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 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:

```
/*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
    S(&block[0]); block[0] ^= key[i%4]; carry = block[0];
    //Feed carry chunk through subsequent block chunks
    block[1] += carry; carry = block[1];
    block[2] += carry; carry = block[2];
    block[3] += carry; carry = block[3];
    //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++)
    block[i] ^= key[i]; //XOR chunks against key
```

}

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

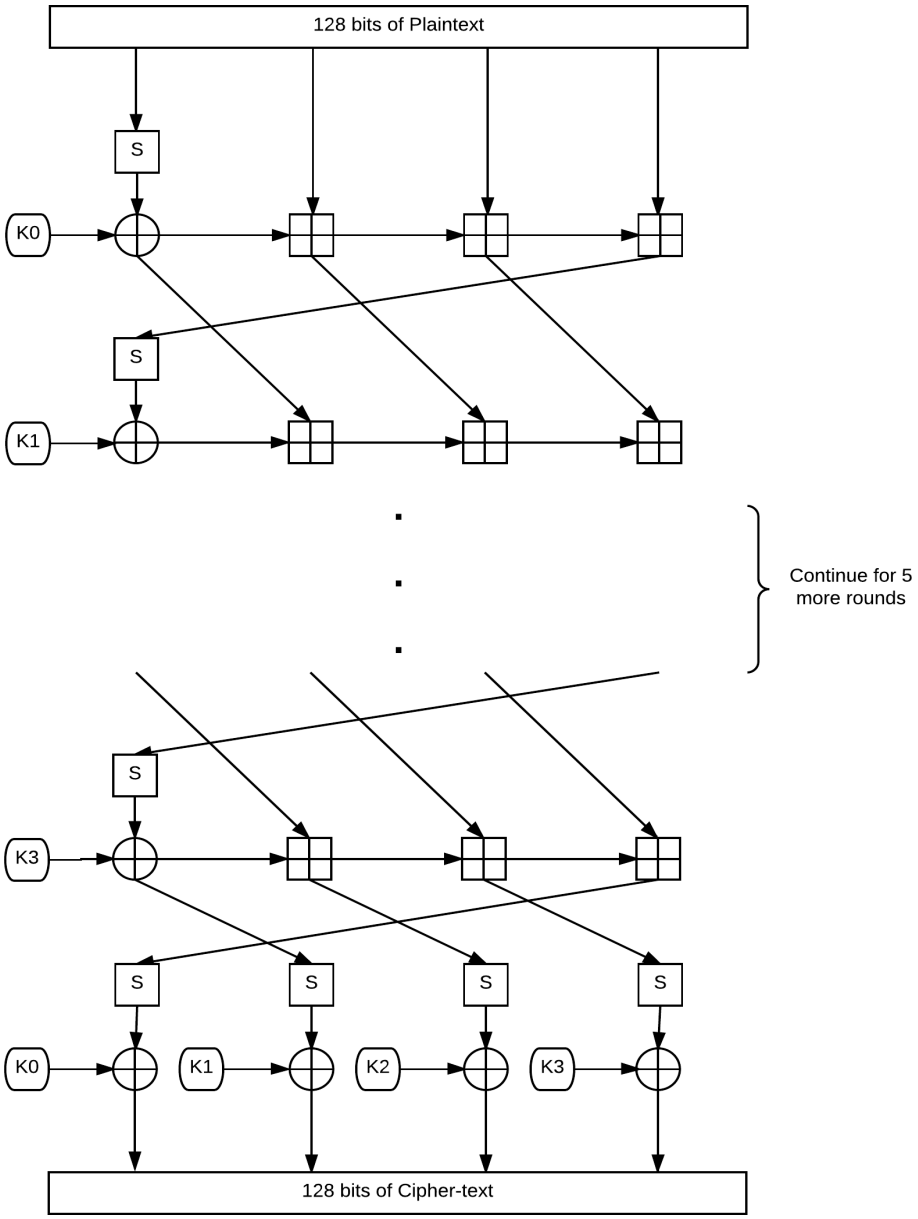
The cipher is a form of substitution-permutation network whereby both the block and the key are divided into equal 32-bit chunks. Through the combined usage of modular additions and the XOR, a high degree of non-linearity is achieved. The substitution boxes are included in order to rapidly introduce 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 required, 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.

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.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3128v1 | CC BY 4.0 Open Access | rec: 7 Aug 2017, publ: 7 Aug 2017

CIPHER DIAGRAM

Below is an overall outline of the SEAL cipher:



ANALYSIS

The security of the algorithm rests primarily on the algebraic incompatibility present between the XOR and modular addition; however, the usage of S-box's is also required in order to ensure that the cipher is resistant to differential cryptanalysis.

Several tests were performed on SEAL for nonlinearity and avalanche effect. From these tests, it was apparent that it takes at least eight internal rounds of SEAL for strong avalanche effects to emerge in the algorithm; however, even after five internal rounds, the cipher begins to show avalanche effects for changes in both the key and block.

Lastly, the cipher's resilience to differential analysis was tested using 100,000 chosen plain-text block pairs. This test was successful in that for the provided sample size, the secret key was still unable to be recovered.

PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3128v1 | CC BY 4.0 Open Access | rec: 7 Aug 2017, publ: 7 Aug 2017

PERFORMANCE

On a modest system using an Intel Core i7-7500U 2.7 GHz processor, SEAL was able to perform at a rate of 48 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 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.

APPENDIX

Decryption Procedure

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

```
/*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]; block[3] -= carry;

    carry = block[1]; block[2] -= carry;

    carry = block[0]; block[1] -= carry;

    /*XOR the block chunk against the key chunk and run through inverse
        S-box*/
    block[0] ^= key[i&0b11];
    INV_S(&block[0]);
  }
}
```


PeerJ Preprints | https://doi.org/10.7287/peerj.preprints.3128v1 | CC BY 4.0 Open Access | rec: 7 Aug 2017, publ: 7 Aug 2017