Sliding Block Ciphers*

Ben P. Wise
bwise@oceanofstones.net

May 16, 2006

* This work was not financially supported by any organization.

ABSTRACT: *Sliding Block Ciphers (SB Ciphers, or SBC) are a new class of ciphers which use a highly variable block size, and do so in a unique way to that enables further innovations. They are innovative in that they enable varying numbers of words in a cipher, which in turn enables the new cryptographic primitive of word-permutation and the use of 'sort' as a new cryptographic primitive. The concept of Sliding Block Ciphers was first reduced to practice (functional C++ source code) in December 2002, in a program called circ. A non-open source version of the* `fpwdman` *program using SBC was distributed, and a provisional patent application filed, in 2003, and an open-source C++ reference implementation of several SBC is included in this release.*

1. Introduction

1.1. General Purpose

I invented Sliding Block Ciphers and the two new cryptographic primitives to provide increased security and flexibility in the following applications:

- efficient and secure communication of data by encrypting it one block at a time, while chaining the blocks together (e.g. by CBC, ECB, CFB, etc.),
- efficient and secure communication of data by encrypting it in one large block,
- secure generation of psuedo random numbers in conjunction with an 'entropy pool'
- secure hash functions, with variable digest size, for use in encryption and authentication protocols.

The advantages of SB Ciphers over prior art are the following:

- less need to develop new cryptographic schemes when different sizes of input or output are required
- increased security through the introduction of new, stronger cryptographic primitives: word-permutation, and the application of sorting as a cryptographic primitive.

1.2. General Definition

This section defines the structure of sliding block ciphers, and how they acheive their results.

The most important data structure in a sliding block cipher is the data, stored in K words: X[0], X[1], ... X[K-1]. This can helpfully be visualized as a ring.

The original key is in words Key[0], ... Key[N-1], which is expanded to make a subkey array, S[ ]. Not changing the key at all is a trivial 'expansion'.

Essentially, the encryption proceeds as a double loop. In the main loop, there is first an inner loop of repeated ring-encryptions, followed by one permutation of the words.

The decryption proceeds as a double loop. In the main loop, there is first one de-permutation of the words, followed by an inner loop of repeated ring-decryptions.

For any implementation of an SBC, a word-permutation function must be supplied. This function permutes (but does not modify) all the words of data in the ring; it may depend on the secret key, on the data, the round number, or other factors.

The SB Cipher scheme has not appeared in prior art. For example, RC5 was extended to RC6 in a completely different manner: a fixed set of two words was replaced by a fixed set of four words. Neither can use arbitrary numbers of words.

2. New Features of SBC

Sliding block sizes are a new concept in cipher design. Some block ciphers allow different word sizes to be used in their internal registers, but they have a fixed number of registers. In our terminology, for example, RC6 has K=4. With old computers with 16-bit words, RC6 uses a 64-bit block; with modern computers with 32-bit words, a 128-bit block; with future computers and 64-bit words, a 256-bit block. But in all cases, RC6 still has exactly 4 internal variables (A,B,C,D in Rivest's notation), while a true sliding block cipher can use an arbitrary number of words. Thus, DES, TEA, SkipJack, RC6, BlowFish, TwoFish, Rijndael (aka AES), MARS, and so on can not handle 6400-bit blocks on modern computers, while a true sliding-block cipher would simply have 32-bit words with K=200 (as in the refence implementation data given below).

For example, the 'Scalable Block Ciphers' of Canda and Trung are 'unbalanced Feistel networks', or UFN, with large and variable 'Left' and 'Right' parts. The cipher still has exactly 2 'word' blocks, like DES, and unlike the SBC concept described here. The disadvantage of the UFN approach is that cryptographic primitives must be developed and verified for the different block sizes - and those crytographic primitives are hard to develop.

Further, the UFN approach can not use the 'word permutation' primitive we introduce below, because UFN still have only 2 words in the block.

SB Ciphers incorporate permutation of the block's words in a flexible manner, that can depend on the key, the data in the block's words, the round number, and the block size (or any combination thereof). This 'word permutation' is a new cryptographic primitive not used in any prior art. By introducing a wide range of nonlinear dependencies, with good diffusion and decorrelation, the use of word permutation will substantially increase resistance to linear cryptanalysis, differential cryptanalyis, and algebraic cryptanalysis.

In prior art, simple and fixed permutation were used. Given the small, fixed block sizes, nothing else was possible. Even when the two "words" had variable size, the fact that there were only 2 made word-permutation crytographically useless. Given the fixed structure of 'variable block ciphers' (patented by Ritter), permutation is also not possible.

The fact that neither "scalable block ciphers" nor "variable block ciphers" can support word-permutation demonstrates that Sliding Block Ciphers are a new and innovative design, that enables new and powerful cryptographic primitives to be used (e.g. data dependent word-permutation).

In a classic Feistel cipher (and the numerous 2-words variants: balanced or unbalanced, target heavy or source heavy, etc.), there is only one permutation possible: the simultaneous assignment (A, B) := (B, A). In a four-word block cipher, such as RC6 and other AES candidates, the cyclic permutation (A,B,C,D) := (B,C,D,A) was used. In either case, the permutations did not depend on the key, the round number, or the data, and the permutations never vary.

Ritter's Variable Block Ciphers consist of several layers, which are interconnected in a very specific way. They are diagrammed as four layers, one above the other. While more copies can be appended to the 'right end' to extend the block, no permutation of words between the layers is possible. Indeed, it would render the cipher non-functional. Thus, Ritter's patented way of introducing variably-sized blocks is fundamentally different from the SBC way, as evidenced by the fact that SBC can be used to support word-permutation over the entire ring, while the VBC approach can not.

3. Notations

We use the following notations.

