Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4

Eli Biham [⋆1], Louis Granboulan[⋆⋆2], and Phong Q. Nguyễn[⋆⋆3]

[1] Computer Science Department, Technion – Israel Institute of Technology, Haifa 32000, Israel. biham@cs.technion.ac.il – http://www.cs.technion.ac.il/∼biham/
[2] École normale supérieure, DI, 45 rue d'Ulm, 75005 Paris, France. Louis.Granboulan@di.ens.fr – http://www.di.ens.fr/˜granboul/
[3] CNRS/École normale supérieure, DI, 45 rue d'Ulm, 75005 Paris, France. Phong.Nguyen@di.ens.fr – http://www.di.ens.fr/˜pnguyen/

Abstract. In this paper we introduce the notion of *impossible fault analysis*, and present an impossible fault analysis of RC4, whose complexity 2^{21} is smaller than the previously best known attack of Hoch and Shamir (2^{26}), along with an even faster fault analysis of RC4, based on different ideas, with complexity smaller than 2^{16}.

1 Introduction

RC4 is a stream cipher designed by Ron Rivest in 1987, and used by RSADSI in their products. It was never officially published, but a reverse-engineered copy of the code appeared anonymously in the sci.crypt newsgroup in 1994. Nowadays, RC4 is one of the most widely used stream ciphers in a wide range of applications.

Fault analysis was introduced in 1996, when an attack on implementations of RSA and other public key algorithms was described [3]. Shortly after, differential fault analysis of secret key cryptosystems such as DES has followed [2]. These attacks can be made in practice, and various techniques have been described that induce faults during cryptographic computations [1, 9, 14, 16].

Various observations were made on the design and properties of RC4 since its publication, e.g., in [5–7, 10, 12, 13, 15], however till recently no fault analysis was performed. In a recent paper [8] Hoch and Shamir study fault analytic attacks against various stream ciphers, including RC4. Their results show that the initial state of RC4 can be recovered given 2^{26} bytes of stream with about 2^{16} key setups, and 2^{26} steps of analysis.

In this paper we present the notion of *impossible fault analysis*, which uses faults to force the internal state of the stream cipher to become an impossible

⋆ Part of this work was done while visiting the École normale supérieure, Paris, France.

⋆⋆ The work described in this article is supported in part by the French government through X-Crypt, and in part by the Commission of the European Communities through the IST program under contract IST-2002-507932 ECRYPT.

Attack	#Faults	Data	#Rekey	Time	Space
Hoch and Shamir [8]	2^{16}	2^{26}	2^{16}	2^{26}	2^{16}
This paper:					
Impossible Fault Analysis	2^{16}	2^{21}	–	online (2^{21})	2^{8}
Differential Fault Analysis	2^{10}	2^{16}	2^{10}	2^{16}	2^{10}

Table 1. Summary of the Results for Finding the Initial States

state, i.e., a state that can never occur in a regular use of the cipher. We use this notion to force RC4 to enter impossible states that were first observed by Hal Finney in [4], and were later described with additional properties in [11]. Once the internal state falls into Finney's impossible states, the output stream become a copy (actually 255 interleaved copies) of the internal state — a phenomena which is very unexpected. In order to fall to these state we need only 2^{21} bytes of stream (without any additional key setups), and once we fall into Finney's states, the recovery of the internal state is one of the simplest ever described for any cipher, as the internal state is simply copied into the output stream.

We then describe another, more standard, fault analytic attack against RC4, which require less than 2^{16} stream bytes with less than a thousand (or a few thousands) key setups, and whose analysis time complexity is also lower than 2^{16}. However, because the assumptions used by the two attacks are different, it is perhaps not clear which attack would be more efficient in practice.

A summary of our results, and of the previously published results is given in Table 1.

The paper is organized as follows: In Section 2 we give a short description of RC4. In Section 3 we describe the notion of impossible fault analysis and the application to RC4. In Section 4 we describe the best known fault analysis of RC4, and in Section 5 we summarize the paper.

2 A Brief Description of RC4

For the discussion of this paper it suffices to describe the step function of RC4. The key setup is very similar, but is not necessary for our analysis.

