L-023

CRYPTREC 推奨暗号の 64 ビット環境下の高速実装

A Fast implementation on the 64 bits environment of the suggested cipher in CRYPTREC

朝日 康介* 増田 豊史† 西川 栄光 † 横山 祐人 † 五十嵐 保隆 *
Kousuke Asahi Toyoshi Masuda Eikou Nishikawa Yuuto Yokoyama Yasutaka Igarashi
金子 敏信 *
Toshinobu Kaneko

1 はじめに

CRYPTREC 推奨暗号が 2003 年に制定されて以来，7 年が経過した．2001 年の CRYPTREC 報告書 [1] で，ソフトウエア実装評価がなされているが，その当時の標準の計算機環境は，32 ビット CPU であったが，現在は 64 ビット CPU が中心となっている．このような環境変化に鑑み，本稿では 128 ビット共通鍵ブロック暗号を中心に選び，64 ビット CPU 環境下での高速実装を意識した性能比較を行った．対象とした暗号は，ブロック暗号として Camellia[2]，CIPHERUNICORN-A[3]，Hierocrypt-3[4]，SC2000[5]，ストリーム暗号として MUGI[6] である．2001 年の実装性能評価においては，暗号提案各社の技術陣が競い，アセンブラ実装まで実施した会社もあった．ここでは，高速実装の容易性の判断材料を供給すべく，本研究室の卒業研究生レベルのプログラミング能力を持つ複数の学生が，C 言語で実装を行った．実装者の能力の同一性は，必ずしも保証出来ないので，絶対的比較はできないが，定性的な判断材料となるであろう．本稿第 2.1 節では，全体に共通する高速実装の考え方を説明し，第 2.2 節以降で各暗号を取り扱い，第 3 節以降で実装環境と実装結果を示す．具体的高速実装法の説明においては，実装のポイントのみを示している．各暗号アルゴリズムの説明は，各々のアルゴリズム仕様書を参照していただきたい．

2 高速化手法

2.1 高速化手法の基本

2.1.1 インライン展開

関数は呼び出し毎にオーバーヘッドが生じる．ループはループ毎に終了判定が行われる．そのため，関数・ループをインライン展開することにより，オーバーヘッドがなくなり，高速化を図ることが可能である．しかし，プログラムのサイズが CPU のキャッシュサイズを超えると逆に遅くなる場合があるので注意が必要である．

2.1.2 論理式のテーブル参照化

論理式の計算結果を全パターンについて計算し，それをテーブルとして作成しておき，論理式を計算する代わりにこのテーブルを参照することにより，高速化を図ることが可能である．また，同様にテーブルを複数回参照する操作をテーブルを結合することによってテーブル参照を減らし，高速化を図ることが可能である．ただし，論理式が単純な場合や，テーブルのサイズが CPU のキャッシュサイズを超えた場合は逆に遅くなる場合があるので注意が必要である．なお，n ビット入力 m ビット出力のテーブルサイズは $2^n \times m$ ビットとなる．

2.1.3 OS のビット数に対する最適化

32 ビット OS において，処理単位を 8 ビットで実装した場合と比べ，32 ビットで実装した場合は 1〜4 倍の高速化を図ることが可能である．ただし，処理単位が OS のビット数より大きい場合，あまり高速化しないかむしろ遅くなることがある．逆に，64 ビットの Windows において 32 ビットのプログラムを実行した場合，エミュレーションレイヤサブシステムによって 32 ビットエミュレーションモードで実行されるため，わずかだが処理速度が低下する．OS のビットサイズを意識した実装が肝要である．

2.2 Camellia

2.2.1 暗号化関数

Camellia は，秘密鍵長 128 or 192 or 256 ビットをサポートする 128 ビットブロック暗号である．データランダム化部は段関数 F を用いた Feistel 構造と補助関数 FL 及び FL^{-1} で構成される．段数は，128 ビット鍵の場合 18 段，192 or 256 ビット鍵の場合 24 段である．FL 及び FL^{-1} は，6 段毎に挿入され，AND,OR,XOR, ローテートシフトで構成される拡大鍵依存の線形変換である．F 関数は，拡大鍵 XOR 加算，8 ビット S-box，バイト単位の XOR 及びバイトスワップで構成される．鍵スケジュール部は，F 関数を用いた Feistel 似の構造を持ち，秘密鍵から 4 つの 128 ビット関連鍵 K_L，K_R，K_A，K_B を生成する．データランダム化部の各ラウンドで使用される 64 ビットの拡大鍵は，これら関連鍵をローテートシフトし，所定の 64 ビット部分を取りだ出したものである．

2.2.2 32 ビット OS・64 ビット OS を対象とした高速化

2.1.3 節での手法を用い，32 ビット OS および 64 ビット OS を対象として処理単位を 32 ビットおよび 64 ビットで実装

* 東京理科大学 理工学研究科電気工学専攻
† 東京理科大学 理工学部電気電子情報学科

した．これにより，2.2.4 節の拡大鍵生成の一部の省略が可能になる．さらに，NTT による最適化コード [8] は入出力を 8 ビット単位で受け取り，内部で 32 ビットに変換して処理をしているが，処理速度の向上のために入出力を 32 ビットおよび 64 ビット単位で受け取るようにした．

2.2.3 インライン展開

2.1.1 節での手法を用い，鍵スケジュール関数・暗号化関数・復号関数の内部について全ての関数・ループをインライン展開した．

2.2.4 拡大鍵生成の一部の省略

仕様書 [2]C.1.5 節の手法の範囲を広げた手法を用いて高速化を行う．

■32 ビットプラットフォームにおける実装

拡大鍵を 32 ビット単位で見ると，重複している部分がある．

128 ビット鍵の場合

$$(kl_{3L}, kl_{3R}, kl_{4L}, kl_{4R}) = (k_{7R}, k_{8L}, k_{8R}, k_{7L})$$
$$(k_{15L}, k_{15R}, k_{16L}, k_{16R}) = (kl_{2L}, kl_{2R}, kl_{1L}, kl_{1R})$$
$$(k_{17L}, k_{17R}, k_{18L}, k_{18R}) = (k_{4R}, k_{3L}, k_{3R}, k_{4L})$$
$$(kw_{3L}, kw_{3R}, kw_{4L}, kw_{4R}) = (k_{6R}, k_{5L}, k_{5R}, k_{6L})$$

