Bias in the LEVIATHAN Stream Cipher

Paul Crowley[1⋆] and Stefan Lucks[2⋆⋆]

[1] cryptolabs Amsterdam
paul@cryptolabs.org
[2] University of Mannheim
lucks@weisskugel.informatik.uni-mannheim.de

Abstract. We show two methods of distinguishing the LEVIATHAN stream cipher from a random stream using 2^{36} bytes of output and proportional effort; both arise from compression within the cipher. The first models the cipher as two random functions in sequence, and shows that the probability of a collision in 64-bit output blocks is doubled as a result; the second shows artifacts where the same inputs are presented to the key-dependent S-boxes in the final stage of the cipher for two successive outputs. Both distinguishers are demonstrated with experiments on a reduced variant of the cipher.

1 Introduction

LEVIATHAN [5] is a stream cipher proposed by David McGrew and Scott Fluhrer for the NESSIE project [6]. Like most stream ciphers, it maps a key onto a pseudorandom keystream that can be XORed with the plaintext to generate the ciphertext. But it is unusual in that the keystream need not be generated in strict order from byte 0 onwards; arbitrary ranges of the keystream may be generated efficiently without the cost of generating and discarding all prior values. In other words, the keystream is "seekable". This property allows data from any part of a large encrypted file to be retrieved efficiently, without decrypting the whole file prior to the desired point; it is also useful for applications such as IPsec [2]. Other stream ciphers with this property include block ciphers in CTR mode [3]. LEVIATHAN draws ideas from the stream ciphers WAKE [9] and SEAL [7], and the GGM pseudo-random function (PRF) construction [1].

The keystream is bounded at 2^{50} bytes. Though the security goals are stated in terms of key recovery, it is desirable that distinguishing this keystream from a random binary string should be as computationally expensive as an exhaustive search of the 128 or 256-bit keyspace. Keystream generation is best modelled as a key-dependent function $\mathrm{Lev} : \{0,1\}^{48} \mapsto \{0,1\}^{32}$, mapping a location in the stream to a 32-bit output word; catenating consecutive values of this function from 0 gives the entire keystream:

$$\mathrm{Lev}(0)|\mathrm{Lev}(1)|\mathrm{Lev}(2)|\dots|\mathrm{Lev}(2^{48}-1)$$

⋆ This research was supported by convergence integrated media GmbH

⋆⋆ This research was supported by Deutsche Forschungsgemeinschaft (DFG) grant Kr 1521/2

Finding Lev(i) for arbitrary i is not especially fast. However, once this is done, intermediate values can usually be reused to find Lev($i+1$), Lev($i+2$)... much more efficiently. This is because the internal structure of the cipher is based on a forest of 2^{32} binary trees, each of which generates 2^{16} words of output, as shown in Figure 1.

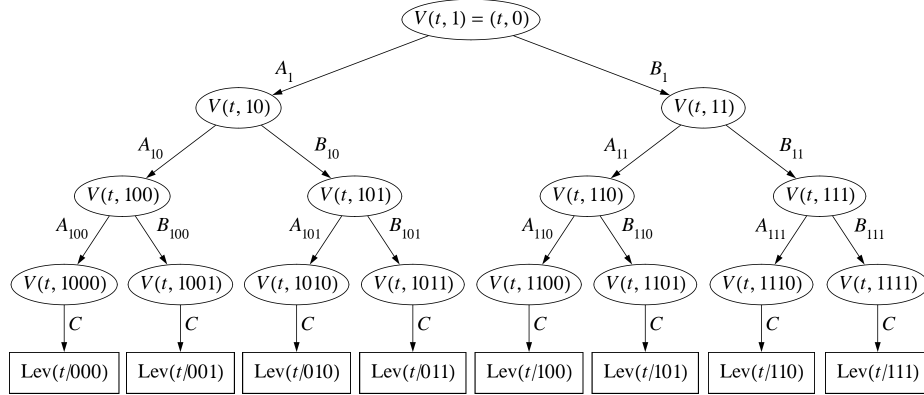

Fig. 1. Computation of an entire output tree of 8 words with $h = 3$. In the full cipher, $h = 16$ and the complete output is built from 2^{32} such trees.

