A Chosen IV Attack Against *Turing*

Antoine Joux and Frédéric Muller

DCSSI Crypto Lab, 18 rue du Docteur Zamenhof
F-92131 Issy-les-Moulineaux Cedex, France
{Antoine.Joux,Frederic.Muller}@m4x.org

Abstract. In this paper, we show that the key scheduling algorithm of the recently proposed stream cipher Turing suffers from important flaws. These weaknesses allow an attacker that chooses the initialization vector (IV) to recover some partial information about the secret key. In particular, when using Turing with a 256-bit secret key and a 128-bit IV, we present an attack that requires the ability to choose 2^{37} IV and then recovers the key with complexity 2^{72}, requiring 2^{36} bytes of memory.

1 Introduction

A stream cipher is a secret key cryptosystem that takes a short random string and transforms it into a long pseudo-random sequence. Usually this sequence is XORed bitwise with the plaintext in order to produce the ciphertext. This purpose can also be reached by iterating a block cipher, however it is widely believed that a dedicated design may offer a better encryption speed. In general, a stream cipher produces a pseudo random sequence $PRNG(K, IV)$ from a secret key K and an initialization vector IV. Then, the ciphertext C is computed from the plaintext P by:

$$C = PRNG(K, IV) \oplus P$$

The main idea behind the use of initialization vectors is to generate different pseudo-random sequences without necessarily changing the secret key, since it is totally insecure to use twice the same sequence. Similar problems may arise when using block ciphers, since they are purely deterministic objects. This is the reason why several modes of operations like CBC or OFB require an initialization vector. Actually, block ciphers are not well suited for that purpose and a more suitable primitive called "Tweakable Block Cipher" was recently proposed [12]. It includes an additional input called the "tweak" which basically plays a role similar to an initialization vector. In both situations, it is extremely difficult to control the mechanism producing the randomization source, since it will usually be provided by a non-cryptographic layer. For instance, it may be implemented using counters or using an unspecified random source. Given this lack of knowledge, any cipher should generally resist attacks where an attacker is able to choose the initialization vector.

Recent developments in the cryptanalysis of stream ciphers have shown the importance of these considerations. In particular, it appears the key scheduling

M. Matsui and R. Zuccherato (Eds.): SAC 2003, LNCS 3006, pp. 194–207, 2004.

algorithm should be analyzed carefully in order to secure real-life systems. The chosen IV attack against RC4 proposed in [7] is a good illustration of this point. Although RC4 still appears as a robust cipher, the key scheduling algorithm leaks partial information concerning the secret key, when observing the first bytes of keystream. Besides, initialization vectors were implemented in the Wireless Equivalent Privacy (WEP) - a standard for wireless network encryption - in a way that allows key reconstruction from chosen IV's. Further analysis has shown that this chosen IV attack was practical (see [18]) and that known IV's were in fact enough to recover the secret key of most wireless LAN's given a reasonable amount of network traffic.

In the context of stream ciphers, most applications require frequent changes of initialization vector. The first reason is that long sequences of keystream imply permanent synchronization, which can become painful especially with high speed connections (for instance, mobile phone communications). The second reason is to handle multiple simultaneous sessions encrypted with the same secret key (for instance with wireless LAN's: WEP or Bluetooth). These practical problems are usually solved by encrypting only short sequences (2745 bits in the case of Bluetooth) and changing the IV value between each sequence. As a consequence, attacks based on known or chosen IV's are made more realistic, while classical attacks are still possible over multiple sequences (see the recent attacks against Bluetooth [8]).

All these elements motivate an analysis of the key scheduling algorithm of new stream ciphers. Here, we focus on Turing, a stream cipher recently proposed by Rose and Hawkes [16]. According to them, Turing is designed in order to achieve a good efficiency in software applications and they claim performances 5 times faster than AES on optimized versions. The keystream generation is based on classical components:

- A large Linear Feedback Shift Register (LFSR) with elements in GF(2^{32}). Such registers offer good statistical properties and a large period, while making correlation and fast-correlation attacks difficult. Good examples of similar designs are SNOW [4], S32 [14] and SOBER-t32 [10].
- A Nonlinear Filter that takes 5 words in the LFSR and transforms them in a key-dependent, highly non-linear manner. The resulting keystream is 160 bits long and is masked using 5 words of the LFSR. This technique is usually called “Linear Masking” and has been extensively studied in [3].
- A Key Scheduling Algorithm using a key of length up to 256 bits and an initialization vector of varying length. This routine fills out a keyed S-box used in the Nonlinear Filter and also sets up the initial LFSR state.

