Block and Stream Ciphers and the Creatures in Between

Alex Biryukov

University of Luxembourg, FSTC,
6, rue Richard Coudenhove-Kalergi,
L-1359 Luxembourg-Kirchberg LUXEMBOURG
`http://www.esat.kuleuven.ac.be/~abiryuko/`

Abstract. In this paper we define a notion of leak extraction from a block cipher. We demonstrate this new concept on an example of AES. A result is LEX: a simple AES-based stream cipher which is at least 2.5 times faster than AES both in software and in hardware.

1 Introduction

In this paper we suggest a simple notion of a *leak extraction* from a block cipher. The idea is to extract parts of the internal state at certain rounds and give them as the output key stream (possibly after passing an additional filter function). This idea applies to any block cipher but a careful study by cryptanalyst is required in each particular case in order to decide which parts of the internal state may be given as output and at what frequency. This mainly depends on the strength of the cipher's round function and on the strength of the cipher's key-schedule. For example, ciphers with good diffusion might allow to output larger parts of the internal state at each round than ciphers with weak diffusion.

In this paper we describe our idea on an example of 128/192/256 bit key AES. Similar approach may be applied to the other block-ciphers, for example to Serpent. Interesting lessons learnt from LEX so far are that: LEX setup and resynchronization which are just a single AES key-setup and a single AES encryption are much faster than for most of the other stream ciphers (see performance evaluation of eSTREAM candidates [8]). This is due to the fact that many stream ciphers aimed at fast encryption speed have a huge state which takes very long time to initialize. Also, the state of the stream ciphers has to be at least double of the keysize in order to avoid tradeoff attacks, but on the other hand it does not have to be more than that. Moreover unlike in a typical stream cipher, where all state changes with time, in LEX as much as half of the state does not need to be changed or may evolve only very slowly.

2 Description of LEX

In this section we describe a 128-bit key stream cipher LEX (which stands for Leak EXtraction, and is pronounced "leks"). In what follows we assume that the reader is familiar with the Advanced Encryption Standard Algorithm (AES) [7]. The design is simple and is using AES in a natural way: at each AES round we output certain four bytes from the intermediate state. The AES with all three different key lengths (128, 192, 256) can be used. The difference with AES is that the attacker never sees the full 128-bit ciphertext but only portions of the intermediate state. Similar principle can be applied to any other block-cipher.

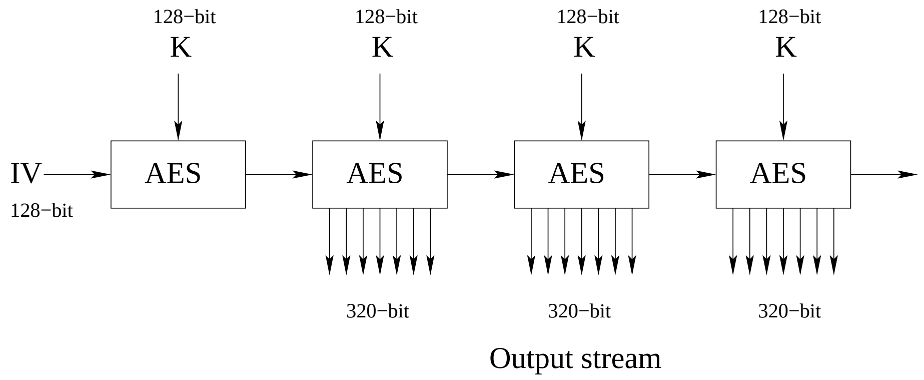

Fig. 1. Initialization and stream generation.

In Fig. 1 we show how the cipher is initialized and chained[1]. First a standard AES key-schedule for some secret 128-bit key K is performed. Then a given 128-bit IV is encrypted by a single AES invocation: $S = AES_K(IV)$. The 128-bit result S together with the secret key K constitute a 256-bit secret state of the stream cipher.[2] S is changed by a round function of AES every round and K is kept unchanged (or in a more secure variant is changing every 500 AES encryptions).

