Cryptogr. Commun. (2010) 2:1–17
DOI 10.1007/s12095-009-0013-3

On the structural weakness of the GGHN stream cipher

Aleksandar Kircanski · Amr M. Youssef

Received: 24 January 2009 / Accepted: 9 September 2009 / Published online: 18 September 2009

Abstract GGHN is an RC4-like stream cipher designed to make use of today's common 32-bit processors. It is 3–5 times faster than RC4. According to its designers, one of the sources of GGHN's high security is the large size of its secret internal state, which totals 8240 bits. In this paper we show that if an attacker can obtain 2064 specific bits of this internal state, then the attacker can deduce the remaining state bits with limited computation, effectively reducing the secret internal state size by approximately a factor of 4. We then present a fault analysis attack that allows the cryptanalyst to obtain these critical 2064 bits. The whole procedure effectively breaks GGHN using 257×255 induced faults, 2 keystream words for each of these faults, around 257 non-faulted keystream words and negligible computational time.

Keywords Cryptanalysis · Fault analysis · Generalized RC4 ciphers · Stream ciphers

1 Introduction

RC4 [20] is a high-speed software-oriented stream cipher that was designed in 1987 by Ron Rivest and kept as a trade secret until it was leaked by an anonymous Internet post in 1994. According to [8], its design approach may have originated from the table-shuffling principle [15]. Due to its high speed and simplicity, RC4 has been implemented in popular protocols such as SSL and WEP.

RC4 is in fact a family of stream ciphers, parameterized by an integer n. The size of all internal state words, as well as the keystream words produced, is n bits. The

A. Kircanski · A. M. Youssef (✉)
Concordia Institute for Information Systems Engineering, Concordia University,
Montreal, Quebec, H3G 1M8, Canada
e-mail: youssef@ciise.concordia.ca

A. Kircanski
e-mail: a_kircan@ciise.concordia.ca

internal state itself consists of a table S of size 2^n and two pointers, one of which is public and the other is secret and pseudorandom. At the output step of each RC4 iteration, a word from the S table, indexed pseudorandomly, is returned. To maintain randomness, two elements of the table are also swapped. An important property of the internal state table, S, is that at each point in the execution of the cipher, it represents a permutation over Z_{2^n}. Usually RC4 with $n = 8$ is used, operating on 8-bit values with a table size of 256 entries.

The security of RC4 has been the subject of extensive research. For example, the key schedule of RC4 was examined in [7, 16, 17, 21, 23, 27], some distinguishing attacks were presented in [6, 8, 17, 18, 24] and internal state recovery attacks were presented in [12, 19].

With the current widespread use of 32-bit and 64-bit processors, the RC4 design is showing signs of obsolescence, since its 8-bit orientation does not take full advantage of these newer processors. It should also be noted that while the majority of currently deployed smartcards are based on 8-bit processors, several companies started to offer 32-bit platform. For example, STMicroelectronics has recently announced the delivery of more than 10 million ST22 32-bit smartcards for 3G mobile applications using SIM cards, as well as to pay-TV, IT security and banking applications. Other examples for 32-bit smartcard processors include the ARM922T^{TM} RISC-based processor family produced by ARM and the AT91SC processor produced by Atmel.

Directly adapting RC4 to work with 32-bit words by setting $n = 32$ is impractical, since the internal table, S, would grow to an unacceptable 2^{32} words. To address this problem, several generalizations of RC4-like stream ciphers have been proposed (e.g., RC4A [23], VMPC [28], NGG [22] and GGHN [9]). In particular, Gong et al. addressed this problem through a modified design in which the S table no longer needs to be permutation, and therefore can be much smaller than 2^n, even though the entries in the table are n bits long. This modified cipher was later named GGHN(n,m) in [26], where 2^n is the S table size, and m is the word size in bits. Instead of swapping two elements in the table, the update step now involves overwriting a pseudorandomly indexed element in S with a new pseudorandom value. This approach maintains the randomness of the GGHN internal table S, which now contains only 2^n out of 2^m possible m-bit words. For a 32-bit architecture, the recommended parameter values [9] are $n = 8$ and $m = 32$.

According to [9], the secret internal state size of GGHN(8,32) is 8240 bits, since it consists of a table of size 256×32 bits, one 32-bit counter k and two 8-bit counters i and j. If i is excluded because it is public, then the internal state size is 8232 bits, which is huge compared to the internal state size of RC4, which is approximately 1700 bits. This significant increase in state size encouraged the authors in [9] to conclude that internal state recovery for GGHN(8,32) is much harder than for RC4.

In this paper, we show that if an attacker is able to obtain the 8 least significant bits of each internal state word, and is able to peek at the next 257 keystream words produced by that state, then the attacker can form a set of linear equations and retrieve the remaining 6168 state bits using a negligible amount of computation. Throughout the paper, we consider only GGHN(8,32), but the attack is applicable for any values of n and m.

In order to obtain these critical $8 \times 256 = 2064$ internal state bits, we use fault analysis [3, 4, 11]. In this technique, an attacker is assumed to be able to introduce physical disturbances into the electric circuits of the device that performs the

encryption and thereby cause faults in its memory. The cryptanalyst then observes the outputs produced by the resulting faulted encryptions, and uses these to deduce secret information such as key bits or internal state bits.

For the purposes of clarity in what follows, we introduce names for different fault analysis models. If the attacker can choose the memory location to be changed, but cannot control the new value in this location, we call this the *chosen-location* fault model. If the attacker cannot choose the memory location where a fault is induced, and also cannot choose the new value for this location, we call this the *random-location* fault model. These two models were used in [4] and [11], respectively, where fault analysis of RC4 was studied. In this paper, we adopt the *chosen-location* model.

The attack we present in this paper can roughly be divided into two parts. In the first part, the secret internal state at some point in time is partially recovered by means of fault analysis. In the second part, the remaining unknown state bits are deduced. The second part employs a method that involves linear equations and requires a negligible amount of computational time.

The paper is organized as follows. The specification of GGHN and previous cryptanalysis is given in Section 2. A brief introduction to fault analysis is given in Section 3. The notation used throughout the paper is introduced in Section 4. Detailed descriptions of the first and second parts of our attack are given in Sections 5 and 6, respectively. In Section 7 these two parts are combined, and the complexity of the overall procedure is discussed. Finally, Section 8 concludes the paper.

