Fault Analysis of Rabbit: Toward a Secret Key Leakage

Alexandre Berzati[1,2], Cécile Canovas-Dumas[1], Louis Goubin[2]

[1] CEA-LETI/MINATEC, 17 rue des Martyrs, 38054 Grenoble Cedex 9, France,
{alexandre.berzati,cecile.canovas}@cea.fr
[2] Versailles Saint-Quentin-en-Yvelines University,
45 Avenue des Etats-Unis, 78035 Versailles Cedex, France
Louis.Goubin@prism.uvsq.fr

Abstract. Although Differential Fault Analysis (DFA) led to powerful applications against public key [15] and secret key [12] cryptosystems, very few works have been published in the area of stream ciphers.
In this paper, we present the first application of DFA to the software eSTREAM candidate Rabbit that leads to a full secret key recovery. We show that by modifying modular additions of the next-state function, 32 faulty outputs are enough for recovering the whole internal state in time $\mathcal{O}\left(2^{34}\right)$ and extracting the secret key. Thus, this work improves the previous fault attack against Rabbit both in terms of computational complexity and fault number.

Keywords: Stream cipher, Rabbit, fault attacks, carry analysis.

1 Introduction

The stream cipher Rabbit has been selected in the final portfolio of the ECRYPT stream cipher project (eSTREAM) [13]. It was first presented at FSE 2003 [14], targeting both hardware and software environments. It has been selected as a software candidate for the third evaluation phase of the project.
Rabbit has a 128-bit key (also supports 80-bit key), 64-bit initialization vector (IV), and 513-bit internal state. Although it has been designed to be faster than commonly used ciphers, the level of security provided by this stream cipher has not been disregarded by the designers. Indeed, they made the efforts of a deep security analysis [13] and published a series of white papers [1,2,3,4,5,6] to prove the robustness of Rabbit to the well-known attacks (*i.e.* algebraic, correlation, guess-and-determine, differential). Until now, only two papers discussing the cryptographic security of Rabbit have been published. Both propose to exploit the bias of the core function "g". In [7], the function "g" was firstly shown to be unbalanced. The resulting distinguishing attack requires the analysis of 2^{247} keystream sub-blocks generated from random keys and IV's, which is higher than the complexity of the key exhaustive search (*i.e.* 2^{128}). The second article provides an improved distinguishing attack based on the use of the Fast Fourier Transform (FFT) for computing the exact Rabbit keystream bias. This

reduces the complexity of the distinguishing attack from $\mathcal{O}\left(2^{247}\right)$ to $\mathcal{O}\left(2^{97.5}\right)$ in the multi-frame extension [7]. Recently, the security of Rabbit in the context of faults has been discussed in [23]. Under a classical fault model, the authors demonstrated that the complete internal state can be recovered from 128 – 256 faults in $\mathcal{O}\left(2^{38}\right)$ steps. The attack also requires to precompute a table of size $\mathcal{O}\left(2^{41.6}\right)$ bytes. From our knowledge, it was the best known attack against Rabbit.

In this paper, we propose a new method to exploit faults against Rabbit implementations. We show that, if an attacker is able to perturb transiently modular additions in the next-state function, then he can recover the whole internal state and predict the keystream. The analysis can also lead to a full key recovery if the first two iterations of Rabbit are targeted.

We provide evidences that this attack does not only improves the previous result in terms of fault number but in terms of complexity. Indeed, from 32 faults injected according to our model, the attacker is able to recover the whole internal state in time $\mathcal{O}\left(2^{34}\right)$.

After a brief presentation of the Rabbit stream cipher, Sect. 3 provides an overview of previous fault attacks against implementations of stream ciphers. Then, we describe in Sect. 4 the fault model we have chosen and a complete differential analysis of the faulty keystreams. The final part of the paper provides the attack algorithm and an analysis of its performance.

2 The Stream Cipher Rabbit

2.1 Notations

The notations we use in this paper to describe Rabbit are extracted from the original description of the stream cipher presented at FSE 2003 [14].

- $\oplus$ denotes logical `XOR`,
- $\wedge$ denotes logical `AND`,
- $\vee$ denotes logical `OR`,
- $\ll$ and $\gg$ denote respectively left and right logical bit-wise shift,
- $\lll$ and $\ggg$ denote respectively left and right logical bit-wise rotation,
- $A^{[g..h]}$ denotes the part of the vector A from bit g to bit h,
- $A_{[k]}$ denotes the value of A mod k.

2.2 Description of Rabbit

Rabbit is a synchronous stream cipher. It takes as input a 128-bit secret key and a 64-bit public initialization vector (IV). For each iteration, it generates a 128-bit pseudo-random output block. This output block, usually referred as the keystream, is `XOR`-ed with a plaintext/ciphertext to perform the encryption/decryption.

The internal state of Rabbit is composed of 513 bits:

– Eight 32-bit state variables denoted by $(x_{j,i})_{0\leq j\leq 7}$ at iteration i,
– Eight 32-bit counters $(c_{j,i})_{0\leq j\leq 7}$,
– One counter carry bit $\phi_{7,i}$.

At epoch $i = 0$, the state variables and the counters are initialized with the Key Setup and IV Setup schemes. We recall neither Key Setup nor IV Setup schemes since our attack does not rely on the initialization process. Further details are provided in [14,13].
Then, for $i \geq 1$, state variables and counters are updated according to the following schemes. Each iteration produces 128 bits of the keystream.

