Cache Timing Analysis of RC4

Thomas Chardin[1], Pierre-Alain Fouque[2], and Delphine Leresteux[3]

[1] DGA Engineering and Integration, 7 rue des Mathurins, 92221 Bagneux Cedex
[2] Département d'informatique, École normale supérieure, 45 rue d'Ulm, F-75230 Paris Cedex 05
[3] DGA Information Superiority, BP7, 35998 Rennes Armées
thomas.chardin@dga.defense.gouv.fr,
pierre-alain.fouque@ens.fr,
delphine.leresteux@dga.defense.gouv.fr

Abstract. In this paper we present an attack that recovers the whole internal state of RC4 using a cache timing attack model first introduced in the cache timing attack of Osvik, Shamir and Tromer against some highly efficient AES implementations. In this model, the adversary can obtain some information related to the elements of a secret state used during the encryption process. Zenner formalized this model for LFSR-based stream ciphers.

In this theoretical model inspired from practical attacks, we propose a new state recovery analysis on RC4 using a belief propagation algorithm. The algorithm works well and its soundness is proved for known or unknown plaintext and only requires that the attacker queries the RC4 encryption process byte by byte for a practical attack. Depending on the processor, our simulations show that we need between 300 to 1,300 keystream bytes and a computation time of less than a minute.

Keywords: cryptanalysis, stream cipher, RC4, cache timing analysis.

1 Introduction

Some side channel attacks have been recently formalized in theoretic work by modelling powerful adversaries that can learn a bounded amount of arbitrary information on the internal state by Dziembowski and Pietrzak in [9]. Here we consider information coming from cache attacks which is of the same kind but more practical since they correspond to real attacks which have been experimented on AES implementation [18,7,4,22,3]. Concretely, when the cipher is looking for a value in a table, a whole line of cache is filled in, containing but not limited to the value looked for in the table. This mechanism allows to achieve better performance since in general when a program needs some data, it also requests the successive ones soon after. Osvik, Shamir and Tromer proposed in 2006 an attack on some AES implementations that use look-up tables to implement the S-box and showed that the adversary can learn the high order bits of the index looked for, but neither the whole index itself nor the corresponding

J. Lopez and G. Tsudik (Eds.): ACNS 2011, LNCS 6715, pp. 110–129, 2011.
© Springer-Verlag Berlin Heidelberg 2011

value of the table. These attacks are rather practical since they have been implemented [18,7] on classical implementations used in the OpenSSL library. Others cache attacks target DSA [1] or ECDSA [8] operations in the OpenSSL library due to branch prediction on instructions.

To gain more information from cache monitoring, Osvik *et al.* propose to run a concurrent process at the same time as the encryption process. Attackers can evict data from the cache using the second process which begins by reading a large table to flush the cache. Then, the encryption process is run; the attacker finally tries to read again the elements of his table. If the element is in the cache, the access is fast (cache hit) and in the other case, the access is slow (cache miss) since the information has been evicted from the cache. Consequently, the adversary is not allowed to read the cache, but since the cache lines correspond to lines in the memory, if the adversary knows how the encryption process organizes the data in the memory (the address of the whole table for instance), the information of which cache line has been removed from the cache allows to recover the index (or a part of it) of the value looked for by the encryption process. Indeed, we do not recover the whole index since the cache is filled in line by line, so we know that the encryption process has read some element of the whole line but not exactly which element. Moreover, if the encryption process performs many table lookups, we do not have the order of the indexes since we perform timing on our own process which is run after the encryption process. These practical analyses allow us to consider such attacks on encryption schemes through a new security model. For example, Zenner *et al.* propose to study security of LFSR-based stream ciphers in [23,15].

RC4 is a stream cipher designed in 1987 by Ron Rivest and widely used in many standards such as in SSL, WEP, WPA TKIP, etc. The internal state of RC4 is composed of two indexes and of a permutation over $\mathbf{F}_{256}$. The initialization of the permutation table depends on the secret key (which size varies between 0 and 256 bits); the table is then updated during the generation of the keystream. Many attacks have been proposed on RC4 since its design was published in 1994 but none of them really breaks RC4. The bad initialization used in the WEP protocol and the key schedule algorithm of RC4 have been attacked by Fluhrer, Mantin and Shamir in [10]. Recent improvement has revealed new linear correlations in RC4 in order to mount key retrieve attacks on WEP and WPA [21]. Since then, from a cryptographic point of view, this scheme has not been broken despite many statistical properties. Finally, more powerful attacks have been taken into account, for instance fault attacks by Hoch and Shamir, Biham *et al.* in [12,6]. However, the number of faults is rather high, 2^{16} for the most efficient attack.

Previous Work. Our analysis is related to the one published in 1998 by Knudsen *et al.* in [14], which try to recover the internal state from the keystream. Once the internal state is recovered, it is possible to run the algorithm backward and efficient algorithms allow to recover the key [5]. Though an improvement was proposed in 2000 [11] and another in 2008 [16], such attacks remain impractical, having a time complexity of 2^{241} operations for the full RC4 version. The basic idea of the "deterministic" attacks (section 4 of [14] and [16]) is to

guess some values of the table and then check if these guesses are valid with the output keystream. These algorithms perform a clever search by guessing bytes when they need them and then use a backtracking approach when a contradiction appears. However, a huge number of values have to be guessed so that the complexity is relatively high in the end. This is basically the algorithm of [14]. Maximov and Khovratovich in [16] improve this algorithm by looking at the equations of RC4:

$$\begin{aligned} i_t &= i_{t-1} + 1 \\ j_t &= j_{t-1} + S_{t-1}[i_t] \\ S_t[i_t] &= S_{t-1}[j_t], \quad S_t[j_t] = S_{t-1}[i_t] \\ Z_t &= S_t[k] \quad \text{where} \quad k = S_t[i_t] + S_t[j_t] \end{aligned}$$

In the algorithm of [14], the number of unknowns is 4 (j, $S[i]$, $S[j]$ and $k = S[i] + S[j]$) even if they are related). Maximov and Khovratovich solve these equations by noting that if j is known for different times t, then $S[i]$ also, and the number of unknowns is reduced to 2. Then, they show that it is possible to have the value of j for consecutive times t, and also to detect such patterns from the keystream. The attack begins by locating in the keystream a good pattern which gives information about the internal state and j, and then since the equations are simpler the complexity is lower. Solving such linear systems with *non-linear terms* has also been recently extended by Khovratovich *et al.* to more complex equations system in [13] in the context of differential trail for hash functions.