2 The GGHN stream cipher

GGHN(n,m) is a family of ciphers parameterized by two integers, n and m, where n determines the table size, $N = 2^n$, and m is the internal word size. All operations on table indices are performed modulo N, and all operations on words are performed modulo $M = 2^m$. As mentioned, for 32-bit processors it is recommended to use GGHN(8,32), for which the the table size is 256 and the word size is 32 bits.

GGHN consists of two parts: the key scheduling algorithm (KSA) and the pseudorandom generation algorithm (PRGA). The KSA, which depends on a secret key, K, of variable length, first fills the S table with 2^n publicly known values (denoted a_i in Fig. 1), and then randomizes the table entries via repeated additions involving table entries and words from the secret key. Each iteration of the PRGA updates counters, outputs a pseudorandom element of the table masked by special m-bit word, k, and then updates the table. Unlike RC4, the GGHN PRGA does not have a swap step. Another difference is that GGHN makes use of the auxiliary word k, which is used to mask both the output and the values written to S. The algorithm is shown in Fig. 1.

Concerning previous GGHN analysis, in 2006, Paul et al. [24] showed that the least significant bit of the keystream output is biased, due to a bias-inducing state which occurs with probability 2^{-16}. The distinguisher that exploits this bias requires around $2^{32.89}$ keystream words. Tsunoo et al. [26] provided another distinguisher that does not rely on the keystream output for a single key, but instead uses keystream outputs for multiple keys. The first two keystream words for approximately 2^{30} keys are needed to distinguish GGHN output from random keystreams.

KSA

Initialization:

For $i = 0, \ldots, N-1$

$S[i] = a_i$

$j = k = 0$

Scrambling:

Repeat r times

For $i = 0, \ldots, N-1$

$j = (j + S[i] + K[i \bmod l]) \bmod N$

Swap$(S[i], S[j])$

$S[i] = (S[i] + S[j]) \bmod M$

$k = (k + S[i]) \bmod M$

PRGA

Initialization:

$i = 0$

$j = 0$

Loop:

$i = (i + 1) \bmod N$

$j = (j + S[i]) \bmod N$

$k = (k + S[j]) \bmod M$

out $= (S[(S[i] + S[j]) \bmod N] + k) \bmod M$

$S[t] = (k + S[i]) \bmod M$

Fig. 1 GGHN(n,m), where $N = 2^n$ and $M = 2^m$

As argued by Rose and Hawkes in [10], it should be noted that unlike the block ciphers case where being able to identify some distinguishing characteristic of the output may lead to a key recovery attack, the existence of distinguishing attacks against stream ciphers, in general, has no security implications in the context of use of the stream cipher. Furthermore, the amount of data required to perform a distinguishing attack against a given stream cipher is unrelated to the key length of the cipher. For example, according to [10], RC4 is still considered perfectly adequate for encryption in SSL and TLS, with 128-bit keys, despite the fairly large number of published distinguishing attacks against it. Also, the existence of distinguishing attacks of order 2^{32} blocks against DES in OFB or CTR mode did not prevent NIST from recommending both modes for use with AES with 256-bit keys, despite the fact that the same straightforward distinguishing attacks have complexity 2^{64} blocks. In fact, these attacks do not really reflect an inherent weakness in the cipher; they are only used as justification for the rule of thumb that one should not generate too much keystream before rekeying.

In the same context, the above two distinguishing attacks on GGHN, do not yield any usable information about the state of the cipher generator, and they cannot be used to attack the generator itself. Thus, the attack presented in our paper, to the best of our knowledge, can be considered the only published internal state recovery attack against GGHN.

3 Fault analysis

Cryptanalytic attacks can be divided in two classes: pure mathematical attacks and side channel attacks. Pure mathematical attacks, such as differential or linear cryptanalysis, are traditional cryptanalytic techniques that rely only on known or chosen input-output pairs of the encryption function, and exploit the inner structure of the cipher to reveal secret key information. In side channel attacks, it is assumed that the attacker has some access to the encryption device, either by being able to make measurements with respect to time [14] or power consumption [13], or by being able to induce errors in the memory of the device (fault analysis) [3]. The additional information gained by utilizing such a "side channel" is then combined with methods that exploit the inner structure of the cipher to reveal the secret key.

In fault analysis, an attacker applies some kind of physical influence on the internal state of the cryptosystem, such as ionizing radiation that flips random bits in memory. By careful study of the results of computations under such faults, an attacker can retrieve information about the secret key. Smartcards are devices that are especially susceptible to these kinds of attacks. Fault attacks were first introduced in 1996 by Boneh et al. [3], who described attacks that targeted the RSA public key cryptosystem by exploiting a faulted Chinese Remainder Theorem computation in order to factor the RSA modulus n. Subsequently, fault analysis attacks were extended to symmetric systems such as DES [2] and later to AES [5] and other primitives. Fault analysis attacks became an even more serious threat after cheap and low-tech methods of inducing faults and transient faults were presented (e.g., [1, 25]).

Hoch and Shamir [11] addressed the problem of fault analysis of stream ciphers in 2004. Ciphers based on LSFR's, LILI-128, SOBER-t32 and also RC4 were analyzed, and it has been shown that none of these constructions are secure in the *random-location* fault model. As for RC4, the key recovery attack required 2^{16} faults and 2^{26} keystream words.

In [4], RC4 was assessed in the *chosen-location* model using an interesting idea, namely inducing faults in order to push the cipher into a specific state called a Finney state, and using this to determine the secret RC4 internal state. In its normal mode of operation, RC4 cannot enter a Finney state. However, once RC4 is artificially pushed into a Finney state, it cannot exit, the length of a table cycle becomes very small, and, what is more, the secret S table can be read solely by observing the keystream output. This attack required 2^{16} *chosen-location* faults. Another more advanced fault analysis attack on RC4 requiring 2^{10} faults was also introduced in the same paper.

As mentioned above, in this paper we adopt the *chosen-location* fault analysis model. Furthermore, we assume that the injected faults are transient in nature, i.e., not permanent, and hence the attacker can use the same cryptographic device (e.g., the same smartcard) to perform the whole attack. Such faults might be injected using optical fault induction techniques described in [25].

4 Notation

Throughout the rest of the paper, unless stated otherwise, by GGHN we mean GGHN(8,32). We say that GGHN is at *time t*, if, so far, t complete GGHN PRGA

iterations have been executed. Consequently, we use S_t, k_t, j_t and i_t to denote the elements of the GGHN internal state at time t. In particular, S_0 denotes the S table just after the KSA and before any PRGA steps and z_t denotes the keystream word produced while the cipher was at time t. For example, z_0 denotes the first keystream word produced.

