Tatra Mt. Math. Publ. **25** (2002), 39–66

TATRA MOUNTAINS
Mathematical Publications

SCALABLE BLOCK CIPHERS BASED ON FEISTEL-LIKE STRUCTURE

VALÉR ČANDA — TRAN VAN TRUNG

ABSTRACT. Scalability of symmetric ciphers is still not as natural as for public-key cryptosystems. Most of the current block ciphers do not support freely variable block and key lengths, nor do they allow free memory-speed balancing. In this work we discuss possible approaches to a scalable block cipher design and propose a new scalable scheme related to the Feistel structure. We analyze the desirable properties of the used building blocks and present one possible implementation. In addition to a security discussion of the cipher we present some experimental results regarding its statistical properties and efficiency. A nice property of the new cryptosystem is its unlimited scalability and the possibility of a memory-speed tradeoff by a controlled security level.

1. Introduction

Scalability is a desirable property of cryptographic primitives. Mathematically well-founded scalable cryptographic primitives are useful not only because they enable us to adjust an encryption configuration according to security and performance requirements for a specific application, but also because they make it possible to adapt our cryptosystems to a new situation in the event of a breakthrough in mathematical theory or computer design.

An algorithm is called *scalable*, if its basic parameters are alterable by design. For instance, a block cipher is scalable, if its block length n and key length k can be adjusted without redefining or extending the cipher. It is particularly important that the security properties of a cryptographic algorithm change in some predictable and naturally expectable way, when one of its parameters has

2000 Mathematics Subject Classification: 94A60.
Keywords: block ciphers, scalability, unbalanced Feistel networks, key expansion algorithm, random key-dependent s-boxes.

been altered. For example, when varying the block length of an iterative cipher, we should be able to tell easily how many rounds need to be performed to achieve a specified security level.

While most public key cryptosystems are intrinsically scalable[1], the scalability of symmetric block ciphers is not that common. The first modern block ciphers like DES were simply not scalable. The candidates for the recent cryptographic standard AES were already *required* to support three different key lengths and some of them (e.g., RC6 [5] or Rijndael [6]) supported even multiple block lengths, yet this is not what we understand by full scalability.

According to their scalability, block ciphers can be classified into the following four categories:

- *strongly scalable*—ciphers which by design support any combination of block length n and key length k (both in bits).
- *fully scalable*—ciphers with some minor restrictions regarding the format of n and k (e.g., n must be a power of 2 and k must be a multiple of 32) but without upper limits for these values.
- *partially scalable*—ciphers supporting only a small finite set of possible values for n and k.
- *not scalable*—ciphers where n and k are fixed by design.

Most of the AES candidates are only partially scalable ciphers. We think that in the future, block cipher design will be moving towards full scalability. Strongly scalable ciphers might be at least of academic interest, even if there is no practical need for them.

Full scalability is especially relevant for a software implementation, because only a generic software implementation of a scalable algorithm can exploit all its capabilities. Hence, it is particularly important to design scalability in such a way that it can be effectively implemented in software. Hardware solutions are usually more restricted because of the hard-wired algorithms and limited resources (e.g., smart cards) and can therefore usually implement only a constrained, fast version of an otherwise fully scalable algorithm.

In what follows we first discuss general approaches to a scalable cipher design. Then we propose our general scalable scheme which will lead us, after some analysis, to a concrete scalable cipher. Our cipher example is more software-oriented but the same ideas might be implemented in a hardware-oriented setting as well. At the end of the paper we present some experimental results regarding the security and efficiency of our cipher.

[1] For example, the key length of RSA can be increased simply by choosing longer primes p and q.

2. Scalability approaches for block ciphers

2.1 Scalable key length

Normally, iterative block ciphers use a main key for generating a sequence of round keys and, possibly, some key-dependent tables. Non-iterative block ciphers based on some specific mathematical objects (e.g., permutations, polynomials, bases, etc.) attempt to generate the appropriate randomly looking objects depending on the key. In both cases the *key expansion algorithm* is a pseudo random number generator which attempts to make all key-dependent components of the cipher depend on as many bits of the main key as possible.

Flexible key expansion algorithms, like that of RC6, support variable lengths of the main key by design. In some of them it is necessary to generate more round keys (and therefore execute more rounds) if we want to incorporate a longer main key with the same quality. In general, it is possible to create a generic key expansion algorithm with parameters k and $k' \geq k$ which expands an input sequence of k bits (the main key) to a unique randomly looking sequence of k' bits (the round key material). Consequently, it is also possible to separate the key expansion algorithm from the encryption algorithm. Being able to choose an arbitrary combination of these two algorithms is not very crucial if we want to implement a cryptographic standard like AES, because simplicity and portability are more important than versatility in this case. The separation, however, might be useful for some high-security applications. For example, precomputation attacks become much harder when the communicating parties settle on one of many possible encryption configurations just before transmission.

2.2 Scalable block length

Scalability of block length n can be achieved either through a modification of the size of the *basic operations* (*primitives*) used, or through a modification of the *encryption scheme.* A 64-bit Feistel cipher in Figure 1A, whose round operates on two 32-bit sub-blocks, might be scaled to a 128-bit version either by "doubling the size" of the round function from $f\colon \{0,1\}^{32} \to \{0,1\}^{32}$ to $F\colon \{0,1\}^{64} \to \{0,1\}^{64}$, as shown in Figure 1B, or by doubling the number of processed sub-blocks from 2 to 4 (Figure 1C).

In either case the scaling method does not automatically ensure that the larger version of the cipher will have security properties equivalent to the original ones. In general, scalability through the primitives is more likely to retain equivalent security properties than scalability through the scheme. For example, if our 64-bit cipher uses only three simple mathematical primitives, say, an 8×8 bit key-dependent random S-box, addition modulo 2^{16} and 16-bit bitwise XOR, then the 128-bit version using the same structure with double-length operations

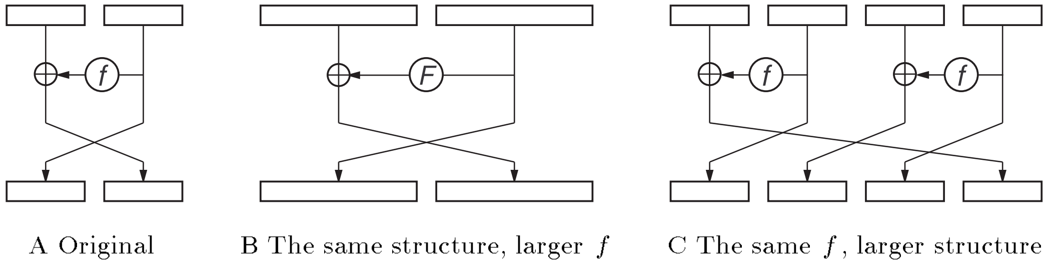

FIGURE 1. Scaling approaches.

(i.e., 16×16 bit S-box, addition modulo 2^{32} and 32-bit XOR) will very probably have the expected security properties.

Ensuring the compatibility of security properties by scaling through the structure is less trivial. One can, for example, define a chain of structures $s_1, s_2, s_3, \ldots$ and show by induction that if s_i has some security properties (e.g., the avalanche property), then s_{i+1} has these properties as well. If this induction works for all important security properties, the chain of structures can be used for cipher scaling. It is usually not enough to change only the "width" of the structure (e.g., number of sub-blocks). The "depth" of the structure (e.g., number of rounds or complexity of a round) also has to be increased to retain equivalence in security properties.

Even though scaling through the primitives is more straightforward, scalability through the scheme is usually more practical for larger cipher versions. For example, if we wish to create a 256-bit version of the Feistel cipher in the same way as described above, we would fail to implement a 32×32 bit S-box. The reason for our problems is the exponentially growing complexity of large non-linear primitives. Scaling through the scheme is more practicable in that case.

