

Cover Sheet

Name of submitted algorithm:	KHAZAD
Type of submitted algorithm, proposed security level, proposed environment	Block cipher, Normal-Legacy, Any (no restrictions)
Principal submitter:	Paulo Sérgio L.M. Barreto. Tel: +55 11 39 09 34 79 Scopus Tecnologia S. A. Av. Mutinga, 4105 – Pirituba 05110-000 São Paulo (SP) Brazil <pbarreto@scopus.com.br>
Auxiliary submitter:	Vincent Rijmen
Algorithm inventors:	Paulo Sérgio L.M. Barreto and Vincent Rijmen
Algorithm owners:	Paulo Sérgio L.M. Barreto and Vincent Rijmen
Backup point of contact:	Vincent Rijmen Tel: +32 16 32 18 62 Fax: +32 16 32 19 86 Cryptomathic NV Lei 8A B-3000 Leuven Belgium <vincent.rijmen@cryptomathic.com>
Signature of submitters:	
Paulo Sérgio L.M. Barreto	Vincent Rijmen

The KHAZAD Legacy-Level Block Cipher

Paulo S.L.M. Barreto^{1*} and Vincent Rijmen²

¹ Scopus Tecnologia S. A.
Av. Mutinga, 4105 - Pirituba
BR-05110-000 São Paulo (SP), Brazil
`pbarreto@scopus.com.br`

² Cryptomathic NV,
Lei 8A,
B-3000 Leuven, Belgium
`vincent.rijmen@cryptomathic.com`

Abstract. KHAZAD is a 64-bit (legacy-level) block cipher that accepts a 128-bit key. The cipher is a uniform substitution-permutation network whose inverse only differs from the forward operation in the key schedule. The overall cipher design follows the Wide Trail strategy, favours component reuse, and permits a wide variety of implementation tradeoffs.

1 Introduction

In this document we describe KHAZAD, a 64-bit (legacy-level) block cipher that accepts a 128-bit key. KHAZAD has been submitted as a candidate cryptographic primitive for the NESSIE project [23].

Although KHAZAD is not a Feistel cipher, its structure is designed so that by choosing all round transformation components to be involutions, the inverse operation of the cipher differs from the forward operation in the key scheduling only. This property makes it possible to reduce the required chip area in a hardware implementation, as well as the code and table size, which can be important when KHAZAD is used e.g. in a Java applet.

KHAZAD was designed according to the Wide Trail strategy [8]. In the Wide Trail strategy, the round transformation of a block cipher is composed of different invertible transformations, each with its own functionality and requirements. The *linear diffusion layer* ensures that after a few rounds all the output bits depend on all the input bits. The *nonlinear layer* ensures that this dependency is of a complex and nonlinear nature. The *round key addition* introduces the key material. One of the advantages of the Wide Trail strategy is that the different components can be specified quite independently from one another. We largely follow the Wide Trail strategy in the design of the key scheduling algorithm as well.

* Co-sponsored by the Computer Architecture and Networking Laboratory (LARC), Department of Computer and Digital Systems Engineering, Escola Politécnica da Universidade de São Paulo (Brazil).

As originally submitted for the NESSIE evaluation effort, KHAZAD employed a randomly generated substitution box (S-box) whose lack of internal structure tended to make efficient hardware implementation a challenging and tricky process. In contrast to that version, though, the present document describes an S-box that is much more amenable to hardware implementation, while not adversely affecting any of the software implementation techniques suggested herein. We propose renaming the original algorithm KHAZAD-0 and using the term KHAZAD for the final, modified version that uses the improved S-box design.

This document is organised as follows. The mathematical preliminaries and notation employed are described in section 2. A mathematical description of the KHAZAD primitive is given in section 3. A statement of the claimed security properties and expected security level is made in section 4. An analysis of the primitive with respect to standard cryptanalytic attacks is provided in section 5 (a statement that there are no hidden weaknesses inserted by the designers is explicitly made in section 5.9). Section 6 contains the design rationale explaining design choices. Implementation guidelines to avoid implementation weaknesses are given in section 7. Estimates of the computational efficiency in software are provided in section 8. The overall strengths and advantages of the primitive are listed in section 9.

2 Mathematical preliminaries and notation

2.1 Finite fields

We will represent the field $\text{GF}(2^4)$ as $\text{GF}(2)[x]/p_4(x)$ where $p_4(x) = x^4 + x + 1$, and the field $\text{GF}(2^8)$ as $\text{GF}(2)[x]/p_8(x)$ where $p_8(x) = x^8 + x^4 + x^3 + x^2 + 1$. Polynomials $p_4(x)$ and $p_8(x)$ are the first primitive polynomials of degrees 4 and 8 listed in [20], and were chosen so that $g(x) = x$ is a generator of $\text{GF}(2^4) \setminus \{0\}$ and $\text{GF}(2^8) \setminus \{0\}$, respectively.

A polynomial $u = \sum_{i=0}^{m-1} u_i \cdot x^i \in \text{GF}(2)[x]$, where $u_i \in \text{GF}(2)$ for all $i = 0, \dots, m-1$, will be denoted by the numerical value $\sum_{i=0}^{m-1} u_i \cdot 2^i$, and written in hexadecimal notation. For instance, we write 13_x to denote $p_4(x)$.

2.2 MDS codes

The Hamming distance between two vectors u and v from the n -dimensional vector space $\text{GF}(2^p)^n$ is the number of coordinates where u and v differ.

The Hamming weight $w_h(a)$ of an element $a \in \text{GF}(2^p)^n$ is the Hamming distance between a and the null vector of $\text{GF}(2^p)^n$, i.e. the number of nonzero components of a .

A *linear* $[n, k, d]$ code over $\text{GF}(2^p)$ is a k -dimensional subspace of the vector space $(\text{GF}(2^p))^n$, where the Hamming distance between any two distinct subspace vectors is at least d (and d is the largest number with this property).

A *generator matrix* G for a linear $[n, k, d]$ code \mathcal{C} is a $k \times n$ matrix whose rows form a basis for \mathcal{C} . A generator matrix is in *echelon* or *standard* form if it

has the form $G = [I_{k \times k} \ A_{k \times (n-k)}]$, where $I_{k \times k}$ is the identity matrix of order k . We write simply $G = [I \ A]$ omitting the indices wherever the matrix dimensions are irrelevant for the discussion, or clear from the context.

Linear $[n, k, d]$ codes obey the *Singleton bound*: $d \leq n - k + 1$. A code that meets the bound, i.e. $d = n - k + 1$, is called a *maximal distance separable* (MDS) code. A linear $[n, k, d]$ code \mathcal{C} with generator matrix $G = [I_{k \times k} \ A_{k \times (n-k)}]$ is MDS if, and only if, every square submatrix formed from rows and columns of A is nonsingular (cf. [22], chapter 11, § 4, theorem 8).

2.3 Cryptographic properties

A product of m distinct Boolean variables is called an m -th order product of the variables. Every Boolean function $f : \text{GF}(2)^n \rightarrow \text{GF}(2)$ can be written as a sum over $\text{GF}(2)$ of distinct m -order products of its arguments, $0 \leq m \leq n$; this is called the algebraic normal form of f .

The *nonlinear order* of f , denoted $\nu(f)$, is the maximum order of the terms appearing in its algebraic normal form. A *linear* Boolean function is a Boolean function of nonlinear order 1, i.e. its algebraic normal form only involves isolated arguments. Given $\alpha \in \text{GF}(2)^n$, we denote by $l_\alpha : \text{GF}(2)^n \rightarrow \text{GF}(2)$ the linear Boolean function consisting of the sum of the argument bits selected by the bits of α :

$$l_\alpha(x) \equiv \bigoplus_{i=0}^{n-1} \alpha_i \cdot x_i.$$

A mapping $S : \text{GF}(2^n) \rightarrow \text{GF}(2^n), x \mapsto S[x]$, is called a *substitution box*, or S-box for short. An S-box can also be viewed as a mapping $S : \text{GF}(2)^n \rightarrow \text{GF}(2)^n$ and therefore described in terms of its component Boolean functions $s_i : \text{GF}(2)^n \rightarrow \text{GF}(2), 0 \leq i \leq n-1$, i.e. $S[x] = (s_0(x), \dots, s_{n-1}(x))$.

The *nonlinear order* of an S-box S , denoted ν_S , is the minimum nonlinear order over all linear combinations of the components of S :

$$\nu_S \equiv \min_{\alpha \in \text{GF}(2)^n} \{\nu(l_\alpha \circ S)\}.$$

The δ -*parameter* of an S-box S is defined as

$$\delta_S \equiv 2^{-n} \cdot \max_{a \neq 0, b} \#\{c \in \text{GF}(2^n) \mid S[c \oplus a] \oplus S[c] = b\}.$$

The value $2^n \cdot \delta$ is called the *differential uniformity* of S .

The *correlation* $c(f, g)$ between two Boolean functions f and g is defined as:

$$c(f, g) \equiv 2^{1-n} \cdot \#\{x \mid f(x) = g(x)\} - 1.$$

The extreme value (i.e. either the minimum or the maximum, whichever is larger in absolute value) of the correlation between linear functions of input bits and linear functions of output bits of S is called the *bias* of S .

The λ -parameter of an S-box S is defined as the absolute value of the bias:

$$\lambda_S \equiv \max_{(i,j) \neq (0,0)} |c(l_i, l_j \circ S)|.$$

The *branch number* \mathcal{B} of a linear mapping $\theta : \text{GF}(2^p)^k \rightarrow \text{GF}(2^p)^m$ is defined as

$$\mathcal{B}(\theta) \equiv \min_{a \neq 0} \{w_h(a) + w_h(\theta(a))\}.$$

Given a $[k + m, k, d]$ linear code over $\text{GF}(2^p)$ with generator matrix $G = [I_{k \times k} \ M_{k \times m}]$, the linear mapping $\theta : \text{GF}(2^p)^k \rightarrow \text{GF}(2^p)^m$ defined by $\theta(a) = a \cdot M$ has branch number $\mathcal{B}(\theta) = d$; if the code is MDS, such a mapping is called an *optimal diffusion mapping* [25].

2.4 Miscellaneous notation

If m is a power of 2, $\text{had}(a_0, \dots, a_{m-1})$ denotes the $m \times m$ matrix with elements $h_{ij} = a_{i \oplus j}$, sometimes called a Hadamard-like matrix [2].

Given a sequence of functions $f_m, f_{m+1}, \dots, f_{n-1}, f_n$, $m \leq n$, we use the notation $\bigcirc_{r=m}^n f_r \equiv f_m \circ f_{m+1} \circ \dots \circ f_{n-1} \circ f_n$, and $\bigcirc_m^{r=n} f_r \equiv f_n \circ f_{n-1} \circ \dots \circ f_{m+1} \circ f_m$; if $m > n$, both expressions stand for the identity mapping.

3 Description of the KHAZAD primitive

The KHAZAD cipher is an iterated ‘involutional’¹ block cipher that operates on a 64-bit *cipher state* represented as a vector in $\text{GF}(2^8)^8$. It uses a 128-bit *cipher key* K represented as a vector in $\text{GF}(2^8)^{16}$, and consists of a series of applications of a key-dependent round transformation to the cipher state. In the following we will individually define the component mappings and constants that build up KHAZAD, then specify the complete cipher in terms of these components.

3.1 The nonlinear layer γ

Function $\gamma : \text{GF}(2^8)^8 \rightarrow \text{GF}(2^8)^8$ consists of the parallel application of a nonlinear substitution box $S : \text{GF}(2^8) \rightarrow \text{GF}(2^8)$, $x \mapsto S[x]$ to all bytes of the argument individually:

$$\gamma(a) = b \Leftrightarrow b_i = S[a_i], \quad 0 \leq i \leq 7.$$

The substitution box S is discussed in detail in section 6.2. One of the design criteria for S imposes that it be an involution, i.e. $S[S[x]] = x$ for all $x \in \text{GF}(2^8)$. Therefore, γ itself is an involution.

¹ We explain in section 3.8 what we mean by an ‘involutional’ block cipher.

3.2 The linear diffusion layer θ

The diffusion layer $\theta : \text{GF}(2^8)^8 \rightarrow \text{GF}(2^8)^8$ is a linear mapping based on the [16, 8, 9] MDS code with generator matrix $G_H = [IH]$, where $H = \text{had}(01_x, 03_x, 04_x, 05_x, 06_x, 08_x, 0b_x, 07_x)$, i.e.

$$H = \begin{bmatrix} 01_x & 03_x & 04_x & 05_x & 06_x & 08_x & 0B_x & 07_x \\ 03_x & 01_x & 05_x & 04_x & 08_x & 06_x & 07_x & 0B_x \\ 04_x & 05_x & 01_x & 03_x & 0B_x & 07_x & 06_x & 08_x \\ 05_x & 04_x & 03_x & 01_x & 07_x & 0B_x & 08_x & 06_x \\ 06_x & 08_x & 0B_x & 07_x & 01_x & 03_x & 04_x & 05_x \\ 08_x & 06_x & 07_x & 0B_x & 03_x & 01_x & 05_x & 04_x \\ 0B_x & 07_x & 06_x & 08_x & 04_x & 05_x & 01_x & 03_x \\ 07_x & 0B_x & 08_x & 06_x & 05_x & 04_x & 03_x & 01_x \end{bmatrix},$$

so that $\theta(a) = b \Leftrightarrow b = a \cdot H$. A simple inspection shows that matrix H is symmetric and unitary. Therefore, θ is an involution.

3.3 The key addition $\sigma[k]$

The affine key addition $\sigma[k] : \text{GF}(2^8)^8 \rightarrow \text{GF}(2^8)^8$ consists of the bitwise addition (exor) of a key vector $k \in \text{GF}(2^8)^8$:

$$\sigma[k](a) = b \Leftrightarrow b_i = a_i \oplus k_i, \quad 0 \leq i \leq 7.$$

This mapping is also used to introduce round constants in the key schedule, and is obviously an involution.