Next-State Function.

$$\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i} \lll 16) + (g_{6,i} \lll 16)\\
x_{1,i+1} &= g_{1,i} + (g_{0,i} \lll 8) + g_{7,i}\\
x_{2,i+1} &= g_{2,i} + (g_{1,i} \lll 16) + (g_{0,i} \lll 16)\\
x_{3,i+1} &= g_{3,i} + (g_{2,i} \lll 8) + g_{1,i}\\
x_{4,i+1} &= g_{4,i} + (g_{3,i} \lll 16) + (g_{2,i} \lll 16)\\
x_{5,i+1} &= g_{5,i} + (g_{4,i} \lll 8) + g_{3,i}\\
x_{6,i+1} &= g_{6,i} + (g_{5,i} \lll 16) + (g_{4,i} \lll 16)\\
x_{7,i+1} &= g_{7,i} + (g_{6,i} \lll 8) + g_{5,i}
\end{aligned}$$

Where $g_{j,i}$ is defined by the following expression:

$$g_{j,i} = (x_{j,i} + c_{j,i+1})^2 \oplus \Big((x_{j,i} + c_{j,i+1})^2 \gg 32\Big) \bmod 2^{32} \tag{1}$$

All additions are performed modulo 2^{32} and squaring modulo 2^{64}.

Counter System.

$$\begin{aligned}
c_{0,i+1} &= c_{0,i} + a_0 + \phi_{7,i} \bmod 2^{32},\\
c_{j,i+1} &= c_{j,i} + a_j + \phi_{j-1,i+1} \bmod 2^{32}, \text{ for } 0 < j < 8.
\end{aligned} \tag{2}$$

where the counter carry bit $\phi_{j,i+1}$ is obtained as follows:

$$\phi_{j,i+1} = \begin{cases} 1 & \text{if } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \wedge j = 0,\\ 1 & \text{if } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \wedge j > 0,\\ 0 & \text{otherwise} \end{cases}$$

Furthermore, the constants $(a_j)_{0\leq j\leq 7}$ are defined as:

– $a_0 = a_3 = a_6 =$ `0x4D34D34D`,
– $a_1 = a_4 = a_7 =$`0xD34D34D3`,
– $a_2 = a_5 =$ `0x34D34D34`.

Extraction Scheme. For each iteration i of the next-state function, the current output keystream ${s_i}^{[127..0]}$ is extracted as follows:

$$\begin{aligned}
{s_i}^{[15..0]} &= {x_{0,i}}^{[15..0]} \oplus {x_{5,i}}^{[31..16]} \\
{s_i}^{[31..16]} &= {x_{0,i}}^{[31..16]} \oplus {x_{3,i}}^{[15..0]} \\
{s_i}^{[47..32]} &= {x_{2,i}}^{[15..0]} \oplus {x_{7,i}}^{[31..16]} \\
{s_i}^{[63..48]} &= {x_{2,i}}^{[31..16]} \oplus {x_{5,i}}^{[15..0]} \\
{s_i}^{[79..64]} &= {x_{4,i}}^{[15..0]} \oplus {x_{1,i}}^{[31..16]} \\
{s_i}^{[95..80]} &= {x_{4,i}}^{[31..16]} \oplus {x_{7,i}}^{[15..0]} \\
{s_i}^{[111..96]} &= {x_{6,i}}^{[15..0]} \oplus {x_{3,i}}^{[31..16]} \\
{s_i}^{[127..112]} &= {x_{6,i}}^{[31..16]} \oplus {x_{1,i}}^{[15..0]}
\end{aligned}$$

2.3 Previous Work on Rabbit

The stream cipher Rabbit has been designed to be faster than commonly used ciphers and to justify a key size of 128 bits for encrypting up to 2^{64} blocks of plaintext. In a series of white papers [1,2,3,4,5,6] and in [14], the designers gave convincing arguments to claim that Rabbit is resistant against algebraic, correlation, differential, guess-and-determine, and statistical attacks. Particularly, in [13], authors claim that Rabbit is immune to the replacement of all additions performed in the next-state function by `XOR`s (see Sect. 2.2). Indeed, since all possible byte-wise combinations of the output depend on at least four different g-functions, they conclude that "it seems to be impossible to verify a guess of fewer that 128 bits against the output".

In 2007, J-P. Aumasson raised a statistical weakness on Rabbit and more specifically on the core function "g" [7]. Although this function, based on a modular square, was expected to be strongly non-linear, J-P. Aumasson highlighted the non-uniformity of the bit distribution given a random initial state. The complexity of the resulted distinguishing attack is about $\mathcal{O}\left(2^{247}\right)$ which is much bigger than the complexity of a key exhaustive search (*i.e.* 2^{128}). That is why he concluded that the bias of the function "g" does not represent a real threat for Rabbit. Inspired by this work, L. Yi *et al.* used Fast Fourier Transformed (FFT) to compute the exact bias of Rabbit's keystream based on the bias of "g" [28]. That way, the distinguishing attack complexity equals to $\mathcal{O}\left(2^{158}\right)$, which is much closer to the key exhaustive search complexity. Moreover they extended their distinguishing attack to a multi-frame key recovery attack. This evolution has a $\mathcal{O}\left(2^{32}\right)$ memory complexity and $\mathcal{O}\left(2^{97.5}\right)$ time complexity. It is the first known key-recovery attack on Rabbit.

The first paper about the robustness of Rabbit implementations in the context of faults is due to A. Kirkanski and A. M. Youssef at SAC 09 [23]. They showed that by randomly flipping bits of the internal state, an attacker is able to recover the whole internal state from 128 – 256 faulty keystreams in $\mathcal{O}\left(2^{38}\right)$ steps with

a precomputed table of size $\mathcal{O}\left(2^{41.6}\right)$ bytes. Nevertheless, the analysis does not succeed if more than one bit of the internal state is flipped at a time. Furthermore the proposed fault analysis is limited to the recovery of the sole internal state.
In this paper, we propose to improve these results by considering another fault model. This new fault model is inspired by the design change studied in the context of a guess-and-verify attack [13]. Under this model, we prove that an attacker can completely recover the internal state from 32 faulty keystreams and in time $\mathcal{O}\left(2^{34}\right)$. The attack can also lead to a full secret key recovery (see Sect. 4.6).

