暗号アルゴリズム試験仕様書

２００８年1月１８日

独立行政法人　情報処理推進機構

目 次

1　はじめに

本稿では，暗号アルゴリズム試験ツールに実装された暗号アルゴリズム試験仕様を記述する．試験の対象とする暗号は，第 2 章に示す通りである．

1.1　暗号アルゴリズム試験ツールの概要

暗号アルゴリズム試験ツールは次の特長を持つ．

- 暗号モジュール試験及び認証制度における認証機関が承認したセキュリティ機能 (ASF-01 を参照) を試験の対象とする．
- 試験対象の実装が暗号アルゴリズム仕様書に記述された事項に従って実装されているかどうかを試験する．
- 例えばディジタル署名の場合は署名生成機能，署名検証機能，鍵ペア生成機能など，各暗号が有する機能ごとに試験を行う．
- 暗号アルゴリズム試験ツールと試験対象の実装は，各種ファイルを介してデータの通信を行う．このことにより，様々なプラットフォーム上の暗号モジュールを試験可能となる．
- 暗号アルゴリズム試験ツールはリファレンスモジュールを動作させてテストベクタを自動生成する．

1.2　本稿の構成

本稿の以降の構成は次の通りである．

- 第 2 章: 暗号アルゴリズム試験ツールが試験の対象とする暗号アルゴリズムを示す．
- 第 3 章以降: 各暗号の試験項目を記述する．

なお，本稿を通して次の略語を使用する．

- JCATT: 暗号アルゴリズム試験ツール
- IUT: JCATT が試験の対象とする実装

2　JCATT の対象暗号アルゴリズム

JCATT が試験対象とする暗号アルゴリズムを次に示す.

署名

- DSA
- ECDSA
- RSASSA-PKCS1-v1_5
- RSA-PSS

公開鍵暗号 (守秘)

- RSA-OAEP
- RSAES-PKCS1-v1_5

鍵共有

- Diffie-Hellman, dhStatic
- Diffie-Hellman, dhEphem
- Diffie-Hellman, dhOneFlow
- Diffie-Hellman, dhHybrid1
- Diffie-Hellman, dhHybrid2
- Diffie-Hellman, dhHybridOneFlow
- Elliptic Curve Diffie-Hellman
- Elliptic Curve Cofactor Diffie-Hellman
- PSEC-KEM

ブロック暗号

64 ビットブロック

- CIPHERUNICORN-E
- Hierocrypt-L1
- MISTY1
- 3-key Triple DES

128 ビットブロック

- AES
- Camellia
- CIPHERUNICORN-A
- Hierocrypt-3
- SC2000

ストリーム暗号

- MUGI
- MULTI-S01
- 128-bit RC4

ハッシュ関数

- RIPEMD-160
- SHA-1
- SHA-224
- SHA-256
- SHA-384
- SHA-512

メッセージ認証

- HMAC

擬似乱数生成関数

- CTR_DRBG
- OFB_DRBG
- Hash_DRBG
- PRNG based on SHA-1 in ANSI X9.42-2001 Annex C.1
- PRNG based on SHA-1 for general purpose in FIPS 186-2 (+ change notice 1) Appendix 3.1
- PRNG based on SHA-1 for general purpose in FIPS 186-2 (+ change notice 1) revised Appendix 3.1

3 署名

ディジタル署名 DSA，ECDSA，RSASSA-PKCS1-v1_5，RSA-PSS の各暗号アルゴリズム試験項目を記述する．

3.1 DSA

DSA の試験対象機能は次の通りである．

- ドメインパラメータ生成機能
- ドメインパラメータ検証機能
- 鍵ペア生成機能
- 署名生成機能
- 署名検証機能

3.1.1 ドメインパラメータ生成機能試験

ドメインパラメータ生成機能試験の試験項目は次の通りである．

FIPS 186-2 Appendix 2 に記述されているドメインパラメータ p，q，$counter$，$SEED$ の正当性も試験する．

- IUT が生成した $SEED$ および $counter$ を JCATT に入力し，JCATT は FIPS 186-2 Appendix 2 のアルゴリズムに従って 2 つの素数 p'，q' を計算する．この p'，q' と，IUT が生成した p，q がそれぞれ等しいこと．
- $g^q \equiv 1 \bmod p$ であること．
- IUT が生成した複数 (別途規定する数) のドメインパラメータ (p, q, g) が全て異なるものであること．