3.4 The round constants c^r

The round constant for the r -th round is a vector $c^r \in \text{GF}(2^8)^8$, defined as:

$$c_i^r = S[8r + i], \quad 0 \leq r \leq R, \quad 0 \leq i \leq 7.$$

3.5 The round function $\rho[k]$

The r -th round function is the composite mapping $\rho[k] : \text{GF}(2^8)^8 \rightarrow \text{GF}(2^8)^8$, parameterised by the key vector $k \in \text{GF}(2^8)^8$ and given by:

$$\rho[k] \equiv \sigma[k] \circ \theta \circ \gamma.$$

3.6 The key schedule

The key schedule expands the cipher key $K \in \text{GF}(2^8)^{16}$ into a sequence of round keys K^0, \dots, K^R , plus two initial values, K^{-2} and K^{-1} , with $K^r \in \text{GF}(2^8)^8$. The initial values K^{-2} and K^{-1} are taken respectively from bytes 0 through 7 and 8 through 15 of the cipher key K :

$$K_i^{-2} = K_i, \quad K_i^{-1} = K_{8+i}, \quad 0 \leq i \leq 7.$$

The sequence of round keys are computed by means of a Feistel iteration based on the round function ρ and the round constants c^r :

$$K^r = \rho[c^r](K^{r-1}) \oplus K^{r-2}, \quad 0 \leq r \leq R.$$

3.7 The complete cipher

KHAZAD is defined for the cipher key K as the transformation $\text{KHAZAD}[K] = \alpha_R[K^0, \dots, K^R]$ applied to the plaintext, where

$$\alpha_R[K^0, \dots, K^R] = \sigma[K^R] \circ \gamma \circ \left(\bigcirc_{r=1}^{R-1} \rho[K^r] \right) \circ \sigma[K^0].$$

The standard number of rounds is $R = 8$.

3.8 The inverse cipher

We now show that KHAZAD is an involutory cipher, in the sense that the only difference between the cipher and its inverse is in the key schedule. We will need the following lemma:

Lemma 1. $\theta \circ \sigma[K^r] = \sigma[\theta(K^r)] \circ \theta$.

Proof. It suffices to notice that $(\theta \circ \sigma[K^r])(a) = \theta(K^r \oplus a) = \theta(K^r) \oplus \theta(a) = (\sigma[\theta(K^r)] \circ \theta)(a)$, for any $a \in \text{GF}(2^8)^8$. \square

Let $\bar{K}^0 \equiv K^R$, $\bar{K}^R \equiv K^0$, and $\bar{K}^r \equiv \theta(K^{R-r})$, $0 < r < R$. We are now ready to state the main property of the inverse KHAZAD cipher $\alpha_R^{-1}[K^0, \dots, K^R]$:

Theorem 1. $\alpha_R^{-1}[K^0, \dots, K^R] = \alpha_R[\bar{K}^0, \dots, \bar{K}^R]$.

Proof. We start from the definition of R -round KHAZAD:

$$\alpha_R[K^0, \dots, K^R] = \sigma[K^R] \circ \gamma \circ \left(\bigcirc_{r=1}^{R-1} \sigma[K^r] \circ \theta \circ \gamma \right) \circ \sigma[K^0].$$

Since the component functions are involutions, the inverse cipher is obtained by applying them in reverse order:

$$\alpha_R^{-1}[K^0, \dots, K^R] = \sigma[K^0] \circ \left(\bigcirc_{r=1}^{R-1} \gamma \circ \theta \circ \sigma[K^r] \right) \circ \gamma \circ \sigma[K^R].$$

The above lemma leads to:

$$\alpha_R^{-1}[K^0, \dots, K^R] = \sigma[K^0] \circ \left(\bigcirc_{r=1}^{R-1} \gamma \circ \sigma[\theta(K^r)] \circ \theta \right) \circ \gamma \circ \sigma[K^R].$$

Using the associativity of functional composition we can slightly change the grouping of operations:

$$\alpha_R^{-1}[K^0, \dots, K^R] = \sigma[K^0] \circ \gamma \circ \left(\bigcirc_{r=1}^{R-1} \sigma[\theta(K^r)] \circ \theta \circ \gamma \right) \circ \sigma[K^R].$$

Finally, by substituting \bar{K}^r in the above equation, we arrive at:

$$\alpha_R[K^0, \dots, K^R] = \sigma[\bar{K}^R] \circ \gamma \circ \left(\bigcirc_{r=0}^{R-1} \sigma[\bar{K}^r] \circ \theta \circ \gamma \right) \circ \sigma[\bar{K}^0].$$

That is, $\alpha_R^{-1}[K^0, \dots, K^R] = \alpha_R[\bar{K}^0, \dots, \bar{K}^R]$, where $\bar{K}^0 \equiv K^R$, $\bar{K}^R \equiv K^0$, and $\bar{K}^r \equiv \theta(K^{R-r})$, $0 < r < R$. \square

Corollary 1. *The KHAZAD cipher has involutonal structure, in the sense that the only difference between the cipher and its inverse is in the key schedule.*

4 Security goals

In this section, we present the goals we have set for the security of KHAZAD. A cryptanalytic attack will be considered successful by the designers if it demonstrates that a security goal described herein does not hold.

In order to formulate our goals, we must define two security-related concepts:

Definition 1 ([8]). *A block cipher is K-secure if all possible attack strategies for it have the same expected work factor and storage requirements as for the majority of possible block ciphers with the same dimensions [block length and key length]. This must be the case for all possible modes of access for the adversary and for any a priori key distribution.*

Definition 2 ([8]). *A block cipher is hermetic if it does not have weaknesses that are not present for the majority of block ciphers with the same block and key length.*

4.1 Goals

The security goals are that the KHAZAD cipher be:

- K-secure;
- Hermetic.

If KHAZAD lives up to its goals, the strength against any known or unknown attacks is as good as can be expected from a block cipher with the given dimensions [8].

5 Analysis

5.1 Differential and linear cryptanalysis

Because the branch number of θ is $\mathcal{B} = 9$ (cf. [26], proposition 1), no differential characteristic over two rounds has probability larger than $\delta^{\mathcal{B}} = (2^{-5})^9 = 2^{-45}$, and no linear approximation over two rounds has input-output correlation larger than $\lambda^{\mathcal{B}} = (16 \times 2^{-6})^9 = 2^{-18}$. This makes classical differential or linear attacks, as well as some advanced variants like differential-linear attacks, very unlikely to succeed for the full cipher.

5.2 Truncated differentials

The concept of truncated differentials was introduced in [18], and typically applies to ciphers in which all transformations operate on well aligned data blocks, as is the case for KHAZAD where all transformations operate on bytes rather than individual bits. However, the fact that all submatrices of H are nonsingular makes a truncated differential attack against more than a few rounds of KHAZAD impossible, because the S/N ratio of an attack becomes too low. For 4 rounds or more, no truncated differential attacks can be mounted.

5.3 Interpolation attacks

Interpolation attacks [15] generally depend on the cipher components (particularly the S-box) having simple algebraic structures that can be combined to give polynomial or rational expressions with manageable complexity. The involved expression of the S-box in $\text{GF}(2^8)$, combined with the effect of the diffusion layer, makes this type of attack infeasible for more than a few rounds.

5.4 Weak keys

The weak keys we discuss are keys that result in a block cipher mapping with detectable weaknesses. The best known case of such weak keys are those of IDEA [8]. Typically, this occurs for ciphers where the nonlinear operations depend on the actual key value. This is not the case for KHAZAD, where keys are applied using xor and all nonlinearity is in the fixed S-box. In KHAZAD, there is no restriction on key selection.

5.5 Related-key cryptanalysis

Related-key attacks generally rely upon slow diffusion and/or symmetry in the key schedule. The KHAZAD key schedule inherits many properties from the round structure itself, and was designed to cause fast, nonlinear diffusion of cipher key differences to the round keys.

5.6 SQUARE *aka* saturation attacks

In this section we present an attack first described in [25]. This attack works against KHAZAD reduced to 3 rounds. We will denote by a^r the cipher state at the beginning of the r -round (input to γ), and by b^r the cipher state at the output of the σ key addition in the r -round; these quantities may be indexed to select a particular byte. For instance, b_i^1 is the byte at position i of the cipher state at the output of round 1.

Take a set of 256 plaintexts different from each other in a single byte (which assumes all possible values), the remaining 7 bytes being constant. After one round all 8 bytes of each cipher state a^2 in the set will take every value exactly

once. After two rounds, the xor of all 256 cipher states a^3 at every byte position will be zero.

Consider a ciphertext $b^3 = \gamma(a^3) \oplus K^3$; clearly $a^3 = \gamma(b^3 \oplus K^3)$. Now take a byte from b^3 , guess the matching byte from K^3 and apply γ to the xor of these quantities. Do this for all 256 ciphertexts in the set and check whether the xor of the 256 results indeed equals zero. If it doesn't, the guessed key byte is certainly wrong. A few wrong keys (a fraction about 1/256 of all keys) may pass this test; repeating it for a second set of plaintexts leaves only the correct K^3 value with overwhelming probability.

This attack recovers one byte of the last round key. The remaining bytes can be obtained by repeating the attack eight times. Overall, this attack requires 2^9 chosen plaintexts. However, almost all wrong key values can be eliminated after processing a single set of 2^8 plaintexts. The workload to recover one key byte is thus 2^8 key guesses $\times 2^8$ chosen plaintexts = 2^{16} S-box lookups.

5.7 A general extension attack

Any n -round attack can be extended against $(n+1)$ or more rounds for long keys by simply guessing the whole K^{n+1} round key and proceeding with the n -round attack [21]. Each extra round increases the complexity by a factor 2^{64} S-box lookups. The best attack known against 3 rounds of KHAZAD has complexity about 2^{16} S-box lookups, hence the 4-round extension costs $2^{16+64} = 2^{80}$ S-box lookups.

5.8 Other attacks

An extension of the Biham-Keller impossible differential attack on RIJNDAEL reduced to 5 rounds [5] can be applied to KHAZAD, reduced to 3 rounds. The attack requires 2^{13} chosen plaintexts and an effort of 2^{64} encryptions.

We see no obvious way to extend the Gilbert-Minier attack [13] against RIJNDAEL and other ciphers of the SQUARE family, since the attack makes direct use of the two-level diffusion structure of those ciphers.

Attacks based on linear cryptanalysis can sometimes be improved by using nonlinear approximations [19]. However, with the current state of the art the application of nonlinear approximations seems limited to the first and/or the last round of a linear approximation. This seems to be even more so for ciphers using strongly nonlinear S-boxes, like KHAZAD.

We were not able to find any other attack method, including slide [6], advanced slide [7], boomerang [27], and rectangle [3] attacks, that could break the KHAZAD cipher faster than exhaustive key search.

5.9 Designers' statement on the absence of hidden weaknesses

In spite of any analysis, doubts might remain regarding the presence of trapdoors deliberately introduced in the algorithm. That is why the NESSIE effort asks for the designers' declaration on the contrary.

Therefore we, the designers of KHAZAD, do hereby declare that there are no hidden weaknesses inserted by us in the KHAZAD primitive.

6 Design rationale

6.1 Self-inverse structure

Involutorial structure is found as part of many cipher designs; in particular, all classical Feistel networks [11] have this property. Self-inverse ciphers similar to KHAZAD were described and analyzed in [31, 32]. The importance of involutorial structure resides not only in the advantages for implementation, but also in the equivalent security of both encryption and decryption.

6.2 Choice of the substitution box

The originally submitted form of KHAZAD used a pseudo-randomly generated S-box, chosen to satisfy the following conditions:

- S must be an involution, i.e. $S[S[x]] = x$ for all $x \in \text{GF}(2^8)$.
- The δ -parameter must not exceed 8×2^{-8} .
- The λ -parameter must not exceed 16×2^{-6} .
- The nonlinear order ν must be maximum, namely, 7.

The bounds on δ and λ correspond to twice the minimum achievable values for these quantities. An additional condition, that the S-box has no fixed point, was imposed in an attempt to speed up the search. This condition was inspired by the empirical study reported in [31, section 2.3], where the strong correlation found between the cryptographic properties and the number of fixed points of a substitution box suggests minimising the number of such points. The polynomial and rational representations of S over $\text{GF}(2^8)$ are checked as well, to avoid any obvious algebraic weakness (which could lead e.g. to interpolation attacks [15]). Finally, affine approximations to S are also considered; no such approximation was found that matches S in more than 19 points², too few to mount any attack we could conceive.

However, the extreme lack of structure in such an S-box hinders efficient hardware implementation. Moreover, a flaw that went unnoticed in the random search program caused the value of λ for the original S-box to be incorrectly reported as 13×2^{-6} instead of the actual value 17×2^{-6} (corresponding to a negative bias), which is slightly worse than the design bound³. Although this is still far too low to make classical linear attacks feasible, it can be easily remedied.

Therefore, we now describe an alternative S-box that, besides exactly satisfying the design conditions, is amenable to much more efficient implementation in hardware, while not affecting the software implementation techniques presented here in any reasonable way. The new S-box is illustrated in figure 1.

² This is true for both the original S-box and the new, improved design.

³ We thank the NESSIE evaluation team for pointing out this discrepancy [24].

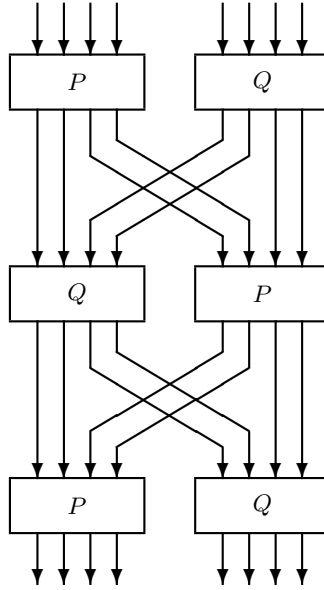


Fig. 1. Structure of the KHAZAD S-box. Both P and Q are pseudo-randomly generated involutions; the output from the upper and middle nonlinear layers are mixed through a simple linear shuffling.

The P and Q tables are pseudo-randomly generated involutions with optimal δ , λ , and ν , chosen so that the S-box built from them satisfies the design criteria listed at the beginning of this section. Tables 1 and 2 show the involutions found by the searching algorithm.

A description of the searching algorithm and a listing of the resulting S-box are given in the appendix.

Table 1. Actual P mini-box

u	0_x	1_x	2_x	3_x	4_x	5_x	6_x	7_x	8_x	9_x	A_x	B_x	C_x	D_x	E_x	F_x
$P[u]$	3_x	F_x	E_x	0_x	5_x	4_x	B_x	C_x	D_x	A_x	9_x	6_x	7_x	8_x	2_x	1_x

Table 2. Actual Q mini-box

u	0_x	1_x	2_x	3_x	4_x	5_x	6_x	7_x	8_x	9_x	A_x	B_x	C_x	D_x	E_x	F_x
$Q[u]$	9_x	E_x	5_x	6_x	A_x	2_x	3_x	C_x	F_x	0_x	4_x	D_x	7_x	B_x	1_x	8_x

6.3 Choice of the diffusion layer

The actual matrix used in the diffusion layer θ was selected by exhaustive search. Although other ciphers of the same family as KHAZAD use circulant matrices for this purpose (cf. [26]), it is not difficult to prove that no such matrix can be self-inverse. On the other hand, unitary Hadamard-like matrices can be easily computed that satisfy the MDS condition.

