Key-Dependent S-Box Manipulations

Sandy Harris[1] and Carlisle Adams[2]

[1] Kaya Consulting, 6 Beechwood Avenue, Suite 16
Vanier, Ontario,Canada, K1L 8B4
sandy.harris@sympatico.ca
[2] Entrust Technologies, 750 Heron Road
Ottawa, Ontario, Canada, K1V 1A7
cadams@entrust.com

Abstract. This paper discusses a method of enhancing the security of block ciphers which use s-boxes, a group which includes the ciphers DES, CAST-128, and Blowfish. We focus on CAST-128 and consider Blowfish; Biham and Biryukov [2] have made some similar proposals for DES.
The method discussed uses bits of the primary key to directly manipulate the s-boxes in such a way that their contents are changed but their cryptographic properties are preserved. Such a strategy appears to significantly strengthen the cipher against certain attacks, at the expense of a relatively modest one-time computational procedure during the set-up phase. Thus, a stronger cipher with identical encryption / decryption performance characteristics may be constructed with little additional overhead or computational complexity.

1 Introduction

Both carefully-constructed and randomly-generated s-boxes have a place in symmetric cipher design. Typically, a given cipher will use one or the other paradigm in its encryption "engine". This paper suggests that a mixture of the two paradigms may yield beneficial results in some environments. In our examples, we use the four 8×32 s-boxes which the Blowfish and CAST-128 ciphers employ, but variations of this technique could be applied to any cipher using s-boxes, whatever their number and sizes.

We propose using strong s-boxes and applying key-dependent operations to them at the time of key scheduling, before the actual encryption begins. The goal is to get the benefits of strong s-boxes (as in CAST-128) and of key-dependent s-boxes (as in Blowfish) without the drawbacks of either.

The technique can be powerful. If a basic cipher can be broken in a second by exhaustive search over the key space and if key-dependent operations on the s-boxes add 32 bits to the effective key length, then breaking the improved cipher by brute force takes 2^{32} seconds (just over a century). If these operations add 64 effective bits, then it would take 2^{32} centuries to break the improved cipher.

Key-dependent operations on s-boxes can use large numbers of bits. For 8×32 s-boxes, XORing constants into the inputs and outputs can use 40 bits per s-box.

S. Tavares and H. Meijer (Eds.): SAC'98, LNCS 1556, pp. 15–26, 1999.

Permuting the inputs and outputs can use $\log_2(8!) + \log_2(32!) > 130$ bits per s-box. A cipher with four such s-boxes could use 680 bits of additional key with these operations (although it is recognized that this will not necessarily be the increase in the effective key length).

We start with the CAST-128 cipher and propose using between 148 and 256 additional key bits for s-box transformations. The increase in effective key length (although difficult to compute precisely) is likely to be considerably lower than this, but the transformations still appear to be worthwhile in at least some applications. In particular, the cost is moderate and the resulting cipher appears to be more resistant to attacks that rely upon knowledge of the s-box contents.

It is important to note that the technique is inherently efficient in one sense: all the s-box transformations are done at set-up time. Thus, there is no increase in the per-round or per-block encryption time of the strengthened cipher.

2 Considerations

2.1 The Extra Key Bits

The additional bits required for this proposal may come from one of two sources: derived key bits or primary key bits. As an example of the former, CAST-128 [1] expands the 128-bit key to 1024 bits but does not use all of them. The actual encryption uses 37 bits per round (592 in the full 16 rounds) so that 432 bits are generated by the key scheduling algorithm but are unused by the cipher. These currently unused bits may be good candidates for the bits needed for s-box manipulation.

Alternatively, additional primary key bits may be used for the proposal in this paper. This has the advantage of increasing the key space for exhaustive search attacks, at the cost of increased key storage and bandwidth requirements. Note, however, that in some environments the bandwidth required for key transfer or key agreement protocols need not increase. In one common use of symmetric ciphers, session keys are transmitted using a public key method such as RSA [5] or Diffie-Hellman [3]. Public key algorithms use large numbers of bits so that to transmit a 128-bit session key, you may need to encrypt, transmit and decrypt a full public-key block of 1024 bits or more. In such a case, any key up to several hundred bits can be transmitted with no additional cost compared with a 128-bit key.

Using derived key bits has no impact on primary key size, but depends upon a key scheduling algorithm that generates extra (i.e., currently unused) bits. Using primary key bits places no such requirement on the key scheduling algorithm, but has storage and bandwidth implications, and may show some susceptibility to chosen-key-type attacks (since the two pieces of the primary key are "separable" in some sense).

