Cryptanalysis of the SEAL 3.0 Pseudorandom Function Family

Scott R. Fluhrer

Cisco Systems, Inc.
170 West Tasman Drive, San Jose, CA 95134
sfluhrer@cisco.com

Abstract. We present an attack on the SEAL Pseudorandom Function Family that is able to efficiently distinguish it from a truly random function with 2^{43} bytes output. While this is not a practical attack on any use of SEAL, it does demonstrate that SEAL does not achieve its design goals.

1 Introduction

SEAL is a series of encryption algorithms designed by Phillip Rogaway and Don Coppersmith. The most recent version is SEAL 3.0, which is described in [6]. In this paper, all references to SEAL without an explicit version number are to version 3.0.

SEAL is designed to be a cryptographic object called a pseudorandom function family, which were defined in [2]. This is an object that, under the control of a key, maps a fixed length input string to an output string in a manner that appears random. SEAL, in particular, has a 160 bit key, and maps a 32 bit input into a 64kbyte output.

Rogaway and Coppersmith gave as an explicit design goal that it would be computationally infeasible to distinguish (with probability significantly greater than 0.5) SEAL with a random fixed key from a truly random function. This paper shows a truncated differential attack that is able to distinguish it, given 2^{43} bytes of output and $O(2^{43})$ work. We note that, since SEAL with a fixed key defines only 2^{48} bytes of output, this attack requires relatively close to the total amount of data that an attacker can conceivably examine with a single key.

This paper is structured as follows. In Section 2, the SEAL pseudorandom function family is described, and previous analysis are summarized. Section 3 lists some key observations about the internals of the SEAL update function, and sections 4 and 5 provide the actual attack. Section 6 presents results of this attack on weakened variants of SEAL, and section 7 discusses what aspects of SEAL allowed this attack to be successful, and summarizes our conclusions.

2 Description of SEAL and Other Work

SEAL is a length increasing pseudorandom function family that, under the control of a 160-bit key, expands a 32-bit string into a 2^{19}-bit string. Internally, SEAL expands the 160-bit keys into 3 secret tables R, S, T, which contain respectively 256, 256, and 512 32-bit values. These tables are fixed once the key has been defined.

To generate 2^{13} bits of output, SEAL takes the 32-bit input string n and 6 bits of submessage index[1] ℓ and expands it into 256 bits of state (the 32 bit A, B, C, D variables and the 32 bit n_1, n_2, n_3, n_4 variables using the R and T tables). Then, SEAL steps through a 6 bit block index[2] i, and alternates between two slightly different noninvertible update functions to update A, B, C, D using the T table. At one point during the update, outputs A, B, C, D combined with the 4 32-bit elements from the S table indexed by i. SEAL iterates through this 64 times to generate 2^{13} bits. To generate the full 2^{19} bits, SEAL steps through all 2^6 submessage indices, and concatenates the results.

For the reader's convenience, the pseudocode for the update function is copied from [6] and is listed as Table 1. In this code, α denotes the key (which is implicitly used to derive the secret R, S, T tables), n is the 32 bit input string, L is the number of output bits desired, λ denotes the empty string, *Initialize* is a function that sets A, B, C, D, n_1, n_2, n_3, n_4 to key, message, and submessage index dependent 32-bit values, & and $\oplus$ denote bit-wise *and* and $exclusive-or$, $+$ denotes addition modulo 2^{32}, $\rangle\rangle\rangle$ denotes a right circular shift, and $\|$ denotes concatenation.

This attack ignores the key expansion and the *Initialize* function, idealizing them as truly random processes, and so the details of how they are specified are not listed in this paper.

SEAL was originally published as version 1.0 in [5]. Version 2.0, modified to use the revised SHA-1 algorithm in the key expansion, appeared in [4]. Handschuh and Gilbert discovered an attack, described in [3], that is able to distinguish both SEAL 1.0 and SEAL 2.0 from a random function, and was also able to gain some information about the hidden table entries. In response to this attack, Rogaway and Coppersmith revised it in [6] to SEAL version 3.0, which is the subject of this paper. There are no other attacks described in the open literature.

