PeerJ Preprints NOT PEER-REVIEWED

A Simple Encryption Algorithm

Joseph Keenan St. Pierre

Worcester Polytechnic Institute

Corresponding author:
Joseph Keenan St. Pierre

Email address: jkstpierre@wpi.edu

ABSTRACT

In this paper I present a Simple Encryption Algorithm (SEAL), by which 128-bit long blocks can be quickly encrypted/decrypted. The algorithm is designed to run efficiently in software without any specialized hardware while still guaranteeing a strong degree of confidentiality. The cipher is composed entirely of simple bit-wise operations, such as the exclusive-or and circular shift, in addition to modular addition, thereby making it exceedingly easy to implement in most programming languages as well as efficient in terms of performance.

INTRODUCTION

SEAL is a symmetric key block cipher that permutes a 128-bit block against a 128-bit key. Based off of statistical analysis, the cipher's output exhibits strong avalanche effects as well as uniform distribution. Furthermore, as the name suggests, SEAL's implementation in software is lightweight, straightforward, and requires minimal overhead.

ENCRYPTION PROCEDURE

Below is the encryption procedure for SEAL written in the C programming language:

```c
/*Substitute the bytes of a block chunk with the SEAL S-Box*/
void S(uint32_t *block_chunk){
   for(int i = 0; i < 4; i++){
      uint8_t byte = (uint8_t)(*block_chunk >> 8*i);
      *block_chunk = *block_chunk & ~(0xFF << 8*i);
      *block_chunk |= ((uint32_t)sbox[byte] << 8*i);
   }
}
/*Encrypts a 128-bit block against a 128-bit key using SEAL*/
void seal_encrypt(uint32_t *block, uint32_t *key){
  /*The carry chunk used for rotation and feeding*/
  uint32_t carry = 0;

  /*Perform 8 rounds of encryption*/
  for(int i = 0; i < 8; i++){
    //Substitute first chunk and XOR against key and round number
    S(&block[0]); block[0] ^= key[i%4] ^ i; carry = block[0];

    //Feed carry chunk through subsequent block chunks
    block[1] += carry; carry = block[1];
    carry = (carry >> 11) | (carry << 21); //Rotate carry chunk

    block[2] += carry; carry = block[2];
    carry = (carry >> 11) | (carry << 21);

    block[3] += carry; carry = block[3];
    carry = (carry >> 11) | (carry << 21);
```

PeerJ Preprints NOT PEER-REVIEWED

```
    //Rotate block chunks
    block[3] = block[2]; block[2] = block[1];
    block[1] = block[0]; block[0] = carry;
  }

  S(&block[0]); S(&block[1]); //Substitute chunks again
  S(&block[2]); S(&block[3]);

  for(int i = 0; i < 4; i++) //Perform "locking" round
    block[i] ^= key[i]; //XOR chunks against key
}
```

SUBSTITUTION BOX

For added reference, below is the 16x16 lookup table used by the S-box to substitute the bytes of a block chunk. This table was generated randomly using a brute-force algorithm for finding a substitution box with strong avalanche effect. Any lookup table will suffice provided it too has good bit distributions and a strong avalanche effect. A byte is substituted by taking its hexadecimal value, reading its first and second digit which correspond to a row and column respectively, and replacing said byte with the value located at the corresponding cell. For example, the hexadecimal value ”ab” would be substituted with ”67” according to the provided table.

Table 1. S-box

S	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	bc	7d	7b	93	01	15	30	21	a5	d6	f8	9b	48	db	ce	29
1	61	b3	34	03	b1	d7	53	98	52	f3	bb	ab	b2	2d	2a	eb
2	08	02	0c	cb	b0	9e	8f	96	40	92	e9	6e	58	6d	44	06
3	88	1c	1f	65	91	85	66	45	9d	4a	b9	8d	20	ca	a0	19
4	5d	5b	12	25	cc	9c	43	a2	50	da	b4	f9	4f	69	17	4b
5	04	6c	11	dd	73	16	df	41	3a	5f	74	47	09	18	fe	99
6	84	62	00	0d	64	7c	d5	72	e1	e5	24	ee	4d	f2	3d	2c
7	26	3b	42	3f	c2	7a	d3	1d	57	0b	fa	75	d0	c4	ec	fb
8	0e	e3	90	80	ff	0a	4e	2f	d9	2e	c8	e6	1b	94	55	9a
9	f5	60	79	d2	71	cf	dc	ad	f7	7f	c0	05	6b	af	38	7e
a	d8	35	70	c9	aa	83	a6	d1	39	e0	6a	67	a4	5a	13	8b
b	f0	fc	e7	c6	a3	97	c3	5e	c7	63	46	ba	37	ea	77	c1
c	cd	4c	33	bf	28	76	b5	b7	f4	ed	5c	fd	68	ae	e4	78
d	e2	1e	2b	ac	59	36	be	6f	1a	b6	9f	22	87	8c	10	31
e	82	a9	07	f6	86	23	e8	95	54	a8	82	8a	a1	49	f1	b8
f	d4	3e	0f	de	3c	8e	56	c5	27	89	14	bd	51	a7	32	ef