The actual choice involves coefficients with the lowest possible Hamming weight (which is advantageous for hardware implementations) and lowest possible integer values (which is important for smart card implementations as discussed in section 7.3).

6.4 Structure of the key schedule

Adopting a Feistel key schedule provides a simple and effective way to expand a $2m$ -bit cipher key onto m -bit round keys reusing the round function itself. This keeps the overall cipher structure uniformly m -bit oriented (in the sense that the natural data units occurring in the cipher are bytes and m -bit blocks).

6.5 Choice of the round constants

Good round constants should not be equal for all bytes in a state, and also not equal for all bit positions in a byte. They should also be different in each round. The actual choice meets these constraints while also reusing an available component (the S-box itself).

7 Implementation

KHAZAD can be implemented very efficiently. On different platforms, different optimisations and tradeoffs are possible. We make here a few suggestions.

7.1 64-bit processors

Implementation of ρ : We suggest the following lookup-table approach. Let H_k be the k -th row of the Hadamard-like matrix H ; using eight tables $T_k[x] \equiv S[x] \cdot H_k$, $0 \leq k \leq 7$, i.e.:

$$\begin{aligned} T_0[x] &= S[x] \cdot [01_x \ 03_x \ 04_x \ 05_x \ 06_x \ 08_x \ 0B_x \ 07_x], \\ T_1[x] &= S[x] \cdot [03_x \ 01_x \ 05_x \ 04_x \ 08_x \ 06_x \ 07_x \ 0B_x], \\ T_2[x] &= S[x] \cdot [04_x \ 05_x \ 01_x \ 03_x \ 0B_x \ 07_x \ 06_x \ 08_x], \\ T_3[x] &= S[x] \cdot [05_x \ 04_x \ 03_x \ 01_x \ 07_x \ 0B_x \ 08_x \ 06_x], \\ T_4[x] &= S[x] \cdot [06_x \ 08_x \ 0B_x \ 07_x \ 01_x \ 03_x \ 04_x \ 05_x], \\ T_5[x] &= S[x] \cdot [08_x \ 06_x \ 07_x \ 0B_x \ 03_x \ 01_x \ 05_x \ 04_x], \\ T_6[x] &= S[x] \cdot [0B_x \ 07_x \ 06_x \ 08_x \ 04_x \ 05_x \ 01_x \ 03_x], \\ T_7[x] &= S[x] \cdot [07_x \ 0B_x \ 08_x \ 06_x \ 05_x \ 04_x \ 03_x \ 01_x], \end{aligned}$$

then one can compute $b = (\theta \circ \gamma)(a) = \bigoplus_{k=0}^7 T_k[a_k]$ with eight table lookups and seven xor operations; the key addition then completes the evaluation of ρ . The T -tables require $2^8 \times 8$ bytes of storage each. An implementation can use the fact that the corresponding entries of different T -tables are permutations of one another and save some memory at the expense of introducing extra permutations at runtime. Usually this decreases the performance of the implementation.

Implementation of θ for the inverse key schedule: The simplest approach is to use tables similar to those suggested for the implementation of ρ . However, instead of defining independent tables $T'_i[x] = x \cdot H_i$, $0 \leq i \leq 7$, one can use the relation $T'_i[x] = T_i[S[x]]$ and extract the value of $S[x]$ from the 01_x entries of the T tables with a masking ‘and’ operation. This way the T tables themselves can be used and no extra storage is needed.

7.2 32-bit processors

Any Hadamard-like matrix H (of order m) shows the following structure:

$$H = \begin{bmatrix} U & V \\ V & U \end{bmatrix},$$

where U and V are themselves Hadamard-like matrices (of order $m/2$). A 32-bit implementation may take advantage of this structure by representing elements $c \in \text{GF}(2^8)^8$ as pairs $c = [\hat{c}_0 \ \hat{c}_1]$ of elements $\hat{c}_i \in \text{GF}(2^8)^4$:

$$b = \theta(a) \Leftrightarrow \begin{cases} \hat{b}_0 = \hat{a}_0 U \oplus \hat{a}_1 V, \\ \hat{b}_1 = \hat{a}_0 V \oplus \hat{a}_1 U, \end{cases}$$

with twice the complexity derived for 64-bit processors regarding the number of table lookups and exors, but using smaller tables (each occupying $2^8 \times 4$ bytes).

7.3 8-bit processors

On an 8-bit processor with a limited amount of RAM, e.g. a typical smart card processor, the previous approach is not feasible. On these processors the substitution is performed byte by byte, combined with the $\sigma[k]$ transformation. For θ , it is necessary to implement the matrix multiplication.

The following piece of pseudo-code calculates $b = \theta(a)$, using a table X that implements multiplication by the polynomial $g(x) = x$ in $\text{GF}(2^8)$ (i.e. $X[u] \equiv x \cdot u$) and fourteen temporary variables t_0 to t_7 and r_0 to r_5 :

```

t0 ← a0 ⊕ a1; t1 ← a2 ⊕ a3; t2 ← a4 ⊕ a5; t3 ← a6 ⊕ a7;
t4 ← a1 ⊕ a4; t5 ← a0 ⊕ a5; t6 ← a3 ⊕ a6; t7 ← a2 ⊕ a7;
r0 ← t0 ⊕ X[X[t1]]; r2 ← X[X[a5 ⊕ a6]];
r1 ← t1 ⊕ X[X[t0]]; r3 ← X[X[a4 ⊕ a7]];
r4 ← t3 ⊕ r2; r5 ← t3 ⊕ r3;

```

$$\begin{aligned}
b_0 &\leftarrow a_3 \oplus r_0 \oplus r_5 \oplus X[t_4 \oplus r_4]; & b_1 &\leftarrow a_2 \oplus r_0 \oplus r_4 \oplus X[t_5 \oplus r_5]; \\
r_4 &\leftarrow t_2 \oplus r_2; & r_5 &\leftarrow t_2 \oplus r_3; \\
b_2 &\leftarrow a_1 \oplus r_1 \oplus r_4 \oplus X[t_6 \oplus r_5]; & b_3 &\leftarrow a_0 \oplus r_1 \oplus r_5 \oplus X[t_7 \oplus r_4]; \\
r_0 &\leftarrow t_2 \oplus X[X[t_3]]; & r_2 &\leftarrow X[X[a_1 \oplus a_2]]; \\
r_1 &\leftarrow t_3 \oplus X[X[t_2]]; & r_3 &\leftarrow X[X[a_0 \oplus a_3]]; \\
r_4 &\leftarrow t_1 \oplus r_2; & r_5 &\leftarrow t_1 \oplus r_3; \\
b_4 &\leftarrow a_7 \oplus r_0 \oplus r_5 \oplus X[t_5 \oplus r_4]; & b_5 &\leftarrow a_6 \oplus r_0 \oplus r_4 \oplus X[t_4 \oplus r_5]; \\
r_4 &\leftarrow t_0 \oplus r_2; & r_5 &\leftarrow t_0 \oplus r_3; \\
b_6 &\leftarrow a_5 \oplus r_1 \oplus r_4 \oplus X[t_7 \oplus r_5]; & b_7 &\leftarrow a_4 \oplus r_1 \oplus r_5 \oplus X[t_6 \oplus r_4];
\end{aligned}$$

This implementation requires 56 exors and 24 table lookups. Notice that, if an additional table $X2$ is available, where $X2[u] \equiv X[X[u]]$, the number of table lookups drops to 16. There may be more efficient ways to implement θ , however; we did not search thoroughly all possibilities.

7.4 Techniques to avoid software implementation weaknesses

The attacks of Kocher *et al.* [16,17] have raised the awareness that careless implementation of cryptographic primitives can be exploited to recover key material. In order to counter this type of attacks, attention has to be given to the implementation of the round transformation as well as the key scheduling of the primitive.

A first example is the *timing attack* [16] that can be applicable if the execution time of the primitive depends on the value of the key and the plaintext. This is typically caused by the presence of conditional execution paths. For instance, multiplication by a constant value over a finite field is sometimes implemented as a shift followed by a conditional exor. This vulnerability is avoided by a table lookup implementation as proposed in sections 7.2 and 7.3.

A second class of attacks are the attacks based on the careful observation of the power consumption pattern of an encryption device [17]. Protection against this type of attack can only be achieved by combined measures at the hardware and software level. We leave the final word on this issue to the specialists, but we hope that the simple structure and the limited number of operations in KHAZAD will make it easier to create an implementation that resists this type of attacks.

7.5 Hardware implementation

We have currently no figures on the attainable performance and required area or gate count of KHAZAD in ASIC or FPGA, nor do we have a description in VHDL. However, we expect that the results on RIJNDAEL [14, 28] will carry over to some extent; in particular, the S-box structure can be implemented in about 1/5 the number of gates used by the implementation of the RIJNDAEL S-box reported in [29], which takes about 500–600 gates [30].

8 Efficiency estimates

To obtain efficiency estimates we used the 32-bit implementation with eight tables described in sections 7.1 and 7.2 on a 550 MHz Pentium III processor. Because the key schedule is based on the round function itself, no extra storage is needed besides that already required for implementing encryption/decryption.

8.1 Key setup

Table 3 lists the observed key setup efficiency. The increased cost of the decryption key schedule (68% more expensive than the setup of the encryption key schedule) is due to the application of θ to $R - 1$ round keys.

Table 3. Key setup efficiency

cycles (encryption schedule)	cycles (decryption schedule)
640	1076

8.2 Encryption and decryption

Since KHAZAD has involutonal structure, encryption and decryption are equally efficient (for the same number of rounds). Table 4 summarises the observed efficiency.

Table 4. Encryption/decryption efficiency

cycles per byte	cycles per block	Mbit/s
51.2	409	86.0

By coding the primitive in assembler and running the test on a native 64-bit processor, we expect a reduction of the cycle counts by a factor of at least 2.

9 Advantages

KHAZAD is much more scalable than most modern ciphers, in the sense of being very fast while avoiding excessive storage space (for both code and tables) and expensive or unusual instructions built in the processor; this makes it suitable for a wide variety of platforms. The same structure also favours extensively parallel execution of the component mappings, and its mathematical simplicity tends to make analysis easier.

9.1 Comparison with SHARK

KHAZAD bears many similarities with the block cipher SHARK [26]. We now list the most important differences.

The involutinal structure: The fact that all components of KHAZAD are involutions should in principle reduce the code size or area in software, respectively hardware applications that implement both encryption and decryption.

The different S-box: The S-box of KHAZAD contains elements generated in a pseudo-random way and lacks a simple mathematical description needed for e.g. interpolation attacks; besides, the internal organisation of these elements potentially facilitates hardware implementation. On the other hand the differential and linear properties are suboptimal.

The different key scheduling: The KHAZAD key scheduling executes much faster than the SHARK counterpart, and still provides adequate security. In particular for the processing of short messages, the performance of the key scheduling is important.

10 Acknowledgements

We are grateful to the Cryptix development team, and particularly to Raif Naffah, for carefully reading and suggesting improvements for the implementation guidelines provided in this paper, and for implementing several versions of KHAZAD in Java.

We are deeply indebted to Brian Gladman for providing software and hardware facilities to search for efficient mini-box implementations in terms of Boolean functions.

We would also like to thank the NESSIE project organisers and evaluation team for making this work possible.

References

1. P.S.L.M. Barreto and V. Rijmen, “The ANUBIS block cipher,” NESSIE submission, 2000.
2. K.G. Beauchamp, “Walsh functions and their applications,” Academic Press, 1975.
3. E. Biham, O. Dunkelman, N. Keller, “The Rectangle Attack - Rectangling the Serpent,” *Advances in Cryptology, Eurocrypt’2001, LNCS 2045*, B. Pfitzmann, Ed., Springer-Verlag, 2001, pp. 340–357.
4. E. Biham, A. Biryukov, A. Shamir, “Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials,” *Advances in Cryptology, Eurocrypt’99, LNCS 1592*, J. Stern, Ed., Springer-Verlag, 1999, pp. 55–64.
5. E. Biham and N. Keller, “Cryptanalysis of reduced variants of RIJNDAEL,” submission to the *Third Advanced Encryption Standard Candidate Conference*.
6. A. Biryukov and D. Wagner, “Slide attacks,” *Fast Software Encryption, LNCS 1636*, L. Knudsen, Ed., Springer-Verlag, 1999, pp. 245–259.