3 Basics of P/Q Collisions

This attack is based on a specific truncated differential within the update function. This differential is between two executions of steps 1 through 8 of the next state function, and is defined to be when all of the following occur:

[1] The submessage index is defined to be the value indicating which 1kbyte section of the 64kbyte output SEAL is currently generating on behalf of a particular input string. This terminology is specific to this paper, and while it is represented by the variable ℓ in Figure 2 of [6], it is not given an explicit name.

[2] The block index is defined to be the value indicating which 16 byte section of the 1kbyte submessage SEAL is currently generating. This terminology is specific to this paper, and while it is represented by the variable i in Figure 2 of [6], it is not given an explicit name.

Table 1. Cipher mapping 32-bit position index n to L-bit string $SEAL(\alpha, n, L)$ under the control of α-derived tables T, R, S

```
   function SEAL($\alpha$, $n$, $L$)
   $y = \lambda$;
   for $\ell \leftarrow 0$ to $\infty$ do
       Initialize($n$,$\ell$, $A$,$B$,$C$,$D$, $n_1$,$n_2$,$n_3$,$n_4$ );
       for $i \leftarrow 1$ to 64 do
1          $P \leftarrow A\&0x7fc$;        $B \leftarrow B + T[P/4]; A \leftarrow A\rangle\rangle\rangle 9; B \leftarrow B \oplus A$;
2          $Q \leftarrow B\&0x7fc$;        $C \leftarrow C \oplus T[Q/4]; B \leftarrow B\rangle\rangle\rangle 9; C \leftarrow C + B$;
3          $P \leftarrow (P + C)\&0x7fc; D \leftarrow D + T[P/4]; C \leftarrow C\rangle\rangle\rangle 9; D \leftarrow D \oplus C$;
4          $Q \leftarrow (Q + D)\&0x7fc; A \leftarrow A \oplus T[Q/4]; D \leftarrow D\rangle\rangle\rangle 9; A \leftarrow A + D$;
5          $P \leftarrow (P + A)\&0x7fc; B \leftarrow B \oplus T[P/4]; A \leftarrow A\rangle\rangle\rangle 9$;
6          $Q \leftarrow (Q + B)\&0x7fc; C \leftarrow C + T[Q/4]; B \leftarrow B\rangle\rangle\rangle 9$;
7          $P \leftarrow (P + C)\&0x7fc; D \leftarrow D \oplus T[P/4]; C \leftarrow C\rangle\rangle\rangle 9$;
8          $Q \leftarrow (Q + D)\&0x7fc; A \leftarrow A + T[Q/4]; D \leftarrow D\rangle\rangle\rangle 9$;
9          $y \leftarrow y \parallel B + S[4i-4] \parallel C \oplus S[4i-3] \parallel D + S[4i-2] \parallel A \oplus S[4i-1]$;
10         if $|y| \geq L$ then return $(y_0 y_1 \ldots y_{L-1})$;
11         if $odd(i)$ then $(A, B, C, D) \leftarrow (A + n_1, B + n_2, C \oplus n_1, D \oplus n_2)$
                    else $(A, B, C, D) \leftarrow (A + n_3, B + n_4, C \oplus n_3, D \oplus n_4)$
```

1. The P and Q internal variables have zero differential at all points through steps 1 through 8.
2. Both sides of the differential have the same block index (i.e. i has zero differential).
3. The common block index is not the first (i.e. $i \neq 1$).

We will call an occurrence of this differential a *P/Q collision.* We will see that an occurrence of a P/Q collision can be seen in the differential output states both before and after the update, and will use that to provide the distinguisher.

We will use the syntax x_n to indicate the differential in variable X at the start of the execution of line[3] n. We will also use the syntax x_{output} to represent the differential in the output in line 9 that the variable X contributes to (including the contribution of the S table). In addition, we will also use the convention that bit 0 is the least significant bit, and bit 31 is the most significant in a 32 bit word.