RC4's internal state consists of two indices i and j, and an array S of size 256. In a real computation of RC4, this array is always a permutation of all the values from 0 to 255. The content of the S array is the result of the key schedule, while i and j are always set to 0 by the key schedule. Each step of RC4 is then as follows

i ++
j += $S[i]$
swap the values of $S[i]$ and $S[j]$
Output $S[S[i]+S[j]]$

3 Impossible Fault Analysis of RC4

In [4] Hal Finney presented the following family of internal states of RC4, where

$$j = i + 1, \qquad \text{and} \qquad S[j] = 1. \tag{1}$$

He observed that if an internal state is a member of this family at some step, then all the next and prior states are also in this family, as if the internal state is in the family, then the value 1 is swapped with the next value, and i and j are incremented by 1. In his thesis, Itsik Mantin [11] also observes that due to these swaps, every 255 steps the array S is rotated by one byte, but remains with the same circular order of byte values, and that every 255 states, the index of the output byte repeats (as it is just the sum of the two entries pointed by i and j, i.e., the first repeats every 255 states, and the other is always 1). As a result of these two observations, the output stream takes the same entry from the array every 255 output bytes, but the byte values are already rotated, making the new value of the entry after 255 states being the next value (that was in the next entry), and any further 255th byte in the output stream being the next value of the internal state in a cyclic order. We thus receive 255 interleaved streams, each of them is a copy of the internal state with a (possibly) different starting value. It is easy to see that once we fall into these states, it is very easy to identify that we fell into these states, and also very easy to deduce the content of the internal state (due to an additional property it is even easy to know which is the first value in the S array, thus to know the exact location of each value in the array, not only the relative order).

However, the key schedule of RC4 sets $i = j = 0$ causing these states to be impossible, thus these states can never appear in real RC4 streams, and we cannot expect to fall into these states in our analysis.[4]

Fortunately, for the purposes of this paper, fault analysis can make the impossible possible, by modifying the indices i or j, or even the content of the S array. Therefore, if a fault occurs during the computation of RC4, it may occur that the resulting internal state becomes a member of this impossible family, and once the internal state is a member of the family, it is very easy to deduce the internal state by looking at the output stream.

For the attack we assume that faults are injected into either register i or register j at any time the attacker wishes. Once a fault modifies the internal state to become one of Finney's states, it is very easy to identify this fact after several hundred bytes (after two or three bytes from each interleaved cycle are given, i.e., after a total of about 500 or 700 bytes). However, as we show below, the identification of non-Finney states can be made much earlier.

We first observe that the probability of falling into Finney's states by a fault in i or j is 2^{-16}, because the two equalities defining Finney's states in Eq. (1) compare a total of 16 bits, where

[4] Some RC4 variants with modified key schedule may allow this family of states to occurs for some fraction (usually about 2^{-16}) of the keys. In such cases, this fraction of the keys form a class of weak keys, in which the initial state is being copied to the output stream.

1. If the fault is in i, the probability that $S[j] = 1$ holds is 2^{-8}, and the probability that the new i is set to $j-1$ is also 2^{-8}.
2. If the fault is in j, the probability that $S[i+1] = 1$ holds is 2^{-8}, and the probability that the new j is set to $i+1$ is also 2^{-8}.

Therefore, we expect that the internal states will fall to Finney's states after about 2^{16} faults are induced.

Once a fault is induced, we would like to identify as soon as possible if the internal state is a Finney's state or not. We observe that Finney's states have the property that the consecutive (or close) output states fetch the output bytes from different locations in the S array (because a different value is added to the 1 to form the index of the output byte), and that the array is a permutation of all the 256 values from 0 to 255. So usually close bytes of the output stream should be distinct, though, bytes that were swapped already with the prior location in the array may appear a second time. Therefore, we expect that a collision of two bytes of the output stream should occur after about 80 bytes of the output stream, while in the case of non-Finney's states a collision should occur after less than 20 bytes due to the birthday paradox (we verified this behavior by a simulation).[5]

Based on these observations, we use the following procedure for identifying whether a state is a Finney's state or not:

1. For each new output byte check whether the byte value already appeared since the last fault.
2. If not, process the next byte (goto step 1).
3. If this byte value already appeared at least once.
   Denote the number of bytes since the last fault by n.
   (a) If $n < 30$, the state is most probably not a Finney's state.
   (b) If $30 \leq n < 40$ and this is the second collision, or if $40 \leq n < 50$ and this is the third collision, (etc., with a few additional similar tests designed to distinguish between the two cases), then the state is most probably not a Finney's state.[6]

