TweLEX: A Tweaked Version of the LEX Stream Cipher

Mainack Mondal *, Avik Chakraborti, Nilanjan Datta **,
Debdeep Mukhopadhyay ***

Abstract. LEX is a stream cipher proposed by Alex Biryukov. It was selected to phase 3 of the eSTREAM competition. LEX is based on the Advanced Encryption Standard (AES) block cipher and uses a methodology called *Leak Extraction*, proposed by Biryukov himself. However Dunkelman and Keller show that a key recovery attack exists against LEX. Their attack requires $2^{36.3}$ bytes of keystream produced by the same key and works with a time complexity of 2^{112} operations. In this work we explore LEX further and have shown that under the assumption of a related key model we can obtain 24 secret state bytes with a time complexity of 2^{96} and a data complexity of $2^{54.3}$. Subsequently, we introduce a tweaked version of LEX, called TweLEX, which is shown to resist all known attacks against LEX. Though the throughput of TweLEX is half of LEX, it is still 1.25 times faster than AES, the underlying block cipher. This work attempts to revive the principle of *leak extraction* as a simple and elegant method to design stream ciphers.
Keywords: Leak Extraction, Differential cryptanalysis, Tweak, Advanced Encryption Standard.

1 Introduction

LEX is a stream cipher designed by Alex Biryukov [1] using the round transformations of the Advanced Encryption Standard (AES). The proposal was built on the concept of *leak extraction*[1] and was based on the analysis of the diffusion of the transformations of AES to decide the extent of the *leak* and its frequency. LEX is a 128 bit key stream cipher, which is developed by extracting 32 bits from the state matrix of AES after each round of the cipher. The property of LEX, which makes it cryptanalytically different from AES is that the attacker never sees the entire 128 bit ciphertext, but sees a portion of it. The design objective of LEX was to design a fast stream cipher, and indeed was faster than AES in both hardware and software by 2.5 times.

LEX was selected as a candidate for eSTREAM competition and was seriously considered till the third phase for its elegance in construction, faster operation speed and high security margin. Its simplicity in construction attracted cryptanalytic efforts ([2] [3]) from several researchers. In response Biryukov submitted a tweaked version of LEX to the second phase of the eSTREAM competition in 2007, which could counter these attacks. However in 2008, Dunkelman and Keller [4] proposed a key recovery attack on *tweaked* LEX which required $2^{36.3}$ bytes of keystream produced by the same key and had time complexity of 2^{112}. This attack removed LEX from the final portfolio of eSTREAM. In spite of the above attacks,

* MPI-SWS, Germany (mainack@mpi-sws.org)
** ISI Kolkata ({avikchkrbrti,nilanjan.datta.isi}@gmail.com)
*** Comp. Sc. and Engg., IIT Kharagpur (debdeep@iitkgp.ac.in)

it may be appreciated that LEX is a stream cipher with high security margin while at the same time having an extremely simple design. The other selling point of LEX is that, the principle of *leak extraction* may be generalized to any block cipher, thus motivating deeper investigations of its strength.

The motivation of the present work is to modify LEX, so as to prevent all known attacks against the original cipher. In this work, we show further that under a related key model, 24 secret bytes can be recovered using an attack which has a time complexity of 2^{96} and a data complexity of $2^{54.3}$. This work motivates the *tweaking* of LEX into a modified cipher called TweLEX, to resist the original attacks by Dunkelman and Keller[4] and also other differential attacks. The work investigates the interesting idea of *leak extraction* for constructing robust stream ciphers from block ciphers. Although TweLEX is slower than LEX, it is still 1.25 times faster than the block cipher AES.

Organization

The paper is organized as follows: In Section 2 we describe the background of this paper. In Section 3 we describe the differential trail identified in the AES key schedule that we use to analyze LEX. Section 4 presents the related key based analysis to recover secret state bytes of LEX. The attack algorithm is analyzed for its data and time complexity in Section 5. The tweaked description of LEX is presented in Section 6, and we conclude this work in Section 7.