3.1.2 ドメインパラメータ検証機能試験

ドメインパラメータ検証機能試験の試験項目は次の通りである．

- JCATT が与えた前節に記述したドメインパラメータ生成機能に対する試験に適合するようなドメインパラメータに対して，IUT が “合格” と判定すること．
- JCATT が与えた前節に記述したドメインパラメータ生成機能に対する試験に違反するようなドメインパラメータに対して，IUT が “不正” と判定すること．

3.1.3 鍵ペア生成機能試験

鍵ペア生成機能試験の試験項目は次の通りである．なお，プライベート鍵を x，公開鍵を y とする．

- $y \equiv g^x \bmod p$ であること
- $1 \leq x \leq q-1$, $2 \leq y \leq p-2$ であること
- $y^q \equiv 1 \bmod p$ であること
- IUT が生成した複数 (別途規定する数) の鍵ペアが全て異なるものであること．

3.1.4 署名生成機能試験

署名生成機能試験の試験項目は次の通りである．

- JCATT が与えたプライベート鍵 x および平文に対して，IUT が生成した署名を，JCATT が署名検証した時に署名検証合格となること．
- 同じ平文，同じプライベート鍵に対して複数 (別途規定する数) 署名を生成させた時，IUT が同じ署名を生成しないこと．

3.1.5 署名検証機能試験

署名検証機能試験の試験項目は次の通りである．

- JCATT が与えた正しい公開鍵 y，平文および署名，ならびに指定されたハッシュ関数に対して，IUT が正しく署名検証合格と判定すること．
- JCATT が改竄した平文，署名，または公開鍵に対して，IUT が署名検証不合格と判定すること．

3.2 ECDSA

ECDSA の試験対象機能は次の通りである.

- ドメインパラメータ生成機能
- ドメインパラメータ検証機能
- 鍵ペア生成機能
- 公開鍵検証機能
- 署名生成機能
- 署名検証機能

3.2.1 ドメインパラメータ生成機能試験

ドメインパラメータ生成機能試験の試験項目は次の通りである.

3.2.1.1 標数 p の場合

検証可能なランダム曲線であるかどうかに応じて試験 1，2 のいずれかを実行する．既定は試験 1 である.

試験 1(既定の試験)

$\mathbb{F}_p$ 上楕円曲線の場合

- n が 160 ビット以上であること.
- $4a^3 + 27b^2 \not\equiv 0 \bmod p$ であること.
- a，b，x_G，y_G が 0 以上 $p-1$ 以下の整数であること.
- $y_G^2 \equiv x_G^3 + ax_G + b \bmod p$ であること.
- n が素数であること.
- p が素数であること.
- $h \leq 4$ かつ $h = \lfloor(\sqrt{p}+1)^2/n\rfloor$ であること.
- $nG = \mathcal{O}$ であること.
- すべての $1 \leq B < 20$ に対して $q^B \not\equiv 1 \bmod n$ であること.
- $nh \neq p$ であること.
- IUT が生成した複数 (別途規定する数) のドメインパラメータが全て異なるものであること.

試験 2(検証可能なランダム曲線に対する試験)

$\mathbb{F}_p$ 上楕円曲線の場合

- IUT が SEED を出力する場合，ANSI X9.62 A.3.3 に記述された “Verifiably at random” な方法で生成された曲線であることを検証する．
- n が 160 ビット以上であること．
- $4a^3 + 27b^2 \not\equiv 0 \bmod p$ であること．
- a，b，x_G，y_G が 0 以上 $p-1$ 以下の整数であること．
- $y_G^2 \equiv x_G^3 + ax_G + b \bmod p$ であること．
- n が素数であること．
- p が素数であること．
- $h \leq 4$ かつ $h = \lfloor (\sqrt{p}+1)^2/n \rfloor$ であること．
- $nG = \mathcal{O}$ であること．
- すべての $1 \leq B < 20$ に対して $q^B \not\equiv 1 \bmod n$ であること．
- $nh \neq p$ であること．
- IUT が生成した複数 (別途規定する数) のドメインパラメータが全て異なるものであること．