Finally, Knudsen *et al.* propose a "probabilistic" algorithm in section 5 of [14], which is different from the deterministic one since the idea is that the output keystream gives conditions on the internal secret state which leads to conditional probability distribution $\Pr(S[i] = v | Z_t = z)$. Now, the internal state is represented with a probabilistic distribution table: to each element $S[i]$ in the table is associated a probability distribution on the 256 possible values. At the beginning, for all, i and v, $\Pr(S[i] = v) = 1/256$. Then according to the output keystream byte Z_t, an a posteriori distribution is computed using Bayes rules and the previous values in the distribution table, and finally the algorithm accordingly updates the distribution table. This probabilistic algorithm does not work if no more information is used. Knudsen *et al. partially* fulfill the table at the beginning with correct values. Their experiments show that they need 170 values so that the algorithm converges. They use the same idea (used later by Maximov and Khovratovich): they fulfill the table such that consecutive values of j can be found which makes the equations easier.

The algorithm we propose here is different from the one described in [14]; however they have in common the manner of using the structure of PRGA, acronym of Pseudo Random Generation Algorithm, to propagate constraints on the values of elements of the secret state used by RC4.

Our Results. RC4 is a good candidate to study cache timing analysis since it uses a rather large table and indexes of the lookups give information about

the table. In this paper, we present a probabilistic algorithm that recovers the current state of the permutation table. Contrary to previous state recovery attacks [14,16] whose complexity is about 2^{241} only based on cryptanalysis, our algorithm is more efficient in practice and its soundness has been proved due to information from cache timing analysis, although same cryptanalysis techniques are used. The experiments are derived using our algorithm and by simulating the information obtained in the model designed by Osvik *et al.* in [18]. The idea is the following: to generate a cipher byte, the generation algorithm uses three lookups in the permutation table and then updates the corresponding elements. Thus, the knowledge of the elements used to generate this byte allows the attacker to eliminate some candidate values for these elements, leading little by little to the recovery of the entire permutation.

The key recovery algorithm we developed is probabilistic as the one of Knudsen *et al.* (section 5 of [14]). However, we use a belief propagation algorithm to use the cache attack information. Belief propagation algorithms have been used in information theory and coding theory and are related to Bayesian networks and Hidden Markov Models, when a state is hidden and has to be recovered given some information. Recently, such algorithms have been successfully used to prove convergence and complexity results on the random assignment problem [20]. The algorithm propagates the partial information on the indexes by modifying the distribution table. According to the distribution table $\Pr(S[i] = v)$ and the partial indexes, the algorithm computes for all possible guesses, the values of the probability of such guesses. Since one guess is the correct one, we then normalize all these probabilities, and we update the distribution table according guesses modify or not the value of $\Pr(S[i] = v)$.

The algorithm gives good results but we improve the data complexity and the success probability by using time to time an assignment algorithm, such as the Hungarian algorithm, because we know that the table S is a permutation. Without cache analysis information, the algorithm cannot be effective and the complexity is too large as the probabilistic algorithm of Knudsen *et al.*. In this paper, we show that in practice, we can recover the secret permutation using only 300 bytes of the keystream in the best case. In the case of ciphertext only attack, our analysis works in practice only when the cache lines are half of the real size. Using partially known plaintext, we can recover the internal table using 3,000 bytes with probability 95%. This attack shows that RC4 must be used with some care when attackers can monitor the cache and query the cipher stream byte by byte.

Organization of the paper. The paper is organized as follows: after having presented the information expected to get from cache monitoring in section 2, we describe in section 3 a first algorithm, that only works on idealized cases. Then our main algorithm is detailed in section 4, which is able to take information from real cache monitoring to recover the permutation table through an example based on OpenSSL implementation. Finally we conclude and point out some practical thoughts on data collection in section 5.

2 The Context of the Attack

Our algorithm is designed to attack RC4 from a side-channel, the cache memory. We will briefly describe RC4 and the structure and the mechanism of cache memory. Then we explain how this cache memory can be considered as a side-channel and how to exploit the resulting data leakage.

2.1 Description of RC4

The design of RC4 has been kept secret until it was leaked anonymously in 1994 on the Cypherpunks mailing list [2]. It consists of two algorithms. The first one, named KSA (key-scheduling algorithm), is used to initialize the permutation table according to the secret key. The other PRGA is used to generate a keystream byte and update the permutation. While the details of KSA are not relevant to our subject, PRGA is described in algorithm 1.

Algorithm 1. Pseudo-random generation algorithm

$i \leftarrow i + 1$
$j \leftarrow j + S[i]$
swap($S[i]$, $S[j]$)
$k \leftarrow S[i] + S[j]$
return $S[k]$

2.2 Structure and Use of Cache Memory

To make up for the increasing gap between the latency of microprocessors and memories, some little but low-latency memory modules have been included in modern microprocessors. The aim of these cache memories is to preload frequently accessed data, reducing the latency of load/store instructions (an average code uses one such instruction out of three). There are often two caches, named L1 and L2. The L1 cache is closer to the CPU but smaller than the L2 cache. These memories are organized in Z sets. Each set contains W cache lines of B bytes; a memory block which address is a in memory is stored in cache at the cache set $a/B \bmod Z$, starting at the byte $a \bmod B$ of a random cache line. The main placement policy is to evict the most ancient stored data to store a new one. The values of these parameters for Pentium 4 processors can be found in table 1.