7. A. Biryukov and D. Wagner, "Advanced slide attacks," *Fast Software Encryption, LNCS 1807*, B. Preneel, Ed., Springer-Verlag, 2000, pp. 589–606.
8. J. Daemen, "Cipher and hash function design strategies based on linear and differential cryptanalysis," *Doctoral Dissertation*, March 1995, K.U.Leuven.
9. J. Daemen, L.R. Knudsen and V. Rijmen, "The block cipher SQUARE," *Fast Software Encryption, LNCS 1267*, E. Biham, Ed., Springer-Verlag, 1997, pp. 149–165.
10. J. Daemen and V. Rijmen, "AES proposal: RIJNDAEL," AES submission (1998), <http://www.nist.gov/aes>.
11. H. Feistel, "Cryptography and computer privacy," *Scientific American*, v. 228, n. 5, 1973, pp. 15–23.
12. N. Ferguson, J. Kelsey, S. Lucks, B. Schneier, M. Stay, D. Wagner, and D. Whiting, "Improved cryptanalysis of RIJNDAEL," *Fast Software Encryption, LNCS 1978*, B. Schneier, Ed., Springer-Verlag, 2001.
13. H. Gilbert and M. Minier, "A collision attack on 7 rounds of Rijndael," *Third Advanced Encryption Standard Candidate Conference*, NIST, April 2000, pp. 230–241.
14. T. Ichikawa, T. Kasuya, M. Matsui, "Hardware evaluation of the AES finalists," *Third Advanced Encryption Standard Candidate Conference*, NIST, April 2000, pp. 279–285.
15. T. Jakobsen and L.R. Knudsen, "The interpolation attack on block ciphers," *Fast Software Encryption, LNCS 1267*, E. Biham, Ed., Springer-Verlag, 1997, pp. 28–40.
16. P.C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," *Advances in Cryptology, Crypto '96, LNCS 1109*, N. Kobitz, Ed., Springer-Verlag, 1996, pp. 104–113.
17. P. Kocher, J. Jaffe, B. Jun, "Introduction to differential power analysis and related attacks," available from <http://www.cryptography.com/dpa/technical/>.
18. L.R. Knudsen, "Truncated and higher order differentials," *Fast Software Encryption, LNCS 1008*, B. Preneel, Ed., Springer-Verlag, 1995, pp. 196–211.
19. L.R. Knudsen, M.J.B. Robshaw, "Non-linear approximations in linear cryptanalysis," *Advances in Cryptology, Eurocrypt'96, LNCS 1070*, U. Maurer, Ed., Springer-Verlag, 1996, pp. 224–236.
20. R. Lidl and H. Niederreiter, "Introduction to finite fields and their applications," Cambridge University Press, 1986.
21. S. Lucks, "Attacking seven rounds of RIJNDAEL under 192-bit and 256-bit keys," *Third Advanced Encryption Standard Candidate Conference*, NIST, April 2000, pp. 215–229.
22. F.J. MacWilliams and N.J.A. Sloane, "The theory of error-correcting codes," *North-Holland Mathematical Library*, vol. 16, 1977.
23. NESSIE Project – New European Schemes for Signatures, Integrity and Encryption – home page: <http://cryptonessie.org>.
24. NESSIE evaluation team, private communication, 2001; see also "Security Evaluation of NESSIE First Phase," NESSIE deliverable D13, S. Murphy and J. White, Eds., 2001 – available online at: <https://www.cosic.esat.kuleuven.ac.be/nessie/deliverables/D13.pdf>.
25. V. Rijmen, "Cryptanalysis and design of iterated block ciphers," *Doctoral Dissertation*, October 1997, K.U.Leuven.
26. V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, E. De Win, "The cipher SHARK," *Fast Software Encryption, LNCS 1039*, D. Gollman, Ed., Springer-Verlag, 1996, pp. 99–111.
27. D. Wagner, "The boomerang attack," *Fast Software Encryption, LNCS 1636*, L. Knudsen, Ed., Springer-Verlag, 1999, pp. 156–170.

28. B. Weeks, M. Bean, T. Rozyłowicz, C. Ficke, “Hardware performance simulations of round 2 AES algorithms,” *Third Advanced Encryption Standard Candidate Conference*, NIST, April 2000, pp. 286–304.
29. D. Whiting, “AES implementations in 0.25 micron ASIC,” NIST AES electronic discussion forum posting, August 2000.
30. D. Whiting, private communication, 2001.
31. A.M. Youssef, S.E. Tavares, and H.M. Heys, “A new class of substitution-permutation networks,” *Workshop on Selected Areas in Cryptography, SAC’96*, Workshop record, 1996, pp. 132–147.
32. A.M. Youssef, S. Mister, and S.E. Tavares, “On the design of linear transformations for substitution permutation encryption networks,” *Workshop on Selected Areas of Cryptography, SAC’97*, Workshop record, 1997, pp. 40–48.

A Generation of the KHAZAD S-box

KHAZAD uses the same S-box as the ANUBIS cipher (cf. [1]); we describe here how it was generated for ease of reference.

The only part of the S-box structure still unspecified in figure 1 consists of the P and Q involutions, which are generated pseudo-randomly in a verifiable way.

The searching algorithm starts with two copies of a simple involution without fixed points (namely, the negation mapping $u \mapsto \bar{u} = u \oplus \mathbf{F}_x$), and pseudo-randomly derives from each of them a sequence of 4×4 substitution boxes (“mini-boxes”) with the optimal values $\delta = 1/4$, $\lambda = 1/2$, and $\nu = 3$. At each step, in alternation, only one of the sequences is extended with a new mini-box. The most recently generated mini-box from each sequence is taken, and the pair is combined according to the shuffle structure shown in figure 1; finally, the resulting 8×8 S-box, if free of fixed points, is tested for the design criteria regarding δ , λ , and ν .

Given a mini-box at any point during the search, a new one is derived from it by choosing two pairs of mutually inverse values and swapping them, keeping the result an involution without fixed points; this is repeated until the running mini-box has optimal values of δ , λ , and ν .

The pseudo-random number generator is implemented with RIJNDAEL [10] in counter mode, with a fixed key consisting of 256 zero bits and an initial counter value consisting of 128 zero bits.

The following pseudo-code fragment illustrates the computation of the chains of mini-boxes and the resulting S-box:

```

// initialize mini-boxes to the negation involution:
for ( $u \leftarrow 0$ ;  $u < 256$ ;  $u++$ ) {
     $P[u] \leftarrow \bar{u}$ ;  $Q[u] \leftarrow \bar{u}$ ;
}
// look for S-box conforming to the design criteria:
do {
    // swap mini-boxes (update the “older” one only)

```

```

P ↔ Q;
// generate a random involution free of fixed points:
do {
  do {
    // randomly select x and y such that
    // x ≠ y and Q[x] ≠ y (this implies Q[y] ≠ x):
    z ← RandomByte(); x ← z ≫ 4; y ← z & 0Fx;
  } while (x = y ∨ Q[x] = y);
  // swap entries:
  u ← Q[x]; v ← Q[y];
  Q[x] ← v; Q[u] ← y;
  Q[y] ← u; Q[v] ← x;
} while (δ(Q) > 1/4 ∨ λ(Q) > 1/2 ∨ ν(Q) < 3);
// build S-box from the mini-boxes (see figure 1):
S ← ShuffleStructure(P, Q);
// test design criteria:
} while (#FixedPoints(S) > 0 ∨ δ(S) > 2-5 ∨ λ(S) > 2-2 ∨ ν(S) < 7);

```

B Hardware implementation

Restricting the allowed logical gates to AND, OR, NOT, and XOR, the P and Q mini-boxes can be implemented with 18 logical gates each. Therefore, the complete S-box can be implemented with 108 gates.

The pseudo-code fragments shown in figure 2 illustrate this ($u = u_3x^3 + u_2x^2 + u_1x + u_0 \in \text{GF}(2^4)$ denotes the mini-box input, $z = z_3x^3 + z_2x^2 + z_1x + z_0 \in \text{GF}(2^4)$ denotes its output, and the t_k denote intermediate values). We point out, however, that the search for efficient Boolean expressions for the mini-boxes has not been thorough, and it is likely that better expressions exist.

For completeness, table 5 lists the resulting 8×8 KHAZAD S-box.

C The name

*Thus he brought me back at last to the secret ways of
Khazad-dûm . . .*

KHAZAD is named after *Khazad-dûm*, “the Mansion of the Khazâd,” which in the tongue of the Dwarves is the name of the great realm and city of Dwarrowdelf, of the haunted *mithril* mines in Moria, the Black Chasm. But all this should be quite obvious – unless you haven’t read J.R.R. Tolkien’s “The Lord of the Rings,” of course :-)

$z = P[u]$	$z = Q[u]$
$t_0 \leftarrow u_0 \oplus u_1;$	$t_0 \leftarrow \neg u_0;$
$t_1 \leftarrow u_0 \oplus u_3;$	$t_1 \leftarrow u_1 \oplus u_2;$
$t_2 \leftarrow u_2 \wedge t_1;$	$t_2 \leftarrow u_2 \wedge t_0;$
$t_3 \leftarrow u_3 \wedge t_1;$	$t_3 \leftarrow u_3 \oplus t_2;$
$t_4 \leftarrow t_0 \vee t_3;$	$t_4 \leftarrow t_1 \wedge t_3;$
$z_3 \leftarrow t_2 \oplus t_4;$	$z_0 \leftarrow t_0 \oplus t_4;$
$t_1 \leftarrow \neg t_1;$	$t_0 \leftarrow u_0 \oplus u_1;$
$t_2 \leftarrow u_1 \wedge u_2;$	$t_1 \leftarrow t_1 \oplus t_2;$
$t_4 \leftarrow u_3 \vee z_3;$	$t_0 \leftarrow t_0 \oplus t_3;$
$t_1 \leftarrow t_1 \oplus t_2;$	$t_1 \leftarrow t_1 \vee t_0;$
$z_0 \leftarrow t_4 \oplus t_1;$	$z_2 \leftarrow u_2 \oplus t_1;$
$t_4 \leftarrow u_2 \wedge t_1;$	$t_1 \leftarrow t_1 \wedge u_0;$
$t_2 \leftarrow t_2 \oplus u_3;$	$t_3 \leftarrow u_3 \wedge t_0;$
$t_2 \leftarrow t_2 \vee t_4;$	$t_3 \leftarrow t_3 \oplus t_2;$
$z_2 \leftarrow t_0 \oplus t_2;$	$z_1 \leftarrow t_1 \oplus t_3;$
$t_3 \leftarrow t_3 \oplus t_4;$	$t_1 \leftarrow u_2 \vee z_0;$
$t_1 \leftarrow t_1 \vee z_3;$	$t_0 \leftarrow t_0 \oplus t_3;$
$z_1 \leftarrow t_3 \oplus t_1;$	$z_3 \leftarrow t_1 \oplus t_0;$

Fig. 2. Boolean expressions for P and Q

Table 5. The KHAZAD S-box

	00 _x	01 _x	02 _x	03 _x	04 _x	05 _x	06 _x	07 _x	08 _x	09 _x	0A _x	0B _x	0C _x	0D _x	0E _x	0F _x
00 _x	BA _x	54 _x	2F _x	74 _x	53 _x	D3 _x	D2 _x	4D _x	50 _x	AC _x	8D _x	BF _x	70 _x	52 _x	9A _x	4C _x
10 _x	EA _x	D5 _x	97 _x	D1 _x	33 _x	51 _x	5B _x	A6 _x	DE _x	48 _x	A8 _x	99 _x	DB _x	32 _x	B7 _x	FC _x
20 _x	E3 _x	9E _x	91 _x	9B _x	E2 _x	BB _x	41 _x	6E _x	A5 _x	CB _x	6B _x	95 _x	A1 _x	F3 _x	B1 _x	02 _x
30 _x	CC _x	C4 _x	1D _x	14 _x	C3 _x	63 _x	DA _x	5D _x	5F _x	DC _x	7D _x	CD _x	7F _x	5A _x	6C _x	5C _x
40 _x	F7 _x	26 _x	FF _x	ED _x	E8 _x	9D _x	6F _x	8E _x	19 _x	A0 _x	FO _x	89 _x	0F _x	07 _x	AF _x	FB _x
50 _x	08 _x	15 _x	0D _x	04 _x	01 _x	64 _x	DF _x	76 _x	79 _x	DD _x	3D _x	16 _x	3F _x	37 _x	6D _x	38 _x
60 _x	B9 _x	73 _x	E9 _x	35 _x	55 _x	71 _x	7B _x	8C _x	72 _x	88 _x	F6 _x	2A _x	3E _x	5E _x	27 _x	46 _x
70 _x	0C _x	65 _x	68 _x	61 _x	03 _x	C1 _x	57 _x	D6 _x	D9 _x	58 _x	D8 _x	66 _x	D7 _x	3A _x	C8 _x	3C _x
80 _x	FA _x	96 _x	A7 _x	98 _x	EC _x	B8 _x	C7 _x	AE _x	69 _x	4B _x	AB _x	A9 _x	67 _x	0A _x	47 _x	F2 _x
90 _x	B5 _x	22 _x	E5 _x	EE _x	BE _x	2B _x	81 _x	12 _x	83 _x	1B _x	0E _x	23 _x	F5 _x	45 _x	21 _x	CE _x
A0 _x	49 _x	2C _x	F9 _x	E6 _x	B6 _x	28 _x	17 _x	82 _x	1A _x	8B _x	FE _x	8A _x	09 _x	C9 _x	87 _x	4E _x
B0 _x	E1 _x	2E _x	E4 _x	E0 _x	EB _x	90 _x	A4 _x	1E _x	85 _x	60 _x	00 _x	25 _x	F4 _x	F1 _x	94 _x	0B _x
C0 _x	E7 _x	75 _x	EF _x	34 _x	31 _x	D4 _x	D0 _x	86 _x	7E _x	AD _x	FD _x	29 _x	30 _x	3B _x	9F _x	F8 _x
D0 _x	C6 _x	13 _x	06 _x	05 _x	C5 _x	11 _x	77 _x	7C _x	7A _x	78 _x	36 _x	1C _x	39 _x	59 _x	18 _x	56 _x
E0 _x	B3 _x	B0 _x	24 _x	20 _x	B2 _x	92 _x	A3 _x	C0 _x	44 _x	62 _x	10 _x	B4 _x	84 _x	43 _x	93 _x	C2 _x
F0 _x	4A _x	BD _x	8F _x	2D _x	BC _x	9C _x	6A _x	40 _x	CF _x	A2 _x	80 _x	4F _x	1F _x	CA _x	AA _x	42 _x

Test vectors -- set 1
=====

Set 1, vector# 0:
key=80000000000000000000000000000000
plain=0000000000000000
cipher=49A4CE32AC190E3F
decrypted=0000000000000000
Iterated 100 times=61FD7EF96CEF52C3
Iterated 1000 times=012072FF15CED085

Set 1, vector# 1:
key=40000000000000000000000000000000
plain=0000000000000000
cipher=BD2226C1128B4AD1
decrypted=0000000000000000
Iterated 100 times=55044C31A26A0F34
Iterated 1000 times=00BEB055592B3F06

Set 1, vector# 2:
key=20000000000000000000000000000000
plain=0000000000000000
cipher=A3C8D3CAB9D196BC
decrypted=0000000000000000
Iterated 100 times=C37E991495ABF57B
Iterated 1000 times=8FC10022B21D781B

Set 1, vector# 3:
key=10000000000000000000000000000000
plain=0000000000000000
cipher=2C8146E405C2EA36
decrypted=0000000000000000
Iterated 100 times=C5CF4F50B526DEBA
Iterated 1000 times=2F215146A89B245C

Set 1, vector# 4:
key=08000000000000000000000000000000
plain=0000000000000000
cipher=9EC02CFC7065D8F8
decrypted=0000000000000000
Iterated 100 times=37FFF3E2C2ECA536
Iterated 1000 times=E64D32D44A1EEE06