3.2.1.2 標数 2 の場合

検証可能なランダム曲線であるかどうかに応じて試験 1，2 のいずれかを実行する．既定は試験 1 である．

試験 1(既定の試験)

$\mathbb{F}_{2^m}$ 上楕円曲線の場合

- n が 160 ビット以上であること．
- $f(x)$ が次数 m のバイナリ既約多項式であること．
- a，b，x_G，y_G が次数 $m-1$ 以下のバイナリ多項式であること．
- $b \neq 0$ in $\mathbb{F}_{2^m}$ であること．
- $y_G^2 + x_G y_G \equiv x_G^3 + ax_G^2 + b$ in $\mathbb{F}_{2^m}$ であること．
- n が素数であること．
- $h \leq 4$ かつ $h = \lfloor (\sqrt{2^m}+1)^2/n \rfloor$ であること．
- $nG = \mathcal{O}$ であること．
- すべての $1 \leq B < 20$ に対して $2^{mB} \not\equiv 1 \bmod n$ であること．
- $nh \neq 2^m$ であること．
- IUT が生成した複数 (別途規定する数) のドメインパラメータが全て異なるものであること．

試験 2(検証可能なランダム曲線に対する試験)

$\mathbb{F}_{2^m}$ 上楕円曲線の場合

- IUT が SEED を出力する場合，ANSI X9.62 A.3.3 に記述された “Verifiably at random” な方法で生成された曲線であることを検証する．
- n が 160 ビット以上であること．
- $f(x)$ が次数 m のバイナリ既約多項式であること．
- a，b，x_G，y_G が次数 $m-1$ 以下のバイナリ多項式であること．
- $b \neq 0$ in $\mathbb{F}_{2^m}$ であること．
- $y_G^2 + x_G y_G \equiv x_G^3 + a x_G^2 + b$ in $\mathbb{F}_{2^m}$ であること．
- n が素数であること．
- $h \leq 4$ かつ $h = \lfloor (\sqrt{2^m}+1)^2/n \rfloor$ であること．
- $nG = \mathcal{O}$ であること．
- すべての $1 \leq B < 20$ に対して $2^{mB} \not\equiv 1 \bmod n$ であること．
- $nh \neq 2^m$ であること．
- IUT が生成した複数 (別途規定する数) のドメインパラメータが全て異なるものであること．

3.2.2 ドメインパラメータ検証機能試験

ドメインパラメータ検証機能試験の試験項目は次の試験 1 または試験 2 である．既定は試験 1 である．

試験 1(既定の試験)

- JCATT が与えた前節に記述したドメインパラメータ生成機能試験項目に適合するようなドメインパラメータに対して，IUT が “合格” と判定すること．
- JCATT が与えた前節に記述したドメインパラメータ生成機能試験項目に違反するようなドメインパラメータに対して，IUT が “不正” と判定すること．

試験 2(検証可能なランダム曲線に対する試験)

- IUT が ANSI X9.62 A.3.3 に記述された “Verifiably at random” な方法で生成された曲線であることを検証する機能を持つ場合，JCATT が与えた正しいドメインパラメータ (SEED を含む) に対して “合格” と判定し，JCATT が与えた不正なドメインパラメータに対して “不正” と判定すること．

3.2.3 鍵ペア生成機能試験

鍵ペア生成機能試験の試験項目は次の通りである．

3.2.3.1 標数 p の場合

- $Q \neq \mathcal{O}$ であること.
- $y_Q^2 \equiv x_Q^3 + ax_Q + b \bmod p$ であること.
- $nQ = \mathcal{O}$ であること.
- $Q = dG$ であること.
- IUT が生成した複数 (別途規定する数) の鍵ペアが全て異なるものであること.