The use of cache is as follows:

- when the processor needs data from memory, it sends a request to L1 cache;
- If the L1 cache contains the required data (cache hit), they are sent to the CPU; otherwise (cache miss), the request is transmitted to the L2 cache;

Table 1. Cache parameters for the Pentium 4

	B	Z	W
L1	64	32	8
L2	64	4096	8

- If the L2 cache contains the data, they are transmitted to L1 cache (for further use) and CPU; if not, the request is transmitted to the main memory, and the data are copied in both caches.

This behavior allows us to make out four different time scales (the figures are given for a Pentium 4 CPU):

- the average execution time of an instruction: 1 CPU cycle (1 nanosecond on a 1 GHz CPU);
- the latency of the L1 cache: 3 cycles;
- the latency of the L2 cache (and time to write the data in the L1 cache): 18 cycles;
- the latency of the RAM: approximately 50 nanoseconds.

Because of this mechanism, the execution time of a process may vary significantly according to the relative number of cache hits and misses during its execution. Such variations can be used as a side-channel, either for covert communication or for cryptanalytic purposes.

2.3 Cache Timing Analysis

The first side-attack using cache memory was described by Page in 2002 [19]. It uses the fact that, in DES, if two rounds use the same element in some S-box, then the global encryption time will be reduced (the second lookup resulting in a cache hit). In 2006, Bonneau and Mironov [7] adapted this attack to AES, allowing to recover a secret key in an average of 10 minutes and needing approximately 2^{20} encryptions.

An even more powerful attack was published in 2006 by Osvik, Shamir and Tromer [18]. The use of a test table filling the cache memory allowed them to know which cache lines were used by the encryption process (causing the eviction from cache of the corresponding lines of the test table, hence a greater latency to access again to these elements). The knowledge of the position of the encryption tables in memory allowed them to know which elements of these tables were used, helping them to recover a secret key with less than 16,000 encryptions (depending on the processor on which the attack was implemented).

2.4 Prerequisites

The context of the analyses presents here is the same as the one of [18]. We assume that the attacker is able to monitor the cache memory and learns partial information on which element of the permutation table is used. Our algorithm is particularly fitted for "synchronous attacks", where the attacker can trigger encryption himself. Osvik *et al.* [18] first present the concept of synchronous cache attacks and Zenner *et al.* [23,15] establish a cache model based on this concept. They define an adversary having access to two oracles. The first oracle allows to obtain a list of output keystream bytes. At the same time, the second oracle delivers a truthful list of all cache requests realized by the first oracle. This cache attack in Zenner's model has several properties like noise-free and

there is no order on the cache accesses. However, if the attacker can produce the same cipher operation with the same input and output many times, the noise and wrong cache accesses could be removed using many samples. Finally, one must note that monitoring the cache does not allow us to know exactly which element of the table is used. In fact, when a request causes a cache miss, the whole cache line corresponding to the requested data is loaded into the cache. As the elements of the RC4 permutation table are stored as 32-bit integers (in 32-bit CPUs), a cache line contains (in the case of a Pentium 4) 16 elements. Thus, using a test table to monitor the cache only allows us to know the index modulo 16 of the element used.

2.5 Conventions and Experimental Set-Up

We study the current version of RC4 with bytes even though some authors have tried to attack weaker versions with smaller permutation table. Consequently, unless explicitly stated, all additions and subtractions will be done modulo 256.

We denote by δ the number of integers per cache line, and S the permutation table used for encryption.

We assume that the attacker can trigger the generation of the stream cipher byte by byte; the data are collected as in [18].

We have simulated cache accesses on OpenSSL library and we have experimented several cache models, from idealized one to real one.

Finally, all the experimental results are presented as follows: "time" represents the time needed by the attack to succeed, "requests" the number of needed cipher byte requests, and "success" the number of times when the table given by the attack is equal to the permutation of RC4. All results are given on average, over 50 random secret keys.

3 A First Algorithm for Idealized Cases

In this section we assume for the sake of simplicity that monitoring the cache allows us to know exactly which element of the permutation has been used by the encryption process. It allows us to introduce our attack on a simple case which is presented in appendix A. We will see further how to adapt the attack to a more complex cache model.

Data Structures Used by the Algorithm. For each table element i we have to keep a trace of the remaining candidates for its value. To do so we use a `256x256`-boolean table S_{values}, where $S_{values}[i][v] = 0$ if and only if we are sure that $S[i] \neq v$. Another `256`-boolean table is used for j, with the same conventions. We do not need to keep an equivalent structure for i, as its value is known through the whole encryption process.

Exploitation of the Data Collected During the Generation of One Cipher Byte. The generation of one cipher byte gives us the following data:

- the first index i;
- the unknown internal index j;

– three indexes a, b and c (possibly equal) of elements used during the byte generation;
– the cipher byte, which we will call out.

We invite the reader to note that we do not know the order of use of the three elements of the table, except for the first (because we know i). We will suppose from now on that $a = i$. Two orders remain: (a, b, c) and (a, c, b). We then use the structure of PRGA, which imposes the following constraints in the case (a, b, c):

$$j + S[a] = b$$
$$S[a] + S[b] = c$$
$$S[c] = out$$

Algorithm 2. Algorithm for one step with order (a, b, c) when $a \neq b \neq c \neq a$

1. **for** $v \leftarrow 0$ **to** $2^n - 1$ **do**
2.   **for** $t \leftarrow 0$ **to** $2^n - 1$ **do**
3.   **if** $v \neq a$ **and** $v \neq b$ **and** $v \neq c$ **then**
4. $\mathcal{S}_{values_bis}[v][t] \leftarrow \mathcal{S}_{values}[v][t]$
5.   **else**
6. $\mathcal{S}_{values_bis}[v][t] \leftarrow 0$
7.   **end if**
8.   **end for**
9. $j_{values_bis}[v] \leftarrow 0$
10. **end for**