Set 1, vector# 5:
key=04000000000000000000000000000000
plain=0000000000000000
cipher=8000A8E00368192F
decrypted=0000000000000000
Iterated 100 times=EAAB29459F5E00B9
Iterated 1000 times=5625F8D85BF5C654

Set 1, vector# 6:
key=02000000000000000000000000000000
plain=0000000000000000
cipher=1EE763EE6BACF669
decrypted=0000000000000000
Iterated 100 times=AF9DBC1D34FFB410
Iterated 1000 times=DC41396677CE7306

Set 1, vector# 7:
key=01000000000000000000000000000000
plain=0000000000000000
cipher=37F1F5997C673921
decrypted=0000000000000000
Iterated 100 times=B4B0B7CB7910E6AF

Iterated 1000 times=86CDA708E36F1B82

Set 1, vector# 8:
key=00800000000000000000000000000000
plain=00000000000000000000000000000000
cipher=A1F1887BDCD62492
decrypted=00000000000000000000000000000000
Iterated 100 times=37DBEA9357279F6E
Iterated 1000 times=4B05EFF2B7711F31

Set 1, vector# 9:
key=00400000000000000000000000000000
plain=00000000000000000000000000000000
cipher=B9AD5EEB66429D84
decrypted=00000000000000000000000000000000
Iterated 100 times=D764D339D772DFCD
Iterated 1000 times=67923E0D0C5D6A93

Set 1, vector# 10:
key=00200000000000000000000000000000
plain=00000000000000000000000000000000
cipher=99D05BA5B40CB879
decrypted=00000000000000000000000000000000
Iterated 100 times=5524DDCB0FD940DF
Iterated 1000 times=08373C3C114C28AA

Set 1, vector# 11:
key=00100000000000000000000000000000
plain=00000000000000000000000000000000
cipher=B32493C83EB3F395
decrypted=00000000000000000000000000000000
Iterated 100 times=55D44AEE01C2CD13
Iterated 1000 times=79DEAB5F359BC482

Set 1, vector# 12:
key=00080000000000000000000000000000
plain=00000000000000000000000000000000
cipher=9298A32F99516E05
decrypted=00000000000000000000000000000000
Iterated 100 times=995E2E7922FE9325
Iterated 1000 times=35FCF917DFB2544B

Set 1, vector# 13:
key=00040000000000000000000000000000
plain=00000000000000000000000000000000
cipher=B23577E8035B1EA5
decrypted=00000000000000000000000000000000
Iterated 100 times=E7F88C416420AD0D
Iterated 1000 times=F5E2D5818D500897

Set 1, vector# 14:
key=00020000000000000000000000000000
plain=00000000000000000000000000000000
cipher=140BE694E0E68B9A
decrypted=00000000000000000000000000000000
Iterated 100 times=27A622097FEC394B
Iterated 1000 times=D739E83FFB097455

Set 1, vector# 15:
key=00010000000000000000000000000000
plain=00000000000000000000000000000000
cipher=BE0114504A5AB78C
decrypted=00000000000000000000000000000000
Iterated 100 times=A1C979ACF3186D25
Iterated 1000 times=9555857E4D39CA93

Set 1, vector# 16:
key=00008000000000000000000000000000
plain=00000000000000000000000000000000
cipher=CDC09EB17D04D355
decrypted=00000000000000000000000000000000
Iterated 100 times=F4EF3D29C39FA340
Iterated 1000 times=478BB698C55C2E05

Set 1, vector# 17:
key=00004000000000000000000000000000
plain=00000000000000000000000000000000
cipher=3C56C3CF9EA42A0F
decrypted=00000000000000000000000000000000
Iterated 100 times=32E1CA900737F4C6
Iterated 1000 times=3D42F3CD03D9BBFD

Set 1, vector# 18:
key=00002000000000000000000000000000
plain=00000000000000000000000000000000
cipher=B9D4BA273E59DF2A
decrypted=00000000000000000000000000000000
Iterated 100 times=720291B7A4486FEE
Iterated 1000 times=AFD2107BFF65C91B

Set 1, vector# 19:
key=00001000000000000000000000000000
plain=00000000000000000000000000000000
cipher=610BB190DB22219C
decrypted=00000000000000000000000000000000
Iterated 100 times=50246CFFB6895E25
Iterated 1000 times=0695FC898C6F6901

Set 1, vector# 20:
key=00000800000000000000000000000000
plain=00000000000000000000000000000000
cipher=F340006AB65403D3
decrypted=00000000000000000000000000000000
Iterated 100 times=4FC4266822AB392B
Iterated 1000 times=A7F3AA960383BD2B

Set 1, vector# 21:
key=00000400000000000000000000000000
plain=00000000000000000000000000000000
cipher=C046B6668971E5E8
decrypted=00000000000000000000000000000000
Iterated 100 times=930C28BE423412CA
Iterated 1000 times=5BD3FCA0F6CBAEC0

Set 1, vector# 22:
key=00000200000000000000000000000000
plain=00000000000000000000000000000000
cipher=28C4E35F5C7C8147
decrypted=00000000000000000000000000000000
Iterated 100 times=948E5E0421133835
Iterated 1000 times=CA9F7B97BFF671AD

Set 1, vector# 23:
key=00000100000000000000000000000000
plain=00000000000000000000000000000000
cipher=F576AF998281AC12
decrypted=00000000000000000000000000000000
Iterated 100 times=DDADAA1FBB72A620
Iterated 1000 times=5FBC1BBBDD9B197

Set 1, vector# 24:
key=00000080000000000000000000000000
plain=0000000000000000
cipher=7BF090A9035A7A90
decrypted=0000000000000000
Iterated 100 times=1A81A4C93EBC49DF
Iterated 1000 times=862DF8D1BF501339

Set 1, vector# 25:
key=00000040000000000000000000000000
plain=0000000000000000
cipher=4B24098F381E31AF
decrypted=0000000000000000
Iterated 100 times=C2CE129E73B126A9
Iterated 1000 times=2B0505034404BFBD

Set 1, vector# 26:
key=00000020000000000000000000000000
plain=0000000000000000
cipher=C161F51F307AC93E
decrypted=0000000000000000
Iterated 100 times=0DC45A9C89ED337F
Iterated 1000 times=16442A406EC67BC3

Set 1, vector# 27:
key=00000010000000000000000000000000
plain=0000000000000000
cipher=C515DFB65DF8AF9C
decrypted=0000000000000000
Iterated 100 times=7BF9BC2718EB5DF9
Iterated 1000 times=C88BDEB25A3B1663

Set 1, vector# 28:
key=00000008000000000000000000000000
plain=0000000000000000
cipher=7791F72D6A6413E2
decrypted=0000000000000000
Iterated 100 times=3A8A5EAF11B8D81
Iterated 1000 times=D9A494B1F2B6D993

Set 1, vector# 29:
key=00000004000000000000000000000000
plain=0000000000000000
cipher=EBB35B41832E1D43
decrypted=0000000000000000
Iterated 100 times=DC219EA66A469A65
Iterated 1000 times=7344BE72B4A5627C

Set 1, vector# 30:
key=00000002000000000000000000000000
plain=0000000000000000
cipher=5167B68060821923
decrypted=0000000000000000
Iterated 100 times=E6E1B2ADDC52E7E3
Iterated 1000 times=1104D6D6CC33BA94

Set 1, vector# 31:
key=00000001000000000000000000000000
plain=0000000000000000
cipher=30FC8AEE3E2166F4
decrypted=0000000000000000
Iterated 100 times=BE2A33459CC4495F

Iterated 1000 times=7FF74C82CD13D919

Set 1, vector# 32:

key=00000000800000000000000000000000
plain=0000000000000000
cipher=35F53E915255E1C8
decrypted=0000000000000000
Iterated 100 times=49567797E7DCBB27
Iterated 1000 times=25356EE882A7EE54

Set 1, vector# 33:

key=00000000400000000000000000000000
plain=0000000000000000
cipher=0AEAE292D1CB2132
decrypted=0000000000000000
Iterated 100 times=024CFE4C82559A84
Iterated 1000 times=03643AD45FD0DF68

Set 1, vector# 34:

key=00000000200000000000000000000000
plain=0000000000000000
cipher=24CCBC45194523AA
decrypted=0000000000000000
Iterated 100 times=D1BB13125D4401C8
Iterated 1000 times=B16BF84942741365

Set 1, vector# 35:

key=00000000100000000000000000000000
plain=0000000000000000
cipher=A27D9DC375C08A16
decrypted=0000000000000000
Iterated 100 times=10E8BF9F04372F57
Iterated 1000 times=7C2464290C960E57

Set 1, vector# 36:

key=00000000800000000000000000000000
plain=0000000000000000
cipher=916C0A5D223C1925
decrypted=0000000000000000
Iterated 100 times=18F76E71533788D8
Iterated 1000 times=87F439BF48A95FCE

Set 1, vector# 37:

key=00000000400000000000000000000000
plain=0000000000000000
cipher=FC20842C1EE0C83E
decrypted=0000000000000000
Iterated 100 times=3568CE8018A32DB5
Iterated 1000 times=C663252106DC7197

Set 1, vector# 38:

key=00000000200000000000000000000000
plain=0000000000000000
cipher=A203778229A5EE4D
decrypted=0000000000000000
Iterated 100 times=C81C586ECA74319A
Iterated 1000 times=FCDAF54FA0C1B51F

Set 1, vector# 39:

key=00000000010000000000000000000000
plain=0000000000000000
cipher=A2A595336AC58A30
decrypted=0000000000000000

Iterated 100 times=4E744E9C0E20B6FE
Iterated 1000 times=6E7949218874F030

Set 1, vector# 40:

key=00000000008000000000000000000000
plain=0000000000000000
cipher=7E8A99132C842AA1
decrypted=0000000000000000
Iterated 100 times=81070B84B9608DA3
Iterated 1000 times=4E2D79C04AB16435

Set 1, vector# 41:

key=00000000004000000000000000000000
plain=0000000000000000
cipher=383E893BDB186C48
decrypted=0000000000000000
Iterated 100 times=E0F80EF997EDF03B
Iterated 1000 times=ED4EDA8247208DC0

Set 1, vector# 42:

key=00000000002000000000000000000000
plain=0000000000000000
cipher=57F443B7FEC2B948
decrypted=0000000000000000
Iterated 100 times=C8E5EB832F282A62
Iterated 1000 times=36944A83C63A4242

Set 1, vector# 43:

key=00000000001000000000000000000000
plain=0000000000000000
cipher=03A5AF737C23B1F9
decrypted=0000000000000000
Iterated 100 times=923AD1887760EB4A
Iterated 1000 times=B804DE103BEAAC29

Set 1, vector# 44:

key=00000000008000000000000000000000
plain=0000000000000000
cipher=FB0CF0796F89F6E0
decrypted=0000000000000000
Iterated 100 times=E8717BF0B37186D1
Iterated 1000 times=57D5DDAFB333B5F8

Set 1, vector# 45:

key=00000000004000000000000000000000
plain=0000000000000000
cipher=BC48E780ED48FC73
decrypted=0000000000000000
Iterated 100 times=3CA5A1055275E91A
Iterated 1000 times=6DE74E0F860C8476

Set 1, vector# 46:

key=00000000002000000000000000000000
plain=0000000000000000
cipher=8C1094794C4ECB3D
decrypted=0000000000000000
Iterated 100 times=F3CF5E35B2F96E15
Iterated 1000 times=F75FD35977E7FA74

Set 1, vector# 47:

key=00000000001000000000000000000000
plain=0000000000000000
cipher=9F7C9A338AEFCD41

decrypted=0000000000000000
Iterated 100 times=CD7D9C17AD63EB10
Iterated 1000 times=47428FF5F40F710E

Set 1, vector# 48:

key=00000000000080000000000000000000
plain=0000000000000000
cipher=AA922DA7AF1457B0
decrypted=0000000000000000
Iterated 100 times=FED5540F162F3C0A
Iterated 1000 times=717FF4CB8C297091

Set 1, vector# 49:

key=00000000000040000000000000000000
plain=0000000000000000
cipher=431E171B58A64E30
decrypted=0000000000000000
Iterated 100 times=33E83412B6C33A23
Iterated 1000 times=001A502C41946FB4

Set 1, vector# 50:

key=00000000000020000000000000000000
plain=0000000000000000
cipher=8D466B407358BC2F
decrypted=0000000000000000
Iterated 100 times=91231B3513F2F048
Iterated 1000 times=88863FC0E05C6981

Set 1, vector# 51:

key=00000000000010000000000000000000
plain=0000000000000000
cipher=2DEC789A694BF5B3
decrypted=0000000000000000
Iterated 100 times=7C550145E9294C54
Iterated 1000 times=95746D7F71D696B8

Set 1, vector# 52:

key=00000000000080000000000000000000
plain=0000000000000000
cipher=74F06925A915664A
decrypted=0000000000000000
Iterated 100 times=152D89EA223BCFFB
Iterated 1000 times=168C19D54EEEAFD1

Set 1, vector# 53:

key=00000000000040000000000000000000
plain=0000000000000000
cipher=427F2D49FF1AC1FB
decrypted=0000000000000000
Iterated 100 times=3AE272AEA736825C
Iterated 1000 times=ED5E2FB908DF18D5

Set 1, vector# 54:

key=00000000000020000000000000000000
plain=0000000000000000
cipher=1241BE7F76DC8273
decrypted=0000000000000000
Iterated 100 times=9E8B5271486E590C
Iterated 1000 times=8746D316126BA745

Set 1, vector# 55:

key=00000000000010000000000000000000
plain=0000000000000000

cipher=C2B0F299B1531293
decrypted=0000000000000000
Iterated 100 times=FE49763C440225A3
Iterated 1000 times=30F3959917A9A655

Set 1, vector# 56:

key=00000000000000080000000000000000
plain=0000000000000000
cipher=D22047FC06E3E6EE
decrypted=0000000000000000
Iterated 100 times=48C67917CBED19C7
Iterated 1000 times=CB6985B1C571AEEF

Set 1, vector# 57:

key=00000000000000400000000000000000
plain=0000000000000000
cipher=FA1A49B57CA77F43
decrypted=0000000000000000
Iterated 100 times=6C5E61AE44BC9F92
Iterated 1000 times=AC4608E04E6D849D

Set 1, vector# 58:

key=00000000000000200000000000000000
plain=0000000000000000
cipher=3196214D6ADC12FC
decrypted=0000000000000000
Iterated 100 times=C8E1828FCB3D73D3
Iterated 1000 times=DA96079B2E7683C3