- $\oplus$ to indicate bitwise XOR,
- $+$ to indicate addition modulo 2^W, where W is the word-width of the machine,
- $-$ to indicate subtraction modulo 2^W,
- $|$ to indicate concatenation,
- % to indicate modulo,
- $<<<$ to indicate left-rotation of bits,
- $>>>$ to indicate right-rotation of bits
- $<<$ to indicate left-shift of bits (excess are lost),
- $>>$ to indicate right-rotation of bits (excess are lost).
- `phi` = 0x9e3779b9 = $\frac{\sqrt{5}-1}{2}\ 2^{32}$

The reference implementation (section 9) assumes 32-bit words, but the logic of SBC's is independent of the word-size.

4. Feistal Implementations of SBC

First, let me recast classical Feistel network in a mathematically more simple form, so that the claims can be presented and contrasted to previous art in a consistent manner. Our innovative claims are not limited to Feistel ciphers, modified Feistel ciphers, or ciphers based on the kind of data-dependent rotations used in RC5 and RC6; these are all alternative embodiements of the basic Sliding Block Cipher concept.

Classic Feistel block ciphers represents a cipher block by two words, traditionally called either A and B or L and R. The plain text is in two 32-bit words (or larger words on future computers), W[0] and W[1]. The secret key is in an N-word array Key[0] ... Key[N-1].

The basic encryption process takes two plain text words and outputs two cipher text words, as follows.

```
A = W[0];
B = W[1];
for 0 ≤ r < R by +1 {
  A = mix(B, r, Key) ⊕ A;
  B = mix(A, r, Key) ⊕ B;
  }
C[0] = A;
C[1] = B;
```

This description presents the Feistel cipher as R 'full rounds' over a small, 2-element ring. Feistel ciphers may also be described as $2R$ 'half round followed by swap', where the swap may be represented by the following simultaneous assignment. (A,B) := (B,A)

There is no logical difference between the two descriptions, as is well known in the community.

The 'mix' function is not necessarily invertible. It combines the index r and the Key array to get a sub-key (e.g. K[r % N]), and combines that with the cipher block (A or B)

The decryption process takes the two cipher text words and uses the secret Key array to exactly invert the above encryption and recover the two plain text words, as follows.

```
A = C[0];
B = C[1];
for R > r ≥ 0 by -1 {
  B = mix(A, r, Key) ⊕ B;
  A = mix(B, r, Key) ⊕ A;
  }
W[0] = A;
W[1] = B;
```

This may be recast as follows:

(X[0], X[1]) = the initial two words of plain text, corresponding to (A,B)

In this notation, the first three full rounds are represented as follows.

```
X[0] = mix(X[1], 0, Key) ⊕ X[0]
X[1] = mix(X[0], 0, Key) ⊕ X[1]
X[0] = mix(X[1], 1, Key) ⊕ X[0]
X[1] = mix(X[0], 1, Key) ⊕ X[1]
X[0] = mix(X[1], 2, Key) ⊕ X[0]
X[1] = mix(X[0], 2, Key) ⊕ X[1]
```

and so on. The decryption is the reverse process.

This may be seen as an example of cycling around a 2-element ring three times. It is a simple but far-reaching generalization to increase the ring-size from two elements to an arbitrary number. Previous art has somewhat considered this generalization in only a limited way, as exemplified by the UFN and VBC approaches.

Previous art mainly has focused on finding better 'mix' functions, or using closely related (but non-Feistal) schemes such as RC5 (or its extension, RC6). The encryption scheme for RC5 is

X[i+2] = ((X[i] ⊕ X[i+1]) <<< X[i+1]) + subKey(i,Key),

where 'subKey(i,Key)' looks up the appropriate subkey in an expanded version of 'Key'.

Decryption is done in RC5 by X[i] = ((X[i+2] - subKey(i,Key)) >>> X[i+1]) ⊕ X[i+1]

RC6 extends the RC5 scheme to four words, (A,B,C,D) and accomplishes the 'swap' via a simple cyclic rotation of words: (A, B, C, D) := (B, C, D, A)

The permutation in the simultaneous assignment is fixed in the definition of RC6, and the possibility of varying the permutation is never discussed. Key- or data-dependent permutation only becomes possible with the K-word blocks of a true SBC.

The Feistal scheme is that $\mathrm{fe}(A, B, S) = mix(B, S) \oplus A$. Because RC5 employs a slightly different fe, it might not be considered literally a Feistal cipher, although it quite clearly draws from the same basic tradition. A Feistal scheme does not require an invertible mix function, whereas RC5's fe function can be inverted operation-by-operation.

Unbalanced Feistal networks (UFN) use two data blocks (L or R), but they are of differing sizes. The size of the data blocks are fixed by the cipher design. Hence, they are not prior examples of SBC.

For an SBC, the function $C = \mathrm{fe}(A, B, S)$ takes as input the two data words A and B, and the subkey word S, and computes a new data word C. Running the fe function over the ring consitutes one ring-encryption:

$$X[K + i] = \mathrm{fe}(X[i], X[K + i - 1], S[i]), \text{for } 0 \leq i < K \tag{4.1}$$

Notice that the index i counts up from 0 to $K - 1$.As the indices are modulo K, i and $K + i$ refer to the same place in memory, but the logic is clearer as presented above.

In the case that $K = 2$, this exactly equivalent to the classic formulation of 2-word Feistal ciphers, going all the way back to Feistal's original work on the Lucifer cipher for IBM. The classic description is to do one half-round, then a swap, then repeat the process. The typical terminology of the 'left' and 'right' words corresponds to the X[0] and X[1] words. The 'swap' step is implicit in the above, as we wrap around the small 2-element ring.