の計 16 個の拡大鍵が重複しているため，16 個の拡大鍵生成および 32 ビット×16 = 512 ビット 文のメモリが削減できる．

192・256 ビット鍵の場合

$$(k_{17L}, k_{17R}, k_{18L}, k_{18R}) = (k_{9R}, k_{10L}, k_{10R}, k_{9L})$$
$$(kl_{5L}, kl_{5R}, kl_{6L}, kl_{6R}) = (k_{11R}, k_{12L}, k_{12R}, k_{11L})$$
$$(k_{19L}, k_{19R}, k_{20L}, k_{20R}) = (kl_{2L}, kl_{2R}, kl_{1L}, kl_{1R})$$

の計 12 個の拡大鍵が重複しているため，12 個の拡大鍵生成および 32 ビット×12 = 384 ビット 分のメモリが削減できる．ここで，各拡大鍵の表記は，技術仕様書 [2] の表 2, 3 に従い，添え字 $_L$ 及び $_R$ は，その上位又は下位 32 ビットを表わす．

■64 ビットプラットフォームにおける実装

拡大鍵を 64 ビット単位で見ると重複している部分がある．

128 ビット鍵の場合

$$(k_{15}, k_{16}) = (kl_2, kl_1)$$

の計 2 個の拡大鍵が重複しているため，2 個の拡大鍵生成および 64 ビット×2 = 128 ビット 文のメモリが削減できる．

192・256 ビット鍵の場合

$$(k_{19}, k_{20}) = (kl_2, kl_1)$$

の計 2 個の拡大鍵が重複しているため，2 個の拡大鍵生成および 64 ビット×2 = 128 ビット 分のメモリが削減できる．

2.2.5 高速処理のための関連鍵生成部の最適化

仕様書 [2]C.1.3 節の手法を用いて高速化を行う．これにより, 排他的論理和の回数が減少し, 高速化を図ることができる．

2.2.6 高速処理のための F 関数の等価変形

仕様書 [2]C.2.7 節の手法を用いて高速化を行う．

■64 ビット OS　同節の式 (3) の手法を用いる．

■32 ビット OS　同節の式 (5) の手法を用いる．

2.3 CIPHERUNICORN-A

2.3.1 暗号化関数

CIPHERUNICORN-A は，秘密鍵長 128 or 192 or 256 ビットをサポートする 128 ビットブロック暗号である [3]．データランダム化部は 64 ビット入力，64 ビット出力の段関数 F を用いた 16 段 Feistel 構造を持つ．段関数 F の内部は，32 ビットデータ線 2 本の 10 段 Feistel 構造で表現されており，T_n 関数，A3 関数，定数乗算，拡大鍵算術加算で構成される．T_n 関数は，テーブルで表現される非線形関数であり，A3 関数は，ローテートシフトと XOR 演算で表わされる線形関数である．鍵スケジュール部は 128 or 192 or 256 ビットの秘密鍵を技術仕様書 [3] の 3.8 節の記述に従い，拡大鍵に変換する．秘密鍵長によって段数が変化するダミーループ後，拡大鍵を生成する．

2.3.2 インライン展開

2.1.1 節での手法を用い，暗号化関数・復号関数の内部について全ての関数・ループをインライン展開した．

2.3.3 T_n 関数の 32 ビット出力テーブル使用による高速化

T_n 関数は，32 ビットデータの内の n で指定されるバイト位置のデータで，4 種類の 8 ビット入出力 S-box テーブル (S_0〜S_3) を引き，32 ビットの出力データを得る関数である．ここでは，これを，8 ビット入力，32 ビット出力の単一テーブル T_n として実装し，テーブル参照回数を 4 回から 1 回に減少させて，高速化を計った．(図 1)

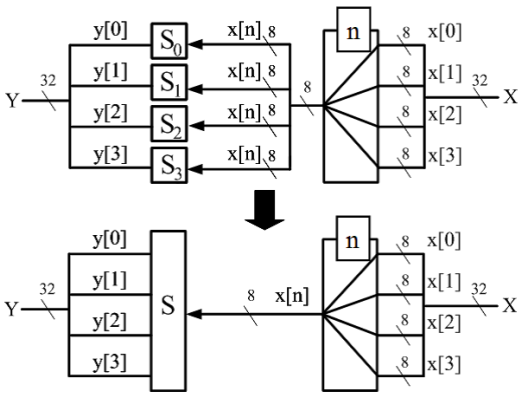

図 1　8bit 入出力テーブル (上) から 8bit 入力 32bit 出力テーブルへ (下)

2.4 Hierocrypt-3

2.4.1 暗号化関数

Hierocrypt-3 のデータランダム化部は，技術仕様書 [4]3.2.1 節に記述されている．16 バイトの平文は，16 or 24 or 32 バイトの秘密鍵から鍵スケジュール部で生成された拡大鍵を使い暗号化される．データランダム化部の構造を図 2 に示す．段関数 ρ は，hcrypt3_xs 及び，HC3_MDSH_ST から構成される．図 2 の XS は，段関数 ρ から MDSH 変換のみを除いた

構造であり，AK は，最終の拡大鍵 XOR 加算である．高速実装にあたっては，hcrypt3_xs を 4 種類の mdsL_ST テーブル，HC3_MDSH_ST を 7 種類の MDSH_ST テーブルを使い実現している．