{Exploit cache data, order (a, b, c)}

11. **for** $j \leftarrow 0$ **to** $2^n - 1$ **do**
12.   **if** $j_{values}[j] = 1$ **and** $\mathcal{S}_{values}[a][b - j] = 1$ **and** $\mathcal{S}_{values}[b][c - b + j] = 1$ **and** $\mathcal{S}_{values}[c][out] = 1$ **then**
13. $\mathcal{S}_{values_bis}[a][c - b + j] \leftarrow 1$
14. $\mathcal{S}_{values_bis}[b][b - j] \leftarrow 1$
15. $\mathcal{S}_{values_bis}[c][out] \leftarrow 1$
16. $j_{values_bis}[b] \leftarrow 1$
17.   **end if**
18. **end for**

{Repeat steps 11. to 18. for order (a, c, b)}
{Write new tables}

19. **for** $v \leftarrow 0$ **to** $2^n - 1$ **do**
20.   **for** $t \leftarrow 0$ **to** $2^n - 1$ **do**
21. $\mathcal{S}_{values}[v][t] \leftarrow \mathcal{S}_{values_bis}[v][t]$
22.   **end for**
23. $j_{values}[v] \leftarrow j_{values_bis}[v]$
24. **end for**

We deduce from these equations that the values of $S[a]$, $S[b]$ and $S[c]$ are completely determined given the order of the lookups and the values of j and out. We then do the following steps. For each order and each value v of j, we check if the corresponding values of $S[a]$, $S[b]$ and $S[c]$ remain possible. If not, we are sure that

these order and values are not possible, which allows us to remove these candidates for $S[a]$, $S[b]$ and $S[c]$. We reduce the number of candidates. Finally, we update the S_{values} and j_{values} tables to take the swap of PRGA into account. Algorithm 2. gives the details of one step of the attack; a special care has to be taken when a, b or c are equal, because the swap operation is performed before reading the last element $S[c]$. A special care has to be taken when a, b or c are equal, because the swap operation is performed before reading the last element $S[c]$.

We continue to use such data until every table element has only one possible candidate value. Other candidates are eliminating due to contradiction. We have then found the current permutation table and value of j.

4 Adaptation to Real Caches

There exist a main difference between the idealized framework used above and the monitoring cache memory. Even if we did not know the order of use of the elements, we knew exactly which elements of the table were used. In particular, we were sure not to modify during an attack step the candidate values of an unused table element. When we monitor real cache memory, we only know some most significant bits of the index of used table elements. Therefore we cannot be sure that the candidate values, on which we are trying to study, really correspond to an effectively used table element.

To take this issue into account, we use the probabilistic frame presented in [14]. Instead of the boolean table S_{values}, we now use a `256x256`-floating point number table S_{probas}. We now noted $\mathbf{P}(S[i] = v)$ where $S_{probas}[i][v]$ is the estimated probability $S[i] = v$. At the beginning of the attack, we suppose that every value is equiprobable for any element of S and for j so these two tables are filled with the value 1/256.

4.1 The Known-Plaintext Attack

We assume that the attacker knows, for each step:

- the first index i;
- three indexes $a \wedge$ `mask`, $b \wedge$ `mask` and $c \wedge$ `mask`, where `mask` corresponds to the known bits and is computed from the value of δ: if $\delta = 16$, then `mask` = `0xf0`;
- the cipher byte *out*.

A Probabilistic Version of the Algorithm. The belief propagation algorithm we design is exactly the same as the one for the idealized case, with the exception that we do not know b and c for sure. As in the previous section, we will assume that $a = i$ and that $a_\delta = a$ `&& mask`, $b_\delta = b$ `&& mask` and $c_\delta = c$ `&& mask` are different (the case with some equalities is treated in a similar way but taking care of the possible collisions in the permutation table).

We know that the indexes b and c used for the PRGA step are of the form $b = b_\delta + o_1$ and $c = c_\delta + o_2$ where $0 \leq o_1, o_2 < \delta$, so the equations giving $S[a]$,

$S[b]$ and $S[c]$ are the same as in the previous section, except that we have two more unknowns, o_1 and o_2.

Our algorithm is hence modified as following:

1. we make two guesses for the values of o_1 and o_2 as well as the guess for the value v of j and for the order of use of a, b and c;
2. we compute the corresponding values of $S[a]$, $S[b]$ and $S[c]$ similarly to what was done in the idealized case;
3. we evaluate the probability of this guess to be exact, assuming for the sake of simplicity that all probabilities are independent:

$$\begin{aligned}\mathbf{P}(guess) = \mathbf{P}(j = v) &\cdot \mathbf{P}(S[a] = b_\delta + o_1 - v)\\ &\cdot \mathbf{P}(S[b_\delta + o_1] = c_\delta + o_2 - b_\delta - o_1 + v)\\ &\cdot \mathbf{P}(S[c_\delta + o_2] = out)\end{aligned}$$

4. having done these computations for all possible guesses, we normalize the corresponding probabilities (these guesses were the only ones having a chance to describe what really happened during the keystream byte generation);
5. finally each guess contributes to modify the tables S_{probas} and j_{probas}, since for each guess we know exactly the action of the PRGA step on the three elements looked up, the others remaining unmodified. Adding all these contributions we obtain the global transformation of the table S_{probas}:

$$\forall i, \forall v, \mathbf{P}(S[i] = v)_{\text{after attack step}} = \left(\sum_{\text{guesses imposing } S[i]=v} \mathbf{P}(\text{guess})\right) \cdot 1$$

$$+ \left(1 - \sum_{\text{guesses imposing } S[i]} \mathbf{P}(\text{guess})\right) \cdot \mathbf{P}(S[i] = v)_{\text{before attack step}}$$

(the first term of the sum gives the action of all transformations where the equation $S[i] = v$ is guaranteed by the action of the swap; the second step gives the action of all transformations where $S[i]$ is not affected, i.e. $S[i] \neq v$). The update of j_{probas} is much simpler since for each guess the new value of j is imposed by the collected data; the formula is similar to the one for S_{probas} without the second term on the right.

How to Know if the Attack Succeeds? Two parameters remain to be set: the time when we decide that the attack has succeeded (or failed), and the processing of the solution. We first use a simple criterion to stop the attack: we consider that it succeeds when for each table element a candidate value has a greater probability than 1/2. The solution is thus “built” by local optimization, retaining for each element the value with greatest probability. We consider that the attack failed if this event does not occur after a certain number of steps.

