International Data Encryption Algorithm

- International Data Encryption Algorithm (IDEA) is a block
- It is a minor revision of an earlier cipher, PES (Proposed Encryption Standard); IDEA was originally called IPES (Improved PES).
- It entirely avoids the use of any lookup tables or S-boxes.
- The IDEA encryption algorithm provides security not based on keeping the algorithm a secret, but rather upon ignorance of the secret key

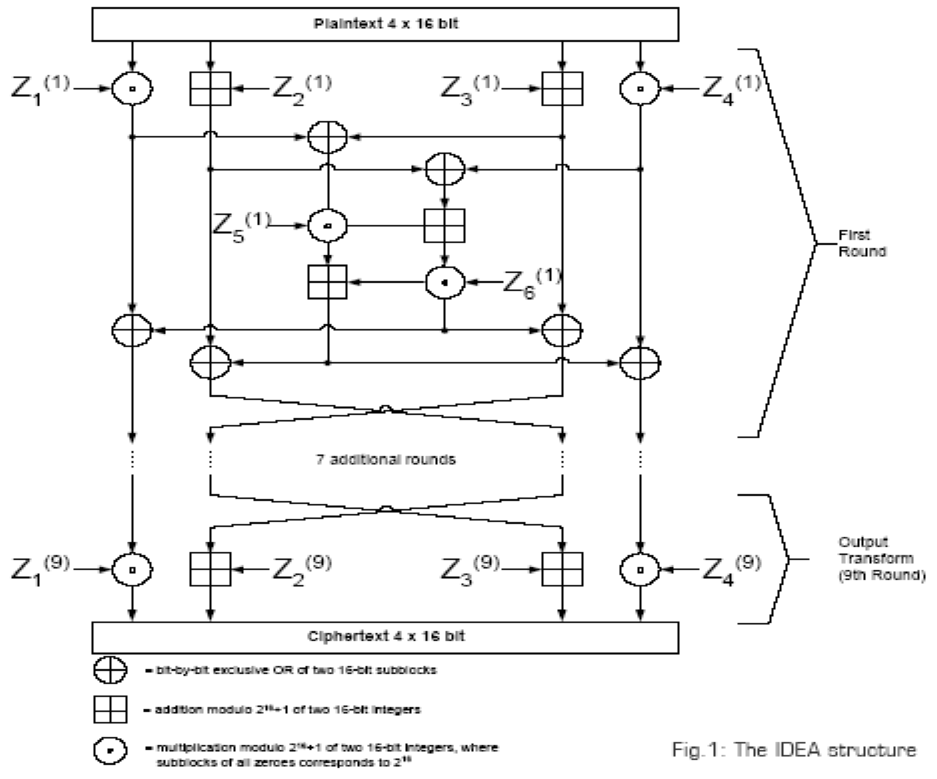
Description of IDEA

The block cipher IDEA operates with 64-bit plaintext and cipher text blocks, controlled by a 128-bit key.

Key Generation

- The 64-bit plaintext block is partitioned into four 16-bit sub-blocks,
- Another process produces six 16-bit key sub-blocks from the 128-bit key. Since a further four 16-bit key-sub-blocks are required for the subsequent output transformation, a total of 52 (= 8 x 6 + 4) different 16-bit sub-blocks have to be generated from the 128-bit key.

Encryption


Plaintext 4 x 16 bit
$Z_1^{(1)}$
$Z_2^{(1)}$
$Z_3^{(1)}$
$Z_4^{(1)}$
$Z_5^{(1)}$
$Z_6^{(1)}$
First Round
7 additional rounds
$Z_1^{(9)}$
$Z_2^{(9)}$
$Z_3^{(9)}$
$Z_4^{(9)}$
Output Transform (9th Round)
Ciphertext 4 x 16 bit
= bit-by-bit exclusive OR of two 16-bit subblocks
= addition modulo $2^{16}+1$ of two 16-bit integers
= multiplication modulo $2^{16}+1$ of two 16-bit integers, where subblocks of all zeroes corresponds to 2^{16}
Fig.1: The IDEA structure

Decryption

The computational process used for decryption of the ciphertext is essentially the same as that used for encryption of the plaintext. The only difference compared with encryption is that during decryption, different 16-bit key sub-blocks are generated.

Weak keys for IDEA

For a class of 223 keys IDEA exhibits a linear factor.

For a certain class of 235 keys the cipher has a global characteristic with probability 1.

SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm

James L. Massey
Signal and Information Processing Laboratory
Swiss Federal Institute of Technology
CH-8092 Zürich

Abstract: A new non-proprietary secret-key block-enciphering algorithm, SAFER K-64 (for Secure And Fast Encryption Routine with a Key of length **64** bits) is described. The blocklength is 64 bits (8 bytes) and only byte operations are used in the processes of encryption and decryption. New cryptographic features in SAFER K-64 include the use of an unorthodox linear transform, called the Pseudo-Hadamard Transform, to achieve the desired "diffusion" of small changes in the plaintext or the key over the resulting ciphertext and the use of additive key biases to eliminate the possibility of "weak keys". The design principles of K-64 are explained and a program is given, together with examples, to define the encryption algorithm precisely.

1. Introduction

This paper describes a new block encryption algorithm called **SAFER K-64** (for Secure And Fast Encryption Routine with a Key of length **64** bits) that the author recently developed for Cylink Corporation (Sunnyvale, CA, USA) as a non-proprietary cipher. SAFER K-64 is a byte-oriented block enciphering algorithm. The block length is 8 bytes (64 bits) for plaintext and ciphertext; the user-selected key is also 8 bytes (64 bits) in length. SAFER K-64 is an *interated* cipher in the sense that encryption is performed by applying the same transformation repeatedly for r rounds, then applying an output transformation; r = 6 is recommended but larger values of r can be used if desired for even greater security. Each round uses two 8-byte (64-bit) subkeys determined by a key schedule from the secret 8-byte user-selected key. The output transformation uses another 8-byte subkey determined by the key schedule. One unusual feature of SAFER K-64 is that, in contrast to most recently proposed iterated block ciphers, encryption and decryption are slightly different (i.e., they differ by more than just the reversal of the key schedule).

SAFER K-64 uses only byte operations in the processes of encryption and decryption, which makes it particularly useful in applications such as smart cards where very limited processing power is available. Some bit-level rotations of bytes are used in the key schedule, but this is done "once and for all", i.e., until the user-selected key is changed. To achieve security with such simple processing, SAFER K-64 exploits two new cryptographic concepts, namely:

(1) an **unorthodox linear transform**, which we call the *Pseudo-Hadamard Transform (PHT)*, that allows the cipher rapidly to achieve the desired "diffusion" of small changes in the plaintext or the key over the resulting ciphertext [It is usually the case in block cipher design that one struggles to obtain such diffusion by carefully selecting permutations to imbed within the cipher and then doing massive statistical testing to see which ones give acceptable diffusion. As will be seen, the PHT provides a systematic way to ensure that the cipher provides the necessary diffisuion--in fact, the diffusion provided by the PHT appears to be better than that in any other cipher that we know.]

and (2) the use of additive **key biases** that eliminate the "weak keys" that plague most block ciphers. [SAFER K-64 includes a recursive procedure for generating these key biases that is easy to implement and that provides very "random" biases desired.]

2. Description of the SAFER K-64 Algorithm

The encrypting structure of the SAFER K-64 cipher is shown in Fig. 1. The enciphering algorithm consists of r rounds of identical transformations that are applied in sequence to the plaintext, followed by an output transformation, to produce the final ciphertext. Our recommendation is to use $r = 6$ for most applications, but up to 10 rounds can be used if desired. Each round is controlled by two 8-byte subkeys and the output transformation is controlled by one 8-byte subkey. These $2r + 1$ subkeys are all derived from the 8-byte user-selected subkey K1 in a manner that will be explained later. The plaintext, ciphertext and all subkeys are 8 bytes (64 bits) long.

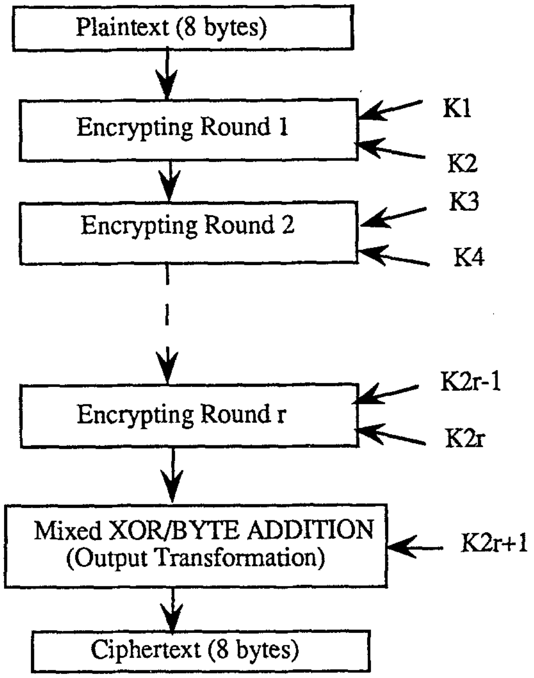


Fig. 1: Encrypting Structure of SAFER K-64

The *output transformation* of SAFER K-64 consists of the bit-by-bit XOR ("exclusive or" or modulo-2 sum) of bytes 1, 4, 5 and 8 of the last subkey, K2r+1, with the corresponding bytes of the output from the r-th round together with the byte-by-byte byte addition (modulo-256 addition) of bytes 2, 3, 6 and 7 of the last subkey, K2r+1, to the corresponding bytes of the output from the r-th round. [Higher order bytes are considered to be those on the left, i.e., byte 1 is the most significant byte--this convention is used throughout this paper.] Hereafter, we refer to this particular combination of two eight-byte words as the *Mixed XOR/Byte-Addition operation.*

The detailed *encryption round structure* of SAFER K-64 is shown in Fig. 2. The first step within the i^{th} round is the Mixed XOR/Byte-Addition of the round input with the subkey K2i-1. The eight bytes of the result are then passed through a *nonlinear layer* and individually subjected to one of two different "highly nonlinear" transformations, namely:
(1) the operation labelled "$45^{(\cdot)}$" in Fig. 2, which notation is to suggest that if the byte input is the integer j then the byte output is 45^j modulo 257 (*except that this output is taken to be 0 if the modular result is 256, which occurs for j = 128*) [The reasoning behind the use of this transformation is the following. Because 257 is a prime, arithmetic

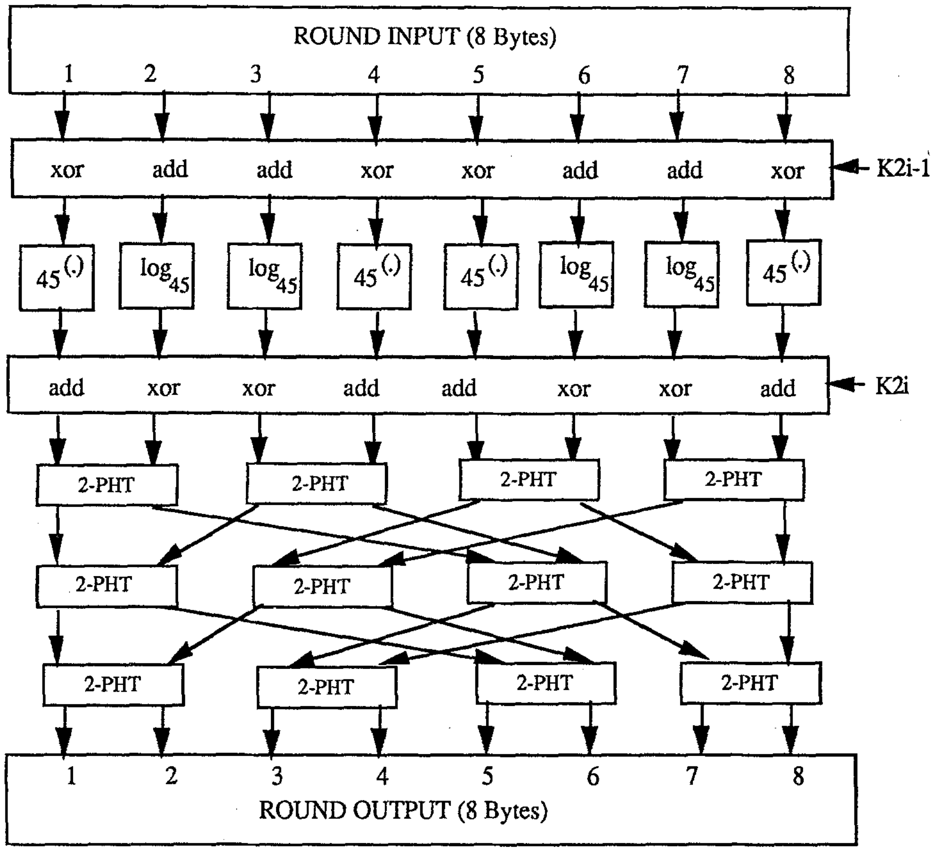

ROUND INPUT (8 Bytes)
1 2 3 4 5 6 7 8
xor add add xor xor add add xor
K2i-1
45^(.) log45 log45 45^(.) 45^(.) log45 log45 45^(.)
add xor xor add add xor xor add
K2i
2-PHT
ROUND OUTPUT (8 Bytes)

Fig. 2: Encryption round structure of SAFER K-64

modulo 257 is the arithmetic of the finite field GF(257). The element 45 is a primitive element of this field, i.e., its first 256 powers generate all 256 non-zero field elements. Thus the mapping "$45^{(\cdot)}$" is an invertible mapping from one byte to one byte that is very nonlinear with respect to the arithmetic of GF(257) as well as with respect to the vector space of 8-tuples over the binary field GF(2) whose addition is bit-by-bit XOR.]
and (2) the operation labelled "$\log_{45}$" in Fig. 2, which notation is to suggest that if the byte is the integer j then the byte output is $\log_{45}(j)$ (*except that this output is taken to be 128 if the input is $j = 0$*), i.e., the power to which one must raise 45 to obtain j modulo 257. [The nonlinear features of this mapping are similar to those described for exponentiation.]

In the appended programs for implementing the SAFER K-64 cipher, these two nonlinear operations are realized with two look-up tables of 256 bytes each, i.e., simple byte-in byte-out look-up tables.