3 Fault Attacks on Stream Ciphers

At the end of the nineties, a new class of active side channel attack appeared when Bellcore researchers proposed a way for recovering secret data by perturbing the behavior of public key cryptographic algorithms [15]. Then E. Biham and A. Shamir [12] proposed an application to the DES and named this class of attack Differential Fault Analysis (DFA).
Although fault attacks have been shown to be powerful against implementations of both public key [8,27,16] and secret key cryptosystems [12,18,24], few attacks have been published against the implementation of stream ciphers.
J. Hoch and A. Shamir [21] first addressed the issue of injecting fault to perturb the behavior of stream ciphers. They published in [21] a method for exploiting perturbations of LFSR based stream ciphers, and successful applications to LILI-128, SOBER-t32 and RC4. The fault attack against RC4 was later improved by Biham *et al.* [11]. In this paper, they showed how to use faults for setting the internal state of RC4 in an "impossible" state and a way to exploit it. Thus, they improved previous results both in terms of fault number and in terms of complexity. So, they concluded that the simplicity of the design of RC4 makes it weak against fault attacks.
The security against perturbations of the pseudo-random bit generator A5/1 has also been evaluated [20]. This stream cipher used in GSM networks for its cheap and efficient hardware implementation is composed of three LFSRs. The authors suggested to stop one of the shift register from clocking at a given moment and exploit the faulty output. According to this model, the use of faults speeds up the previous resynchronization attack on A5/1 by a factor 100.
A fault attack against Trivium was presented at FSE 2008 [22]. This hardware eSTREAM candidate is based on a 288-bit internal state split into three non-linear shift registers. The principle of this DFA is to perturb the internal state by flipping one bit at a random position. Then, the attacker obtains a system of equations in the internal state bits and takes advantage of the simplicity of the Trivium non-linear feedback function for solving it. According to this fault model, 43 fault injections (12 in the optimized version) are enough for recovering the secret key and the IV. This attack against Trivium is also the first application of DFA to a non-linear shift register based stream cipher.

A variant of an other eSTREAM finalist, GRAIN-128, has been evaluated in the context of fault attacks [10]. From an average of 24 consecutive bit-flips in the GRAIN-128 LFSR, A. Berzati *et al.* showed that it is possible to recover the secret key in a couple of minutes. Since the best known mathematical attack against GRAIN-128 is the brute force key-search, fault injections dramatically improve the efficiency of the key recovery.
The fault attack against Rabbit presented at SAC 09 [23] and the one proposed in this paper complete the *state-of-the-art* of fault attacks against stream ciphers (see Sect. 2.3).

4 Fault Attack on Rabbit

4.1 Preliminaries

Theorem 1. *If an attacker knows the values of the* $(g_{j,i})_{0\leq j\leq 7}$ *for two consecutive iterations* i *and* $i+1$*, then he can reduce the number of candidates for the remaining part of the internal state from* 2^{256+1} *to* 80 *in average, and predict the keystream.*

Proof. We assume that the attacker knows all the values of $(g_{j,i-1})_{0\leq j\leq 7}$ and $(g_{j,i})_{0\leq j\leq 7}$. From these values, he can compute respectively $(x_{j,i})_{0\leq j\leq 7}$ and $(x_{j,i+1})_{0\leq j\leq 7}$ by using the relations described in Sect. 2.2.
To completely determine the internal state at iteration $i+1$, the attacker has to find the counter variables $(c_{j,i+1})_{0\leq j\leq 7}$ and the carry bit $\phi_{7,i+1}$. But, the counters are the input of the function g (see (1)). Although this function is not bijective [7], previous results [28] emphasized by our own experimentation have shown that there are in average only 1.59 possible inputs that map to the same output $g_{i,j}$. Hence, as the attacker already knows all the $(g_{j,i})_{0\leq j\leq 7}$ and a part of the input $(x_{j,i})_{0\leq j\leq 7}$, he will find in average 1.59 candidate values for each $c_{j,i+1}$. As a consequence, he will find only $1.59^8 = 40$ candidate values for all the $(c_{j,i+1})_{0\leq j\leq 7}$ among $2^{8\cdot 32}$. Then, it remains the carry bit $\phi_{7,i+1}$ that can be found by exhaustive search or by comparing $c_{7,i+1}$ to a_7[3]. Thus the average number of candidates for the remaining part of the keystream is $2 \times 40 = 80$ □

4.2 Motivations

To evaluate the level of security of Rabbit, the designers have considered in [13] a guess-and-verify attack on a weak version of Rabbit. Indeed, they slightly modified the design of Rabbit by replacing all additions performed in the next-state function by `XOR`s (see Sect. 2.2). Under this assumption they showed that this weaker Rabbit was also immune against this kind of attack since all possible byte-wise combinations of the output depend on at least four different g-functions.

[3] Indeed, if $c_{7,i+1} < a_7$, it means that a modular reduction occurs in the addition, and then $\phi_{7,i+1} = 1$

But authors have not considered the security of Rabbit if only one addition is punctually replaced by a `XOR`. In the following study, we show that this state can be obtained by injecting faults and that it can be exploited to recover the secret key.

4.3 Fault Model

The principle of our attack is based on the recovery of all the values of $(g_{j,i})_{0\leq j\leq 7}$, for two consecutive iterations. Then Theorem 1 is applied to predict the keystream. These values are involved in the next-state function (see Sect. 2.2), as a consequence, we have chosen to perturb the behavior of that specific function.
According to the next-state scheme, the computation of each $x_{j,i+1}$ requires two consecutive modular additions (*i.e.* mod 2^{32}) that involves three values: $g_{j,i}$, $g_{(j+7)_{[8]},i}$ and $g_{(j+6)_{[8]},i}$. In our fault model, we assume that the attacker is able to perturb transiently one of these additions such that it becomes a bit-wise `XOR` only for the current operation. Indeed, all subsequent additions performed must result *error-free*. As several faults are necessary for recovering the key, the attacker must have the ability to run the stream cipher with the same initialization vector (not necessarily chosen). Like this, the state remains always the same.
Moreover we suppose the attacker can choose the iteration i and the index j of the affected value $x_{j,i+1}$ and which addition will be corrupted. This implies a preliminary fault setup stage. First as the algorithm implementation is software, the operations are executed sequentially and locating the time of the computation of the chosen value $x_{j,i+1}$ is possible. A transient fault generated by power glitches or light stimulation can produce various effects [9,19,26,25]. In our case, the transformation of addition to `XOR` can occur by two ways:

- Corruption of the carry register: if it is cleared, the addition is equivalent to a binary addition, *i.e.* an exclusive or, if the carry is set to 1, the addition is changed into a binary addition followed by a complement operation.
- Corruption of the processed code: The non volatile memory where the operating code is supposed to be stored can be modified while the reading of the memory is performed. For example, from the instruction set of the 8051 microprocessor, we can see that the code for `ADD` is `0x20` while it is `0x60` for `XRL`, so only one bit is distinctive. Let's note that the fetch code can also be corrupted in the cache memory of the internal register of the CPU.