Unless specified otherwise, "+" denotes modular addition, where the modulus value is determined from the context, i.e., by the size of the smallest operand. For example, $x + y$ denotes addition *mod* 2^{32} if both x and y are 32 bits in length. If x is 8 bits long and y is 32 bits long, $x + y$ represents an 8-bit value obtained by *mod* 2^8 addition of x and the 8 least significant bits of y.

For a 32-bit value x, x^L denotes the 8 least significant bits of x, and x^R denotes the 24 most significant bits of x.

Certain values in the S table do not influence the evolution of the internal state of GGHN. This can be utilized for cryptanalysis. We give these values a specific name.

Definition 1 Let the GGHN cipher be at step t_0. An index $0 \leq a \leq 255$ into the S table is called *irrelevant with respect to time* t_0 if in future steps $S[a]$ is updated before it is touched by indices i and j, i.e., if there exists $t_1 \geq t_0$ such that $(S_{t_1}[i_{t_1+1}] + S_{t_1}[j_{t_1+1}]) \bmod 256 = a$, and for all t such that $t_0 + 1 \leq t \leq t_1 + 1$, $i_t, j_t \neq a$.

If some index $0 \leq a \leq 255$ is irrelevant with respect to a certain time, $S[a]$ will be changed before its value influences any future internal state value, which justifies the term "irrelevant". If the time with respect to which a position is irrelevant is clear from the context, the position will simply be called *irrelevant*. All other S values will be called *relevant*.

It is interesting to note our experimental results in which we generated 1000 random GGHN states, and for each one counted the number of irrelevant indices. For these states, the expected number of irrelevant indices was 69 out of 256.

5 Internal state size reduction

In this section, we show how to determine the whole internal state of GGHN, given the 8 least significant bits of every internal state word.

More precisely, we show that the values

$$k_0^R, S_0^R[0], S_0^R[1], \ldots, S_0^R[255] \tag{1}$$

can be calculated if we are given

$$k_0^L, S_0^L[0], S_0^L[1], \ldots, S_0^L[255] \tag{2}$$

along with the 257 keystream words, $z_0, \ldots, z_{256}$.

On the other hand, if the values k_t^L, $S_t^L[0]$, $S_t^L[1], \ldots, S_t^L[255]$ are given for $t > 0$, the same procedure can be applied in order to recover the remaining bits of the internal state at time t. In this case, however, it is necessary to provide j_t as well, since it is not necessarily equal to 0, as it was in the case $t = 0$.

Observation 1 *At any time t, the values* j_{t+1}, k_{t+1}^L, $S_{t+1}^L[0], S_{t+1}^L[1], \ldots, S_{t+1}^L[255]$ *depend only on* j_t, k_t^L, $S_t^L[0], S_t^L[1], \ldots, S_t^L[255]$, *and not on* k_t^R, $S_t^R[0], \ldots, S_t^R[255]$.

Proof Let + denote addition modulo 256. From the cipher specification we have:

$$j_{t+1} = j_t + S_t^L[i_{t+1}]$$
$$k_{t+1}^L = k_t^L + S_t^L[j_{t+1}]$$
$$S_{t+1}^L[m] = \begin{cases} S_t^L[m], & \text{if } m \neq S_t^L[i_{t+1}] + S_t^L[j_{t+1}]; \\ S_t^L[i_{t+1}] + k_{t_1}^L, & \text{otherwise} \end{cases}$$

The proof follows by noting that j and i are 8-bit values. □

From Observation 1, we conclude that if an attacker is given only (2), then the attacker can discover the 8 least significant bits of every internal word for all subsequent states. Therefore, if we ignore the 24 most significant bits of each word, a truncated version of the cipher is in operation, using the same algorithm as the original cipher. For this truncated cipher, 8-bit output words will be equal to the 8 least significant bits of the output words for the full 32-bit version. This is, indeed, an unwanted property.

Now assume that the attacker knows (2) (and therefore also knows j_t, k_t^L, $S_t^L[0], S_t^L[1], \ldots, S_t^L[255]$ for $t > 0$), as well as $z_0, \ldots, z_{256}$. According to the GGHN specification, the keystream word produced at time t is

$$z_t = S_t[(S_t[i_{t+1}] + S_t[j_{t+1}]) \bmod 256] + k_{t+1} \tag{3}$$

For each $t = 0, 1, \ldots, 256$, the attacker can use known values to calculate the index value $a_t = (S_t[i_{t+1}] + S_t[j_{t+1}]) \bmod 256$, and then form the following system of linear equations over $Z_{2^{24}}$:

$$S_t^R[a_t] + k_{t+1}^R + \sigma_t = z_t^R, \ t = 0, 1, \ldots, 256 \tag{4}$$

where, for each equation, σ_t represents a carry corrector, defined as

$$\sigma_t = \begin{cases} 0, & \text{if } S_t^L[a_t] + k_t^L \leq 255; \\ 1, & \text{otherwise} \end{cases}$$

To express the system above in terms of values from the internal state at time 0, $S_t^R[a_t]$ and k_{t+1}^R have to be replaced by S_0^R and k_0^R values, based on the way these values were changed during the first t steps of cipher execution. According to the cipher specification, we have:

$$k_t^R = k_{t-1}^R + S_{t-1}^R[j_t] + \delta_t, \ t = 1, 2, \ldots, 257 \tag{5}$$

where δ_t represents a carry correction which is set to 1 if $k_{t-1}^L + S_{t-1}^L[j_t]$ produces a carry, and is set to 0 otherwise. As for S values, for each step $t = 1, \ldots, 256$, if $S_t[m]$ was updated during step $t - 1$, that is, if $m = S_{t-1}^L[i_t] + S_{t-1}^L[j_t]$, we have

$$S_t^R[m] = S_{t-1}^R[i_t] + k_t^R + \theta_t \tag{6}$$

where θ_t is the usual carry corrector. If $S_t[m]$ was not updated during step $t - 1$, we simply have

$$S_t^R[m] = S_{t-1}^R[m] \tag{7}$$

Formally, we have a system that consists of equations (4), (5), (6) and (7). It can be naturally reduced to a system of 257 linear equations by expressing the values k_t and

$S_t[m]$, $m = 0, \ldots, 255$, in terms of k_0 and $S_0[m]$, $m = 0, \ldots, 255$, by applying (5), (6) and (7) and then substituting the resulting expressions into (4). This yields a system of 257 equations in 257 unknown variables, which can easily be solved. If some of the equations turn out to be linearly dependent, more keystream words can be utilized to produce more equations.

