Fault Analysis of the ChaCha and Salsa Families of Stream Ciphers

Arthur Beckers, Benedikt Gierlichs, and Ingrid Verbauwhede

imec-COSIC KU Leuven
Kasteelpark Arenberg 10, 3001 Leuven-Heverlee, Belgium
`firstname.lastname@esat.kuleuven.be`

Abstract. We present a fault analysis study of the ChaCha and Salsa families of stream ciphers. We first show that attacks like differential fault analysis that are common in the block cipher setting are not applicable against these families of stream ciphers. Then we propose two novel fault attacks that can be used against any variant of the ciphers. We base our attacks on two different fault models: the stuck-at fault model and the biased fault model. Each of them is exploited differently by the attacker. If the attacker knows the plaintexts and the ciphertexts both fault models can be successfully exploited. If the ciphers operate on fixed yet unknown plaintexts only the biased fault model can be successfully exploited. We evaluate exemplary attacks using both models in simulation. Their low complexity confirms that they are practical. To the best of our knowledge these are the first fault attacks against ChaCha and Salsa that do not require faults in the control flow (e.g. instruction skip).

Keywords: ChaCha, Salsa, stream cipher, fault analysis

1 Introduction

We analyse fault attacks that target implementations of the Salsa [1] and ChaCha [2] families of stream ciphers in all their variants. Both stream cipher families are designed by Dan J. Bernstein. The Salsa family of stream ciphers was designed in 2005 for the eSTREAM project. In 2009 Salsa20/12 was selected as one of the stream ciphers for the eSTREAM portfolio [3]. In 2008 the ChaCha family was proposed as a variant on the Salsa family. The goal of ChaCha was to increase the diffusion within a single round while maintaining the same performance as the Salsa family of ciphers. ChaCha is becoming more widely deployed since Google implemented it in its openssl cipher suite [4]. Salsa20 and ChaCha support both 128- and 256-bit keys. We focus on the ChaCha and Salsa ciphers but the proposed techniques can be applied to every cryptographic function that has the same feed-forward and modular addition structure.

Fault attacks were introduced by Boneh et al. [5] in 1996 on RSA signature generation with CRT. Since then numerous attacks on many cryptographic primitives were published. A fault attack can be split up in three components. The injection method, the resulting fault model and the exploitation of the fault. The

most common injection methods include introducing glitches in the clock [6, 7] or power supply [8], laser fault injection [9, 10] and EM fault injection [11, 12]. Each of these methods will result in a specific fault model that will strongly depend on the targeted device, making it hard to predict the outcome of a certain fault injection method. For our attacks we assume that a fault injection method is capable of delivering the fault model that we need and build up our attack from there.

The proposed attacks do not require fault injection in the control flow of the cipher [25], but they aim to exploit faults injected in intermediate values of the ciphers operation.

1.1 Background

Although ChaCha and Salsa are software oriented stream ciphers their operation resembles that of a block cipher in counter mode. Therefore we look at common fault analysis techniques for both block and stream cipher implementations.

Most stream ciphers have an initial state that holds the key and an initialization vector (IV). In an initialisation phase the initial state is randomized such that the key stream resulting from the internal state can not be used to recover the initial state. After the initialization phase a key stream is produced starting from the randomized state by updating the state continuously. For the randomization stream ciphers often use feedback shift registers (FSRs).

Fault attacks on stream cipher implementations generally try to recover the initial state by either targeting the initialization phase [13], [14] or by forcing the cipher in an unintended state [15], [16]. Often cryptanalytic techniques are applied to the gathered faulted outputs in order to recover the initial state.

The ChaCha and Salsa families of stream ciphers do not rely on FSRs for their operation. They instead rely mainly on a hash function to generate a pseudorandom key stream. Therefore common fault attack techniques for stream ciphers are not applicable to ChaCha and Salsa.

Common fault attacks on block cipher implementations are differential fault analysis (DFA) [17, 18], collision fault analysis (CFA) [19], ineffective fault analysis (IFA) [20] and safe-error analysis (SEA) [21]. All these methods require multiple encryptions of chosen messages under a fixed key. In the stream cipher setting this would be possible only if the adversary was able to re-initialize the stream cipher with the same secret key, nonce and counter values. But of course this would be against any reasonable stream cipher specification, since in such a scenario also much more powerful cryptanalytic attacks would be possible.

Recently, statistical attacks against block cipher implementations have been proposed that do not require an adversary to have any control over the cipher's inputs [22–24]. They can be used to target block ciphers that operate for instance in counter mode. These attacks rely on fault injections in one of the last rounds of the cipher combined with statistical techniques to retrieve the key. The key retrieval is possible due to the addition of a round key in the last rounds. This key addition (XOR) results in an exploitable relationship between the faulted output and the key. ChaCha and Salsa, however, have a different structure and

use a hash function and not a block cipher as their source of randomness making the attacks proposed in [22–24] not applicable.