The notation we use to specify this function precisely is somewhat different from that given in [5], but is convenient for our purposes; we treat z as a parameter, rather than as a word of state. The cipher is parameterised on n and h, where n is divisible by 4 and $n \geq h$; LEVIATHAN sets $n = 32$ and $h = 16$. $|$ denotes catenation of bit strings, $\overline{x}$ bitwise complementation of x, $\oplus$ the XOR operation (addition in Z_2^n or $Z_2^{n/4}$ as appropriate), and $+$ addition in the group Z_{2^n}, treating the first bit of the bitstring as the most significant and padding bitstrings shorter than n bits with zeroes on the left. We specify the forest structure illustrated in Figure 1 recursively:

$$
\begin{aligned}
\text{Lev} : \{0,1\}^{n+h} &\mapsto \{0,1\}^n \\
\text{Lev}(t|z) &= C(V(t,1|z)) \qquad (|t| = n, |z| = h) \\
V(t,1) &= (t,0) \\
V(t,z|0) &= A_z(V(t,z)) \\
V(t,z|1) &= B_z(V(t,z))
\end{aligned}
$$

The internal state that functions A, B, and C operate on (and the functions D, F, G used to define them) is a 2-tuple of bitstrings (x, y); we treat this as distinct from the catenated bitstring $x|y$. The functions L, R, and S operate on bytes within a word: L and R are rotates, while S provides nonlinearity with the key-dependent permutations $S_{0...3}$ which map $\{0,1\}^{n/4}$ onto itself. These

permutations are generated by the key schedule, which we omit. Note that F and G operate on each word of the tuple independently; mixing is provided by D.

$$\begin{aligned}
C(x,y) &= x \oplus y \\
A_z &= F \circ D_z \\
B_z &= G \circ D_z \\
D_z(x,y) &= (2x+y+2z, x+y+z) \\
F(x,y) &= (L(S(L(S(x)))), S(R(S(R(y))))) \\
G(x,y) &= (S(R(S(R(\overline{x})))), L(S(L(S(y))))) \\
L(x_3|x_2|x_1|x_0) &= x_2|x_1|x_0|x_3 \qquad (|x_3| = |x_2| = |x_1| = |x_0| = n/4) \\
R(x_3|x_2|x_1|x_0) &= x_0|x_3|x_2|x_1 \\
S(x_3|x_2|x_1|x_0) &= x_3 \oplus S_3(x_0)|x_2 \oplus S_2(x_0)|x_1 \oplus S_1(x_0)|S_0(x_0)
\end{aligned}$$

[5] gives a functionally different definition of D ($D_z(x,y) = (2x+y+z, x+y+z)$); the one given here is that intended by the authors [4] and used to generate the test vectors, though the difference is not relevant for our analysis.

We present two biases in the LEVIATHAN keystream that distinguish it from a random bit string. We know of no other attacks against LEVIATHAN more efficient than brute force.

2 PRF-PRF Attack

Both attacks focus on consecutive pairs of outputs generated by $\mathrm{LevPair}(i) = (\mathrm{Lev}(i|0), \mathrm{Lev}(i|1))$. Clearly, LevPair generates the same 2^{50}-byte keystream as Lev, so a distinguisher for one is a distinguisher for the other. Such pairs are interesting because they are the most closely related outputs in the tree structure; [5] refers to attacks using such pairs as “up-and-down attacks”. We can expand the formula for LevPair as follows:

$$\begin{aligned}
\mathrm{LevPair}(t|z) &= (\mathrm{Lev}(t|z|0), \mathrm{Lev}(t|z|1)) \\
&= (C(V(t,1|z|0)), C(V(t,1|z|1))) \\
&= (C(F(D_{1|z}(V(t,1|z)))), C(G(D_{1|z}(V(t,1|z)))))
\end{aligned}$$

From this we define functions LevAbove which generates the last common ancestor of such an output pair as illustrated in Figure 2, and PairCom which generates the output pair from the ancestor:

$$\begin{aligned}
\mathrm{LevAbove}(t|z) &= D_{1|z}(V(t,1|z)) \qquad (|z| = h-1) \\
\mathrm{PairCom}(x,y) &= (C(F(x,y)), C(G(x,y)))
\end{aligned}$$

from which we can see LevPair = PairCom ∘ LevAbove as stated. We model LevAbove as a random function throughout, and focus on the properties of PairCom.