The most crucial part of this design is the exact location of the four bytes of the internal state that are given as output as well as the frequency of outputs (every round, every second round, etc.). So far we suggest to use the bytes $b_{0,0}, b_{2,0}, b_{0,2}, b_{2,2}$ at every odd round and the bytes $b_{0,1}, b_{2,1}, b_{0,3}, b_{2,3}$ at every even round. We note that the order of bytes is not relevant for the security but is relevant for the fast software implementation. The order of bytes as given above allows to extract a 32-bit value from two 32-bit row variables t_0, t_2 in just four operations (that can be pipelined):

$$out32 = ((t_0 \& 0xFF00FF) << 8) \oplus (t_2 \& 0xFF00FF),$$

[1] There is a small caveat: we use full AES to encrypt the IV, but we use AES with slightly modified last round for the stream generation, as will be explained further in this section.

[2] In fact the K part is expanded by the key-schedule into ten 128-bit subkeys.

while each round of AES uses about 40 operations. Here t_i is a row of four bytes: $t_i = (b_{i,0}, b_{i,1}, b_{i,2}, b_{i,3})$. So far we do not propose to use any filter function and output the bytes as they are. The choice of the output byte locations (see also

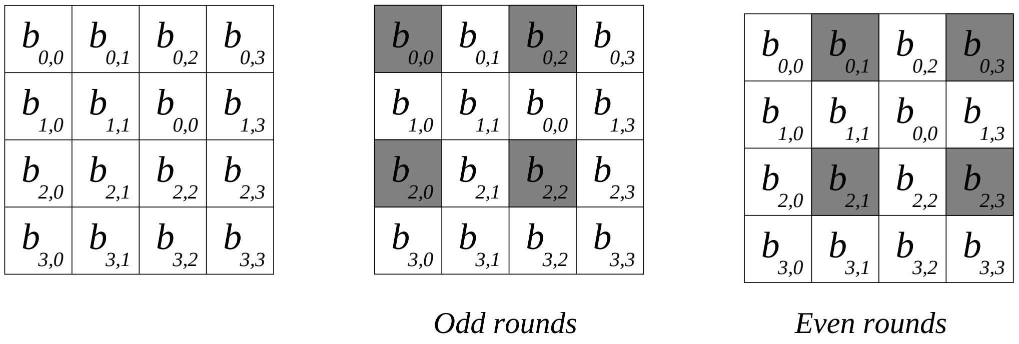

Fig. 2. The positions of the leak in the even and in the odd rounds.

Fig. 2) is motivated by the following: both sets constitute an invariant subset of the `ShiftRows` operation (the first row is not shifted and the third is rotated by two bytes). By alternating the two subsets in even and odd rounds we ensure that the attacker does not see input and output bytes that are related by a single `SubBytes` and a single `MixColumn`. This choice ensures that the attacker will have to analyze two consecutive rounds. The two rounds of AES have full diffusion thus limiting divide-and-conquer capabilities of the attacker. Note also that in AES the 10th round differs from the rest, there is no `MixColumn` and there is a XOR of the last (11th) subkey. In LEX there is no need to make the 10th round different from any other round. Any LEX encryption round consists of:

```
Round(State, i)
{ SubBytes(State);
  ShiftRows(State);
  MixColumns(State);
  AddRoundKey(State, ExpandedKey[i mod N_r]);
}
```

Here N_r is the number of rounds and is equal to 10 for 128-bit key AES. The full T iterations of LEX would then look like:

```
LEX(State, SecretKey)
{
AESKeyExpansion(SecretKey, ExpandedKey);
State = AESEncrypt(IV, ExpandedKey);
AddroundKey(State, ExpandedKey[0]);
for (i=1; i < T; i++){
  Round(State, i);
  Output[i] = LeakExtract(State, i mod 2);
 }
}
```

It is advisable to change the *SecretKey* at least every 2^{32} IV setups, and to change the IV every $T = 500$ iterations.

Note also that IV setup is performed by full AES encryption and the subtle difference in the last round of AES and absence of such difference in encryption rounds of LEX is crucial to break similarity which otherwise could be exploited by slide attacks [5, 11] (see Section 3.8 for a discussion).

The speed of this cipher is more than 2.5 times faster than 128-bit key AES, 3 times faster than 192-bit key AES, and 3.5 times faster than 256-bit key AES. So far there are no weaknesses known to the designers as well as there are no hidden weaknesses inserted by the designers.

3 Analysis of LEX

In this section we analyze resistance of LEX to various attacks.