2 The ChaCha and Salsa ciphers

In this section we describe mainly the ChaCha family of stream ciphers since the different attacks are simulated using ChaCha. The design of Salsa differs only slightly, the general structure remains the same. We detail the differences at the end of this section, but they do not influence the proposed fault attacks.

2.1 ChaCha

The ChaCha stream cipher operates on a state matrix M that contains 16 words. Each word is 32 bits wide. M is initialized as shown in Eq. (1).

$$M = \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix} \Leftarrow \begin{pmatrix} c & c & c & c \\ k_1 & k_2 & k_3 & k_4 \\ k_5 & k_6 & k_7 & k_8 \\ cnt & cnt & n & n \end{pmatrix} \quad (1)$$

The first four words denoted by c are constants given by the ChaCha specification. The next eight words are the key words denoted by k_i ($i = 1, \dots, 8$). The last row is made up out of two counter words cnt and two nonce words n . ChaCha works with both 128- and 256-bit key lengths. In the 256-bit case the key words are copied in order in $(k_1, \dots, k_8)$ of M . In the 128-bit case the key words are copied in order in both the second and third row of M , $(k_1, \dots, k_4) = (k_5, \dots, k_8)$.

M is put through a hash function which has as its basic building block the quarter-round function (see Algorithm 1). The quarter-round function is made up out of ARX operations (addition modulo 2^{32} , rotation, exclusive-or) and is the source of non-linearity for the ChaCha stream cipher.

Algorithm 1: Quarter-round function

```

1  $(a, b, c, d) = quarterround(a, b, c, d)$ 
2 begin
3    $a \leftarrow a \boxplus b$ 
4    $d \leftarrow (a \oplus d) \lll 16$ 
5    $c \leftarrow c \boxplus d$ 
6    $b \leftarrow (c \oplus b) \lll 12$ 
7    $a \leftarrow a \boxplus b$ 
8    $d \leftarrow (a \oplus d) \lll 8$ 
9    $c \leftarrow c \boxplus d$ 
10   $b \leftarrow (c \oplus b) \lll 7$ 
11  return  $a, b, c, d$ 
12 end
```

One round of the hash function is built from four quarter-rounds. The number of rounds used in the hash function of ChaCha is a trade-off between security and efficiency. r is the number of rounds used in the hash function. The rounds of ChaCha are invertible therefore we need to modularly add M to the round output to form the hash. We focus on ChaCha20 but the proposed attacks are independent of the number of rounds used.

Figure 1 shows the general structure of ChaCha. M is permuted using r rounds. After permuting M , the output of the round function (R) is modulo added to M which results in the key stream (K). K is xored with the plain text (PT) or cipher text (CT) for en- or decryption. After every hashing of M the counter cnt in M must be incremented (Eq. 1, [2]).

Fig. 1. Structure of ChaCha family of stream ciphers.

2.2 Salsa

Salsa differs in the structure of M and has a slightly different round function design. The Salsa state matrix differs from the ChaCha state matrix in the ordering of the key, constant, nonce and counter words. The rounds in the Salsa hash function use a different quarter-round function and order the inputs of their quarter-rounds differently. We refer to [1, 2] for a detailed overview of the differences.

3 Fault attacks on ChaCha and Salsa

We describe the attacks on ChaCha, application to Salsa is straightforward. The attacks are explained from an encryption point of view. Due to the symmetric nature of ChaCha an attack demonstrated on encryption is equally valid for decryption.

Our attacks aim to recover the secret key $k_1, \dots, k_8$ by exploiting the modulo addition of the corresponding elements of M and R (m_i and r_i ($i = 4, \dots, 11$)). In the rest of the paper we detail the attacks on one element of M (one key word k_i) but for readability we will write M , R and K without index. Application to the other key words is straightforward. For our attacks we assume the adversary has no control over M . The adversary is thus not capable of resetting the cipher

to a previous state by manipulating the counter or nonce words. Note that M is constant for the relevant m_i , $i = 4, \dots, 11$.

In normal operation R is uniformly distributed and therefore K does not reveal any information about M . The general idea for both attacks is to change R 's uniform distribution into a distribution that can be exploited for key recovery.

Our attacks make use of two different distributions. Firstly the distribution of the values of R . Under normal operation R has a discrete uniform distribution, every value is equally likely to occur. Secondly the distribution of the Hamming weight of the values of R . The Hamming weight distribution of a discrete uniform distribution is a binomial distribution.

The distribution of R will change due to the fault injection. The resulting distribution of R and subsequently K and CT will depend on the type of fault. Throughout the remainder of the paper we will note faulted values with an asterisk (R^*, K^*, CT^*). To perform the proposed attacks an attacker has to make guesses for M , R^* and K^* . These guesses are denoted with an apostrophe (M', R', K').