In Section 2 we will briefly describe Turing. One should refer to [16] to obtain a more precise description. In Section 3 we show some weaknesses in the key scheduling algorithm of Turing, using chosen initialization vectors. Furthermore, we show how these weaknesses allow a key recovery attack. In Section 4, we show some extensions of our observations to other kind of attacks.

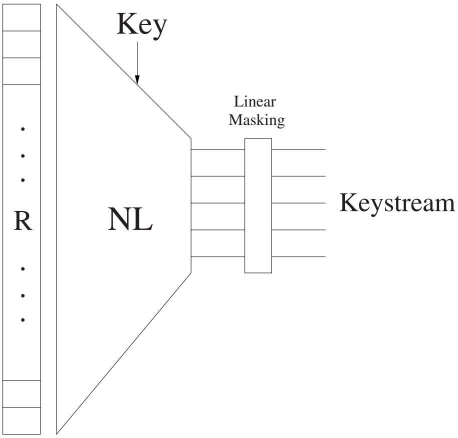

Fig. 1. General Structure of Turing

2 Description of *Turing*

The Turing keystream generation function is based on a classical structure : a linear feedback register R combined with a nonlinear filter NL . The particularity of this cipher lies in the key dependence of NL and on the linear masking. An overview of this structure is given in Figure 1.

Turing combines additions in GF(2^{32}) with additions modulo 2^{32}. To avoid any confusion, $\oplus$ will denote additions in GF(2^{32}), i.e. bitwise XOR, while $+$ will denote addition modulo 2^{32}.

Our description of the cipher mostly focuses on particular points that are relevant to our attacks. We dedicate a substantial space to describe several aspects of Turing that are only briefly described in the initial paper. They can be found by directly looking at the reference implementation code [2].

2.1 The LFSR

Turing uses a Linear Feedback Shift Register R operating on elements in GF(2^{32}). This choice allows good software performances on 32 bits processors. R has 17 cells, so the total number of internal state bits is 544. A register offering the same security with elements in GF(2) would require a very dense feedback polynomial, which would increase the cost of each advance. In the case of Turing, the following feedback polynomial was chosen :

$$x^{17} \oplus x^{15} \oplus x^4 \oplus \alpha$$

where α is a fixed element in GF(2^{32}). Thus, denoting by R_i the content of cell number i, the update function can be expressed as

$$z = R_{15} \oplus R_4 \oplus \alpha R_0$$

This new value z is introduced at position 16 and all cells are shifted by one position (R_0 is discarded).

Such registers are expected to offer good resistance against a wide class of attacks against stream ciphers based on LFSR, the correlation attacks [17] and fast correlation attacks [13]. The importance of choosing adequate feedback polynomials in this type of registers was recently shown by the new attacks against SNOW (see [3], [5] and [11]). However, the designers of Turing have apparently been very careful in the choice of polynomial to prevent similar attacks. Besides, the Nonlinear Filter being key-dependent, it seems very unlikely that a bias could be found independently of the secret key.

2.2 The Nonlinear Filter

The Nonlinear Filter (NL) of Turing is very similar to the construction of a block cipher round function. It combines two applications of a linear transformation on GF(2^{32}) and a layer of key-dependent 32-to-32 S-box applied to each word in-between the two linear layers. The input of NL consists in 5 cells of the LFSR : R_0, R_1, R_6, R_{13} and R_{16}.

The linear operation is the so-called Pseudo-Hadamard transform (PHT) (see [16] for more details). The keyed S-box is described more precisely in Section 2.4. Besides, a circular rotation is also applied to 3 of the 5 words before computing S in order to differentiate the trails followed by each input of NL.

2.3 Linear Masking

Linear Masking consists in adding to each output word of the Nonlinear Filter some word of the LFSR to hide any kind of undesirable statistical properties. Variants of this idea of a masking scheme have been used in SNOW [5] and SCREAM [9].

Here the linear masking is very simple. For each application of the Nonlinear filter, the LFSR is clocked twice :

- After the first clocking, the content of cells number 0, 1, 6, 13 and 16 is used as input of NL.
- After the second clocking, the same cells are used to mask the outputs of NL using a simple addition over GF(2^{32}).

This requires the LFSR to be clocked twice in order to produce $5 \times 32 = 160$ bits of keystream.