2.2 CAST's Strong S-Boxes

The CAST design procedure uses fixed s-boxes in the construction of each specific CAST cipher. This allows implementers to build strong s-boxes, using bent

Boolean functions for the columns and choosing combinations of columns for high levels of s-box nonlinearity and for other desirable properties. Details can be found in Mister and Adams [4].

For example, the CAST-128 s-boxes appear to be strong but they are fixed and publicly-known. This may allow some theoretical attacks (e.g., linear or differential cryptanalysis) to be mounted against a given CAST cipher which uses these s-boxes, although the computational cost of these attacks can be made to be infeasibly high with a suitable choice in the number of rounds.

2.3 Blowfish's Key-Dependent S-Boxes

Blowfish generates key-dependent s-boxes at cipher set-up time. This means the attacker cannot know the s-boxes, short of breaking the algorithm that generates them.

There are at least two disadvantages, which can to some extent be traded off against each other. One is that generating the s-boxes has a cost. The other is that the generated s-boxes are not optimized and may even be weak.

A Blowfish-like cipher might, with some increase in set-up cost, avoid specific weaknesses in its s-boxes. Schneier discusses checking for identical rows in Blowfish [6, page 339] but considers this unnecessary. In general, it is clearly possible to add checks which avoid weaknesses in randomly-generated s-boxes for Blowfish-like ciphers, but it is not clear whether or when this is worth doing.

On the other hand, generating cryptographically strong s-boxes at run time in a Blowfish-like cipher is impractical, at least in software. Mister and Adams [4] report using 15 to 30 days of Pentium time to generate one 8×32 s-box suitable for CAST-128, after considerable work to produce efficient code. This is several orders of magnitude too slow for a run-time operation, even for one used only at set-up time and not in the actual cipher.

2.4 Resistance to Attack

Schneier [6, p.298] summarizes the usefulness of randomly-generated s-boxes with respect to the most powerful statistical attacks currently known in his introduction to the Biham and Biryukov work on DES with permuted s-boxes [2]:

"Linear and differential cryptanalysis work only if the analyst knows the composition of the s-boxes. If the s-boxes are key-dependent and chosen by a cryptographically strong method, then linear and differential cryptanalysis are much more difficult. Remember, though, that randomly-generated s-boxes have very poor differential and linear characteristics, even if they are secret."

This inherent dilemma leads to the proposal presented in this paper: we suggest s-boxes that are key-dependent but are not randomly generated.

3 The Proposal

Start with carefully-prepared strong s-boxes, such as those described for CAST in Mister and Adams [4] and apply key-dependent operations to them before use. The goal is to introduce additional entropy so that attacks which depend on knowledge of the s-boxes become impractical, without changing the properties which make the s-boxes strong.

We apply the operations before encryption begins and use the modified s-boxes for the actual encryption, so the overhead is exclusively in the set-up phase. There is no increase in the per-block encryption cost.

It can be shown that important properties of strong s-boxes are preserved under carefully-chosen key-dependent operations. Given this, it is possible to prepare strong s-boxes off-line (as in CAST-128) and manipulate them at cipher set-up time to get provably strong key-dependent s-boxes (in contrast with ciphers such as Blowfish).

The question is what operations are suitable; that is, what operations are key-dependent, reasonably efficient, and guaranteed not to destroy the cryptographic properties of a strong s-box.

The first two requirements can be met relatively easily; simultaneously achieving the third is somewhat more difficult. However, several classes of operations may be used.

- Permuting s-box columns
  - this has the effect of permuting output bits.
- Adding affine functions to s-box columns
  - this has the effect of complementing output bits, possibly depending upon the values of other output bits.
- Permuting s-box inputs
  - this has the effect of producing certain s-box row permutations.
- Adding affine functions to s-box inputs
  - this has the effect of producing other s-box row permutations, possibly depending upon the values of other input bits.

In general, then, the Boolean function for an s-box column may be modified from

$$f(\overline{x}) = f(x_1, x_2, x_3, \ldots, x_m) \ ,$$

for binary variables x_i, to

$$f\left(P\left(g_1\left(\overline{x}\right), g_2(\overline{x}), g_3(\overline{x}), \ldots, g_m(\overline{x})\right)\right) \oplus h(\overline{x}) \ ,$$

for some Boolean functions $g_i(\overline{x})$ and $h(\overline{x})$ and a permutation P. The set of columns may then be further permuted. We will consider the above classes of operations in the order presented.