2.4.2 実装方式

参考文献 [10] の手法を用いているため，詳細な説明については，参考文献 [10] を参照されたい．

2.4.3 hcrypt3_xsS 関数部の高速化

hcrypt3_xs は，技術仕様書 [4]3.3.2〜3.3.6 節の記述に従った関数である．図 2 のように，hcrypt3_xs は，32 ビット (4 バイト) 毎に区切る事が出来，各々は，4 個の S-box (S8) と線形変換 mds_L で構成されている．高速化を図るために，この 4 バイトの処理を，mdsL_ST テーブルとして実装している．S-box と mdsL 関数の合成関数の計算は，次のように行った．mdsL 関数は，入出力長 4 byte の線形拡散層であり，4 バイト入力 $\mathbf{Y} = (y_1, y_2, y_3, y_4)^T$ を 4 行 4 列の MDS 行列を用いて変換し，4 バイト出力 $\mathbf{Z} = (z_1, z_2, z_3, z_4)^T$ を得る．

$$\mathbf{Z} = \begin{pmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{pmatrix} = \begin{pmatrix} 0xc4 & 0x65 & 0xc8 & 0x8b \\ 0x8b & 0xc4 & 0x65 & 0xc8 \\ 0xc8 & 0x8b & 0xc4 & 0x65 \\ 0x65 & 0xc8 & 0x8b & 0xc4 \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix}$$

ここで，y_i は $GF(2^8)$ の元である．

出力 $\mathbf{Z}$ は，入力 $\mathbf{Y}$ の各バイトデータ $y_i (1 \le i \le 4)$ に対する列の積和として与えられる．S 層の 4 バイト入力を $\mathbf{X} = (x_1, x_2, x_3, x_4)^T$ とすれば，各 y_i は，対応する S-box 出力 $S8(x_i)$ であり，出力 $\mathbf{Z}$ は以下の式となる．

$$\mathbf{Z} = \mathbf{Z_1} + \mathbf{Z_2} + \mathbf{Z_3} + \mathbf{Z_4}$$

ただし，$\mathbf{Z_i} (i = 1, 2, 3, 4)$ は，以下の式である．

$$\mathbf{Z_1} = \begin{pmatrix} 0xc4 \\ 0x65 \\ 0xc8 \\ 0x8b \end{pmatrix} S8(x_1), \mathbf{Z_2} = \begin{pmatrix} 0x8b \\ 0xc4 \\ 0x65 \\ 0xc8 \end{pmatrix} S8(x_2),$$

$$\mathbf{Z_3} = \begin{pmatrix} 0xc8 \\ 0x8b \\ 0xc4 \\ 0x65 \end{pmatrix} S8(x_3), \mathbf{Z_4} = \begin{pmatrix} 0x65 \\ 0xc8 \\ 0x8b \\ 0xc4 \end{pmatrix} S8(x_4)$$

したがって，x_i に対して $\mathbf{Z_i}$ を出力するテーブルを，$i = 1, 2, 3, 4$ に対し作成すればよい．このテーブルは，8 ビット入力，32 ビット出力のテーブルである．このテーブルを mdsL_ST テーブルと呼ぶ．hcrypt3_xs 関数は，このテーブルを 16 回引くことで実現できる．

2.4.4 MDSH_ST テーブル

mdsL_ST テーブルと同様に HC3_MDSH_ST テーブルも実装できる．テーブル量を削減する工夫を以下の様に行った．

MDSH 関数は，入出力長 16 バイトの線形拡散層として，hcrypt3_xs 関数出力 $\mathbf{X}$ の 16 バイトを入力と見なし，16 行 16 列の MDS 行列で線形変換を行う関数である．出力 $\mathbf{Y}$ の 16 バイトは，$\mathbf{X}$ の各バイト $x_i (1 \le i \le 16)$ と列ベクトルの積和として与えられる．ここで，x_i は $GF(2^8)$ の元である．この式を式 1 に示す．