The output of the eight nonlinear transformations is then combined with subkey K2i in an operation that consists of the byte-by-byte byte addition (modulo-256 addition) of bytes 1, 4, 5 and 8 of the subkey K2i to the corresponding bytes of the output from the nonlinear transformations together with the bit-by-bit XOR (modulo-2 sum) of bytes 2, 3, 6 and 7 of the subkey K2i to the corresponding bytes of the output from the nonlinear transformations. Hereafter, we refer to this particular combination of two eight-byte words as the *Mixed Byte-Addition/XOR operation.* [It is important to note the distinction between this Mixed XOR/Byte-Addition operation and the previously described Mixed Byte-Addition/XOR operation.]

The output of the Mixed Byte-Addition/XOR operation then passes through a *three-level "linear layer"* of boxes that are labelled "2-PHT" in Fig. 2. This notation indicates a 2-point PHT. If the two input bytes to a 2-PHT are (a_1, a_2), where a_1 is the more significant byte, then the two output bytes are (b_1, b_2) where

$$\begin{aligned} b_1 &= 2a_1 + a_2 \\ b_2 &= a_1 + a_2 \end{aligned} \tag{1}$$

and where the arithmetic is normal byte arithmetic, i.e., arithmetic modulo 256. Between levels of the linear layer, the decimation-by-2 permutation [familiar from the Cooley-Tukey FFT and the ordinary discrete Hadamard Transform] is applied--as will be seen, this is what creates diffusion in the SAFER K-64 cipher. The output of this linear layer constitutes the round output.

3. Decryption for SAFER K-64

The decrypting structure of SAFER K-64 is shown in Fig. 3. The deciphering algorithm consists of an *input transformation* that is applied to the ciphertext block,

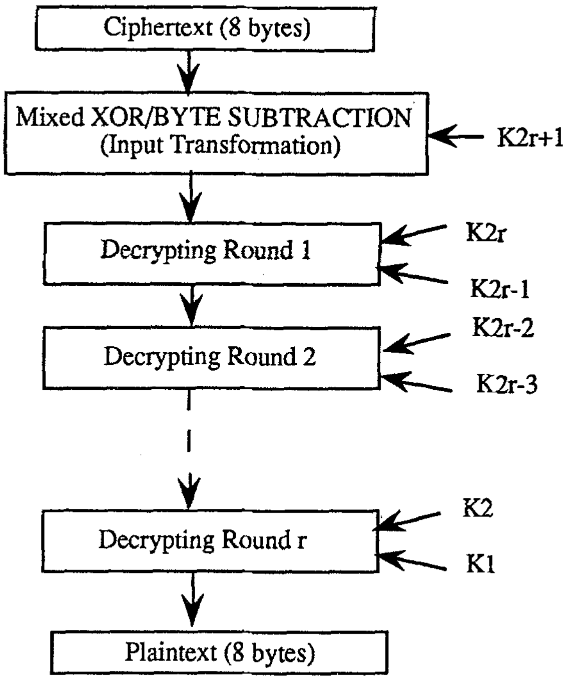


Fig. 3: Decrypting Structure of SAFER K-64

followed by r rounds of identical transformations. The input transformation consists of the Mixed XOR/Byte-Subtraction of subkey K2r+1 from the ciphertext block. A characterizing feature of SAFER K-64 is that decrypting rounds differ from encrypting rounds so that an encrypter cannot be converted to a decrypter by simply reversing the key schedule.

The detailed *decryption round structure* of SAFER K-64 is shown in Fig. 4. The i^{th} encryption round begins by passing the round input through the *three-level inverse linear layer*. It is easy to check from equations (1) that the Inverse PHT (IPHT) is given by

$$a_1 = b_1 - b_2$$

$$a_2 = -b_1 + 2b_2. \tag{2}$$

This IPHT is just as simple to compute as the direct PHT. The fan-out-by-two permutation between levels of this inverse linear layer is the inverse of the decimate-by-two permutation used in the linear layer of an encryption round.

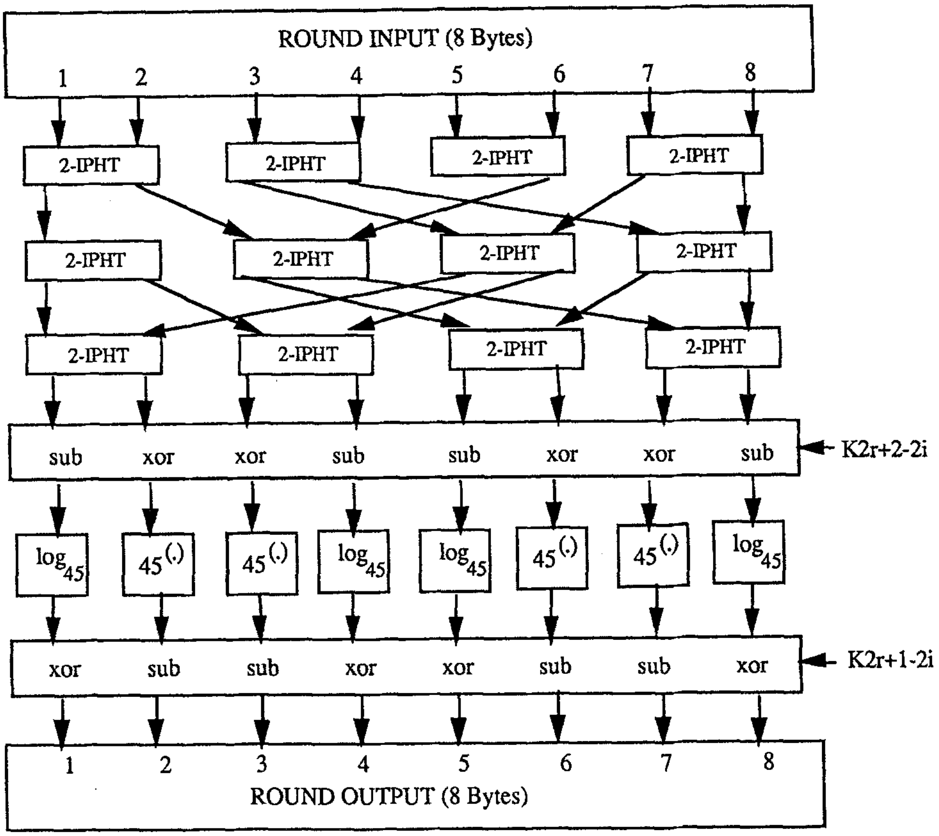

ROUND INPUT (8 Bytes)
1 2 3 4 5 6 7 8
2-IPHT
sub xor xor sub sub xor xor sub
K2r+2-2i
log45 45(.) 45(.) log45 log45 45(.) 45(.) log45
xor sub sub xor xor sub sub xor
K2r+1-2i
1 2 3 4 5 6 7 8
ROUND OUTPUT (8 Bytes)

Fig. 4: Decryption round structure of SAFER K-64

The next step within the ith decryption round is the *Mixed Byte-Subtraction/XOR* of the output of the inverse linear layer with the subkey K2r+2-2i, which consists of the byte-by-byte byte subtraction (modulo-256 subtraction) of bytes 1, 4, 5 and 8 of the subkey K2r+2-2i from the corresponding bytes of the output from the previous round together with the bit-by-bit XOR (modulo-2 sum) of bytes 2, 3, 6 and 7 of the subkey K2r+2-2i with the corresponding bytes of the output from the previous round.

In the next step of the decryption round, the eight bytes from the previous step are passed through the "*inverse nonlinear layer*", which differs from the "nonlinear layer" in the encryption round by interchanging of the locations of the four exponentiating boxes and the four logarithm-taking boxes.

The last step within the ith decryption round is the *Mixed XOR/Byte-Subtraction* of the round input with the subkey K2r+1-2i, which consists of the bit-by-bit XOR (modulo-2 sum) of bytes 1, 4, 5 and 8 of the subkey K2r+1-2i with the corresponding bytes of the output from the previous round together with the byte-by-byte byte subtraction (modulo-256 subtraction) of bytes 2, 3, 6 and 7 of the subkey K2r+1-2i from the corresponding bytes of the output from the previous round.

4. How SAFER K-64 Works and Why

To see that the SAFER K-64 cipher correctly decrypts, we first note that the Mixed XOR/Byte-Subtraction of K2r+1 in the Input Transformation for decryption (cf. Fig. 3) undoes the Mixed XOR/Byte-Additon of K2r+1 in the Output Transformation for encryption (cf. Fig. 1). Then the inverse linear layer of the first decryption round (cf. Fig. 4) undoes the transformation performed by the linear layer in the last encryption round (cf. Fig. 2). Next, the Mixed Byte-Subtraction/XOR of K2r in the first decryption round (cf. Fig. 4) undoes the Mixed Byte-Addition/XOR of K2r in the last encryption round (cf Fig. 2). Then the inverse nonlinear layer in the first decryption round (cf. Fig. 4) undoes the transformation performed by the nonlinear layer in the last encryption round (cf. Fig. 2). Finally, the Mixed XOR/Byte-Subtraction of K2r-1 in the first decryption round (cf. Fig. 4) undoes the Mixed XOR/Byte-Addition of K2r-1 in the last encryption round (cf. Fig. 2). In the same way, decryption round i undoes the transformation performed by encryption round r + 1 - i for i = 2, 3, ... , r so that decryption indeed recovers the original plaintext.

SAFER K-64 was designed in accordance with Shannon's principles of confusion and diffusion for obtaining security in secret-key ciphers [1]. When a round subkey is not all-zero in SAFER K-64 encryption, its combination by Mixed XOR/Byte Addition (or Mixed Byte-Addition/XOR) with the signal within the round acts like a nonlinear combination with respect to the subsequent transformations in the nonlinear layer and in the linear layer. This gives the cipher the *confusion* required to make the statistics of the ciphertext depend in a complicated way on the statistics of the plaintext--provided that small changes diffuse quickly through the cipher. To guarantee this *diffusion* in SAFER K-64 is, in fact, why we developed a new and unorthodox linear transform, the Pseudo-Hadamard Transform (PHT).

The standard Hadamard Transform (HT) [sometimes called the "Walsh transform" or the "Walsh-Hadamard transform"] has in place of (1) the equations

$$\begin{aligned} b_1 &= a_1 + a_2 \\ b_2 &= a_1 - a_2. \end{aligned} \qquad (3)$$

Notice that the determinant of the matrix of coefficients is -2, which makes these equations non-invertible for byte arithmetic (arithmetic modulo 256) where -2 = 254 has no multiplicative inverse. It also has the unpleasant effect of requiring a multiplication by 1/2 in the inverse transform in those number systems where 2 has a multiplicative inverse. By choosing equations (1), whose matrix of coefficients has determinant 1, we avoid both of these problems--we can use normal byte arithmetic and there is no unpleasant scale factor in the inverse transform! Moreover, we can still mimic the HT in the multi-dimensional case, which is what the decimations by-two and fanning-outs by-two accomplish. We are in fact using a three-dimensional PHT, i.e., independent 2-PHTs in each of 3 dimensions, which is why there are $2^3 = 8$ bytes in the input and output of the PHT within SAFER K-64.

Just as for the HT in number systems appropriate to it, every digit (here read "byte") of the input to the PHT effects every output byte, i.e., the PHT provides guaranteed complete diffusion within one linear layer. In Appendix A, we show the PHT for the unit-vector inputs where one sees this diffusion over all eight output bytes very clearly. By linearity, the PHT of any vector can be computed as the corresponding linear combination of these unit-vector PHT's. The "guaranteed complete diffusion" within one layer does not hold fully when one considers single-bit changes in the input bytes. Because of the factor of 2 in equations (1), a few bits of the input will effect only 4 bytes (or 2 bytes or 1 byte) of the output within one linear layer, but their effect is immediately spread over all 8 bytes in the next linear layer encountered. This can be seen from the last three examples in Appendix A. For instance, because (1,0,0,0,0,0,0,0) has the PHT (8,4,4,2,4,2,2,1), it follows [from the fact that $2 * 128 = 0 \bmod 256$] that (128,0,0,0,0,0,0,0), which contains a single non-zero bit, will have the PHT (0,0,0,0,0,0,0,128), which shows no diffusion at all. However, in turn (0,0,0,0,0,0,0,128) has the PHT (128,128,128,128,128,128,128,128), which shows complete diffusion over output bytes. In fact, consideration of the unit-vector PHT's in Appendix A shows that (128,0,0,0,0,0,0,0) is the only vector that shows no diffusion under one application of the PHT. We know of no other cipher with such rapid and guaranteed diffusion. This rapid diffusion is the main reason that r = 6 rounds of encipherment are enough to make SAFER K-64 crack-resistant.

5. The Key Schedule for SAFER K-64

The *key schedule* for SAFER K-64, i.e., the procedure for generating the subkeys K2, K3, ... , K2r+1 from the user-selected subkey K1, is indicated in Fig. 5. The quantities B2, B3, ... , B2r-1 are the *key biases* that have the purpose of ensuring that the round subkeys appear individually "random" and, in particular, that no more than one round subkey can be all-zero. Letting b[i,j] denote the j-th byte of bias Bi, we can express this byte as the double exponential

$$b[i,j] = 45^{**}[45^{**}(9i+j) \bmod 257] \bmod 257, \qquad (4)$$

which equation defines the key biases used in SAFER K-64. We note here that we might have used the factor 8 instead of 9 in the exponent in (4)--we chose 9 to introduce an extra measure of "staccato" in the key schedule." A Table giving the precise values of the key biases for SAFER K-64 is given in Appendix B. Examination of the Table in Appendix B shows that the resulting sequence of biases is indeed very random appearing, which is all that is really needed. The use of such biases, which appears to be new, is clearly a good idea in general for iterated ciphers. The "weak keys" (also called "self-dual keys" and "keys with a dual") of the Data Encryption Standard (DES) [2] are a direct result of the fact that no key biases are used so that, for instance, all 16 round subkeys in DES can be all-zero.]