---

[5] A short approximate calculation of the case of Finney's states is as follows. We ignore the case where the colliding output byte has value 1, since it is rare, and affects the result only marginally. Then, (when the output is not 1) a collision of a byte a states after the fault and another byte b states after the fault can occur only if $S[i_a]+S[j_a] = S[i_b]+S[j_b]+1$ and $i_a \leq S[i_a]+S[j_a] \leq i_b$ (in a cyclic manner, where i_x and j_x are the values of the registers x states after the fault). The probability of the first equation to hold is $1/256$, while the probability of the second equation is $(b-a)/256$. Therefore, under some independence assumptions, the probability that a collision occurs within the first L bytes after the fault is approximated by $\sum_{0 \leq a < b < L}(1/256)((b-a)/256)$ which is about $L^3/(3 \cdot 2^{17})$. By forcing this probability to be about 1 we get that $L \approx 73$, and by forcing it to be about 1/2 we get that $L \approx 58$. We expect that accurate computation will show slightly higher values.

[6] Our actual simulation used a slightly different set of thresholds, but for simplicity of the description we present a rounded version.

(c) If $n > 255$, check the relation to the 255th preceding byte, and (unless one of these two values is 1) verify that they are different, and that no other earlier pair of this kind appeared with a combination of the first value with another second value, or the second value with another first value. If such a pair appeared, the state is not a Finney's state.
(d) If $n > 600$, the state is most probably a Finney's state.
(e) Otherwise, process the next byte (goto step 1).

4. Once the state is identified as either (most probably) a Finney's state, or (most probably) not a Finney's state, the algorithm stops and outputs this information (for the rest of the attack, we will not distinguish between a most probable identification or an absolute identification).

The attack is thus as follows: start encryption with RC4 with some unknown key, and apply the procedure repeatedly, as many times as required, till a Finney's state is identified. Each time the procedure identifies a non-Finney's state, inject a new fault to either i or j before the next application of the procedure. Once the procedure identifies a Finney's state, there are a few options

1. The simplest but slow method is to process extra $255 \cdot 256$ bytes, and select one of the interleaved cycles as the internal state.
2. A much faster method is to use the information we already got about consecutive values in the internal state to recover the full state.
3. In both cases the assignment of the cycle to the exact location in the S array can be done using information obtained at time that the output byte is taken from either location i or location j (i.e., $S[i] + S[j]$ is either i or j), in which the outputs are easily identified, e.g., when $S[i] + S[j] = i$ then the output byte is necessarily 1.

Once the internal state is obtained at some point in time, it is easy to find the current values of i and j, and then to process the states backwards till the prior fault. It is also possible to identify the prior value of the faulty register by careful analysis of the prior bytes of the output stream (i.e., given the internal state, one of the two registers, and the output byte, it is easy to recover the value of the other register), thus enabling the attacker to obtain the key-dependent initial state.

The attacker can deduce the key from this initial state, e.g., with the technique described in section A.2 of [11].

We programmed most parts of this attack, and verified that it works and that the expected number of required bytes of the output stream is about 2^{21} (i.e., the average number of bytes required for recognition of a non-Finney's state is not far from 32). In all our tests, our simulation of the attack always identified Finney's states correctly, and was not sensitive to whether the faults were induced in register i or register j (in fact, even if the attacker have no control on the selection of the register in which the fault is induced, and even if each fault is induced on one of the registers at random, still the attacker will be able to recover the initial state without additional complexity).

This attack have various variants: The faults can also be injected to the S array, but then if the fault is injected at a random location in the array, the probability to get a Finney's state is only 2^{-24} (i.e., $i = j - 1$ occurs with probability 2^{-8}, the fault occurs at the new j with probability 2^{-8}, and the fault modifies this location to a value 1 with probability 2^{-8}). An additional problem in this case is that the fault change the content of the S array quite rapidly, making it difficult to follow the changes backwards to the initial state once a Finney's state occurs (a solution to this latter problem is to make a new key setup once in a while, like is made is [8] and in the next section).