OVERVIEW AND DESIGN RATIONALE

The cipher is a form of substitution-permutation network whereby both the block and the key are divided into equal 32-bit chunks. Through the combined usage of modular additions, circular shifts, and the XOR, a high degree of non-linearity is achieved. Prior to the start of each round, the round-key is XOR’d against the round number; this is done in order to protect the cipher against slide attacks by removing the symmetry of the internal rounds. The substitution boxes are included in order to rapidly introduce the avalanche effect resulting from a minor change in the input block. Furthermore, to ensure that any change in the key results in a major change in the output, at least eight rounds of encryption are required; however, if additional security is needed, the number of rounds can be scaled up provided that the number of rounds is a multiple of four. This guarantees that each chunk of the key is used the same number of times thereby preventing any bias from emerging. Lastly, in order to obfuscate the internal permutations of the block, the entire block is substituted and XOR’d against the key in a final ”locking” round.

SEAL has been specifically designed for efficiency in software without the need for dedicated hardware. For this reason, there is no key schedule in the default implementation presented here as such an additional process unnecessarily slows the rate of encryption. This omission is justified on the basis that the statistical analysis of the cipher's output distribution and avalanche effect showed no conclusive advantage to using SEAL with a key schedule as opposed to using the same key repeatedly. That being said, if performance is not a concern, a key schedule could easily be implemented and used alongside SEAL if one so desired.

CIPHER DIAGRAM

Below is an overall outline of the SEAL cipher:

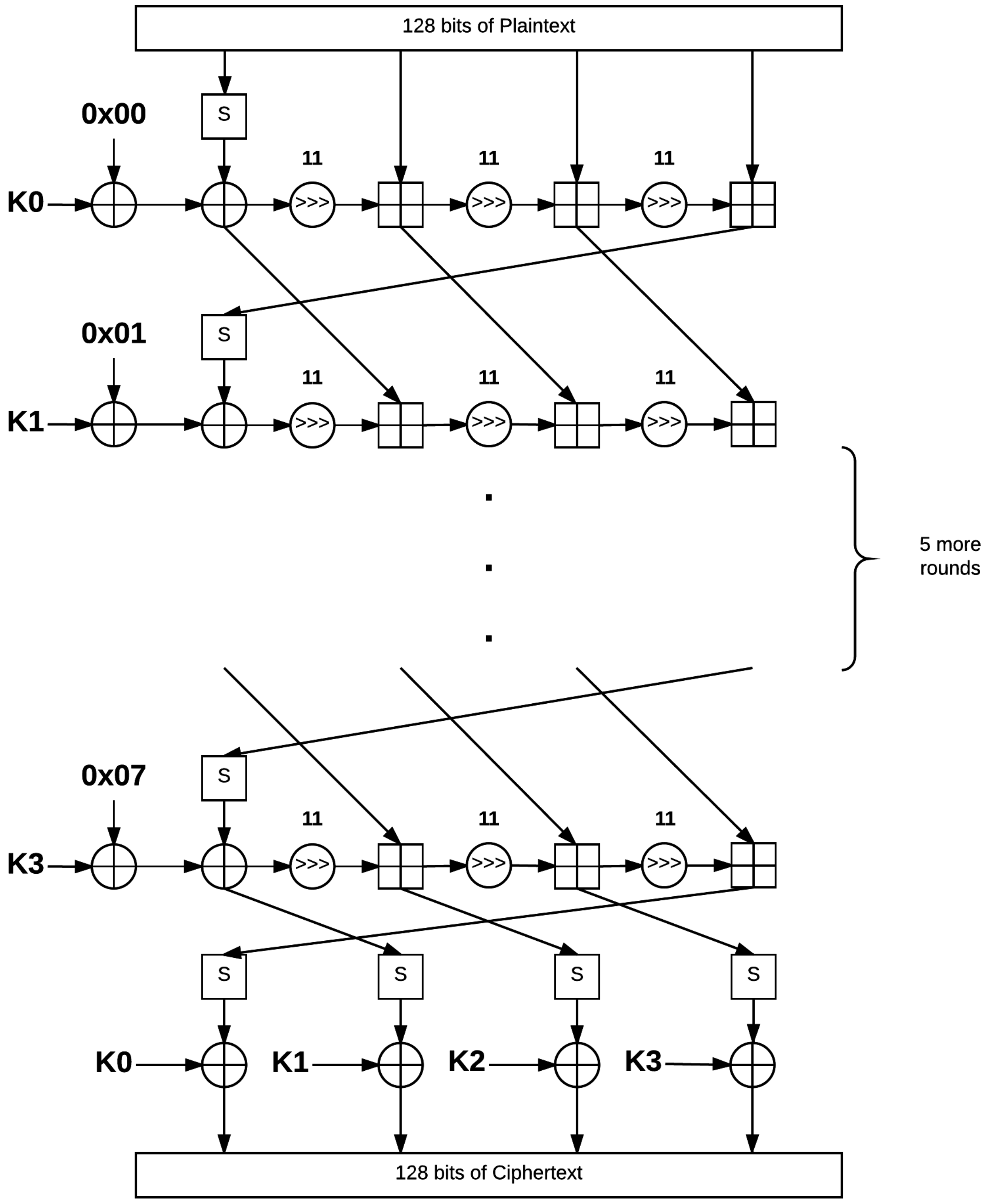

Figure 1. SEAL Cipher Digram

PeerJ Preprints NOT PEER-REVIEWED

ANALYSIS

Substitution Box

The primary function of an S-box is to remove the linear relationship between a plaintext-ciphertext pair produced by a cryptographic function. In order to effectively accomplish this task, it is imperative that an S-box does not have any strong differential characteristics that could be feasibly exploited by cryptanalysis.

The S-box articulated in this paper was produced via a brute-force algorithm that searched for a 16x16 lookup table that followed an avalanche criterion in addition to possessing only weak differential characteristics; as such, the best differential characteristic across the SEAL S-box, (e5 $\rightarrow$ 8f), only holds true with a probability of 0.023438.

While the S-boxes produced by this algorithm are probably not secure enough to resist an extensive differential attack on their own, it should be noted that when used in conjunction with other non-linear functions (such as modular addition), the cipher as a whole becomes infeasible to attack.

Internal Round Functions

SEAL's internal round functions in and of themselves are not secure as only the first chunk is technically kept a secret via the XOR. However, when used in a group of 4, the entire block is permuted and cannot easily be recovered despite the fact that intermediary computations can be trivially found if the ciphertext output is known. This is best articulated using a 1x4 matrix which represents the outputted permuted block, a 4x4 matrix which represents the states of the various XOR and ADD nodes for each internal round, and a final 1x4 matrix which represents the original inputted plaintext block. For demonstration purposes, the matrices below will only contain 8-bit values and the circular shifts will be by 3 bits instead of 11.

$$\overset{C}{\begin{bmatrix} ab \\ 16 \\ 99 \\ f2 \end{bmatrix}} \Leftarrow \overset{Round_{1-4}}{\begin{bmatrix} ab & a0 & c2 & 5c \\ 16 & d6 & e4 & xx \\ 99 & bf & xx & xx \\ f2 & xx & xx & xx \end{bmatrix}} \Leftarrow \overset{P}{\begin{bmatrix} xx \\ xx \\ xx \\ xx \end{bmatrix}}$$

Figure 2. After 4 internal rounds, the plaintext is impossible to recover by simply reversing the modular additions as shown in the cipher diagram on page 3.

Due to the modular additions feeding into eachother, the circular shifts, and the chunk rotations at the end of each round, each bit of the block becomes highly dependent on every other bit thereby creating a strong avalanche effect with respect to changes in the input block. However, it is not enough for the avalanche effect to only occur for changes in the input block. In order for the cipher to be secure, an avalanche effect must also occur for changes in the key. Fortunately, this problem can be easily solved by adding another four internal rounds to the cipher, hence why SEAL has 8 internal rounds.

Locking Round Function

The reason behind the addition of a final locking round to the cipher was twofold. First, by adding a round composed entirely of XOR'ing the block chunks against the key, no information about the internal rounds is leaked via the ciphertext produced. This prevents the analysis shown in Figure 2 from even being able to occur. Secondly, by adding this fundamentally different round to the end of the cipher, a slide attack is no longer possible since there is no longer any way to partition the cipher into symmetric components.

Round Keys

In a similar manner to TEA, SEAL has no key schedule and as such each round key is nothing more than a 32-bit chunk taken from the 128-bit key. This was done in order to maximize ease of use as well as processing speed. That being said, the absence of a key schedule potentially leaves the internal rounds vulnerable to a slide attack. In order to alleviate this concern, the key is XOR'd against the round number before being XOR'd against the first chunk thereby removing the symmetry between the internal rounds needed for a slide attack to take place.

PeerJ Preprints NOT PEER-REVIEWED

PERFORMANCE

On a modest system using an Intel Core i7-7500U 2.7 GHz processor, SEAL was able to perform at a rate of 45 MB/s, on average, making it a competitive cipher in terms of speed. If dedicated hardware were to be introduced for SEAL, this speed would undoubtedly increase.