Fig. 5 shows how K1, the user-selected 64-bit subkey, is used to generate the additional 64-bit subkeys K2, K3, ... , K2r+1 that are required within the r-round SAFER K-64 algorithm. Note that, in the generation process, the subkey register is byte-wise rotated by 3-bits to the left between additions of a new bias. [The addition of a bias is always byte-by-byte byte addition (modulo-256 addition).] Ideally, one wishes the entire subkey sequence K1, K2, K3, ... , K2r+1 to have the character of a sequence of independently-chosen uniformly-random subkeys. Of course, this cannot be achieved in a strict sense because all of the subkeys in this sequence are determined entirely by the first (user-selected) subkey, K1. The real goal in the design of the key schedule is to make the departure from independence so complicated that it cannot be exploited by an attacker--and this is the purpose of both the byte rotations and the addition of subkey biases within the key schedule for SAFER K-64

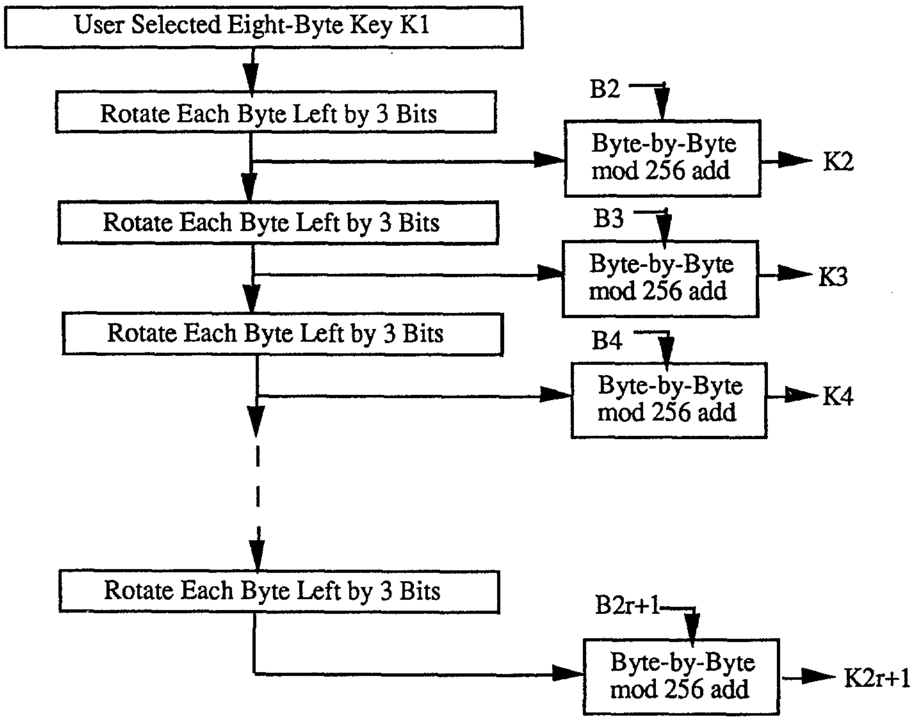

User Selected Eight-Byte Key K1
Rotate Each Byte Left by 3 Bits
B2
Byte-by-Byte mod 256 add
K2
Rotate Each Byte Left by 3 Bits
B3
Byte-by-Byte mod 256 add
K3
Rotate Each Byte Left by 3 Bits
B4
Byte-by-Byte mod 256 add
K4
Rotate Each Byte Left by 3 Bits
B2r+1
Byte-by-Byte mod 256 add
K2r+1

Fig. 5: Key Schedule for SAFER K-64

6. SAFER K-64 Program and Examples

Appendix C gives a TURBO PASCAL program that implements the full r-round SAFER K-64 cipher, both for encryption and decryption.. This program should be taken as the definition of the SAFER K-64 enciphering algorithm. Appendix C also gives examples of r = 6 round encryption (the recommended number of rounds) for use in checking implementations of SAFER K-64.

7. Security Considerations for SAFER K-64

In Section 4, we indicated how SAFER K-64 achieves both good diffusion and good confusion, the two basic features that contribute to the security of a block cipher. The best measure of security available today for an iterated block cipher is its resistance to attack by differential cryptanalysis [3]. It is easy to show that, for the appropriate definition of difference between a pair of plaintext blocks (or a pair of ciphertext blocks), SAFER K-64 is a *Markov cipher* [4], a fact that greatly simplifies its analysis for resistance to differential cryptanalysis. Cylink Corporation has contracted for such an analysis of SAFER K-64 by a group of cryptanalysts that does not include the designer of the algorithm. A considerable effort has been invested in this effort, whose conclusion is that six-round SAFER K-64 appears to be secure against differential cryptanalysis. This group of cryptanalysts has also done extensive statistical testing of SAFER K-64 with no detection of any weakness. The evidence available today suggests that SAFER K-64 is a strong cipher whose strength is well measured by the length (64 bits) of its user-selected key.

References

[1] C.E. Shannon, "Communication Theory of Secrecy Systems", *Bell System Tech. J.*, vol. 28, pp. 656-715, Oct., 1949.

[2] U.S. Department of Commerce/National Bureau of Standards, FIPS Pub 46, *Data Encryption Standard*, April 1977.

[3] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard.* New York: Springer-Verlag, 1993.

[4] X. Lai, J. L. Massey and S. Murphy, "Markov Ciphers and Differential Cryptanalysis," pp. 17-38 in *Advances in Cryptology - EUROCRYPT '91* (Ed. D. W. Davies), Lecture Notes in Computer Science No. 547. Heidelberg and New York: Springer-Verlag, 1991

APPENDIX A:
Examples of the Pseudo-Hadamard Transform (PHT)

```
INPUT  VECTOR is    0   0   0   0   0   0   0   1
OUTPUT VECTOR is    1   1   1   1   1   1   1   1

INPUT  VECTOR is    0   0   0   0   0   0   1   0
OUTPUT VECTOR is    2   2   1   1   2   2   1   1

INPUT  VECTOR is    0   0   0   0   0   1   0   0
OUTPUT VECTOR is    2   2   2   2   1   1   1   1

INPUT  VECTOR is    0   0   0   0   1   0   0   0
OUTPUT VECTOR is    4   4   2   2   2   2   1   1

INPUT  VECTOR is    0   0   0   1   0   0   0   0
OUTPUT VECTOR is    2   1   2   1   2   1   2   1

INPUT  VECTOR is    0   0   1   0   0   0   0   0
OUTPUT VECTOR is    4   2   2   1   4   2   2   1

INPUT  VECTOR is    0   1   0   0   0   0   0   0
OUTPUT VECTOR is    4   2   4   2   2   1   2   1

INPUT  VECTOR is    1   0   0   0   0   0   0   0
OUTPUT VECTOR is    8   4   4   2   4   2   2   1

INPUT  VECTOR is  128   0   0   0   0   0   0   0
OUTPUT VECTOR is    0   0   0   0   0   0   0 128

INPUT  VECTOR is    0   0   0   0   0   0   0 128
OUTPUT VECTOR is  128 128 128 128 128 128 128 128
```

APPENDIX B:

Table of Key Biases for SAFER K-64 Cipher.

(Biases B2 to B21 are listed here although only B2 to B13 are required when r = 6 rounds are used with SAFER K-64.)

```
Bias  B2 is   22 115  59  30 142 112 189 134
Bias  B3 is   71 126  36  86 241 119 136  70
Bias  B4 is  177 186 163 183  16  10 197  55
Bias  B5 is  201  90  40 172 100 165 236 171
Bias  B6 is  198 103 149  88  13 248 154 246
Bias  B7 is  102 220   5  61 211 138 195 216
Bias  B8 is  106 233  54  73  67 191 235 212
Bias  B9 is  155 104 160 101  93  87 146  31
Bias B10 is  113  92 187  34 193 190 123 188
Bias B11 is   99 148  95  42  97 184  52  50
Bias B12 is  253 251  23  64 230  81  29  65
Bias B13 is  143  41 221   4 128 222 231  49
Bias B14 is  127   1 162 247  57 218 111  35
Bias B15 is  254  58 208  28 209  48  62  18
Bias B16 is  205  15 224 168 175 130  89  44
Bias B17 is  125 173 178 239 194 135 206 117
Bias B18 is   19   2 144  79  46 114  51 133
Bias B19 is  141 207 169 129 226 196  39  47
Bias B20 is  122 159  82 225  21  56  43 252
Bias B21 is   66 199   8 228   9  85  94 140
```

APPENDIX C:
Examples of Six-Round SAFER K-64 Encryption and Program for Implementation

```
PLAINTEXT is        1   2   3   4   5   6   7   8
The KEY is          0   0   0   0   0   0   0   0
after round 1       0  46 170 144 255 118   2 238
after round 2      35 175 193 103 246  87  43 202
after round 3      64 252   4  38   1 140  36 104
after round 4       2  62 127  41  25  97 179 196
after round 5      59 221   9 152 113  50 224  52
after round 6     242 255  38 130 179 219  71 133
CRYPTOGRAM is     125  40   3 134  51 185  46 180

PLAINTEXT is        0   0   0   0   0   0   0   0
The KEY is          1   2   3   4   5   6   7   8
after round 1     240 174  18 192  79 214   2  46
after round 2      51 154 197 181 138 198 236  83
after round 3     178  36  41  77  26  13 222  86
after round 4     111  39 188 122  73 216  30 100
after round 5     132  78 244 157 225  84 106 144
after round 6     197 105 114  54 196 101 227  80
CRYPTOGRAM is      90 178 127 114  20 163  58 225

PLAINTEXT is        1   2   3   4   5   6   7   8
The KEY is          8   7   6   5   4   3   2   1
after round 1     101  42 122 106  63 111 225 227
after round 2     102 122  66 171  75 196 228  30
after round 3     114 219 165 207  71  24 132 155
after round 4     117  53 164  99 161 204 201  48
after round 5     132  77 246 149   5 187 182  27
after round 6     199  89  95 137  71 106  55 152
CRYPTOGRAM is     200 242 156 221 135 120  62 217

PLAINTEXT is        0   0   0   0   0   0   0   0
The KEY is          0   0   0   0   0   0   0   0
after round 1     203 244 158 176 123 197  11  39
after round 2      27  47   1  53 133  49 233 187
after round 3     134 147 160 151  93   5 125 185
after round 4     190 249 153 140 109 203 139  58
after round 5     143  72 176 126  51 175  84  69
after round 6     140 255  43 205 142   9 196  78
CRYPTOGRAM is       3  40   8 201  14 231 171 127
```

```
PROGRAM  Full_r_Rounds_max_10_of_SAFERK64_cipher;

VAR a1, a2, a3, a4, a5, a6, a7, a8, b1, b2, b3, b4, b5, b6, b7, b8, r: byte;
  k: ARRAY[1..21,1..8] OF byte; k1: ARRAY[1..8] OF byte;
  logtab, exptab: ARRAY[0..255] OF integer; i, j, flag: integer;

PROCEDURE mat1(VAR a1, a2, b1, b2: byte);
BEGIN b2:= a1 + a2; b1:= b2 + a1; END;

PROCEDURE invmat1(VAR a1, a2, b1, b2: byte);
BEGIN b1:= a1 - a2; b2:= -b1 + a2; END;

BEGIN
{The program here computes the powers of the primitive element 45 of the
 finite field GF(257) and stores these in the table "exptab".  Corresponding
 logarithms to the base 45 are stored in the table "logtab".}
 logtab[1]:= 0; exptab[0]:= 1;
 FOR i:= 1 TO 255 DO
 BEGIN
  exptab[i]:= (45 * exptab[i - 1]) mod 257;
  logtab[exptab[i]]:= i;
 END;
 exptab[128]:= 0; logtab[0]:= 128; exptab[0]:= 1;

 flag:= 0; writeln;
 writeln('Enter number of rounds r (max 10) desired then hit CR'); readln(r);

 REPEAT
  BEGIN
   writeln; writeln('Enter plaintext in 8 bytes with spaces');
   writeln('  between bytes, then hit CR.');
   writeln('(A byte is an integer between 0 and 255 inclusive.)');
   readln(a1, a2, a3, a4, a5, a6, a7, a8);
   writeln('Enter a key in 8 bytes');
    readln(k[1,1],k[1,2],k[1,3],k[1,4],k[1,5],k[1,6],k[1,7],k[1,8]);
   k1[1]:= k[1,1]; k1[2]:= k[1,2]; k1[3]:= k[1,3]; k1[4]:= k[1,4];
   k1[5]:= k[1,5]; k1[6]:= k[1,6]; k1[7]:= k[1,7]; k1[8]:= k[1,8];
   writeln('PLAINTEXT is ', a1:8,a2:4,a3:4,a4:4,a5:4,a6:4,a7:4,a8:4);
    writeln('The KEY is   ', k[1,1]:8,k[1,2]:4,k[1,3]:4,k[1,4]:4,
                         k[1,5]:4,k[1,6]:4,k[1,7]:4,k[1,8]:4);
{The next instructions implement the key schedule needed to derive keys
 K2, K3, ... K2r+1 from the user-selected key K1.}
   FOR i:= 2 TO 2*r + 1 DO
   FOR j:= 1 TO 8 DO
   BEGIN
    {Each byte of the  key K1 is further left rotated by 3.}
    k1[j]:= (k1[j] shl 3) + (k1[j] shr 5);
    {The key bias is added here.}
    k[i,j]:= k1[j] + exptab[exptab[9*i+j]];
   END;

{The r rounds of encryption begin here.}
   FOR i:= 1 TO r DO
   BEGIN
    {Key 2i-1 is mixed bit and byte added  to the round input.}
```

```
        a1:= a1 xor k[2*i-1,1]; a2:= a2  +  k[2*i-1,2];
        a3:= a3  +  k[2*i-1,3]; a4:= a4 xor k[2*i-1,4];
        a5:= a5 xor k[2*i-1,5]; a6:= a6  +  k[2*i-1,6];
        a7:= a7  +  k[2*i-1,7]; a8:= a8 xor k[2*i-1,8];

        {The result now passes through the nonlinear layer.}
        b1:= exptab[a1]; b2:= logtab[a2]; b3:= logtab[a3]; b4:= exptab[a4];
        b5:= exptab[a5]; b6:= logtab[a6]; b7:= logtab[a7]; b8:= exptab[a8];
        {Key 2i is now mixed byte and bit added to the result.}
        b1:= b1  +  k[2*i,1]; b2:= b2 xor k[2*i,2];
        b3:= b3 xor k[2*i,3]; b4:= b4  +  k[2*i,4];
        b5:= b5  +  k[2*i,5]; b6:= b6 xor k[2*i,6];
        b7:= b7 xor k[2*i,7]; b8:= b8  +  k[2*i,8];

        {The result now enters the first level of the linear layer.}
        mat1(b1, b2, a1, a2); mat1(b3, b4, a3, a4);
        mat1(b5, b6, a5, a6); mat1(b7, b8, a7, a8);
        {The result now enters the second level of the linear layer.}
        mat1(a1, a3, b1, b2); mat1(a5, a7, b3, b4);
        mat1(a2, a4, b5, b6); mat1(a6, a8, b7, b8);
        {The result now enters the third level of the linear layer.}
        mat1(b1, b3, a1, a2); mat1(b5, b7, a3, a4);
        mat1(b2, b4, a5, a6); mat1(b6, b8, a7, a8);

        {The round is now completed!}
         writeln('after round',i:2,a1:8,a2:4,a3:4,a4:4,a5:4,a6:4,a7:4,a8:4);
      END;

      {Key 2r+1 is now mixed bit and byte added to produce the cryptogram.}
      a1:= a1 xor k[2*r+1,1]; a2:= a2  +  k[2*r+1,2];
      a3:= a3  +  k[2*r+1,3]; a4:= a4 xor k[2*r+1,4];
      a5:= a5 xor k[2*r+1,5]; a6:= a6  +  k[2*r+1,6];
      a7:= a7  +  k[2*r+1,7]; a8:= a8 xor k[2*r+1,8];
      writeln('CRYPTOGRAM is',a1:8,a2:4,a3:4,a4:4,a5:4,a6:4,a7:4,a8:4); writeln;
      writeln('Type 0 and CR to continue or -1 and CR to stop run.'); read(flag);
    END
  UNTIL flag < 0;
END.
```

Simple Encrypted Arithmetic Library - SEAL v2.1

Hao Chen[1], Kim Laine[2], and Rachel Player[3]

[1] Microsoft Research, USA
haoche@microsoft.com
[2] Microsoft Research, USA
kim.laine@microsoft.com
[3] Royal Holloway, University of London, UK**
rachel.player.2013@live.rhul.ac.uk

Abstract. Achieving fully homomorphic encryption was a longstanding open problem in cryptography until it was resolved by Gentry in 2009. Soon after, several homomorphic encryption schemes were proposed. The early homomorphic encryption schemes were extremely impractical, but recently new implementations, new data encoding techniques, and a better understanding of the applications have started to change the situation. In this paper we introduce the most recent version (v2.1) of Simple Encrypted Arithmetic Library - SEAL, a homomorphic encryption library developed by Microsoft Research, and describe some of its core functionality.