Once we make fault in the array, it is possible to replace usage of the value 1 in the attack by two consecutive 2's in the following way

$$j = i + 2, \qquad \text{and} \qquad S[j-1] = S[j] = 2,$$

or with 3's

$$j = i + 3, \qquad \text{and} \qquad S[j-2] = S[j-1] = S[j] = 3,$$

or similarly with larger values. The probability to reach such states is smaller, but once they appear, the internal state is copied to the output in the same way as in Finney's states.

If we assume a stronger fault model than we did, in which the attacker can induce the faults and select the exact new value of the register (i or j or an entry of the S array), the attack would be much faster (but then we assume that there exists an even more efficient attack designed specifically for this stronger model).

Finally, as in Finney's states the output stream repeats with known period, it may be possible to mount a ciphertext only attack where the attacker is given the ciphertext rather than the output stream. Given a known statistics of the plaintext (e.g., the knowledge that the plaintext is written in English), these attacks would required much longer streams for the identification of Finney's internal states, but once these states are identified, the combined knowledge on the statistics of the plaintext's language and the properties of the internal state may leak full information on the plaintext and on the initial state (this attack may be applicable in cases where a smartcard encrypts using RC4 while being at the possession of the attacker).

4 A Differential Fault Analysis of RC4

This attack makes use of the fact that each step of RC4 accesses only three bytes of the S array. In this section we present the main ideas behind this attack, as there are some delicate cases where a more complicated handling should be performed, which we will not describe here. We first collect the following data

1. Process a non-faulty stream
   (a) Perform a key setup with the unknown key

(b) Run RC4 256 steps, keeping the output stream for later analysis. Call this output stream R.

2. Process the following 256 times with l being set from 0 to 255, giving 256 output streams of length about 30
   (a) Perform a key setup with the unknown key
   (b) Make a fault in $S[l]$
   (c) Run RC4 30 steps, keeping the output stream for later analysis. Call this output stream O_l.
3. Repeat again but with the fault being injected after 30 steps of RC4, calling the output streams T_l, and keeping longer output streams in the T_l's.
4. Possibly repeat again with a few additional such sets.

Evidently, the first byte of all the O_l's, except for three of them, are the same as in the real stream R. The identification of these three streams leak the values of i, j, and $S[i] + S[j]$, but not which is which. Evidently, the value of i is always known, thus it remains only to identify which is j and which is $S[i] + S[j]$. We continue doing so for the following bytes of the output streams (ignoring streams O_l which already had earlier faults). This technique can be called *cascade guessing*.

Now, in any location in the stream, whichever of the two values is j, we can subtract the previous j from the new j and get $S[i]$, then testing whether we already know that this value appears in another location in the array, allowing us to discard many wrong cases. Similarly, the other value is $S[i] + S[j]$, and by knowing the output byte (of the real stream R) we know the content of the location pointed by this value $S[i] + S[j]$, which also allows us to make a test for a contradiction. In addition, also $S[j]$ is observed, and can also be tested for a contradiction. Due to the birthday paradox, we expect to get a contradiction after an average of less than 20 observed bytes, i.e., after processing about 7 bytes of the output stream. Therefore, the analysis for finding the internal state should find a considerable number of bytes of the internal state with a small complexity.

However, as the faults (made before the first step of RC4) become more and more distant from the analyzed bytes (computed after several steps of RC4), the identification of the three values become less and less successful, due to earlier effects of these faults on the streams. This problem have several solutions: one of them is to have several sets with faults made at different times (like the T_l's mentioned above). Another solution is to make a more delicate analysis and use the faulty streams to recover more information on the key (as the faults changed the state slightly, different bytes may be reached from these streams). Or alternatively, it is possible to allow the algorithm to ignore some unknown values during the computation, and continue to the next stream byte. A combination of these methods can recover a large majority of the bytes of the internal state with a relatively small complexity, after which much simpler analysis can complete the full content of the internal state.

The complexity of this attack is bounded by 2^{16} using a total of less than 2^{16} stream bytes (with less than a thousand, only a few thousands, key setups).

5 Summary

In this paper we presented two fault attacks on RC4. The first introduces the notion of impossible fault analysis, and shows how to apply it with complexity smaller than of previously published fault attacks against RC4. The second attack uses more key setups and analyzes the difference behavior of the key stream once the internal states is modified due to faults. This later attack is currently the fastest known fault analysis of RC4.

Acknowledgments

We would like to thank Adi Shamir, Jonathan Hoch, and Itsik Mantin for various valuable comments and discussions which improved the results of this paper.