Once the solution is found, we must first reverse it back to the initial permutation, then test it. The reversion is easy, given that a step of PRGA is invertible if the values of i and out are known. To test it, several options can be chosen:

- a first and very simple test is to make sure that the solution given by local search on each element is a permutation. If this is the case, we have a good hint that this solution will be the good one: using the Stirling formula, we can evaluate the probability of a random application on $\mathbf{F}_{256}$ to be a permutation to approximately $2 \cdot 10^{-110}$;
- we can then verify that the solution is the good permutation trying to predict some following cipher bytes.

First Experimental Results. We use to implement our attack a simulation of the cache monitoring to collect the data necessary to the attack. The experimental results obtained are reproduced in Table 2 - the case where $\delta = 1$ being given only for comparison purposes, as the concerned results are similar to those obtained in idealized cases.

Table 2. Known-plaintext cache attack of RC4, first version

δ	1	2	4	8	16
Time	0.393 s	0.542 s	0.962 s	2.836 s	25.48 s
Requests (maximum)	417	498	498	594	898
Requests (average)	326	384	400	456	666
Requests (minimum)	268	310	344	387	556
Success	100%	78%	66%	44%	56%

We can bring out from these results that our intuition on the convergence of the probabilities is correct. However, the results for $\delta = 16$ make this attack nearly impractical for real CPUs.

4.2 An Improvement: Searching Permutations

The main issue concerning the previous test of success is that we never use the fact that what we search is not any application on $\mathbf{F}_{256}$ but a permutation. We take this crucial constraint into account reducing the search of a solution to a constraint-solving problem: once we have the densities of probability for each table element, searching the best permutation (which probability is given by the product of the probabilities of the values of all its elements) is equivalent to solve an assignment problem.

An assignment problem takes the following canonical formulation: given an integer n, a set $\mathcal{V}$ of size n (historically representing n workers), another set $\mathcal{D}$ of size n (representing n tasks) and an application $c : \mathcal{V} \times \mathcal{D} \to \mathbf{R}$ (representing the cost of assigning a worker to a task), find a bijection $f : \mathcal{V} \to \mathcal{D}$ that minimizes the global cost given by:

$$\sum_{v \in \mathcal{V}} c(a, f(a))$$

Our problem is easily reducible to an assignment problem, using for $0 \leq i, v < 256$ the cost function $c(i, j) = -\log(\mathbf{P}(S[i] = j))$.

Table 3. Known-plaintext cache attack of RC4 with global search of the solution

δ	1	2	4	8	16
Time	0.704 s	1.320 s	2.587 s	5.750 s	33.53 s
Requests (maximum)	300	400	400	500	700
Requests (average)	297	318	377	413	597
Requests (minimum)	268	300	300	368	533
Success	100%	98%	100%	100%	98%

The assignment problem can easily be written as a linear programming problem, but we prefer to solve it using the Hungarian algorithm, designed by Kuhn and Munkres in 1955–1957 [17] and which time complexity is cubic. We decide to stop the attack when the probability of the best found permutation is greater than 2^{-16}, value experimentally optimized according to a compromise between the economy of encryption requests and the improvement of the success rate. As the cost for the resolution of this problem is far greater than the cost of the previous local search, this criterion is used for one step amongst 100. Furthermore, it is not used at the beginning of the attack (for the 300 first steps). The previous criterion is used again in the latter cases.

The attack is carried out in the same conditions as before. The results are shown in Table 3.

As guessed, the global search allows to reduce significantly the number of needed encryption requests and greatly improves the success rate. We have no decisive argument so far to prove the termination or estimate the convergence of this algorithm, except for hints from information theory; however, its proofs of soundness can be found in appendix B.

4.3 The Unknown-Plaintext Attack

In this paragraph the considered hypothesis does not use the properties of probability distribution for each language characters and their probability to appear. We suppose that the attacker does not know the language or the byte code employed. The known-plaintext attack is easily adapted into an unknown plaintext one, simply by considering that all values are uniformly possible for the cipher byte (this is not exactly true, as most of the existing attacks against RC4 lie on the non-uniformity of the distribution of the first cipher bytes; however, this remains a good approximation). As a matter of consequence:

– the probability for each order, j value and offsets is computed as above, simply summing on all possible values for the output;

$$\begin{aligned}\mathbf{P}(guess) = \sum_{out=0}^{255} \mathbf{P}(j = v) \cdot \mathbf{P}(S[a] = b_\delta + o_1 - v) \\ \cdot \mathbf{P}(S[b_\delta + o_1] = c_\delta + o_2 - b_\delta - o_1 + v) \\ \cdot \mathbf{P}(S[c_\delta + o_2] = out)\end{aligned}$$

$$\sum_{out=0}^{255} \mathbf{P}(S[c_\delta + o_2] = out) = 1$$

$$\mathbf{P}(guess) = \mathbf{P}(j = v) \cdot \mathbf{P}(S[a] = b_\delta + o_1 - v) \cdot \mathbf{P}(S[b_\delta + o_1] = c_\delta + o_2 - b_\delta - o_1 + v)$$

– the probabilities of the table element corresponding to the output (that is to say, the third one in the chosen order) are not updated anymore, since we know nothing on its value and the PRGA does not modify it.

Results. We test the resulting attack in the same conditions as before, except the number of steps without global search of the solution, which we adapt using the minimal number of needed requests; the results are shown in Table 4.

Table 4. Unknown plaintext cache attack of RC4 (with global search of the solution)

δ	1	2	4	8
Time	1.512 s	6.395 s	3.409 s	12.06 s
Requests (maximum)	350	600	900	1,287
Requests (average)	318	490	765	1,099
Requests (minimum)	300	429	700	950
Success	100%	96%	92%	96%

For $\delta = 16$, we do not succeed to make the probabilities converge.