1 Introduction

In many traditional encryption schemes (e.g. RSA, ElGamal, Paillier) the plaintext and ciphertext spaces have a tremendous amount of algebraic structure, but the encryption and decryption functions either do not respect the algebraic structure at all, or respect only a part of it. Many schemes, such as ElGamal (resp. e.g. Paillier), are multiplicatively homomorphic (resp. additively homomorphic), but this restriction to one single algebraic operation is a very strong one, and the most interesting applications would instead require a ring structure between the plaintext and ciphertext spaces to be preserved by encryption and decryption. The first such encryption scheme was presented by Craig Gentry in his famous work [22], and since then researchers have introduced a number of new and more efficient *fully* homomorphic encryption schemes.
The early homomorphic encryption schemes were extremely impractical, but recently new implementations, new data encoding techniques, and a better understanding of the applications have started to change the situation. In 2015 we released the *Simple Encrypted Arithmetic Library - SEAL* [19] with the goal of providing a well-engineered and documented homomorphic encryption library, with no external dependencies, that

** Much of this work was done during an internship at Microsoft Research, Redmond.

would be equally easy to use both by experts and by non-experts with little or no cryptographic background.
SEAL is written in C++11, and contains a .NET wrapper library for the public API. It comes with example projects demonstrating key features, written both in C++ and in C#. SEAL compiles and is tested on modern versions of Visual Studio and GCC. In this paper we introduce the most recent version, SEAL v2.1, and describe some of its core functionality. The library is publicly available at

`http://sealcrypto.codeplex.com`

and is licensed under the Microsoft Research License Agreement.

1.1 Related Work

A number of other libraries implementing homomorphic encryption exist, e.g. HElib [2] and $\Lambda \circ \lambda$ [18]. The FV scheme has been implemented in [9, 1], both of which use the ideal lattice library NFLlib [31]. Perhaps the most comparable work to SEAL is the C++ library HElib [2] which implements the BGV homomorphic encryption scheme [12].
A comparison of popular homomorphic encryption schemes, including BGV and FV, was presented by Costache and Smart in [14]. An comparison of the implementations, respectively, of BGV as in HElib and of FV as in SEAL would be very interesting, but appears challenging. One reason for this is that the documentation available for HElib [24, 25, 26] does not in general make clear how to select optimal parameters for performance, and in [26, Appendix A] it is noted '*[t]he BGV implementation in HElib relies on a myriad of parameters ... it takes some experimentation to set them all so as to get a working implementation with good performance*'. On the other hand, we know better how to select good parameters for performance for SEAL (see Section 4 below). Such a comparison is therefore deferred to future work.

2 Notation

We use $\lfloor\cdot\rfloor$, $\lceil\cdot\rceil$, and $\lfloor\cdot\rceil$ to denote rounding down, up, and to the nearest integer, respectively. When these operations are applied to a polynomial, we mean performing the corresponding opearation to each coefficient separately. The norm $\|\cdot\|$ always denotes the infinity norm. We denote the reduction of an integer modulo t by $[\cdot]_t$. This operation can also be applied to polynomials, in which case it is applied to every integer coefficient separately. The reductions are always done into the symmetric interval $[-t/2, t/2)$. $\log_a$ denotes the base-a logarithm, and log always denotes the base-2 logarithm. Table 1 below lists commonly used parameters, and in some cases their corresponding names in SEAL.

3 Implementing the Fan-Vercauteren Scheme

In this section we present our implementation of the Fan-Vercauteren (FV) scheme [20].

Parameter	Description	Name in SEAL
q	Modulus in the ciphertext space (coefficient modulus)	`coeff_modulus`
t	Modulus in the plaintext space (plaintext modulus)	`plain_modulus`
n	A power of 2	
$x^n + 1$	The polynomial modulus which specifies the ring R	`poly_modulus`
R	The ring $\mathbb{Z}[x]/(x^n + 1)$	
R_a	The ring $\mathbb{Z}_a[x]/(x^n + 1)$	
w	A base into which ciphertext elements are decomposed during relinearization	
$\log w$		`decomposition_bit_count`
ℓ	There are $\ell + 1 = \lfloor \log_w q \rfloor + 1$ elements in each component of each evaluation key	
δ	Expansion factor in the ring R ($\delta \leq n$)	
Δ	Quotient on division of q by t, or $\lfloor q/t \rfloor$	
$r_t(q)$	Remainder on division of q by t, i.e. $q = \Delta t + r_t(q)$, where $0 \leq r_t(q) < t$	
χ	Error distribution (a truncated discrete Gaussian distribution)	
σ	Standard deviation of χ	`noise_standard_deviation`
B	Bound on the distribution χ	`noise_max_deviation`

Table 1: Notation used throughout this document.

As described in [20], the FV scheme consists of the following algorithms: `SecretKeyGen`, `PublicKeyGen`, `EvaluateKeyGen`, `Encrypt`, `Decrypt`, `Add`, `Mul`, and `Relin` (version 1). In SEAL we generalize the scheme a little bit, as will be discussed below.

3.1 Plaintext Space and Encodings

In FV the plaintext space is the polynomial quotient ring $R_t = \mathbb{Z}_t[x]/(x^n + 1)$. The homomorphic addition and multiplication operations on ciphertexts (that will be described later) will carry through the encryption to addition and multiplications operations in R_t. Plaintext polynomials are represented by instances of the `BigPoly` class in SEAL. In order to encrypt integers or rational numbers, one needs to encode them into elements of R_t. SEAL provides a few different encoders for this purpose (see Section 5).

3.2 Ciphertext Space

Ciphertexts in FV are vectors of polynomials in R_q. These vectors contain at least two polynomials, but grow in size in homomorphic multiplication operations, unless relinearization is performed. Homomorphic additions are performed by computing a component-wise sum of these vectors; homomorphic multiplications are slightly more complicated and will be described below. Ciphertexts are represented by instances of the `BigPolyArray` class in SEAL.
Textbook-FV only allows ciphertexts of size 2, resulting in minor changes to the homomorphic operations compared to their original description in [20]. We will describe below the algorithms that are implemented in SEAL.

3.3 Encryption and Decryption

Ciphertexts in SEAL are encrypted exactly as described in [20]. A SEAL ciphertext $\mathtt{ct} = (c_0, \ldots, c_k)$ is decrypted by computing

$$\left[\left\lfloor \frac{t}{q}[\mathtt{ct}(s)]_q \right\rceil\right]_t = \left[\left\lfloor \frac{t}{q}\left[c_0 + \cdots + c_k s^k\right]_q \right\rceil\right]_t .$$

Encryption are decryption are implemented in SEAL by the `Encryptor` and `Decryptor` classes, respectively.

3.4 Addition

Suppose two SEAL ciphertexts $\mathtt{ct}_1 = (c_0, \ldots, c_j)$ and $\mathtt{ct}_2 = (d_0, \ldots d_k)$ encrypt plaintext polynomials m_1 and m_2, respectively. Suppose WLOG $j \leq k$. Then

$$\mathtt{ct}_{\text{add}} = ([c_0 + d_0]_q, \ldots, [c_j + d_j]_q, d_{j+1}, \ldots, d_k)$$

encrypts $[m_1 + m_2]_t$.
In SEAL homomorphic addition is implemented as `Evaluator::add`. Similarly, homomorphic subtraction is implemented as `Evaluator::sub`.

3.5 Multiplication

Let $\mathtt{ct}_1 = (c_0, c_1, \ldots, c_j)$ and $\mathtt{ct}_2 = (d_0, d_1, \ldots, d_k)$ be two SEAL ciphertexts of sizes $j+1$ and $k+1$, respectively. The output of $\mathtt{Mul}(\mathtt{ct}_1, \mathtt{ct}_2)$ is a ciphertext $\mathtt{ct}_{\text{mult}} = (C_0, C_1, \ldots, C_{j+k})$ of size $j + k + 1$. The polynomials $C_m \in R_q$ are computed as

$$C_m = \left[\left\lfloor \frac{t}{q}\left(\sum_{r+s=m} c_r d_s\right) \right\rceil\right]_q .$$

In SEAL we define the function `Mul` (or rather family of functions) to mean this generalization of the Textbook-FV multiplication operation (without relinearization). It is implemented as `Evaluator::multiply`.

Algorithms for polynomial multiplication. Multiplication of polynomials in $\mathbb{Z}[x]/(x^n+1)$ is the most computationally expensive part of `Mul`, which in SEAL we implement using Nussbaumer convolution [16]. Note that here polynomial multiplication needs to be performed with integer coefficients, whereas in other homomorphic operations it is done modulo q, which is significantly easier, and can always be done more efficiently using the Number Theoretic Transform (NTT).
It is also possible to implement a Karatsuba-like trick to reduce the number of calls to Nussbaumer convolution, reducing the number of polynomial multiplications to multiply two ciphertexts of sizes k_1 and k_2 from k_1k_2 to ck_1k_2, where $c \in (0,1)$ is some constant depending on k_1 and k_2. For example, if $k_1 = k_2 = 2$, then $c = 3/4$, which is currently the only case implemented in SEAL.

3.6 Relinearization

The goal of relinearization is to decrease the size of the ciphertext back to (at least) 2 after it has been increased by multiplications as was described in Section 3.5. In other words, given a size $k+1$ ciphertext $(c_0, \dots, c_k)$ that can be decrypted as was shown in Section 3.3, relinearization is supposed to produce a ciphertext $(c'_0, \dots, c'_{k-1})$ of size k, or—when applied repeatedly—of any size at least 2, that can be decrypted using a smaller degree decryption function to yield the same result. This conversion will require a so-called *evaluation key* (or *keys*) to be given to the evaluator, as we will explain below.
Let w denote a power of 2, and let $\ell+1 = \lfloor \log_w q \rfloor + 1$ denote the number of terms in the decomposition into base w of an integer in base q. We will also decompose polynomials in R_q into base-w components coefficient-wise, resulting in $\ell+1$ polynomials. Now consider the `EvaluateKeyGen` (version 1) algorithm in [20], which for every $i \in \{0, \dots, \ell\}$ samples $a_i \overset{\$}{\leftarrow} R_q$, $e_i \leftarrow \chi$, and outputs the vector

$$\texttt{evk}_2 = \left[\left([-(a_0 s + e_0) + w^0 s^2]_q, a_0 \right), \dots, \left([-(a_\ell s + e_\ell) + w^\ell s^2]_q, a_\ell \right) \right] .$$

In SEAL we generalize this to j-power evaluation keys by sampling several a_i and e_i as above, and setting instead

$$\texttt{evk}_j = \left[\left([-(a_0 s + e_0) + w^0 s^j]_q, a_0 \right), \dots, \left([-(a_\ell s + e_\ell) + w^\ell s^j]_q, a_\ell \right) \right] .$$

Suppose we have a set of evaluation keys $\texttt{evk}_2, \dots, \texttt{evk}_k$. Then relinearization converts $(c_0, c_1, \dots, c_k)$ into $(c'_0, c'_1, \dots, c'_{k-1})$, where

$$c'_0 = c_0 + \sum_{i=0}^{\ell} \texttt{evk}_k[i][0] c_k^{(i)}, \quad c'_1 = c_1 + \sum_{i=0}^{\ell} \texttt{evk}_k[i][1] c_k^{(i)} ,$$

and $c'_j = c_j$ for $2 \le j \le k-1$.
Note that in order to generate evaluation keys access to the secret key is needed. This means that the owner of the secret key must generate an appropriate number of evaluation keys and share them with the evaluating party in advance of the relinearization computation, which further

means that the evaluating party needs to inform the owner of the secret key beforehand whether or not they intend to relinearize, and if so, by how many steps. Note that if they choose to relinearize after every multiplication, only $\texttt{evk}_2$ will be needed. SEAL implements the above operation as `Evaluator::relinearize`.

3.7 Other Homomorphic Operations

In addition to the operations described above, SEAL implements a few other useful operations, such as negation (`Evaluator::negate`), multiplication by a plaintext polynomial (`Evaluator::multiply_plain`), addition (`Evaluator::add_plain`) and subtraction (`Evaluator::sub_plain`) of a plaintext polynomial, noise-optimal product of several ciphertexts (`Evaluator::multiply_many`), exponentiation with relinearization at every step (`Evaluator:exponentiate`), and a sum of several ciphertexts (`Evaluator::add_many`).
SEAL has a fast algorithm for computing the product of a ciphertext with itself. The difference is only in computational complexity, and the noise growth behavior is the same as in calling `Evaluator::multiply` with a repeated input parameter. This is implemented as `Evaluator::square`.

3.8 Key Distribution

In Textbook-FV the secret key is a polynomial sampled uniformly from R_2, i.e. it is a polynomial with coefficients in $\{0, 1\}$. In SEAL we instead sample the key uniformly from R_3, i.e. we use coefficients $\{-1, 0, 1\}$.

4 Encryption Parameters

Everything in SEAL starts with the construction of an instance of a container that holds the encryption parameters (`EncryptionParameters`). These parameters are:

- `poly_modulus`: a polynomial $x^n + 1$;
- `coeff_modulus`: an integer modulus q;
- `plain_modulus`: an integer modulus t;
- `noise_standard_deviation`: a standard deviation σ;
- `noise_max_deviation`: a bound for the error distribution B;
- `decomposition_bit_count`: the logarithm $\log w$ of w (Section 3.6);
- `random_generator`: a source of randomness.

