H. Raddum. *Cryptanalytic Results on TRIVIUM.* eSTREAM, ECRYPT stream cipher project, Report 2006/039, 2006. http://www.ecrypt.eu.org/stream/
11. Y. Tsunoo, T. Saito, H. Kubo, T. Suzaki and H. Nakashima. *Differential Cryptanalysis of Salsa20/8.* SASC 2007 - The State of the Art of Stream Ciphers, 2007.
12. M. S. Turan and O. Kara. *Linear Approximations for 2-round Trivium.* SASC 2007 - The State of the Art of Stream Ciphers, 2007.
13. M. Vielhaber. *Breaking ONE.Fivium by AIDA an Algebraic IV Differential Attack.* Cryptology ePrint Archive, Report 2007/413, 2007

A Salsa20

A.1 System of equations for a slid pair

Word equations given by the equations $S' = F(S)$ and $X' = F(X)$.

1. $k'_0 = k_3 \oplus ((\sigma_0 + k_5) \lll 7)$
2. $k'_1 = c_0 \oplus ((k'_0 + \sigma_0) \lll 9)$
3. $k'_2 = k_5 \oplus ((k'_1 + k'_0) \lll 13)$
4. $\sigma_0 = \sigma_0 \oplus ((k'_2 + k'_1) \lll 18)$

5. $n'_0 = c_1 \oplus ((\sigma_1 + k_0) \lll 7)$
6. $n'_1 = k_6 \oplus ((n'_0 + \sigma_1) \lll 9)$
7. $k'_3 = k_0 \oplus ((n'_1 + n'_0) \lll 13)$
8. $\sigma_1 = \sigma_1 \oplus ((k'_3 + n'_1) \lll 18)$

9. $k'_4 = k_7 \oplus ((\sigma_2 + n_0) \lll 7)$
10. $c'_0 = k_1 \oplus ((k'_4 + \sigma_2) \lll 9)$
11. $c'_1 = n_0 \oplus ((c'_0 + k'_4) \lll 13)$
12. $\sigma_2 = \sigma_2 \oplus ((c'_1 + c'_0) \lll 18)$

13. $k'_5 = k_2 \oplus ((\sigma_3 + k_4) \lll 7)$
14. $k'_6 = n_1 \oplus ((k'_5 + \sigma_3) \lll 9)$
15. $k'_7 = k_4 \oplus ((k'_6 + k'_5) \lll 13)$
16. $\sigma_3 = \sigma_3 \oplus ((k'_7 + k'_6) \lll 18)$

17. $x'_1 = x_4 \oplus ((x_0 + x_{12}) \lll 7)$
18. $x'_2 = x_8 \oplus ((x'_1 + x_0) \lll 9)$
19. $x'_3 = x_{12} \oplus ((x'_2 + x'_1) \lll 13)$
20. $x'_0 = x_0 \oplus ((x'_3 + x'_2) \lll 18)$

21. $x'_6 = x_9 \oplus ((x_5 + x_1) \lll 7)$
22. $x'_7 = x_{13} \oplus ((x'_6 + x_5) \lll 9)$
23. $x'_4 = x_1 \oplus ((x'_7 + x'_6) \lll 13)$
24. $x'_5 = x_5 \oplus ((x'_4 + x'_7) \lll 18)$

25. $x'_{11} = x_{14} \oplus ((x_{10} + x_6) \lll 7)$
26. $x'_8 = x_2 \oplus ((x'_{11} + x_{10}) \lll 9)$
27. $x'_9 = x_6 \oplus ((x'_8 + x'_{11}) \lll 13)$
28. $x'_{10} = x_{10} \oplus ((x'_9 + x'_8) \lll 18)$

29. $x'_{12} = x_3 \oplus ((x_{15} + x_{11}) \lll 7)$
30. $x'_{13} = x_7 \oplus ((x'_{12} + x_{15}) \lll 9)$
31. $x'_{14} = x_{11} \oplus ((x'_{13} + x'_{12}) \lll 13)$
32. $x'_{15} = x_{15} \oplus ((x'_{14} + x'_{13}) \lll 18)$

Word equations given by the feedforward for the first key-stream words and for the second key-stream words.

33. $z_0 = x_0 + \sigma_0$
34. $z_1 = x_1 + k_0$
35. $z_2 = x_2 + k_1$
36. $z_3 = x_3 + k_2$
37. $z_4 = x_4 + k_3$
38. $z_5 = x_5 + \sigma_1$
39. $z_6 = x_6 + n_0$
40. $z_7 = x_7 + n_1$
41. $z_8 = x_8 + c_0$
42. $z_9 = x_9 + c_1$
43. $z_{10} = x_{10} + \sigma_2$
44. $z_{11} = x_{11} + k_4$
45. $z_{12} = x_{12} + k_5$
46. $z_{13} = x_{13} + k_6$
47. $z_{14} = x_{14} + k_7$
48. $z_{15} = x_{15} + \sigma_3$