A toy example of this internal state reduction procedure is provided in Appendix A.

6 Recovering the 8 least significant bits of each relevant internal state word

In this section, we show how it is possible to find the values

$$k_0^L,\ S_0^L[0],\ S_0^L[1],\ \ldots,\ S_0^L[255] \tag{8}$$

that are used in the attack presented in the previous section. We do this by means of fault analysis. As mentioned above, we adopt a *chosen-location* model. Recall that this means an attacker is able to choose a memory location at which a fault is induced, but cannot choose the new value to be written into this memory location.

Unless stated otherwise, all words in this section will be reduced to 8 bits, i.e., we consider only the 8 least significant bits of all words we use. For example, $S[5]$ will denote only the 8 least significant bits of the corresponding 32-bit word.

First, we provide a procedure for recovering

$$j_{t+1} \quad \text{and} \quad S_t[i_{t+1}] + S_t[j_{t+1}] \tag{9}$$

for any time t. The idea is to induce a fault in S at some known location m, and then check how subsequent keystream words react to this change. If the next keystream word does not differ from the original one, it means that the next keystream value did not depend on $S[m]$. In other words, m is not equal to i, j or $S[i] + S[j]$. Thus, discrepancies between the original keystream and the keystream produced by a faulted internal state reveal some information about the internal state of the cipher. If the process is repeated for $m = 0, 1, \ldots, 255$, the values in (9) can be determined for time t. As will be shown, knowing the values in (9) for $t = 0, 1, \ldots, 256$ leads to complete recovery of (8).

Formally, suppose that when GGHN is at time t, an attacker faults $S_t[m]$, where $m \neq i_{t+1}$. Let z_t denote the keystream word produced during the iteration of the cipher from time t to $t+1$. Also, let z'_t denote the keystream word produced by GGHN with an internal state that was faulted at some previous time. The attacker observes and compares the keystreams $z_t, z_{t+1}, \ldots$ and $z'_t, z'_{t+1}, \ldots$. In the following, we discuss the impact of the induced fault on these keystreams. More precisely, the following cases show how the two keystream values z'_t and z'_{t+1} are affected by the relationships among m, i_{t+1}, j_{t+1} and $S_t[i_{t+1}] + S_t[j_{t+1}]$:

1. $i_{t+1} \neq j_{t+1} \neq S_t[i_{t+1}] + S_t[j_{t+1}]$:
   (a) $m \neq j_{t+1}$ and $m \neq S_t[i_{t+1}] + S_t[j_{t+1}]$: the next keystream word will not change, i.e., $z_t = z'_t$
   (b) $m = S_t[i_{t+1}] + S_t[j_{t+1}]$: in this case, $z_t \neq z'_t$, but $z_{t+1} = z'_{t+1}$. In other words, an irrelevant index in S was faulted

(c) $m = j_{t+1}$: the next two keystream words will change, i.e., $z_t \neq z'_t$ and $z_{t+1} \neq z'_{t+1}$. Other possibilities for this case are very improbable and thus have no practical significance. For further discussion of these possibilities, see Appendix B.

Thus, if a fault is induced at $S_t[m]$, $m = 0, 1, \ldots, i_t - 1, i_t + 1, \ldots, 255$, the value z'_t will differ from z_t for exactly 2 out of 255 m values.

2. $i_{t+1} = j_{t+1}$, but $i_{t+1} \neq S[i_{t+1}] + S[j_{t+1}]$:

(a) $m \neq S_t[i_{t+1}] + S_t[j_{t+1}]$: $z_t = z'_t$
(b) $m = S_t[i_{t+1}] + S_t[j_{t+1}]$: $z_t \neq z'_t$, but $z_{t+1} = z'_{t+1}$

Thus, if a fault is induced at $S_t[m]$, $m = 0, 1, \ldots, i_t - 1, i_t + 1, \ldots, 255$, the value z'_t will differ from z_t for exactly 1 out of 255 m values.

3. $i_{t+1} \neq j_{t+1}$, but $j_{t+1} = S[i_{t+1}] + S[j_{t+1}]$:

(a) $m = j_{t+1} = S[i_{t+1}] + S[j_{t+1}]$: $z_t \neq z'_t$ and $z_{t_1} \neq z'_{t_1}$
(b) $m \neq j_{t+1} = S[i_{t+1}] + S[j_{t+1}]$: $z_t = z'_t$

Again, if a fault is induced at $S_t[m]$, $m = 0, 1, \ldots, i_t - 1, i_t + 1, \ldots, 255$, the value z'_t will differ from z_t for exactly 1 out of 255 m values.

4. $i_{t+1} \neq j_{t+1}$, but $i_{t+1} = S[i_{t+1}] + S[j_{t+1}]$:

(a) $m = j_{t+1}$: $z_t \neq z'_t$ and $z_{t+1} \neq z'_{t+1}$
(b) $m \neq j_{t+1}$: $z_t = z'_t$

As in the previous two cases, if a fault is induced at $S_t[m]$, $m = 0, 1, \ldots, i_t - 1, i_t + 1, \ldots, 255$, the value z'_t will differ from z_t for exactly 1 out of 255 m values.

5. $i_{t+1} = j_{t+1} = S[i_{t+1}] + S[j_{t+1}]$:

(a) $m \neq j_{t+1}$: since the fault does not affect any of the values that the next keystream word depends on, $z_t = z'_t$.

In this case, if a fault is induced at $S_t[m]$, $m = 0, 1, \ldots, i_t - 1, i_t + 1, \ldots, 255$, the value z'_t will be equal to z_t for all of the 255 m values.

The attacker's goal is to reverse the above analysis, i.e., based on the observed differences between z and z', the attacker needs to recover the relationship among m, i_{t+1}, j_{t+1} and $S_t[i_{t+1}] + S_t[j_{t+1}]$. Since m is known to the attacker, this will leak information about hidden j_{t+1} and $S_t[i_{t+1}] + S_t[j_{t+1}]$. To completely recover these two values, it is necessary to induce a fault at every $S_t[m]$, $m = 0, 1, \ldots, i_t - 1, i_t + 1, \ldots, 255$.