Some of these parameters are optional, e.g. if the user does not specify σ or B they will be set to default values. If the the decomposition bit count is not set (to a non-zero value), SEAL will assume that no relinearization is going to be performed, and prevents the creation of any evaluation keys. If no randomness source is given, SEAL will automatically use `std::random_device`.
In this section we will describe the encryption parameters and their impact on performance. We will discuss security in Section 7. In Section 4.4 we will discuss the automatic parameter selection tools in SEAL, which can assist the user in determining (close to) optimal encryption parameters for many types of computations.

4.1 Default Values

The constructor of `EncryptionParameters` sets the values for σ and B by default to the ones returned by the static functions

`ChooserEvaluator::default_noise_standard_deviation()`, and
`ChooserEvaluator::default_noise_max_deviation()` .

Currently these default values are set to 3.19 and 15.95, respectively. As we also mentioned above, unless they want to use relinearization, the user does not need to set `decomposition_bit_count`. By default the constructor will set its value to zero, which will prevent the construction of evaluation keys.
SEAL comes with a list of pairs (n, q) that are returned by the static function

`ChooserEvaluator::default_parameter_options()`

as a keyed list (`std::map`). The default (n, q) pairs are presented in Table 2.

n	q
1024	$2^{35} - 2^{14} + 2^{11} + 1$
2048	$2^{60} - 2^{14} + 1$
4096	$2^{116} - 2^{18} + 1$
8192	$2^{226} - 2^{26} + 1$
16384	$2^{435} - 2^{33} + 1$

Table 2: Default pairs (n, q).

4.2 Polynomial Modulus

The polynomial modulus (`poly_modulus`) is required to be a polynomial of the form $x^n + 1$, where n is a power of 2. This is both for security and performance reasons (see Section 7).
Using a larger n decreases performance. On the other hand, it allows for a larger q to be used without decreasing the security level, which in turn increases the noise ceiling and thus allows for larger t to be used. A large value of t allows the scheme to support larger integer arithmetic. When CRT batching is used (Section 5.3), a larger n will allow for more elements of $\mathbb{Z}_t$ to be batched into one plaintext.

4.3 Coefficient Modulus and Plaintext Modulus

Suppose the polynomial modulus is held fixed. Then the choice of the coefficient modulus q affects two things: the upper bound on the inherent

noise that a ciphertext can contain[4] (see Section 6), and the security level[5] (see Section 7.2 and references therein).
In principle we can take q to be any integer, but taking q to be of special form provides performance benefits. First, if q is of the form $2^A - B$, where B is an integer of small absolute value, then modular reduction modulo q can be sped up, yielding overall better performance.
Second, if q is a prime with $2n|(q-1)$, then SEAL can use the Number Theoretic Transform (NTT) for polynomial multiplications, resulting in huge performance benefits in encryption, relinearization and decryption. SEAL uses David Harvey's algorithm for NTT, as described in [27], which additionally requires that $4q \leq \beta$, where β denotes the *word size* of q:

$$\beta = 2^{64\lceil \log(q)/64 \rceil} .$$

Third, if $t|(q-1)$ (i.e. $r_t(q) = 1$), then the noise growth properties are improved in certain homomorphic operations (recall Table 3).
The default parameters in Table 2 satisfy all of these guidelines. They are prime numbers of the form $2^A - B$ where B is much smaller than 2^A. They are congruent to 1 modulo $2n$, and not too close to the word size boundary. Finally, $r_t(q) = 1$ for t that are reasonably large powers of 2, for example the default parameters for $n = 4096$ provide good performance when t is a power of 2 up to 2^{18}.
We note that when using CRT batching (see Section 5.3) it will not be possible to have t be a power of 2, as t needs to instead be a prime of a particular form. In this case the user can try to choose the entire triple (n, q, t) simultaneously, so that $t = 1 \pmod{2n}$ and q satisfies as many of the good properties listed above as possible.

4.4 Automatic Parameter Selection

To assist the user in choosing parameters for a specific computation, SEAL provides an automatic parameter selection tool. It consists of two parts: a `Simulator` component that simulates noise growth in homomorphic operations using the estimates of Table 3, and a `Chooser` component, which estimates the growth of the coefficients in the underlying plaintext polynomials, and uses `Simulator` to simulate noise growth. `Chooser` also provides tools for computing an optimized parameter set once it knows what kind of computation the user wishes to perform.

5 Encoding

One of the most important aspects in making homomorphic encryption practical and useful is in using an appropriate *encoder* for the task at hand. Recall that plaintext elements in the FV scheme are polynomials in R_t. In typical applications of homomorphic encryption, the user would instead want to perform computations on integers or rational numbers.

[4] Bigger q means higher noise bound (good).
[5] Bigger q means lower security (bad).

Encoders are responsible for converting the user's inputs to polynomials in R_t by applying an encoding map. In order for the operations on ciphertexts to reflect the operations on the inputs, the encoding and decoding maps need to respect addition and multiplication.

5.1 Integer Encoder

In SEAL the *integer encoder* is used to encode integers into plaintext polynomials. Despite its name, the integer encoder is really a *family* of encoders, one for each integer base $\beta \geq 2$.
When $\beta = 2$, the idea of the integer encoder is to encode an integer a in the range $[-(2^n - 1), 2^n - 1]$ as follows. It forms the (up to n-bit) binary expansion of $|a|$, say $a_{n-1} \dots a_1 a_0$, and outputs the polynomial

$$\texttt{IntegerEncode}(a, \beta = 2) = \mathrm{sign}(a) \cdot \left(a_{n-1}x^{n-1} + \ldots + a_1 x + a_0\right) .$$

Decoding (`IntegerDecode`) amounts to evaluating a plaintext polynomial at $x = 2$. It is clear that in good conditions (see below) the integer encoder respects addition and multiplication:

$$\texttt{IntegerDecode}\left[\texttt{IntegerEncode}(a) + \texttt{IntegerEncode}(b)\right] = a + b \,,$$
$$\texttt{IntegerDecode}\left[\texttt{IntegerEncode}(a) \cdot \texttt{IntegerEncode}(b)\right] = ab \,.$$

When β is set to some integer larger than 2, instead of a binary expansion (as was done in the example above) a base-β expansion is used. SEAL uses a *balanced* base-β representation to keep the absolute values of the coefficients as small as possible [19].
Note that the infinity norm of a freshly encoded plaintext polynomial is bounded by $\beta/2$, and the degree of the polynomial encoding a is bounded by $\lceil \log_\beta(|a|) \rceil$. However, as homomorphic operations are performed on the encryptions, the infinity norm and degree will both grow. When the degree becomes greater than or equal to n, or the infinity norm greater than $t/2$, the polynomial will "wrap around" in R_t, yielding an incorrect result. In order to get the correct result, one needs to choose n and t to accommodate the largest plaintext polynomial appearing during the computation. For a very nice estimate on how large n and t need to be, we refer the reader to [15].
The integer encoder is available in SEAL through the `IntegerEncoder` class. Its constructor will require both the `plain_modulus` and the base β as parameters. If no base is given, the default value $\beta = 2$ is used.

5.2 Fractional Encoder

There are several ways for encoding rational numbers in SEAL. One way is to simply scale all rational numbers to integers, encode them using the integer encoder described above, and record the scaling factor in the clear as a part of the ciphertext. We then need to keep track of the scaling during computations, which results in some inefficiency. Here we describe what we call the *fractional encoder*, which has the benefit of automatically keeping track of the scaling. Just like the integer encoder,

the fractional encoder is really a family of encoders, parametrized by an integer base $\beta \geq 2$. The function of this base is exactly the same as in the integer encoder, and we will only explain how the fractional encoder works when $\beta = 2$.
Consider the rational number 5.8125, with the finite binary expansion

$$5.875 = 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4} .$$

First we take the integer part and encode it as usual with the integer encoder, obtaining the polynomial $\texttt{IntegerEncode}(5, \beta = 2) = x^2 + 1$. Then we take the fractional part, add n (degree of the polynomial modulus) to each exponent, and convert it into a polynomial by changing the base 2 into the variable x. Finally we flip the signs of each of the terms, in this case obtaining $-x^{n-1} - x^{n-2} - x^{n-4}$. This defines $\texttt{FracEncode}(r, \beta = 2)$ for rational numbers $r \in [0, 1)$. For any rational number r with a finite binary expansion, we set

$$\begin{aligned}\texttt{FracEncode}(r, \beta = 2) = \operatorname{sign}(r) \cdot [&\texttt{IntegerEncode}(\lfloor |r| \rfloor, \beta = 2) \\ &+ \texttt{FracEncode}(\{|r|\}, \beta = 2)] \ ,\end{aligned}$$

where the fractional part is denoted by $\{\cdot\}$. Concluding our example, $\texttt{FracEncode}(5.8125, \beta = 2)$ yields the polynomial $-x^{n-1} - x^{n-2} - x^{n-4} + x^2 + 1$. Decoding works by reversing the steps described above. It is easy to see that `FracEncode` respects both addition and multiplication [19].
The fractional encoder is implemented by the class `FractionalEncoder`. Its constructor will take as parameters the `plain_modulus`, the base β, and positive integers n_f and n_i with $n_f + n_i \leq n$, which describe how many coefficients are reserved for the fractional and integer parts, respectively.[6] If no base is given, the default value $\beta = 2$ is used.

Comparing the two fractional encoding approaches. The *scale-to-integer* technique mentioned above, and our fractional encoder, have similar performance and limitations, but are not equivalent. In some cases the fractional encoder is strictly better.
For example, suppose the homomorphic operations result in some cancellations in the underlying plaintext. Since the level of a scaled encoder never drops, it does not recognize this cancellation, and once the level reaches its maximum (n coefficients), decoding will fail. For the fractional encoder, however, cancellations take care of themselves, permitting potentially more homomorphic operations. As a concrete example, consider $n = 8$, base $\beta = 2$, and the computation $(12 \cdot 0.25)^3$. With the scale-to-integer technique, a rational number $a/2^i$ is encoded as $(p(x), i)$, where $p(x)$ is an integer encoding of a. Hence, the inputs are encoded as $(x^3 + x^2, 0)$, and $(0, 2)$. The result of the computation is $(3x^7 + x^6 - x - 3, 6)$, which does not decode to the correct result since the first entry wrapped around $x^n + 1$. On the other hand, with the fractional encoder, the two inputs are encoded as $x^3 + x^2$ and $-x^6$, and the resulting plaintext polynomial is equal to $(x + 1)^3$, which decodes correctly.

[6] More precisely, n_f describes how many coefficients are used when truncating possibly infinite base-β expansions of rational numbers.

Remark 1. In [15] the authors claimed that the two fractional encoding methods above are equivalent, by claiming the existence of an isomorphism between the underlying rings. We would like to point out that their object R_1 does not satisfy the distribution law, hence is not a ring. This was likely an innocent typo (indeed, with a sign mistake fixed R_1 does become a ring), but even then the map $\phi : R_1 \to R_2$ in their paper is only a surjective homomorphism, and not injective, due to the fact that encoding is not unique: e.g. (x^i, i) encodes the integer 1 for all i.

5.3 CRT Batching

The *CRT (Chinese Remainder Theorem) batching* technique allows up to n integers modulo t to be packed into one plaintext polynomial, and operating on those integers in a *SIMD (Single Instruction, Multiple Data)* manner. For more details and applications we refer the reader to [11, 36, 19].
Batching provides the maximal number of plaintext slots when the plaintext modulus t is chosen to be a prime number and congruent to 1 $(\mathrm{mod}\ 2n)$, which we assume to be the case. Then there exists (see e.g. [19]) a ring isomorphism $\mathtt{Decompose} : R_t \to \prod_{i=0}^{n-1} \mathbb{Z}_t$, whose inverse we denote by `Compose`. In SEAL, `Compose` and `Decompose` are computed using a negacyclic variant of the Number Theoretic Transform (NTT).
When used correctly, batching can provide an enormous performance improvement over the other encoders. Note, however, that for computations on encrypted integers rather than on integers modulo t one needs to ensure that the values in the individual *slots* never wrap around t during the computation.
SEAL provides all of the batching-related tools in the `PolyCRTBuilder` class.

6 Inherent Noise

Definition 1 (Inherent noise). *Let* $\mathtt{ct} = (c_0, c_1, \ldots, c_k)$ *be a ciphertext encrypting the message* $m \in R_t$. *Its inherent noise is the unique polynomial* $v \in R$ *with smallest infinity norm such that*

$$\mathtt{ct}(s) = c_0 + c_1 s + \cdots + c_k s^k = \Delta m + v + aq$$

for some polynomial a.

It is proved in [20], that the function (or family of functions) `Decrypt`, as presented in Section 3.3, correctly decrypts a ciphertext as long as the inherent noise satisfies $\|v\| < \Delta/2$.

6.1 Overview of Noise Growth

We present in Table 3 probabilistic estimates of noise growth in some of the most common homomorphic operations. Even though these are estimates, they are simple and work well in practice. For input ciphertexts $\mathtt{ct}_i$ we denote their respective inherent noises by v_i. When there is a single encrypted input $\mathtt{ct}$ we denote its inherent noise by v.

Operation	Input description	Estimated output noise
`Encrypt`	Plaintext $m \in R_t$	$2B\sqrt{2n/3}$
`Negate`	Ciphertext `ct`	$\|v\|$
`Add/Sub`	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$	$\|v_1\| + \|v_2\| + r_t(q)$
`AddPlain/` `SubPlain`	Ciphertext `ct` and plaintext m	$\|v\| + r_t(q)$
`MultiplyPlain`	Ciphertext `ct` and plaintext m with N non-zero coefficients	$N\|m\| \left(\|v\| + r_t(q)/2\right)$
`Multiply` (with integer encoders)	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$ of sizes $j_1 + 1$ and $j_2 + 1$	$t\left(\|v_1\| + \|v_2\| + r_t(q)\right) \times \left\lceil\sqrt{2n/3}\right\rceil^{j_1+j_2-1} 2^{j_1+j_2}$
`Multiply` (with `PolyCRTBuilder`)	Ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$ of sizes $j_1 + 1$ and $j_2 + 1$	$nt\left(\|v_1\| + \|v_2\| + r_t(q)\right) \times \left\lceil\sqrt{2n/3}\right\rceil^{j_1+j_2-1} 2^{j_1+j_2}$
`Square`	Ciphertext `ct` of size j	Same as $\texttt{Multiply}(\texttt{ct}, \texttt{ct})$
`Relinearize`	Ciphertext `ct` of size K and target size $L < K$	$\|v\| + (K - L)\sqrt{n}B(\ell + 1)w$

Table 3: Noise estimates for homomorphic operations in SEAL.