Consider two separate iterations of lines 1-8 of the update function where a P/Q collision occurs. Then, because we know that p_2 has zero differential (by the definition of a P/Q collision) we know that bits 10-2 of a_1 must have zero differential, which implies that bits 31-25 and 1-0 of a_2 must have zero differential. And, we know, since both p_5 and p_6 have zero differential, that bits 10-2 of a_5 must also have zero differential. Since bits 10-2 of d_4 must have zero differential (because of step 4) and bits 31-25 and 1-0 of d_4 must have zero differential (because of step 8), then bits 10-0 of a_4 must have zero differential.

[3] All references to lines refer to the numbered lines of code in Table 1.

Then, along with step 2, we see that, because the carries into bit 2 of the sum $B+T[P/4]$ in line 1 may be different[4], there is an additive differential within the set {0, +1, -1} (with probabilities 1/2, 1/4, 1/4 respectively[5]. We will shorthand this relationship by saying that bits 10-2 of b_1 has a differential of ϵ. In addition, we will shorthand a differential that represents the sum of two such independent additive differentials (in other words, has an additive differential within the set {0, +1, +2, -1, -2} with probabilities 3/8, 1/4, 1/16, 1/4, 1/16 respectively) as 2ϵ.

We can continue this argument to find the differentials before step 1 and after step 8. We see that (after step 8) bits 1-0 and 31-16 of b_9 and d_9 have zero differential, those bits in c_9 has an ϵ differential, and those bits in a_9 has a 2ϵ differential. Now, both halves of the differential have the same block index (again, by the definition of a P/Q collision), and so the S table entries used at step 9 have zero differential. Since bits 1-0 and 31-17 of C has zero differential with probability 3/4, and A has zero differential in those same bits with probability 5/8, the corresponding bits output in step 9 will have zero differential with that same probability. In addition, the differential in B and D is isolated to bits 15-2, and so after step 9, the corresponding outputs in step 9 will have an additive differential that consists of either a number that is zero everywhere except in bits 15-2, or the negative of such a number[6]. We will shorthand such a differential as a differential of δ[7].

Summarizing the above, and listing the differentials found before step 1, we find the differentials which are listed in Table 2. Note that, in addition to the listed differentials, a_{output} has a zero differential in bits 1-0,31-16 with probability 3/8, and that c_{output} has a zero differential in bits 1-0,31-16 with probability 1/2. However, it turns out that using the differentials listed makes the attack somewhat more efficient.

Since P and Q are effectively 9 bit variables, and are updated 8 times with independent values, a P/Q collision has a 2^{-72} probability of occurring for two particular iterations with the same block index.

4 Searching for P/Q Collisions

To prosecute the attack, we must recognize with nontrivial probability when a P/Q collision has occurred. To do that, we search the keystream for places that resemble the differentials that occur immediately after a P/Q collision.

[4] Throughout this analysis, we will assume that the internal carries that occur within various additions will be unbiased and independently distributed.

[5] Because the carry from a sum of two random bits is biased towards zero, the additive differential will be somewhat more based towards 0 than stated. This effect turns out to make our attack more effective than expected, and so it can be safely ignored for our purposes.

[6] This number is biased towards zero, but this attack does not currently attempt to take advantage of that.

[7] Unlike other differentials discussed here, we consider a δ differential to be across the entire 32 bit variable.

Table 2. Known differentials before step 1 and at output at step 9 during a P/Q collision

Variable	Bits	Differential
a_1	19-2	Zero
b_1	10-2	ϵ
c_1	10-2	ϵ
d_1	10-2	ϵ
a_{output}	1-0,31-17	Zero with probability 5/8
b_{output}		δ
c_{output}	1-0,31-17	Zero with probability 3/4
d_{output}		δ

In a P/Q collision, Table 2 shows us that there are 34 output bits in a_{output} and c_{output} that agree with probability $5/8 \times 3/4 = 15/32$. In addition, the additive differential in b_{output} and d_{output} must both be one of $2^{15} - 1$ values. We will term any pair of outputs that meet the output criteria in Table 2, and which occur for the same nonfirst block index as a *potential P/Q collision.*

Assume that we have 2^{43} bytes of SEAL output that consist of the first two block indices from the entire SEAL output stream (that is, block indices 1 and 2 for all possible input strings and submessage indices). Then, we search through the outputs with block index 2 for pairs of output blocks that have the above properties. There are 2^{38} available output blocks with block index 2, and by the birthday paradox, and assuming that the internal states of the Ps and Qs are effectively random, we will have an expected $2^{2\times 38-72-1} = 8$ P/Q collisions, of which an expected $8 \times 15/32 \approx 4$ of which will be detectable in this manner.