The location of the fault injection might be the registers storing the output of the round function or any arithmetic operation in one of the last quarter-round functions. We assume a fault is injected somewhere in the last round resulting in the desired fault model, but we do not make an assumption on which exact operation is faulted. It is *important* to note that we do not use the resulting value of any single fault but the changed *distribution* of R to retrieve the key.

3.1 Attack for the biased fault model

R is a 32-bit word. The probability that any bit X in R is one or zero is equal: $P(X = 1) = P(X = 0) = 0.5$. With the biased fault model we assume that a fault injection alters these probabilities. The probability distribution could for instance become $P(X = 1) = 0.75, P(X = 0) = 0.25$ for every bit of R^* . This fault model has been used previously in [22, 26, 27]. The influence of such a bias on the distribution of an 8-bit word is illustrated in Figure 2a. The biased distribution is plotted in blue, here $P(X = 1) = 0.35, P(X = 0) = 0.65$. The uniform distribution is plotted in red. The bias introduced on the bit level alters the distribution on the word level. We get a distribution that is skewed towards the lowest (in the example) or highest value of the distribution depending on the bias at bit level. The bias changes the distribution of Hamming weights on the word level too. It is *important* to note that the biased distribution on word level is the consequence of a bias introduced at bit level.

As explained in Section 2, R^* gets modulo added to M resulting in K^* (Figure 1). Recall that we will recover the m_i one word at a time but do not write the index. The addition of R^* and a constant modulo a power of two merely rotates the biased distribution of R^* . Figure 2b illustrates the effect of modulo adding 170 to the biased distribution from Figure 2a.

In order to retrieve the key we make use of the distribution of the Hamming weight of R^* . Note that this approach does *not* require any knowledge about the effect of any single fault injection or the amount of bias. It requires only that

Fig. 2. Biased output distribution.

the probability distribution of the bits of R^* is biased and that the bias is equal for all the bits of R^* . We sample the biased distribution of K^* and subtract all possible key guesses from the key stream values in order to compute hypothetical R' . For each key guess, we look at the resulting distribution of Hamming weights of R' . The correct key guess will be the one for which the mean Hamming weight μ_{HW} of R' is maximal if $P(X = 1) > 0.5$ or minimal if $P(X = 1) < 0.5$.

The mean Hamming weight of a distribution can be calculated using Eq. (2). W denotes the values of a word of size n , $W = [0, \dots, 2^n - 1]$. $P(w)$, $w \in W$ is the probability that a value w of W occurs. Under a biased bit model this probability is given by Eq. (3). $Pr(q, n, p)$ (Eq. 4) is the binomial probability mass function and is used to calculate the probability that a certain HW occurs. p is the probability that a bit is 1 and thus p is equal to the amount of bias.

$$\mu_{HW} = \sum_{w=0}^{2^n-1} HW(w) \times P(w) \quad w \in W \quad (2)$$

$$P(w) = \frac{Pr(HW(w), n, p)}{\binom{n}{HW(w)}} \quad w \in W \quad (3)$$

$$Pr(HW, n, p) = \binom{n}{HW} p^{HW} (1-p)^{n-HW} \quad (4)$$

Eq. (3) shows that the probability that a value w of W occurs depends solely on its HW and the bias p . If $p \neq 0.5$ the probabilities of $P(w)$ have an ascending or descending order depending on the Hamming weight and p as shown by Eq. (5) and (6).

$$\frac{Pr(0, n, p)}{\binom{n}{0}} > \frac{Pr(1, n, p)}{\binom{n}{1}} > \dots > \frac{Pr(n, n, p)}{\binom{n}{n}} \quad \text{if } p < 0.5 \quad (5)$$

$$\frac{Pr(0, n, p)}{\binom{n}{0}} < \frac{Pr(1, n, p)}{\binom{n}{1}} < \dots < \frac{Pr(n, n, p)}{\binom{n}{n}} \quad \text{if } p > 0.5 \quad (6)$$

When we make a guess for the key we look at μ_{HW} of $R' = R^* \boxplus M \boxminus M'$ with M' our key guess and R' our resulting guess for R^* . We can distinguish the correct key guess since μ_{HW} is maximal for $p > 0.5$ or minimal for $p < 0.5$ if the distribution of the HW is binomial and thus if $R' = R^*$. The μ_{HW} is maximal for $p > 0.5$ since the probability that a value occurs rises with its Hamming weight (Eq. 6). If we permute the labels of a binomial distribution as it happens for a wrong key guess, for instance by swapping two elements with different Hamming weights, we inevitably assign a lower probability to the value with a higher Hamming weight and a higher probability to the value with a lower Hamming weight, see Eq. (7). i and j in Eq. (7) are the Hamming weights of two $w \in W$ with $i \neq j$.

$$\frac{Pr(i, n, p)}{\binom{n}{i}} \times i + \frac{Pr(j, n, p)}{\binom{n}{j}} \times j > \frac{Pr(i, n, p)}{\binom{n}{j}} \times j + \frac{Pr(j, n, p)}{\binom{n}{i}} \times i \quad (7)$$

The consequence of the swap will be a lowering of μ_{HW} . We provide an example in Table 1. A similar reasoning can be made for the case of $p < 0.5$.

value	00	01	10	11
HW	0	1	1	2
prob	0.09	0.21	0.21	0.49
$\mu_{HW} = 0.3725$				

→ swap 10 and 11 →

value	00	01	11	10
HW	0	1	2	1
prob	0.09	0.21	0.21	0.49
$\mu_{HW} = 0.28$				

Table 1. Example with $P(X = 1) = 0.7$ and $P(X = 0) = 0.3$.