The inverse of fe is fd, with $A = \mathrm{fd}(C, B, S)$. For proper en/decryption, we require that $A = \mathrm{fd}(\mathrm{fe}(A, B, S), B, S)$ for all A, B, and S.

Running the fd function over the ring consitutes one ring-decryption:

$$X[i] = \mathrm{fd}(X[K + i], X[K + i - 1], S[i]), \text{for } K > i \geq 0 \tag{4.2}$$

Notice that the index i counts down from $K - 1$ to 0.

The parameter R determines how many times the fe function is to be applied in encryption, and how many times the fd function is to be applied in decryption.

A word-permutation function is also used; the parameter M determines how frequently the words-permutation is applied.

The psuedo-code for Feistal-based SBC is as follows.

Encryption:

```
for 0 ≤ i < K*R by +1:
  X[K+i] = fe ( X[i], X[K+i-1], S[i])
  if ((i > 0) && (0 == i % M))
    permute(X, i, Key);
```

Decryption:

```
for K*R > i ≥ 0 by -1:
  if ((i > 0) && (0 == i % M))
    depermute(X, i, Key);
  X[i] = fd ( X[K+i], X[K+i-1], S[i])
```

This is the version implemented in the reference implementation `sbcref`. There are many possible choices for K, M, fe/fd, key expansion, and permutation. All are controllable by the command-line inputs to `sbcref`, as described in section (9.4). A wide variety of combinations are demonstrated in the script `demo-sbcref.sh`

5. Non-Feistal Implementations of SBC

The explanation in section 4 and the reference implementation described in section 9 use the Feistal approach. However, Sliding Block Ciphers are not tied in anyway to the Feistal approach, as demonstrated in the following three sub-sections.

The first section simply describes how to make an SBC based on the RC6 fe/fd pairs, rather than the Feistal approach. Like the Feistal technique, the RC6 technique propagates changes around the ring.

The second section describes a very different technique. This uses a classical substitution cipher for the ring-encryption.

5.1. An SBC based on RC6

The structure described in section 4 would be used without change. All that needs changing is the particular fe/fd pair, so as to not use the Feistal scheme.

A suitable pair, as used in RC6, follows.

$$\begin{aligned} fe : C &= ((A \oplus B) <<< B) + S \\ fd : A &= ((C - S) >>> B) \oplus B \end{aligned} \tag{5.1}$$

Any variation using different permutation functions, K, M, etc. would still be a non-Feistal instance of SBC.

5.2. An SBC based on substitution cipher

We demonstrate how to build an SBC based upon a subsitution cipher. As it is not even similar to a Feistal network, this example demonstrates how greatly SBC differs from schemes such as Unbalance Feistal Networks (UFN).

Substitution ciphers are among the oldest known. The cipher used by Julius Caesar was a simple substitution cipher, where each letter of plaintext was replaced by a letter of ciphertext later in the Roman alphabet. In modern mathematical terms, this would represent each letter by an integer from 0 to 25, and encipherment would just be addition of 9, modulo 26, to each letter separately.

An example of polyalphabetic substitution cipher would be the classic Vigenere cipher. It essentially uses 26 different Caesarian ciphers in a data-dependent way; see section (11).

Substitution ciphers are quite vulnerable to frequency analysis or index of coincidence analysis, and are no longer seriously used. They retain pedagogical value, and they are used here simply to illustrate the range of possible Sliding Block Ciphers.

A simple SBC might go as follows. For reasons which will become clear in subsection (5.3), we will construct a Vignere cipher over 2^{64} symbols, using two adjacent 32-bit words for each symbol. A blocksize of 1024 bits could be represented as 32 unsigned long integers (ULI) of 32 bits each. We will use a 512-bit (16 ULI) key. The main loop will be repeated 24 times. Each loop is one round of ring-encryption followed by one word-permutation.

A simple ring-encryption step would be to consider the 32 words as 16 pairs. Call each pair A and B. They could be simply transformed with the following simultaneous assignment:

$$\begin{aligned} A'_i &:= pA_i + B_i + u_i \\ B'_i &:= qB_i + A_i + v_i \end{aligned} \tag{5.2}$$

The parameters p, q are fixed and public parameters of the transformation, while u_i, v_i are depenent on the key.

Notice that:

- A_i and B_i get mixed together by the transformation,
- changing 1 bit of A would probably change only a few bits of A' and of B', and vice versa,
- the secret key is needed to invert the transformation,
- this transformation can be represented as a 2-by-2 matrix multiplication and addition,
- that the complete transformation of the ring can be represented as multiplication by a public block-diagonal matrix M (32-by-32, with 16 2-by-2 blocks on the diagonal) and addition of a secret 32-vector U acting on the 32-vector X.

In other words, X is encrypted in one ring-encryption with an affine transformation, so equation (5.2) can be written more compactly as follows:

$$X' := MX + U \tag{5.3}$$

Therefore one encryption step is decrypted with the following:

$$X := M^{-1}(X' - U) \tag{5.4}$$

This encryption exhibits the poor "diffusion" typical of substitution ciphers, but twice that of the simple 32-bit version discussed in section (11).

Please note that, at this level of abstraction, **many** ciphers could be represented as a 1024-by-1024 permutation matrix, of which there are 1024!. To specify which one was desired (i.e. to specify both encryption algorithm and the secret key), would require $log_2(1024!)$ or approximately 8763 bits of information. Using a secret permutation matrix is quite cryptographically powerful.[1]

The above decryption (5.4) can be written in detail as following simultaneous assignment

$$\begin{aligned} A_i &:= (1-pq)^{-1}[B'_i + qu_i - (v_i + qA'_i)] \\ B_i &:= (1-pq)^{-1}[A'_i + pv_i - (u_i + pB'_i)] \end{aligned} \tag{5.5}$$

To make the required inverse exist uniquely, we require that $1 = \gcd(2^w, r)$, where $r = 1 - pq$. The simplest way to assure that is to make r be prime. Many suitable p, q pairs exist; one such randomly generated pair is given below.