Of course, combinations of the two aforementioned scaling approaches are also possible. It might be more convenient in some applications to use "a somewhat larger" structure together with "a somewhat larger" primitives, rather than a completely doubled structure or completely doubled primitives. A memory-time tradeoff with a constant security level might be achieved this way.

3. Feistel networks

Many 64-bit iterative block ciphers (e.g., Lucifer, DES, FEAL, LOKI, GOST, etc.) are based on the Feistel network (FN) because of its simplicity and guaranteed self-invertibility. When we denote the block length of a cipher by n, the round function of an FN utilizes a function of the form $f\colon \{0,1\}^m \to \{0,1\}^m$, where $m = \frac{n}{2}$ (see, e.g., Figure 1A). As the block length of an FN can not be easily scaled through the primitives[2], the classical FN is not very suitable for constructing ciphers with block length $n > 64$. For this reason some of the AES candidates introduced *extended Feistel networks* (EFN). These modified structures use four equally-sized sub-blocks and an $\frac{n}{4}$-bit function f. Even if EFN were originally not constructed to provide full scalability, a generalization of the number of their sub-blocks can be used for scaling the block length. For instance, when using a fixed $f\colon \{0,1\}^m \to \{0,1\}^m$, the schemes of Cast256 and Mars can be principally scaled to any block lengths of the form $n = s \cdot m$, $s \in \mathbb{N}$, and the RC6 scheme can be scaled to any $n = s \cdot m$, s even. This is a typical scaling through the structure. Unfortunately, we are aware of no papers discussing either the cryptographic properties of EFN with $s > 4$ in general, or presenting concrete scalable block ciphers based on these structures. General properties of EFN with $s = 4$ based on an ideal f were analyzed in [1].

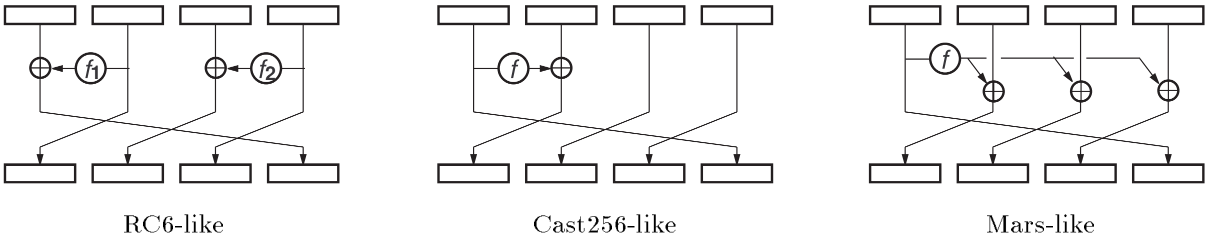

FIGURE 2. Extended Feistel networks.

Another generalization of the Feistel structure are the so-called *Unbalanced Feistel Networks* (UFN) introduced in [2]. This generalization works with two sub-blocks of different lengths, i.e., the original Feistel structure (Figure 3A) processing two $\frac{n}{2}$-bit halves is generalized to a structure with two input blocks of lengths s and t bits respectively ($n = s + t$).

Such a structure is called an *s-on-t UFN*. The s-bit sub-block is called the *source*, and the t-bit sub-block the *target*. A UFN is called *source heavy* when $s > t$ (Figure 3B) and *target heavy* when $t > s$ (Figure 3C). The special case of UFN for $s = t$ is the classical FN. A UFN is said to be *homogenous* if its round function is identical in each round (except for the round keys) and *heterogenous* otherwise.

[2] Doubling the "size" of f while preserving its cryptographic complexity requires exponentially more resources. Hence, designing and implementing a good $f\colon \{0,1\}^{64} \to \{0,1\}^{64}$, required for $n = 128$, is much harder than constructing $f\colon \{0,1\}^{32} \to \{0,1\}^{32}$, used for $n = 64$.

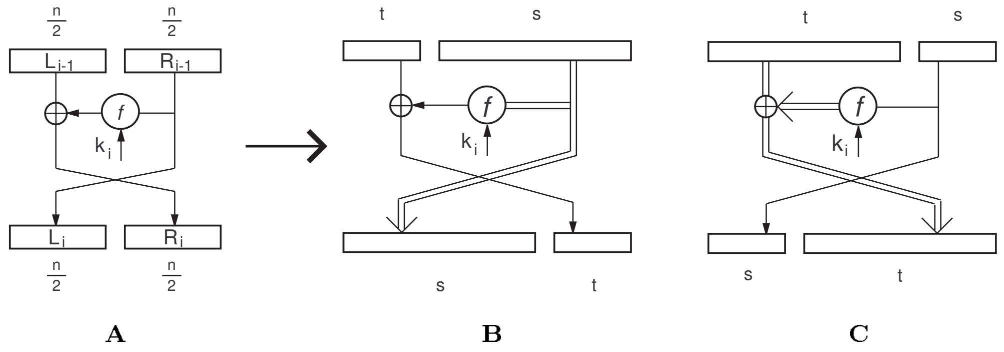

FIGURE 3. Unbalanced Feistel networks.

Even if the main goal for introducing UFN was generalizing the FN rather than implementing scalability, these structures can be used to provide fully scalable block length as well. For instance, the block cipher BEAR [3], based on UFN, provides full scalability by admitting variable s in its first source-heavy round. BEAR and the other two ciphers presented in [3] are, however, not conventional block ciphers, because they are built from components that must be cryptographically secure themselves. Such a design (a meta-cipher) is in fact a conversion from one cryptographic primitive into another. Besides the three meta-ciphers introduced in [3] we know of no other fully scalable block cipher proposals based on UFN.

4. A scalable Feistel-like cipher

4.1 The encryption scheme

Our scalable cipher design, presented in this section, was initially motivated by symmetric encryption based on *group bases* (see, e.g., [8]). It can, however, be described even without the knowledge of that theory, because it is similar to the Feistel networks.

Let n denote the *block length* of a cipher, k the *key length*, and r the *number of rounds*. An additional characteristic parameter $m \le \frac{n}{2}$ of our encryption scheme will be called the *segment length*. A consecutive sequence of m bits $x_{c\cdot m+0}, \dots, x_{c\cdot m+(m-1)}$, $c \in \mathbb{N}$, of a binary vector x will be called a *segment*. The leftmost segment of a binary vector x will be denoted by x_L, and the rest of the vector by x_R. One round of our encryption scheme is displayed in Figure 4. The binary operations $\odot$, $\oplus$, and $\boxplus$ are not particularly important at this point. The scheme will work with any three invertible binary operations.

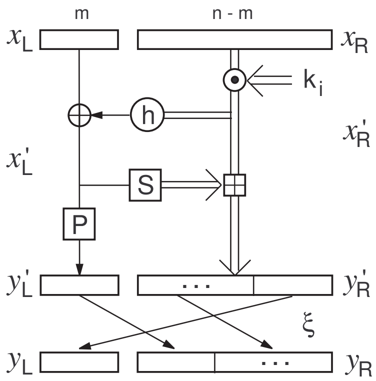

FIGURE 4. Round structure.

The round works as follows. An n-bit input x is split into two parts x_L and x_R. First, a round key k_i is added to x_R and the resulting value $x'_R = x_R \odot k_i$ is fed into the hash function[3] h which produces an m-bit output. The output of h is added to x_L and the resulting m-bit value $x'_L = x_L \oplus h(x'_R)$ is transformed by a key-dependent random S-box S into a unique $(n-m)$-bit value. The mapping $S\colon \{0,1\}^m \longrightarrow \{0,1\}^{n-m}$ is realized by a table of 2^m uniformly distributed random $(n-m)$-bit values s_i, thus, a computation of $S(i)$ requires a single table lookup $S(i) = s_i$. The output of S is added to x'_R which results in $y'_R = x'_R \boxplus S(x'_L)$. The x'_L is furthermore transformed by a permutation S-box P which maps it onto a unique m-bit value y'_L. The mapping $P : \{0,1\}^m \longrightarrow \{0,1\}^m$ is represented by a table of 2^m key-dependent random m-bit values p_i, such that $p_i \neq p_j$ for all $i \neq j$. (P is in fact a permutation of 2^m elements.) The intermediate output of the transformations above is an n-bit vector $y' = y'_L \mid\mid y'_R$. The final output y of one round is obtained by a ξ-bit rotation of y', i.e., $y = \mathrm{rot}_\xi(y')$.