2.4 The Key Scheduling Algorithm

The Key Scheduling Algorithm of Turing receives a secret key K and an initialization vector IV. The number of words of K and IV are respectively denoted by n_K and n_I. Both should be multiples of 32 bits. Besides, n_K should not exceed 8 and $n_I + n_K$ should not exceed 12. Words of K are denoted by $\{K_i,$

$i = 0, \ldots, n_K - 1\}$. Each word K_i is further divided into four bytes which we denote by $\{K_{i,j},\ j = 0, \ldots, 3\}$. Similar notations are used for IV.

This initialization vector is only used to build the initial state of the LFSR while the secret key is used both in the keyed S-box S and to build the initial state of the LFSR. The Key Scheduling Algorithm of Turing basically consists in three steps.

The Key Transform To prevent related key attacks, the actual secret key K is turned into a secret value of the same length using a reversible transformation G applied to each word of K. The result is used to initialize the LFSR and to build the keyed S-box S. So the knowledge of this result is actually equivalent to the knowledge of K. That is why in the following sections, notation K will be used to represent this resulting value instead of the true secret key.

The Keyed S-box The keyed S-box S - also used in the nonlinear filter - is used to compute the initial state of the LFSR. It operates on a word w represented as four bytes $w = (w_0, w_1, w_2, w_3)$.

For each byte w_i, a 32-bit value $s_i(w_i)$ is computed using a 8-to-32 function depending on byte number i of each word K_j, denoted as $K_{j,i}$, for $0 \leq j < n_K$. For instance, $s_0(w_0)$ is computed using only the least significant byte of w and the least significant byte of each word of K. The four values $s_0(w_0), \ldots, s_3(w_3)$ are finally added bitwise :

$$S(w) = s_0(w_0) \oplus s_1(w_1) \oplus s_2(w_2) \oplus s_3(w_3)$$

Each s_i uses a chained construction similar to the CBC mode of operation on block ciphers and a 32 bits long accumulator. The general structure of s_i is summarized in Fig. 2 where key bytes are represented by $k0, \ldots, k7$. Two fixed S-boxes, Sb and Qb, are used during this computation. More details on these constants are given in [16] and [2]. Depending on which s_i is considered, the output byte of the CBC chain is appended at different positions in the final word and different key bytes are used.

The Initial State of the LFSR The LFSR is initially filled out as follows :

- The words of IV are processed using the transformation G previously applied to the key words and then copied at positions 0 to $n_I - 1$ in the LFSR. The transformation is fixed, so the value of these cells in the LFSR is initially known.
- The words of K are appended at positions n_I to $n_I + n_K - 1$.
- The binary value $\texttt{0x010203}n_K n_I$ is appended at position $n_I + n_K$.
- The remaining cells R_i for $n_I + n_K < i < 17$ are computed by the formula

$$R_i = S(R_{i-1} + R_{i-n_K-n_I-1})$$

Finally, a 17-word Pseudo-Hadamard transform is applied :

$$T = \sum_{i=0}^{i=16} R_i \qquad R_{16} := T \qquad R_i := R_i + T$$

for $i < 16$. Once this is done, keystream generation starts.

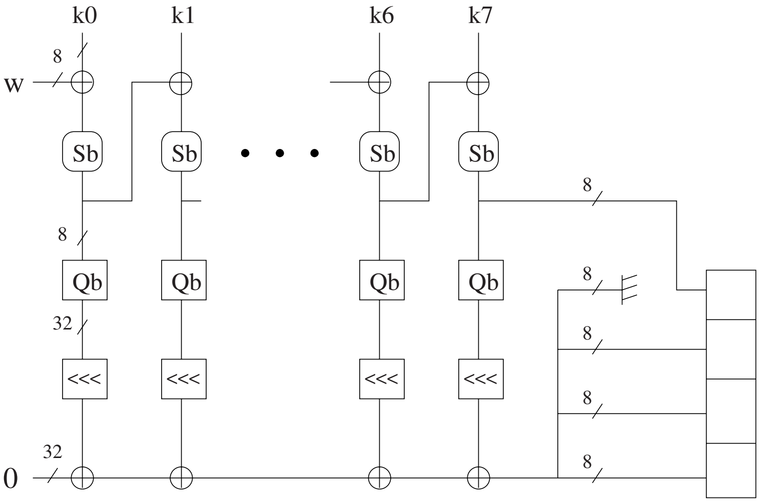

Fig. 2. Structure of s_i

3 A Chosen IV Attack

We observed that the influence of the initialization vector on the internal LFSR state is very limited. In spite of the Pseudo-Hadamard transform, it is possible that different IV's yield very similar initial states. In this case, the first words of keystream generated may even be equal.