2 Preliminaries

2.1 Description of AES

The Advanced Encryption Standard (AES) is an 128 bit block cipher. It supports three key sizes, 128, 192 and 256. The construction of LEX is based on the structure of AES. So we describe the structure of AES very briefly.

AES considers a 128 bit plaintext as a byte matrix or state matrix of size 4×4. Here each byte represents an element of $GF(2^8)$. The state matrix undergoes 10 AES rounds of transformation for each AES encryption. An AES round consists of 4 operations applied to the state matrix in the following order :

- SubByte − An invertible non-linear transformation performed on each byte.
- ShiftRow (SR) − The bytes of the state matrix are permuted.
- MixColumn (MC) − Each column of the state matrix is multiplied by a constant 4×4 matrix.
- AddRoundKey (ARK) − The state matrix is xor-ed with a 128 bit round subkey.

For the ARK operation, 10 rounds an AES encryption requires a total of 10 subkeys. An extra subkey is also needed for *'key whitening'* step before the first round. AES derives these 11 subkeys from the original key using a key schedule algorithm [5].

2.2 Description of LEX

We describe the version of LEX explained in [1]. We define an AES′ encryption as the consecutive application of 10 AES rounds. There are two phases in LEX. In the

first phase or key initialization phase we choose a random string IV and encrypt it using AES encryption. Let us call this encrypted IV S. In the key generation phase S is encrypted using AES′ in the output feedback mode. During each round of AES′ 4 bytes from the state matrix are *leaked* as the output bytes of LEX. Thus for each AES′ encryption we have 40 keystream bytes from LEX. The bytes leaked

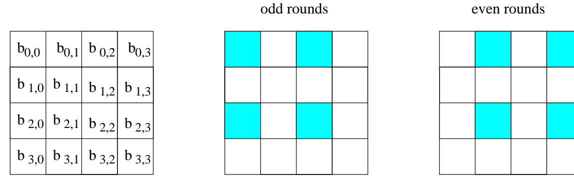

Fig. 1. Output bytes in odd and even rounds of LEX

in different rounds are shown in *Figure 1* by the colored squares. Therefore each encryption produces 40 bytes of key stream.

After 500 encryptions, another IV is chosen, and the process is repeated. After 2^{32} different IVs the key is replaced. Hence under this restriction by a single secret key we can get at most $500 \times 2^{32} \times 40$ bytes of key stream $= 2^{46.3}$ bytes of keystream.

2.3 Notations

In the rest of this paper AES will refer to the cipher AES′ as described in Section 2.2. The bytes of any intermediate state matrix B, of an AES encryption are denoted by $\{B_{i,j}\}_{i,j=0}^{3}$. Let E and E' be two AES encryptions. The secret key for E is K and that for E' is K^*. Let an intermediate state matrix of $E(E')$ be B (B'). Then the difference between B and B' be denoted by ΔB. The bytes of ΔB are denoted by $\{\Delta B_{i,j}\}_{i,j=0}^{3}$. The rth round key derived from a key K is named as K^r and its bytes are $\left\{K_{i,j}^r\right\}_{i,j=0}^{3}$.

In our analysis we have used two related keys K and K^*. The difference between bytes of rth round key derived from K and K^* is $\left\{\Delta K_{i,j}^r\right\}_{i,j=0}^{3}$. In this work $SB(x)$ denotes the output difference for an input difference of x to the *SubByte* operation. On the other hand $SubByte(x)$ denotes the output byte value for an input byte value of x to the SubByte operation. Throughout this work we have used mathematical operations in $GF(2^8)$. So the symbol '+' in this work refers to the operation xor.

2.4 Properties of AES SubBytes operation

Throughout this work we use the following properties of AES SubBytes operation.

Property 1. Given every non-zero input and output difference pair for the SubBytes operation, there are at most 4 input-output pairs which produce the given differences

We use this Property in situations where we know the input and output differences to SubBytes operation. In such cases, using the observation we can deduce candidates for actual values of the input and the output.

Property 2. Given any non zero input(output) difference to the SubByte operation, there are a total of 127 output(input) differences possible.

As a result, given a non zero input difference β to the SubByte operation and a random non zero difference γ, probability of the event that γ can be obtained using SubByte operation on difference $\beta = \frac{127}{255} \approx \frac{1}{2}$.