3.2.3.2 標数 2 の場合

- $Q \neq \mathcal{O}$ であること.
- $y_Q^2 + x_Q y_Q \equiv x_Q^3 + ax_Q^2 + b$ in $\mathbb{F}_{2^m}$ であること.
- $nQ = \mathcal{O}$ であること.
- $Q = dG$ であること.
- IUT が生成した複数 (別途規定する数) の鍵ペアが全て異なるものであること.

3.2.4 公開鍵検証機能試験

公開鍵検証機能試験の試験項目は次の通りである.

- JCATT が与えた前節に記述した公開鍵ペア生成機能試験項目に適合するような公開鍵に対して，IUT が “合格” と判定すること.
- JCATT が与えた前節に記述した公開鍵ペア生成機能試験項目に違反するような公開鍵に対して，IUT が “不正” と判定すること.

3.2.5 署名生成機能試験

署名生成機能試験の試験項目は次の通りである.

- JCATT が与えたプライベート鍵 d および平文に対して，IUT が生成した署名を，JCATT が署名検証した時に署名検証合格となること.
- 同じ平文，同じプライベート鍵に対して複数 (別途規定する数) 署名を生成させた時，IUT が同じ署名が生成されないこと.

ハッシュ関数は，SHA-1，SHA-224，SHA-256，SHA-384，SHA-512，RIPEMD-160 の中から指定する.

3.2.6 署名検証機能試験

署名検証機能試験の試験項目は次の通りである．

- JCATT が与えた正しい公開鍵 Q，平文および署名，ならびに指定されたハッシュ関数に対して，IUT が署名検証合格と判定すること．
- JCATT が改竄した平文，署名，または公開鍵に対して，IUT が署名検証不合格と判定すること．

ハッシュ関数は，SHA-1，SHA-224，SHA-256，SHA-384，SHA-512，RIPEMD-160 の中から指定する．

3.3 RSASSA-PKCS1-v1_5

RSASSA-PKCS1-v1_5 の試験対象機能は次の通りである．

- 鍵ペア生成機能
- 署名生成機能
- 署名検証機能

3.3.1 鍵ペア生成機能試験

RSA 暗号 (RSAES-PKCS1-v1_5，RSA-OAEP，RSASSA-PKCS1-v1_5，RSA-PSS) のプライベート鍵は，プライベート鍵演算 (復号および署名生成) において Chinese Remainder Theorem (CRT) を用いるかどうかによって，2 種類に分かれる．CRT を用いない場合，プライベート鍵は (n, d) の組であり，CRT を用いる場合は $(p, q, dP, dQ, qInv)$ の組である．したがって，試験対象の鍵ペア生成機能が CRT 用の $dP, dQ, qInv$ を出力するかどうかによって，試験項目を 2 通りに分けることとする．それぞれ試験項目は次の通りである．

CRT を用いない場合の鍵ペア生成機能試験項目

- n が指定されたビット長であること．
- p と q のビット長が等しいこと．
- p は素数であること．
- q は素数であること．
- $n = pq$ を満たすこと．
- $e \cdot d \equiv 1 \bmod \lambda(n)$ を満たすこと．
- 生成された複数 (別途規定する数) の鍵ペアが全て異なるものであること．

CRT を用いる場合の鍵ペア生成機能試験項目

- n が指定されたビット長であること．
- p と q のビット長が等しいこと．
- p は素数であること．
- q は素数であること．
- $n = pq$ を満たすこと．
- $e \cdot dP \equiv 1 \bmod p - 1$ を満たすこと．
- $e \cdot dQ \equiv 1 \bmod q - 1$ を満たすこと．
- $q \cdot qInv \equiv 1 \bmod p$ を満たすこと．
- 生成された複数 (別途規定する数) の鍵ペアが全て異なるものであること．

ここで，$\lambda(n)$ は $p - 1$ と $q - 1$ の最小公倍数である．

3.3.2 署名生成機能試験

署名生成機能試験の試験項目は次の通りである．

- JCATT が与えたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$，および平文，ならびに指定されたハッシュ関数に対して IUT が正しい署名を生成すること．