The following procedure, which we call `fault(t)`, returns a tuple $(j_{t+1}, S_t[i_{t+1}] + S_t[j_{t+1}])$. With small probability, it will also be possible for the algorithm to return two candidates for this tuple, as will be explained below.

1. For $m = 0, 1, \ldots, t - 1, t + 1, \ldots, 255$, do:

1.1 Induce a fault at $S_t[m]$
1.2 Iterate the cipher 2 times and store z'_t and z'_{t+1}

2. Let SM be the set of m values for which $z'_t \neq z_t$
3. If $|SM| = 0$: return $(t + 1, t + 1)$

4. If $|SM| = 1$: let m_0 denote the single member of SM. If for that m_0, $z'_{t+1} = z_{t+1}$, return $(t+1, m_0)$. If not, return two candidates: (m_0, m_0) and $(m_0, t+1)$
5. If $|SM| = 2$: let m_0 denote a value from SM for which $z'_{t+1} = z_{t+1}$, and let m_1 denote the other element of SM. Return (m_1, m_0)

The correctness of the procedure `fault(t)` can be justified as follows:

According to cases 1, 2, 3, 4 and 5 from the previous discussion, $|SM| \leq 2$. If $|SM| = 0$, case 5 occurred and j_{t+1} and $S[i_{t+1}] + S[j_{t+1}]$ are equal to $i_{t+1} = t+1$. Similar straightforward reasoning applies when $|SM| = 2$. If $|SM| = 1$, it means that one of cases 2, 3 or 4 occurred. However, for $m_0 \in SM$, only in case 2 does $z'_{t+1} = z_{t+1}$ hold. This makes case 2 distinguishable from 3 and 4, and the validity of the returned tuple $(t+1, m_0)$ stems from the assumption that we are in case 2. Since there is no way to distinguish between cases 3 and 4, an alternative that involves returning two candidates is provided. Namely, in both cases 3 and 4, $j_{t+1} = m_0$. As for $S_t[i_{t+1}] + S_t[j_{t+1}]$, in case 3 it is equal to m_0, and in case 4 it is equal to $i_{t+1} = t+1$, which justifies returning two proposed candidates.

The probability that case 3 or 4 will occur can be calculated by modelling i_{t+1}, j_{t+1} and $S[i_{t+1}] + S[j_{t+1}]$ as random numbers. Requiring that $i_{t+1} \neq j_{t+1}$ and $S[i_{t+1}] + S[j_{t+1}]$ be equal either to i_{t+1} or to j_{t+1} yields the probability $2 \times \frac{255}{256} \times \frac{1}{256} = 0.0079$. This is at the same time the probability that two candidates will be returned by `fault(t)` procedure.

After repeating the previously described fault analysis process for consecutive times $t-1$ and t, the relations

$$j_{t+1} - j_t = S_t[i_{t+1}] \tag{10}$$

$$S_t[i_{t+1}] + S_t[j_{t+1}] - S_t[i_{t+1}] = S_t[j_{t+1}] \tag{11}$$

which hold for all $t \geq 1$, can be used to obtain

$$S_t[i_{t+1}] \text{ and } S_t[j_{t+1}] \tag{12}$$

Eventual multiple candidates for $S_t[i_{t+1}] + S_t[j_{t+1}]$ will imply the existence of two candidates for $S_t[j_{t+1}]$. Since (12) will be calculated for $t = 0, 1, \ldots, 256$ in the procedure below, it is relevant to compute the average number of t values for which procedure `fault(t)` will produce two candidates. Let X_t denote a random variable which equals 0 if `fault(t)` returns one candidate, and equals 1 otherwise. Then, according to the calculated probability of occurrence of cases 3 and 4, $P[X_t = 0] = 0.9921$ and $P[X_t = 1] = 0.0079$. The expected number of times `fault(t)`, $t = 0..257$, will return two candidates is given by $E(X_0 + X_1 + \ldots X_{256}) = E(X_0) + E(X_1) + \ldots E(X_{256}) = 257 \times 0.0079 = 2.0303$. Thus, it can be concluded that the number of candidates for

$$S_t[i_{t+1}] \text{ and } S_t[j_{t+1}] \text{ for } t = 0, 1, \ldots 256 \tag{13}$$

will be negligible. Since each candidate for (13) will be transformed to a candidate for the S table at a fixed time, wrong candidates can be easily discarded by comparing which solution produces a keystream that is identical to the original one.

What is left is to *rewind* from the values in (13) which are in different times t, to S_0 values. This procedure, which we call `recover-rewind`, is given by the following algorithm:

1. Initialize *Sarray* as an array of 256 integers and assign -1 to each position
2. Execute procedure `fault(0)` to find values j_1 and $S_0[i_1] + S_0[j_1]$. Let $Sarray[1] = j_1$ and $Sarray[j_1] = S_0[i_1] + S_0[j_1] - j_1$, and mark $Sarray[1]$ and $Sarray[j_1]$ as *used*
3. For $t = 1$ to 256
    4. Execute procedure `fault(t)` to find $j_{t+1}$and $S_t[i_{t+1}] + S_t[j_{t+1}]$. Derive $S_t[i_{t+1}]$ and $S_t[j_{t+1}]$ by applying relations (10) and (11)
    5. If $Sarray[t + 1]$ is not marked as *used*, assign it $S_t[i_{t+1}]$ and mark it as *used*
    6. If $Sarray[j_{t+1}]$ is not marked as *used*, assign it $S_t[j_{t+1}]$ and mark it as *used*
    7. Mark $Sarray[S[i_{t+1}] + S[j_{t+1}]]$ as *used* if it is not already marked
    8. Increase t by 1
9. Return *Sarray*

After procedure `recover-rewind` terminates, *Sarray* will be equal to S_0 at relevant indices and -1 at irrelevant indices. This can be explained as follows:

We note that if a certain value in *Sarray* is marked as *used* either in line 2 or during one of the loop passes, it will not be changed again during the execution of the procedure. The *Sarray* updates are performed in lines 2, 5, 6 and 7, not counting the initialization in line 1. The update in line 2 is justified because $j_0 = 0$. Marking an updated *Sarray* entry as *used* prevents the algorithm from further modifying this location. The updates in lines 5, 6 and 7 are done at indices $i_{t_0+1} = t_0 + 1$, j_{t_0+1} and $S_{t_0}[i_{t_0+1}] + S_{t_0}[j_{t_0+1}]$. To justify the last of these updates, in step 7, suppose the corresponding value has not already been marked as *used*. This means that $S_{t_0}[S_{t_0}[i_{t_0+1}] + S_{t_0}[j_{t_0+1}]]$ has not been touched by GGHN in steps before t_0, and now it is to be overwritten. In other words $S_{t_0}[i_{t_0+1}] + S_{t_0}[j_{t_0+1}]$ is an irrelevant index. It will remain -1 since it will not be changed in loop passes for $t \geq t_0$. Next, suppose that in loop pass t_0, $S_{t_0}[i_{t_0+1}]$, i.e., $S_{t_0}[t_0 + 1]$, is found. If it has not been marked as *used* in previous loop passes, it has not been touched and therefore was not overwritten by GGHN at any time before t_0. In other words, $S_{t_0}[t_0 + 1] = S_0[t_0 + 1]$, i.e., the element $S_0[t_0 + 1]$ is determined. However, if the element $S_{t_0}[i_{t_0+1}] = S_{t_0}[t_0 + 1]$ is found, and if it is already marked as *used*, this means that the element has either been already found or overwritten. In the first case, the correct value $S_0[t_0 + 1]$ has already been found. In the second case, the value $S_0[t_0 + 1]$ is irrelevant and thus does not need to be found in this phase of the algorithm. The same reasoning applies if at loop pass t_0, value $S_{t_0}[j_{t_0+1}]$ is found. All relevant values are guaranteed to be found, because at each fault analysis step for a given t, index $t + 1$ is either found or marked as *used*, and 256 such steps are executed.

After executing the `recover-rewind` procedure, what remains is to find k_0 and the values at irrelevant indices of S_0. It is assumed that all the S values found by fault analysis during procedure `recover-rewind` have been stored. No new faults will be used below.

To find k_0, it is sufficient to find k_t, for some $0 < t \leq 256$, since from this k_0 can be easily obtained as

$$k_0 = k_t - \sum_{d=0}^{t-1} S_d[j_{d+1}] \tag{14}$$

and the $S_d[j_{d+1}]$, $0 \leq d \leq 255$, are known from `recover-rewind` procedure.

Let $m_0 = S_0[i_1] + S_0[j_1]$ and assume $m_0 \geq 2$. After the first GGHN PRGA step, $S_1[m_0] = k_1 + S_0[j_1]$. Since $S_0[j_1]$ is known from the `recover-rewind` procedure, knowing $S_1[m_0]$ would reveal k_1. During the `recover-rewind` procedure, the value $S_{m_0-1}[m_0]$ is recovered. However, this value might not be equal to $S_1[m_0]$. This will happen if at one or more of the times $t = 1, \ldots, m_0 - 2$, the S value at index m_0 was updated again. Let t_0 denote the last such time. Then,

$$S_{m_0-1}[m_0] = k_{t_0+1} + S_{t_0}[j_{t_0+1}] \tag{15}$$

The values $S_{m_0-1}[m_0]$ and $S_{t_0}[j_{t_0+1}]$ are known from `recover-rewind`, and this reveals k_{t_0+1}. Finally, k_0 can be recovered by (14). In cases $m_0 = 0$ and $m_0 = 1$, let t_0 be the maximum time when the S value at position m_0 was updated, for the range of times $t = 1, \ldots, 254$ and $t = 1, \ldots, 255$, respectively. Relation (15) then becomes $S_{255}[0] = k_{t_0+1} + S_{t_0}[j_{t_0+1}]$ and $S_0[1] = k_{t_0+1} + S_{t_0}[j_{t_0+1}]$, respectively. In both cases, k_{t_0+1} can be recovered, and by means of (14), k_0 is determined. The latter case explains why the loop pass for $t = 256$ was required.

To find S values at irrelevant indices, the following procedure can be used. At each time $t_0 \geq 0$ when $S_{t_0}[i_{t_0+1}] + S_{t_0}[j_{t_0+1}]$ is equal to m, where m is an irrelevant index and t_0 is the first time this index was visited, $S_{t_0}[m]$ can be recovered as $S_{t_0}[m] = z_{t_0} - k_{t_0+1}$, since k_{t_0+1} is known by the previously explained procedure. Since t_0 is the first time this index is visited, $S_{t_0}[m] = S_0[m]$ and the required value is recovered. If not all of the S values at irrelevant indices are recovered by repeating this procedure until step 256, more k_t values, for $t > 256$, should be derived until all of the S values at irrelevant indices are found. Later k_t values are found by running the PRGA procedure with all words reduced to 8-bit size for an arbitrary number of steps, starting with an internal state in which irrelevant values are missing. This is possible due to Observation 1 and the definition of irrelevant states.

7 Complete attack and its complexity

The attack components described above can be glued together in order to form the whole attack procedure as follows:

1. The attacker gains the means to induce memory faults for the cipher and to reinitialize the cipher whenever needed
2. The attacker reconstructs values for all relevant indices in S_0 by executing the `recover-rewind` procedure
3. The attacker recovers k_0 and the S values at irrelevant indices, according to the procedure described in the previous section
4. The attacker then applies the procedure given in Section 5 and recovers the whole internal state at time 0, which breaks the cipher

In general, the attack complexity can be measured by the number of faults needed, the keystream data required and the amount of computation and storage needed to perform the attack. During the `recover-rewind` algorithm in step 2 of the attack, the `fault(t)` procedure is called 257 times, and each time 255 faults are induced. Thus the number of induced faults during the attack is 257×255. Each call to `fault(t)` utilizes two faulted keystream words. Finally, to apply step 4 of the attack, given in Section 5, 257 non-faulted keystream words are required. The resulting system of linear equations can be solved efficiently on a standard personal computer.

According to an experiment in which we simulated the above attack process 1000 times, and recorded the number of faults used to completely recover the internal state, we found that on average the internal state was recovered after $2^{15.3}$ faults, with a standard deviation of $2^{10.3}$.

8 Conclusion and future work

We have shown that in GGHN, the design approach of extending the word size from 8 to 32 bits while maintaining a table size of 256 entries was not done properly. Specifically, not all the bits in the internal state table affect the future behavior of the cipher in an equal way. Furthermore, knowing 2064 specific bits of the internal state and a small portion of keystream, it is possible to deduce the remaining bits of the internal state in negligible computation time. If these 2064 bits can be obtained by some other means, for example, by means of fault analysis, as we demonstrate in this paper, it is possible to recover the entire internal state and thereby break the cipher.