6.2 Maximal Levels for Default Parameters

In Table 4 we give the maximal supported levels for various power-of-2 plaintext moduli, only taking the noise growth into account. The coefficient moduli are chosen to be the defaults, given in Table 2. We chose to use a uniformly random polynomial in R_t as the plaintext.

n	$\log_2 q$	$\log_2 t$	Max. level
2^{10}	35	6	1
2^{11}	60	7	2
		16	1
2^{12}	116	1	6
		8	4
		20	2
2^{13}	226	8	8
		20	5
		30	3
2^{14}	435	8	15
		32	7
		64	4

Table 4: Maximal levels for different choices of polynomial modulus and plaintext modulus.

7 Security of FV

7.1 Ring-Learning With Errors

The security of the FV encryption scheme is based on the apparent hardness of the famous *Ring-Learning with Errors (RLWE)* problem [30]. Each RLWE sample can be used to extract n *Learning with Errors (LWE)* samples [34, 28]. The concrete hardness depends on the parameters n, q, and the standard deviation of the error distribution σ.

7.2 Security of the Default Parameters in SEAL v2.1

We now give an estimate of the security of the default parameters in SEAL v2.1 based on the LWE estimator of [7].[7] The estimator takes as input an LWE instance given by a dimension n, a modulus q, and a *relative error* $\alpha = \sqrt{2\pi}\sigma/q$. For various attacks it returns estimates for the number of bit operations, memory, and number of samples required to break the LWE instance. In Table 5 we give the expected number of bit operations required to attack the LWE instances induced by the SEAL v2.1 default parameters, assuming that the attacker has as many samples, and as much memory, as they would require. Recall from Section 4.1 that in SEAL the default standard deviation is $\sigma = 3.19$, so we always have $\alpha q = \sigma\sqrt{2\pi} \approx 8$, and we use $\alpha = 8/q$. We use the default n and q as presented in Table 2.

n	q	α	small sis	bkw	sis	dec	kannan
1024	$2^{35} - 2^{14} + 2^{11} + 1$	$8/q$	97.6	237.4	126.5	116.1	116.6
2048	$2^{60} - 2^{14} + 1$	$8/q$	115.1	391.2	136.2	129.0	129.5
4096	$2^{116} - 2^{18} + 1$	$8/q$	119.1	615.3	132.7	128.2	129.2
8192	$2^{226} - 2^{26} + 1$	$8/q$	123.1	1168.6	132.2	—	131.1
16384	$2^{435} - 2^{33} + 1$	$8/q$	130.5	1783.5	134.4	—	135.9

Table 5: Estimates of log *of the bit operations required to perform the above named attacks on the SEAL v2.1 default parameters. The symbol '—' denotes that the estimator did not return a result.*

Recently, Albrecht [3] described new attacks on LWE instances where the secret is very small, and presented estimates of the cost of these attacks on the default parameters used in SEAL v2.0. Estimates for cost of the attacks described in [3] have been included into the LWE estimator of [7]. In Table 5 we have included the attack presented in [3, Sections 3 and 4], labelled 'small sis', which performs best against the SEAL v2.1 parameters. To label the other attacks we follow the notation of [7]: 'bkw' denotes a variant [23] of the BKW attack [10, 5], 'sis' denotes a distinguishing attack as described in [32]; 'dec' denotes a decoding attack as described in e.g. [29]; 'kannan' denotes the attack described in [6]. The

[7] We used the version available on February 23rd, 2017 (commit `d70e1e9`).

estimator was not run for Arora-Ge type attacks [8, 4] or for meet-in-the-middle type attacks, since these are both expected to be very costly.

Remark 2. At the time of writing this, determining the concrete hardness of parametrizations of (R)LWE is an active area of research (see e.g. [17, 13, 7]), and no standardized (R)LWE parameter sets exist. Therefore, when using SEAL or any other implementation of (R)LWE-based cryptography, we strongly recommend the user to consult experts in the security of (R)LWE when choosing which parameters to use.

References

[1] FV-NFLlib. `https://github.com/CryptoExperts/FV-NFLlib`. Accessed: 2017-02-17.

[2] HElib. `https://github.com/shaih/HElib`. Accessed: 2016-11-21.

[3] Martin R. Albrecht. On dual lattice attacks against small-secret LWE and parameter choices in HElib and SEAL. Cryptology ePrint Archive, Report 2017/047, 2017. `http://eprint.iacr.org/2017/047`.

[4] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. Algebraic algorithms for LWE problems. *IACR Cryptology ePrint Archive*, 2014:1018, 2014.

[5] Martin R. Albrecht, Carlos Cid, Jean-Charles Faugère, Robert Fitzpatrick, and Ludovic Perret. On the complexity of the BKW algorithm on LWE. *Des. Codes Cryptography*, 74(2):325–354, 2015.

[6] Martin R. Albrecht, Robert Fitzpatrick, and Florian Göpfert. On the efficacy of solving LWE by reduction to unique-svp. In Hyang-Sook Lee and Dong-Guk Han, editors, *Information Security and Cryptology - ICISC 2013 - 16th International Conference, Seoul, Korea, November 27-29, 2013, Revised Selected Papers*, volume 8565 of *Lecture Notes in Computer Science*, pages 293–310. Springer, 2013.

[7] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3):169–203, 2015.

[8] Sanjeev Arora and Rong Ge. New algorithms for learning in presence of errors. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part I*, volume 6755 of *Lecture Notes in Computer Science*, pages 403–415. Springer, 2011.

[9] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. Cryptology ePrint Archive, Report 2016/510, 2016. `http://eprint.iacr.org/2016/510`.

[10] Avrim Blum, Adam Kalai, and Hal Wasserman. Noise-tolerant learning, the parity problem, and the statistical query model. *J. ACM*, 50(4):506–519, 2003.

[11] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *Public-Key Cryptography–PKC 2013*, pages 1–13. Springer, 2013.

[12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325. ACM, 2012.

[13] Johannes A. Buchmann, Niklas Büscher, Florian Göpfert, Stefan Katzenbeisser, Juliane Krämer, Daniele Micciancio, Sander Siim, Christine van Vredendaal, and Michael Walter. Creating cryptographic challenges using multi-party computation: The LWE challenge. In Keita Emura, Goichiro Hanaoka, and Rui Zhang, editors, *Proceedings of the 3rd ACM International Workshop on ASIA Public-Key Cryptography, AsiaPKC@AsiaCCS, Xi'an, China, May 30 - June 03, 2016*, pages 11–20. ACM, 2016.

[14] Ana Costache and Nigel P. Smart. Which ring based somewhat homomorphic encryption scheme is best? In Sako [35], pages 325–340.

[15] Anamaria Costache, Nigel P Smart, Srinivas Vivek, and Adrian Waller. Fixed point arithmetic in SHE schemes. Technical report, Cryptology ePrint Archive, Report 2016/250, 2016. http://eprint. iacr. org/2016/250.

[16] Richard Crandall and Carl Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006.

[17] Eric Crockett and Chris Peikert. Challenges for ring-lwe. Cryptology ePrint Archive, Report 2016/782, 2016. http://eprint.iacr.org/2016/782.

[18] Eric Crockett and Chris Peikert. $\Lambda o\lambda$: Functional lattice cryptography. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 993–1005. ACM, 2016.

[19] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Manual for using homomorphic encryption for bioinformatics. Technical report, Microsoft Research, 2015. http://research.microsoft.com/apps/pubs/default.aspx?id=258435.

[20] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/.

[21] Rosario Gennaro and Matthew Robshaw, editors. *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*. Springer, 2015.

[22] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[23] Qian Guo, Thomas Johansson, and Paul Stankovski. Coded-bkw: Solving LWE using lattice codes. In Gennaro and Robshaw [21], pages 23–42.

[24] Shai Halevi and Victor Shoup. Design and Implementation of a Homomorphic-Encryption Library. `http://people.csail.mit.edu/shaih/pubs/he-library.pdf`, 2013.

[25] Shai Halevi and Victor Shoup. Algorithms in helib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014.

[26] Shai Halevi and Victor Shoup. Bootstrapping for helib. In Oswald and Fischlin [33], pages 641–670.

[27] David Harvey. Faster arithmetic for number-theoretic transforms. *Journal of Symbolic Computation*, 60:113–119, 2014.

[28] Tancrède Lepoint and Michael Naehrig. A comparison of the homomorphic encryption schemes fv and yashe. In *Progress in Cryptology–AFRICACRYPT 2014*, pages 318–335. Springer, 2014.

[29] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for lwe-based encryption. In *Topics in Cryptology–CT-RSA 2011*, pages 319–339. Springer, 2011.

[30] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2010.

[31] Carlos Aguilar Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancrède Lepoint. Nfllib: Ntt-based fast lattice library. In Sako [35], pages 341–356.

[32] Daniele Micciancio and Oded Regev. *Lattice-based Cryptography*, pages 147–191. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[33] Elisabeth Oswald and Marc Fischlin, editors. *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*. Springer, 2015.

[34] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[35] Kazue Sako, editor. *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, volume 9610 of *Lecture Notes in Computer Science*. Springer, 2016.

[36] Nigel P Smart and Frederik Vercauteren. Fully homomorphic simd operations. *Designs, codes and cryptography*, 71(1):57–81, 2014.

FEAL Cipher

Team RZC:
Carlos Leonardo
Zachary Miller

Outline

- FEAL block cipher.
- Our Ad-Hoc Attack.
- Results and analysis
- Other Attacks on FEAL
- What we learned
- Future work
- Conclusion
- Questions
- References

FEAL block cipher.

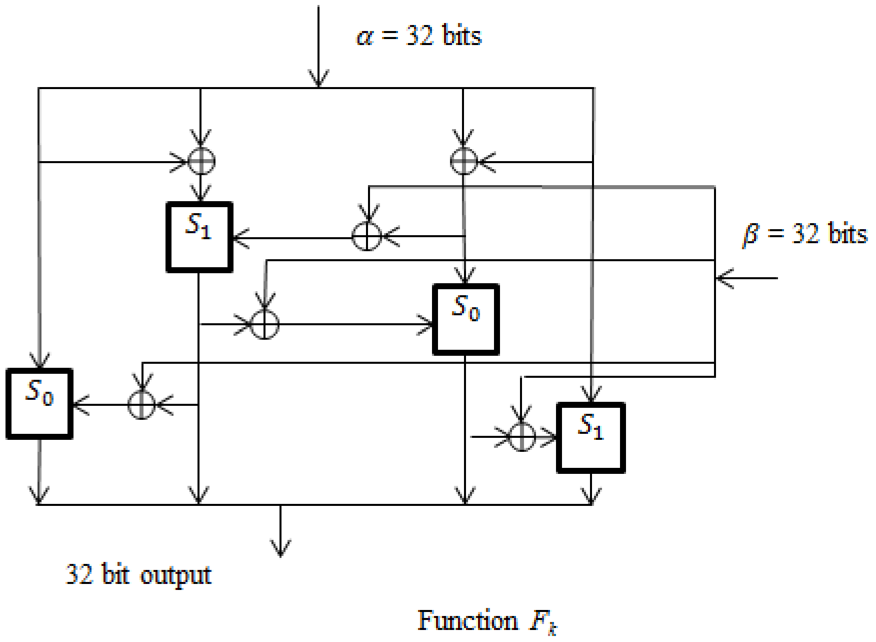
- It uses 64 bits long Key, plaintext and ciphertext [1].
- It was supposed to be as secure as DES without using special hardware [2].
- Various Attacks have proven FEAL insecure[2,3].
- The 1st version used 4 Feistel rounds and later it was extended to 8 rounds [1].

S-Function

- Implements the S-Boxes and accepts three byte parameters and return a single byte:
  - byte A
  - byte B
  - byte δ = 0 or 1
- S(A,B,δ) = RotateL2(T)
- T = A + B + δ **mod** 256

Function F_k

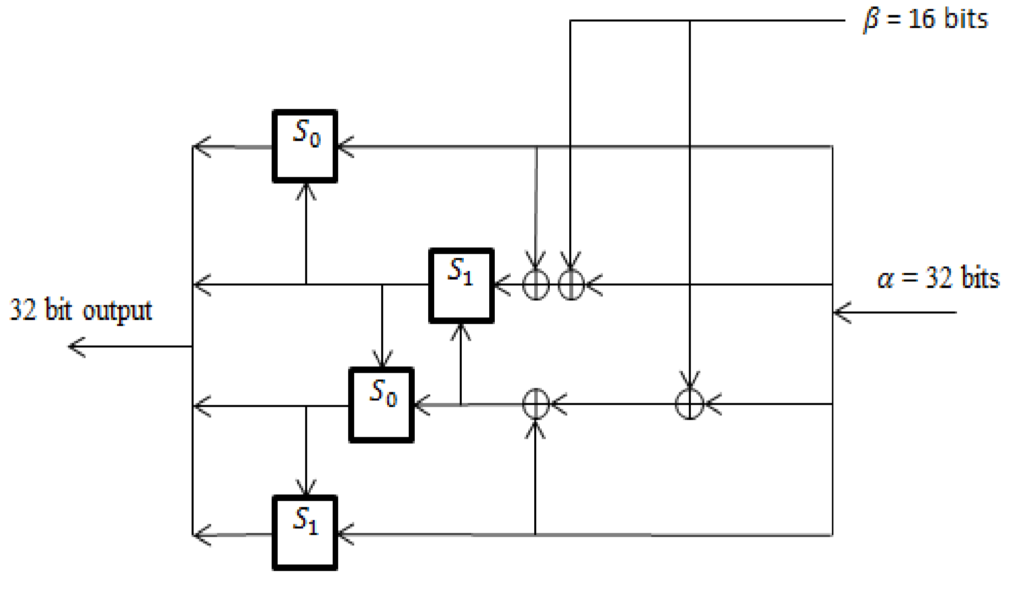
- Used during Sub-Key Generation.
- Parameters: α and β: 32 bit halves of initial key
- Returns two 16-bit Sub-Keys



[1]