[1] This is not a complete model of the information-theoretic complexity of a cryptosystem. Actually, a full cryptosystem is not just as a permution of the bits. If it were, then changing one bit of plaintext would change just the corresponding permuted bit of the ciphertext. In a real cryptosystem, changing one bit of the plaintext or of the key can potentially affect any bit of the ciphertext. A better model of the full range of cryptosystems is to consider each as a 1-to-1 mapping of each of the 2^{1024} possible plaintexts to one of the 2^{1024} possible ciphertexts. There are $2^{1024}!$ such mappings: to specify which was desired would take about 2^{1033} bits.

$$
\begin{aligned}
p &= 87003 \\
q &= 6166 \\
r &= (1 - pq) \ \% \ 2^{32} = 3758506799 \\
s &= r^{-1} \ \% \ 2^{32} = 87286735
\end{aligned} \tag{5.6}
$$

This is essentially a simple subsitution cipher, over an "alphabet" of 2^{64} symbols. Alone, it would not be very secure. As it is an affine transformation, simply repeating the encipherment would leave it affine without improving security at all.

If a permutation is applied between each ring-encryption,then each permutation would split up the pairs and scatter them over the ring. Because equation (5.2) makes the new A value depend on both the old A and the old neighbor B, splitting them up creates diffusion that equation (5.2) alone does not.

If u, v, p, and q were public constants in equation (5.2), then we would not be using any secret information or cryptographic primitives at all: even if repeated, it would be a plain and simple affine transformation. And it could be inverted with a simple, public affine transformation.

If the permutation were fixed and public, then it could be represented as a 32-by-32 permutation matrix, P, and each round could be represented as a transformation followed by a permutation:

$$X' := P((MX) + V) \tag{5.7}$$

and

$$X := M^{-1}((P^{-1}X') - V) \tag{5.8}$$

Clearly, it would remain an insecure affine transformation.

One way to increase security is to make the permutation matrix, P, secret and dependent on both the key and the data. The word-permutation function could be any of those in the reference implementation. Note that there are 32! permutations over 32 objects, and it would take about 118 bits of data to specify which was intended, as $2^{117} < 32! < 2^{118}$.

The `permFromArray` function in the reference implementation generates permutations over large sets of objects. Given just 4 ULI of data (128 bits), it can generate any of the possible permutations over 32 objects. With 512 bits (16 ULI) of key, we can sample 128 bits (4 ULI) of data from the key each round, stepping up 2 ULI each round, and use the full key in 8 rounds. The first round uses K[0] ... K[3] to generate the permutation; the second round uses K[2] ... K[5]; the third round use K[4] ... K[7]; and so on. With 24 rounds, we will cycle through the key exactly 3 times.

The key- and data-dependent permutation function `perm5` in the reference implementation uses `permFromArray` to permute the data words. Of course, this introduces exactly the kind of "diffusion" and "confusion" required, as the final location of each word potentially depends on every bit of the key, as well as on the data encrypted (this is necessary so that a random initial vector can be used).

Hence, we could use the simple substitution of equation (5.2) and the permutation function `perm5` to generate an SBC cipher. The security of this cipher has not been explored; its purpose is to demonstrate the range of ciphers that can be constructed within the SBC paradigm.

5.3. An SBC based only on permutation

Indeed, we could make the u_i and v_i in equation (5.2) be public constants (e.g. 0 or `phi`) and rely entirely on data- and key-dependent secret permutations for security. This would be an SBC that did not use any traditional encryption primitives, only the new cryptographic primitive of word-permutation - again emphasizing how different SB ciphers are from previous art.

We could use a 288-bit (9 ULI) key with 1024 bits (32 ULI) of data.

In each main loop, the ring-encryption step would be 16 mixes:

$$\begin{aligned} A_i &:= pA_i + B_i \\ B_i &:= qB_i + A_i \end{aligned} \tag{5.9}$$

As before, (5.9) alone is a simple affine transformation with no security. Thus, we have to provide security through a secret permutation which varies with data and with the key at each application.

To make the permutation depend on both data and key, we could make a 32-bit digest of the data at each round:

$$\alpha = qt(X[0] \oplus X[1] \oplus X[2] \oplus ... \oplus X[K-1])$$

where qt is the 1-to-1 nonlinear mapping defined in section (9.1).

Notice that α_j depends on the data in the j-th round, but it is independent of the order. Hence $\alpha_j$can be readily computed after the permutation has been done (as is required to invert the permutation while decrypting).

We do 27 rounds, where round j uses $[K_{3j}|K_{3j+1}|K_{3j+2}|\alpha_j]$ in `permFromArray` to determine the permutation. Notice that this provides 96 bits of data from the key and 32 bits from the digest. As `permFromArray` ignores excess data, the inputs are ordered so that all the K's get used, and the high order bits of α_j get used. With 27 rounds, we make 3 complete passes through the 9-ULI key, fully using each word of the key exactly three times. Twenty-seven ring encryptions alternate with twenty-seven permutations.

This particular cipher is intended to draw attention to the unique features of SB Ciphers. It is not claimed to be particularly efficient, and its cryptographic security has not been analyzed. The key observation is that the new cryptographic primitive of word permutation is so different from prior art that we can use it to define a plausible cryptosystem which does not use **any** of the prior cryptographic primitives.

6. SBC for Hashing

Sliding block ciphers can be used in fast, secure, and efficient hash algorithms, with highly variable output lengths. Any such uses of SB Ciphers are examples of SB Ciphers, not a new cipher.