We will assume that, if the internal state does not correspond to a P/Q collision, those output bits will match approximately as often as they will in a random sequence[8]. Then, we expect that an arbitrary pair will match if 17 bits within A and within C match, and the differences between B and D will be $2^{15}-1$ possible values (out of a total of 2^{32} possible values). Hence, the probability for an arbitrary pair of matching will be $2^{-17} \times 2^{-17} \times (2^{15} - 1)/2^{32} \times (2^{15} - 1)/2^{32} \approx 2^{-68}$. Hence, we expect approximately $2^{2\times 38-1-68} = 128$ false hits, that is, potential P/Q collisions that do not correspond to actual P/Q collisions.

The above analysis assumed that we have the first 32 bytes from every possible 1024 byte submessage. If we assume instead that we have a contiguous section of SEAL output (for example, the entire 64kbyte expansion of a subset of the input strings), then this attack can still be prosecuted, but with less efficiency, requiring 2^{45} bytes of output. The problem is that P/Q collisions are only possible within the same block, and hence increasing the number of block indices decreases the efficiency of the birthday paradox. There are 2^6 possible block indices (one of which is useless for searching for P/Q collisions), and with

[8] If this assumption is not valid, then potential P/Q collisions will be either much more numerous or much less numerous than expected, which can itself be used as a distinguisher.

2^{41} output blocks, we have 2^{35} output blocks for each index. Hence, each block index is expected to have $2^{2\times 35-72-1} = 1/8$ P/Q collisions, of which an expected $1/8 \times 15/32 \approx 1/16$ will be detectable. In addition, each block index is expected to have $2^{2\times 35-68-1} = 2$ false hits. Hence, we expect $1/16 \times 63 \approx 4$ true hits throughout the stream, and an expected $2 \times 63 = 126$ false hits. We can then continue to prosecute the attack as shown in section 5.

5 Verifying P/Q Collisions

Now that we have identified potential places where a P/Q collision may have occurred, we then examine the differential in the output states that occur at the corresponding block 1 to see if they resemble the output state immediately before a P/Q collision. If we look at the state of the cipher immediately before a P/Q collision, we see in Table 2 that bits 10-2 of a_1 have zero differential[9], and bits 10-2 of b_1, c_1, d_1 has an ϵ differential. Stepping back through steps 11 and 9 of the previous iteration, and canceling out the effects of the S array[10] by exclusive-oring the A and C outputs and subtracting the B and D outputs, the attacker observes[11]:

$$O_A = A_{Output} \oplus \hat{A_{Output}} = (A_1 - n_1) \oplus (\hat{A_1} - \hat{n_1}) \tag{1}$$

$$O_C = C_{Output} \oplus \hat{C_{Output}} = (C_1 \oplus n_1) \oplus (\hat{C_1} \oplus \hat{n_1}) \tag{2}$$

$$O_B = B_{Output} - \hat{B_{Output}} = (B_1 - n_2) - (\hat{B_1} - \hat{n_2}) \tag{3}$$

$$O_D = D_{Output} - \hat{D_{Output}} = (D_1 \oplus n_2) - (\hat{D_1} \oplus \hat{n_2}) \tag{4}$$

where O_A, O_B, O_C, O_D are defined as above, X_1, $\hat{X_1}$ are the contents of the X variable in the two sides of the differential immediately after step 11, n_1, n_2, $\hat{n_1}$, $\hat{n_2}$ are the values used in step 11 on the two sides, and X_{Output}, $\hat{X_{Output}}$ are the values corresponding to the variable X output by the two sides at step 9.