Given a fixed pair of tables (S,P) we will denote the mapping performed by one round as $y = R^{(S,P)}(x, k_i)$. An encryption $y = e_K(x)$ is performed by r subsequent executions of a round, i.e., $x_0 = x$, $x_i = R^{(S,P)}(x_{i-1}, k_i)$, for $i = 1, \ldots, r$, and $y = x_r$. The rotation by ξ bits performed at the end of the last round can possibly be omitted (or undone) because it does not contribute to the cryptographic strength of the cipher.

The decryption operation $x = d_K(y)$ can be performed by executing r inverse rounds: $x_r = y$, $x_{i-1} = R^{(S,P)}_{\mathrm{inv}}(x_i, k_i)$, for $i = r, \ldots, 1$, and $x = x_0$, where $R^{(S,P)}_{\mathrm{inv}}(y, k_i)$ denotes the sequence of operations: $y' = y'_L \mid\mid y'_R = \mathrm{rot}_{-\xi}(y)$; $x'_L = P^{-1}(y'_L)$; $x'_R = y'_R \boxminus S(x'_L)$; $x_L = x'_L \ominus h(x'_R)$; $x_R = x'_R \oslash k_i$; and

[3]This is *not* a cryptographic hash function. It is just an ordinary unkeyed hash function like CRC or similar checksums.

$x = x_L \,||\, x_R$. The binary operations $\oslash$, $\ominus$ and $\boxminus$ denote the operations inverse to $\odot$, $\oplus$, and $\boxplus$ respectively.

Obviously, the encryption scheme is similar to a heterogenous UFN as demonstrated in Figure 5.

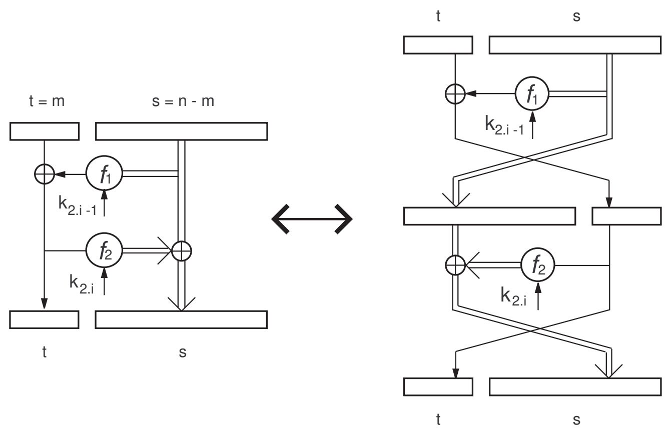

FIGURE 5. Related to our scheme.

One round of our cipher corresponds to two rounds of a UFN—the first one source-heavy and the second one target-heavy. The hash function h corresponds to f_1 and the S-box S corresponds to f_2, but unlike f_1 and f_2 the mappings h and S do not depend directly on the round keys. Other specific attributes of our scheme are the operations P and ξ which are not present in the UFN. These two elements are implied by the structure of an *extended group basis*. For more information about how our structure has been derived from group bases we refer to Section 6 of [9].

We would like to stress the fact that the S-boxes S and P, used in our design, are large, pseudorandom, and key-dependent. This approach does not only increase the adversary's uncertainty about the round function, but also makes it harder to find strong differentials and input-output sums that could be used for differential and linear cryptanalysis respectively. Moreover, a random key-dependent S-box is most likely to be resistant against all attacks, including new ones that might be invented in the future. A more detailed discussion on the usage of key-dependent S-boxes can be found, for instance, in [7] and [11].

4.2 Scalability

The encryption scheme based on the round function presented in Figure 4 is suitable for implementing fully scalable cryptosystems. The key expansion algorithm can be implemented in a straightforward way by using a pseudorandom number generator (PRNG) which, given a seed of an arbitrary length k, generates a unique pseudorandom sequence of an arbitrary length $k' \geq k$. Such a sequence can be used for constructing k_i, S and P independently on the values of the parameters n, m, k, and r. In order to provide full scalability, the particular function h used must be able to process inputs of variable length (in fact, virtually all hash functions can). Moreover, the binary operations $\odot$, $\oplus$, and $\boxplus$ must be computable for operands of any length. Based on these premisses, our design can provide a cipher instance for any combination of the parameters n, m, k, and r.

Depending on the particular building blocks used (i.e., PRNG, h, $\odot$, $\oplus$, $\boxplus$, and ξ), one can implement various ciphers based on our scheme. Hence, the scheme can be understood as a framework for constructing scalable block ciphers. In what follows we discuss the variable building blocks in more detail and present one concrete scalable block cipher based on this scheme.

4.3 Key expansion algorithm

The task of the key expansion algorithm of our cipher is generating randomly-looking objects S, P, and k_i from a main key K. The PRNG used for that purpose should accept seeds of variable length and should produce a pseudo-random number (PRN) sequence with very strong statistical properties. The larger the amount of the expanded key material needed (this amount depends mainly on the value of m), the more important is the quality of the PRNG. The maximum seed length and period length of the generator delimit the maximum achievable key space. The efficiency of the generator is important for providing short key-setup delays.

The criteria above suggest, on one hand, not using very simple generators (e.g., linear congruential generator) because of the security requirements and, on the other hand, not using very computationally intensive generators (e.g., Blum-Blum-Shub generator) because of efficiency. Supposing the encryption scheme itself is secure (i.e., it is not possible to reveal parts of S, P or k_i by less than $\min(2^k, 2^n)$ trial encryptions), the PRNG does not necessarily need to be cryptographically secure. Note that the output of the PRNG is hidden within the scheme and, thus, the generator can not be attacked directly.

A generator suitable for our key expansion is the lagged Fibonacci generator with Lüscher's approach [12]. It not only generates a PRN sequence with very good statistical properties, but it also enables a scalable seed length. The period of the generated sequence is very long. For instance, the combination of lags (37, 100) and word length 32 bits guarantees that the period of the sequence will be 2^{131} and seeds of length up to 3200 bits can be used. These values can be further improved by changing the lags [12].

To strictly prevent any hypothetical attacks based on a reconstruction of the secret PRN sequence, one might use a cryptographically secure PRNG as well. Nevertheless, cryptographically secure generators are significantly slower than normal generators while providing (statistically seen) "equally good" output. Moreover, cryptographically secure generators are usually based on some other cryptographic primitives (e.g., block ciphers etc.) that are supposed to be cryptographically secure themselves. Hence, usage of such a PRNG would simply make our cipher a conversion from one cryptographic primitive to another.

As we want to provide for practical (rather than provable) security, we suggest using a statistically strong (rather than a cryptographically strong) PRNG. If necessary, the reconstruction of the PRN sequence can be hindered by discarding some parts of the sequence (e.g., by using only the most significant byte of every 32-bit word provided by the lagged Fibonacci generator). Nevertheless, when performing a sufficient number of rounds, no parts of the PRN sequence can be revealed anyway.

4.4 Function h

The purpose of the function h is to compute an m-bit hash value for an $(n-m)$-bit input vector x. The output of h should be balanced, i.e., when computing $h(x)$ for all possible inputs x, every possible output value should appear roughly $\frac{2^{n-m}}{2^m}$ times. Furthermore, h should be highly non-linear (see, e.g., [10]) and the output of the composed function $h(x_R \odot k_i)$ should not be predictable without the knowledge of k_i.