To hash a long message, Msg, down to K words, simply split Msg into 'n' segments:

$$\mathrm{Msg} = [\mathrm{M}_1|\mathrm{M}_2|...|\mathrm{M}_\mathrm{n}]$$

Then use each segment as a key to encrypt a fixed public, 1-block value (V), and xor them together:

$$\mathrm{Hash(Msg)} = \mathrm{SBC}(\mathrm{M}_1, \mathrm{V}) \oplus \mathrm{SBC}(\mathrm{M}_2, \mathrm{V}) \oplus ... \oplus \mathrm{SBC}(\mathrm{M}_\mathrm{n}, \mathrm{V}) \tag{6.1}$$

for a publicly known parameterization of sliding block ciphers, SBC. The length of the hash is the length of the single block, V, which can be adjusted arbitrarily.

Use of SB Ciphers for such hash functions is an improvement over the prior state of the art, because it is no longer necessary to develop a whole new fixed-length hash algorithm for each desired output length. Such variable-length hash functions have not appeared in prior art, as previous ones (SHA1, MD5, and such) have fixed output length.

Hash functions for a wide variety of lengths are required by encryption schemes such as DHIES (as defined in "DHIES: An encryption scheme based on the Diffie-Hellman Problem" by Michel Abdalla, Mihir Bellare and Phillip Rogaway). We outline the DHIES system so as to clarify the unavoidable need for variable-length hashing. DHIES uses a Diffie-Hellman scheme to handle keys for a message authentication code (MAC) algorithm and a symmetric cipher (E,D). Of course, any secure MAC and (E,D) pair can be used, including those based on SBC; the essence of their system is the scheme for handling keys.

Setup:

g is a generator of a large group, say N1 = 2048 bits. This is public.

x_i = permanent secret key of principal i = psk

y_i = permanent public key of principal i = ppk = g^{x_i}

x_i and y_i are each N1 bits

Encipher:

M = plaintext to be enciphered, with principal i as the recipient.

r = random N1 bits

u = ephemeral public key = epk = g^r, N1 bits

v = ephemeral secret key = esk = ${y_i}^r$, N1 bits. Note that $v = g^{x_i r}$

$[\ ak \mid ek\ ] = H(v)$, where

ak = authentication key, say N2 = 128 bits

ek = encryption key, say N3 = 256 bits

Hence, the hash function H must reduce 2048 bits of v (N1) down to 384 (N2+N3), or whatever other length is implied by different lengths of g, ak, ek. Use of Sliding Block Ciphers make it easy to generate hashes of whatever length are required, without extensive redesign, merely by changing the blocksize parameter in (6.1).

The bare cipher text, using the symmetric cryptosystem (E,D) , is c:

c = E(ek, M)

As (E,D) can be any symmetric cryptosystem (e.g. RC6 in CBC mode), the message M can be arbitrarily many blocks.

h = MAC(ak, c) = hash for checking

There appears to be a typographical error in the paper cited above. It says h = MAC(ak, M), whereas the picture says h = MAC(ak,c), which also is the scheme for which the deciphering algorithm works.

C = (u, h, c) = (key, check, text) = complete enciphered message

Decipher:

C = (u, h, c) = ciphertext

$v' = u^{x_i}$ = recovered esk = g^{rx_i}

$[ak'|ek'] = H(v')$, where

ak' = recovered authentication key

ek' = recovered encryption key

```
if ( h ≠ MAC(ak', c) )
  output FAIL
else
  M' = D(ek, c)
  output M'
```

7. SBC for Psuedo Random Number Generation

Sliding block ciphers can be used in fast, secure, and efficient random number generators. Any such uses of SB Ciphers are examples of SB Ciphers, not a new cipher.

The basic 'entropy pool' approach, as outlined by Gutman and others, is to keep picking up semi-random numbers from some measurement process (timing of disk access, mouse movement, geiger counter clicks, etc.), then repeatedly mix them into the pool. To output 'b' bits from the PRNG, b bits are revealed (and the entropy count decremented by b), then the pool is stirred again.

Suppose X[0], ... X[K-1] is a large entropy pool of semi-random data, represented as K words of 32 bits each, and an entropy count H. H is initially 0, as the pool is in its known, initial state. There are $h_P = H/K$ bits of entropy per word if the pool is well-mixed.

Suppose the entropy source is outputting 32-bit words, with entropy of h_S bits per word ($0 \leq h_S \leq 32$). We could 'mix in' the new measurement, Z, by setting $X[0] = Z \oplus X[0]$, incrementing H by h_S, then 'stir' the pool by re-encrypting it with an SB Cipher and a fixed, public "key".

With H initially 0, mixing in Z raises H to h_S (and $h_P = H_S/K$). Mixing in another Z raises H by very slightly less than h_S, and so on such that H asympotically approaches 32*K bits of entropy. The curve is almost perfectly linear until the last few bits. $H_1 = \min(M, H_0 + h_S)$ is always accurate to within a fraction of a bit.

Of course, the SB Cipher would have its block size set to exactly the pool size. Because the SB Cipher is 1-to-1 (else decryption would be impossible), entropy is not changed by re-encrypting the pool, and it does get thoroughly mixed.

8. Guide to the Source Code

All the concepts described in this application have been reduced to practice, and are incorporated in the accompanying source code. This code is meant as a reference implementation to fully and unambiguously specify the invention, as well as to demonstrate that it has been reduced to practice, to demonstrate that the concepts discussed herein do in fact perform the functions claimed, and to act as a guide to later implementors of more optimized and/or secured implementations. It has been built and verified under Red Hat Linux, MinGW/Msys, Debian, and Windows.