If the attacker has a reference device, he can precompute the different expected bitstreams with a known key and compare the faulty ciphertexts until he obtains the setup corresponding to the fault model.
Otherwise, among the different faulty ciphertexts obtained by the attacker, some of them correspond to our fault model, and some others must be discarded. As our model corresponds to only 32 different faults, the attacker must only obtain 32 different faulty ciphertexts and can thus try the different combinations (see Appendix B).
Depending on the modified addition, the faulty state variable $\hat{x}_{j,i+1}$ can be expressed by:

– If j is even,

$$\hat{x}_{j,i+1} = \begin{cases} \left(g_{j,i} + \left(g_{(j+7)_{[8]},i} \lll 16\right)\right) \oplus \left(g_{(j+6)_{[8]},i} \lll 16\right) \\ \text{or } \left(g_{j,i} \oplus \left(g_{(j+7)_{[8]},i} \lll 16\right)\right) + \left(g_{(j+6)_{[8]},i} \lll 16\right) \end{cases}$$

– Else, if j is odd,

$$\hat{x}_{j,i+1} = \begin{cases} \left(g_{j,i} + \left(g_{(j+7)_{[8]},i} \lll 8\right)\right) \oplus g_{(j+6)_{[8]},i} \\ \text{or } \left(g_{j,i} \oplus \left(g_{(j+7)_{[8]},i} \lll 8\right)\right) + g_{(j+6)_{[8]},i} \end{cases}$$

In Rabbit, the output keystream ${s_i}^{[127..0]}$ depends on the values of the internal state. Thus, depending on which $\hat{x}_{j,i}$ that is perturbed, the output keystream will be infected as:

– If j is even, then the faulty part of the keystream is $\hat{s_i}^{[(16\cdot(j+2)-1)..(16\cdot j)]}$,
– Else, if j is odd, the faulty parts of the keystream are $\hat{s_i}^{[16\cdot((j-2)_{[8]}+1)-1..16\cdot(j-2)_{[8]}]}$ and $\hat{s_i}^{[16\cdot((j+3)_{[8]}+1)-1..16\cdot(j+3)_{[8]}]}$

This effect of the fault is helpful in case of a wrong time location. Indeed by computing ${s_i}^{[127..0]} \oplus \hat{s_i}^{[127..0]}$ and analyzing the position of non-zero values, one can immediately identify the state variable that has been infected by the fault during its update.

4.4 Fault Analysis

In the previous section, we have detailed the fault model used to perform our attack and the different ways to practice it. In this section, we provide the different steps for exploiting a set of faulty outputs.

Useful Propositions. This section provides some propositions that are used in the following description of our fault attack.

Proposition 1. *For all pairs $(x, y) \in (\mathbb{Z}/n\mathbb{Z})^2$, the resulted carry vector of the operation $x + y \ mod\ 2^n$, denoted by Carry (x, y), can be obtained by computing:*

$$Carry\ (x, y) = (x + y) \oplus (x \oplus y) \tag{3}$$

Proof. This is just a rewriting of the additional carry definition.

Proposition 2. *For all pairs $(x, y) \in (\mathbb{Z}/n\mathbb{Z})^2$, the i-th carry bit of the operation $x + y \ mod\ 2^n$, $Carry_i(x, y)$, can be defined recursively as:*

– *For $i = 0$, $Carry_0(x, y) = 0$*
– *For $i = 1$, $Carry_1(x, y) = x_0 \wedge y_0$*
– *For $1 < i \leq n$, $Carry_i(x, y) = x_{i-1} \wedge y_{i-1} \vee \left(Carry_{i-1}(x, y) \wedge (x_{i-1} \vee y_{i-1})\right)$*

Proof. This is the formula of the additive carry propagation.

Proposition 3. *For all triplets* $(x, y, z) \in (\mathbb{Z}/n\mathbb{Z})^3$*, we have:*

$$(x + y + z) \oplus ((x + y) \oplus z) = \mathit{Carry}\ (x + y, z) \tag{4}$$

Proof. This equality is a direct consequence of Proposition 1.

Proposition 4. *For all triplets* $(x, y, z) \in (\mathbb{Z}/n\mathbb{Z})^3$*, we have:*

$$\begin{aligned} &(x + y + z) \oplus ((x \oplus y) + z) \\ &= \mathit{Carry}\ (x + y, z) \oplus \mathit{Carry}\ (x \oplus y, z) \oplus \mathit{Carry}\ (x, y) \end{aligned} \tag{5}$$

Proof. This is also a consequence of Proposition 1. For a given triplet $(x, y, z) \in (\mathbb{Z}/n\mathbb{Z})^3$, $x + y + z$ can be written as:

$$\begin{aligned} x + y + z &= (x + y) \oplus z \oplus \text{Carry}\ (x + y, z) \\ &= x \oplus y \oplus \text{Carry}\ (x, y) \oplus \\ &\quad z \oplus \text{Carry}\ (x + y, z) \end{aligned}$$

Moreover, $((x \oplus y) + z) = x \oplus y \oplus z \oplus \text{Carry}\ (x \oplus y, z)$. Finally, we have:

$$\begin{aligned} &(x + y + z) \oplus ((x \oplus y) + z) \\ &= x \oplus y \oplus z \oplus \text{Carry}\ (x, y) \oplus \text{Carry}\ (x + y, z) \oplus \\ &\quad x \oplus y \oplus z \oplus \text{Carry}\ (x \oplus y, z) \\ &= \text{Carry}\ (x + y, z) \oplus \text{Carry}\ (x \oplus y, z) \oplus \text{Carry}\ (x, y) \end{aligned}$$