To illustrate this, we will focus on the case where the secret key is 256 bits long (that is $n_K = 8$) and the initialization vector is 128 bits long (that is $n_I = 4$). The initial content of the LFSR - before the application of the PHT - is represented in Table 1.

In the following sections, we will fix the value of the secret key, IV_1 and IV_2. Then, we will study what is the influence on the initial state of successively randomizing IV_3 and IV_0.

Table 1. Initial State of the LFSR

$R_0 = G(IV_0)$	$R_9 = K_5$
$R_1 = G(IV_1)$	$R_{10} = K_6$
$R_2 = G(IV_2)$	$R_{11} = K_7$
$R_3 = G(IV_3)$	$R_{12} = \texttt{0x01020384}$
$R_4 = K_0$	$R_{13} = S(R_{12} + R_0)$
$R_5 = K_1$	$R_{14} = S(R_{13} + R_1)$
$R_6 = K_2$	$R_{15} = S(R_{14} + R_2)$
$R_7 = K_3$	$R_{16} = S(R_{15} + R_3)$
$R_8 = K_4$	

3.1 Randomizing IV_3

In the previous table, one sees that only R_3 and R_{16} actually depend on IV_3. Furthermore when the secret key and the first three words of the IV are fixed, it may happen for two distinct values of IV_3, say x and x', to observe the same value of $R_{16} + R_3$, before the PHT. Thus :

$$S(R_{15} + G(x)) + G(x) = S(R_{15} + G(x')) + G(x') \tag{1}$$

This is a birthday paradox situation. Thus, such an event should happen if about 2^{16} different values of IV_3 are tested.

In fact, this can be immediately detected. As a matter of fact, if (1) holds, the value of T computed by the PHT will be equal in both cases, and thus, only cell number 3 of the LFSR will differ after the PHT. Then, the first 5 words of keystream are computed as

$$(C_1, \ldots, C_5) = NL(X_1, X_2, X_7, X_{14}, X_{17}) \oplus (X_2, X_3, X_8, X_{15}, X_{18})$$

where C_i denotes the i-th keystream word and X_i the cell number i of the LFSR, after the PHT. There are only 17 cells in the LFSR, however we denote by X_{17} and X_{18} the following terms in the LFSR sequence :

$$X_{17} = X_{15} \oplus X_4 \oplus \alpha X_0$$
$$X_{18} = X_{16} \oplus X_5 \oplus \alpha X_1$$

Since no input of NL depends on IV_3, the value of C_1,C_2,C_3 and C_5 will be equal with both IV's. It is very unlikely that such keystream collisions on 128 bits will occur if (1) is not verified. Therefore, relation (1) can be efficiently detected by looking at the first words of keystream.

3.2 Randomizing IV_0

The keyed S-box S is not a permutation because of its particular structure based on 4 smaller functions. In fact, it should rather be seen as a pseudo-random function when the key is unknown. Therefore, we can expect to find collisions after 2^{16} queries to this S-box for an unknown secret key. Some experiments made on randomly chosen secret keys have confirmed this property.

It follows that different values - say y_1 and y_2 - for IV_0 may exist such that

$$S(G(y_1) + \texttt{0x01020384}) = S(G(y_2) + \texttt{0x01020384}) \tag{2}$$

The essential property is that values of IV_0 verifying (2) will induce the same value of R_{15}. Thus, they will verify (1) with the same IV_3's. The influence of these IV words on the first keystream words can be summarized by :

Indeed, the difference in the initial state brought by IV_0 vanishes when computing R_{13} because of (2). Then, when applying the PHT, the same value of T is computed for both IV_0's, since x_1 and x_1' verify (1). Therefore, collisions on S can be detected by looking at the keystream produced with different chosen initialization vectors.