Makefile: Under various forms of Linux, this builds the executable programs from source code.

sbcref.sh: This script file demonstrates the flexibility of Sliding Block Ciphers by running a series of demonstration cases, parameterized by command-line inputs. Each case has its own output file; all output files are included in this application.

sbcref.cc: This is the main demonstration program; it is called by sbcref.sh. The interface to the 'sbcref' demonstration program is explained in the section (9.4).

sbc.h: This header file specifies the main class, SBCipher, and interface to the main operations (encipher, decipher, encipherCBC, decipherCBC) discussed in this application.

sbc.cc: This file specifies algorithms to implement the main operations (encipher, decipher, encipherCBC, decipherCBC) discussed in this application.

fefd.h: This header file specifies the interface to the primitive fe/fd encryption pairs discussed in this application.

fefd.cc: This file implements the primitive fe/fd encryption pairs discussed in this application.

perm.h: This header file specifies the interface to the word-permutation functions discussed in this application.

perm.cc: This file implements the word-permutation functions discussed in this application.

subkey.h: This header file specifies the interface to the key-expansion functions discussed in this application.

subkey.cc: This file implements the key-expansion functions discussed in this application.

rng.h: This header file specifies the interface to the psuedo-random number generator used to generate example data.

rng.cc: This file implements the psuedo-random number generator used to generate example data.

utils.h: This header file specifies the interface to miscellaneous low-level utility functions which are incidental to SB Ciphers.

utils.cc: This file implements miscellaneous low-level utility functions.

9. Reference Implementation

A reference implementation of Sliding Block Ciphers in C++ is provided in source code form as part of this application. It is designed to be very obviously correct, so that people can use it as the 'gold standard' for verifying highly optimized implementations. Considerations of space and/or time efficiency have been deemphasized in favor of manifest correctness. Operational security measures, such as preventing virtual memory from writing sensitive data to the hard disk, are not relevant to a reference implmentation and so were omitted.

9.1. Development of the *qt* function

In the RC6 cipher, a particular quadratic function is used to produce a nonlinear permutation of integers:

$$j := i * (2i + 1)\%2^w \tag{9.1}$$

However, RC6's quadratic function in (9.1) turns out to have several potentially problematic features arising from the presence of fixed points. This section describes the problem, and how to avoid them.

First, we repeat Rivest's *reducto ad absurdem* proof that the equation (9.1) is 1-to-1.

Suppose we have two distinct integers x and y that map to the same result:

$$\begin{aligned} x(2x+1) &= y(2y+1) \\ 2x^2 + x &= 2y^2 + y \\ 0 &= 2(x^2 - y^2) + (x - y) \\ 0 &= 2(x+y) + 1 \end{aligned} \tag{9.2}$$

However, this is impossible, as the right hand side is the sum of an even term and an odd term, and so it can not be equal to 0 mod 2^w. Thus, no such distinct x and y can exist (and if they are equal, we can no longer divide by $x - y$ and so we no longer arrive at a contradiction).

Clearly, if $i = 2^{(w-1)}$, then $j = i * (2 * 2^{(w-1)} + 1)\%2^w = i(2^w + 1) = i$, so we have found one fixed point.

If $i = 2^{(w-1)} + z$, then i is a fixed point iff the following condition holds modulo 2^w:

$$\begin{aligned} 2^{(w-1)} + z &= (2^{(w-1)} + z) * (2 * (2^{(w-1)} + z) + 1) \ \% \ 2^w \\ 2^{(w-1)} + z &= (2^{(w-1)} + z) * (2^w + 2z) + 1) \ \% \ 2^w \\ 2^{(w-1)} + z &= (2^{(w-1)} + z) * (2z + 1) \ \% \ 2^w \\ 2^{(w-1)} + z &= 2^w z + 2^{w-1} + 2z^2 + z \\ 0 &= 2z^2 \end{aligned} \tag{9.3}$$

or, for some a: $a * 2^w = 2z^2$

Whenever $z = b * 2^{(w/2)}$, for any b, such an a exists: $a = 2b^2$.

Thus, RC6's 32-bit quadratic function has a fixed point at every offset from 2^{31} which is a multiple of 2^{16} - which is 2^{16} fixed points, of course.

Thus, when w is even (e.g. 32 bits, 8 bits, etc.) then there are many fixed points of RC6's quadratic function, which present potential vulnerability to chosen cipher text attack, differential cryptanalysis, linear cryptanalysis, etc.

One simple generalization of the quadratic mapping is the following:

$$j := (i + p) * (ni + c)\%2^w \tag{9.4}$$

By a proof very similar to Rivest's, we can show that the new function is 1-to-1 if n is even and c is odd:

$$\begin{aligned}(x + p)(nx + c) &= (y + p)(ny + c)\\ 0 &= n(x^2 - y^2) + (pn + c)(x - y)\\ 0 &= n(x + y) + (pn + c)\end{aligned} \tag{9.5}$$

As before, making n even and c odd ensures that the right hand side is the sum of an even term and an odd term, which can not be equal to zero. Hence, no two such distinct x and y can exist.

Note that p is completely unrestricted. In particular, it could be part of a secret key, part of a data-digest, and so on.

If we make n any other even number, simply by using $j = i(2fi + 1)$ for various f, we get fixed points at every $a2^{(w/2)}$ offset, by the same proof.

If we explore different odd c by using $j = i * (2fi + (2g + 1))$, we can use a simple numerical scan to find a variable number of fixed points, depending on g.

So neither n nor c alone fix the problem.

If we set $p \neq 0$, i.e. $j = (i + p) * (2i + 1)\%2^w$, we get better results.