The simplest method for software implementation of h is chaining of a function $p : \{0,1\}^m \to \{0,1\}^m$ as shown in Figure 6A.

The binary operation $\otimes$ used here should be an effectively computable group operation on $\{0,1\}^m$, like m-bit XOR, addition mod 2^m, etc. The function p must be bijective to ensure that all 2^m possible output values appears equally frequently. The strongest candidate for p is a key-dependent random permutation of 2^m elements, because it makes the output of h unpredictable without knowledge of the key. (Note that the table P might be reused for that purpose.)

However, the chaining displayed in Figure 6A is not very efficient. Especially for small m it needs too many serial steps to compute the hash value. Significantly faster implementations of h are possible with more parallelism. A tree structure like in Figure 6B is most suitable for hardware. A two-level chaining shown in Figure 6C can perform well in software. In the first step one computes an intermediate m'-bit hash value using usual chaining as in 6A. The constant m' should be chosen as the maximum word length which supports an effective execution of $\otimes$ on the underlying platform (usually 32 bits). In the second step one reduces the m'-bit intermediate hash to the final m-bit value. When $m = 8$

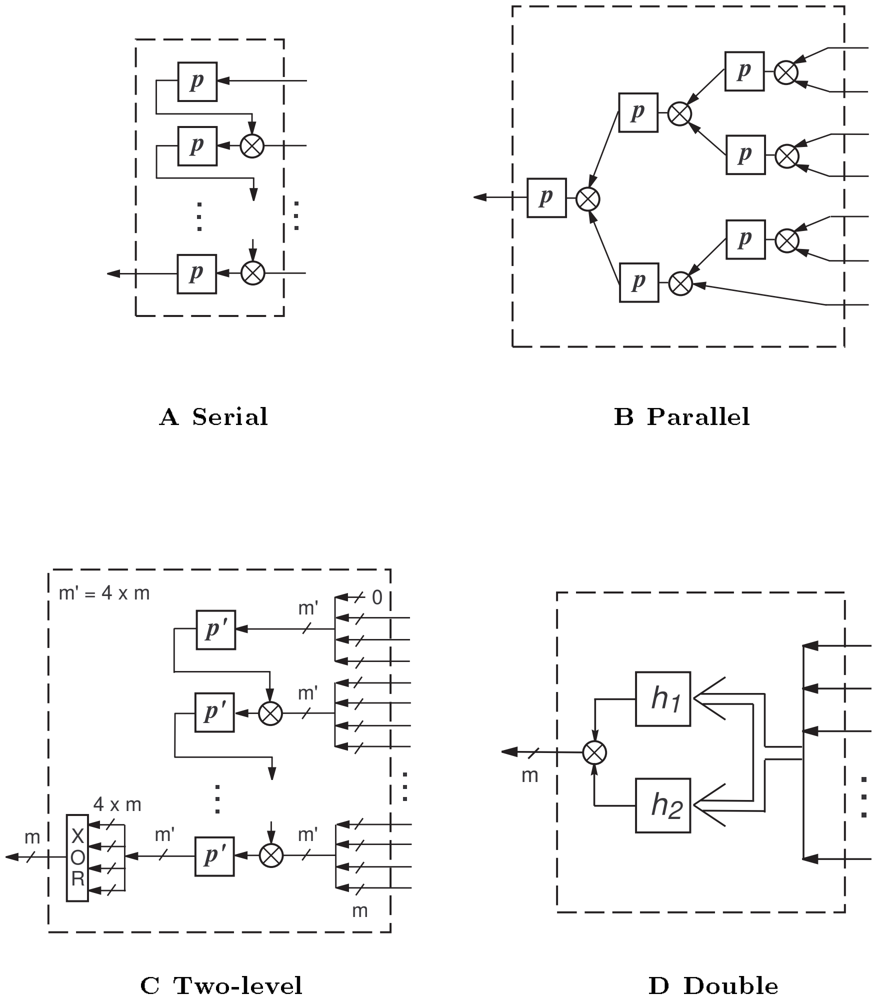

FIGURE 6. Possible implementations of h.

and $m' = 32$, say, the word-wise computation of h can be about four times faster than the byte-wise version.

Among the fastest candidates for the bijection $p'\colon \{0,1\}^{m'} \to \{0,1\}^{m'}$ are some simple non-linear functions like:

- $f(x) = \mathrm{rot}_c(x)$, where c is relatively prime to m',
- $f(x) = cx \pmod{2^{m'}}$, where c is an m'-bit prime,
- $f(x) = x(2x+1) \pmod{2^{m'}}$, this function is used in RC6, for example.

Because of the secret value k_i (see, e.g., Figure 4) it is not possible to directly manipulate the output of h by varying x_R. Nevertheless, the designs A and B presented in Figure 6 make it at least possible to achieve all 2^m possible outputs of h with just 2^m different inputs x_R. A similar manipulation (with $2^{m'}$ instead of 2^m inputs) is possible in design C. Although this property of h does not seem to be particularly useful for an attack[4], it can be, if desired, easily avoided, e.g., by using a construction based on two independent sub-hashes shown in Figure 6D. For instance, when h_1 is computed according to Figure 6C and h_2 is defined as $h_2(x) = h_1(\mathrm{rot}_c(x))$, where c is relatively prime to m', any contrived manipulation of the output $h(x) = h_1(x) \otimes h_2(x)$ becomes much harder.

4.5 Binary operations

Until now there have been four binary operations used in our cipher:

- $\odot$, acting on $\{0,1\}^{n-m}$, used for the round-key addition,
- $\otimes$, acting on $\{0,1\}^{m'}$, used for the chaining in h,
- $\oplus$, acting on $\{0,1\}^m$, used for adding the output of h to x_L, and
- $\boxplus$, acting on $\{0,1\}^{n-m}$, used for adding the output of S to x'_R.

These operations must be efficiently computable on a given platform for all allowed combinations of n, m, and m'. Furthermore, the operations $\odot$, $\oplus$, and $\boxplus$ must be easily invertible. For example, there must exist an efficiently computable binary operation $\ominus$ such that $a \oplus b \ominus b = a$ for all $a, b \in \{0,1\}^m$.

It has been shown in [4] that using operations from incompatible mathematical groups is advantageous for the quality of a cipher. Whenever an output of a group operation o_1 is used as an input into operation o_2, the two operations should not be associative and distributive [4].

In accordance with the requirements above we suggest alternating XOR with an integer addition mod $2^{m'}$ in the following way:

- $\odot$-bit-wise XOR,
- $\otimes$-integer addition (mod $2^{m'}$),
- $\oplus$-bit-wise XOR,
- $\boxplus$-word-wise addition (mod $2^{m'}$) of two vectors.

The binary XOR and the modular $+$ are included in the instruction sets of virtually all processors and can be thus performed very efficiently on any platform. The operations $\odot$ and $\boxplus$ process their operands as arrays of m'-bit words. This improves the speed in comparison with a segment-wise processing and, in case of $\boxplus$, it also improves the cryptographic properties. An m'-bit addition ($m' > m$) creates more complex dependencies between the input and output bits than an m-bit addition. Such a word-wise processing of vectors can

[4] It would be much more useful for an adversary if he could *minimize* the number of different outputs of h when varying x_R, because that would decimate the randomizing effect of S. *Maximizing* the number of outputs is the opposite of what the adversary wants.

either be performed directly by a processor (e.g., using the MMX instruction set of the Pentium CPU family), or else, it can be easily implemented in software with a simple loop.

4.6 Bit rotation

The purpose of the bit rotation at the end of a round is to ensure that in different rounds a segment of the encrypted n-bit vector is processed in different ways. Because of the rotation a particular segment of the plaintext is acting in some rounds as x_L, in the other rounds as a part of x_R, and is always influenced by different columns of the random table S. The number of different ways in which an input segment can be transformed into an output segment is significantly increased in this manner.