IV_0	IV_3	First Keystream Words				
y_1	x_1	C_1	C_2	C_3	C_4	C_5
y_1	x'_1	C_1	$C_2 \oplus \Delta$	C_3	C_4	C_5
y_2	x_1	C'_1	C'_2	C'_3	C'_4	C'_5
y_2	x'_1	C'_1	$C_2 \oplus \Delta'$	C'_3	C'_4	C'_5

3.3 Efficient Detection of Collisions

In this section, we assume the attacker has access to the key schedule routine with a fixed unknown secret key and can choose the initialization vector. We also assume the attacker can observe the first words of keystream produced after this key schedule. We will show how such an attacker can efficiently recover collisions on S with chosen input values.

```
– Fix the secret key $K$
– Fix $IV_1$ and $IV_2$
– While (no relation (2) found)
    • Choose a value of $IV_0$
    • While (no relation (1) found)
        ∗ Pick a random value of $IV_3$
        ∗ Do a key scheduling with current value of $IV$ words
        ∗ Look for collisions on the first words of keystream
    • Store $IV_0$ with its two corresponding $IV_3$'s
    • Do a key scheduling with current $IV_0$ and all previously stored pairs of
      $IV_3$
    • Look for collisions on the first words of keystream
– Output the two corresponding $IV_0$'s (they verify (2))
```

Collisions described in the previous sections are observed after about 2^{16} queries. Thus, the total number of key scheduling necessary to observe once relation (2) is roughly

$$\sum_{i=1}^{i=2^{16}} (2(i-1) + 2^{16}) \simeq 2^{32}$$

We have implemented this basic technique with several different secret keys and managed to detect collisions on S quite efficiently.

An improved technique consists in fixing several values (say 2^{16}) of IV_3 before any key scheduling request. Then test, say M, different values of IV_0 with all these fixed IV_3's. Then, it is possible to detect which pairs of IV_0 among the M values considered yield collisions with the same pairs of IV_3's. These pairs verify relation (2). This technique allows to recover $M^2\ 2^{-32}$ S-box collisions after $M\ 2^{16}$ key scheduling requests although the chances of false alarm slightly increase. This technique is less expensive when more than one collision is needed.

3.4 A Key Recovery Attack

It turns out that these collisions on the keyed S-box reveal direct information about the secret key. More precisely, remember that collisions on S are obtained with chosen inputs. Moreover, outputs of S depend only on part of the key bits, when the corresponding inputs agree on a certain number of bytes. Accordingly, let us consider only collisions where inputs disagree on 3 bytes at most (for instance input values having the same most significant byte). This means only 2^{24} of all possible IV_0's may be tested. By choosing appropriate values for IV_0, the collisions will yield relations of the form :

$$S(w) = S(w')$$

where

$$S(w) = s_0(w_0) \oplus s_1(w_1) \oplus s_2(w_2) \oplus s_3(w_3)$$
$$S(w') = s_0(w'_0) \oplus s_1(w'_1) \oplus s_2(w'_2) \oplus s_3(w'_3)$$

and, since $w_0 = w'_0$,

$$s_1(w_1) \oplus s_2(w_2) \oplus s_3(w_3) = s_1(w'_1) \oplus s_2(w'_2) \oplus s_3(w'_3) \qquad (3)$$

Thus, we get a relation involving only three of the four 8-to-32 key-dependent functions that constitute S. This brings some significant information about the 64 key bytes used in each of these functions. A straightforward approach would be to use relation (3) to decrease the complexity of an exhaustive search on the 256 bits of secret key. The number of triplets of outputs of s_1, s_2 and s_3 is :

$$(2^8)^3 = 2^{24}$$

Therefore the expected value for the number of existing relations is about

$$2^{24} \times 2^{24} \times 2^{-32} = 2^{16}$$

Hence, many relations similar to (3) should exist. They involve only 192 key bits, so a small number of them is sufficient to reduce the cost of key exhaustive search to 2^{192}. Furthermore, it may happen that collisions involving only 2 bytes exist. In that case, the cost of a brute-force attack may even be reduced to 2^{128}. However, this event occurs with small probability, so keys yielding these collisions may be considered as weak keys.

3.5 Solving the Underlying Linear System

However, we believe relations similar to (3) may be used in a more efficient manner to recover the secret key. Let us call $X_i = s_1(i)$, $Y_i = s_2(i)$ and $Z_i = s_3(i)$ for $0 \leq i \leq 255$. (3) is a linear relation between 6 of these $3 \times 256 = 768$ unknown values in GF(2^{32}), of the form :

$$X_i \oplus X'_i \oplus Y_j \oplus Y'_j \oplus Z_k \oplus Z'_k = 0 \qquad (4)$$

for some i, i', j, j', k, k' between 0 and 255.