4.4 A Partially-Known-Plaintext Attack

We finally develop another kind of the attack adapted to the main use of RC4: communications. In this case, the attacker partially knows the stream cipher, which corresponds for example to the case of TCP packets of which one attacker can guess some header bytes. If the output is known, we obtain the $\mathbf{P}(guess)$ of the known-plaintext attacks. On the contrary, if the output is unknown, the $\mathbf{P}(guess)$ formula of the unknown-plaintext attacks is used. The results are shown in Figure 1, for $\delta = 16$; the average time does not vary much, and the success rate is better than 47 over 50 trials.

5 Remarks on Collecting Data

In order to mount a practical attack on existing implementations from the above analysis, we make some comments on the data collection:

– in some implementations of RC4, as in OpenSSL, the internal state is stored as fields of the secret key; in particular, the two integers containing the value of i and j are stored just before the permutation table, which may have two effects:

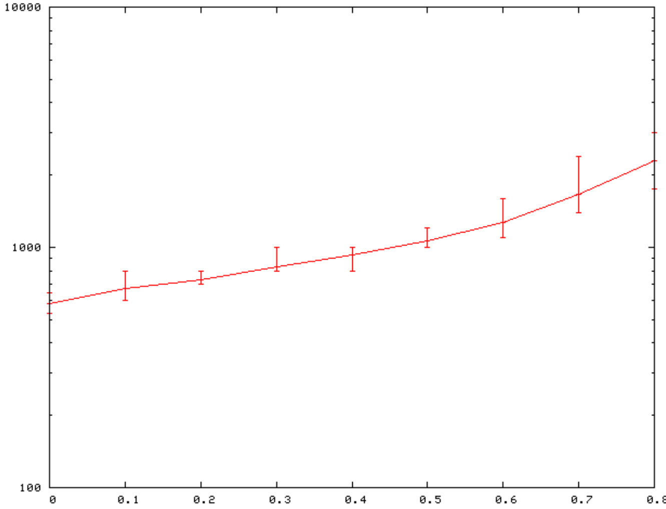

Fig. 1. Partially-known-plaintext cache attack on RC4: number of requests (logarithmic scale) as a function of the rate of unknown plaintext

  - first, there is high probability that the beginning of the permutation table does not correspond to the beginning of a cache line. On the one hand, the attacker is not supposed to know where the table begins, so he must use the attack algorithm for each possible offset between the first element of the table and the beginning of the nearest cache line. On the other hand, this offset causes the first and last cache lines covered by the table to contain less than δ elements, which gives the attacker a little more information,
  - besides, the PRGA reads the value of both index registers to generate each cipher byte, so the attacker cannot distinguish if the elements stored in the first covered cache line have been really used, which leads to a loss of information;
- last but not least, the attacker must be able to use the PRGA byte by byte; if he cannot, all the given information he will get will be a set of used cache line numbers through a large amount, say k, of cipher byte generation. He will then have to test all ordered sets of $3k$ cache line numbers using all the elements of the read set (but knowing with certainty the element out of three corresponding to the current value of i), which number is at least $(2k)!$, instead of k times the two orders used above.

6 Conclusion

We have detailed an efficient way of recovering the internal state of a RC4 process from cache monitoring. The presented algorithm is efficient in both known- and unknown-plaintext contexts. It allows to recover the internal permutation using

the generation of on average about 550 keystream bytes and less than a minute of computation time in an OpenSSL implementation. The only prerequisite is the possibility for the attacker to run a process on the attacked machine and to trigger the keystream generation by itself.

To avoid cache attacks, many countermeasures have been proposed in [18]. The main thing consists of removing or masking data links to memory access. Cache could be replaced by registers or several copies of the lookup table could be used. These countermeasures cost more in time and resources required. Shuffling memory and adding random allow to avoid cache timing analysis and execution branch analysis too.

The recovering algorithm is based on a belief propagation mechanism to infer information on a hidden state which evolves in a deterministic manner and output some values of this state. Our algorithm is rather simply but very efficient, and its soundness has been proved. However, we are not able to prove its termination and complexity, which we leave as an open problem.

This article draws attention to RC4 implementation has to be carefully used to avoid cache attacks. Indeed, it would prevent from monitoring the cache and querying ciphertexts, from accessing cache from remote computer.

References

1. Aciiçmez, O., Brumley, B.B., Grabher, P.: New results on instruction cache attacks. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 110–124. Springer, Heidelberg (2010)
2. Anonymous: RC4 source code. Cypherpunks mailing list (September 1994), `http://cypherpunks.venona.com/date/1994/09/msg00304.html`
3. Bernstein, D.J.: Cache-timing attacks on AES. Technical report (2005)
4. Bertoni, G., Zaccaria, V., Breveglieri, L., Monchiero, M., Palermo, G.: Aes power attack based on induced cache miss and countermeasure. In: ITCC, vol. (1), pp. 586–591. IEEE Computer Society, Los Alamitos (2005)
5. Biham, E., Carmeli, Y.: Efficient reconstruction of rc4 keys from internal states. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 270–288. Springer, Heidelberg (2008)
6. Biham, E., Granboulan, L., Nguyen, P.Q.: Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 359–367. Springer, Heidelberg (2005)
7. Bonneau, J., Mironov, I.: Cache-Collision Timing Attacks against AES. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 201–215. Springer, Heidelberg (2006), `http://www.springerlink.com/content/v34t50772r87g851/fulltext.pdf`
8. Brumley, B.B., Hakala, R.M.: Cache-Timing Template Attacks. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 667–684. Springer, Heidelberg (2009)
9. Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: FOCS, pp. 293–302. IEEE Computer Society, Los Alamitos (2008)
10. Fluhrer, S.R., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 1–24. Springer, Heidelberg (2001)