3.1 Period of the Output Sequence

The way we use AES is essentially an Output Feedback Mode (OFB), in which instead of using the ciphertexts as a key-stream we use the leaks from the intermediate rounds as a key-stream. The output stream will eventually cycle when we traverse the full cycle of the AES-generated permutation. If one assumes that AES is indistinguishable from a random permutation for any fixed key, one would expect the cycle size to be of the order $O(2^{128})$ since the probability of falling into one of the short cycles is negligible[3].

3.2 Tradeoff Attacks

For a stream cipher to be secure against time-memory and time-memory-data tradeoff attacks [1, 9, 4] the following conditions are necessary: $|K| = |IV| = |State|/2$. This ensures that the best tradeoff attack has complexity roughly the same as the exhaustive key-search. The IV's may be public, but it is very important that full-entropy IV's are used to avoid tradeoff-resynchronization attacks [3, 10]. In the case of LEX $|K| = |IV| = |Block| = 128$ bits, where *Block* denotes an intermediate state of the plaintext block during the encryption. Internal state is the pair (IV, K) at the start and $(Block, Key)$ during the stream generation, and thus $|K| + |IV| = |K| + |S| = 256$ bits which is enough to avoid

[3] A random permutation over n-bit integers typically consists of only about $O(n)$ cycles, the largest of them spanning about 62% of the space.

the tradeoff attacks. Note that if one uses LEX construction with larger key variants of AES this might be a "problem". For example for 192-bit key AES the state would consist of 128-bit internal variable and the 192-bit key. This would allow to apply a time-memory-data tradeoff attack with roughly 2^{160} stream, memory and time. For 256-bit key AES it would be 2^{192} stream, memory and time. Such attack is absolutely impractical but may be viewed as a certificational weakness.

3.3 Algebraic Attacks

Algebraic attack on stream ciphers [6] is a recent and a very powerful type of attack. Applicability of these to LEX is to be carefully investigated. If one could write a non-linear equation in terms of the outputs and the key – that could lead to an attack. Re-keying every 500 AES encryptions may help to avoid such attacks by limiting the number of samples the attacker might obtain while targeting a specific subkey. We expect that after the re-keying the system of non-linear equations collected by the attacker would become obsolete. Shifting from AES key-schedule to a more robust one might be another precaution against these attacks. Note also that unlike in LFSR-based stream ciphers we expect that there do not exist simple relations that connect internal variables at distances of 10 or more steps. Such relations if they would exist would be useful in cryptanalysis of AES itself.

3.4 Differential, Linear or Multiset Resynchronization Attacks

If mixing of IV and the key is weak the cipher might be prone to chosen or known IV attacks similar to the chosen plaintext attacks on the block-ciphers. However in our case this mixing is performed via a single AES encryption. Since AES is designed to withstand such differential, linear or multiset attacks we believe that such attacks pose no problem for our scheme either.

3.5 Potential Weakness – AES Key-schedule

There is a simple way to overcome weaknesses in AES key-schedule (which is almost linear) and which might be crucial for our construction. One might use ten consecutive encryptions of the IV as subkeys, prior to starting the encryption. This method will however loose in key agility, since key-schedule time will be 11 AES encryptions instead of one. If better key-agility is required a faster dedicated key-schedule may be designed.

If bulk encryption is required then it might be advisable to replace the static key with a slowly time-varying key. One possibility would be to perform an additional 10 AES encryptions every 500 AES encryptions and to use the 10 results as subkeys. This method is quite efficient in software but might not be suitable for small hardware due to the requirement to store 1280 bits (160 bytes) of the subkeys. The overhead of such key-change is only 2% slowdown, while it might stop potential attacks which require more than 500 samples gathered for a specific subkey. An alternative more gate-efficient solution would be to perform a single AES encryption every 100 steps without revealing the intermediate values and use the result as a new 128-bit key. Then use the keyschedule of AES to generate the subkeys. Note, that previously by iterating AES with the same key we explored a single cycle of AES, which was likely to be of length $O(2^{128})$ due to the cipher being a permutation of 2^{128} values. However by doing intermediate key-changes we are now in a random mapping scenario. Since state size of our random mapping is 256 bits (key + internal state), one would expect to get into a "short cycle" in about $O(2^{128})$ steps, which is the same as in the previous case and poses no security problem.

3.6 No Weak Keys

Since there are no weak keys known for the underlying AES cipher we believe that weak keys pose no problem for this design either. This is especially important since we suggest frequent rekeying to make the design more robust against other cryptanalytic attacks.