In order to maximize this effect, the value ξ should be relatively prime to n. In terms of cycles (as discussed, e.g., in [2]) this will make our scheme a *prime network*. Furthermore, to improve the diffusion of the word-wise operation $\boxplus$, ξ should be roughly equal to $\frac{m'}{2}$ and should be relatively prime to m'. That will ensure that every bit of the input will alternately appear on both lower-order and higher-order positions within different m'-bit words. These regular bit exchanges will create more complex dependencies between the bits, because without them the modular addition used in $\boxplus$ would only spread the information within every m'-bit word from the lower order bits to the higher order bits, but not vice-versa.

When supposing a fixed m' for a given platform, we can make a universal suggestion that works fine for all usual values of n. For instance, when $m' = 32$ and n is even, a suitable rotation length is $\xi = 17$. Analogously, $\xi = 31$ might be used when $m' = 64$.

4.7 Cipher example

In Appendix A we present a C code sample of a fully scalable cipher based on our scheme. The building blocks of the cipher are implemented according to the discussion in Sections 4.3 to 4.6. The key expansion algorithm is based on the lagged Fibonacci generator with Lüscher's approach (Section 4.3), the hash function based on Figure 6C uses p' of the form $p'(x) = c \cdot x \mod 2^{m'}$, and the binary operations are implemented according to Section 4.5. The values of the constants m' and ξ are 32 and 17 bits respectively.

5. Security considerations

The most important question regarding security of our encryption scheme is the number of rounds that need to be executed for a given pair (n, m) in order to ensure practical security. We consider the cipher as practically secure when the number of trial encryptions needed to reconstruct the decryption function is equal or larger than the minimum of the following two values: the number of all possible plaintexts and the number of all possible keys. We attempt to find an answer to this question in two ways—an analytical and an experimental one.

5.1 Analytical approach

Referring to Figure 4, we now analyze the properties of our encryption scheme based on statistically strong components. Let us suppose that each of the 2^m possible outputs of h is produced with the same probability. An h based on Figure 6 meets this requirement whenever the used function p (resp. p') is a bijection. Let us furthermore suppose that S produces a random $(n-m)$-bit vector for each of its possible inputs, and P produces a unique random m-bit vector for each of its possible inputs. S and P meet these requirements with very high probability when they have been generated by a statistically strong PRNG. On the basis of these assumptions, the sum $x_L \oplus h(x'_R)$ takes on every possible value with probability $\frac{1}{2^m}$ and, consequently, both y_L and y_R will change in one of 2^m possible ways randomly, whenever either x_L or x_R have changed. From a statistical point of view, the transformations performed by different rounds of our cipher can be considered as independent, because the round keys k_i are generated at random, and the operation $\odot$ is incompatible with both $\otimes$ and $\boxplus$. Consequently, the number of different random ways in which an input x can be modified into an output y by r rounds is $2^{r\cdot m}$. To reconstruct a mapping consisting of $2^{r\cdot m}$ point-wise independent random pairs of input and output values one needs to encrypt $2^{r\cdot m}$ different inputs. This number is larger than the number of *all possible* inputs when $r \geq \frac{n}{m}$. It follows that $\frac{n}{m}$ is the minimal practically secure number of rounds. When $k < n$, the complexity $O(2^{r\cdot m})$ must only be greater than the complexity of an exhaustive key search $O(2^k)$ and, hence, the minimal practically secure number of rounds is $\frac{k}{m}$ in that case. It follows that at least $\min\left(\lceil \frac{n}{m} \rceil, \lceil \frac{k}{m} \rceil\right)$ rounds should be executed for a given configuration (n, m, k).

The analysis above is based on assumptions which might not always be fulfilled. For instance, even a strong PRNG can sometimes generate an array S whose rows S_i and S_j are equal for some i and j. A cipher using such an S can not ensure 2^m possible differences between x_R and y'_R in every round and, hence, should perform more than just $\min\left(\lceil \frac{n}{m} \rceil, \lceil \frac{k}{m} \rceil\right)$ rounds. Even though the probability of generating such an unfortunate S is very low for typical values of n and m (e.g., 0.25×10^{-31} for $n = 128$, $m = 8$, [9]), we suggest performing one additional round[5] to provide a certain security margin. Consequently, the

[5] Note that the cryptographic contribution of this one additional round in our cipher is at least equal to a contribution of *two* additional rounds in a UFN (Fig. 5).

total number of rounds which we propose for a given configuration (n, m, k) is

$$r = \min\left(\left\lceil \frac{n}{m} \right\rceil, \left\lceil \frac{k}{m} \right\rceil\right) + 1\,.$$

5.2 Statistical approach

Statistical analysis based on randomness testing is another technique for evaluating of the quality of a block cipher (see, e.g., [13]). We have used this experimental approach for evaluating the secure number of rounds. Hereby we encrypted a long aperiodic sequence, once without and once with a small difference Δ added to every block. Then, we measured the randomness of differences between the two resulting ciphertexts (Figure 7).

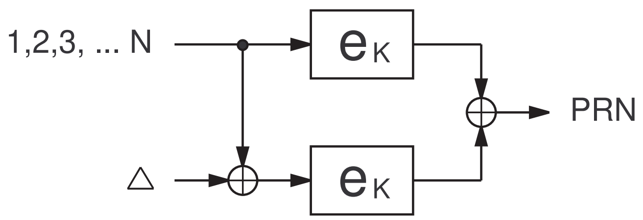

FIGURE 7. Randomness test setup.

This approach statistically simulates a differential analysis. We repeated the experiment for several different keys and differences. The randomness of every output was measured by the DieHard battery of randomness tests [14] and classified as passed, or suspect. Because every sequence of uniformly distributed random bits should appear with the same probability, even a perfect PRNG sometimes generates a sequence which “fails” a randomness test. In the particular case of DieHard test suite, on average, 0.0023 of the sub-tests will falsely suspect a sequence to be “not random”, even if it were. The rate of such false suspections will stay between 0.0006 and 0.0040 for virtually all strong PRN sequences. The graph in Figure 8 shows the dependence between the number of rounds and the probability of producing a suspect result.

It is obvious that the probability sinks exponentially with the number of rounds and after a specific number of rounds it keeps oscillating around 0.0023. The curves in the graph show, for instance, that the configuration ($n = 64$, $m = 8$) is statistically secure after four rounds which is about half of the suggested value $\lceil \frac{n}{m} \rceil + 1$. From the statistical point of view the suggested security margin (twice as many rounds as needed) appears to be robust.

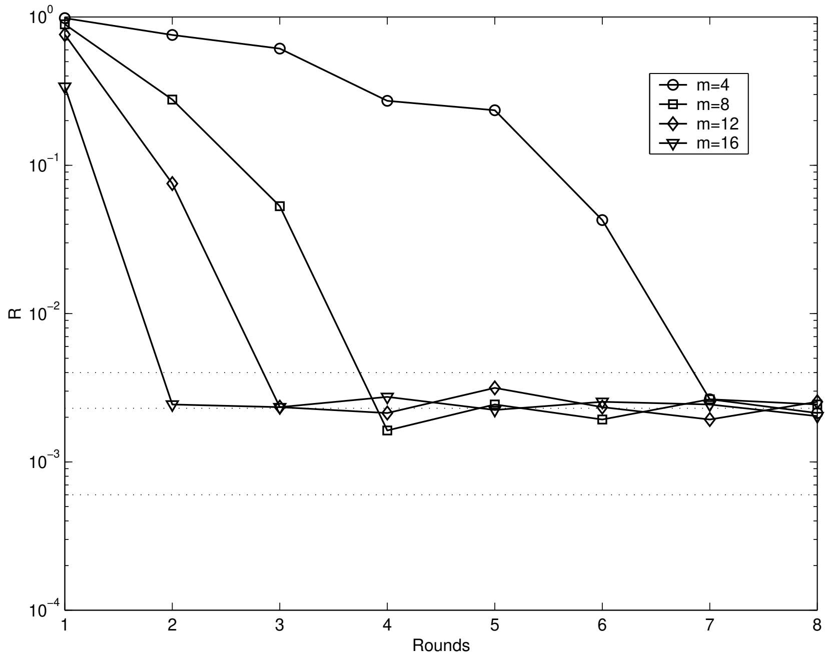

FIGURE 8. Round profile for $n = 64$, $m = 4, 8, 12, 16$.