Set 1, vector# 59:

key=00000000000000100000000000000000
plain=0000000000000000
cipher=2A9D9922F5A7BC9C
decrypted=0000000000000000
Iterated 100 times=76B8D71BAE89432D
Iterated 1000 times=468494A2B74B3F22

Set 1, vector# 60:

key=00000000000000800000000000000000
plain=0000000000000000
cipher=98C7B2C0EE8A76B2
decrypted=0000000000000000
Iterated 100 times=700C896A4E1EAB6C
Iterated 1000 times=6C66AAE5B9FD8C9A

Set 1, vector# 61:

key=00000000000000400000000000000000
plain=0000000000000000
cipher=7AF5C2601CD26A5F
decrypted=0000000000000000
Iterated 100 times=C6E12E2732649F86
Iterated 1000 times=B787DB907B234830

Set 1, vector# 62:

key=00000000000000200000000000000000
plain=0000000000000000
cipher=7F89E6F27DA017DA
decrypted=0000000000000000
Iterated 100 times=89C12DC8621D44FF
Iterated 1000 times=3AF0E574F48AEFAE

Set 1, vector# 63:

key=00000000000000100000000000000000

plain=0000000000000000
cipher=95562543AA40D405
decrypted=0000000000000000
Iterated 100 times=B9050DF582B0A65A
Iterated 1000 times=977E69B571DB5D5A

Set 1, vector# 64:

key=00000000000000008000000000000000
plain=0000000000000000
cipher=8BB387854F57AA71
decrypted=0000000000000000
Iterated 100 times=FFD22CAEA7DB4B5F
Iterated 1000 times=A255E3183E8A38F5

Set 1, vector# 65:

key=00000000000000004000000000000000
plain=0000000000000000
cipher=43FF079D45E73E2F
decrypted=0000000000000000
Iterated 100 times=232E12A4B0A1997E
Iterated 1000 times=E468363058A659F3

Set 1, vector# 66:

key=00000000000000002000000000000000
plain=0000000000000000
cipher=1C6FE1C745EE9FB7
decrypted=0000000000000000
Iterated 100 times=F84CEFC165DDC3FA
Iterated 1000 times=EA4CEC47B4C68296

Set 1, vector# 67:

key=00000000000000001000000000000000
plain=0000000000000000
cipher=F4E8B2DB8A74C16E
decrypted=0000000000000000
Iterated 100 times=383DE23BAC182991
Iterated 1000 times=E6B9225D30E75B7E

Set 1, vector# 68:

key=00000000000000008000000000000000
plain=0000000000000000
cipher=E0CC338D0309C6A5
decrypted=0000000000000000
Iterated 100 times=E2E884E99E891635
Iterated 1000 times=DFCD5F4D559221B7

Set 1, vector# 69:

key=00000000000000004000000000000000
plain=0000000000000000
cipher=DE439BBD9E85BA17
decrypted=0000000000000000
Iterated 100 times=86DD4B4F0B7F61B5
Iterated 1000 times=B3F0FC89FDE5C60E

Set 1, vector# 70:

key=00000000000000002000000000000000
plain=0000000000000000
cipher=22F711183665D455
decrypted=0000000000000000
Iterated 100 times=854FF029FECF8A14
Iterated 1000 times=C180CFA5F4934FFC

Set 1, vector# 71:

Iterated 100 times=3AC062217C5F9374
Iterated 1000 times=2EEEBB83ACFFE091D

Set 1, vector#103:

key=0000000000000000000000001000000
plain=0000000000000000
cipher=5D83F838F592F2DB
decrypted=0000000000000000
Iterated 100 times=D1C9907330EE3F1E
Iterated 1000 times=440FEA16E494FF1B

Set 1, vector#104:

key=0000000000000000000000000800000
plain=0000000000000000
cipher=58B3140D2EB4E931
decrypted=0000000000000000
Iterated 100 times=73FB65AE7162A6FA
Iterated 1000 times=762ADD07D9DA7277

Set 1, vector#105:

key=0000000000000000000000000400000
plain=0000000000000000
cipher=9BDE6B193F3840E5
decrypted=0000000000000000
Iterated 100 times=9C1835028ADDBD1D
Iterated 1000 times=3A00AB6D182C2A00

Set 1, vector#106:

key=0000000000000000000000000200000
plain=0000000000000000
cipher=3C9DBB93F2E955F5
decrypted=0000000000000000
Iterated 100 times=3349CE416EB1CBFB
Iterated 1000 times=30E64E67ADCAF159

Set 1, vector#107:

key=0000000000000000000000000100000
plain=0000000000000000
cipher=EAB213645DB16DF9
decrypted=0000000000000000
Iterated 100 times=E5910FE3A9F61B0A
Iterated 1000 times=38BEFE3368D050E6

Set 1, vector#108:

key=000000000000000000000000080000
plain=0000000000000000
cipher=0AE1D9D6C9BDAEB9
decrypted=0000000000000000
Iterated 100 times=FBE152FFDD40DCBD
Iterated 1000 times=49BA44D7A325CF96

Set 1, vector#109:

key=000000000000000000000000040000
plain=0000000000000000
cipher=4DD7C74688044A9A
decrypted=0000000000000000
Iterated 100 times=6921B8D331F14355
Iterated 1000 times=8FBC1CC351CB2B3B

Set 1, vector#110:

key=000000000000000000000000020000
plain=0000000000000000
cipher=10017A2B7D504674


```
plain=0000000000000000
cipher=ECC4949D1992491A
decrypted=0000000000000000
Iterated 100 times=BD39139B6D0E516C
Iterated 1000 times=517BB8F19CC14E66
```

Set 1, vector#127:

```
key=00000000000000000000000000000001
plain=0000000000000000
cipher=645D773E40ABDD53
decrypted=0000000000000000
Iterated 100 times=95D8A4304A03933A
Iterated 1000 times=0EE895DC06F9A009
```

Test vectors -- set 2
=====

Set 2, vector# 0:

```
key=00000000000000000000000000000000
plain=8000000000000000
cipher=9E399864F78ECA02
decrypted=8000000000000000
Iterated 100 times=F5B4EB0741FD742E
Iterated 1000 times=B91A19B78ADE17CB
```

Set 2, vector# 1:

```
key=00000000000000000000000000000000
plain=4000000000000000
cipher=3EABB25778098FF7
decrypted=4000000000000000
Iterated 100 times=1D351B8E3B43962C
Iterated 1000 times=7371BD97E645C20D
```

Set 2, vector# 2:

```
key=00000000000000000000000000000000
plain=2000000000000000
cipher=A359C027CB02BC47
decrypted=2000000000000000
Iterated 100 times=5774E0E83B8C2DF4
Iterated 1000 times=5183EE2A53C1EF48
```

Set 2, vector# 3:

```
key=00000000000000000000000000000000
plain=1000000000000000
cipher=36E62B8D8DDF2929
decrypted=1000000000000000
Iterated 100 times=EEA3045804E51E70
Iterated 1000 times=33FF041FE13CFF01
```

Set 2, vector# 4:

```
key=00000000000000000000000000000000
plain=0800000000000000
cipher=CB4204ACEDDFE80E
decrypted=0800000000000000
Iterated 100 times=C1EDE779DDA7D0D9
Iterated 1000 times=EFC46725E98EA8DF
```

Set 2, vector# 5:

```
key=00000000000000000000000000000000
plain=0400000000000000
cipher=EF33DA42D27535CE
decrypted=0400000000000000
Iterated 100 times=CD81E874389BC252
```

Iterated 1000 times=616B64CE84AD5B02

Set 2, vector# 6:
key=00000000000000000000000000000000
plain=0200000000000000
cipher=2780289382A498D3
decrypted=0200000000000000
Iterated 100 times=CDAC8B3117321137
Iterated 1000 times=B5B2CC0A2C1A7D5E

Set 2, vector# 7:
key=00000000000000000000000000000000
plain=0100000000000000
cipher=335CC26627D36D77
decrypted=0100000000000000
Iterated 100 times=09ED85D5A4F99DC1
Iterated 1000 times=6E3F699A91C7898E

Set 2, vector# 8:
key=00000000000000000000000000000000
plain=0080000000000000
cipher=2632C049F89488B4
decrypted=0080000000000000
Iterated 100 times=0AAB78F92103E73F
Iterated 1000 times=6DB5DCCBE8CCE2BD

Set 2, vector# 9:
key=00000000000000000000000000000000
plain=0040000000000000
cipher=3827E825E650E5CD
decrypted=0040000000000000
Iterated 100 times=907559E340662879
Iterated 1000 times=83B04F717C467F92

Set 2, vector# 10:
key=00000000000000000000000000000000
plain=0020000000000000
cipher=8AED6305731C75B0
decrypted=0020000000000000
Iterated 100 times=9F270E9FF32DA27E
Iterated 1000 times=66B3C451D121B277

Set 2, vector# 11:
key=00000000000000000000000000000000
plain=0010000000000000
cipher=110508C1B7F2C089
decrypted=0010000000000000
Iterated 100 times=793EBC7B02611AF1
Iterated 1000 times=4D2BBAE8A6D42C8F

Set 2, vector# 12:
key=00000000000000000000000000000000
plain=0008000000000000
cipher=0B8E0E3BFD07BAA1
decrypted=0008000000000000
Iterated 100 times=5DF094C916EEC5BF
Iterated 1000 times=551C2BB980673F1A

Set 2, vector# 13:
key=00000000000000000000000000000000
plain=0004000000000000
cipher=95F69BE26C69DABE
decrypted=0004000000000000
Iterated 100 times=6A69A6A3FB7A8AF4
Iterated 1000 times=7C5D3597DBB9F8A9

Set 2, vector# 14:
key=00000000000000000000000000000000
plain=0002000000000000
cipher=41F61F71EA7096C4
decrypted=0002000000000000
Iterated 100 times=33F6AF23DF236263
Iterated 1000 times=02C86C1CFEC2FABB

Set 2, vector# 15:
key=00000000000000000000000000000000
plain=0001000000000000
cipher=BC623482919D6C66
decrypted=0001000000000000
Iterated 100 times=B9BF3A4D584B310E
Iterated 1000 times=EF1459E5D55CF902

Set 2, vector# 16:
key=00000000000000000000000000000000
plain=0000800000000000
cipher=EEEF2941208DC802
decrypted=0000800000000000
Iterated 100 times=D6C060FEF7D4BF1D
Iterated 1000 times=39536C90FCDFCA6C

Set 2, vector# 17:
key=00000000000000000000000000000000
plain=0000400000000000
cipher=B4D957A14D83EDD3
decrypted=0000400000000000
Iterated 100 times=10A813AAE122B574
Iterated 1000 times=AE0B294282DC104E

Set 2, vector# 18:
key=00000000000000000000000000000000
plain=0000200000000000
cipher=7E845A40078EDD29
decrypted=0000200000000000
Iterated 100 times=7C83760E5AA79727
Iterated 1000 times=4385B2EAB3DE5753

Set 2, vector# 19:
key=00000000000000000000000000000000
plain=0000100000000000
cipher=543301ED8F7F46F6
decrypted=0000100000000000
Iterated 100 times=57AB4BED7E39C5F6
Iterated 1000 times=57F4768596D3D9C5

Set 2, vector# 20:
key=00000000000000000000000000000000
plain=0000080000000000
cipher=6B1EDA94C6DC880D
decrypted=0000080000000000
Iterated 100 times=A7E0B77ECAAF688BB
Iterated 1000 times=B04B8A1B5382A717

Set 2, vector# 21:
key=00000000000000000000000000000000
plain=0000040000000000
cipher=6C9239CC644CD635
decrypted=0000040000000000
Iterated 100 times=C8390D0A2F5E371B

Iterated 1000 times=3C8842EE9959B688

Set 2, vector# 22:

key=00000000000000000000000000000000
plain=0000020000000000
cipher=5DD833EE8FBC0F71
decrypted=0000020000000000
Iterated 100 times=EFC1A7F9B7A1B3F8
Iterated 1000 times=D51C19BA100003CD

Set 2, vector# 23:

key=00000000000000000000000000000000
plain=0000010000000000
cipher=8B2D8642B630BD02
decrypted=0000010000000000
Iterated 100 times=5A722DD0B535B9D4
Iterated 1000 times=522191DE17A7F30F

Set 2, vector# 24:

key=00000000000000000000000000000000
plain=0000080000000000
cipher=E836116F01A3B804
decrypted=0000080000000000
Iterated 100 times=12161BD6D90DDB62
Iterated 1000 times=111B9AEA0211740D

Set 2, vector# 25:

key=00000000000000000000000000000000
plain=0000040000000000
cipher=61EF722B92517DB4
decrypted=0000040000000000
Iterated 100 times=79C29652C4E085DE
Iterated 1000 times=A305E26554F69310

Set 2, vector# 26:

key=00000000000000000000000000000000
plain=0000020000000000
cipher=3B89889CE806C563
decrypted=0000020000000000
Iterated 100 times=1A9F25371123CBA9
Iterated 1000 times=DF2E65460487E603

Set 2, vector# 27:

key=00000000000000000000000000000000
plain=0000010000000000
cipher=5B4ACD22EB218A7C
decrypted=0000010000000000
Iterated 100 times=331E9D2CFCF54F34
Iterated 1000 times=115E00D91BA3E971

Set 2, vector# 28:

key=00000000000000000000000000000000
plain=0000008000000000
cipher=70F4165B4AB7D496
decrypted=0000008000000000
Iterated 100 times=0DBBF8F509D5079E
Iterated 1000 times=CDEB9EABF52503D3

Set 2, vector# 29:

key=00000000000000000000000000000000
plain=0000004000000000
cipher=002187BB1A7A9244
decrypted=0000004000000000
Iterated 100 times=DE752355804AE318

Iterated 1000 times=76B048C6D46194C3

Set 2, vector# 30:

key=00000000000000000000000000000000
plain=0000000200000000
cipher=F71194BF062FCC14
decrypted=0000000200000000
Iterated 100 times=A15A56242875F588
Iterated 1000 times=1B12D9C5887434EC

Set 2, vector# 31:

key=00000000000000000000000000000000
plain=0000000100000000
cipher=8320CE99E7E79B5C
decrypted=0000000100000000
Iterated 100 times=14D5CFF5CB977284
Iterated 1000 times=22F2454C228D7371