3.7 Dedicated Attacks

An obvious line of attack would be to concentrate on every 10th round, since it reuses the same subkey, and thus if the attacker guesses parts of this subkey he still can reuse this information $10t, t = 1, 2, \ldots$ rounds later. Note however that unlike in LFSR or LFSM based stream ciphers the other parts of the intermediate state have hopelessly changed in a complex non-linear manner and any guesses spent for those are wasted (unless there is some weakness in a full 10-round AES).

3.8 The Slide Attack

In [11] a slide attack [5] on resynchronization mechanism of LEX (as it was described for the eSTREAM project) is shown. The attack requires the ability to perform 2^{61} resynchronizations and uses 2^{75} bytes of output stream data produced under a single key and different IVs, which need to be stored and sorted in 2^{75} bytes of memory. This attack is comparable in complexity to time-memory-key tradeoff attacks which are applicable to any block cipher in popular modes of operation like ECB, CBC (time-memory-data complexity of $O(2^{64})$ for any 128-bit cipher) [2, 3][4] This attack thus does not make LEX weaker than 128-bit key AES.

[4] One may argue that attack on a single key is more interesting than the tradeoff attack that breaks one key out of 2^{64}. Firstly we think that it is subjective and depends on the appliation. Secondly, if we limit the amount of stream produced per key to 2^{32} as is typical for many other stream-ciphers, this argument will not be valid any more. The slide attack will have 2^{96} complexity and will need to try the same amount of keys as the tradeoff attack – 2^{64}, before it succeeds.

However the observation leading to the attack is of interest since it can be easily generalized and would apply to any leak-extraction cipher in which resynchronization and encryption are performed by the same function. The idea of the attack is simple: iterations of LEX explore a cycle of the size about 2^{128} starting from IV. Random IV selections would sample random points on this cycle. If the IV setup is performed by the same function as the subsequent stream generation then one may pick an IV which is equal to the block-state just after the IV setup of another sample. This causes the attacker to know the full block input of the cipher and the result of the leak one round later, which clearly leaks lots of information about the secret subkey of that round. In order to find such colliding block-states the attacker needs at least 2^{65} block samples stored and sorted in memory. The attack assumes the ability to perform about 2^{64} resynchronizations for the same key.

A natural way to increase resistance against the attack would be to require a change of keys every 2^{32} IV's. There would still remain a chance of 2^{-64} to find colliding block-states in a collection of 2^{32} IV samples. However the complexity of the attack would increase to 2^{96} and the attacker would need to try the attack for 2^{64} different keys – the same number as in the tradeoff attack. Such high complexity should be a sufficient protection for most of the practical purposes. In addition, in order to completely get rid of the sliding property one should use two different functions for the resynchronization and the encryption. Moreover even a small difference between the two would suffice. For example, if one uses the full AES with the XOR of the last subkey for the IV setup and AES without the XOR of this subkey for the encryption – this is enough to break the similarities used by sliding.

4 Implementation

As one may observe from software performance test done by ECRYPT [8], LEX holds to its promise and runs 2.5 times faster than 128-bit key AES. We expect that the same holds for hardware implementations. It is also somewhat pleasantly surprising that LEX is one of the fastest ciphers out of the 32 candidates on many of the platforms: 6th on Intel Pentium M, 1700MHz; 4th on Intel Pentium 4, 2.40GHz; 6th on AMD Athlon 64 3000+, 1.80GHz; 7th on PowerPC G4 533MHz; 6th on Alpha EV5.6, 400MHz; 5th on HP 9000/785, 875MHz; 5th on UltraSPARC-III, 750MHz). It is also one of the best in terms of agility of the key-setup, the IV-setup, and the combined Internet packet metric IMIX. LEX is thus very well suited for the short packet exchanges typical for the Internet environment.

Since LEX could reuse existing AES implementations it might provide a simple and cheap speedup option in addition to the already existing base AES encryption. For example, if one uses a fast software AES implementation which runs at 14-15 clocks per byte we may expect LEX to be running at about 5-6 clocks per byte. The same leak extraction principle naturally applies to 192 and 256-bit AES resulting in LEX-192 and LEX-256. LEX-192 should be 3 times faster than AES-192, and LEX-256 is 3.5 times faster than AES-256. Note that unlike in AES the speed penalty for using larger key versions is much smaller in LEX (a slight slowdown for a longer keyschedule and resynchronization but not for the stream generation).