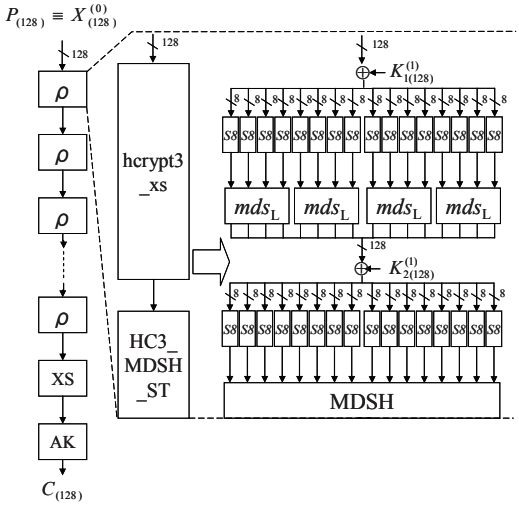

図 2　Hierocrypt-3 データランダム化部

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \\ y_{10} \\ y_{11} \\ y_{12} \\ y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \end{pmatrix} = \begin{pmatrix} 1\,0\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,1\,1\,1\,1 \\ 1\,1\,0\,1\,1\,1\,0\,1\,1\,1\,1\,0\,0\,1\,1\,1 \\ 1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,1\,1\,0\,0\,1\,1 \\ 0\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,0\,1\,1\,1\,0 \\ 1\,1\,1\,1\,1\,0\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1 \\ 0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,1\,1\,0 \\ 0\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,0\,1\,1\,1\,1 \\ 1\,1\,1\,0\,0\,1\,0\,1\,0\,1\,0\,1\,1\,0\,1\,0 \\ 1\,1\,0\,1\,1\,1\,1\,1\,1\,0\,1\,0\,1\,0\,1\,0 \\ 1\,1\,1\,0\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,0\,1 \\ 1\,1\,1\,1\,0\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,0 \\ 1\,0\,1\,0\,1\,1\,1\,0\,0\,1\,0\,1\,0\,1\,0\,1 \\ 1\,0\,1\,0\,1\,1\,0\,1\,1\,1\,1\,1\,1\,1\,0\,1\,0 \\ 1\,1\,0\,1\,1\,1\,1\,1\,0\,0\,1\,1\,1\,1\,1\,0\,1 \\ 1\,1\,1\,0\,1\,1\,1\,1\,0\,0\,1\,1\,1\,1\,1\,1\,0 \\ 0\,1\,0\,1\,1\,0\,1\,0\,1\,1\,1\,0\,0\,1\,0\,1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \\ x_{10} \\ x_{11} \\ x_{12} \\ x_{13} \\ x_{14} \\ x_{15} \\ x_{16} \end{pmatrix} \quad (1)$$

これをそのままテーブル化すると，8 ビット入力 128 ビット出力のテーブル 16 種類で実装できる．

このままでは，テーブルサイズが大きいので，入力を 1 バイト，出力を 4 バイト毎に区切って表わせば次式となる．

$$\begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} x_1 + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} x_2 + \cdots + \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} x_{15} + \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} x_{16} \quad (2)$$

$$\begin{pmatrix} y_5 \\ y_6 \\ y_7 \\ y_8 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} x_1 + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} x_2 + \cdots + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} x_{15} + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} x_{16} \quad (3)$$

$$\begin{pmatrix} y_9 \\ y_{10} \\ y_{11} \\ y_{12} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} x_1 + \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} x_2 + \cdots + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} x_{15} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} x_{16} \quad (4)$$

$$\begin{pmatrix} y_{13} \\ y_{14} \\ y_{15} \\ y_{16} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \end{pmatrix} x_1 + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \end{pmatrix} x_2 + \cdots + \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} x_{15} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} x_{16} \qquad (5)$$

式 (2) から式 (5) までには計 64 個の 4 バイト列ベクトルが存在するが，これらには重複があるため，実際に異なる列ベクトルは 7 種類のみである．従って，テーブルは，7 種類で十分である．これを，S-box と結合し，8 ビット入力，32 ビット出力の MDSH_ST テーブルとして実装している．

2.4.5 鍵スケジュール

技術仕様書 [4]3.2.3 節 (鍵スケジューリング) の記述に従い実装する．中間鍵の更新用関数である σ 関数（技術仕様書 [4]3.2.6 節と 3.2.7 節の平文側）と σ^{-1} 関数（技術仕様書 [4]3.2.6 節と 3.2.7 節の暗号文側）で構成される．ただし，M_{B3},M_{5E},F σ,$P^{(n)}$（技術仕様書 [4]3.3.8 節〜3.3.11 節）は，呼出し型の関数ではなく，σ，σ^{-1} 関数内で M_{B3},M_{5E},F σ,$P^{(n)}$ 関数を展開して実装し高速化を図っている．

2.5 SC2000

2.5.1 暗号化関数

技術仕様書 [5]3.1 節に記述されている関数であり，16 バイトの平文は，16 or 24 or 32 バイトの秘密鍵から鍵スケジュール部で生成された拡大鍵を使い暗号化される．データランダム化部の構造を図 3 に示す．

データランダム化部は，*I* 関数，*B* 関数，*R* 関数から構成される．

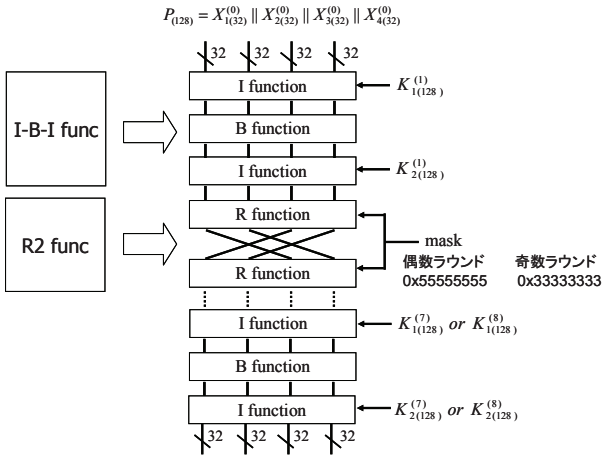

図 3　SC2000 データランダム化部

2.5.2 実装方式

参考文献 [10] の手法を用いているため，詳細な説明については，参考文献 [10] を参照されたい．

2.5.3 関数の合成

実装にあたって，高速化を図るため，*I* 関数，*B* 関数を合成し，$I-B-I$ 関数とし，*R* 関数 2 段も 1 つの関数へ合成し，*R*2 関数として実装を行っている．

技術仕様書 [5]3.2.1 節の記述に従った関数である．

2.5.4 *B* 関数へのビットスライス処理の適用

B 関数は，技術仕様書 [5]6.2 節にあるビットスライス処理による高速化を図っている．

2.5.5 *R*2 関数へのテーブル結合と関数結合の適用

*R*2 関数は，技術仕様書 [5]3.4 節に定義している *R* 関数（*S* 関数，*M* 関数，*L* 関数から構成)2 段で構成している．

ここでは，技術仕様書 [5]6.1.1（テーブルの結合) と 6.1.2 (関数の結合) による高速化を図っている．

SC2000 のアルゴリズム提案では，技術仕様書 [5] で S 関数は，ビットとビットの S-Box テーブルを $(6, 5, 5, 5, 5, 6)$ という形で索引している．

本実装では，隣接する S-Box を 1 つの S-Box と見なして結合し $(6 + 5, 5 + 5, 5 + 6) = (11, 10, 11)$ の形でテーブル索引を行うことで，索引回数を削減している．