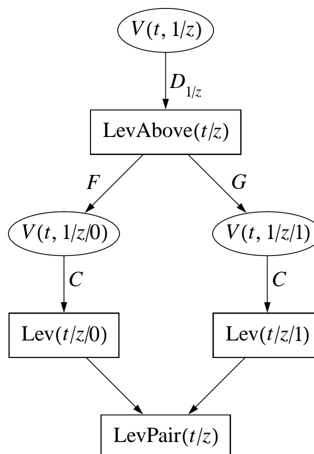

Fig. 2. Final stage of LevPair output; LevAbove finds the last common ancestor of the pair.

This structure gives us our first distinguisher. Though PairCom has the same domain as range, it is not in general bijective; it can be modelled more accurately as a random function. Thus a collision can occur in LevPair, given two distinct inputs, if there is a collision either in LevAbove or in PairCom, and if we model both as random functions the probability of an output collision for two random distinct inputs to LevPair is thus approximately $2^{-2n} + (1 - 2^{-2n})2^{-2n} \approx 2^{1-2n}$, twice what it should be if the keystream were a random binary string.

For $n = 32$, this increased probability of collisions between output word pairs can be observed with a birthday attack after around 2^{33} output pairs (2^{36} bytes) have been generated; the techniques of [8] may be used to reduce the memory demands of this attack, though this slows the attack by a factor of approximately $(h+1)/4 = 4.25$ where h is the height of the tree, since probes can no longer take advantage of the higher efficiency of sampling consecutive values of LevPair.

3 S-Box Matching Attack

The definitions of the F and G functions are very similar; G is the same as F except that it treats its inputs in the opposite order, and inverts one of them. If G did not apply bitwise inversion to its first input (call this function G'), then the two functions would be related by $F \circ \text{Swap} = \text{Swap} \circ G'$ (with Swap having the obvious definition $\text{Swap}(x, y) = (y, x)$); this would mean in turn that $F(a, a) = \text{Swap}(G'(a, a))$ for any a, and thus that $C(F(a, a)) = C(G'(a, a))$, with the result, as we shall see, that repeating pairs were visible in the output roughly twice as often as they should be. The inversion on the first input of G breaks this symmetry; however, it turns out that it does not prevent a related attack.

Computation of PairCom requires 32 S-box lookups, but for each computation of the S function the same 8-bit index, drawn from the least significant byte, is fed to each of the four S-boxes. Changes to the other bytes carry directly into the output of S, without nonlinearity or mixing; in other words, where $\Delta x = \Delta x_3|\Delta x_2|\Delta x_1|0^{n/4}$, we find $S(x \oplus \Delta x) = S(x) \oplus \Delta x$. We call this least significant byte the *index* to the S-box. If (x, y) is the input to PairCom, only bytes x_3, x_0 of x are indices to S-boxes in F, and only bytes x_2, x_1 are indices in G; by inverting only these two bytes in our pair (a, a), we can avoid the symmetry-breaking effect of the inversion as far as the nonlinear components are concerned, which results in the same four S-box indices being used in both the F and G branches of PairCom.