3.1 Permuting S-Box Columns

Permuting s-box columns can be accomplished by permuting each row in the same way (done in one loop through the s-box).

Various important properties are conserved under this operation. In particular, if a column is bent, it will clearly remain so when moved; if a group of columns satisfies the bit independence criterion, it will still do so after being permuted. Finally, since s-box nonlinearity is defined to be the minimum nonlinearity of any function in the set of all non-trivial linear combinations of the columns (see [4], for example), then s-box nonlinearity is also conserved through a column permutation.

In carefully designed s-boxes, rearranging the columns in a key-dependent way does not degrade cryptographic strength. However, such an operation can make it significantly more difficult to align characteristics in a linear cryptanalysis attack and so can increase the security of the cipher by raising the computational complexity of mounting this attack.

3.2 Adding Affine Functions to S-Box Columns

In the extreme case in which the affine functions are simply all-one vectors, the addition can be done by XORing a constant into all rows (done in one loop through the s-box). More generally, other techniques (perhaps involving storage of specific affine vectors) may be necessary to accomplish this addition.

Various important properties are conserved under this operation. Because the nonlinearity of a Boolean function is unchanged by the addition of an affine function, s-box column bentness, s-box bit independence criterion, and s-box nonlinearity are all conserved.

The addition of affine functions, therefore, does nothing to degrade cryptographic security in the s-boxes. However, such an operation, by modifying the contents of the s-boxes in a key-dependent way, can make it significantly more difficult to construct characteristics in a differential cryptanalysis attack (because it cannot be computed in advance when the XOR of two given s-box outputs will produce one value or another). Hence, this operation can increase the security of the cipher by raising the computational complexity of mounting this attack.

3.3 Permuting S-Box Inputs

Permuting the rows of an s-box seems attractive because of its potential for thwarting linear and differential cryptanalysis. However, it is not always possible to permute rows without compromising desirable s-box properties. In particular (e.g., for CAST-128 s-boxes), not all row permutations are permissible if column bentness is to be preserved.

Biham and Biryukov [2] made only one small change to the s-box row order in the DES s-boxes: they used one key bit per s-box, controlling whether or not the first two and the last two rows should be swapped. However, an operation

that used more key bits and that provably could not weaken strong s-boxes may be preferable in some environments.

One such operation is to use the subset of row permutations that result from a permutation on the s-box inputs. We will show that these do not damage the desirable s-box properties.

Mister and Adams [4] introduce the notion of *dynamic distance of order j* for a function f

$$f : \{0,1\}^m \rightarrow \{0,1\}$$

and define it as

$$DD_j(f) = \max_{\overline{d}} \frac{1}{2} \left| 2^{m-1} - \sum_{\overline{x}} (f(\overline{x}) \oplus f(\overline{x} \oplus \overline{d})) \right|$$

where both $\overline{d}$ and $\overline{x}$ are binary vectors of length m, $\overline{d}$ ranges through all values with Hamming weight $1 \leq \mathrm{wt}(\overline{d}) \leq j$ and $\overline{x}$ ranges through all possible values.

It is shown in [4] that cryptographic properties such as Strict Avalanche Criterion (SAC) and Bit Independence Criterion (BIC), higher-order versions of these (HOSAC and HOBIC), maximum order versions of these (MOSAC and MOBIC), and distances from these (DSAC, DBIC, DHOSAC, DHOBIC, DMOSAC, and DMOBIC) can all be defined in terms of dynamic distance.

In an s-box, all bits are equal. There is no most- or least-significant bit in either the input or the output. Thus, permuting the bits of x in the formula above does not change the value of the summation term for a given d, provided we apply the same permutation to d. Hence it does not change the maximum (the value of the dynamic distance).

Therefore, column properties defined in terms of dynamic distance (DSAC, DHOSAC, and DMOSAC) remain unchanged. In particular, if the columns are bent functions (i.e., DMOSAC = 0) then permuting inputs preserves bentness.

Furthermore, the s-box properties DBIC, DHOBIC, and DMOBIC also remain unchanged because these are defined in terms of dynamic distance of a Boolean function f comprised of the XOR of a subset of s-box columns. (Note that permuting the inputs of each of the column functions with a fixed permutation P is identical to permuting the inputs of the combined function f using P.) By a similar line of reasoning, s-box nonlinearity is also unaffected by a permutation of its input bits.