Differential Analysis. Fault attacks are often based on exploiting differences between a correct and a faulty output. Our attack is not different from it. We assume that the attacker is able to access the keystream[4]. Hence to perform the analysis, he differentiates the faulty keystream block with a correct block. In Sect. 4.3, we concluded that a fault injected according to our model only infects 32 bits of the output keystream. As a consequence, the difference is null except for the 32 infected bits:

– If j is even, then

$$\begin{aligned} &{s_i}^{[(16\cdot(j+2)-1)..(16\cdot j)]} \oplus {\hat{s}_i}^{[(16\cdot(j+2)-1)..(16\cdot j)]} \\ &= x_{j,i}^{[31..0]} \oplus \hat{x}_{j,i}^{[31..0]} \end{aligned} \tag{6}$$

and 0 elsewhere,

[4] The attacker knows a pair plaintext/ciphertext

– Else, if j is odd, then

$$\begin{aligned} & s_i^{[16\cdot((j-2)_{[8]}+1)-1..16\cdot(j-2)_{[8]}]} \oplus \hat{s_i}^{[16\cdot((j-2)_{[8]}+1)-1..16\cdot(j-2)_{[8]}]} \\ &= x_{j,i}^{[15..0]} \oplus \hat{x}_{j,i}^{[15..0]}, \end{aligned} \tag{7}$$

$$\begin{aligned} & s_i^{[16\cdot((j+3)_{[8]}+1)-1..16\cdot(j+3)_{[8]}]} \oplus \hat{s_i}^{[16\cdot((j+3)_{[8]}+1)-1..16\cdot(j+3)_{[8]}]} \\ &= x_{j,i}^{[31..16]} \oplus \hat{x}_{j,i}^{[31..16]} \end{aligned} \tag{8}$$

and 0 elsewhere,

Furthermore, depending on the modular addition that has been modified, the difference $x_{j,i}^{[31..0]} \oplus \hat{x}_{j,i}^{[31..0]}$ can be reformulated thanks to Propositions 3 and 4. As an example, we obtain for the perturbation of the second addition:

– If j is even,

$$\begin{aligned} & x_{j,i}^{[31..0]} \oplus \hat{x}_{j,i}^{[31..0]} \\ &= \text{Carry } ((g_{j,i-1} + g_{(j+7)_{[8]},i-1} \lll 8), g_{(j+6)_{[8]},i-1}) \end{aligned} \tag{9}$$

– Or if j is odd,

$$\begin{aligned} & x_{j,i}^{[31..0]} \oplus \hat{x}_{j,i}^{[31..0]} \\ &= \text{Carry } ((g_{j,i-1} + g_{(j+7)_{[8]},i-1} \lll 16), g_{(j+6)_{[8]},i-1} \lll 16) \end{aligned} \tag{10}$$

Similar expressions can be obtained if the first addition is perturbed by applying Proposition 4. The complete system of equations obtained after gathering faulty outputs modified at different locations in the next-state function is described in Appendix A. Hence, the differential fault analysis of the faulty output provides a set of particular equations that involves carries from the computation of additions in the next-state function (see Sect. 2.2).

Carry Analysis. The purpose of the attack is to use faults to recover the values of $(g_{j,i})_{0\leq j\leq 7}$. Thus, the attacker has modified both first and second modular additions in the next-state function one-by-one at iteration i for all $0 \leq j \leq 7$. This means that the attacker has to gather $8 + 8 = 16$ faulty keystream blocks modified at the same iteration i. Hence, the attacker can extract a system of equations that involves all the $(g_{j,i})_{0\leq j\leq 7}$ (see Appendix A). The number of obtained binary equations is[5] $16 \times 31 = 496$ for $8 \times 32 = 512$ binary unknowns $(g_{j,i})_{0\leq j\leq 7}$.
Because of the carry propagation, the degree of multivariate polynomials in the equations increases with the depth of the carry bit to analyze (*i.e.* the degree of the multivariate polynomial obtained by expressing Carry_i is $i + 1$). So, relinearization method like XL algorithm [17] are not relevant. The best way we

[5] $\forall (x, y) \in \left(\mathbb{Z}/2^{32}\mathbb{Z}\right)^2, \text{Carry}_0(x, y) = 0$, so the number of binary equations that results from a carry is 31

found to solve this system is performing an exhaustive search on each 8-bit parts of $x_{j,i+1} \oplus \hat{x}_{j,i+1}$ and so, on the 4 resulting sub-equations:

$$x_{j,i+1} \oplus \hat{x}_{j,i+1} \Rightarrow \begin{cases} x_{j,i+1} \oplus \hat{x}_{j,i+1}^{[7..0]} \\ x_{j,i+1} \oplus \hat{x}_{j,i+1}^{[15..8]} \\ x_{j,i+1} \oplus \hat{x}_{j,i+1}^{[23..16]} \\ x_{j,i+1} \oplus \hat{x}_{j,i+1}^{[31..24]} \end{cases} \tag{11}$$