CONCLUSION

In this paper, I have presented a Simple Encryption Algorithm which can be easily implemented in software, performs quite well in spite of the lack of dedicated hardware, and is seemingly secure without the need for a key schedule or more than eight internal rounds.

REFERENCES

1. A. F. Webster and Stafford E. Tavares, ”On the design of S-boxes”, Advances in Cryptology - Crypto ’85 (Lecture Notes in Computer Science), vol. 219, pp. 523–534, 1985.

2. Wheeler, David J.; Needham, Roger M. (1994-12-16). ”TEA, a tiny encryption algorithm”. Lecture Notes in Computer Science. Leuven, Belgium: Fast Software Encryption: Second International Workshop. 1008: 363–366. ISBN: 978-3-540-47809-6

3. Alex Biryukov and David Wagner (March 1999). Slide Attacks. 6th International Workshop on Fast Software Encryption (FSE ’99). Rome: Springer-Verlag. pp. 245–259. Retrieved 2007-09-03.

APPENDIX

Decryption Procedure

Below is the decryption procedure for SEAL written in the C programming language:

```c
/*Substitute the bytes of a block chunk with the SEAL inverse S-Box*/
void INV_S(uint32_t *block_chunk){
   for(int i = 0; i < 4; i++){
      uint8_t byte = (uint8_t)(*block_chunk >> 8*i);
      *block_chunk = *block_chunk & ~(0xFF << 8*i);
      *block_chunk |= ((uint32_t)inv_sbox[byte] << 8*i);
   }
}
/*Decrypts a 128-bit block against a 128-bit key using SEAL*/
void seal_decrypt(uint32_t *block, uint32_t *key){
  /*The carry chunk used for rotation and feeding*/
  uint32_t carry = 0;

   /*Unlock the block by XOR'ing against key*/
  for(int i = 0; i < 4; i++)
    block[i] ^= key[i];

  /*Substitute the block chunks through the inverse S-box*/
  INV_S(&block[0]); INV_S(&block[1]);
  INV_S(&block[2]); INV_S(&block[3]);

  /*Perform 8 rounds of decryption*/
  for(int i = 7; i >= 0; i--){
    /*Set the carry*/
    carry = block[0];

    /*Rotate the block chunks*/
    block[0] = block[1]; block[1] = block[2];
    block[2] = block[3]; block[3] = carry;

    /*Perform modular subtraction on the subsequent chunks*/
    carry = block[2];
    carry = (carry >> 11) | (carry << 21); //Rotate carry chunk
    block[3] -= carry;

    carry = block[1];
    carry = (carry >> 11) | (carry << 21);
    block[2] -= carry;

    carry = block[0];
    carry = (carry >> 11) | (carry << 21);
    block[1] -= carry;

    /*XOR the block chunk against the key chunk and round number and run
       through inverse S-box*/
    block[0] ^= key[i%4] ^ i;
    INV_S(&block[0]);
  }
}
```

PeerJ Preprints NOT PEER-REVIEWED

Inverse Substitution Box

Below is the 16x16 inverse lookup table utilized by the decryption routine. The same rules described previously apply to this table with regards to byte substitution.

Table 2. Inverse S-box

S	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	62	04	21	13	50	9b	2f	e2	20	5c	85	79	22	63	80	f2
1	de	52	42	ae	fa	05	55	4e	5d	3f	d8	8c	31	77	d1	32
2	3c	07	db	e5	6a	43	70	f8	c4	0f	1e	d2	6f	1d	89	87
3	06	df	fe	c2	12	a1	d5	bc	9e	a8	58	71	f4	6e	f1	73
4	28	57	72	46	2e	37	ba	5b	0c	ed	39	4f	c1	6c	86	4c
5	48	fc	18	16	e8	8e	f6	78	2c	d4	ad	41	ca	40	b7	59
6	91	10	61	b9	64	33	36	ab	cc	4d	aa	9c	51	2d	2b	d7
7	a2	94	67	54	5a	7b	c5	be	cf	92	75	02	65	01	9f	99
8	83	ea	e0	a5	60	35	e4	dc	30	f9	eb	af	dd	3b	f5	26
9	82	34	29	03	8d	e7	27	b5	17	5f	8f	0b	45	38	25	da
a	3e	ec	47	b4	ac	08	a6	fd	e9	e1	a4	1b	d3	97	cd	9d
b	24	14	1c	11	4a	c6	d9	c7	ef	3a	bb	1a	00	fb	d6	c3
c	9a	bf	74	b6	7d	f7	b3	b8	8a	a3	3d	23	44	c0	0e	95
d	7c	a7	93	76	f0	66	09	15	a0	88	49	0d	96	53	f3	56
e	a9	68	d0	81	ce	69	8b	b2	e6	2a	bd	1f	7e	c9	6b	ff
f	b0	ee	6d	19	c8	90	e3	98	0a	4b	7a	7f	b1	cb	5e	84