Permuting inputs, therefore, does nothing to degrade cryptographic security in the s-boxes. However, such an operation, by rearranging the order of the s-box rows in a key-dependent way, can make it significantly more difficult to construct linear or differential characteristics (because specific outputs corresponding to specific inputs cannot be predicted in advance). Hence, this operation can increase the security of the cipher by raising the computational complexity of mounting these attacks.

3.4 Adding Affine Functions to S-Box Inputs

Adding selected affine functions to s-box inputs is another effective way of producing a subset of row permutations that does not reduce the cryptographic security of the s-box.

In the extreme case in which the affine functions are constant values, the addition simply complements some of the s-box input bits. Inverting some input bits is equivalent to XORing a constant binary vector into the input, making the summation in the dynamic distance

$$\sum_{\overline{x}} \left(f(\overline{x} \oplus \overline{c}) \oplus f(\overline{x} \oplus \overline{c} \oplus \overline{d}) \right)$$

Clearly $(\overline{x} \oplus \overline{c})$ goes through the same set of values that $\overline{x}$ goes through, so this does not change the sum and, consequently, does not change the dynamic distance. Therefore, column properties and s-box properties are unchanged.

In the more general case in which the affine functions are not constant values, the addition conditionally complements some of the s-box inputs (depending upon the particular values of some subset of input variables). Consider the following restriction. Choose any k input variables and leave these unchanged. For the remaining m-k input variables, augment each with the same randomly-chosen, but fixed, affine function of the chosen k input variables. For example, in a $4 \times n$ s-box, we may choose input variables x_1 and x_2 to be unchanged and augment x_3 and x_4 with the affine function $g(x_1, x_2) = x2 \oplus 1$ so that the Boolean function $f_i(x_1, x_2, x_3, x_4)$ defining each s-box column i becomes

$$f_i(x_1, x_2, x_3 \oplus g(x_1, x_2), x_4 \oplus g(x_1, x_2)) = f_i\left(x_1, x_2, (x_3 \oplus x_2 \oplus 1), (x_4 \oplus x_2 \oplus 1)\right).$$

With the operation restricted in this way it is not difficult to see that as the chosen k variables go through their values, at each stage the remaining m-k variables go through all their values (either all simultaneously complemented, or all simultaneously not complemented, depending upon the binary value of the affine function). Thus, rewriting the summation in the dynamic distance equation as

$$\sum_{\overline{x}} \left(f(\overline{x}') \oplus f(\overline{x}' \oplus \overline{d}) \right)$$

where $\overline{x}'$ is in accordance with the restriction as specified, we see that $\overline{x}'$ goes through the full set of values that $\overline{x}$ goes through, so the sum is unchanged and the resulting dynamic distance is unchanged.

Adding affine functions (restricted as specified above[1]) to s-box inputs, therefore, does not degrade cryptographic security in the s-boxes. Like permuting inputs, this operation, by rearranging the order of the s-box rows in a key-dependent way, can make it significantly more difficult to construct linear or

[1] Note that other restrictions on the type and number of affine functions that may be added to preserve s-box properties may also exist. This area is for further research.

differential characteristics. The security of the cipher may therefore be increased by raising the computational complexity of mounting these attacks.

3.5 Other Possible Operations

Other key-dependent operations that preserve s-box properties are also possible. For example, it is theoretically possible to construct strong s-boxes with more than 32 columns and select columns for actual use at set-up time, but this would likely be prohibitively expensive in practice since Mister and Adams [4] report that s-box generation time doubles for each additional column.

Another possibility is to order the s-boxes in a key-dependent way. This is not particularly useful with only four s-boxes since only 4! orders are possible, adding less than five bits of entropy to the key space. However, with the eight s-boxes in CAST-128, this operation becomes somewhat more attractive. A CAST-143 might be created in a very straightforward way: $\log_2(8!) = 15$ bits of key puts the eight s-boxes into some key-dependent order (cheaply by adjusting pointers), and then key expansion and encryption proceeds exactly as in CAST-128 except with the s-boxes in the new order. The overhead (set-up time) is quite low and the new cipher uses 15 extra bits of unexpanded key.

3.6 Limitations in Key-Dependent Operations

Ciphers With XOR-Only Round Functions A cipher which combines s-box outputs with XOR, such as the CAST example in Applied Cryptography [2, page 334]), does not work well with some types of s-box manipulation. For example, permuting the order of the four round function s-boxes is of no benefit in such a cipher, since XOR is commutative.

XORing different constants into the four s-boxes in such a cipher has exactly the same effect as XORing a single constant into each round function output, or into any one s-box.