Figure 3 illustrates this attack. For an arbitrary n-bit string $a = a_3|a_2|a_1|a_0$, we define symbols for intermediate values in $F(a, a)$:

$$\begin{aligned}
b_3|b_2|b_1|b_0 &= S(a_3|a_2|a_1|a_0)\\
b'_0|b'_3|b'_2|b'_1 &= S(a_0|a_3|a_2|a_1)\\
c_2|c_1|c_0|c_3 &= S(b_2|b_1|b_0|b_3)\\
c'_1|c'_0|c'_3|c'_2 &= S(b'_1|b'_0|b'_3|b'_2)\\
d_3|d_2|d_1|d_0 &= (c_3|c_2|c_1|c_0) \oplus (c'_3|c'_2|c'_1|c'_0)
\end{aligned}$$

With these definitions, we find that $\mathrm{PairCom}(a_3|\overline{a_2}|\overline{a_1}|a_0,\ a_3|a_2|a_1|a_0) = (\overline{d_1}|d_0|d_3|\overline{d_2},\ d_1|\overline{d_0}|\overline{d_3}|d_2)$:

$$\begin{aligned}
C(F(x,y)) &= C(L(S(L(S(x)))),\ S(R(S(R(y)))))\\
&= C(L(S(L(S(a_3|\overline{a_2}|\overline{a_1}|a_0)))),\ S(R(S(R(a_3|a_2|a_1|a_0)))))\\
&= C(L(S(L(b_3|\overline{b_2}|\overline{b_1}|b_0))),\ S(R(S(a_0|a_3|a_2|a_1))))\\
&= C(L(S(\overline{b_2}|\overline{b_1}|b_0|b_3)),\ S(R(b'_0|b'_3|b'_2|b'_1)))\\
&= C(L(\overline{c_2}|\overline{c_1}|c_0|c_3),\ S(b'_1|b'_0|b'_3|b'_2))\\
&= C(\overline{c_1}|c_0|c_3|\overline{c_2},\ c'_1|c'_0|c'_3|c'_2)\\
&= \overline{d_1}|d_0|d_3|\overline{d_2}\\
C(G(x,y)) &= C(S(R(S(R(\overline{x})))),\ L(S(L(S(y)))))\\
&= C(S(R(S(R(\overline{a_3}|a_2|a_1|\overline{a_0})))),\ L(S(L(S(a_3|a_2|a_1|a_0)))))\\
&= C(S(R(S(\overline{a_0}|\overline{a_3}|a_2|a_1))),\ L(S(L(b_3|b_2|b_1|b_0))))\\
&= C(S(R(\overline{b'_0}|\overline{b'_3}|b'_2|b'_1)),\ L(S(b_2|b_1|b_0|b_3)))\\
&= C(S(b'_1|\overline{b'_0}|\overline{b'_3}|b'_2),\ L(c_2|c_1|c_0|c_3))\\
&= C(c'_1|\overline{c'_0}|\overline{c'_3}|c'_2,\ c_1|c_0|c_3|c_2)\\
&= d_1|\overline{d_0}|\overline{d_3}|d_2
\end{aligned}$$

From this it is clear that for any input of the appropriate form, one output word is the inverse of the other; or in other words, if we now XOR the two word outputs from PairCom together (which, conveniently, is the same as applying the LEVIATHAN compression function C a second time), we find

$$C(\mathrm{PairCom}(a_3|\overline{a_2}|\overline{a_1}|a_0,\ a_3|a_2|a_1|a_0)) = \overline{d_1}|d_0|d_3|\overline{d_2} \oplus d_1|\overline{d_0}|\overline{d_3}|d_2 = 1^n$$