We can thus distinguish a skewed binomial distribution ($p \neq 0.5$) from a skewed binomial distribution that is permuted. If the permutation however maps onto a skewed binomial distribution it will be indistinguishable from the original one.

An example is plotted in Figure 3. The red curve shows the Hamming weight distribution if the 8-bit values are uniformly distributed, i.e. when there is no bias. The blue curve shows the Hamming weight distribution for the correct key guess and the grey curves show the Hamming weight distributions for all wrong key guesses. The mean of the Hamming weight of the distribution for the correct key guess lies farthest from the mean of the Hamming weight of the uniform distribution. Hence the correct key guess can be easily identified.

Different distinguishers can be used to identify the correct key guess. In [24] the authors propose to use Maximum likelihood, the mean Hamming weight or the Squared Euclidean Imbalance(SEI) as distinguishers. The Maximum likelihood approach requires an exact knowledge of the bias and is therefore discarded as an identifier. We will use the SEI and a standard two sample t-test (Eq. (8)) as distinguishers. The use of the t-test is similar to the mean Hamming weight distinguisher.

Fig. 3. Hamming weight distribution of K .

For the standard two sample t-test we can compute the mean and standard deviation of the uniform distribution using Eq. (9). n is the word size in bits. z is the number of samples, s is the standard deviation of the Hamming weight of the sampled values and $\bar{x}$ is the mean Hamming weight of the sampled values.

$$t = \left| \frac{\mu - \bar{x}}{\sqrt{\frac{\sigma^2 + s^2}{z}}} \right| \quad (8)$$

$$\mu = \sum_{i=0}^n \left[\frac{\binom{n}{i}}{2^n} i \right], \sigma = \sqrt{\sum_{i=0}^n \left[\frac{\binom{n}{i}}{2^n} (i - \mu)^2 \right]} \quad (9)$$

In order to compute the SEI we use Eq. (10). $Pr(i, n, 0.5)$ is the binomial probability mass function from Eq. 4 .

$$SEI = \sum_{i=0}^n \left[\frac{\#((HW)^z = i)}{z} - Pr(i, n, 0.5) \right]^2 \quad (10)$$

When performing an actual attack the sampling of the distribution becomes injecting a fault and collecting the faulted output. We can exploit the biased distribution at two positions either at K or at CT . When attacking at position K we need to know both PT and CT^* for the attack to succeed. When attacking at position CT we do not need to know PT but we require PT to be constant, since having a random unknown PT would be the equivalent of a one time pad making it impossible to retrieve the key.

Attack at position K Assume that all bits of K^* are equally biased. We split the 32-bit word K^* in four 8-bit chunks that all have a biased distribution. We do this to reduce the computational complexity. By splitting up the 32-bit word in h chunks we need to compute only $h2^{32/h}$ t-tests instead of 2^{32} . We are of

course not restricted to a chunk size of 8-bits. The presence of a carry when splitting up a word will influence the bias but does not need to be accounted for by the attacker.

For every chunk we make a guess M' for the corresponding part of the key in M and compute $R' = PT \oplus CT^* \boxminus M' = K^* \boxminus M' = R^* \boxplus M \boxminus M'$. The only permutations performed on R^* are rotations due to the modulo additions. These rotations are not able to map R^* onto an R' such that R' has a probability distribution as described by Eq. (3) except for the one corresponding to the correct key guess. Therefore the known $PT \ CT^*$ case will yield a single solution.

Attack at position CT We can perform a similar attack if we have a fixed unknown PT . The main difference is that we have to make a guess PT' on the value of PT too. We compute $R' = CT^* \oplus PT' \boxminus M' = R^* \boxplus M \oplus PT \oplus PT' \boxminus M'$, with PT' and M' our guesses for plaintext and key respectively.

The combination of xor operations and modulo additions maps R^* onto an R' that results in a binomial distribution of the Hamming weight in four cases. The attack will thus recover the correct guess and three false positives. If $PT \oplus PT' = 0$ we have the correct key guess, if $PT \oplus PT' = 2^n/2$ we get a false positive that has a distribution that is skewed according to the same bias as R^* . If $PT \oplus PT' = 2^n - 1$ or $PT \oplus PT' = 2^n/2 - 1$ we get a distribution for R' that is skewed in the opposite direction ($P(X = 1) = 1 - \text{bias}$). Recall that n is the size of the chunks in which we split the 32-bit word.