For future consideration, it would be interesting to assess whether uneven influences by internal state bits on the behavior of the cipher can be used for other forms of cryptanalysis.

Acknowledgements The authors would like to thank Dr. Liam Keliher for proof-reading the initial submission of this paper. The authors would also like to thank the anonymous reviewers for their comments that helped improve the presentation of the paper.

Appendix A: Reducing GGHN internal state: a toy example

In this appendix, we provide an example of applying the method described in Section 5. Consider GGHN(2,8), a toy version of GGHN, in which the S table has $2^2 = 4$ elements and the internal word size is 8 bits. By x^L we shall now denote the 2 least significant bits of x, and the remaining 6 most significant bits are denoted x^R. Thus x^L can take values from $0, \ldots, 3$, and x^R from $0, \ldots, 63$.

Let the indices i, j and the value k of the internal state of GGHN(2,8) at some time be as shown in Table 1.

Table 1 Internal state variables: i, j and k

i	j	k^R	k^L
1	3	10	2

Table 2 The S table

$S^R[0]$	$S^L[0]$	$S^R[1]$	$S^L[1]$	$S^R[2]$	$S^L[2]$	$S^R[3]$	$S^L[3]$
19	0	62	1	51	3	34	3

and let the S table be as shown in Table 2.

Suppose that the attacker was able to recover the 2 least significant bits of every internal state word, given by the first column in Table 3. Then the attacker can continue iterating the cipher reduced to 2 bits and obtain the remaining columns of the table.

The attacker observes the next 5 keystream words, given in Table 4.

Next, equations of the form in (4) can be constructed. For $t = 0$, the equation is of the form

$$S_0^R[a_0] + k_1^R + \sigma_0 = z_0^R$$

From Table 3, we have $a_0 = S_0^L[i_1] + S_0^L[j_1] = S_0^L[2] + S_0^L[2] = 1 + 1 = 2$, and from Table 4, we know $z_0^R = 50$. Since $S_0^L[2] + k_1^L > 3$, $\sigma_0 = 1$. Thus the equation is:

$$S_0^R[2] + k_1^R + 1 = 50$$

Similarly, other equations of the form in (4) are:

$$S_1^R[0] + k_2^R = 15$$
$$S_2^R[1] + k_3^R = 44$$
$$S_3^R[2] + k_4^R = 46$$
$$S_4^R[3] + k_5^R = 44$$

Now, the unknowns in the system should be expressed in terms of $S_0^R[0], \ldots, S_0^R[3], k_0^R$. For this, we use equations (5), (6) and (7):

$$S_1^R[2] = k_1^R + S_0^R[2] + 1,\ S_1^R[0] = S_0^R[0], S_1^R[1] = S_0^R[1], S_1^R[3] = S_0^R[3]$$
$$S_2^R[0] = k_2^R + S_1^R[3] + 1,\ S_2^R[1] = S_1^R[1], S_2^R[2] = S_1^R[2], S_2^R[3] = S_1^R[3]$$
$$S_3^R[1] = k_3^R + S_2^R[0],\quad S_3^R[0] = S_2^R[0], S_3^R[2] = S_2^R[2], S_3^R[3] = S_2^R[3]$$
$$S_4^R[2] = k_4^R + S_3^R[1] + 1,\ S_4^R[0] = S_3^R[0], S_4^R[1] = S_3^R[1], S_4^R[3] = S_3^R[3]$$

Table 3 Information known by the attacker

t	0	1	2	3	4
i_t	1	2	3	0	1
j_t	3	2	1	2	1
k_t^L	2	1	2	2	1
$S_t^L[0]$	0	0	1	1	1
$S_t^L[1]$	1	1	1	3	3
$S_t^L[2]$	3	0	0	0	0
$S_t^L[3]$	3	3	3	3	3

Table 4 Keystream words observed by the attacker

z_t^R	z_t^L	z_{t+1}^R	z_{t+1}^L	z_{t+2}^R	z_{t+2}^L	z_{t+3}^R	z_{t+3}^L	z_{t+4}^R	z_{t+4}^L
50	0	15	2	44	3	46	1	44	3

and

$$k_1^R = k_0^R + S_0^R[2] + 1$$
$$k_2^R = k_1^R + S_1^R[1]$$
$$k_3^R = k_2^R + S_2^R[2]$$
$$k_4^R = k_3^R + S_3^R[1] + 1$$
$$k_5^R = k_4^R + S_4^R[1] + 1$$

Substituting appropriate values yields the following system:

$$k_0^R + 2S_0^R[2] = 48$$
$$k_0^R + S_0^R[0] + S_0^R[1] + S_0^R[2] = 14$$
$$2k_0^R + 2S_0^R[1] + 3S_0^R[2] = 41$$
$$6k_0^R + 3S_0^R[1] + 9S_0^R[2] + S_0^R[3] = 35$$
$$8k_0^R + 5S_0^R[1] + 11S_0^R[2] + 3S_0^R[3] = 29$$

It can easily be verified that the set of values $S^R[0], \ldots, S^R[3]$ in Table 2 is the solution for this system of linear equations.

Appendix B: On the probability of other outcomes when $S_t[j_{t+1}]$ is faulted

Consider the output step of the cipher at step t:

$$z_t = k_{t+1} + S_t[d], \text{ there } d = S_t[i_{t+1}] + S_t[j_{t+1}]$$

Let the assumptions from case 1(c) hold, i.e., $i_{t+1} \neq j_{t+1} \neq S_t[i_{t+1}] + S_t[j_{t+1}]$ and $m = j_{t+1}$. As will be shown, with small probability, contrary to case 1(c) above, $z'_t = z_t$ and $z'_{t+1} = z_{t+1}$ can hold. Namely, the faulted $S_t[j_{t+1}]$ will cause a difference in k_{t+1}, i.e., $k'_{t+1} \neq k_{t+1}$. However, d will also be different, i.e., another S element will be used. Thus it is possible that $z'_t = k'_{t+1} + S_t[d'] = z_t$, if such $S_t[d']$ exists, i.e., it is possible that the difference in k is annulled. The probability that this will happen is $p_A = 1/2^{32}$. As for z'_{t+1}, even though the assumptions of case 1(c) are fulfilled, this value can also be equal to z_{t+1}, with small probability, again contrary to the reasoning in case 1(c). Namely, this will occur if $S_{t+1}[j_{t+2}]$ is such that $k'_{t+2} = k'_{t+1} + S_{t+1}[j_{t+2}] = k_{t+2}$. Since this can only happen if j_{t+2} is equal to indices that point to corrupted S values j_{t+1} and $S_t[i_{t+1}] + S_t[j_{t+1}]$, the probability is $p_B = (2/256) \times 1/2^{32} \approx 2^{-39}$. Similar probabilities are obtained if case 3 or case 4 is assumed. In case 2 and case 5, this problem cannot arise.