関数の結合では，*S* 関数と *M* 関数はどちらもテーブル索引を行う関数なので，この 2 つの関数を結合処理し，テーブル *S*11_*M*(11 ビット入力 32 ビット出力) と *S*10_*M*(10 ビット入力 32 ビット 出力) を作成することで，テーブルの索引回数を削減し，高速化を行った．

2.6 MUGI

2.6.1 疑似乱数生成器

MUGI は，ストリーム暗号であり，その疑似乱数生成器は 128 ビットの秘密鍵と 128 ビットの初期値を使い，ビット単位に乱数系列を発生する PANAMA 型構造を持つ [6]．内部構造は，ビット×16 のバッファ部と ビット×3 のステート部からなり，内部状態は，それぞれ，線形関数 λ，非線形関数 ρ で状態推移する．非線形関数 ρ は，非線形関数 F を 2 つ使って構成される．

2.6.2 F 関数のテーブル参照化

技術仕様書 [6](4.7.3 節) から関数 F は S-box と MDS 行列による行列変換 M, さらにバイト置換から成る．ここでは，高速化のため S-box と行列変換 M をひとまとめにした T_n テーブル (n=0,1,2,3) を作成し実装している．テーブル参照化変更前と変更後の F 関数内の構造を図 4 に表す．ただしこれは F 関数の一部分である．ここで入力は x_n(1byte)，出力は

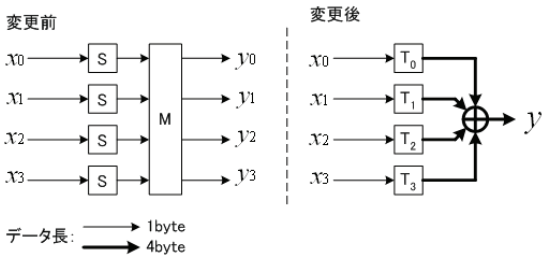

図 4　T_n テーブル参照化

y_n(1byte) で表している．次に T_n テーブルの作成方法について述べる．変更前の S-box と行列変換 M による変換を式 6 で

表す．

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = M \begin{pmatrix} S(x_0) \\ S(x_1) \\ S(x_2) \\ S(x_3) \end{pmatrix} = \begin{pmatrix} 0x02 & 0x03 & 0x01 & 0x01 \\ 0x01 & 0x02 & 0x03 & 0x01 \\ 0x01 & 0x01 & 0x02 & 0x03 \\ 0x03 & 0x01 & 0x01 & 0x02 \end{pmatrix} \begin{pmatrix} S(x_0) \\ S(x_1) \\ S(x_2) \\ S(x_3) \end{pmatrix} \quad (6)$$

この式を展開すると，式 7 で表される．

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} = \begin{pmatrix} 0x02 \\ 0x01 \\ 0x01 \\ 0x03 \end{pmatrix} S(x_0) \oplus \begin{pmatrix} 0x03 \\ 0x02 \\ 0x01 \\ 0x01 \end{pmatrix} S(x_1) \oplus \begin{pmatrix} 0x01 \\ 0x03 \\ 0x02 \\ 0x01 \end{pmatrix} S(x_2) \oplus \begin{pmatrix} 0x01 \\ 0x01 \\ 0x03 \\ 0x02 \end{pmatrix} S(x_3) \quad (7)$$

式 7 において列ベクトルと S-box 変換を統合して 1byte 入力 4byte 出力の置換表 T_n で表すと，式 7 は式 8 で書き直せる．

$$y_0 \| y_1 \| y_2 \| y_3 = T_0(x_0) \oplus T_1(x_1) \oplus T_2(x_2) \oplus T_3(x_3) \quad (8)$$

2.6.3 状態遷移関数の簡略化

技術仕様書 [6](4.4 節) では，状態遷移関数は F 関数，排他的論理和，ワードの代入で構成されている．そのため，ステート部を実装するならば

$$\begin{aligned} a_0^{(t+1)} &= a_1^t \\ a_1^{(t+1)} &= a_0^t \oplus F(a_1^t, a_4^t) \oplus C_1 \\ a_2^{(t+1)} &= a_0^t \oplus F(a_1^t, b_{10}^t) <<< 17) \oplus C_1 \end{aligned}$$

と 3 回の代入が必要である．しかし，本実装では視点を変え，図 5，図 6 のように，ワード代入を行わずに，排他的論理和と F 関数のみで状態遷移関数を構成し，代入回数を減らし高速化を図る．つまり，ワード代入を行わない代わりに，1 ラウンドの状態遷移を行うごとに，排他的論理和と F 関数が差し込まれる内部状態のポインタを 1 つずつずらしている．このポインタはラウンド番号 T から定まる．これにより，ワード代入を行う処理を削減できる．この様な簡略化をバッファ部の更新関数 λ に対しても適用することにより，ラウンドあたり 16 回のワード代入を 3 回に削減できる．

ρ 関数の構造については，変更前を図 5 に，変更後を図 6 に示す．図中の $A_n^{(t+1)}$ と $a_n^{(t+1)}$ はステートを現しており，の関係は次式の通りである．

$$A_0^{(t+1)} = a_2^{(t+1)}, A_1^{(t+1)} = a_0^{(t+1)}, A_2^{(t+1)} = a_1^{(t+1)}$$

λ 関数の構造については，変更前を図 7 に，変更後を図 8 に示す．図中の $B_n^{(t+1)}$ と $b_n^{(t+1)}$ はバッファを現しており，の関係