ハッシュ関数は，SHA-1，SHA-256，SHA-384，SHA-512 の中から指定する．

3.3.3 署名検証機能試験

署名検証機能試験の試験項目は次の通りである．

- JCATT が与えた公開鍵 (n, e)，平文および署名，ならびに指定されたハッシュ関数に対して，IUT が署名検証合格と判定すること．
- JCATT が改竄した平文，署名，または公開鍵に対して，IUT が署名検証不合格と判定すること．

ハッシュ関数は，SHA-1，SHA-256，SHA-384，SHA-512 の中から指定する．

3.4 RSA-PSS

RSA-PSS の試験対象機能は次の通りである．

- 鍵ペア生成機能
- 署名生成機能
- 署名検証機能

3.4.1 鍵ペア生成機能試験

第 3.3 節に記述した鍵ペア生成機能試験項目と同じである．

3.4.2 署名生成機能試験

署名生成機能試験の試験項目は次の試験 1 または試験 2 または試験 3 である．既定は試験 1 である．

試験 1(既定の試験)

- JCATT が与えたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$，平文に対して IUT が生成した署名を，JCATT が署名検証した時に署名検証合格となること．
- *salt* 長が 0 でない場合，同じ平文，同じプライベート鍵に対して，複数 (別途規定する数) 署名を生成させた時，IUT が同じ署名が生成しないこと．

試験 2(任意で実施する試験．既知入出力試験)

- JCATT が与えたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$，平文，ならびに指定されたハッシュ関数，擬似乱数生成関数，乱数シードに対して IUT が正しい署名を生成すること．

試験 3(任意で実施する試験．*salt* を指定して行う既知入出力試験)

- JCATT が与えたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$，平文，ならびに指定されたハッシュ関数，*salt* に対して IUT が正しい署名を生成すること．

ハッシュ関数は，SHA-1，SHA-256，SHA-384，SHA-512 の中から指定する．
また試験 2 を実施する場合，擬似乱数生成関数は第 2 章に示したアルゴリズムの中から指定する．

3.4.3 署名検証機能試験

第 3.3 節に記述した RSASSA-PKCS1-v1_5 の署名検証機能試験項目と同じである．

4　公開鍵暗号 (守秘)

公開鍵暗号 (守秘)
RSA-OAEP RSAES-PKCS1-v1_5，の各暗号アルゴリズム試験項目を記述する．

4.1　RSA-OAEP

RSA-OAEP の試験対象機能は次の通りである．

- 鍵ペア生成機能
- 暗号化機能
- 復号機能

4.1.1　鍵ペア生成機能試験

鍵ペア生成機能試験の試験項目は第 3.3 節に記述した試験項目と同じである．

4.1.2　暗号化機能試験

暗号化機能試験の試験項目は次の試験 1 または試験 2 または試験 3 である．既定は試験 1 である．

試験 1(既定の試験)

- JCATT が与えた公開鍵 (n, e) および平文，ならびに指定されたハッシュ関数およびマスク生成関数 MGF およびラベル L に対して IUT が生成した暗号文を，JCATT が復号した時に，もとの平文に復号されること．
- 同じ平文，同じ公開鍵，同じラベル値に対して，複数 (別途規定する数) 暗号文を生成させた時，同じ暗号文が生成されないこと．

試験 2(任意で実施する試験．暗号文に対する既知入出力試験)

- JCATT が与えた公開鍵 (n, e)，平文およびラベル L，ならびに指定された擬似乱数生成関数，指定されたハッシュ関数およびマスク生成関数 MGF と, 乱数シードに対して正しい暗号文を IUT が生成すること．

試験 3(任意で実施する試験．中間値 $seed$ を指定して行う既知入出力試験)

- JCATT が与えた公開鍵 (n, e)，平文およびラベル L，ならびに指定されたハッシュ関数，マスク生成関数 MGF および中間値 $seed$ 対して正しい暗号文を IUT が生成すること．

ハッシュ関数は，SHA-1，SHA-256，SHA-384，SHA-512 の中から指定する．
マスク生成関数 MGF は，ANSI X9.44 に記載の SHA-1，SHA-256，SHA-384，SHA-512 ベースの関数の中から指定する．
また試験 2 を実施する場合，擬似乱数生成関数は第 2 章に示したアルゴリズムの中から指定する．

4.1.3 復号機能試験

復号機能の試験項目は次の通りである.