The analysis proposed in this work is based on a special differential trail in the key schedule of LEX. We describe the differential trail below.

3 Differential Trail in the Key Schedule of LEX

Let α be *a 32-bit word*. Let $\beta = \mathrm{SB}(\mathrm{RotByte}(\alpha))$. The operation RotByte and its use in AES is explained in [5]. Also α can be expressed as $[a, b, c, d]^T$. a, b, c, d are nonzero bytes. When we choose the key difference of the differential as

$$\Delta(K^r) = (\alpha \oplus \beta, \beta, 0, 0)$$

we can easily show that the next six 128-bit subkeys starting from K^r used in AES-128 have the following difference structure with probability 1:

$$\begin{array}{l}(\alpha \oplus \beta, \beta, 0, 0)\\(\alpha \oplus \beta, \alpha, \alpha, \alpha)\\(\alpha, 0, \alpha, 0)\\(\alpha, \alpha, 0, 0)\\(\alpha, 0, 0, 0)\\(\alpha, \alpha, \alpha, \alpha)\end{array}$$

Here each row describes the differences between two subkeys for the AES rounds, starting with the original key difference. In this key schedule we observe that starting from the third subkey to sixth subkey we have no sbox computation involved. This differential trail from third subkey to sixth subkey is depicted in the *Figure 2*. The box denotes SubBytes operation and $\oplus$ denotes xor.

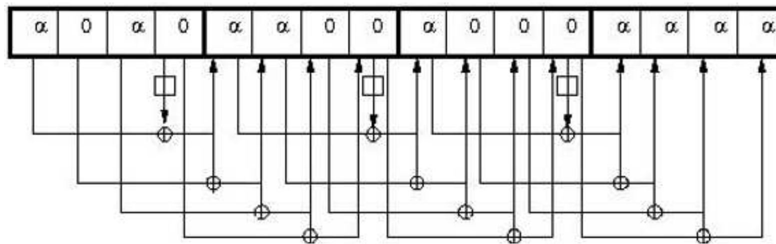

Fig. 2. Differential trail in the key scheduling of AES. The box denotes SubBytes operation and $\oplus$ denotes xor

In the coming sections while doing related key cryptanalysis on LEX we shall use this key schedule extensively. Here one thing needs to be noticed. If $\Delta(K)$ is $(\alpha \oplus \beta, \beta, 0, 0)$ then at third round we shall have $\Delta(K^3)$ as $(\alpha, 0, \alpha, 0)$. Hence if we want to have $\Delta(K^r)$ as $(\alpha, 0, \alpha, 0)$ in an odd round we can just set $\Delta(K)$ as $(\alpha \oplus \beta, \beta, 0, 0)$ and $r = 3$.

This is indeed the trick applied to the coming sections. Hence whenever we talk about r as an odd round, one can assume that $r = 3$. In fact if r is any higher odd round the analysis of this paper have to use related-subkey cryptanalysis. This is more complex form of cryptanalysis and less practical compared to related-key based analysis.

So from this point we consider all the results provided in the later sections are based on related - keys and not on related - subkeys.

3.1 Special Differential Properties in the LEX State Matrices

In this analysis we have used two related keys K and K^*. The rth round keys generated from K and K^* have the difference $(\alpha, 0, \alpha, 0)$. Here α is a 32 bit word which can also be expressed as $[a, b, c, d]^T$. Thus this difference pattern follows the differential trail shown in Section 3. Our cryptanalysis is successful in extracting secret state bytes when a special difference pattern appears just before the AddRoundKey step of the rth round. We first describe how to find the difference pattern using the key streams.