5.3 Adjustability

The analysis in Section 5.1 suggests that the security of our cryptosystem can be adjusted in two different ways:

1. By increasing the number of rounds. This approach does not increase the memory requirements. Security is improved at the cost of speed.
2. By increasing the segment length m. This approach does not slowdown the cipher. Security is improved at the cost of memory.

The experimental results presented in Section 5.2 have confirmed this behavior as well.

Obviously, our cipher enables a tradeoff between security, memory, and speed. Any one of these three characteristics can be improved at the cost of the other two. In this way the cipher can be adjusted for usage in various environments. For instance, when implementing a 64-bit version of our cipher on a smart card, we can use $m = 8$ and perform 9 rounds ($9 = \lceil \frac{64}{8} \rceil + 1$). The resulting memory requirements of roughly 2 KB will comfortably fit into the restricted memory space of a smart card. On the other hand, when implementing a 64-bit cipher on a modern PC, we can rather use $m = 16$ which ensures the same security with just 5 rounds ($5 = \lceil \frac{64}{16} \rceil + 1$). The resulting memory requirements of 512 KB which would be too high for a smart card will cause no problems in this case.

Besides the full scalability, this is another attractive property of our encryption scheme.

6. Efficiency

The achievable encryption speed substantially depends on the number of performed rounds. This number can be reduced without compromising the security when a larger m is used. However, the size of the S-box as well as permutation P grow exponentially with larger m. For example, we need altogether only 4 kB for the configuration $(n = 128,\ m = 8)$, 64 kB for $(n = 128,\ m = 12)$ but already 1 MB for $(n = 128,\ m = 16)$. One should basically use m as large as possible (i.e., a value which does not cause any implementation problems on a given platform) because of both speed and security. Nevertheless, the combination of $r \cong m \cong \sqrt{n}$ appears to be a reasonable memory-speed tradeoff for most applications.

We have implemented a generic version of our algorithm in C++ and tested the encryption speeds on a Pentium II 350 MHz system. In spite of the fact that our implementation preferred versatility to performance, the achieved throughput of, e.g., 3682 KB/s with $(n = 128,\ m = 12,\ r = 12)$ would make us a middle class among the AES candidates [15]. We believe that a highly optimized C code written specially for one particular configuration could do much better. A speed up by factor of 2 is thinkable. Some other measured encryption speeds and the corresponding memory requirements are listed in Table 1.

TAB. 1. Throughput and memory requirements.

Configuration			Throughput	Memory Req.
n	m	r	KB/s	KB
64	8	9	3488	2
64	12	7	3986	32
64	16	5	4542	512
128	8	17	2849	4
128	12	12	3682	64
128	16	9	3752	1024
256	16	17	2248	2048
512	16	33	1408	4098

7. Conclusions

We have discussed the scalability of block ciphers in general and proposed a new scalable scheme for a block cipher design. Our main goal was to keep the proposal general and place the emphasis on the scalability and efficiency of the scheme, rather than to design a concrete standard-like cipher, optimal on all platforms.

Our design is similar to a heterogenous UFN. We have discussed desired properties of the used building blocks and suggested possible realization which led us to a particular fully scalable block cipher, just one of many possible examples. Our encryption scheme not only enables to scale for block and key length, but also makes it possible to find an appropriate memory-speed-security tradeoff for a particular application.

According to our experimental results, the cipher appears to have robust security properties combined with a sufficient encryption speed. Nevertheless, we believe that many similar scalable schemes can be designed—some of them certainly more effective than ours. With this proposal we wish to stress the need for fully scalable block ciphers and to initiate more discussion on the topic. We hope to see more mathematically well-founded scalable designs in the future.