More generally, when many similar relations are available - at least 768 hopefully independent relations - one can expect to solve the underlying linear system. Unfortunately, it turns out it never has full rank. Indeed, all coefficients in relation (4) are either 0 or 1. Therefore this system can also be viewed as a linear system in GF(2) and, from each solution in GF(2^{32}), it is possible to build 32 solutions in GF(2). This implies a decrepancy of at least 32 in the rank of the system since at least one solution exists in GF(2^{32}), corresponding to the real outputs of the keyed S-box.

Furthermore, since all X_i's appear twice in each relation, adding a fixed word C to these 256 unknowns provides an extra non trivial solution. Similar considerations also hold for Y_i and Z_i, which implies an additional discrepancy of 3. So the kernel of the system has dimension at least $32 + 3 = 35$. An open question is to know whether this dimension is actually always 35. Using different secret keys, we built the corresponding system by observing all collisions on S. Then, we computed its rank using the software MAGMA [1]. We always obtained a rank equal to 733, which corresponds to the expected value

$$768 - 3 - 32 = 768 - 35 = 733$$

Besides, it appears from our experiments that, using about $1000 \simeq 2^{10}$ relations is enough to always obtain a system of maximal rank 733. In particular, testing 2^{21} values for IV_0 should be sufficient to obtain enough collision and thus a system of maximal rank :

$$2^{21} \times 2^{21} \times 2^{-32} = 2^{10}$$

Thus $2^{21} \times 2^{16} = 2^{37}$ chosen initialization vectors are needed.

Building a basis of the kernel of the underlying linear application can be done very quickly. We used the software MAGMA and, given the size of the system, it takes only a few seconds to complete this task. However, since the system has not full rank, linear algebra does not provide immediately a unique solution, but we can easily obtain a basis of the kernel and span its 2^{35} elements.

Using additional constraints resulting from the structure of the underlying keyed S-box, an efficient key recovery attack directly follows. Assume these 2^{35} solutions are sorted and stored in a table. An exhaustive search on the 64 key bits involved in s_3 will provide a candidate for $(Z_0, \ldots, Z_{255})$. Wrong guesses can be filtered out using the table, since the vector of least significant bits of all Z_i's must appear somewhere in this table, as part of a GF(2) solution. A similar technique can be used to find key bytes involved in s_1 and s_2 and the final 64 key bits can be guessed separately.

Therefore we obtain the 256 secret key bits, using a table containing 2^{35} words of 768 bits, with 2^{37} chosen IV's and $256 \times 2^{64} = 2^{72}$ steps of computation. A basic improvement is not to consider all solutions of the previous system in GF(2). Indeed, the output byte corresponding to the CBC chain (see Figure 2) in any function s_i is a permutation of the inputs (since this operation is reversible). Therefore, instead of verifying the least significant bit, we can use an other bit position corresponding to the output of this permutation and store only balanced vectors of 256 bits. Thus the size of the table required is reduced to the number of solutions balanced on these 256 bits, that is about 2^{31}, since

$$\binom{256}{128} \times 2^{-256} \times 2^{35} \simeq 2^{31}$$

Each entry of the table is a 256 bits vector. Thus, this attack requires 2^{36} bytes of memory.

4 Extensions of this Attack

The attack we have proposed is a little restrictive since it works only with 256 bits long secret key and 128 bits long IV's. In this section, we demonstrate that although this attack does not work in every case, some residual weaknesses of the key scheduling algorithm cannot be prevented.

4.1 Other Key and IV Lengths

The attack proposed in the previous section relies on the lack of diffusion brought by the PHT. This weakness exist for any key and IV length. However, the possibility to observe collisions on the first keystream words vary greatly depending on the choice of these parameters. We have demonstrated our attack when $n_K = 8$ and $n_I = 4$. When key length is less than 8 words but key and IV still sum up to 12 words, the attack proposed in Section 3 still works by keeping the additional IV words constant. The resulting attack has time complexity

$$2^8 \times 2^{8n_K}$$

Memory complexity is still about 2^{31} and the number of chosen IV's needed is still about 2^{37}.

To demonstrate residual weaknesses for other key lengths, we briefly give a sketch of a possible attack using 128 bits long secret key and IV in Appendix A. It seems very likely that similar residual weaknesses hold for most key length and IV length, although they seem to be more efficient when large keys and IV's are used. For shorter lengths (less than 64 bits), they might become even more difficult to exploit, however the security of the system does not reach the initial security goal.