Consider two AES encryptions generated by keys K and K^*. Here K and K^* are related keys. Let us consider the event when the state matrices just before the rth AddRoundKey operation in both encryptions will have zero differences in 12 specific byte positions. Mathematically, if the corresponding differential state matrix is denoted by Δb, then $\{\Delta b_{0,0}, \Delta b_{0,1}, \Delta b_{0,2}, \Delta b_{0,3}, \Delta b_{1,0}, \Delta b_{1,2}, \Delta b_{2,0}, \Delta b_{2,1}, \Delta b_{2,2}, \Delta b_{2,3}, \Delta b_{3,0}, \Delta b_{3,2}\} = 0$, and $\{\Delta b_{1,1}, \Delta b_{1,3}, \Delta b_{3,1}, \Delta b_{3,3}\}$ are non zero values. This pattern is shown in *Figure 3* along with the value of the differences after the rth AddRoundkey operation.

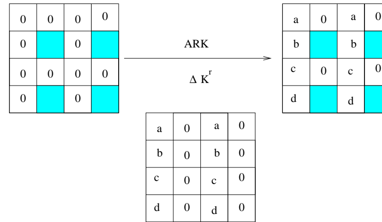

Fig. 3. The Special Difference Pattern

Next we compute the minimum number of key streams required to obtain such a difference pattern. Let the AES encryptions under the key K be sequenced as $E_1, E_2 \ldots E_n$, and with the key K^* be denoted as $E'_1, E'_2 \ldots E'_n$. The two colliding AES encryption pairs are obtained from the two sequences and may be denoted as E_i and E'_j.

Let us assume that with a minimum number of n AES encryptions for each of the keys K and K^* the attacker has at least one pair of encryptions with high probability where the required pattern holds. Consequently for that pair before the AddRoundKey operation of rth round, at 12 byte positions the actual values of the two state matrices generated by K and K^* will be same. We call this event a *collision* for a pair.

It can be shown that the optimal value of n is 2^{48} and hence to get a colliding pair with high probability we need 2^{48} encryptions by the key K. For each of them we have to consider 2^{48} encryptions by the key K^*. So there are a total of 2^{96} pairs.

Now using a 32 bit condition on the key stream we shall reduce the number of pairs to be considered. Let E_i and E'_j have a collision. If the state matrix after r^{th} round of E_i is denoted by B, then $\Delta B_{0,0}, \Delta B_{2,0}, \Delta B_{0,2}, \Delta B_{2,2}$ should be a, c, a, c respectively. It is expected that out of 2^{96} pairs only 2^{64} pairs will satisfy this condition.

Also if the state matrix after $(r+1)^{th}$ round of E_i is denoted by C, we can observe the differences $\Delta C_{0,1}, \Delta C_{0,3}, \Delta C_{2,1}, \Delta C_{2,3}$ for the remaining 2^{64} pairs from the key stream. Using the propagation of difference pattern it can be shown that these values need to satisfy 4 non linear equations.

Using Property 2 of Section 2.4, the probability of satisfying all of these 4 equations is 2^{-4}. So after checking this condition, out of 2^{64} pairs only 2^{60} pairs need to be considered.

The algorithm presented in the next Section uses this difference pattern. Hence we have to repeat the analysis procedure explained in that Section for each of the 2^{60} pairs to extract secret state bytes.

4 Algorithm to Recover Secret State Bytes

So far we have stated all the necessary properties required for our attack. In this Section using those properties we describe our algorithm which recovers certain state bytes of `LEX`. Our algorithm consists of two steps as presented below.

1. This step retrieves 8 bytes of the state matrix after round r. Let E_i and E'_j form a colliding encryption pair. Then the special difference shown in *Figure 3* holds for this pair. Now we shall concentrate on the propagation of this special difference pattern through round $(r+1)$. This propagation is shown in *Figure 4*. We shall explain the rest of this step using this Figure.

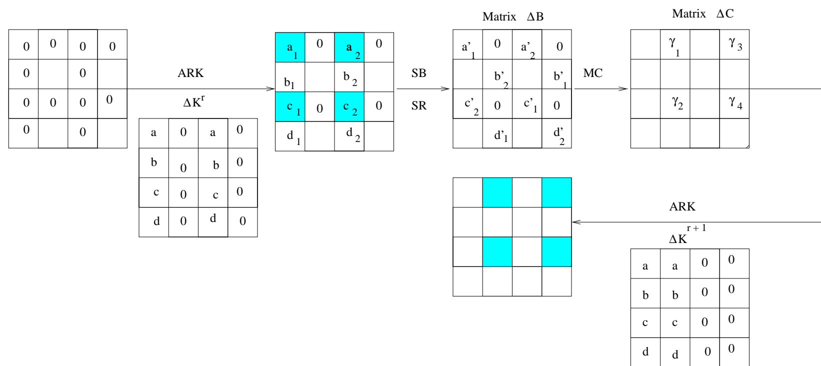

Fig. 4. Propagation of difference pattern in round (r + 1)(Bytes with known actual values are colored)