Set 2, vector# 32:

key=00000000000000000000000000000000
plain=0000000080000000
cipher=3F200DF2B8B6798B
decrypted=0000000080000000
Iterated 100 times=300EE089A6C5DB87
Iterated 1000 times=A2088E716824C54C

Set 2, vector# 33:

key=00000000000000000000000000000000
plain=0000000040000000
cipher=E516DC8E3436E7E1
decrypted=0000000040000000
Iterated 100 times=6DA2098D4D8CE56B
Iterated 1000 times=22D1EC19FE79AD5B

Set 2, vector# 34:

key=00000000000000000000000000000000
plain=0000000020000000
cipher=A05460F133AF35DA
decrypted=0000000020000000
Iterated 100 times=436571DCAFCFF9DE
Iterated 1000 times=765E9379616DBBE2

Set 2, vector# 35:

key=00000000000000000000000000000000
plain=0000000010000000
cipher=3267D006F836077B
decrypted=0000000010000000
Iterated 100 times=11B6FE823D53F829
Iterated 1000 times=380FA9C3B8C5A91E

Set 2, vector# 36:

key=00000000000000000000000000000000
plain=0000000008000000
cipher=41890BBFA1ED8CF1
decrypted=0000000008000000
Iterated 100 times=4C681F0DF0C2EFC1
Iterated 1000 times=4BFFEB40C0D4ED86

Set 2, vector# 37:

key=00000000000000000000000000000000
plain=0000000004000000
cipher=8DF64358C50A48EB
decrypted=0000000004000000

Iterated 100 times=B041CEE2786BCB90
Iterated 1000 times=B24D2C48134B35FA

Set 2, vector# 38:

key=00000000000000000000000000000000
plain=0000000002000000
cipher=69DB8C154EA6B8A7
decrypted=0000000002000000
Iterated 100 times=C522102E13D895D7
Iterated 1000 times=E2A939DD0566870D

Set 2, vector# 39:

key=00000000000000000000000000000000
plain=0000000001000000
cipher=7D8A9B8B320B7441
decrypted=0000000001000000
Iterated 100 times=9BF6BB0BFAB3EE0C
Iterated 1000 times=5AF367F576D5B789

Set 2, vector# 40:

key=00000000000000000000000000000000
plain=0000000000800000
cipher=28BF6116E60EC023
decrypted=0000000000800000
Iterated 100 times=114FA11E54B8E80C
Iterated 1000 times=89C0391E5566AC14

Set 2, vector# 41:

key=00000000000000000000000000000000
plain=0000000000400000
cipher=4EC56FA8BBB2DD03
decrypted=0000000000400000
Iterated 100 times=240AB8C6623F7762
Iterated 1000 times=A8ACABDB672AB84B

Set 2, vector# 42:

key=00000000000000000000000000000000
plain=0000000000200000
cipher=B41763577949156A
decrypted=0000000000200000
Iterated 100 times=ED07AA401A27C1AF
Iterated 1000 times=AACEBFE7F3B7B14E

Set 2, vector# 43:

key=00000000000000000000000000000000
plain=0000000000100000
cipher=78DCAE5E3C0B462B
decrypted=0000000000100000
Iterated 100 times=9FDFFFB6CCDF7F4
Iterated 1000 times=B62D00410AE674AE

Set 2, vector# 44:

key=00000000000000000000000000000000
plain=0000000000800000
cipher=21919B5AF343D7D7
decrypted=0000000000800000
Iterated 100 times=84498F6765EB3C57
Iterated 1000 times=25B1EA0F9BDDD8EC

Set 2, vector# 45:

key=00000000000000000000000000000000
plain=0000000000400000
cipher=4098566E1757368D

cipher=03AD2EDC7ED6D8EC
decrypted=000000000000400
Iterated 100 times=C4E7194CE2C62D3C
Iterated 1000 times=CA17B94F3556021A

Set 2, vector# 54:

key=00000000000000000000000000000000
plain=000000000000200
cipher=B93CB83B10A07E21
decrypted=000000000000200
Iterated 100 times=99F6396FBE43173F
Iterated 1000 times=D54CFBAEE01C527E

Set 2, vector# 55:

key=00000000000000000000000000000000
plain=000000000000100
cipher=914B739AFBE71CFA
decrypted=000000000000100
Iterated 100 times=CDDD5CAA1A174C57
Iterated 1000 times=C50F839309D6174D

Set 2, vector# 56:

key=00000000000000000000000000000000
plain=000000000000080
cipher=9980D79CD7B1FAAA
decrypted=000000000000080
Iterated 100 times=440F2F8343F9AAD8
Iterated 1000 times=5A74F2787D257ADD

Set 2, vector# 57:

key=00000000000000000000000000000000
plain=000000000000040
cipher=4C0DC4993FFD7292
decrypted=000000000000040
Iterated 100 times=DFF659D690B8EC97
Iterated 1000 times=BBDB8A09427FF39C

Set 2, vector# 58:

key=00000000000000000000000000000000
plain=000000000000020
cipher=A1DB72C70C18463E
decrypted=000000000000020
Iterated 100 times=1321B3028CC3934C
Iterated 1000 times=9F846C10EBDEA8E6

Set 2, vector# 59:

key=00000000000000000000000000000000
plain=000000000000010
cipher=3809D47A53830B59
decrypted=000000000000010
Iterated 100 times=7DC571F6589CA033
Iterated 1000 times=D2E2E508D179F773

Set 2, vector# 60:

key=00000000000000000000000000000000
plain=000000000000008
cipher=4322C95620D3D5F6
decrypted=000000000000008
Iterated 100 times=CE0DAECCEECBA416
Iterated 1000 times=FEE5869857DA565E

Set 2, vector# 61:

key=00000000000000000000000000000000

plain=0000000000000004
cipher=849F75D2F0DC157E
decrypted=0000000000000004
Iterated 100 times=810C82E879469BC4
Iterated 1000 times=DE38EADEC8D14B2B

Set 2, vector# 62:

key=00000000000000000000000000000000
plain=0000000000000002
cipher=8FA75552F369AFFE
decrypted=0000000000000002
Iterated 100 times=A897CEAAF6EA772A
Iterated 1000 times=F37ABC25B35311DC

Set 2, vector# 63:

key=00000000000000000000000000000000
plain=0000000000000001
cipher=A9DF3D2C64D3EA28
decrypted=0000000000000001
Iterated 100 times=585EBABF4536C41A
Iterated 1000 times=AFBDC0861C3D3D95

Test vectors -- set 3
=====

Set 3, vector# 0:

key=00000000000000000000000000000000
plain=0000000000000000
cipher=2325D00F3E76A22D
decrypted=0000000000000000
Iterated 100 times=495661DAED403F46
Iterated 1000 times=F3BFAA4CEE292DED

Set 3, vector# 1:

key=01010101010101010101010101010101
plain=0101010101010101
cipher=3D666F991262FD70
decrypted=0101010101010101
Iterated 100 times=DEB134D6A6AF0C99
Iterated 1000 times=600DFFCB1FD6251B

Set 3, vector# 2:

key=02020202020202020202020202020202
plain=0202020202020202
cipher=D5B53E4CF8BBA7E4
decrypted=0202020202020202
Iterated 100 times=97837DA2F1127591
Iterated 1000 times=AF89A08AD21C84F4

Set 3, vector# 3:

key=03030303030303030303030303030303
plain=0303030303030303
cipher=9BC7395BF39227D9
decrypted=0303030303030303
Iterated 100 times=748A5BE3954A6847
Iterated 1000 times=BF55BC1F9DAB2BBB

Set 3, vector# 4:

key=04040404040404040404040404040404
plain=0404040404040404
cipher=38EA4668530CBE68
decrypted=0404040404040404
Iterated 100 times=58747A4FD89949F6

decrypted=2C2C2C2C2C2C2C2C
Iterated 100 times=5454058A53BEEBBE
Iterated 1000 times=D9DC071C70008F0E

Set 3, vector# 45:

key=2D2D2D2D2D2D2D2D2D2D2D2D2D2D2D2D
plain=2D2D2D2D2D2D2D2D2D2D
cipher=FDA3E4D68F2A01C7
decrypted=2D2D2D2D2D2D2D2D2D2D
Iterated 100 times=D8A09ACD97AAC7E1
Iterated 1000 times=4A3445AED9FA6D6C

Set 3, vector# 46:

key=2E2E2E2E2E2E2E2E2E2E2E2E2E2E2E2E
plain=2E2E2E2E2E2E2E2E2E2E2E2E
cipher=C7D410F6CB42DAC0
decrypted=2E2E2E2E2E2E2E2E2E2E2E2E
Iterated 100 times=FC56C32D6B207EA3
Iterated 1000 times=D2C02098BAD89049

Set 3, vector# 47:

key=2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F
plain=2F2F2F2F2F2F2F2F2F2F2F2F
cipher=0474F5705016D3B8
decrypted=2F2F2F2F2F2F2F2F2F2F2F2F
Iterated 100 times=B29AA7FE98E97DC1
Iterated 1000 times=B977EB0F6015E72D

Set 3, vector# 48:

key=30303030303030303030303030303030
plain=303030303030303030303030
cipher=17CDB4ABA682218A
decrypted=303030303030303030303030
Iterated 100 times=7F72B7036E6AD9D6
Iterated 1000 times=389046B57D016FCB

Set 3, vector# 49:

key=31313131313131313131313131313131
plain=313131313131313131313131
cipher=975005916E86F0B3
decrypted=313131313131313131313131
Iterated 100 times=9A3F4283D8F2C4A6
Iterated 1000 times=279B536B03A05BBC

Set 3, vector# 50:

key=32323232323232323232323232323232
plain=323232323232323232323232
cipher=0C04F43006FAF3C1
decrypted=323232323232323232323232
Iterated 100 times=C8CED02B13BB40C2
Iterated 1000 times=A4A25435EBF70690

Set 3, vector# 51:

key=33333333333333333333333333333333
plain=333333333333333333333333
cipher=FAAABD4A215CE600
decrypted=333333333333333333333333
Iterated 100 times=1364B73434C75912
Iterated 1000 times=41A831C7CF62E1B2

Set 3, vector# 52:

key=34343434343434343434343434343434
plain=3434343434343434

plain=3C3C3C3C3C3C3C3C
cipher=D32EAAEC89DF954D
decrypted=3C3C3C3C3C3C3C3C
Iterated 100 times=549B7864C80E8118
Iterated 1000 times=0C2E212A5DFE3504

Set 3, vector# 61:

key=3D3D3D3D3D3D3D3D3D3D3D3D3D3D3D3D
plain=3D3D3D3D3D3D3D3D
cipher=D2D32B4E0F7635A2
decrypted=3D3D3D3D3D3D3D3D
Iterated 100 times=641AEFF9B3DE21CF
Iterated 1000 times=C09CC7F126DB5477

Set 3, vector# 62:

key=3E3E3E3E3E3E3E3E3E3E3E3E3E3E3E3E
plain=3E3E3E3E3E3E3E3E
cipher=FEFA25BDE88197AB
decrypted=3E3E3E3E3E3E3E3E
Iterated 100 times=848BC675463F7F22
Iterated 1000 times=954BB252E3A4C71F

Set 3, vector# 63:

key=3F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3F
plain=3F3F3F3F3F3F3F3F
cipher=6CE2FBBD5AF384E9
decrypted=3F3F3F3F3F3F3F3F
Iterated 100 times=3FB8A0AFF4538718
Iterated 1000 times=420A839262DEF08E

Set 3, vector# 64:

key=40404040404040404040404040404040
plain=4040404040404040
cipher=4A8FE3ADA6E9EED3
decrypted=4040404040404040
Iterated 100 times=98F08030EFE397E8
Iterated 1000 times=1FF86B3748BA9952

Set 3, vector# 65:

key=41414141414141414141414141414141
plain=4141414141414141
cipher=BFAD02C644E2393B
decrypted=4141414141414141
Iterated 100 times=E28D01A530B9E277
Iterated 1000 times=104FBEB9B4B10A30

Set 3, vector# 66:

key=42424242424242424242424242424242
plain=4242424242424242
cipher=F2398BAB23277FB9
decrypted=4242424242424242
Iterated 100 times=128DACBB48DC7838
Iterated 1000 times=BA14B9DF9FD51047

Set 3, vector# 67:

key=43434343434343434343434343434343
plain=4343434343434343
cipher=38F71A1C6F3297E3
decrypted=4343434343434343
Iterated 100 times=3E883756826FFCA5
Iterated 1000 times=405C1A3FE6A48256

Set 3, vector# 68:

Set 3, vector# 84:
key=54545454545454545454545454545454
plain=5454545454545454
cipher=E460871A707B82D4
decrypted=5454545454545454
Iterated 100 times=6B6E78A047FFD67A
Iterated 1000 times=4598017C8A6964F2

Set 3, vector# 85:
key=55555555555555555555555555555555
plain=5555555555555555
cipher=4008E5118D9823C3
decrypted=5555555555555555
Iterated 100 times=9FB44DA56961EAA8
Iterated 1000 times=0C56C487D42D579B

Set 3, vector# 86:
key=56565656565656565656565656565656
plain=5656565656565656
cipher=AEF66FDFC6CD1EAE
decrypted=5656565656565656
Iterated 100 times=D9246391191413E8
Iterated 1000 times=F43E6F09D8EFAC40

Set 3, vector# 87:
key=57575757575757575757575757575757
plain=5757575757575757
cipher=9AF81050A1A44971
decrypted=5757575757575757
Iterated 100 times=AA5911A56BDB9852
Iterated 1000 times=BEF28430468BDAC0

Set 3, vector# 88:
key=58585858585858585858585858585858
plain=5858585858585858
cipher=1BC7C4217D844E3D
decrypted=5858585858585858
Iterated 100 times=FBD42A12F65C0BCE
Iterated 1000 times=5A9DE0243133289E

Set 3, vector# 89:
key=59595959595959595959595959595959
plain=5959595959595959
cipher=4AA68ECFFF9FBBC4
decrypted=5959595959595959
Iterated 100 times=F7B2AD0D39C0EE57
Iterated 1000 times=183F6114B04EEC5C

Set 3, vector# 90:
key=5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A5A
plain=5A5A5A5A5A5A5A5A
cipher=49EF1787726C0409
decrypted=5A5A5A5A5A5A5A5A
Iterated 100 times=5264A786A6395E58
Iterated 1000 times=175A33543A87291E