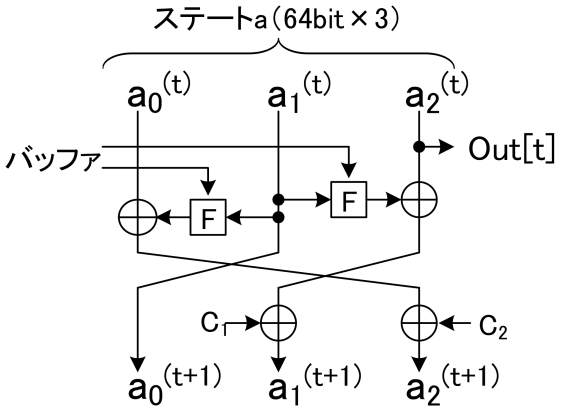

図 5　ステート部 (変更前)

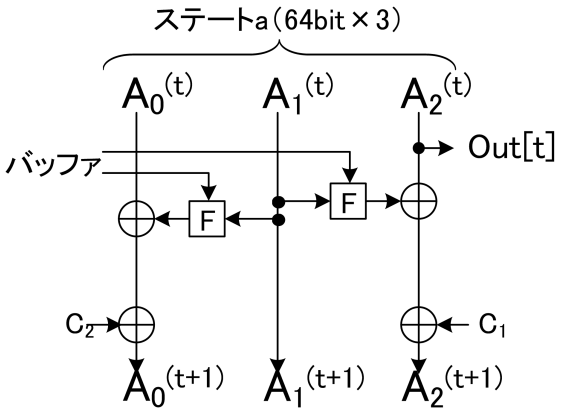

図 6　ステート部 (変更後)

は次式の通りである．

$$\begin{aligned} &B_0^{(t+1)} = b_1^{(t+1)}, B_1^{(t+1)} = b_2^{(t+1)}, B_2^{(t+1)} = b_3^{(t+1)}, B_3^{(t+1)} = b_4^{(t+1)}, \\ &B_4^{(t+1)} = b_5^{(t+1)}, B_5^{(t+1)} = b_6^{(t+1)}, B_6^{(t+1)} = b_7^{(t+1)}, B_7^{(t+1)} = b_8^{(t+1)}, \\ &B_8^{(t+1)} = b_9^{(t+1)}, B_9^{(t+1)} = b_{10}^{(t+1)}, B_{10}^{(t+1)} = b_{11}^{(t+1)}, B_{11}^{(t+1)} = b_{12}^{(t+1)}, \\ &B_{12}^{(t+1)} = b_{13}^{(t+1)}, B_{13}^{(t+1)} = b_{14}^{(t+1)}, B_{14}^{(t+1)} = b_{15}^{(t+1)}, B_{15}^{(t+1)} = b_0^{(t+1)} \end{aligned}$$

3　実装評価

3.1　実装環境

実行プログラムのコンパイル条件を表 1 に示す．L1 は L1 キャッシュの容量を意味する．なお，D0 とは Core i7 のステッピングであり，ステッピングでの違いでも処理能力がわずかに異なる場合があるため記載している．64 ビットの実装環境を表 2 に示す．2001 年の CRYPTREC 報告書 [1] との比較のために，同種の CPU を用いた実装環境を表 3 に示す．

3.2　処理時間の測定方法

アセンブリ命令である rdtsc を用い，CPU のクロックサイクルを処理時間として測定する．

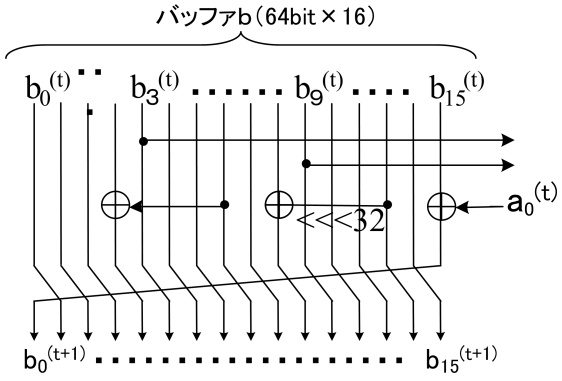

図 7　バッファ部 (変更前)

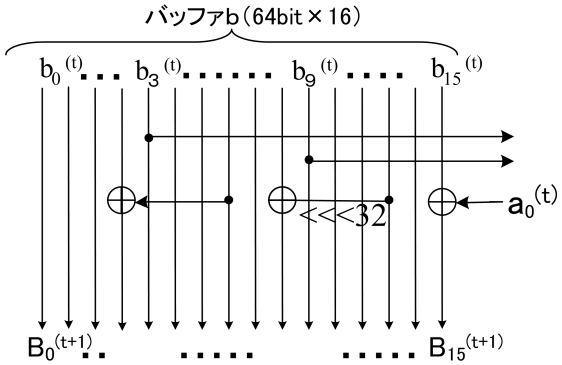

図 8　バッファ部 (変更後)

鍵スケジュール処理の処理時間の測定手順

ランダムな鍵に対し鍵スケジュール処理時間を測定するため，第 i 回目の鍵 K_i から生成した拡大鍵の一部を K_{i+1} として用いる．なお，$K_0 = 0$ とする．次の手順 1 から 4 までを実行する．

1. rdtsc でクロックを取得
2. 鍵 K_i を用い，拡大鍵を生成．この処理を $i = 1, 2, 3, \cdots, l$ と l 回繰り返し実行.
3. rdtsc でクロックを取得
4. 一回の鍵スケジュール処理に要したクロックサイクル $t = \frac{\text{(手順 3 で得たクロック値)}-\text{(手順 1 で得たクロック値)}}{l}$ を処理時間として出力

なお，実際の測定では $l = 2^{24}$ とした.