We know the actual values of the colored bytes of *Figure 4* from the key stream. The symbols written in the byte positions are corresponding differences obtained for the pair E_i and E'_j. Let us first express the differences $a_1, a_2, b_1, b_2, c_1, c_2, d_1, d_2$ in *Figure 4* in terms of known differences a, b, c, d . The difference of the rth subkeys for E_i and E'_j is known from the related key differential of Section 3 and shown in *Figure 4*. Observing the differences before and after xoring with round key K^r and using the linearity of ARK operation we immediately get $a_1 = a_2 = a$, $b_1 = b_2 = b$, $c_1 = c_2 = c$, $d_1 = d_2 = d$. In *Figure 4* x' denotes output difference after SubByte operation applied to input difference x. Mathematically if x is a difference, $x' = SB(x)$ where x can be $a_1, a_2, b_1, b_2, c_1, c_2, d_1$ or d_2.

Now we observe the differences between the key stream generated by E_i and E'_j after $(r+1)^{th}$ round. From the related key differentials in `LEX` (Section 3) we know the subkey difference ΔK^{r+1} as shown in *Figure 4*. Hence using the linearity of ARK operation we can get the values of the differences $\gamma_1, \gamma_2, \gamma_3, \gamma_4$ in *Figure 4*. Let us denote the state matrix after SR operation of $(r+1)^{th}$ round as B. C is the state matrix after applying MC operation to B. Then using the linearity of MC operation we can express each of $\Delta C_{0,1}, \Delta C_{2,1}$ as linear combination of b'_2, d'_1. Also $\Delta C_{0,3}, \Delta C_{2,3}$ can be expressed as linear combination of b'_1, d'_2. Since $\Delta C_{0,1}, \Delta C_{2,1}, \Delta C_{0,3}, \Delta C_{2,3}$ are nothing but known values $\gamma_1, \gamma_2, \gamma_3, \gamma_4$ respectively hence we get 4 equations in 4 variables b'_1, b'_2, d'_1, d'_2, The attacker deduces b'_1, b'_2, d'_1, d'_2 from these equations. By virtue of knowing the differences b, d we know b_1, b_2, d_1, d_2. Hence using Property 1 of Section 2.4, actual values of bytes corresponding to these four differences b_1, b_2, d_1, d_2 can be retrieved by four table look ups. In this way we recover four bytes of the state matrix after the rth round. Combined with the bytes we get from the key stream a total of eight bytes of the state matrix after the rth round is known. The bytes of round $(r+1)$ whose actual values are known after this step are shown in *Figure 5* by coloring them.

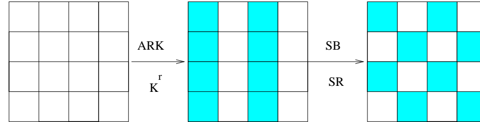

Fig. 5. Known bytes after first step of algorithm

2. We recover some more bytes of another state matrix in this step. We continue with the colliding encryption pair E_i and E'_j from step 1. Here we concentrate on the propagation of differences in the rth round. *Figure 6* shows the differences of different bytes in the rth round. The bytes whose actual values are known to us from the key stream are colored. In this diagram, all of c_1, c_2, c_3, c_4 are differences we can get directly from the key stream. We want to determine x_i, $i = 1, 2, \ldots, 8$. They are unknown differences.