Function F

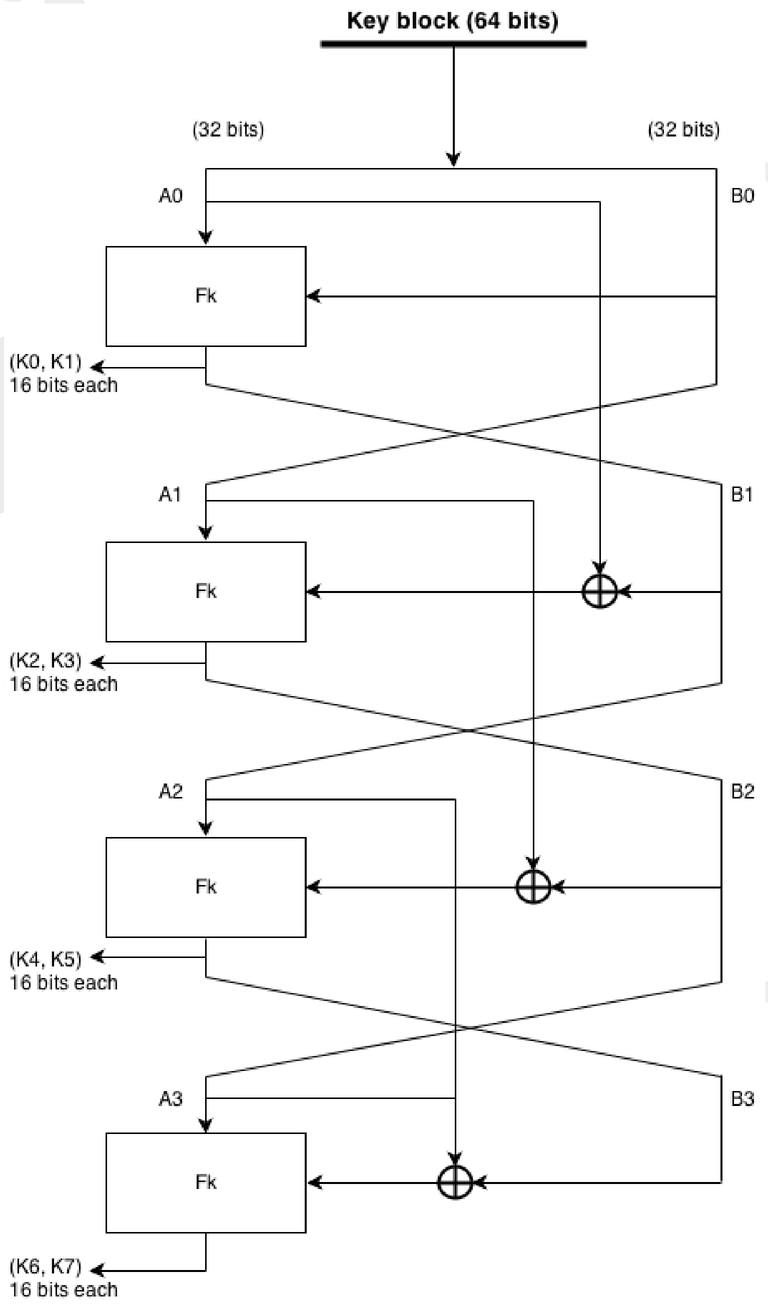
- Used for Encryption
- Parameters:
  - α: 32 bit from Feistel Round
  - β: 16 bit sub-key



Function F

[1]

Key Generation

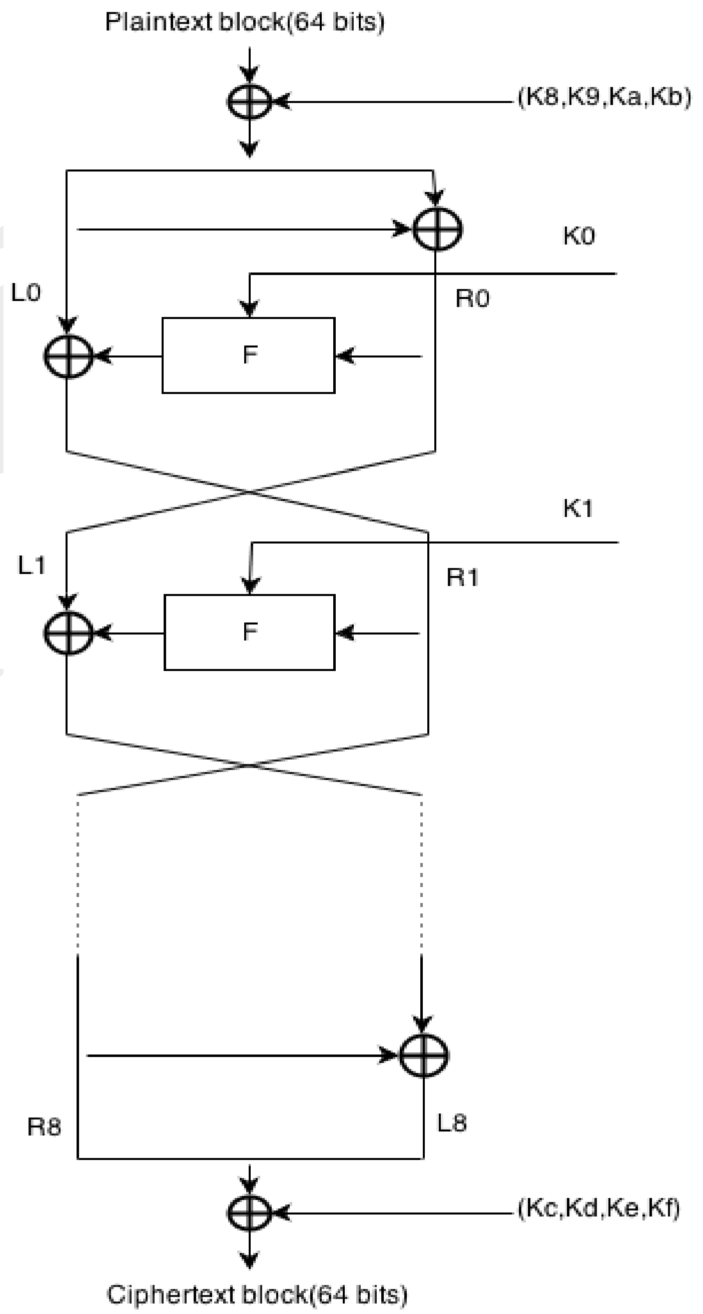


[1]

Encryption

Decryption

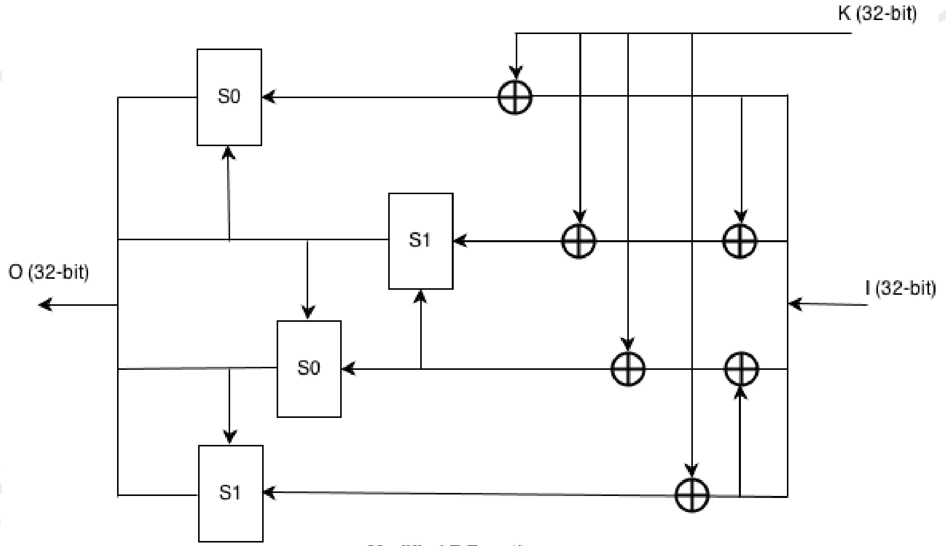
[1]



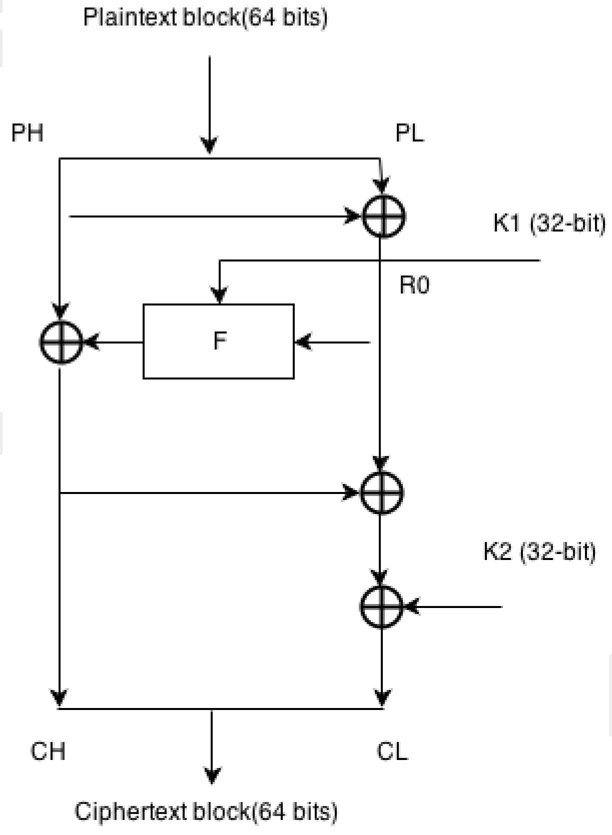
Ad-Hoc Attack on FEAL

- To attach the FEAL Cipher, we first reduced the number of rounds to 1.
- We then modify the F-Function as described in [2].

Modified FEAL Algorithm



Modified F-Function



One Round FEAL

Ad-Hoc Attack

- We can directly derive K2 for any plaintext-ciphertext pair
- To find K1, we set up equations and guess a value for a given byte in the key
- Each guess is independent so we are doing 256 operations 4 times.
- Each guess for a given byte of k1 is called a candidate key

Ad-Hoc Attack Cont.

- Calculate candidate keys for each plaintext-ciphertext pair
- Unfortunately each byte for K1 is not the same
- However, candidate keys occur frequently
- In our observations
  - 4 candidate bytes was the smallest
  - 84 was the largest
- Based on this, we figure:
  - To derive a workable key, find a permutation of the common occurring key bytes(84^4 ~49 million) in worse observed case.
  - Fortunately this is the worse case:
    - Observed much better

Our Ad-Hoc Program

- Takes in 2 parameters
  - n = number of plaintext-ciphertext pairs
  - k = original 64-bit key used to find ciphertexts
- Runs the FEAL Encipherment algorithm on each randomized plaintext
- Derives a value for K2
- Finds candidate keys for each byte of K1
- Repeats on all plaintext-ciphertext pairs
- Outputs the list of all candidate keys for each byte of K1 and the value for K2.

Results

- This actually performed well.
  - The orders of the number of candidate keys for a given byte stayed fairly small(computable).
- Needed between 1,000 and 10,000 message pairs to compute all possible candidate keys
  - We even tried up to 10 million and we were not adding any new keys
- This means that the guessing of the keys takes 10,000(2^8 * 4) or 10,000(2^10) or 10,240,000 operations.
  - Could be less depending on the original 64-bit key

Sample of Results

Original Key	#K1_1	#K1_2	#k1_3	#k1_4	Total Operations
0123456789ABCDEF	54	45	27	60	3936600
AAAAAAAAAAAAAAAA	80	79	24	72	10920960
FEDCBA9876543210	14	46	40	4	103040
5738290BC37D3FEA	80	84	48	56	18063360

Comparison

- [2] Presented a Linear Attack on FEAL-4, FEAL-6 and FEAL-8
  - Needed 5 plaintexts to break FEAL-4 in 6 minutes.
  - Unsure of how many operations.
  - FEAL-6 needed 100 texts and FEAL-8 needed 2^15.
  - Much better than our attack
- [3] Differential Attack which broke FEAL-8 with 1,000 pairs with a 95% accuracy

What We learned

- In general, Cryptography is a difficult subject. Algorithms can be relatively easy to implement, but can be fairly difficult to break given certain circumstances.
- One approach will not solve all of the problems, cryptanalysis often requests to adapt to new situations.
- Split big tasks and keep going.

Future Work

- Running some statistical measurements to gauge how effective this method is.
- Find another place to add a 3rd key for the attack which could increase the likelihood that a given key would work for more plaintext ciphertext pairs.

References

[1] Akihiro Shimizu and Shoji Miyaguchi, *Fast Data Encipherment Algorithm FEAL*. Advances in Cryptography *EUROCRYPT* '87. Amsterdam, 1987. pp. 267-278.

[2]Mitsuru Matsui and Atsuhiro Yamagishi. *A New Method for Known Plaintext Attack of FEAL Cipher*. Advances in Cryptography EUROCRYPT '92. Balatonfured, Hungary, 1992. pp. 81-91.

[3] Eli Biham and Adi Shamir *Differential Cryptanalysis of Feal and N-Hash*. Advances in Cryptography *EUROCRYPT '91*. Brighton UK, 1991. pp.1-16.

Questions?

Blowfish

Sankeeth Kumar Chinta.
# 991730264
Sept 18, 2015

Contents

1 Introduction

In cryptographic circles, plaintext is the message you're trying to transmit. That message could be a medical test report, a firmware upgrade, or anything else that can be represented as a stream of bits. The process of encryption converts that plaintext message into ciphertext, and decryption converts the ciphertext back into plaintext. Encryption algorithms are technically classified in two broad categories- Symmetric key Cryptography and Asymmetric Key Cryptography.

In symmetric type of Cryptography, the key that is used for encryption is same as the key used in decryption. So the key distribution has to be made before transmission of any information. The key plays a very important role in this cryptography since the nature of the key length has an impact on security. Examples of various symmetric key algorithms are Data encryption standard(DES), Triple DES, Advanced Encryption Standard(AES) and Blowfish Encryption Algorithm.