4.2 Distinguishing Attacks

It is also worth noticing that collisions in the PHT can also be used to distinguish Turing from a truly Random Generator with a much smaller complexity than the key recovery attack. For instance, when considering the case of 256 bits long secret key and 128 bits long IV's, randomizing only IV_3 will yield keystream collisions very quickly - after about 2^{16} queries.

If, in practice, the IV's where implemented using a counter, it seems likely that only IV_3 would differ between two consecutive sessions. In that case, one could distinguish Turing from random by detecting these collisions. Moreover, a condition on S is obtained this way (literally, relation (1) with an unknown value of R_{15}). This relation does not appear to be very useful, however it may allow to reduce the complexity of brute-force attacks by a small constant.

Recently, Rose and Hawkes [15] proposed a distinction between "weak" and "powerful" distinguishing attacks, by arguing that many proposed distinguishers against stream ciphers (see [6] and [15]) were not related to the practical security of the cipher. We believe however that in this particular case, early distinguishers against Turing (with less than 2^{16} chosen IV's) already leak partial information about the secret key and should not be tolerated. Therefore, although one can argue that the key recovery attack we propose is not practical, since the use of 2^{37} different IV's with the same secret key can be avoided, we think the weaknesses of the key scheduling algorithm we identified should still be fixed somehow.

5 Conclusion

In this paper, we identified several weaknesses in the key scheduling algorithm of the new stream cipher Turing. We believe these attacks will significantly lower the security of this cryptosystem when the same secret key can be used with many different initialization vectors. Our best attack works on 256 bits long secret key and 128 bits long IV's. In this case, it recovers the key using 2^{37} chosen IV, 2^{72} of computation time and 2^{36} bytes of memory.

Altough our attacks concern only the key scheduling, we think Turing should be modified to remove this vulnerability resulting from chosen IV attacks. The classical and probably simplest way to achieve that is to clock the LFSR several times before starting the keystream generation (16 clocking should be sufficient to remove undesirable properties on the LFSR initial state resulting from the IV). This might slightly decrease the speed of the cipher when IV needs to be frequently changed but it appears like a sound countermeasure.

An open problem would be, given the particular structure of Turing S-boxes, to improve the resolution of the linear system of equations we obtain in Section 3.5.

References

[1] The Magma Home Page. `http://www.maths.usyd.edu.au:8000/u/magma/`. 203

[2] Turing reference source code. Available at `http://people.qualcomm.com/ggr/QC/turing.tqz`. 196, 198

[3] D. Coppersmith, S. Halevi, and C. Jutla. Cryptanalysis of Stream Ciphers with Linear Masking. In M. Yung, editor, *Advances in Cryptology – Crypto'02*, volume 2442 of *Lectures Notes in Computer Science*, pages 515–532. Springer, 2002. 195, 197

[4] P. Ekdahl and T. Johansson. SNOW - a New Stream Cipher. In *First Open NESSIE Workshop, KU-Leuven*, 2000. Submission to NESSIE. Available at `http://www.it.lth.se/cryptology/snow/`. 195

[5] P. Ekdahl and T. Johansson. A New Version of the Stream Cipher SNOW. In *Selected Areas in Cryptography – 2002*, Lectures Notes in Computer Science. Springer, 2002. 197

[6] P. Ekdahl and T. Johansson. Distinguishing Attacks on SOBER-t16 and t32. In J. Daemen and V. Rijmen, editors, *Fast Software Encryption – 2002*, volume 2365 of *Lectures Notes in Computer Science*, pages 210–224. Springer, 2002. 205

[7] S. Fluhrer, I. Mantin, and A. Shamir. Weaknesses in the Key Scheduling Algorithm of RC4. In S. Vaudenay and A. M. Youssef, editors, *Selected Areas in Cryptography – 2001*, volume 2259 of *Lectures Notes in Computer Science*, pages 1–24. Springer, 2001. 195

[8] J.Dj. Golic, V. Bagini, and G. Morgari. Linear Cryptanalysis of Bluetooth Stream Cipher. In L. Knudsen, editor, *Fast Software Encryption – 2002*, volume 2332 of *Lectures Notes in Computer Science*, pages 238–255. Springer, 2002. 195

[9] S. Halevi, D. Coppersmith, and C. Jutla. Scream : a Software-efficient Stream Cipher. In L. Knudsen, editor, *Fast Software Encryption – 2002*, volume 2332 of *Lectures Notes in Computer Science*, pages 195–209. Springer, 2002. 197