A. Source code example

This appendix contains core routines of our cipher implemented in C. Complete code in both C and C++ can be found on `http://www.exp-math.uni-essen.de/~valer/Iterative`.

```
/*== General definitions ====================================================*/
#include<stdlib.h>
#include<memory.h>

typedef unsigned char BYTE;
typedef unsigned int  UINT;
typedef int           BOOL;
#define FALSE         0
#define TRUE          1

/*== Type and constant definitions ==========================================*/
#define MYWORD_LEN    4
#define MYWORD_BITS   ((MYWORD_LEN) <<3)
#define MYWORD_MASK   (((UINT)-1) >> ((sizeof(UINT) << 3) - MYWORD_BITS))
#define SHIFT         17
#define COMPL_SHIFT    (MYWORD_BITS - SHIFT)
```

```
#define SEGM_BITS     12
#define HASH_MASK     ((MYWORD)(((MYWORD)-1) >> (MYWORD_BITS - SEGM_BITS)))
#define INV_HASH_MASK ((MYWORD) HASH_MASK)
#define TABLE_ROWS    (1 << (SEGM_BITS))

typedef UINT  MYWORD; // The optimal type for fast word operations (32 bits)
typedef UINT  HASH;
typedef struct {
    HASH   Perm[TABLE_ROWS];     // Permutation P
    HASH   InvPerm[TABLE_ROWS];  // The inverse permutation P^{-1}
    MYWORD SBox[1];              // The S-Box beginning
} TABLE;

/*== Global variables ======================================================*/
UINT      m_uBlockLen;
UINT      m_nRounds;
MYWORD*   m_pRoundKeys;
TABLE*    m_pTables;

/*== Encryption routines ===================================================*/

void XorBlocks(MYWORD* pA, const MYWORD* pB, const UINT uLen)
{
    const MYWORD* const pLast = pA + uLen;
    while(pA < pLast)
        *(pA++) ^= *(pB++);
}

void AddBlocks(MYWORD* pA, const MYWORD* pB, const UINT uLen)
{
    const MYWORD* const pLast = pA + uLen;
    while(pA < pLast)
        *(pA++) += *(pB++);
}

void SubtractBlock(MYWORD* pA, const MYWORD* pB, const UINT uLen)
{
    const MYWORD* const pLast = pA + uLen;
    while(pA < pLast)
        *(pA++) -= *(pB++);
}

void RotRight(MYWORD* pA, const UINT uLen)
{
    const MYWORD* const pFirst = pA;
    MYWORD Tmp;

    pA += uLen - 1;
    Tmp = *pA >> COMPL_SHIFT;
```

```
    while(pA > pFirst)
    {
        *pA = (MYWORD_MASK & (*pA << SHIFT)) |
              (*(pA - 1) >> COMPL_SHIFT);
        pA--;
    }
    *pA = (MYWORD_MASK & (*pA << SHIFT)) | Tmp;
}

void RotLeft(MYWORD* pA, const UINT uLen)
{
    const MYWORD* const pLast = pA + uLen - 1;
    const MYWORD Tmp = (MYWORD_MASK & (*pA << COMPL_SHIFT));

    while(pA < pLast)
    {
        *pA = (*pA >> SHIFT) |
              (MYWORD_MASK & (*(pA + 1) << COMPL_SHIFT));
        pA++;
    }
    *pA = (*pA >> SHIFT) | Tmp;
}

MYWORD GetHash(const MYWORD* pVect, const UINT uLen)
{
    #define _PRIME 3010192529
    MYWORD Sum = 0;
    MYWORD Res = 0;
    const MYWORD* const pLast = pVect + uLen;

    while(pVect < pLast)
        Sum = _PRIME * (Sum + *pVect++);

    while(Sum)
    {
        Res  ^= Sum;
        Sum >>= SEGM_BITS;
    }
    return Res & HASH_MASK;
}



void EncryptBlock(BYTE* Block)
{
    MYWORD* const pBlock = (MYWORD*)Block;
    const UINT nWordsInBlock = m_uBlockLen / MYWORD_LEN;
    UINT jj, ii = m_nRounds;
    MYWORD* pKey = m_pRoundKeys;
```

```
    while(ii--)
    {
        jj = *pBlock & HASH_MASK;
        *pBlock &= INV_HASH_MASK;
        XorBlocks(pBlock, pKey, nWordsInBlock);
        pKey += nWordsInBlock;
        jj ^= GetHash(pBlock, nWordsInBlock);
        AddBlocks(pBlock, m_pTables->SBox + (jj * nWordsInBlock), nWordsInBlock);
        *pBlock |= m_pTables->Perm[jj];
        RotRight(pBlock, nWordsInBlock);
    }
}

void DecryptBlock(BYTE* Block)
{
    MYWORD* const pBlock = (MYWORD*)Block;
    const UINT nWordsInBlock = m_uBlockLen / MYWORD_LEN;
    UINT jj, ii = m_nRounds;
    MYWORD* pKey = m_pRoundKeys + (m_nRounds - 1) * nWordsInBlock;

    while(ii--)
    {
        RotLeft(pBlock, nWordsInBlock);
        jj = m_pTables->InvPerm[*pBlock & HASH_MASK];
        *pBlock &= INV_HASH_MASK;
        SubtractBlock(pBlock, m_pTables->SBox + (jj * nWordsInBlock), nWordsInBlock);
        pKey -= nWordsInBlock;
        jj ^= GetHash(pBlock, nWordsInBlock);
        XorBlocks(pBlock, pKey, nWordsInBlock);
        *pBlock |= jj;
    }
}

/*== Pseudorandom number generator ===========================================*/

typedef double ULONGLONG;
#define MY_MAX_LONG_RAND   0xFFFFFFFF
#define MY_LONG_RAND_BITS  32

/* Generator lags and Luetscher's constants */
enum Constants
{
    L =  37,
    K = 100,
    M =  MY_LONG_RAND_BITS,
    MAX_SUCCESSIVE_CALLS = 100,
    SEPARATING_CALLS_NUM = 1009 - MAX_SUCCESSIVE_CALLS
};
```

```
/* Constants used for spreading a seed of an arbitrary length */
/* to (K * sizeof(UINT) * 8) bits during the Srand procedure  */
typedef struct
{
    UINT uA;
    UINT uC;
    UINT uS;
}
CONGRUENT;

#define CONGRUENT_NUM 3
static const CONGRUENT _CongCfg[CONGRUENT_NUM] =
{ {2521463613U,  457419133U, 11U},
  {1812433253U, 2610540697U, 13U},
  {4092042125U, 3010192481U, 17U} };

/* Global variables used by the PRNG */
static UINT m_uNextA;
static UINT m_uNextB;
static UINT m_uList[K];
static UINT m_uNumOfCalls;

static void Clear(void)
{
    memset(m_uList, 0, sizeof(m_uList));
}

static void GenXor(UINT uSeed, UINT uA, UINT uC, UINT uStep)
{
    UINT ii, jj;

    for(ii = 0, jj = 0; ii < K; ii++, jj += uStep)
    {
        uSeed = uA * uSeed + uC;
        // Rotate uSeed to the left by (ii % 32) bits
        m_uList[jj % K] ^= (uSeed << (ii & 31)) | (uSeed >> (32 - (ii & 31)));
    }
}

static void GenShuffle(UINT uSeed, UINT uA, UINT uC)
{
    UINT ii, n, tmp;

    for(ii = K - 1; (int)ii > 0; ii--)
    {
        uSeed = uA * uSeed + uC;
        n = (UINT)( ((ULONGLONG)(ii + 1) * uSeed) /
                    ((ULONGLONG)MY_MAX_LONG_RAND + 1) );
        // swap the ii-th and n-th element
        tmp = m_uList[ii];
```

```
        m_uList[ii] = m_uList[n];
        m_uList[n] = tmp;
    }
}

static void GenWarmUp(void)
{
    UINT ii;

    m_uNextA = 0;
    m_uNextB = K - L;
    m_uNumOfCalls = 0;

    // Warm up the generator by executing four full cycles
    for(ii = 0; ii < (K << 2); ii++)
    {
        m_uList[m_uNextA] = m_uList[m_uNextA] - m_uList[m_uNextB];
        if( ++m_uNextA == K ) m_uNextA = 0;
        if( ++m_uNextB == K ) m_uNextB = 0;
    }
}

void Srand(const void* pSeed, UINT uSeedLen)
{
    UINT* pSeedWords = (UINT*)pSeed;
    UINT  uSeedWords = uSeedLen / sizeof(UINT);
    UINT  uRestLen = uSeedLen % sizeof(UINT);
    UINT  ii = 0;
    Clear();

    // Use the seed words for XOR-ing and shuffling alternately
    while(uSeedWords)
    {
        GenXor(
            *pSeedWords,
            _CongCfg[ii % CONGRUENT_NUM].uA,
            _CongCfg[ii % CONGRUENT_NUM].uC,
            _CongCfg[ii % CONGRUENT_NUM].uS
        );
        uSeedWords--;
        pSeedWords++;
        ii++;

        if(!uSeedWords) break;

        GenShuffle(
            *pSeedWords,
            _CongCfg[ii % CONGRUENT_NUM].uA,
            _CongCfg[ii % CONGRUENT_NUM].uC
        );
```

```
        uSeedWords--;
        pSeedWords++;
        ii++;
    }

    // Use also the last few bytes of seed, if present
    if(uRestLen > 0)
    {
        UINT uSeedRest = 0;
        memcpy(&uSeedRest, pSeedWords, uRestLen);

        if(ii & 1)
            GenShuffle(
                uSeedRest,
                _CongCfg[ii % CONGRUENT_NUM].uA,
                _CongCfg[ii % CONGRUENT_NUM].uC
            );
        else
            GenXor(
                uSeedRest,
                _CongCfg[ii % CONGRUENT_NUM].uA,
                _CongCfg[ii % CONGRUENT_NUM].uC,
                _CongCfg[ii % CONGRUENT_NUM].uS
            );
    }
    GenWarmUp();
}

UINT Rand(void)
{
    UINT ii;
    UINT uResult;

    // Luetscher's approach
    if(++m_uNumOfCalls > MAX_SUCCESSIVE_CALLS)
    {
        for(ii=0; ii < SEPARATING_CALLS_NUM; ii++)
        {
            m_uList[m_uNextA] = m_uList[m_uNextA] - m_uList[m_uNextB];
            if( ++m_uNextA == K ) m_uNextA = 0;
            if( ++m_uNextB == K ) m_uNextB = 0;
        }
        m_uNumOfCalls = 1;
    }

    // Generate a 32-bit pseudorandom number
    uResult = m_uList[m_uNextA] - m_uList[m_uNextB];
    m_uList[m_uNextA] = uResult;
    if( ++m_uNextA == K ) m_uNextA = 0;
    if( ++m_uNextB == K ) m_uNextB = 0;
```

```
    return uResult;
}

UINT RandMax(UINT uMax)
{
    // Generate a 32-bit pseudorandom number from [0,uMax-1]
    return (UINT)((ULONGLONG)uMax * Rand() /
           ((ULONGLONG)MY_MAX_LONG_RAND + 1));
}

void RandBuffer(BYTE* pBuffer, UINT uLen)
{
    // Initialize a buffer with pseudorandom values
    const BYTE* pEnd1 = pBuffer + (uLen & 0xFFFFFFFC);
    BYTE* pEnd2;
    UINT uLastBlock;

    while(pBuffer < pEnd1)
    {
        *(UINT*)pBuffer = Rand();
        pBuffer += 4;
    }

    pEnd2 = pBuffer + (uLen & 0x3);
    uLastBlock = Rand();

    while(pBuffer < pEnd2)
    {
        *pBuffer = uLastBlock & 0xFF;
        pBuffer++;
        uLastBlock >>= 8;
    }
}

/*== Cipher initialization routines ==========================================*/
void Init(UINT uBlockLen)
{
    m_uBlockLen  = uBlockLen;
    m_nRounds    = ((m_uBlockLen << 3) + SEGM_BITS - 1) / SEGM_BITS + 1;
    m_pRoundKeys = NULL;
    m_pTables    = NULL;
}

void Finish(void)
{
    free(m_pRoundKeys);
    free(m_pTables);
}
```

```
static BOOL GenerateTables(void)
{
    /* Allocate buffer for P and S */
    const UINT nWordsInArray = TABLE_ROWS * m_uBlockLen / MYWORD_LEN;
    int jj;
    MYWORD* pValue;
    MYWORD* pEnd;
    MYWORD* pTableBuffer = (MYWORD*)malloc(
       sizeof(MYWORD) *
       (sizeof(TABLE) / MYWORD_LEN + nWordsInArray - 1)
    );

    if(pTableBuffer == NULL) return FALSE;
    m_pTables = (TABLE*)pTableBuffer;

    /* Generate random P and P^(-1) */
    for(jj = 0; jj < TABLE_ROWS; jj++)
        m_pTables->Perm[jj] = jj;

    for(jj = TABLE_ROWS - 1; jj > 0; jj--)
    {
        UINT n = RandMax(jj + 1);

        UINT Tmp = m_pTables->Perm[jj];
        m_pTables->Perm[jj] = m_pTables->Perm[n];
        m_pTables->Perm[n] = Tmp;

        m_pTables->InvPerm[m_pTables->Perm[jj]] = jj;
    }
    m_pTables->InvPerm[m_pTables->Perm[0]] = 0;

    /* Generate random S */
    RandBuffer((BYTE*)m_pTables->SBox, nWordsInArray * MYWORD_LEN);

    /* ... clear the lowest m bits */
    pValue = m_pTables->SBox;
    pEnd = m_pTables->SBox + nWordsInArray;

    while(pValue < pEnd)
    {
        *pValue &= INV_HASH_MASK;
        pValue += m_uBlockLen / MYWORD_LEN;
    }
    return TRUE;
}

static BOOL GenerateRoundKeys(void)
{
    /* Allocate buffer for k_i */
    MYWORD* pKey;
```

```
    MYWORD* pEnd;
    const UINT nWordsInArray = m_nRounds * m_uBlockLen / MYWORD_LEN;
    m_pRoundKeys = (MYWORD*)malloc(sizeof(MYWORD) * nWordsInArray);

    if(m_pRoundKeys == NULL) return FALSE;

    /* Generate random k_i */
    RandBuffer((BYTE*)m_pRoundKeys, nWordsInArray * MYWORD_LEN);

    /* ... clear the lowest m bits */
    pKey = m_pRoundKeys;
    pEnd = m_pRoundKeys + nWordsInArray;

    while(pKey < pEnd)
    {
        *pKey &= INV_HASH_MASK;
        pKey += m_uBlockLen / MYWORD_LEN;
    }
    return TRUE;
}

BOOL SetKey(BYTE* pKey, UINT uKeyLen)
{
    Srand(pKey, uKeyLen);
    return( GenerateTables() && GenerateRoundKeys() );
}
```