   Here G is the state matrix after SR operation of round r in the encryption E_i. H is obtained after MC operation is applied to G. Since MC operation is linear we apply MC operation on 2nd and 4th column of ΔG and equate them

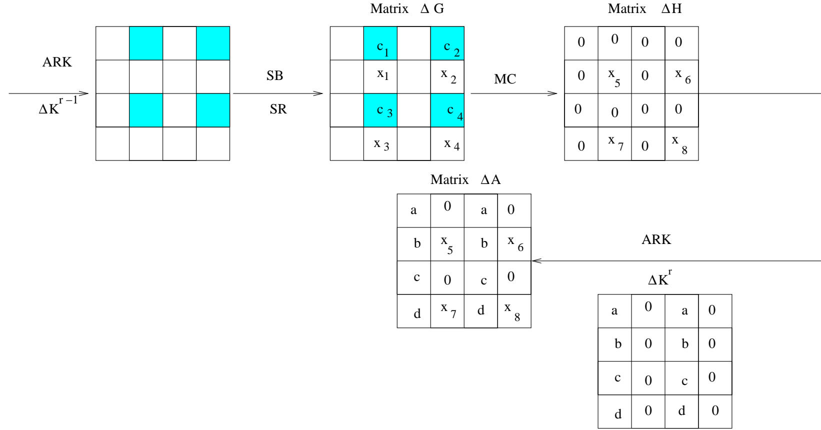

Fig. 6. Propagation of differences in round r

with corresponding columns of ΔH. In this way we can form 8 linear equations involving x_is, $i = 1, 2, ..., 8$.
We solve this system of linear equation and in particular focus on the values of x_1, x_2, x_3, x_4 i. e the values of $\Delta G_{1,1}, \Delta G_{1,3}, \Delta G_{3,1}, \Delta G_{3,3}$. Thus we get the total 2nd and 4th column of ΔG. Next we guess the actual values of the 2nd and 4th column of the matrix G. This takes a complexity of 2^{32}. Hence using the MC operation we get the actual values of 2nd and 4th column of H too. Let the state matrix at the end of rth round of E_i is denoted by A. Using the values of x_5, x_6, x_7, x_8 combined with the known value of ΔK^r we know the 2nd and 4th column of ΔA. The bytes whose actual values are known after step 2 are shown in *Figure 7* by coloring them.

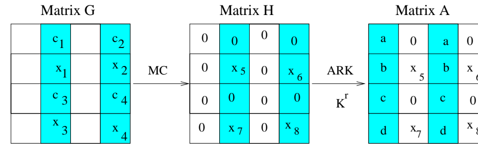

Fig. 7. Known bytes after second step of the algorithm

5 Analysis of the algorithm

5.1 Complexity Analysis

The algorithm depends on finding the special difference pattern mentioned in Section 3.1. We also state in the same section that we need a minimum of 2^{48} encryptions under each of the key K and K^*. Hence the total number of key stream bytes required is $\approx 2^{54.3}$. This is the data complexity of the algorithm.

Step 1 of the algorithm has a complexity of 2^4 on average. The second step has a complexity of 2^{32} since we have to make 2^{32} guesses and we have corresponding

key suggestions. Thus the 2 steps of the attack procedure have a total complexity of 2^{36}. We have to repeat this procedure for 2^{60} pairs as explained in Section 3.1. Hence the time complexity of the algorithm is 2^{96}.

5.2 Efficiency of the Algorithm

In this Section we show that the algorithm retrieves the intermediate state bytes more efficiently than brute force search.

We can see from *Figure 7* we have recovered 8 bytes of matrix H and 8 bytes of matrix A. These 8 bytes of matrix A also include 4 bytes given by the key stream. Hence we have recovered total 12 secret state bytes of matrix H and A.

A careful look at the algorithm reveals that we have retrieved not only secret bytes of matrix A and matrix H for the encryption E_i but also the corresponding secret bytes of E'_j. So we have retrieved 24 bytes of secret data with a cost of 2^{96}. This is much lesser than an exhaustive search for these 24 bytes which should cost 2^{192}.

In the following Section we present a tweaked version of `LEX` which is resistant against all the attacks mentioned so far in this paper.

6 Proposal of the tweaked version of `LEX`: `TweLEX`

In this Section we propose a modification of `LEX` which provides resistance against existing attacks on `LEX`. We also argue that this simple modification provides protection against any kind of differential attack on `LEX`.