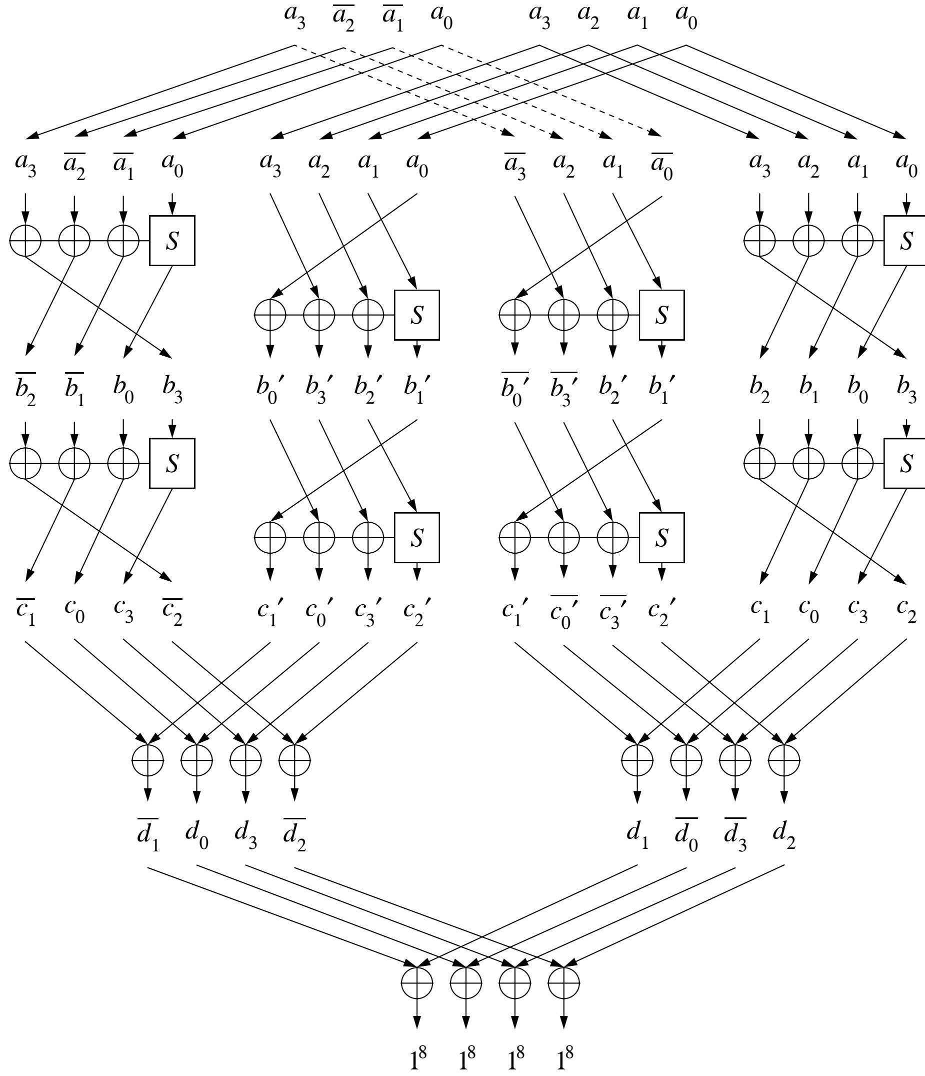

Fig. 3. S-box matching input to $C \circ \mathrm{PairCom}$. The function F is on the left, G on the right, and C underneath; dotted lines indicate bitwise inversion (the first step of the G function) and the "$\oplus \oplus \oplus S$" symbol represents the function $S(x_3|x_2|x_1|x_0) = x_3 \oplus S_3(x_0)|x_2 \oplus S_2(x_0)|x_1 \oplus S_1(x_0)|S_0(x_0)$.

for *all* values of $a_{3...0}$.

Since we model LevAbove as a random function we conclude that inputs to PairCom have probability 2^{-n} of matching this form in the normal calculation of LevPair. Where inputs do not match this form, we assume that PairCom behaves as a random function and therefore that for random (x, y) not matching this form, $\Pr(C(\text{PairCom}(x, y)) = 1^n) = 2^{-n}$; this assumption is borne out by experiment. From this we conclude that LevPair is twice as likely as a random function to produce an output (x_0, x_1) such that $C(x_0, x_1) = 1^n$

$$\Pr(C(\text{LevPair}(t|z)) = 1^n) = 2^{-n} + (1 - 2^{-n})2^{-n} \approx 2^{1-n}$$

which in turn implies that 64-bit aligned segments of keystream of this form are twice as frequent as they should be, yielding another distinguisher.