REFERENCES

[1] MORIAI, S.—VAUDENAY, S.: *Comparison of Randomness Provided by Several Schemes for Block Ciphers*, *Third AES Candidate Conference*, 2000, http://csrc.nist.gov/encryption/aes/round2/conf3/papers/34-smoriai.pdf.

[2] SCHNEIER, B., KELSEY, J.: *Unbalanced Feistel Networks and Block Cipher Design*, Fast Software Encryption, 3rd International Workshop Proceedings, Lecture Notes in Comput. Sci, Vol. 1039, Springer-Verlag, Berlin, 1996, pp. 121–144.

[3] ANDERSON, R. BIHAM, E.: *Two Practical and Provably Secure Block Ciphers: BEAR and LION*, Fast Software Encryption, 3rd International Workshop Proceedings, Lecture Notes in Comput. Sci, Vol. 1039, Springer-Verlag, Berlin, 1996, pp. 114–120.

[4] LAI, X.—MASSEY, J.: *A proposal for a New Block Encryption Standard*, Advances in Cryptology, EUROCRYPT '90 Proceedings, Springer Verlag, Berlin, 1991, pp. 389–404, (htpp://www.isi.ee.ethz.ch/publications/isipap/xlai-mass-inspec-1991-2.pdf).

[5] RIVEST, R.—ROBSHAW, M.—SIDNEY, R.—YIN, Y.: *The RC6 Block Cipher*, *NIST AES Proposal*, 1998, http://csrc.nist.gov/encryption/aes.

[6] DAEMEN, J.—RIJMEN, V.: *AES Proposal: Rijndael*, *NIST AES Proposal*, 1998, http://csrc.nist.gov/encryption/aes.

[7] SCHNEIER, B.—KELSEY, J.—WHITING, D.—WAGNER, D.—HALL, C.—FERGUSON, N.: *AES Proposal: Twofish: A 128-bit Block Cipher*, *NIST AES Proposal*, 1998, `http://csrc.nist.gov/encryption/aes`.

[8] CANDA, V.—TRAN VAN TRUNG, MAGLIVERAS, S.—HORVATH, T.: *Symmetric Block Ciphers Based on Group Bases*, *Selected Areas in Cryptography, SAC '00 Proceedings*, Lecture Notes in Comput. Sci., Vol 2012, Springer-Verlag, Berlin, 2001, pp. 89–105, (`http://www.exp-math.uni-essen.de/~valer/GroupBases.pdf`).

[9] CANDA, V.: *Scalable Symmetric Ciphers Based on Group Bases, Ph.D. Thesis*, *Institute for Experimental Mathematics, University of Essen*, Germany, 2001, (`http://www.exp-math.uni-essen.de/~valer/Thesis`).

[10] MISTER, S.—ADAMS, C.: *Practical S-Box Design*, *Selected Areas in Cryptography, SAC '96 Proceedings*, Queens University, Canada, 1996, pp. 61–76, (`http://saturn.ee.queensu.ca:8000/cast/psbd.ps`).

[11] KELIHER, L.—MEIJER, H.: *A New Substitution-Permutation Network Cipher Using Key-Dependent S-Boxes*, *Selected Areas in Cryptography, SAC '97 Proceedings* Queens University, Canada, 1997, (`http://saturn.ee.queensu.ca:8000/sac/sac97/papers/paper25.ps`).

[12] KNUTH, D. E.: *The art of computer programming*, 3-rd Edition, Addison Wesley, 1998, pp. 27–29, 35, 186–188.

[13] SOTO, J.—BASSHAM, L.: *Randomness Testing of the Advanced Encryption Standard Finalist Candidates*, *Third AES Candidate Conference*, 2000, `http://csrc.nist.gov/encryption/aes/round2/conf3/papers/30-jsoto.pdf`.

[14] MARSAGLIA, G.: *Diehard—battery of tests*, 1997, `http://stat.fsu.edu/~geo/diehard.html`, `http://www.helsbreth.org/random/diehard.html`.

[15] NECHVATAL, J. et al.: *Status Report on the First Round of the Development of the Advanced Encryption Standard, 1999*, `http://csrc.nist.gov/encryption/aes/round1/r1report.pdf`.

Received March 24, 2002

Institute for Experimental Mathematics
University of Essen
Ellernstrasse 29
45326 Essen
GERMANY

E-mail: valer,trung@exp-math.uni-essen.de