By assuming that the various unknown quantities above are independently and uniformly distributed, we can compute the expected distribution of O_A, O_B, O_C, O_D. When we do so, we find that the distribution is strongly skewed from uniform. For example, the output is within a subset that is only 1.113% of the size of the entire distribution 50% of the time. The most probable pattern (bits 10-2 of O_A, O_B, O_C, O_D are all zero) occurs with a probability 25,000 times the expected rate. In addition, nearly 37% of the possible O_A, O_B, O_C, O_D values are impossible for any value of unknown quantities.

[9] As do bits 19-11, but those bits cannot be used in our attack.

[10] We note again that both sides of the differential use the same block index, and so the S array contents here again have zero differential.

[11] Depending on whether the block index is even or odd, equations (1)-(4) may actually depend on n_3 and n_4 rather than n_1 and n_2. This attack does not depend on the actual identity of the variables, and so by convention we will call those variables n_1 and n_2.

This happens because exclusive-or and addition/subtraction are similar operations, and produce related results on a bit level more often than random operations would. For example, if you examine O_A and O_C, and ignore the effects of ϵ_1 and ϵ_3 (which, with good probability, affects only the lower order bits), then the equations (1) and (2) reduce to (over bits 10-0):

$$O_A \approx (X - n_1) \oplus (X - \hat{n_1}) \tag{5}$$

$$O_C \approx n_1 \oplus \hat{n_1} \tag{6}$$

where the approximation means that each bit has a probability of matching the corresponding bit of the other side, and the agreement probability is best at bit 10 and lesser as you go down. Now, consider the case where several consecutive bits of O_C are zero. This implies that the corresponding bits of n_1 and $\hat{n_1}$ agree with good probability. This, in turn, implies that the higher order bits of the corresponding section of $X - n_1$ and $X - \hat{n_1}$ will also agree with good probability, and hence a sequence of consecutive bits of O_C being zero tend to imply that the higher order bits of the corresponding sequence of O_A will also be zero.

There are similar, but weaker, relationships between O_B and O_D that can also be exploited. Rather than explicitly listing all these relationships, we can take advantage of this by computing the probability distribution that is formed by a source that generates the distribution expected by (O_A, O_B, O_C, O_D) with probability 4/128, and an equidistributed pattern with probability 124/128 (which we have shown is what is observed with SEAL), and compare that with the probability distribution of a source that generates an equidistributed pattern all the time (which is what a truly random source produces). If we use a Neyman-Pearson test ([1]), we are able to distinguish the two sources with 90% certainty with approximately 110 outputs. And, since the examined SEAL output is expected to have at least that many detected outputs, we can use that distinguisher to distinguish SEAL from a random source.

In an actual implementation of this attack, the bulk of the work (searching for potential P/Q collisions) can be done by sorting the adjacent output blocks on the bits that are known to match (bits 1-0,31-17 of a_{output}, c_{output} of the second output block, and block index), and then b_{output} value of the second output block. This sort will bring the potential P/Q collisions near each other, and so one sequential scan can find them. Once the potential P/Q collisions have been found, the computations required to verify the collisions are comparatively trivial. Hence, the run time is dominated by the time to do the sort, which is $O(2^{38} \log 2^{38}) \approx O(2^{43})$.

6 Experimental Verification on Weaked SEAL Variants

To be able to verify experimentally with the available computer resources that this attack is valid, we weakened SEAL by creating a variant, which we will refer to as wSEAL/5. This is defined by modifying each T table lookup and circular rotates in the next state function to use only 5 bits, rather than 9. This change was done specifically to allow the attack to be run on the available hardware, while attempting to retain the essential aspects of the cipher.

The distinguisher translates in a straightforward manner to wSEAL/5. The only nonobvious change is that, when we recomputed the (O_A, O_B, O_C, O_D) distributions for these variants, they were considerably more uniform than the distribution for SEAL. This occurs because the bias in the distribution was due to relationships between the higher order bits, and when we reduce the size of the T table lookups, we also reduce the number of higher order bits that are available for verification. Because of the increased uniformity, it turns out that we require 800 potential P/Q collisions to achieve the same confidence level for wSEAL/5.