Furthermore, if the cipher's round function combines its input and the round key with XOR, then XORing a constant into the output of one round is equivalent to XORing that constant into the key of the next round. If the round keys are already effectively random, unrelated, and unknown to the attacker (as they should be), then XORing them with a constant does not improve them.

In terms of the difficulty of an attack, then, the net effect of XORing four constants into the s-boxes is equivalent to XORing a single constant into the output of the last round, for a cipher which uses XOR both to combine s-box outputs and to combine round input with the round key.

Ciphers With Mixed Operations Combining S-Box Outputs A cipher which uses operations other than XOR to combine s-box outputs, such as Blowfish or CAST-128, will give different round outputs if the order of the s-boxes is changed or if a constant is XORed into each row. This makes these operations more attractive in such ciphers.

Even in such ciphers, however, the precise cryptographic strength of XORing a constant into the rows is unclear. Certainly it is an inexpensive way to mix many key bits (128 if the cipher uses four $m \times 32$ s-boxes) into the encryption, but it is not clear exactly how much this increases the effective key length.

Ciphers With Mixed Operations Combining Key and Input In Blowfish and in some CAST ciphers, the round input is XORed with the round key at the start of a round, then split into four bytes which become inputs to the four s-boxes. XORing an 8-bit constant into each s-box input is equivalent to XORing a 32-bit constant (the concatenation of the 8-bit constants) into each of the round keys.

Suppose an attack exists that discovers the round keys when the s-boxes are known. Then the same attack works against the same cipher with s-boxes that are known but have had their rows permuted in this way. The attack discovers a different set of round keys equivalent to the real ones XORed with a 32-bit constant, but it still breaks the cipher, and with no extra work for the attacker.

However, for ciphers that use other operations to combine the round input and the round key (CAST-128, for example, which uses addition and subtraction modulo 2^{32} for input masking in some of its rounds), such an operation seems to add value.

Options For both inputs and outputs, the addition of affine functions appears stronger than just XORing with a constant, and performing permutations appears to be stronger again (but at much higher computational cost). In a practical cipher, however, there appears to be no disadvantage to using XOR (for both input and output if mixed operations are used everywhere in the round function) because it is inexpensive and offers some protection against the construction of iterated characteristics.

4 Practical Considerations

4.1 Stage One

Since XORing a constant into the s-box rows is the cheapest way to bring many extra key bits into play; we should do that if we're going to use this approach at all. The cipher's round function should use operations other than XOR to mix s-box outputs so that this will be effective.

If we are iterating through the s-box rows for that, it makes sense to permute the columns in the same loop. We suggest simply rotating each row under control of 5 bits of key. A CAST-128 implementation will have code for this, since the same rotation is used in the round function, and rotation is reasonably efficient.

At this point, we have used 37 bits per s-box, 148 bits in all. In many applications, this will be quite sufficient.

Costs of this are minimal: 1024 XOR and rotation operations. This is much less than CAST-128's round key generation overhead, let alone Blowfish's work to generate s-boxes and round keys.

4.2 Stage Two

To go beyond that, you can add affine functions to s-box columns or you can permute s-box rows in a manner equivalent to permuting the input bits.

The choice would depend upon the relative strength of these two methods, along with the relation between their overheads and the resources available in a particular application. In our suggested implementation, using affine functions requires more storage while permuting the rows involves more computation. Neither operation looks prohibitively expensive in general, but either might be problematic in some environments.

For purposes of this paper, we will treat permuting the inputs as the next thing to add, and then go on to look at adding affine functions.

To permute the rows in a manner equivalent to permuting the input bits we add the following mechanism. We use a 256-row array, each row composed of an 8-bit index and a 32-bit output. We can rearrange rows as follows:

- put the (256*32)-bit s-box in the output part of the array;
- fill the index part with the 8-bit values in order from hex 00 to FF;
- operate in some way on the index parts (without affecting the 32-bit s-box rows) so as to give each row a new index;
- sort the 256 rows so that the index parts are again in order (00 to FF), moving the s-box rows along with the indexes so they are in a new order;
- discard the index portion.

This results in a cryptographically identical s-box with rows in the new order. The operations permitted in the third step for changing the 8-bit values are just those which are equivalent to permuting and inverting the s-box inputs. We can XOR a constant into all index rows or we can permute index columns. Neither operation alters the XOR difference between index rows, so cryptographic properties are conserved as shown earlier.