We can group the four key candidates in two sets by looking at the means of the probability distribution of the Hamming weight. A set with the bits biased towards zero and a set with the bits biased towards one. If we split up all the 16 words in 8-bit chunks we end up with 2×2^{32} key candidates for a 256-bit key. This reduction of the key space is sufficient to find the 256-bit key by exhaustive search.

3.2 Attack for the stuck-at fault model

The stuck-at fault model assumes that some bits of R^* (see Figure 1) are fixed to either one or zero. Due to the fixed bits the set of values R^* can attain will be limited. We look at a single 8-bit chunk of a 32-bit word R^* . Due to the fixed value of some of the bits certain values of R will no longer appear. Figure 4 shows a possible distribution for an 8-bit word with two bits stuck at zero. If we have enough faulty outputs we can compute the histogram of this distribution and deduce the amount of bits that are stuck at zero by counting the empty bins. The number of faults we need is determined by the number of bits that are stuck. For instance if we fault an 8-bit chunk and two bits get stuck at zero then there are 2^6 possible values left. In order to make a correct guess for the number of bits that are stuck we need to draw sufficiently many samples in order to fill the remaining bins.

The location of the empty bins in the histogram allows us to reduce the key space. We can again exploit the effect at two positions, either at K or at CT .

Fig. 4. Stuck-at fault model output distribution.

As with the biased fault model $K = M \boxplus R$. This modular addition modulo a power of two again only rotates the distribution. In order to reduce the key space we subtract our key guess modularly from K^* and obtain a hypothetical R' . Based on our reduction criteria we keep or discard the key guess. By iterating over all possible key guesses we are able to reduce the key space.

When reducing the key space we use two reduction criteria. The first one is the distribution of the Hamming weight of the words of R' for the different key guesses and the second is the location of the stuck bits. For the first reduction criterion we look at the Hamming weight distribution for a certain key guess. There should not be any Hamming weight values higher than the number of bits that are not stuck (in the case of a stuck-at 0 model) or a Hamming weight value lower than the number of stuck bits (in the case of a stuck-at 1 model).

From the histogram of the faulty values (K^* or CT^*) we can determine the number of bits that are stuck by counting the empty bins. Besides the number of empty bins we also need to know whether the bits get stuck at either 0 or 1 since we cannot deduce this from CT^* or K^* .

For the second criterion we look at the consistency of the location of the stuck bits in R' . If the distribution that results from our key guess does not have a fixed position for its stuck bits we can discard the key guess. The number of key candidates that remain after applying the reduction criteria depends on the number of bits that are not stuck and on their location.

As an example of the application of the reduction criteria we consider the following case. We have the distribution of a 4-bit chunk of R^* that has 2 bits stuck at 0, thus only 4 bins of the distribution of R^* are filled. We look at K^* and make three guesses for the key. As a result of these guesses we get three distributions R' which have the following bins filled, (11, 6, 1, 4), (5, 6, 4, 12) and (8, 12, 4, 0). If we convert these to binary we get, (1011, 0110, 0001, 0100), (0101, 0110, 0100, 1100) and (1000, 1100, 0100, 0000). The first key guess gets rejected since it fails the first reduction criteria, 11 has a Hamming weight of three while the maximum Hamming weight is two. The second key guess passes the first reduction criteria but fails the second. For the second key guess the location of the stuck bits is not consistent since there are three bit positions that have both a 1 and a 0, bit

one, three and four. The last key guess results in a distribution R' that fits both criteria.

The exploitation of the stuck-at model is deterministic in nature. Using our reduction criteria we are certain that we do not discard valid key guesses. When we use a fixed but unknown PT with the stuck-at model we have to guess the values of PT and M .

Guessing full 32-bit words would require too many faulty outputs since we have to fill the entire histogram. Therefore we split the words of K^* or CT^* into smaller chunks. The chunks are not completely independent since there might be a carry propagating from one chunk to the other. The presence of a carry might reduce the efficiency of the attack. The carry bit will potentially increase the Hamming weight observed by the attacker by one. The attacker can not and does not have to account for this since the attack remains deterministic in the presence of a carry.