Depending on the infected modular addition, each 8-bit sub-equation may have two different expressions (see Appendix A) that involves 8 bits of $g_{j,i}$, $g_{(j+7)_{[8]},i}$, and $g_{(j+6)_{[8]},i}$. As an example, for the expression $x_{0,i+1} \oplus \hat{x}_{0,i+1}^{[7..0]}$, the attacker will simultaneously search $g_{0,i}^{[7..0]}$, $g_{7,i}^{[23..16]}$ and $g_{6,i}^{[23..16]}$ that satisfy:

$$\begin{cases} \Delta_1 = \text{Carry}\left(\left(g_{0,i}^{[7..0]} + g_{7,i}^{[23..16]}\right), g_{6,i}^{[23..16]}\right) \\ \Delta_2 = \quad \text{Carry}\left(\left(g_{0,i}^{[7..0]} + g_{7,i}^{[23..16]}\right), g_{6,i}^{[23..16]}\right) \\ \qquad \oplus \text{Carry}\left(\left(g_{0,i}^{[7..0]} \oplus g_{7,i}^{[23..16]}\right), g_{6,i}^{[23..16]}\right) \\ \qquad \oplus \text{Carry}\left(g_{0,i}^{[7..0]}, g_{7,i}^{[23..16]}\right) \end{cases}$$

where Δ_1 and Δ_2 are equal to $x_{0,i+1} \oplus \hat{x}_{0,i+1}^{[7..0]}$ respectively when the second and first modular additions of the next state function are modified. Then, four pairs of equations have to be solved for each $0 \leq j \leq 7$. So, the obtained system of equations is considered as a 8-bit system of $2 \times 4 \times 8 = 64$ equations of $8 \times 4 = 32$ unknowns, as each $g_{j,i}$ is split in four 8-bit windows.

Solving each 8-bit sub-equation requires to search simultaneously 8 bits of three $g_{j,i}$. Moreover sub-equations from bit 8 to 31, the attacker has to speculate on $\text{Carry}_7(g_{j,i}, g_{(j+7)_{[8]},i})$, $\text{Carry}_{15}(g_{j,i}, g_{(j+7)_{[8]},i})$ and $\text{Carry}_{23}(g_{j,i}, g_{(j+7)_{[8]},i})$ since their values are unknown at the beginning of the search. As a consequence, for each j, the computational complexity equals to $\mathcal{O}\left(4 \times 2^{3\times 8+3}\right) = \mathcal{O}\left(2^{29}\right)$. To solve the whole system, the resolution has to be performed for all the eight j. So, the computational complexity of the resolution is $\mathcal{O}\left(2^{32}\right)$.

In order to study the characteristics of the system, and particularly, if the number of solutions for all the $(g_{j,i})_{0\leq j\leq 7}$ is bounded, we have randomly generated 20000 possible faulty outputs and counted the number of solutions provided by the system of equations. The experimental results show that the real 32-bit system of equations has an average of $2^{13.72}$ solutions. To recover the whole state of Rabbit, two systems from consecutive iterations have to be solved. Hence, the average number of possible solutions for $(g_{j,i-1})_{0\leq j\leq 7}$ and $(g_{j,i})_{0\leq j\leq 7}$ is $2^{2\times 13.72} = 2^{27.44}$. By combining these results with Theorem 1, the number of possible Rabbit states is the number of $(g_{j,i})_{0\leq j\leq 7}$ obtained for two consecutive iterations multiplied by the number of associated $((c_{j,i+1})_{0\leq j\leq 7}$ and $\phi_{7,i+1})$ found to complete the internal state: $2^{27.44} \times 80 \approx 2^{34}$.

Finally, for determining the attacked Rabbit state, at iteration $i+1$, among the 2^{34} candidates, the attacker has just to compute the corresponding internal state $\left((x_{j,i+1})_{0\leq j\leq 7}, (c_{j,i+1})_{0\leq j\leq 7}, \phi_{7,i+1}\right)$, generate the output keystream block for iterations $i+1$ and $i+2$, for each candidate, and compare it to the attacked Rabbit keystream at same iterations.

4.5 Attack Algorithm

Algorithm. Our fault attack against Rabbit can be divided into 5 distinguishable steps that have been presented in previous sections. This paragraph provides a summary that lists these steps:

Step 1: Gather faulty outputs, for iterations i and $i+1$, by perturbing one-by-one, all the sixteen additions of the next-state function. As a consequence, the attacker has to execute (with the same initialization vector) and perturb the Rabbit algorithm according to our model $2 \times 16 = 32$ times,

Step 2: Differentiate the faulty outputs $\hat{s}_i^{[127..0]}$ and $\hat{s}_{i+1}^{[127..0]}$, with correct outputs, ${s_i}^{[127..0]}$ and ${s_{i+1}}^{[127..0]}$. Then, check that faults were correctly injected (see Sect. 4.4),

Step 3: Built two systems of equations (see Appendix A) from the difference between faulty and correct outputs at iteration i and $i+1$. Then, recover possible candidates for $(g_{j,i-1})_{0\leq j\leq 7}$ and $(g_{j,i})_{0\leq j\leq 7}$. Compute $(x_{j,i})_{0\leq j\leq 7}$ and $(x_{j,i+1})_{0\leq j\leq 7}$,

Step 4: Solve $(c_{j,i+1})_{0\leq j\leq 7}$ and $\phi_{7,i+1}$ from previously recovered $(g_{j,i-1})_{0\leq j\leq 7}$ and $(g_{j,i})_{0\leq j\leq 7}$ (see Theorem 1),

Step 5: For each possible Rabbit state candidate at iterations $i+1$, compare the output keystream to the expected one until they are equal. When it is satisfied, the attacker has recovered the whole Rabbit state at iteration $i+1$ and can predict the subsequent keystream blocks.

Complexity. The efficiency of a fault attack is not only based on the fault model but in the number of faults to inject for obtaining secret information. Theoretically, to have an exploitable number of equations and performing the resolution, the attacker has to inject 32 faults that suits the model, at different locations of the next-state function and two consecutive iterations of the algorithm. In practice, the fault number can be more important, depending on its ability for reproducing the attack and the targeted device (see Sect. 4.3).
In terms of computational complexity, the overall complexity of the attack is dominated by the complexity for testing the possible solutions obtained from the differentiation of faulty outputs. So, the computational complexity of our attack is $\mathcal{O}\left(2^{34}\right)$. Moreover, since our analysis does not require any precomputation, the memory complexity of our fault attack is negligible compared to A. Kirkanski and A. M. Youssef proposal [23]. Hence, our new fault attack improves the best known time complexity from 2^{38} to 2^{34} [23] with a negligible memory consumption.