If we try $p = 1$, we get a 1-to-1 quadratic permuation with no fixed points[1]. This is the version of qt used unless otherwise specified:

$$qt(i) = (i + 1) * (2i + 1) \tag{9.6}$$

However, choosing $p = 1$ retains the problem that the function is monotonically increasing for all i less than about $2^{15.5}$. If one were expecting consistently small inputs, it might be worthwhile to use $p =$ `phi`, so that it "rolls over" and is non-monotonic for even small i.

[1] This was verified by a simple scan of all 32-bit numbers. The same result holds for most (all?) $p \neq 0$.

9.2. The Rotate-Strech-Fold Permutation

The generation of permutations by the `permFromArray` function is theoretically complete, in that (for fixed permutation size and sufficiently large array) every possible permutation is generated by some value for the data array. It essentially treats the array as a single large integer, say n, and generates the n-th permutation, where the identity permutation is number 0. However, it is not extremely fast.

Therefore, another permutor was developed, based on the well known "Smale Horseshoe". It is arbitrarily designated `perm1`.

Imagine streching a piece of string to double its length, then folding it back on itself. If the positions on the string were given by a parameter u, we would have $0 \leq u \leq 1$. The stretch and fold operation would be defined by $h(u) = 2u$ if $0 \leq u \leq 1/2$ and $h(u) = 2 - 2u$ if $1/2 \leq u \leq 1$. The distance between closely spaced points would double with each iteration, which gives the exponential sensistivity to initial conditions which is characteristic of chaotic systems. The rotate part of the operation is represented by adding a constant, r, modulo 1. The full RSF operation on the string is represented by $u' = h((u + r)\%1)$.

This is what we do with the indices from 0 to $K - 1$ in the ring. First, they are incremented by a constant, modulo K. Then the low indices of the ring have their indices doubled, so $0 \rightarrow 0$, $1 \rightarrow 2$, $2 \rightarrow 4$, and so on. Correspondingly, the high indices are folded down: $K - 1 \rightarrow 1$, $K - 2 \rightarrow 3$, $K - 3 \rightarrow 5$, and so on.

The suggested `perm1` performs an RSF operation 8 times, using a different data-dependent rotation each time. Before the first, the qt-transform of each word of the data is xor-ed together to make one 32-bit 'sum'. That is the value used in the first rotation. Then it is rotated 7 bits to the left, and used for the rotation of another RSF round. After eight RSF operations are completed, `perm1` is finished.

Clearly, variants of RSF could be used which combined the key with the data to determine the rotation amount; these would consistent different examples of SBC systems. However, the suggested `fefd4` functions mix the data words using the key, so it was felt that RSF using data-dependency alone would be sufficient for the illustrative purposes of a reference implementation.

9.3. Suggested Parameters

Suggestions for a concrete SBC example, illustrated by the sbcex.D.00.TXT, ... sbcex.D.03.TXT example files, are as follows:

Suggested fe4/fd4:

$$C = (qt(B \oplus S) <<< 7) + A$$

$$A = C - (qt(B \oplus S) <<< 7)$$

where

C = X[K+i],

A = X[i],

B = X[K+i-1], and

S = subkey(i)

Suggested subkey function is subkey1(). The point of this simple subKey expansion is to ensure that, even if the key is entirely zero, the subkeys are not all zeros. The use of 'phi' ensures that it does not repeat, even if the key itself is short.

subkey1(i) = qt(i * phi + Key[i % N]),

Suggested number of rounds is 24.

Suggested permuter is the perm1() function. It does eight data-dependent rotate-stretch-fold (RSF) operations, bit-rotating the sum by 7 bits each time as described in section (9.2).

Suggested M = 2*K, so that we permute every two full rounds.

9.4. Demonstration program `sbcref`

The demonstration program 'sbcref' takes the following command line arguments.

```
-fefd <n>:  0, 1, 2, 3, or 4
   4 is recommended.
   see fefd.h and fefd.cc


-subkey <n>:  0 or 1
   1 is recommended.
   see subkey.h and subkey.cc


-perm <n>:  0, 1, 2, 3, or 4
   perm0:  i' = (i+1)*(2*i + 1)
   perm1:  repeated folding, based on intermediate ciphertext
   perm2:  treats intermediate ciphertext and key as large integer to index
permutation
   perm3:  sort-based permuter, based on intermediate ciphertext and key
   perm4:  multiplicative-inverse based permuter, rotation based on
intermediate ciphertext
   1 is recommended.
   see perm.h and perm.cc


-flatK <n>:  use a key which is all the given unsigned long int value.
   a good test value is 0


-flatP <n>:  use a plaintext which is all the given unsigned long int value.
   a good test value is 0


-seed <n>:  random number seed, which must be non-negative.
   positive values are used to seed the PRNG.
   0 means use random system data generate a seed for the PRNG
   in either case, the seed actually used is displayed in the output


-K <n>:  length of the block


-N <n>:  length of the key
```

```
-M <n>:  how often to permute word (M=2K means one permute every two rounds)

-R <n>:  the number of rounds
```

The `sbcref` program is demonstrates SBC in a flexible way, depending on the parameters typed by the user, thus demonstrating that the parameters are completely flexible.

To help others verify that they have a correct implementation, the script `sbcref.sh` is provided to demonstrate several uses of sliding block ciphers. Given the exact same inputs for key, plain text, fe/fd, permutation, etc., your implementation should produce exactly the same ciphertext.

The script `sbcref.sh` demonstrates an array subkey expansion algorithms, permutation algorithms, fe/fd pairs, key sizes, block sizes, round numbers, and permutation intervals. All these combinations are examples of sliding block ciphers, and have all been reduced to practice.

The output files from `sbcref.sh` are called `sbcex.A.00.TXT` through `sbcex.G.02.TXT`. A PGP-signed listing of all their MD5 signatures is provided in the text file `md5.TXT`

9.5. Challenge Problem

A challenge problem is also provided. An HTML file called example1.html was SBC encrypted with the default parameters using the `sbcfile` program. The source code is provided, so that programmers can verify the algorithm, defaults, etc. Only the key is secret.