The number of faults needed depends on the number of bits that are stuck and the number of chunks we divide the 32-bit words in. The probability that any bin that will not stay empty due to stuck bits has at least one value is given by Eq. (11). s is the number of possible output values (2 to the power of the number of non stuck bits), f is the number of faults and b is the probability that every bin in the histogram that will not stay empty due to stuck bits has at least one value. We can simplify this equation to Eq. (11) in order to make an estimate of the number of faults needed to fill the histogram. Table 2 shows an example. If we have an 8-bit chunk which has 4 non-stuck bits, then we need 149 faults to have a probability of 99.9% that there is at least one value in every bin of the histogram.

$$b = 1 - \left[\sum_{i=1}^{s-1} \binom{s}{i} \left(\frac{s-i}{s} \right)^f (-1)^{i+1} \right] \implies f = \frac{\log \frac{1-b}{s}}{\log \frac{s-1}{s}} \quad (11)$$

Table 2. Number of faults to fill the non-stuck bins (b=99.9%)

# non-stuck bits	1	2	3	4	5	6	7	8
# faults	11	29	67	149	326	702	1500	3181

When dealing with larger words such as the 32-bit words used in ChaCha we need to split up the words in smaller chunks and apply the reduction criteria described above to each of the individual chunks. The simplest approach is dividing the 32-bit word in four 8-bit chunks and applying the reduction criteria to each chunk individually. An alternative approach is to divide the 32-bit word in overlapping chunks, applying the reduction criteria to every chunk. Each chunk will result in a subset of key candidates. By looking at the intersection of the different key sets we end up with a single set of key candidates for the 32-bit word. The sizes of the different chunks do not have to be constant and can be chosen by the attacker.

4 Simulation Results

For our simulations we assume that with each fault injection an entire 32-bit word is faulted according to the assumed fault model. We operate the cipher as described in the specification. For our simulations we use a random key and nonce for M . The counter is also incremented for every encryption thus we have a continuously varying and random R^* .

4.1 Biased fault model

For our simulation for both the known PT and fixed unknown PT case we set the bias of the bits at $P(X = 1) = 0.75$, $P(X = 0) = 0.25$. We split up the 32-bit word in 8-bit chunks and apply the technique described in Section 3.1 to each chunk.

In the following we report results for a single 8-bit chunk. In the simulated scenario where all bits of the word are equally affected by the faults the average number of faults needed to recover 8-bits or an entire word of the key are the same. Thus the total amount of faults needed to recover the key equals the number of faults needed to recover the key from an 8-bit chunk times the number of key words.

Figure 5 shows the result of the two sample t-test for the different key guesses in case we have a known PT . The correct key guess is depicted in black. The t-scores for wrong key guesses do not converge towards zero since the distribution of hypothetical R' based on a wrong key guess will not be uniform but just the biased distribution rotated over a number of positions equal to the difference between the wrong and the correct key guess. The distribution for a wrong key guess will therefore also be different from the Hamming weight distribution of a uniform distribution.

Fig. 5. Result of biased fault attack for known PT and CT^* ; t-test values on y-axis in log scale, number of faults on x-axis.

The number of faulty outputs needed in order to retrieve the key is determined by the amount of bias there is on R^* . Table 3 shows the average number

of faulty outputs needed for different biases and distinguishers. Once the distribution is biased the attack always succeeds given sufficiently many faulty outputs. We can see that the t-test consistently requires less faults than the SEI method to distinguish the correct key guess.

Table 3. bias vs. faults for known PT and CT^*

bias	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%	100%
# faults t-test	/	400	115	50	27	11	7	5	5	5	/
# faults SEI	/	570	173	80	46	35	31	26	36	78	/

For the case where we have a fixed but unknown PT we have to make a guess for both the PT and M . We thus need more faults in order to distinguish the correct guess. Table 4 shows the average number of faults needed. The needed number of faults goes back up if we have a bias $> 85\%$. This is due to the large probability attributed to singular values with a high or low Hamming weight. If for a wrong key guess one of those values also maps to a value with a high or low Hamming weight it will take more faults to distinguish these wrong guesses with certainty from the correct one. When we use SEI as a distinguisher we need significantly more faults. When the bias becomes larger than 85% the number of faults becomes too large to be considered practical.

The result of the two sample t-test between the Hamming weight of one of the 8-bit chunks and the uniform distribution can be seen in Figure 6. The four possible key, PT pairs are situated in the two lowest curves. Due to the false positive and mirrored results we have to do a brute-force search over the remaining key candidates. As for the known PT case the number faulty outputs needed to retrieve the key will depend on the amount of bias.

Table 4. bias vs. faults for fixed but unknown PT

bias	50%	55%	60%	65%	70%	75%	80%	85%	90%	95%	100%
# faults t-test	/	1356	420	158	100	77	65	63	78	134	/
# faults SEI	/	2235	548	367	329	517	877	11720	>20000	>20000	/

4.2 Stuck at fault model

For the stuck at fault model we use the techniques described in Section 3.2 to reduce the key space. The average number of key bits we can retrieve depends on the number of bits that are stuck. Figure 7 shows the number of key bits we can recover in a 32-bit word versus the percentage of bits that are stuck. The number of faults needed for this attack is dependant on the number of stuck bits and the chosen chunk size and can be estimated using Equation 11. Section 3.2

Fig. 6. Result of biased fault attack for fixed unknown PT ; t-test values on y-axis in log scale, number of faults on x-axis.

suggests two ways to apply the reduction criteria. We either split up the word in chunks and apply the reduction criteria on the individual chunks or we run the reduction criteria iteratively on all possible subsets of chunks. First we consider the known PT CT^* case. The green curve shows the average number of bits that can be recovered if we split the 32-bit word in individual 8-bit chunks. A better result, shown by the blue line, can be achieved if we apply the iterative algorithm. The brown curve shows the result for a fixed but unknown PT for individual 8-bit chunks. The curve for the iterative algorithm applied to a fixed but unknown PT is plotted in pink.