4.6 Extension to a full key recovery

The fault attack against Rabbit presented in this article allows the attacker to recover the whole Rabbit state at a given iteration i. As we previously noticed, it can be used to predict the keystream, but, if i is small enough, the attacker can recover the secret key. According to [28], if $i = 2$ then the attacker can recover the secret key used to generate the keystream in time $\mathcal{O}\left(2^{32}\right)$. To do it, the attacker guesses the values of the missing $\phi_{i,j}$'s to revert the Rabbit next-state function (see Sect. 2.2) and the key setup scheme. More details about the key recovery are provided in [28]. Hence, if $i = 2$ the complexity of this additional step is dominated by the full internal state recovery, and so, the global time complexity of this attack remains $\mathcal{O}\left(2^{34}\right)$.

5 Conclusion

This paper introduces an improved fault attack against implementations of Rabbit. Our theoretical results emphasized by our experimentation show that the fault analysis reduces the best known attack complexity against Rabbit from $\mathcal{O}\left(2^{38}\right)$ to $\mathcal{O}\left(2^{34}\right)$ [23]. This improvement is also effective in terms of memory consumption. Moreover, our attack requires only 32 faulty outputs, and we provide evidence that the fault model is practicable on various devices. The algorithm can be protected against faults by adding redundancy in the next-state function. As our attack only uses an addition corruption, the result of the additions can also be doubled and computed differently. Since this operation is faster than *"g"* function, this countermeasure does not increase the global complexity of Rabbit.
As a consequence, we can conclude that Differential Fault Analysis is a real threat for Rabbit implementations. Hence, protecting it against DFA is now challenging.

References

1. Cryptico A/S. Algebraic analysis of Rabbit. White paper, 2003.
2. Cryptico A/S. Analysis of the key setup function in Rabbit. White paper, 2003.
3. Cryptico A/S. Hamming weights of the g-function. White paper, 2003.
4. Cryptico A/S. Periodic properties of Rabbit. White paper, 2003.
5. Cryptico A/S. Second degree approximations of the g-function. White paper, 2003.
6. Cryptico A/S. Security analysis of the IV-setup for Rabbit. White paper, 2003.
7. J.P. Aumasson. On a Bias of Rabbit. In *State of the Art of Stream Ciphers (SASC 2007)*, 2007.
8. F. Bao, R.H. Deng, A. Jeng, A.D. Narasimhalu, and T. Ngair. Breaking Public Key Cryptosystems on Tamper Resistant Devices in the Presence of Transient Faults. In M. Lomas and B. Christianson, editors, *Security Protocols*, volume 1361 of *Lecture Notes in Computer Science*, pages 115–124. Springer-Verlag, 1998.
9. H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The Sorcerer's Apprentice Guide to Fault Attacks. Cryptology ePrint Archive, Report 2004/100, 2004.

10. A. Berzati, Cécile Canovas, G. Castagnos, B. Debraize, L. Goubin, A. Gouget, P. Paillier, and S. Salgado. Fault Analysis of Grain-128. In *IEEE International Workshop on Hardware-Oriented Security and Trust (HOST 2009)*. IEEE Computer Society, 2009.
11. E. Biham, L. Granboulan, and P. Nguyen. Impossible Fault Analysis of RC4 and Differential Analysis of RC4. In H. Gilbert and H. Handschuh, editors, *Fast Software Encryption (FSE 2005)*, volume 3557, pages 359–367. Springer, 2005.
12. E. Biham and A. Shamir. Differential Fault Analysis of Secret Key Cryptosystems. In *Crypto'97*, 1997.
13. M. Boesgaard, M. Vesterager, T. Christiensen, and E. Zenner. The stream cipher Rabbit. eStream Report 2005/024, the ECRYPT stream cipher project, 2005.
14. M. Boesgaard, M. Vesterager, T. Pedersen, J. Christiansen, and O. Scavenius. Rabbit: A High-Performance Stream Cipher. In T. Johansson, editor, *Fast Software Encryption (FSE 2003)*, volume 2887 of *Lecture Notes In Computer Science*, pages 307–329. Springer, 2003.
15. D. Boneh, R.A. DeMillo, and R.J. Lipton. On the Importance of Checking Cryptographic Protocols for Faults. In W. Fumy, editor, *EUROCRYPT'97*, volume 1233 of *Lecture Notes in Computer Science*, pages 37–51. Springer-Verlag, 1997.
16. E. Brier, B. Chevallier-Mames, M. Ciet, and C. Clavier. Why One Should Also Secure RSA Public Key Elements. In L. Goubin and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems (CHES 2006)*, volume 4249 of *Lecture Notes in Computer Science*, pages 324–338. Springer-Verlag, 2006.
17. N. Courtois, A. Klimov, J. Patarin, and A. Shamir. Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In B. Preneel, editor, *Advances in Cryptology - Eurocrypt 2000, International Conference on the Theory and Application of Cryptographic Techniques*, volume 1807 of *Lecture Notes in Computer Science*, pages 392–407. Springer, 2000.
18. P. Dusart, G. Letourneux, and O. Vivolo. Differential Fault Analysis on AES. In M. Yung, Y. Han, and J. Zhou, editors, *Applied Cryptography and Network Security (ANCS 2003)*, volume 2846 of *Lecture Notes in Computer Science*, pages 293–306. Springer, 2003.
19. C. Giraud. A survey on fault attacks. In *CARDIS 2004*, Smart Card Research and Advanced Applications IV, pages 159–176, 2004.
20. M. Gomulkiewicz, M. Kutilwoski, and P. Wlaz. Synchronization Fault Analysis for Breaking A5/1. In Sotiris E. Nikoletseas, editor, *Experimental and Efficient Algorithms (WEA 2005)*, volume 3503 of *Lecture Notes in Computer Science*, pages 415–427. Springer, 2005.
21. J. Hoch and A. Shamir. Fault Analysis of Stream Ciphers. In M. Joye and J-J. Quisquater, editors, *Cryptographic Hardware and Embedded Systems (CHES 2004)*, volume 3156 of *Lecture Notes in Computer Science*, pages 240–253. Springer, 2004.
22. M. Hojsik and B. Rudolf. Differential Fault Analysis of Trivium. In Kaisa Nyberg, editor, *Fast Software Encryption (FSE 2008)*, volume 5086 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2008.
23. A. Kirkanski and A. M. Youssef. Differential Fault Analysis of Rabbit. In *Selected Areas in Cryptography (SAC 2009)*, Lecture Notes in Computer Science, pages 200–217. Springer, 2009.
24. G. Piret and J-J. Quisquater. A Differential Fault Attack Technique against SPN Structures, with Application to the AES and KHAZAD. In C. Paar, Ç.K. Koç, and C.D. Walter, editors, *Cryptographic Hardware and Embedded Systems (CHES 2003)*, volume 2779 of *Lecture Notes in Computer Science*, pages 77–88. Springer, 2003.