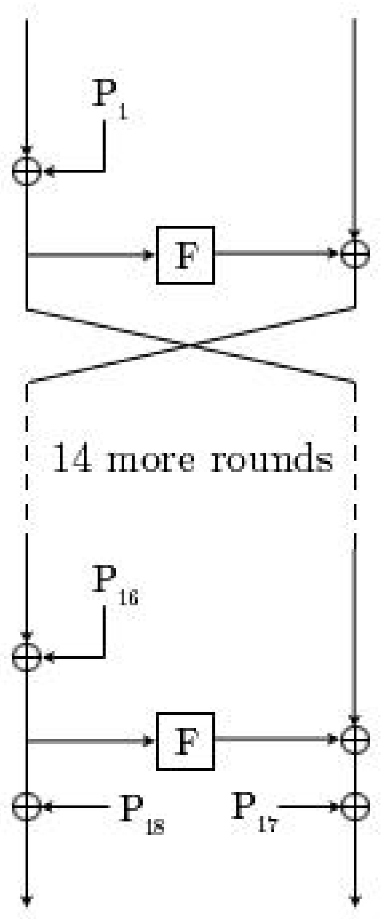
In Asymmetric Cryptography, two unique keys are used for encryption and decryption. One is public and the other one is private. The public key is available to anyone on the network i.e., whoever wants to encrypt the plaintext should know the Public Key of receiver. Decryption of the cipher text through his/her own private key is by authorizes personnel. Private Key is kept secretly from the others. Symmetric Encryption Algorithm is much faster as compared to Asymmetric key algorithm. The memory requirement of Symmetric algorithm is comparatively less than asymmetric algorithm. Eg: RSA

DES (Data Encryption Standard) is the first encryption standard recommended by NIST (National Institute of Standards and Technology). It was developed by IBM in 1974 and in 1977 it was adopted as a national standard.DES is a 64-bit block cipher under 56-bit key. The algorithm computes with a permutation of sixteen rounds block cipher and a final permutation. DES application is popularly used in the commercial, military, and other domains. DES standard is public.

In the year 2000, AES was invented by scientists named Joan and Vincent Rijmen. AES makes use of the Rijndael block cipher, Rijndael key and block length can be of 128, 192 or 256-bits, If both the key-length and block length are 128-bit then Rijndael performs 9 processing rounds. If the block/key is 192-bit, it will perform 11 processing rounds and in the same way for 256-bits, Rijndael performs 13 processing rounds..

Block Cipher is a deterministic algorithm operating on fixed-length groups of bits, called blocks, with an unvarying transformation that is specified by a symmetric key. Block ciphers are important elementary components in the design of many cryptographic protocols. The modern design of block ciphers is based on the concept of an iterated product cipher. Iterated product ciphers carry out encryption in multiple rounds, each of which uses a different sub-key derived from the original key. One widespread implementation of such ciphers is called a Feistel network.

One of the most popular fiestel network cipher is Blowfish. Blowfish is a symmetric-key block cipher proposed as a new encryption standard. It is a 16- round fiestel system, which uses large key-dependent S-boxes and iterates a simple encryption 16 times. The block size is 64 bits and the key-length may varies from 32 bits upto 448bits.Although there is a complex initialization phase required before any encryption can take place, the actual encryption of data is very efficient on large microprocessors. Blowfish provides a good encryption rate in software and no effective cryptanalysis of it has been found to date.



2 History

DES is the factotum of cryptography algorithms which was a long past time to replace the old standard. In DES, the length of the key is too small for today emerging technologies. The world just incompletely trusted DES in light of the fact that it survived the investigation of the NSA. Specialists trusted DES on the grounds that it was a distributed standard, and in light of the fact that it survived 20 years of escalated cryptanalysis by cryptographers around the globe. Cryptography is similar to that: trust in an algorithm develops as many groups tries to break it and fails Contenders for a substitution are rising, yet none has taken far reaching hold. Triple-DES is the moderate methodology; IDEA (utilized as a part of PGP) is the most encouraging new algorithm. What's more, there is a gathering of unpatented additionally rans: RC4 (once a prized formula of RSA Data Security, Inc. be that as it may, now openly accessible on the Internet), SAFER, and Blowfish. In 1993, Scheneir first presented Blowfish at the Cambridge Algorithms Workshop ("Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)," Fast Software Encryption, R. Anderson, ed., Lecture Notes in Computer Science 809, Springer-Verlag, 1994) and in Dr. Dobb's Journal (April 1994). From the start Blowfish was intended to be a completely free–unpatented, unlicensed, and uncopyrighted–alternative to DES. Since then it has been analyzed by some people and has started to see use in some systems, both public and private.

Since 1993, it was analysed considerably and gaining acceptance as a strong encryption algorithm. The first implementation of blowfish was done in LabVIEW.. This was proposed as the world needs a new encryption standard as the workhorse encryption algorithm is near ending of its useful life.

3 Description of Algorithm:

Blowfish symmetric block cipher algorithm encrypts block data of 64-bits at a time. The algorithm follows fiestal network and is divided into 2 main parts:

1. Key-expansion
2. Data Encryption
3. Data Decryption

3.1 Key Expansion

Prior to any data encryption and decryption, these keys should be computed before-hand.

The p-array consists of 18, 32-bit sub-keys:

P1, P2,., P18

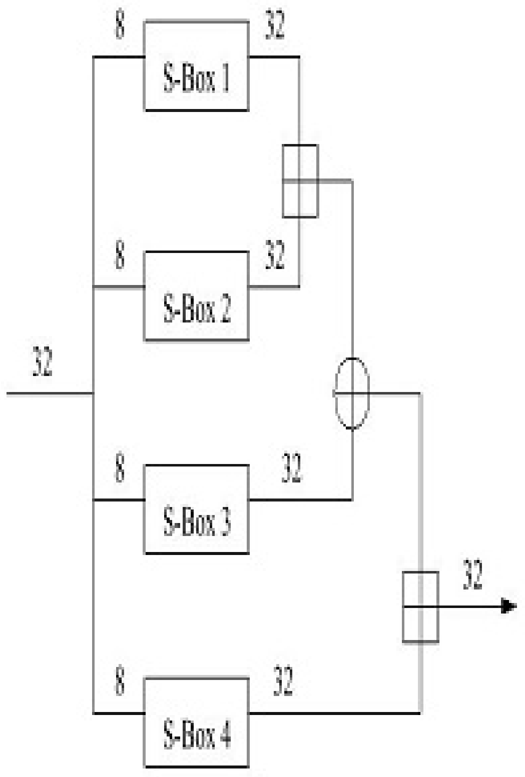
Four 32-bit S-Boxes consist of 256 entries each:

S1, 0, S1, 1,. S1, 255

S2, 0, S2, 1,.. S2, 255

S3, 0, S3, 1,.. S3, 255

S4, 0, S4, 1 ....S4, 255



Generating the Sub-keys:

The sub-keys are calculated and generated using the Blowfish algorithm:

1. Initialize first the P-array and then the four S-boxes, in order, with a fixed string. This string consists of the hexadecimal digits of pi (less the initial 3): P1 = 0x243f6a88, P2 = 0x85a308d3, P3 = 0x13198a2e, P4 = 0x03707344, etc.

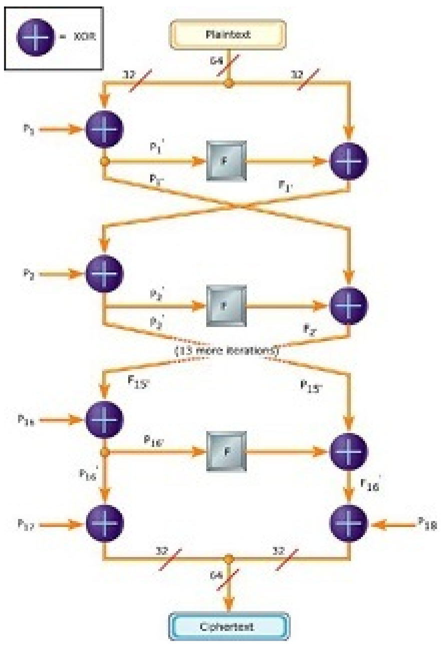
2. XOR P1 with the first 32 bits of the key, XOR P2 with the second 32-bits of the key, and so on for all bits of the key (possibly up to P14). Repeatedly cycle through the key bits until the entire P-array has been XORed with key bits. (For every short key, there is at least one equivalent longer key; for example, if A is a 64-bit key, then AA, AAA, etc., are equivalent keys.)
3. Encrypt the all-zero string with the Blowfish algorithm, using the sub-keys described in steps (1) and (2).
4. Replace P1 and P2 with the output of step (3).
5. Encrypt the output of step (3) using the Blowfish algorithm with the modified sub-keys.
6. Replace P3 and P4 with the output of step (5).
7. Continue the process, replacing all entries of the P array, and then all four S-boxes in order, with the output of the continuously changing Blowfish algorithm.

In total, 521 iterations are required to generate all required sub-keys. Applications can store the sub-keys rather than execute this derivation process multiple times.

3.2 Data Encryption

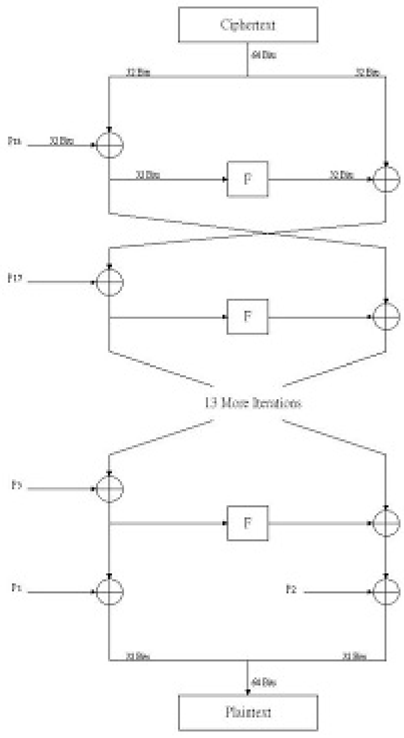
It is having a function to iterate 16 times of network. Each round consists of key-dependent permutation and a key and data-dependent substitution. All operations are XORs and additions on 32-bit words. The only additional operations are four indexed array data lookup tables for each round.

```
Algorithm: Blowfish Encryption
Divide x into two 32-bit halves: xL, xR
For i = 1to 16:
xL = XL XOR Pi
xR = F(XL) XOR xR
Swap XL and xR
Swap XL and xR (Undo the last swap.)
xR = xR XOR P17
xL = xL XOR P18
Recombine xL and xR
```



3.3 Data Decryption

Decryption is exactly the same as encryption, except that P1, P2 ..... P18 are used in the reverse order.



4 Design Decisions

A 64-bit block size yields a 32-bit word size, and maintains block-size compatibility with existing algorithms. Blowfish is easy to scale up to a 128-bit block, and down to smaller block sizes. The fundamental operations were chosen with speed in mind. XOR, ADD, and MOV from a cache are efficient on both Intel and Motorola architectures. All sub-keys fit in the cache of a 80486, 68040, Pentium, and PowerPC. The Feistel Network that makes up the assemblage of Blowfish is intended to be as straightforward as would be prudent, while as yet holding the desirable cryptographic properties of the structure. In calculation outline, there are two fundamental approaches to guarantee that the key is sufficiently long to guarantee a specific security level. One is to painstakingly plan the algorithm so that the whole entropy of the key is safeguarded, so there is no better approach to cryptanalyze the algorithm other than brute force. The other is to plan the algorithm with such a variety of key bits that attacks that diminish the powerful key length by a few bits are unimportant. Since Blowfish is intended for huge microprocessors with a lot of memory, the recent has been picked. Be that as it may, it works just as well on Handheld frameworks with a better than average microprocessors. The sub key generation procedure is intended to save the whole entropy of the key and to convey that entropy consistently all through the Sub-keys. It is likewise intended to appropriate the arrangement of permitted Sub-keys arbitrarily all through the domain of Sub-keys. The digits of pi were picked as the starting Sub-key table for two reasons: on the grounds that it is an random grouping not identified with the algorithm, and in light of the fact that it could either be put away as a major aspect of the algorithm or derived when required. In the Sub-key generation prepare, the Sub-keys change somewhat with each pair of Sub-keys created. This is primarily to protect against any attacked of the Sub-key generation process that exploit the fixed and known Sub-keys. It also reduces storage requirements. The 448 limit on the key size ensures that the every bit of every Sub-key depends on every bit of the key. The key bits are over and over XORed with the digits of pi in the starting P-array to keep the accompanying potential attack: Assume that the key bits are not rehashed, but rather padded with zeros to stretch out it to the length of the P-array. An attacker may discover two keys that vary just in the 64-bit value XORed with P1 and P2 that, utilizing the initial known sub-keys, produce the same encrypted value. Assuming this is the case, he can discover two keys that deliver all the same sub-keys. This is an exceedingly enticing assault for a malignant key generator. To keep this same kind of attack, the starting plain text value in the sub-key generation procedure is fixed. The sub-key generation algorithm does not expect that the key bits are arbitrary. Indeed, even exceptionally related key bits, for example, an alphanumeric ASCII string with the bit of each byte set to 0, will deliver random sub-keys. On the other hand, to create sub-keys with the same entropy, a more longer alphanumeric key is required. The time consuming sub-key generation procedure includes significant unpredictability for a brute- force attack. The sub-keys are too long to be put away on a massive tape, so they would need to be produced by a brute-force breaking machine as required. A total of 522 iterations of the encryption calculation are required to test a single key, successfully adding 29 stages to any brute-force attack.

5 Areas Of Applications

A standard encryption algorithm must be suitable for many different applications:

1. Bulk encryption: The algorithm should be efficient in encrypting data files or a continuous data stream.

2. Random bit generation: The algorithm should be efficient in producing

single random bits.

3. Packet encryption: The algorithm should be efficient in encrypting packet-sized data. (An ATM packet has a 48- byte data field.) It should implementable in an application where successive packets may be encrypted or decrypted with different keys.

4. Hashing: The algorithm should be efficient in being converted to a one-way hash function.

5.1 Products Using Blowfish Algorithm

1. Blowfish Advanced CS by Markus Hahn: File encryption and wipe utility for all Win32 systems. File browser, job automation, auto password confirmation, secure key setup with SHA-1, and data compression with LZSS. Uses Blowfish, Twofish, and Yarrow. Open source.

2. Access Manager by Citi-Software Ltd: A password manager for Windows. Free for personal use.

3. AEdit: A free Windows word processor incorporating text encryption.

6 Conclusion

The proposed algorithm of the Blowfish can achieve an efficiency of data encryption up to 4 bits per clock. We avoid I/O limited constraint by changing the I/O from 64 bits to 16 bits in this design. The proposed architecture ought to fulfill the need of high data encryption and can be connected to different devices separately. Blowfish algorithm, it is a variable-length key block cipher. Applications where there is a strong communication link and where the key will not be changed too often there we will be using the Blowfish algorithm It is quicker than DES. Blowfish is a 16 pass block encryption algorithm that can be never broken. BLOWFISH is used frequently because:

1. It is fast as it encrypts data on large 32-bit microprocessors at a rate of 26 clock cycles per byte.

2. It is compact as it can run in less than 5K of memory.

3. It simply uses addition, XOR, lookup table with 32-bit operands.

4. It is secure as the key length is variable ,it can be in the range of 32 448 bits: default 128 bits key length.