When we analyze these variants to determine if the assumptions that we made during our analysis is valid, we find:

- P/Q collisions are produced at the predicted rate. For example, when searching through 64 separate sections of 2^{20} blocks of wSEAL/5 output, we found 30 P/Q collisions, when the analysis would predict that 32 collisions were expected.
- P/Q collisions are proceeded by the a_1, b_1, c_1, d_1 differentials listed in Table 2. This was verified by examining the differentials from the 30 P/Q collisions referenced above.
- P/Q collisions that are detectable using the criteria of section 4 were produced at least the expected rate. For example, when searching through the 30 P/Q collisions referenced above, we found that 21 of them would be detected by that criteria. The analysis predicted that 15 were expected.
- Potential P/Q appear in the output stream at the expected rate. For example, in the above output stream that produced 21 detectable P/Q collisions, a total of 529 potential P/Q collisions were detecteds. The analysis predicted that 527 were expected[12].
- The P/Q collisions within the output stream are verifiable using the criteria of section 5. In particular, wSEAL/5 could be reliably distinguished from RC4 output using this analysis. In 100 tests where the distinguisher was presented either wSEAL/5 or an unrelated function, it correctly identified the source 88 times, which is consistent with our 90% confidence criteria we used.

Because wSEAL/5 acts in a manner consistent with our analysis, it appears reasonable to suppose that this analysis also applies to the full SEAL cipher.

7 Conclusions

We have presented a method for distinguishing SEAL from a random keystream source with approximately 2^{43} output bytes (or alternatively, with 2^{45} consecutive output bytes). Our method is the only such method in the open literature that is able to do so.

[12] This is 512 false hits, and 15 actual detectable P/Q collisions.

The attack appears to work by taking advantage of several aspects of the SEAL next state function:

- The actual number of sbox inputs are relatively small compared to the number of bytes actually output. This fact is listed in [6] as one of the reasons for its efficiency, however, we were able to use this property to control inputs to the sboxes.
- In line 11 of the SEAL update function, two of the n_i variables are used twice. The attack was able to use this redundancy to be able to verify when a collision occurred, by essentially verifying that the same values were used to update the internal variables. Had SEAL used all four n_i variables to update the four internal variables, there would have been no way to verify a P/Q collision.
- Lines 1 through 8 has very good differential propagation in the forward direction, but comparatively poor propagation in the reverse direction. In particular, if you track the effect of a difference, the output of an sbox in one step is immediately fed into the sbox on the next step. However, if you assume a difference at one step, and trace the differential backwards by logically inverting the next state function, you often skip several steps before the effect of an sbox lookup is used. We suspect that this is one of the reasons why this differential happens to work, and that a different structure that had reasonably good differential properties in both directions would be rather more resistant to similar attacks.

It will require more research to determine whether this attack can be modified to require less keystream, and whether it can be extended to rederive the secret key or to gain information on entries in the R, S, T tables.

References

1. Blahut, R., "Principles and Practice of Information Theory", Addison-Wesley, 1983.
2. O. Goldreich, S. Goldwasser, S. Micali, "How to construct random functions", Journal of the ACM, Vol 33, No. 4, 1986, pp. 210-217
3. H. Handschuh, H. Gilbert, "χ^2 cryptanalysis of the SEAL encryption algorithm", Fast Software Encryption, Lecture Notes in Computer Science, Vol. 1267, Springer-Verlag, 1997, pp. 1-12
4. A. Menezes, P. van Oorschot, S. Vanstone, "Handbook of applied cryptography", CRC Press, 1997
5. P. Rogaway, D. Coppersmith, "A software-optimized encryption algorithm", Fast Software Encryption, Lecture Notes in Computer Science, Vol 809, Springer-Verlag, pp. 56-63.
6. P. Rogaway, D. Coppersmith, "A software-optimized encryption algorithm", Journal of Cryptography, Vol. 11, No. 4, 1998, pp. 273-287