XORing a constant into each index row is of little benefit. This is also true of rotation, which uses only 3 bits per s-box (hardly enough to justify the overhead of sorting the s-boxes).

To operate usefully on the inputs, then, we should do a full permutation on the index columns. In code, this would need a function to permute 8-bit values under control of a 15-bit key. It would use 15 key bits per s-box.

At this point, we are using 52 bits per s-box, 208 bits in all, and are permuting both rows and columns or both inputs and outputs. Again, this would be quite sufficient for many applications.

4.3 Stage Three

We can, however, go further by adding affine functions to the columns.

There are exactly 512 affine Boolean functions of 8 variables. In theory, it would be possible to add a key-selected affine function to each s-box column, using 9 bits of key per column, or 1152 bits for a set of four 8×32 s-boxes, but this seems unacceptably expensive in practice.

Since the inverse of an affine function is also affine, we need only store half of the 512 possible functions to have them all available. Consider a (256*256)-bit Boolean array with affine functions in all columns and no two columns either identical or inverses. From this, create four 256×32 arrays. This can be done using $\log_2 \binom{256}{128}$ key bits, but implementing this would also be expensive. As a more practical alternative, using $\log_2 \binom{16}{8} \approx 13$ bits of key to select eight of sixteen "chunks of 16 columns" from the original array may be a reasonable compromise.

4.4 Putting It All Together

Given four 256×32 arrays of affine functions, we add a few operations inside the loop that runs through each s-box. The inner loop ends up as follows (in pseudo-C with "<<<" for rotation and "^" for XOR):

```
unsigned *s, *a ;   // pointers to s-box & affine array
unsigned char *p;   // pointer into index array
// initialize pointers here...
for( i = 0 ; i < 256 ; i++, s++, a++, p++ )
   {
   *s = (*s <<< k1) ^ (*a <<< k2) ^ k3; // 5+5+32 key bits
   *p = permute8(*p, k4) ;              // 15 key bits
   }
qsort(.....) ;
```

This uses 57 key bits per s-box inside the loop. With the 13 used outside the loop setting up the A-boxes, and another 15 used in re-arranging the original 8 s-boxes, we have 256 key bits in total exclusively used for s-box manipulations.

5 Further Work

Further work in this area can be done both on the theoretical side and on the practical side. For example, a formal characterization of the set of affine functions that can be added to an s-box without reducing its cryptographic strength (beyond the subset specified in this paper) would be of interest. As well, since factorials do not correspond to powers of 2, more precise practical specifications need to be given to convert expressions such as "$\log_2(8!)$" to bit lengths (i.e., it needs to be stated exactly which particular permutation corresponds to each value of a 15-bit key segment).

6 Conclusions

This paper has proposed the concept of using key-dependent s-box manipulations to strengthen specific block ciphers against attacks which depend upon knowledge of the s-box contents (such as linear and differential cryptanalysis

and their variations). The manipulations described include a permutation of the output bits, a permutation of the input bits, the addition of affine functions to the s-box columns, and the addition of a restricted set of affine functions to the s-box inputs. It has been shown (using the concept of dynamic distance [4]) that such manipulations do not degrade the cryptographic properties of carefully-constructed s-boxes, and therefore do not degrade the cryptographic strength of the corresponding ciphers with respect to existing analysis. On the contrary, it is possible that cryptographic strength may be substantially increased by such manipulations because the most effective cryptanalytic attacks to date would appear to require a significant exhaustive search phase in addition to their current complexity in order to be mounted against ciphers with such "hidden" s-box contents.

Some implementation considerations for this proposal were also discussed, and options were presented with respect to the level of complexity that might be employed in various environments.

References

1. C. Adams, "Constructing Symmetric Ciphers Using the CAST Design Procedure", Designs, Codes and Cryptography, vol.12, no.3, Nov. 1997, pp.71-104.
2. E. Biham and A. Biryukov, "How to Strengthen DES Using Existing Hardware" Advances in Cryptology - ASIACRYPT 94 Proceedings.
3. W. Diffie and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, vol.22, 1976, pp.644-654.
4. S. Mister and C. Adams, "Practical s-box Design", Workshop Record of the Workshop on Selected Areas in Cryptography (SAC '96), Queen's University, Kingston, Ontario, Aug. 1996, pp.61-76.
5. R. Rivest, A. Shamir, and L. Adelman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM, v. 21, n. 8, Feb. 1978, page 120.
6. B. Schneier, Applied Cryptography, Second Edition, John Wiley & Sons, 1996.