Fig. 7. Key space reduction by the stuck at attack.

In the known PT CT^* case the number of unknown bits converges towards zero as the number of stuck bits increases. In the fixed but unknown PT case the number of unknown bits has a minimum when half of the bits are stuck. Intuitively this can be understood as follows. In the known PT CT^* case $K^* = M \boxplus R^*$. The distribution of R^* will get sparser, the histogram will have a lot

of empty bins, if more bits are stuck. Thus the key words of M will be easier to recover the more bits are stuck at one or zero. In the fixed but unknown PT case $CT^* = (M \boxplus R^*) \oplus PT$. If the distribution of R^* is sparse then the distribution of CT will also be sparse. In order to retrieve the key words of M we have to iterate over all possible PT values. Iterating over all possible PT values equals making a guess for the distribution of K^* since $K^* = CT^* \oplus PT$. Since the distribution of CT^* is sparse there is a higher probability (increasing with the number of bits stuck once greater 50%) that a PT guess will result in a valid stuck at distribution that fits our reduction criteria. The key space will thus become larger as the number of stuck bits increases.

We assume an attacker can brute force a key space of 2^{60} and the key length is 256 bits. Under these assumptions we end up with an upper limit on the average number of remaining bits per word of 7.5 in order to make the key retrievable. This upper limit corresponds to the dashed red line in Figure 7. In the 128-bit case we can have up to 15 unknown bits per word. This upper limit is depicted by the dotted red line. Due to the replication of the key in the 128-bit case we can also fault both copies of the key and compare the two sets of remaining key candidates. This allows us to reduce the remaining key space even more since the correct key guess has to be present in both candidate sets. In the known PT CT^* case we can always recover the key if we have enough stuck bits. In the fixed but unknown PT case we are able to reduce the number of unknown bits per word to 17 in the best case. For an 128-bit key this would require an attacker capable of brute-forcing a key space of 2^{68} .

Having all bits stuck at zero or one is a special case of the stuck at model. When we know both PT and CT^* retrieving the key is trivial since then $M = K^*$ or $M = K^* \boxplus 1$. When we have a fixed unknown PT however we are xoring two unknown constants making it impossible to retrieve the key.

5 Conclusion

We presented a fault analysis study of the ChaCha and Salsa families of stream ciphers. These software oriented stream ciphers gained popularity since ChaCha was included in google's openssl cipher suite. We explained why classical fault analysis techniques are ineffective against these ciphers. Then we proposed two novel fault attacks using ChaCha as example. Targeting K proved to be effective and efficient for both the biased fault model and the stuck-at fault model. Attacks based on these models however assume an adversary that knows both PT and CT^* . In the fixed but unknown PT case only the attack based on the biased fault model will succeed when we attribute realistic computing powers to the adversary. The attack based on the stuck-at fault model is able to reduce the key space but not to identify the key uniquely. The proposed attacks all exploit a non-uniform distribution of R . An on-line randomness test that detects when the distribution of R^* differs significantly from uniform may be a possible countermeasure.

Acknowledgements. We thank the anonymous reviewers for their insightful comments. This work was supported in part by the Research Council KU Leuven: C16/15/058. In addition, this work was supported by the Flemish Government, FWO G.00130.13N, FWO G.0876.14N and Thresholds G0842.13; by the Hercules Foundation AKUL/11/19, and through the Horizon 2020 research and innovation programme under Cathedral ERC Advanced Grant 695305. Benedikt Gierlichs is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (FWO).