49. $z'_0 = x'_0 + \sigma_0$
50. $z'_1 = x'_1 + k'_0$
51. $z'_2 = x'_2 + k'_1$
52. $z'_3 = x'_3 + k'_2$
53. $z'_4 = x'_4 + k'_3$
54. $z'_5 = x'_5 + \sigma_1$
55. $z'_6 = x'_6 + n'_0$
56. $z'_7 = x'_7 + n'_1$
57. $z'_8 = x'_8 + c'_0$
58. $z'_9 = x'_9 + c'_1$
59. $z'_{10} = x'_{10} + \sigma_2$
60. $z'_{11} = x'_{11} + k'_4$
61. $z'_{12} = x'_{12} + k'_5$
62. $z'_{13} = x'_{13} + k'_6$
63. $z'_{14} = x'_{14} + k'_7$
64. $z'_{15} = x'_{15} + \sigma_3$

A.2 System of equations for a column in general

The columnround in function $\boldsymbol{\mathcal{F}}$ applies to each column of the matrix a quarterround. All computations of a quarterround for one column are independent from the computations of the other three columns. If one takes together for one column the quarterround coming from $S' = \boldsymbol{\mathcal{F}}(S)$ the corresponding quarterround from $X' = \boldsymbol{\mathcal{F}}(X)$ and the feedforward one gets a system with 16 equations shown below. We assume all 16 variables are unknown.

$$
\begin{array}{ll}
s'_1 = s_4 \oplus ((s_0 + s_{12}) \lll 7) & x'_1 = x_4 \oplus ((x_0 + x_{12}) \lll 7) \\
s'_2 = s_8 \oplus ((s'_1 + s_0) \lll 9) & x'_2 = x_8 \oplus ((x'_1 + x_0) \lll 9) \\
s'_3 = s_{12} \oplus ((s'_2 + s'_1) \lll 13) & x'_3 = x_{12} \oplus ((x'_2 + x'_1) \lll 13) \\
s'_0 = s_0 \oplus ((s'_3 + s'_2) \lll 18) & x'_0 = x_0 \oplus ((x'_3 + x'_2) \lll 18)
\end{array}
$$

$$
\begin{array}{ll}
z_0 = x_0 + s_0 & z'_0 = x'_0 + s'_0 \\
z_4 = x_4 + s_4 & z'_1 = x'_1 + s'_1 \\
z_8 = x_8 + s_8 & z'_2 = x'_2 + s'_2 \\
z_{12} = x_{12} + s_{12} & z'_3 = x'_3 + s'_3
\end{array}
$$

This system can be reduced to four equations. In the first equation two variables must be guessed to solve it. In the remaining three equations always two variables are known either the guessed s-variable or the calculated s'-variable. Thus they can be solved without guessing any more variables.

$$
\begin{array}{l}
z'_1 = [(z_4 - s_4) \oplus ((z_0 - s_0 + z_{12} - s_{12}) \lll 7)] + [s_4 \oplus ((s_0 + s_{12}) \lll 7)] \\
z'_2 = [(z_8 - s_8) \oplus ((z'_1 - s'_1 + z_0 - s_0) \lll 9)] + [s_8 \oplus ((s'_1 + s_0) \lll 9)] \\
z'_3 = [(z_{12} - s_{12}) \oplus ((z'_2 - s'_2 + z'_1 - s'_1) \lll 13)] + [s_{12} \oplus ((s'_2 + s'_1) \lll 13)] \\
z'_0 = [(z_0 - s_0) \oplus ((z'_3 - s'_3 + z'_2 - s'_2) \lll 18)] + [s_0 \oplus ((s'_3 + s'_2) \lll 18)]
\end{array}
$$

Therefore the system of equations for one column with complete unknown variables can be solved by guessing only two variables.

B Trivium

Two examples for slid pairs written in hexadecimal numbers are given below. The bits for keys and IVs are ordered from 1 to 80 but in the key and IV setup they are used the other way around.

$[(K_1, IV_1), (K_2, IV_2), 111]$

K_1 :	70011000001E00000000
IV_1 :	AF9D635BCEF9AE376CF7
key-stream[1]:	2E7338CB404272ABEE3F7BEC2F8D
	55E27536D29AFFFF15DFDFD711AECC78D13D7B61 ...
K_2 :	780000001DA2000003C1
IV_2 :	1DF35CF6D4FFF4E3A6C0
key-stream:	55E27536D29AFFFF15DFDFD711AECC78D13D7B61 ...

$[(K_3, IV_3), (K_4, IV_4), 112]$

K_3 :	02065B9C001730000000
IV_3:	609FC141828705160A3C
key-stream:	A48BCA9143685F03DE646F83AB52
	88BC9542798983349A959503E63BBF29C4755DE6 ...
K_4 :	B98000003E96E70005CE
IV_4 :	2B7C1483BC476A62E4CB
key-stream:	88BC9542798983349A959503E63BBF29C4755DE6 ...

[1] The shift is $c = 111$ which means the first 111 bits are a prefix. When rewriting these prefix from hexadecimal to binary numbers the leading zero must be omitted because 111 is not a multiple of 4.