暗号化・復号処理の処理時間の測定手順

ランダムな平文に対し暗号化時間を測定するため，第 i ブロックの平文 P_i を暗号化した暗号文を P_{i+1} として用いる．なお，$P_0 = 0$ とし，鍵は 0 とする．次の手順 1 から 4 までを実行する．

1. rdtsc でクロックを取得
2. 平文 P_i に対し暗号化処理を行ない，暗号文 P_{i+1} を生成．この処理を $i = 1, 2, 3, \cdots, l$ と l 回繰り返し実行.
3. rdtsc でクロックを取得
4. 一回の暗号化処理に要したクロックサイクル $t = \frac{\text{(手順 3 で得たクロック値)}-\text{(手順 1 で得たクロック値)}}{l}$ を処理時間として出力

復号の際は，上記の平文が暗号文に，暗号文が平文に，暗号化処理が復号処理とする．

なお，実際の測定では $l = 2^{24}$ とした.

処理が同一であるものは，処理時間も同一と考えられるため，いずれか 1 つのみを測定する．以下に Camellia の例を示す.

- 暗号化処理は復号処理とは拡大鍵を与える順番の違いのみであるため，暗号化処理のみを測定.
- 暗号化 (復号) 処理にて 192 ビット鍵と 256 ビット鍵での処理は同一であるため，256 ビット鍵での処理のみを測定.

測定結果を 3.3 節に示す.

3.3　処理時間の測定結果

暗号化・復号処理時間の測定結果を表 5 に示す．プラットフォームの意味は以下の通りである.

Core i7 x86　表 2 の環境で 32 ビットプログラムを 32 ビット OS で実行

Core i7 WOW64　表 2 の環境で 32 ビットプログラムを 64 ビット OS で実行

Core i7 x64　表 2 の環境で 64 ビットプログラムを 64 ビット OS で実行

Pentium III x86　表 3 の環境で 32 ビットプログラムを 32 ビット OS で実行

表 1　コンパイル条件

プログラミング言語	C 言語
コンパイラ	Visual C++ 2008 Professional
コンパイラオプション	/O2(最適化オプション:実行速度)

表 2　実装環境 (Core i7)

CPU	Core i7 960 (D0) (L1:16KB)
OS	Windows 7 Enterprise (x86, x64)

表 3　実装環境 (Pentium III)

CPU	Pentium III 800MHz (L1:32KB)
OS	Windows XP Professional (x86)

表 4　鍵スケジュール処理時間の測定結果

処理			クロックサイクル		
暗号名	実装名	プラットフォーム	128 ビット	192 ビット	256 ビット
Camellia	NTT 最適化コード	Core i7 x86	392	553	555
		Core i7 WOW64	401	566	566
		Core i7 x64	340	491	482
		Pentium III x86	569	809	805
	32 ビット高速実装	Core i7 x86	142	237	233
		Core i7 WOW64	147	241	235
		Core i7 x64	133	210	209
		Pentium III x86	196	326	324
	64 ビット高速実装	Core i7 x86	191	273	270
		Core i7 WOW64	205	283	270
		Core i7 x64	99	140	152
		Pentium III x86	262	383	381

MUGI は 64 ビット単位で擬似乱数を出力する擬似乱数生成器であるが，本稿で扱っている 128 ビットブロック暗号と単位を揃えるために，128 ビット単位でのクロックサイクルとする．参考のために，2001 年の CRYPTREC 報告書 [1] の結果の一部および参照コードの測定結果も示す．

3.4　考察

3.4.1　プラットフォームによる違い

■32 ビット向けに実装したプログラム

クロックサイクルは *x*64 < *x*86 < *WOW*64 となった．これは，32 ビット向けに実装していても可能であれば CPU が 64 ビット向けに最適化しているためであると考えられる．また，WOW64 は 32 ビットを 64 ビット上でエミュレーション実行しているため，最も遅くなっていると考えられる．

■64 ビット向けに実装したプログラム

クロックサイクルは *x*64 < *x*86 < *WOW*64 となった．また，32 ビット環境下では，32 ビット向けに実装したプログラムよりも低速になった．64 ビット向けに実装したプログラムではテーブルサイズが大きくなっているため，低速な L2 キャッシュを使うことによるテーブル参照の低速化が発生するが，その一方で 64 ビットへの最適化による高速化自体も行われているため，64 ビット環境下では結果的に高速化に繋がっていた．しかし,，同じプログラムを 32 ビット環境下で実行すると，64 ビット向けへの最適化では 32 ビット環境下では高速化しないため，テーブル参照の低速化の影響により，低速化したと考えられる．

■2001 年の CRYPTREC 報告書 [1] の実装結果との比較

Pentium III で実行した結果を比較すると，全体的に低速化していることが分かる．これは，CRYPTREC 報告書においての実装ではアセンブリを用いているが，本稿での実装ではアセンブリを用いていないためであると考えられる．なお，Core i7 と Pentium III はアーキテクチャが異なるため，比較はできない．

以下では，特に 32 ビットおよび 64 ビットにおいて実装した暗号について記述する．

3.4.2　Camellia

■32 ビット高速実装

鍵スケジュール

NTT の最適化コードよりも大幅に高速になった．これは，拡大鍵生成の重複部を省略したことが大幅な高速化に繋がっていると考えられる．

また，64 ビット高速実装では 64 ビット環境下にて更に高速化している．

暗号化・復号

NTT の最適化コードよりも高速になった．これは，変数の入出力ビット数を 8 ビットからアーキテクチャと同じビット数に広げたためと考えられる．

■64 ビット高速実装

64 ビット環境下においては，NTT の最適化コードより高速になったが，32 ビット高速実装と比べるとほとんど変わらなかった．これは，前述のテーブルサイズの低速キャッシュへの移行によるテーブル参照の低速化が原因であると考えられる．

テーブルサイズが 16KB であるが，Core i7 960 の L1 キャッシュも 16KB であるため，拡大鍵も L1 キャッシュに入ることを考えると，テーブルの一部が L1 キャッシュより低速な L2 キャッシュに移動していると考えられる．