References

1. Bernstein, D.J.: Salsa20. ECRYPT Stream Cipher Project. 025 edn. (2005) <http://cr.yp.to/snuffle.html>.
2. Bernstein, D.J.: Chacha, a variant of salsa20. In: SASC 2008 The State of the Art of Stream Ciphers, ECRYPT (2008)
3. <http://www.ecrypt.eu.org/stream/>
4. Bursztein, E.: Speeding up and strengthening HTTPS connections for chrome on android. <https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html>
5. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology* **14**(2) (2001) 101–119
6. Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., Whelan, C.: The sorcerer’s apprentice guide to fault attacks. *Proceedings of the IEEE* **94**(2) (Feb 2006) 370–382
7. Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In: 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography. (Sept 2011) 105–114
8. Schmidt, J.M., Herbst, C.: A practical fault attack on square and multiply. In: 2008 5th Workshop on Fault Diagnosis and Tolerance in Cryptography. (Aug 2008) 53–58
9. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In Kaliski, B.S., Koç, ç.K., Paar, C., eds.: *Cryptographic Hardware and Embedded Systems - CHES 2002: 4th International Workshop Redwood Shores, CA, USA, August 13–15, 2002 Revised Papers*, Springer Berlin Heidelberg (2003) 2–12
10. Agoyan, M., Dutertre, J.M., Mirbaha, A.P., Naccache, D., Ribotta, A.L., Tria, A.: How to flip a bit? In: 2010 IEEE 16th International On-Line Testing Symposium. (July 2010) 235–239
11. Dehbaoui, A., Dutertre, J.M., Robisson, B., Tria, A.: Electromagnetic transient faults injection on a hardware and a software implementations of AES. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography. (Sept 2012) 7–15
12. Quisquater, J.J., Samyde, D.: Eddy current for Magnetic Analysis with Active Sensor. In: *Esmart 2002*, Nice, France. (9 2002)
13. Hojsík, M., Rudolf, B.: Floating fault analysis of trivium. In Chowdhury, D.R., Rijmen, V., Das, A., eds.: *Progress in Cryptology - INDOCRYPT 2008: 9th International Conference on Cryptology in India, Kharagpur, India, December 14–17, 2008. Proceedings*, Springer Berlin Heidelberg (2008) 239–250
14. Debraize, B., Corbella, I.M.: Fault analysis of the stream cipher snow 3G. In: 2009 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC). (Sept 2009) 103–110

15. Biham, E., Granboulan, L., Nguyen, P.Q.: Impossible fault analysis of RC4 and differential fault analysis of RC4. In Gilbert, H., Handschuh, H., eds.: Fast Software Encryption: 12th International Workshop, FSE 2005, Paris, France, February 21-23, 2005, Revised Selected Papers, Springer Berlin Heidelberg (2005) 359–367
16. Hoch, J.J., Shamir, A.: Fault analysis of stream ciphers. In Joye, M., Quisquater, J.J., eds.: Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings, Springer Berlin Heidelberg (2004) 240–253
17. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: CRYPTO. (1997) 513–525
18. Giraud, C.: DFA on AES. In Dobbertin, H., Rijmen, V., Sowa, A., eds.: Advanced Encryption Standard – AES: 4th International Conference, AES 2004, Bonn, Germany, May 10-12, 2004, Revised Selected and Invited Papers, Springer Berlin Heidelberg (2005) 27–41
19. Blömer, J., Krummel, V.: Fault based collision attacks on AES. In Breveglieri, L., Koren, I., Naccache, D., Seifert, J.P., eds.: Fault Diagnosis and Tolerance in Cryptography: Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006. Proceedings, Springer Berlin Heidelberg (2006) 106–120
20. Blömer, J., Seifert, J.P.: Fault based cryptanalysis of the advanced encryption standard (AES). In Wright, R.N., ed.: Financial Cryptography: 7th International Conference, FC 2003, Guadeloupe, French West Indies, January 27-30, 2003. Revised Papers, Springer Berlin Heidelberg (2003) 162–181
21. Yen, S.M., Joye, M.: Checking before output may not be enough against fault-based cryptanalysis. IEEE Transactions on Computers **49**(9) (Sep 2000) 967–970
22. Lashermes, R., Reymond, G., Dutertre, J.M., Fournier, J., Robisson, B., Tria, A.: A DFA on AES based on the entropy of error distributions. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography. (Sept 2012) 34–43
23. Dobraunig, C., Eichlseder, M., Korak, T., Lomné, V., Mendel, F.: Statistical fault attacks on nonce-based authenticated encryption schemes. In Cheon, J.H., Takagi, T., eds.: Advances in Cryptology – ASIACRYPT 2016: 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I, Berlin, Heidelberg, Springer Berlin Heidelberg (2016) 369–395
24. Fuhr, T., Jaulmes, E., Lomné, V., Thillard, A.: Fault attacks on AES with faulty ciphertexts only. In: Proceedings of the 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography. FDTC '13, IEEE Computer Society (2013) 108–118
25. Kumar, S.V.D., Patranabis, S., Breier, J., Mukhopadhyay, D., Bhasin, S., Chattopadhyay, A., Baksi, A.: A practical fault attack on arx-like ciphers with a case study on chacha20. Cryptology ePrint Archive, Report 2017/1074 (2017) <https://eprint.iacr.org/2017/1074>.
26. Ghalaty, N.F., Yuce, B., Schaumont, P.: Analyzing the efficiency of biased-fault based attacks. IEEE Embedded Systems Letters **8**(2) (June 2016) 33–36
27. Järvinen, K., Blondeau, C., Page, D., Tunstall, M.: Harnessing biased faults in attacks on ECC-based signature schemes. In: 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography. (Sept 2012) 72–82