11. Golić, J.D.: Iterative Probabilistic Cryptanalysis of RC4 Keystream Generator. In: Clark, A., Boyd, C., Dawson, E.P. (eds.) ACISP 2000. LNCS, vol. 1841, pp. 220–233. Springer, Heidelberg (2000), http://www.springerlink.com/content/l1510525523352p4/fulltext.pdf
12. Hoch, J.J., Shamir, A.: Fault Analysis of Stream Ciphers. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 240–253. Springer, Heidelberg (2004)
13. Khovratovich, D., Biryukov, A., Nikolic, I.: Speeding up collision search for byte-oriented hash functions. In: Fischlin, M. (ed.) CT-RSA 2009. LNCS, vol. 5473, pp. 164–181. Springer, Heidelberg (2009)
14. Knudsen, L.R., Meier, W., Preneel, B., Rijmen, V., Verdoolaege, S.: Analysis Methods for (Alleged) RC4. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 327–341. Springer, Heidelberg (1998), http://www.springerlink.com/content/tyqqary0p5kfw7tp/fulltext.pdf
15. Leander, G., Zenner, E., Hawkes, P.: Cache Timing Analysis of LFSR-Based Stream Ciphers. In: Parker, M.G. (ed.) Cryptography and Coding 2009. LNCS, vol. 5921, pp. 433–445. Springer, Heidelberg (2009)
16. Maximov, A., Khovratovich, D.: New State Recovery Attack on RC4. In: Wagner, D. (ed.) CRYPTO 2008. LNCS, vol. 5157, pp. 297–316. Springer, Heidelberg (2008)
17. Munkres, J.: Algorithms for the Assignment and Transportation Problems. Journal of the Society for Industrial and Applied Mathematics 5, 32–38 (1957), http://www.jstor.org/stable/2098689
18. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: The Case of AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 1–20. Springer, Heidelberg (2006), http://www.springerlink.com/content/f52x1h55g1632l17/fulltext.pdf
19. Page, D.: Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. Technical Report CSTR-02-003, Department of computer science, university of Bristol (2002), http://www.cs.bris.ac.uk/Publications/Papers/1000625.pdf
20. Salez, J., Shah, D.: Belief propagation: An asymptotically optimal algorithm for the random assignment problem. Math. Oper. Res. 34(2), 468–480 (2009)
21. Sepehrdad, P., Vaudenay, S., Vuagnoux, M.: Discovery and exploitation of new biases in rc4. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 74–91. Springer, Heidelberg (2011)
22. Tromer, E., Osvik, D.A., Shamir, A.: Efficient cache attacks on aes, and countermeasures. J. Cryptology 23(1), 37–71 (2010)
23. Zenner, E.: A Cache Timing Analysis of HC-256. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 199–213. Springer, Heidelberg (2009)

A A Numerical Example of the Algorithm With Idealized Cache

Let us imagine that the generation of a cipher byte gives the following information:

$$a = \texttt{0xf3}, \quad b = \texttt{0xd9}, \quad c = \texttt{0x43}, \quad out = \texttt{0x1c}$$

The candidates values for j, $S[a]$, $S[b]$ and $S[c]$, given by the preceding attack steps, are:

$$\begin{aligned} j &\in \{\texttt{0x2f}, \texttt{0xc6}\} \\ S[a] &\in \{\texttt{0x7d}, \texttt{0xaa}\} \\ S[b] &\in \{\texttt{0x14}, \texttt{0x1c}, \texttt{0x5d}, \texttt{0x99}\} \\ S[c] &\in \{\texttt{0x1c}, \texttt{0x5c}, \texttt{0xd4}\} \end{aligned}$$

Let us suppose first that the table elements are used in the order (a, b, c):

- if the value of j is `0x2f`, the constraints given above are:

  $$\texttt{0x2f} + S[a] = \texttt{0xd9}, \quad S[a] + S[b] = \texttt{0x43}, \quad S[c] = \texttt{0x1c}$$

  The imposed values are therefore $S[a] = \texttt{0xaa}$, $S[b] = \texttt{0x99}$ and $S[c] = \texttt{0x1c}$. At this step of the attack, these three values are considered possible: this may correspond to what really happened when generating the keystream byte;
- using analogous computations for $j = \texttt{0xc6}$, we deduce that the value of $S[a]$ must be `0x13`, which is not possible.

We make similar computations for the order (a, c, b) and finally obtain two candidates for what happened during the step of PRGA:

- if the elements are used in the order (a, b, c), the corresponding values must have been $j = \texttt{0x2f}$, $S[a] = \texttt{0xaa}$, $S[b] = \texttt{0x99}$ and $S[c] = \texttt{0x1c}$;
- otherwise, they must be $j = \texttt{0xc6}$, $S[a] = \texttt{0x7d}$, $S[b] = \texttt{0x1c}$ and $S[c] = \texttt{0x5c}$.

We can now update the candidate values for j and the three table elements: after this pseudo-random generation step, the value of j can be `0xd9` or `0x43` (as we were not able to determine the order of use of the table elements with certainty); we also take the swap operation into account, which leads us to:

$$\begin{aligned} j &\in \{\texttt{0x43}, \texttt{0xd9}\} \\ S[a] &\in \{\texttt{0x5c}, \texttt{0x99}\} \\ S[b] &\in \{\texttt{0x1c}, \texttt{0xaa}\} \\ S[c] &\in \{\texttt{0x1c}, \texttt{0x7d}\} \end{aligned}$$

We proceed like above to determine which values are possible or not and so on.

B Proofs of Soundness

B.1 Algorithm for the Idealized Case

The proof of soundness of the algorithm for the idealized case is very straightforward:

- at the beginning of the attack, all the elements of the tables $S_{values}[i]$ have the value 1. In particular, for each i, the value v corresponding to the secret permutation table used by PRGA is associated to the value 1. Moreover, the value of j is well-known;
- we now suppose that before an attack step the tables $S_{values}[i]$ and j_{values} contain the value 1 for the value v corresponding to the corresponding value of the secret state of the encryption process. Consequently, when trying the good order for a, b and c and the good value for j, the imposed values $S[a]$, $S[b]$ and $S[c]$, which correspond to the values of the secret state, all will have the value 1 in S_{values}: this try will be considered as successful. Finally, the attack algorithm will update S_{values} and j_{values} to take this attack step into account. Since the guess corresponding to the real secret state has been considered as successful, the values for j, $S[a]$, $S[b]$ and $S[c]$ will be marked with the boolean 1, all other values of S_{values} remaining untouched as in PRGA: after the attack step, the tables S_{values} and j_{values} also associate the boolean 1 to the values of $S[i]$ and j corresponding to the secret state used by PRGA.