[10] P. Hawkes and G. Rose. Primitive Specification and Supporting Documentation for SOBER-t32. In *First Open NESSIE Workshop*, 2000. Submission to NESSIE. 195

[11] P. Hawkes and G. Rose. Guess-and-Determine Attacks on SNOW. In *Selected Areas in Cryptography – 2002*, Lectures Notes in Computer Science. Springer, 2002. 197

[12] M. Liskov, R. Rivest, and D. Wagner. Tweakable Block Ciphers. In M. Yung, editor, *Advances in Cryptology – Crypto'02*, volume 2442 of *Lectures Notes in Computer Science*, pages 31–46. Springer, 2002. 194

[13] W. Meier and O. Stafflebach. Fast Correlations Attacks on Certain Stream Ciphers. In *Journal of Cryptology*, pages 159–176. Springer-Verlag, 1989. 197

[14] G. Rose. S32: A Fast Stream Cipher based on Linear Feedback over GF(2^{32}). Unpublished report, QUALCOMM, Australia, Available at `http://people.qualcomm.com/ggr/QC/`. 195

[15] G. Rose and P. Hawkes. On the Applicability of Distinguishing Attacks Against Stream Ciphers, 2002. Available at `http://eprint.iacr.org/2002/142.pdf`. 205

[16] G. Rose and P. Hawkes. Turing : a Fast Stream Cipher. In T. Johansson, editor, *Fast Software Encryption – 2003*, Lectures Notes in Computer Science. Springer, 2003. To appear. 195, 197, 198

[17] T. Siegenthaler. Correlation-immunity of Nonlinear Combining Functions for Cryptographic Applications. In *IEEE Transactions on Information Theory*, volume 30, pages 776–780, 1984. 197

[18] A. Stubblefield, J. Ioannidis, and A. D. Rubin. Using the Fluhrer, Mantin and Shamir Attack to Break WEP, 2001. 195

A Possible Attacks on 128 bits Long Secret Key

We now briefly describe what happens during the key scheduling when considering 128 bits long key and IV's. In this case, using previous notations, the initial LFSR state - before the PHT - is given by :

$$\begin{aligned} R_0 &= G(IV_0) \\ R_1 &= G(IV_1) \\ R_2 &= G(IV_2) \\ R_3 &= G(IV_3) \\ R_4 &= K_0 \\ R_5 &= K_1 \\ R_6 &= K_2 \\ R_7 &= K_3 \\ R_8 &= \texttt{0x01020344} \end{aligned} \qquad \begin{aligned} R_9 &= S(R_8 + R_0) \\ R_{10} &= S(R_9 + R_1) \\ R_{11} &= S(R_{10} + R_2) \\ R_{12} &= S(R_{11} + R_3) \\ R_{13} &= S(R_{12} + R_4) \\ R_{14} &= S(R_{13} + R_5) \\ R_{15} &= S(R_{14} + R_6) \\ R_{16} &= S(R_{15} + R_7) \end{aligned}$$

The influence of the initialization vector on the initial state is more complex than previously and the first application of NL seems to resist any collision. However, when looking at the second application of NL, we have

$$(C_1, \ldots, C_5) = NL(X_3, X_4, X_9, X_{16}, X_{19}) \oplus (X_4, X_5, X_{10}, X_{17}, X_{20})$$

using the same notations as in Section 3. More precisely,

$$\begin{aligned} X_{17} &= X_{15} \oplus X_4 \oplus \alpha X_0 \\ X_{19} &= X_{15} \oplus X_6 \oplus X_4 \oplus \alpha(X_0 \oplus X_2) \end{aligned}$$

Assume we use two IV's that differ on IV_0 and IV_1, verify $IV_2 = IV_0$, and have the same value of IV_3. If simultaneously, collisions on R_9, R_{15} and the word T computed by the PHT occur, then inputs of the second application of NL will be equal. It is a birthday paradox situation and should be observed after $(2^{16})^3 = 2^{48}$ experiences. Only two masking words are guaranteed to be equal in that case, which is not enough to detect efficiently this event. However, false alarms can be discarded by repeating the same experience with an other value of IV_3. Furthermore, collisions on the keyed S-box are again detected, so an attack similar to the one of Section 3 will work.

To summarize, a key recovery attack can also be mounted in this case, although the data complexity will be higher than in the previous case. In practice, the number of chosen IV's needed will be more than 2^{48}.