Since in one execution of the `fault(t)` procedure, m will equal j_{t+1} at most once, and `fault(t)` is called 257 times, the probability that any of these errors will occur is $1 - ((1 - p_A) \times (1 - p_B))^{257} \approx 2^{-23.9}$, which does not present a problem in practical applications of the attack.

References

1. Anderson, R., Kuhn, M.: Low cost attacks on tamper resistant devices. In: Security Protocols, 5th International Workshop. LNCS, vol. 1361, pp. 125–136. Springer, New York (1997)
2. Biham, E., Shamir, A.: Differential fault analysis of secret key cryptosystems. In: CRYPTO'97. LNCS, vol. 1294, pp. 513–525. Springer, New York (1997)
3. Boneh, D., Demillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Eurocrypt'97. LNCS, vol. 1233, pp. 37–51. Springer, New York (1997)
4. Biham, E., Granboulan, L., Nguyen, P.Q.: Impossible fault analysis of RC4 and differential fault analysis of RC4. In: Fast Software Encryption 2005. LNCS, vol. 3557, pp. 359–367. Springer, New York (2005)
5. Dusart, P., Letourneux, G., Vivolo, O.: Differential fault analysis on AES. In: Applied Cryptography and Network Security 2003. LNCS, vol. 2846, pp. 293–306. Springer, New York (2003)
6. Fluhrer, S.R., McGrew, D.A.: Statistical analysis of the alleged RC4 keystream generator. In: Fast Software Encryption 2000. LNCS, vol. 1978, pp. 66–71. Springer, New York (2000)
7. Fluhrer, S.R., Mantin, I., Shamir, A.: Weaknesses in the key scheduling algorithm of RC4. In: Selected Areas in Cryptography 2001. LNCS, vol. 2259, pp. 1–24. Springer, New York (2001)
8. Golić, J.D.: Linear statistical weakness of alleged RC4 keystream generator. In: EUROCRYPT 1997. LNCS, vol. 1233, pp. 226–238. Springer, New York (1997)
9. Gong, G., Gupta, K.C., Hell, M., Nawaz, Y.: Towards a general RC4-like keystream generator. In: Proc. of Information Security and Cryptology 2005. LNCS, vol. 3822, pp. 162–174 Springer, New York (2005)
10. Hawkes, P., Rose, G.G.: On the applicability of distinguishing attacks against stream ciphers. In: Proc. of the Third NESSIE Workshop (2002)
11. Hoch, J., Shamir, A.: Fault analysis of stream ciphers. In: CHES 2004. LNCS, vol. 3156, pp. 240–253. Springer, New York (2004)
12. Knudsen, L.R., Meier, W., Prenel, B., Rijmen, V., Verdoolaege, S.: Analysis methods for (Alleged) RC4. In: ASIACRYPT'98. LNCS, vol. 1514, pp. 327–341. Springer, New York (1998)
13. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: CRYPTO' 99. LNCS, vol. 1666, pp. 388–397. Springer, New York (1999)
14. Kocher, P.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: CRYPTO '96. LNCS, vol. 1109, pp. 104–113. Springer, New York (1996)
15. MacLaren, M.D., Marsaglia, G.: Uniform random number generation. J. ACM **15**, 83–89 (1965)
16. Mantin, I., Shamir, A.: A practical attack on broadcast RC4. In: Fast Software Encryption 2001. LNCS, vol. 2355, pp. 152–164. Springer, New York (2001)
17. Mantin, I.: Predicting and distinguishing attacks on RC4 keystream generator. In: EUROCRYPT '2005. LNCS, vol. 3494, pp. 491–506. Springer, New York (2005)
18. Maximov, A.: Two linear distinguishing attacks on VMPC and RC4A and weakness of RC4 family of stream ciphers. In: Fast Software Encryption 2005. LNCS, vol. 3357, pp. 342–358. Springer, New York (2005)
19. Maximov, A., Khovratovich, D.: New state recovery attack on RC4. In: CRYPTO'08. LNCS, vol. 5365, pp. 40–52. Springer, New York (2008)
20. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptographic Research. CRC, Boca Raton (1996)
21. Mironov, I.: Not (So) random shuffle RC4. In: CRYPTO' 02. LNCS, vol. 2442, pp. 304–319. Springer, New York (2002)
22. Nawaz, Y., Gupta, K.C., Gong, G.: A 32-bit RC4-like keystream generator. Technical Report CACR 2005-21, Center for Applied Cryptographic Research, University of Waterloo. http://www.cacr.math.uwaterloo.ca/tech_reports.html (2005) (Also available at Cryptology ePrint Archive, 2005-175, http://eprint.iacr.org/2005/175)

23. Paul, S., Preneel, B.: A new weakness in RC4 keystream generator and an approach to improve the security of the cipher. In: Fast Software Encryption 2004. LNCS, vol. 3017, pp. 245–259. Springer, New York (2004)
24. Paul, S., Preenel, B.: On the (In)security of stream ciphers based on arrays and modular addition. In: ASIACRYPT 2006. LNCS, vol. 4284, pp. 69–83. Springer, New York (2006)
25. Skorobogatov, S.P., Anderson, R.J.: Optical fault induction attacks. In: CHES 2003. LNCS, vol. 2523, pp. 2–12. Springer, New York (2003)
26. Tsunoo, Y., Saito, T., Kubo, H., Suzaki, T.: A distinguishing attack on a fast software-implemented RC4-like stream cipher. IEEE Trans. Inf. Theory **53**(9) (2007)
27. Vaudenay, S., Vuagnoux, M.: Passive-only key recovery attacks on RC4. In: Selected Areas in Cryptography 2007. LNCS, vol. 4876, pp. 344–359. Springer, New York (2007)
28. Zoltak, B.: VMPC one-way function and stream cipher. In: Proc. of Fast Software Encryption, FSE 2004. LNCS, vol. 3017, pp. 210–225. Springer, New York (2004)