Programmers are invited to try to decrypt the file. More information, as well as the MD5 signature of the original plaintext, is provided in `example1.txt`. Please contact the author (including the decrypted file) if you think you have broken the code.

10. Efficiency

Basic algorithmic analysis indicates that SBC have the same time- and space-efficiency as common non-SBC ciphers. NB: The supplied reference implmentation is completely unoptimized, as it emphasizes clarity over speed.

10.1. Time Efficiency

Consider how a 128-bit block cipher (cipher A) would be used to encrypt 131,072 bits of data, with 30 rounds of the 'fe' function. First, the data would be divided into 1024 blocks. Each block would be encrypted using cipher A. The blocks would typically be chained together CBC mode or something similar. As each round would take 4 evaluations of 'fe' on 32-bit words, this would require 1024*30*4 or 122880 evaluations of 'fe' (plus the overhead of 1023 xor's of blocks in the CBC chaining).

Suppose the data were encrypted with a sliding block cipher (cipher B) with 4096-bit blocks (i.e. 128 words of 32 bits each). First, the data would be divided into 32 blocks. Each block would be encrypted using cipher B. The blocks would typically be chained together CBC mode or something similar. As each round would take 128 evaluations of 'fe' on 32-bit words, this would require 32*30*128 or 122880 evaluations of 'fe' (plus the overhead of 31 xor's of blocks in the CBC chaining).

Suppose the data were encrypted with a sliding block cipher (cipher C) with a 131,072-bit block (i.e. 4096 words of 32 bits each). The single block would be encrypted using cipher C. As each round would take 4096 evaluations of 'fe' on 32-bit words, this would require 1*30*4096 or 122880 evaluations of 'fe' (plus zero overhead in CBC chaining).

10.2. Space Efficiency

The subkey schedule could be pre-computed and stored in an array, if it were small. This would be quite similar to fixed-block ciphers, so it would represent no change in efficiency. If the subkey schedule had a very long period, such as S[i] = (i*P1) $\oplus$ Key [(i*P2) % N] for two large primes P1 and P2, then S[i] would have to be computed rather than stored in an array. As this example shows, high efficiency can be acheived while still using all the bits of the Key in a very nonlinear way.

Similarly, the permutation could be either computed or stored in a lookup table.

11. Vignere Ciphers

An example of polyalphabetic substitution cipher would be the classic Vigenere cipher. It essentially uses 26 different Caesarian ciphers in a data-dependent way. The classic description of the cipher is as follows.

First, one makes a 26-by-26 matrix. The first column is the alphabet in order from top to bottom, starting with 'a'. The second column is rotated by one position, so that it starts with 'b', the third is rotated so that it starts with 'c', and so on. In modern mathematical notation, this could be represented as matrix, C, with C[i,j] = (i+j) % 26.

While this is the standard C matrix for the Vignere cipher, any matrix that represents all 26 letters in each column will do.

Second, choose a key, such as "LURID". If the key has N letters, then it is represented as k[0], ..., k[N-1].

The plaintext, say "MEET AT MIDNIGHT BY THE OLD OAK TREE". If the plaintext has M letters, then it is represented as p[0], ..., p[M-1]. Spaces and punctuation are dropped when using a 26-by-26 matrix.

Third, use the matrix to read off the ciphertext. The cipher letter is the letter in the tableau determined by the plaintext row and the keyword column. Thus the first "M" of the message is encrypted as "X", the first "E" as "Y", the second "E" as "V", and so on.

While a 26-by-26 matrix is small enough to handle explicitly, it is useful to see the implicit representation as addition. This will be not merely handy but necessary when the ring size, M_R, is much larger than 26.

In an equation,

$$\begin{aligned} c[i] &= C[\ p[i],\ k[i\%N]\ ] \\ &= (p[i] + k[i\%N])\ \%\ M_R \end{aligned} \tag{11.1}$$

To decrypt the Vigenere cipher, you just subtract modulo M_R, which was classically 26: $p[i] = c[i] - k[i\%N]\ \%\ M_R$

As mentioned earlier, the classic Vignere cipher over Roman letters (26 symbols) is vulnerable to analysis by the index of coincidence.

Nevertheless, for our illustrative purposes, one can imagine a straightforward extension of the Vignere cipher over 32-bit words (2^{32} 'symbols' in a ring with $M_R = 2^{32}$). The basic logic is similar, except that the 2^{32}-by-2^{32} C matrix has to be implicitly represented as addition. As mentioned earlier, the classic C[i,j] = i+j is merely one of many possible ways to generate a matrix will all the symbols in each column. For 32-bit words, C[i,j] = i + qt(j) would also list all symbols in each column. Each column would be a different cyclic permutation of the symbols, but starting out at very different initial positions due to the nonlinearity of qt.

$$\begin{aligned} c[i] &= C[\ p[i],\ k[i\%N]\ ] \\ &= (p[i] + qt(\ k[i\%N]\ ))\%M_R \end{aligned} \tag{11.2}$$

While tougher than the classic Vignere, this is still a very weak cipher by modern standards. Because of the simple addition, changing one bit of the plaintext changes only a few bits of the ciphertext: it has very little diffusion.

Equation (11.2) could be slightly generalized to include a scaling factor, q. We require $1 = \gcd(q, M_R)$ so that the affine transformation is invertible.

$$c[i] = ((q * p[i]) + qt(\ k[i\%N]\ ))\ \%\ M_R \tag{11.3}$$

12. References

1. Clausewitz, C. v., *On War*, Translated by M. Howard and P. Paret, Princeton University Press, New Jersey (1976).

Table of Contents