Set 3, vector# 91:
key=5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B
plain=5B5B5B5B5B5B5B5B
cipher=61309D25084A3900
decrypted=5B5B5B5B5B5B5B5B
Iterated 100 times=33CE82BFBBF59F3B

cipher=CB4EFF9EEFFB3A7D
decrypted=7373737373737373
Iterated 100 times=D88663668133EEA9
Iterated 1000 times=530E0ABF31307FE4

Set 3, vector#116:

key=74747474747474747474747474747474
plain=7474747474747474
cipher=A1AC364DAB38DEA4
decrypted=7474747474747474
Iterated 100 times=60C8584810C20D7C
Iterated 1000 times=B544BEF81EDBD854

Set 3, vector#117:

key=75757575757575757575757575757575
plain=7575757575757575
cipher=4C3A89DF30C6F614
decrypted=7575757575757575
Iterated 100 times=78D652FC0E4EB625
Iterated 1000 times=CC64C0A40E97AB17

Set 3, vector#118:

key=76767676767676767676767676767676
plain=7676767676767676
cipher=CA5B4FCD84A0199A
decrypted=7676767676767676
Iterated 100 times=1A8510BE03DA3312
Iterated 1000 times=8D33C786C277AE0F

Set 3, vector#119:

key=77777777777777777777777777777777
plain=7777777777777777
cipher=91C5785615F50383
decrypted=7777777777777777
Iterated 100 times=313F6CC183EF3930
Iterated 1000 times=2CD4491333B66A5A

Set 3, vector#120:

key=78787878787878787878787878787878
plain=7878787878787878
cipher=2346210D6A6CA0D1
decrypted=7878787878787878
Iterated 100 times=E410055E7A4867EF
Iterated 1000 times=A34D8049D6C471F8

Set 3, vector#121:

key=79797979797979797979797979797979
plain=7979797979797979
cipher=FADCA44633356AF2
decrypted=7979797979797979
Iterated 100 times=CD61A856747B2522
Iterated 1000 times=4075F54ECE377579

Set 3, vector#122:

key=7A7A7A7A7A7A7A7A7A7A7A7A7A7A7A7A
plain=7A7A7A7A7A7A7A7A
cipher=4E2E9CE9C1D4E232
decrypted=7A7A7A7A7A7A7A7A
Iterated 100 times=0DBF888EF4B86288
Iterated 1000 times=0E99D14E3FE41B47

Set 3, vector#123:

key=7B7B7B7B7B7B7B7B7B7B7B7B7B7B7B7B

plain=7B7B7B7B7B7B7B7B
cipher=49633804FCB7C19D
decrypted=7B7B7B7B7B7B7B7B
Iterated 100 times=AFC4E4229E633EBA
Iterated 1000 times=803EEDAA9129563D

Set 3, vector#124:

key=7C7C7C7C7C7C7C7C7C7C7C7C7C7C7C7C
plain=7C7C7C7C7C7C7C7C
cipher=2FEAC54B97CB0655
decrypted=7C7C7C7C7C7C7C7C
Iterated 100 times=A73AEA21A3F54B49
Iterated 1000 times=D4D7970410540B74

Set 3, vector#125:

key=7D7D7D7D7D7D7D7D7D7D7D7D7D7D7D7D
plain=7D7D7D7D7D7D7D7D
cipher=7AF7DCF2409425EF
decrypted=7D7D7D7D7D7D7D7D
Iterated 100 times=0346FBF0811E11D6
Iterated 1000 times=A1B64C466276887B

Set 3, vector#126:

key=7E7E7E7E7E7E7E7E7E7E7E7E7E7E7E7E
plain=7E7E7E7E7E7E7E7E
cipher=883BCADE712D4E13
decrypted=7E7E7E7E7E7E7E7E
Iterated 100 times=D0DAF00A787F90BF
Iterated 1000 times=9DD483C1D756E2A6

Set 3, vector#127:

key=7F7F7F7F7F7F7F7F7F7F7F7F7F7F7F7F
plain=7F7F7F7F7F7F7F7F
cipher=DDF9350F57E1721C
decrypted=7F7F7F7F7F7F7F7F
Iterated 100 times=C28656590A37EAA0
Iterated 1000 times=AE78E93E0425F348

Set 3, vector#128:

key=80808080808080808080808080808080
plain=8080808080808080
cipher=221300E434D6A8CA
decrypted=8080808080808080
Iterated 100 times=15AE60FF0D0A9BED
Iterated 1000 times=5F2F08C1DA027C08

Set 3, vector#129:

key=81818181818181818181818181818181
plain=8181818181818181
cipher=2E015CD0ACD03613
decrypted=8181818181818181
Iterated 100 times=66A8BC059CE66FF0
Iterated 1000 times=760CDCD2BDB6D8B5

Set 3, vector#130:

key=82828282828282828282828282828282
plain=8282828282828282
cipher=83E5D042544FCA78
decrypted=8282828282828282
Iterated 100 times=420143E21AABC302
Iterated 1000 times=FD0D2147D6992E3E

Set 3, vector#131:

cipher=14E0B5400A3C6731
decrypted=B2B2B2B2B2B2B2B2
Iterated 100 times=DDB1CE9DE086F780
Iterated 1000 times=C69A537E629378D0

Set 3, vector#179:

key=B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3B3
plain=B3B3B3B3B3B3B3B3
cipher=6382C9FFA803621C
decrypted=B3B3B3B3B3B3B3B3
Iterated 100 times=FBA7F82C4E8AB66E
Iterated 1000 times=A0D950F1F7FD325A

Set 3, vector#180:

key=B4B4B4B4B4B4B4B4B4B4B4B4B4B4B4B4
plain=B4B4B4B4B4B4B4B4
cipher=153745AE5A35B986
decrypted=B4B4B4B4B4B4B4B4
Iterated 100 times=C31453BF8C279E0C
Iterated 1000 times=F4ED2171BA5D3C00

Set 3, vector#181:

key=B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5B5
plain=B5B5B5B5B5B5B5B5
cipher=00007C2B1C29B162
decrypted=B5B5B5B5B5B5B5B5
Iterated 100 times=6F0BF1DF76D11F96
Iterated 1000 times=5E195DEFDE08A7D0

Set 3, vector#182:

key=B6B6B6B6B6B6B6B6B6B6B6B6B6B6B6B6
plain=B6B6B6B6B6B6B6B6
cipher=3F7DD39702F26434
decrypted=B6B6B6B6B6B6B6B6
Iterated 100 times=1B2843ADA1F9CFA9
Iterated 1000 times=FBC011ED693A7627

Set 3, vector#183:

key=B7B7B7B7B7B7B7B7B7B7B7B7B7B7B7B7
plain=B7B7B7B7B7B7B7B7
cipher=D332D2A68FF5CAB9
decrypted=B7B7B7B7B7B7B7B7
Iterated 100 times=7A58B623E90D3DA8
Iterated 1000 times=3934B052B98BD1D5

Set 3, vector#184:

key=B8B8B8B8B8B8B8B8B8B8B8B8B8B8B8B8
plain=B8B8B8B8B8B8B8B8
cipher=711DB3ECF160D922
decrypted=B8B8B8B8B8B8B8B8
Iterated 100 times=E02C2C5F2C35FF12
Iterated 1000 times=5A9DF6B76A3F51E8

Set 3, vector#185:

key=B9B9B9B9B9B9B9B9B9B9B9B9B9B9B9B9
plain=B9B9B9B9B9B9B9B9
cipher=F01BA49EE20499DC
decrypted=B9B9B9B9B9B9B9B9
Iterated 100 times=0C92DC3123264B64
Iterated 1000 times=35D09C954BDF381A

Set 3, vector#186:

key=BABABABABABABABABABABABABABABABABA

Set 3, vector#202:
key=CACACACACACACACACACACACACACACACACA
plain=CACACACACACACACA
cipher=4C5F0D68A1CAAC80
decrypted=CACACACACACACACA
Iterated 100 times=C56D75960D6C0ACF
Iterated 1000 times=E7BF996B34E7C713

Set 3, vector#203:
key=CBCBCBCBCBCBCBCBCBCBCBCBCBCBCBCBCB
plain=CBCBCBCBCBCBCBCB
cipher=C8A9A23970A8E32A
decrypted=CBCBCBCBCBCBCBCB
Iterated 100 times=48D1C1A3E4F0EEB5
Iterated 1000 times=FC14174D7838E4E5

Set 3, vector#204:
key=CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
plain=CCCCCCCCCCCCCCCCC
cipher=3B4ABB7DDD1088EC
decrypted=CCCCCCCCCCCCCCCCC
Iterated 100 times=C155C105CC468F10
Iterated 1000 times=B107E08C2A424A4E

Set 3, vector#205:
key=CDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCDCD
plain=CDCDCDCDCDCDCDCD
cipher=0884EC3AA733C44F
decrypted=CDCDCDCDCDCDCDCD
Iterated 100 times=D761D3A7CF7BB7AB
Iterated 1000 times=D64C41B91F9EFCC5

Set 3, vector#206:
key=CECECECECECECECECECECECECECECECECE
plain=CECECECECECECECE
cipher=239CEAC173866ABE
decrypted=CECECECECECECECE
Iterated 100 times=7DBE69E9A7A1353B
Iterated 1000 times=62F104355281B3C3

Set 3, vector#207:
key=CFCFCFCFCFCFCFCFCFCFCFCFCFCFCFCF
plain=CFCFCFCFCFCFCFCF
cipher=CCDEA8AEF37CF286
decrypted=CFCFCFCFCFCFCFCF
Iterated 100 times=FB60BCE4E1A00D37
Iterated 1000 times=11017CB4A39D4C15

Set 3, vector#208:
key=D0D0D0D0D0D0D0D0D0D0D0D0D0D0D0D0D0
plain=D0D0D0D0D0D0D0D0
cipher=E0437C6AE2A1CF40
decrypted=D0D0D0D0D0D0D0D0
Iterated 100 times=428181BFAD4C6986
Iterated 1000 times=D41E9D21C3E96A59

Set 3, vector#209:
key=D1D1D1D1D1D1D1D1D1D1D1D1D1D1D1D1D1
plain=D1D1D1D1D1D1D1D1
cipher=2D60B139BB35C3BE
decrypted=D1D1D1D1D1D1D1D1
Iterated 100 times=1C97DA897A4AFB04
Iterated 1000 times=BA9C505A4E5CEE77

cipher=D57DA95FA2399691
decrypted=F1F1F1F1F1F1F1F1
Iterated 100 times=702F4E1305161F7A
Iterated 1000 times=4D3D5576F0B17B03

Set 3, vector#242:

key=F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2
plain=F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2
cipher=F60586962FBAEDFE
decrypted=F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2F2
Iterated 100 times=D7292F899A3C5560
Iterated 1000 times=34A0ACA58248F21D

Set 3, vector#243:

key=F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3
plain=F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3
cipher=3F2D63BEA388BA83
decrypted=F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3F3
Iterated 100 times=C5A70BF43DF55B5E
Iterated 1000 times=E1F6AF1A2A3D6935

Set 3, vector#244:

key=F4F4F4F4F4F4F4F4F4F4F4F4F4F4F4F4
plain=F4F4F4F4F4F4F4F4F4F4F4F4F4F4F4F4
cipher=C441931CE7907427
decrypted=F4F4F4F4F4F4F4F4F4F4F4F4F4F4F4F4
Iterated 100 times=C1BF731D86FF2987
Iterated 1000 times=575422BB7A89D536

Set 3, vector#245:

key=F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5
plain=F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5
cipher=44B92C8ADF5918A4
decrypted=F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5F5
Iterated 100 times=D99D7CD558F106F2
Iterated 1000 times=F7F5600B86CB6E63

Set 3, vector#246:

key=F6F6F6F6F6F6F6F6F6F6F6F6F6F6F6F6
plain=F6F6F6F6F6F6F6F6F6F6F6F6F6F6F6F6
cipher=40E9EF286DEFC6BB
decrypted=F6F6F6F6F6F6F6F6F6F6F6F6F6F6F6F6
Iterated 100 times=22E987BB1E15A700
Iterated 1000 times=A7CA5C6D897CDA78

Set 3, vector#247:

key=F7F7F7F7F7F7F7F7F7F7F7F7F7F7F7F7
plain=F7F7F7F7F7F7F7F7F7F7F7F7F7F7F7F7
cipher=56116167CBCFE07F
decrypted=F7F7F7F7F7F7F7F7F7F7F7F7F7F7F7F7
Iterated 100 times=BC603B61DF9210DF
Iterated 1000 times=D281C1543F50DC81

Set 3, vector#248:

key=F8F8F8F8F8F8F8F8F8F8F8F8F8F8F8F8
plain=F8F8F8F8F8F8F8F8F8F8F8F8F8F8F8F8
cipher=CB3FD8CEA0C1B700
decrypted=F8F8F8F8F8F8F8F8F8F8F8F8F8F8F8F8
Iterated 100 times=21087F19A5F2AE63
Iterated 1000 times=BA86ECD86B61809B

Set 3, vector#249:

key=F9F9F9F9F9F9F9F9F9F9F9F9F9F9F9F9

Intellectual Property Statement

1. Statement by the Submitters

We, PAULO SÉRGIO L. M. BARRETO and VINCENT RIJMEN, do hereby declare that to the best of our knowledge the practice of the KHAZAD algorithm, as well as the reference implementations we have submitted, are not covered by any patents or patent applications worldwide.

We do hereby agree to provide the statements for any patent or patent application identified to cover practice of the KHAZAD algorithm or its reference implementations and the right to use such implementations for the purposes of the NESSIE evaluation process.

We do hereby understand that our submitted algorithm may not be selected by the NESSIE project. We also understand and agree that after the close of the submission period our submission may not be withdrawn from public consideration for the goals of the NESSIE project.

Should our submission be selected by the NESSIE effort, we hereby agree not to place any restrictions on the use of the algorithm intending it to be available on a worldwide, non-exclusive, royalty-free basis.

2. Permission and Authors' Copyright Release

We, PAULO SÉRGIO L. M. BARRETO and VINCENT RIJMEN, do hereby grant the NESSIE project the non-exclusive right to reproduce or to have reproduced, prepare or have prepared in derivative form, and distribute or have distributed copies of all materials included in our submission to the NESSIE effort.

We also represent that the exercise of these rights by the NESSIE project will not infringe or otherwise violate any rights of another person or organisation.

Agreed and accepted,

Paulo Sérgio L. M. Barreto
Cryptographer
Scopus Tecnologia S.A.

Vincent Rijmen
Cryptographer
Cryptomathic NV