References

1. R. J. Anderson and S. Skorobogatov, "Optical fault induction attacks." in *Proceedings of CHES'02* (B. S. Kaliski, Çetin Kaya Koç, and C. Paar, eds.), no. 2535 in Lecture Notes in Computer Science, Springer-Verlag, 2002.
2. E. Biham and A. Shamir, "Differential fault analysis of secret key cryptosystems." in *Proceedings of Crypto'97* (B. S. Kaliski, Jr, ed.), no. 1294 in Lecture Notes in Computer Science, pp. 513–525, Springer-Verlag, 1997.
3. D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults." in *Proceedings of Eurocrypt'97* (W. Fumy, ed.), no. 1233 in Lecture Notes in Computer Science, pp. 37–51, Springer-Verlag, 1997.
4. H. Finney, *An RC4 Cycle that Can't Happen*, September 1994.
5. S. R. Fluhrer, I. Mantin, and A. Shamir, "Weaknesses in the key scheduling algorithm of RC4." in *Proceedings of Selected Areas in Cryptography – SAC'01* (S. Vaudenay and A. M. Youssef, eds.), no. 2259 in Lecture Notes in Computer Science, pp. 1–24, Springer-Verlag, 2001.
6. S. R. Fluhrer and D. A. McGrew, "Statistical analysis of the alleged RC4 stream cipher." in *Proceedings of Fast Software Encryption – FSE'00* (B. Schneier, ed.), no. 1978 in Lecture Notes in Computer Science, pp. 19–30, Springer-Verlag, 2000.
7. J.D. Golic, "Linear Statistical Weakness of Alleged RC4 Keystream Generator." in *Proceedings of Eurocrypt'97* (W. Fumy, ed.), no. 1233 in Lecture Notes in Computer Science, pp. 226–238, Springer-Verlag, 1997.
8. J. J. Hoch and A. Shamir, "Fault Analysis of Stream Ciphers." in *Proceedings of CHES'04* (M. Joye and J.-J. Quisquater, eds.), no. 3156 in Lecture Notes in Computer Science, pp. 240–253, Springer-Verlag, 2004.
9. O. Kömmerling and M.G. Kuhn, "Design Principles for Tamper-Resistant Smartcard Processors." in *Proceedings of the USENIX Workshop on Smartcard Technology (Smartcard '99)*, pp. 9–20, USENIX Association, 1999.
10. L.R. Knudsen, W. Meier, B. Preneel, V. Rijmen, and S. Verdoolaege, "Analysis Methods for (Alleged) RC4." in *Proceedings of Asiacrypt'98* (K. Ohta and D. Pei, eds.), no. 1514 in Lecture Notes in Computer Science, pp. 327–341, Springer-Verlag, 1998.

11. I. Mantin, *Analysis of the Stream Cipher RC4*, M.Sc. thesis, The Weizmann Institute of Science, 2001. Available at `http://www.wisdom.weizmann.ac.il/~itsik/RC4/rc4.html`.
12. I. Mantin and A. Shamir, "A practical attack on broadcast RC4." in *Proceedings of Fast Software Encryption – FSE'01* (M. Matsui, ed.), no. 2355 in Lecture Notes in Computer Science, pp. 152–164, Springer-Verlag, 2001.
13. I. Mironov, "(Not So) Random Shuffles of RC4." in *Proceedings of Crypto'02* (M. Yung, ed.), no. 2442 in Lecture Notes in Computer Science, pp. 304–319, Springer-Verlag, 2002.
14. D. Naccache, P.Q. Nguyễn, M. Tunstall and C. Whelan, "Experimenting with Faults, Lattices and the DSA" in *Proceedings of PKC '05* (S. Vaudenay, ed.), no. 3386 in Lecture Notes in Computer Science, Springer-Verlag, 2005.
15. Souradyuti Paul, Bart Preneel, "A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher.", proceedings of Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, Lecture Notes in Computer Science 3017, pp. 245–259, Springer-Verlag, 2004.
16. J.-J. Quisquater and D. Samyde, "ElectroMagnetic Analysis (EMA): Measures and counter-measures for smart cards." in *Proceedings of E-smart 2001* (I. Attali and T. P. Jensen, eds.), no. 2140 in Lecture Notes in Computer Science, pp. 200–210, Springer-Verlag, 2001.