Using the axiom of induction, we obtain that during each step of the attack, the tables S_{values} and j_{values} associate the boolean 1 to the values corresponding to the secret state used by PRGA, so that the $S_{values}[i]$ and j_{values} always have at least one boolean with value 1, and that when all of these tables contain only one boolean with value 1, this boolean is the one corresponding to the value of the secret stage used by PRGA.

B.2 Probabilistic Algorithm for Realistic Caches

An Impractical Straightforward Algorithm. The proof of soundness of the probabilistic algorithm is a little more complicated. To do so, we will introduce the following algorithm:

- we consider the space of couples (S, j), where S is a permutation over $\mathbf{F}_{256}$ and j an element of $\mathbf{F}_{256}$;
- we use a `256.256!`-boolean table C_{values}, using for each couple (S, j) the same meaning as in the algorithm for the idealized case: if the value is 1, the couple is considered as probable as it explains the successive observed cipher bytes and cache lookups; if it is 0, the couple has failed to explain the observations and we know it to be impossible;
- during an attack step, we begin a new table C^*_{values}, filled with the value 0. For each couple (S, j) that has the value 1 in the table C_{values} and that explains the observed cipher byte and cache lookups, we compute the updated couple (S^*, j^*) with PRGA and give it the value 1 in C^*_{values};

– at the end of the attack step, we replace the table C_{values} by the table C^*_{values}.

As for the algorithm of the idealized case, the proof of soundness of this algorithm is very simple to do using induction, so we will not discuss it further. However, this algorithm is very inefficient, as the number of possible couples in the first steps if far too high to allow a computer to enumerate them.

The Probabilistic Algorithm, an Improvement of the Straightforward Algorithm. We introduce a 256x256-floating point table S_p and a 256-floating point table j_p, which values are computed as follows:

$$\forall i, \forall v, S_p[i][v] = \frac{\#\{(S,j)|C_{values}[(S,j)] = 1, S[i] = v\}}{\#\{(S,j)|C_{values}[(S,j)] = 1\}}$$
$$\forall v, j_p[v] = \frac{\#\{(S,j)|C_{values}[(S,j)] = 1, j = v\}}{\#\{(S,j)|C_{values}[(S,j)] = 1\}}$$

We will now prove that the evolution of S_p and j_p is the same as the one of S_{probas} and j_{probas}. First, it is clear that, at the beginning of the attack:

$$\forall i, \forall v, S_p[i][v] = \frac{256 \cdot 255!}{256 \cdot 256!} = 1/256$$
$$\forall v, j_p[v] = \frac{1 \cdot 256!}{256 \cdot 256!} = 1/256$$

Now, choosing an order (a, b, c), a value for j and for o_1, o_2 and being given the values of i, b_δ, c_δ and out, we can compute the number P_g of probable couples (S, j) concerned by this guess. We obtain, assuming that the different elements of the permutation are independent (the same assumption as what was done in the probabilistic algorithm) and using the constraints on $S[a]$, $S[b]$ and $S[c]$ given by the structure of PRGA, an equation analogous to the one giving $\mathbf{P}$(guess):

$$P_g = j_p[v] \cdot S_p[a][b - v] \cdot S_p[b][c - b + v] \cdot S_p[c][out] \cdot N$$

where N is the number of probable couples.

For given values of i and v, the couples (S', j') of C^*_{values} can have two origins:

– either probable couples where PRGA does not affect $S[i]$. Assuming again the independence of all concerned random variables, their number is:

$$\left(N - \sum_{\text{guesses imposing } S[i]} P_g\right) \cdot S_p[i][v]$$

– or couples where PRGA imposes $S[i] = v$. Their number is given by:

$$\sum_{\text{guesses imposing } S[i]=v} P_g$$

Adding these two contributions, we obtain that the evolution of S_p is the same as that of S_{probas}, except for a coefficient $\sum_{\text{any guesses}} P_g/N$, which is the proportion of couples (S, j) that are compatible with the observed cipher byte and cache lookups, and that corresponds to the renormalization of $\mathbf{P}(\text{guess})$. Using the axiom of induction, this achieves proving that the behavior of S_p is the same as the one of S_{probas} in the probabilistic algorithm.

For the behavior of j_p the proof is the same, as previously. In conclusion, the behavior of the tables S_{probas} and j_{probas} in the probabilistic algorithm is a consequence of the behavior of the table C_{values} in the algorithm introduced here. Furthermore, we know that if this algorithm only has one probable couple (S, j), then this couple is the right one (we also know that the right couple is always considered as probable). Consequently, we can deduce that if the tables $S_{probas}[i]$ and j_{probas} contain only one value 1.0 and the other elements have the value 0.0, then the value with probability 1 is the one of the secret state used by PRGA, which concludes this proof.

Further Thoughts on the Practical Results. First, it is important to stress that despite the previous proofs of soundness of the algorithms presented in this paper, it was neither possible to estimate the speed of convergence to the solution, nor to prove the termination of these algorithms. The only hints according to this problem are given considering arguments from information theory, since the entropy of the secret state is easy to evaluate to the first order, as is the entropy of every cipher byte and cache lookup. Furthermore, the number of cases when the probabilistic algorithm fails to give the right secret state do not go against the soundness of the algorithm, but rather show that the choice of stopping the attack before only one probable state remains is done with the risk of giving a wrong answer. Finally, as the number of possible couples is less than `256x256!`, the number of bits needed to store the floating-point values of the tables S_{probas} and j_{probas} is $\log_2(256 \cdot 256!) \approx 2056$. To improve the speed of the attack, we used basic floating-point numbers, which is also a probable cause for the failure of the attack in some cases.