6.1 Modification of `LEX`

We have seen that the most important part of the design of `LEX` is wherefrom a designer should output the bytes. Let after an odd round the state matrix is M and in the next round, which is essentially an even round the state matrix is N.

Then in the basic design the output bytes for those two consecutive rounds would have been $M_{0,0}, M_{0,2}, M_{2,0}, M_{2,2}, N_{0,1}, N_{0,3}, N_{2,1}, N_{2,3}$. In the tweaked version of `LEX` we shall output $(M_{0,0} \oplus N_{0,1})$,$(M_{0,2} \oplus N_{0,3})$,$(M_{2,0} \oplus N_{2,1})$,$(M_{2,2} \oplus N_{2,3})$. We call the resulting stream cipher `TweLEX`. It is advisable to change the secret key of the cipher at least every 2^{32} IV set ups, and to change the IV after every $T = 1000$ iterations to prevent algebraic attacks.

We output 4 bytes after every two `AES` round. This indeed halves the throughput of `LEX` but it provides better security than the actual `LEX`. We shall now provide some argument in order to justify the design.

6.2 Security Analysis of `TweLEX`

In this subsection, we provide an overview on the performance of the existing attacks against `LEX` on the new cipher `TweLEX`.

Protection Against the Key Recovery attack by Dunkelman: The attack in [4] relies on finding the event that the difference between two state matrices is zero at 8 specific bytes. The probability of this event is 2^{-64}.

They have used a 32 bit filter on the output key stream of `LEX`. This filter effectively reduces the complexity of their algorithm by a factor of 2^{32}. In `TweLEX` the adversary does not directly know the values of state bytes. Hence he could not use the 32 bit filter and the complexity of attack will become $2^{112} \times 2^{32} = 2^{144}$. This is worse than a brute force attack.

Protection Against the state byte recovery attack proposed in this work: In this work we have used filters on the key stream of `LEX` to find out a differential pattern. Hence using a argument similar to previous paragraph we can show that our attack is ineffective agaisnt `TweLEX`.

Protection Against other Differential Attacks: A close look on all the differential attacks on `AES` or `LEX` which relies on differential propagation through rounds reveals a similarity. All of them use the Property 1 of Section 2.4. They try to locate input-output difference pairs for the `AES` S-box and using a table look up retrieve corresponding actual input-output byte values.

Let us use this strategy for `TweLEX`. We assume that we have two or more key streams generated by `TweLEX`. By construction the key stream will be xor of certain state bytes. Using some differential propagation on the differences of two key streams one can retrieve some output differences for `AES` s-box for certain bytes of the state matrix. Then it can be shown that, using Property 1 of Section 2.4, it is not possible to retrieve the state bytes with a complexity better than brute force search. Hence this is apparent that any strategy that uses the Property 1 of Section 2.4 will fail against this modified version of `LEX`.

7 Conclusion

In this paper we have revisited the stream cipher `LEX` and have presented a related key cryptanalysis for the cipher. Consequently we proposed `TweLEX`, a modification of `LEX` which prevents the existing attacks on `LEX`. Although the throughput of `TweLEX` is half of `LEX` it provides a good optimization between speed and security and attempts to show that *leak extraction* as an elegant method to design stream ciphers.

References

1. Alex Biryukov, "A New 128-bit Key Stream Cipher LEX," Ecrypt Stream Cipher Project Report, 2005/013, 2005, http://ecrypt.eu.org/stream.
2. H. Wu and B. Preneel, "Attacking the IV setup of the stream cipher LEX," Ecrypt Stream Cipher Project Report, 2005/059, 2005, http://ecrypt.eu.org/stream.
3. Håkan Englund, Martin Hell and Thomas Johansson, "A Note on Distinguishing attacks," in *Preproceedings of State of the Art of Stream Ciphers workshop (SASC 2007)*, 2007, pp. 73–78.
4. Orr Dunkelman and Nathan Keller, "A new attack on the lex stream cipher," in *ASIACRYPT*, 2008, pp. 539–556.
5. Federal Information Processing Standards Publication 197, "Announcing the Advanced Encryption Standard (AES)," 2001.