25. S.P. Skorobogatov. Optically Enhanced Position-Locked Power Analysis. In *Cryptographic Hardware and Embedded Systems (CHES 2006)*, volume 4249 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 2006.
26. S.P. Skorobogatov and R.J. Andersson. Optical Fault Induction Attacks. In B.S. Kaliski Jr., Ç.K. Koç, and C. Parr, editors, *Cryptographic Hardware and Embedded Systems (CHES 2002)*, volume 2523 of *Lecture Notes in Computer Science*, pages 2–12. Springer-Verlag, 2002.
27. D. Wagner. Cryptanalysis of a provably secure CRT-RSA algorithm. In *Proceedings of the 11th ACM Conference on Computer Security (CCS 2004)*, pages 92–97. ACM, 2004.
28. L. Yi, W. Huaxiong, and S. Ling. Cryptanalysis of Rabbit. In T.-C. Wu, C.-L. Lei, V. Rijmen, and D.-T. Lee, editors, *11th International Conference on Information Security (ISC 2008)*, volume 5222 of *Lecture Notes In Computer Science*, pages 204–214. Springer-Verlag, 2008.

A System Extracted

We assume that the attacker has injected a fault, at iteration $i+1$, on different modular additions of the next-state function.

A.1 First Set of Equations

By modifying the second addition for all eight equations of the next-state function at iteration $i+1$, computing the difference $s_{i+1} \oplus \hat{s_{i+1}}$ (see Sect. 4.4), and using Proposition 3 the attacker obtains the following set of equations:

$$\begin{aligned}
x_{0,i+1} \oplus \hat{x}_{0,i+1} = \ & \mathrm{Carry}((g_{0,i} + (g_{7,i} \lll 16)), \\
& (g_{6,i} \lll 16)) \\
x_{1,i+1} \oplus \hat{x}_{1,i+1} = \ & \mathrm{Carry}\left((g_{1,i} + (g_{0,i} \lll 8)), g_{7,i}\right) \\
x_{2,i+1} \oplus \hat{x}_{2,i+1} = \ & \mathrm{Carry}((g_{2,i} + (g_{1,i} \lll 16)), \\
& (g_{0,i} \lll 16)) \\
x_{3,i+1} \oplus \hat{x}_{3,i+1} = \ & \mathrm{Carry}\left((g_{3,i} + (g_{2,i} \lll 8)), g_{1,i}\right) \\
x_{4,i+1} \oplus \hat{x}_{4,i+1} = \ & \mathrm{Carry}((g_{4,i} + (g_{3,i} \lll 16)), \\
& (g_{2,i} \lll 16)) \\
x_{5,i+1} \oplus \hat{x}_{5,i+1} = \ & \mathrm{Carry}\left((g_{5,i} + (g_{4,i} \lll 8)), g_{3,i}\right) \\
x_{6,i+1} \oplus \hat{x}_{6,i+1} = \ & \mathrm{Carry}((g_{6,i} + (g_{5,i} \lll 16)), \\
& (g_{4,i} \lll 16)) \\
x_{7,i+1} \oplus \hat{x}_{7,i+1} = \ & \mathrm{Carry}\left((g_{7,i} + (g_{6,i} \lll 8)), g_{5,i}\right)
\end{aligned}$$

A.2 Second Set of Equations

This second set of equations results from the perturbation of all the eight first additions of the next-state function and the application of Proposition 4:

– If j is even, $x_{j,i+1} \oplus \hat{x}_{j,i+1}$ equals to:

$$\begin{aligned} & \mathrm{Carry}((g_{j,i} + (g_{(j+7)_{[8]},i} \lll 16)), \\ & \qquad (g_{(j+6)_{[8]},i} \lll 16)) \\ \oplus\ & \mathrm{Carry}((g_{j,i} \oplus (g_{(j+7)_{[8]},i} \lll 16)), \\ & \qquad (g_{(j+6)_{[8]},i} \lll 16)) \\ \oplus\ & \mathrm{Carry}(g_{j,i}, (g_{(j+7)_{[8]},i} \lll 16)) \end{aligned}$$

– Else, if j is odd, $x_{j,i+1} \oplus \hat{x}_{j,i+1}$ equals to

$$\begin{aligned} & \mathrm{Carry}\left(\left(g_{j,i} + \left(g_{(j+7)_{[8]},i} \lll 8\right)\right), g_{(j+6)_{[8]}}\right) \\ \oplus\ & \mathrm{Carry}\left(\left(g_{j,i} \oplus \left(g_{(j+7)_{[8]}} \lll 8\right)\right), g_{(j+6)_{[8]}}\right) \\ \oplus\ & \mathrm{Carry}\left(g_{j,i}, \left(g_{(j+7)_{[8]}} \lll 8\right)\right) \end{aligned}$$

B Case of unexploitable faults

Among the different faulty ciphertexts obtained by the attacker, some of them correspond to our fault model, and some others must be discarded. This case happens when the faults have not been injected according to our model. We have simulated this situation by trying to solve the system of equations with wrong ones. With our detection strategy no 8-uplet for $(g_{j,i})_{0 \leq j \leq 7}$ was found for such wrong systems.
So the attacker has to try to withdraw some equations until the system has solutions. Thus he determines the wrong equations. Once identified, the equations must replaced by other ones obtained from new faults.