For $n = 32$, a test for the presence of this bias should therefore take on the order of 2^{33} samples of LevPair, ie 2^{36} bytes, as for the previous attack.

4 Experiments

We looked for these biases on a reduced version of LEVIATHAN with $n = 16, h = 16$.

For the PRF-PRF attack, we ran over 256 distinct keys generating $N = 6291456$ 32-bit LevPair outputs for each, and sorting them to find collisions. We count as a collision each instance where a distinct pair of inputs result in the same output; thus, where $m > 2$ outputs have the same value, we count this as $m(m-1)/2$ distinct collisions. For a random function we would expect to find approximately[1] $256(N(N-1)/2)/2^{2n} \approx 1179678$ collisions in total across all keys, while the PRF-PRF attack would predict an expected $256(N(N-1)/2)/2^{2n-1} \approx 2359296$. The experiment found 2350336 collisions; this is 1077.9 standard deviations (SDs) from the expected value in the random function model, and 5.83 SDs from the expected value in the model provided by the PRF-PRF attack. This shows that this model identifies a substantial bias in the cipher, but there is a further bias in the collision probability of roughly 0.38% yet to be accounted for.

For the S-box matching attack, we generated $N = 16777216$ LevPair outputs for each of 256 keys, counting outputs with the $C(x, y) = 1^{32}$ property. A random function would generate an expected $256N/2^{16} = 65536$ such outputs, while the S-box matching attack predicts that LevPair will generate an expected $256N/2^{15} = 131072$ such outputs. The experiment found 135872 such outputs; this is 274.8 SDs from the expected value in the random function model, and 13.26 SDs from the expected value in the model provided by the S-box matching

[1] The approximation $E(|\{\{x, y\} : f(x) = f(y)\}|) \approx |A|(|A| - 1)/2|B|$ for the number of collisions in a random function $f : A \mapsto B$ is very precise where $|B|$ is large; where we refer to the predictions of the random function model, it is the model with this approximation.

attack. Again, this shows that while a substantial source of bias has been identified, there is still a bias of 3.66% yet to be accounted for. Scott Fluhrer has reported finding this attack effective in experiments against the full LEVIATHAN with $n = 32, h = 16$.

5 Conclusions

We have shown two forms of bias in the output of the LEVIATHAN keystream generator, either of which distinguish it from a random function with 2^{36} known bytes of output; we have not as yet found a way to recover key material using these distinguishers. These distinguishers can both be applied to the same portion of keystream for greater statistical significance. Both make use of compression in the PairCom function.

Despite these attacks, LEVIATHAN demonstrates that a tree-based cipher could offer many advantages. It is to be hoped that similar designs, offering the same speed and flexibility but resistant to this and other attacks, will be forthcoming.

Acknowledgements

Thanks to Rüdiger Weis for helpful commentary and suggestions, and to the LEVIATHAN authors for providing an implementation of the first experiment and for useful discussion.

References

1. Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, 1986.
2. IP security protocol (ipsec). http://www.ietf.org/html.charters/ipsec-charter.html.
3. Helger Lipmaa, Philip Rogaway, and David Wagner. Comments to NIST concerning AES modes of operation: CTR-mode encryption, 2000.
4. David A. McGrew. Re: Possible problems with leviathan? Personal email, November 2000.
5. David A. McGrew and Scott R. Fluhrer. The stream cipher LEVIATHAN. NESSIE project submission, October 2000.
6. NESSIE: New European schemes for signatures, integrity, and encryption. http://www.cryptonessie.org/.
7. Phillip Rogaway and Don Coppersmith. A software-optimized encryption algorithm. In Ross Anderson, editor, *Fast Software Encryption*, pages 56–63. Springer-Verlag, 1994.
8. Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *Journal of Cryptology*, 12(1):1–28, 1999.
9. David Wheeler. A bulk data encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop*, volume 1008 of *Lecture Notes in Computer Science*, Leuven, Belgium, 14–16 December 1994. Springer-Verlag. Published 1995.

URL for this paper: **http://www.ciphergoth.org/leviathan**