3.4.3　Hierocrypt-3

Core i7 x64 における 32 ビット高速実装と 64 ビット高速実装を比較すると，64 ビット高速実装の方がやや高速化している．これは，64 ビット高速実装で用いたテーブルが L1 キャッシュよりも大きいため，前述のテーブル参照の低速化が発生したが，64 ビットに最適化したことによる高速化によりテー

ブル参照の低速化を上回ったと考えられる．

4 結論

CRYPTREC 推奨暗号の Camellia，CIPHERUNICORN-A，Hierocrypt-3，SC2000，MUGI について C 言語のみを用いて高速実装を行った．その結果，32 ビット環境よりも 64 ビット環境の方が高いパフォーマンスを得られる方が判明した．さらに、プログラムを 64 ビットに最適化することにより，より高いパフォーマンスを得られる傾向にあることが判明した．しかし，比較対象である 2001 年の CRYPTREC 報告書 [1] の実装結果はアセンブリを用いているため、アセンブリを用いていない本稿の実装結果の方が低速であった。

参考文献

[1] 情報処理振興事業協会，通信・放送機構，”暗号技術評価報告書 (2001 年度版) CRYPTREC Report 2001”, http://www.ipa.go.jp/security/fy13/report/cryptrec/c01.pdf, 2002

[2] 青木和麻呂, 市川哲也, 神田雅透, 松井充, 盛合志帆, 中嶋純子, 時田俊雄,“ 128 ビットブロック暗号 Camellia アルゴリズム仕様書 (第二版) ”, http://info.isl.ntt.co.jp/crypt/camellia/dl/01jspec.pdf , 2001

[3] 日本電気株式会社,“ 暗号技術仕様書 CIPHERUNICORN-A ”， http://www.cryptrec.go.jp/cryptrec_03_spec_cypherlist_files/PDF/07_01jspec.pdf, 2001

[4] 株式会社東芝，“ 暗号技術仕様書:Hierocrypt-3 ”， http://www.toshiba.co.jp/rdc/security/hierocrypt/files/hc3_02jspec.pdf , 2002

[5] 富士通研究所,“ 共通鍵ブロック暗号 SC2000 暗号技術仕様書 (2001 年 9 月 26 日) ”, http://www.cryptrec.go.jp/cryptrec_03_spec_cypherlist_files/PDF/09_01jspec.pdf, 2001]

[6] 株式会社日立製作所,“ 擬似乱数生成器 MUGI 仕様書 Ver.1.3 ”, http://www.cryptrec.go.jp/cryptrec_03_spec_cypherlist_files/PDF/10_02jspec.pdf, 2002

[7] 下山武司, 屋並仁史, 横山和弘, 武仲正彦, 伊藤孝一, 矢嶋純, 鳥居直哉, 田中秀磨,“ 共通鍵ブロック暗号 SC2000 ” ISEC2000-72, 2000

[8] 日本電信電話株式会社, 三菱電機株式会社,“ Camellia 暗号エンジン (C, version 1.2.0) ”, http://info.isl.ntt.co.jp/crypt/camellia/engine.html, 2007

[9] 佐野文彦, 村谷博文, 大熊建司, 川村信一, 本山雅彦,“ 次世代暗号 Hierocrypt の C 言語による実装 ”, コンピュータセキュリティ, 11-9, 2000

[10] 増田豊史, 西川栄光, 五十嵐保隆, 金子敏信,“ 高性能サーバ環境を意識した共通鍵ブロック暗号の高速実装 ”, SCIS2010, 2010

表 5　暗号化・復号処理時間の測定結果

処理			鍵長		
暗号名	実装名	プラットフォーム	128	192	256
Camellia	NTT 最適化コード	Core i7 x86	527		695
		Core i7 WOW64	538		707
		Core i7 x64	525		692
		Pentium III x86	756		1009
	32 ビット高速実装	Core i7 x86	463		612
		Core i7 WOW64	478		626
		Core i7 x64	483		649
		Pentium III x86	557		746
	64 ビット高速実装	Core i7 x86	615		824
		Core i7 WOW64	699		920
		Core i7 x64	481		659
		Pentium III x86	773		1025
	CRYPTREC	Pentium III x86	327	-	
CIPHERUNICORN-A	参照コード	Core i7 x86			1695
		Core i7 WOW64			1736
		Core i7 x64			1608
		Pentium III x86			2327
	高速実装	Core i7 x86			1628
		Core i7 WOW64			1654
		Core i7 x64			1523
		Pentium III x86			2065
	CRYPTREC	Pentium III x86			1574
Hierocrypt-3(暗号化)	32 ビット高速実装	Core i7 x86	984	1150	1319
		Core i7 WOW64	1000	1168	1336
		Core i7 x64	928	1080	1260
		Pentium III x86	1365	1616	2215
	64 ビット高速実装	Core i7 x86	1193	1398	1619
		Core i7 WOW64	1206	1424	1631
		Core i7 x64	917	1065	1226
		Pentium III x86	1738	2060	2359
	CRYPTREC	Pentium III x86	406	-	-
Hierocrypt-3(復号)	32 ビット高速実装	Core i7 x86	1176	1384	1619
		Core i7 WOW64	1211	1415	1629
		Core i7 x64	1076	1259	1451
		Pentium III x86	1651	1949	2215
	64 ビット高速実装	Core i7 x86	1363	1398	1859
		Core i7 WOW64	1392	1652	1888
		Core i7 x64	1042	1223	1409
		Pentium III x86	1905	2282	2611
	CRYPTREC	Pentium III x86	428	-	-
SC2000	高速実装	Core i7 x86	525		601
		Core i7 WOW64	524		600
		Core i7 x64	413		467
		Pentium III x86	729		821
	CRYPTREC	Pentium III x86	391		-
MUGI	高速実装	Core i7 x86			128
		Core i7 WOW64			159
		Core i7 x64			113
		Pentium III x86			198