5 Strong Points of the Design

Here we list some benefits of using this design:

– AES hardware/software implementations can be reused with few simple modifications. The implementors may use all their favorite AES implementation tricks.
– The cipher is at least 2.5 times faster than AES. In order to get an idea of the speed of LEX divide cycles-per-byte performance figures of AES by a factor 2.5. The speed of key and IV setup is equal to the speed of AES keyschedule followed by a single AES encryption. In hardware the area and gate count figures are essentially those of the AES.
– Unlike in the AES the key-setup for encryption and decryption in LEX are the same.
– The cipher may be used as a speedup alternative to the existing AES implementation and with only minor changes to the existing software or hardware.
– Security analysis benefits from existing literature on AES.
– The speed/cost ratio of the design is even better than for the AES and thus it makes this design attractive for both fast software and fast hardware implementations. The design will also perform reasonably well in restricted resource environments.
– Since this design comes with explicit specification of IV size and resynchronization mechanism it is secure against time-memory-data tradeoff attacks. This is not the case for the AES in ECB mode or for the AES with IV's shorter than 128-bits.
– Side-channel attack countermeasures developed for the AES will be useful for this design as well.

6 Summary

In this paper we have suggested a new concept of conversion of block ciphers into stream ciphers via *leak extraction*. As an example of this approach we have described efficient extensions of AES into the world of stream ciphers, which we called LEX. We expect that (if no serious weaknesses would be found) LEX may provide a very useful speedup option to the existing base implementations of AES. We hope that there are no attacks on this design faster than $O(2^{128})$ steps. The design is rather bold and of course requires further study.

7 Acknowledgment

This paper is a result of several inspiring discussions with Adi Shamir. We would like to thank Christophe De Cannière, Joseph Lano, Ingrid Verbauwhede and other cosix for the exchange of views on the stream cipher design. We also would like to thank anonymous reviewers for comments that helped to improve this paper.

References

[1] S. Babbage, "Improved "exhaustive search" attacks on stream ciphers," in *ECOS 95 (European Convention on Security and Detection)*, no. 408 in IEE Conference Publication, May 1995.

[2] E. Biham, "How to decrypt or even substitute DES-encrypted messages in 2^{28} steps," *Information Processing Letters*, vol. 84, pp. 117–124, 2002.

[3] A. Biryukov, S. Mukhopadhyay, and P. Sarkar, "Improved Time-Memory Trade-offs with Multiple Data," in *Proceedings of SAC'05*, Lecture Notes in Computer Science, Springer-Verlag, 2005.

[4] A. Biryukov and A. Shamir, "Cryptanalytic time/memory/data tradeoffs for stream ciphers," in *Proceedings of Asiacrypt'00* (T. Okamoto, ed.), no. 1976 in Lecture Notes in Computer Science, pp. 1–13, Springer-Verlag, 2000.

[5] A. Biryukov and D. Wagner, "Slide attacks," in *Proceedings of Fast Software Encryption – FSE'99* (L. R. Knudsen, ed.), no. 1636 in Lecture Notes in Computer Science, pp. 245–259, Springer-Verlag, 1999.

[6] N. T. Courtois and W. Meier, "Algebraic attacks on stream ciphers with linear feedback," in *Advances in Cryptology – EUROCRYPT 2003* (E. Biham, ed.), Lecture Notes in Computer Science, pp. 345–359, Springer-Verlag, 2003.

[7] J. Daemen and V. Rijmen, *The design of Rijndael: AES — The Advanced Encryption Standard*. Springer-Verlag, 2002.

[8] eSTREAM, "eSTREAM Optimized Code HOWTO," 2005. `http://www.ecrypt.eu.org/stream/perf/`.

[9] J. D. Golic, "Cryptanalysis of alleged A5 stream cipher," in *Advances in Cryptology – EUROCRYPT'97* (W. Fumy, ed.), vol. 1233 of *Lecture Notes in Computer Science*, pp. 239–255, Springer-Verlag, 1997.

[10] J. Hong and P. Sarkar, "Rediscovery of time memory tradeoffs," 2005. `http://eprint.iacr.org/2005/090`.

[11] H. Wu and B. Preneel, "Attacking the IV Setup of Stream Cipher LEX," in *Proceedings of Fast Software Encryption – FSE'06* (M. Robshaw, ed.), Lecture Notes in Computer Science, Springer-Verlag, 2006.