- 与えられたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$ と，与えられたラベル L と，指定されたハッシュ関数及びマスク生成関数 MGF と, 与えられた暗号文に対して，もとの平文に復号できること.
- 与えられたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$ と，与えられたラベル L と，指定されたハッシュ関数及びマスク生成関数 MGF と, 改竄された暗号文に対して不正検出を正しく行うこと.

ハッシュ関数は，SHA-1，SHA-256，SHA-384，SHA-512 の中から指定する.

マスク生成関数 MGF は，ANSI X9.44 に記載の SHA-1，SHA-256，SHA-384，SHA-512 ベースの関数の中から指定する.

4.2 RSAES-PKCS1-v1_5

RSAES-PKCS1-v1_5 の試験対象機能は次の通りである．

- 鍵ペア生成機能
- 暗号化機能
- 復号機能

4.2.1 鍵ペア生成機能試験

第 3.3 節に記述した鍵ペア生成機能試験項目と同じである．

4.2.2 暗号化機能試験

暗号化機能試験の試験項目は次の試験 1 または試験 2 または試験 3 である．既定は試験 1 である．

試験 1(既定の試験)

- JCATT が与えた公開鍵 (n, e) および平文に対して IUT が生成した暗号文を，JCATT で復号した時に，もとの平文に復号されること．
- 同じ平文，同じ公開鍵に対して，複数 (別途規定する数) 暗号文を生成させた時，IUT が同じ暗号文を生成しないこと．

試験 2(任意で実施する試験．既知入出力試験)

- JCATT が与えた公開鍵 (n, e) および平文，ならびに指定された擬似乱数生成関数および乱数シードに対して正しい暗号文を IUT が生成すること．

試験 3(任意で実施する試験．中間値 PS を指定して行う既知入出力試験)

- JCATT が与えた公開鍵 (n, e) および平文，ならびに指定された中間値 PS に対して正しい暗号文を IUT が生成すること．

試験 2 を実施する場合，擬似乱数生成関数は第 2 章に示したアルゴリズムの中から指定する．

4.2.3 復号機能試験

復号機能試験の試験項目は次の通りである．

- 与えられたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$ と，与えられた暗号文に対して，もとの平文に復号できること．
- 与えられたプライベート鍵 (n, d) または $(p, q, dP, dQ, qInv)$ と，改竄された暗号文に対して不正検出を正しく行うこと．

5 鍵共有

鍵共有アルゴリズム DH，ECDH，PSEC-KEM の試験項目を記述する．

5.1 Diffie-Hellman (DH)

DH の試験項目は，dhStatic，dhEphem，dhOneFlow，dhHybrid1，dhHybrid2，dhHybridOneFlow ともに共通である．

DH の試験対象機能は次の通りである．

- ドメインパラメータ生成機能
- ドメインパラメータ検証機能
- 鍵ペア生成機能
- 公開鍵検証機能
- 鍵共有機能

5.1.1 ドメインパラメータ生成機能試験

ドメインパラメータ生成機能試験の試験項目は次の試験 1 または試験 2 である．既定は試験 1 である．

ANSI X9.42 7.1 節に記述されているドメインパラメータ p，q，g，$pgenCounger$，$seed$ の正当性を試験する．

試験 1(既定の試験)

- IUT が生成した $seed$ および $pgenCounter$ を JCATT に入力し，JCATT は ANSI X9.42 Annex B.1.3 記載のアルゴリズムに従って 2 つの素数 p'，q' を JCATT が計算する．この p'，q' と，IUT が生成した p，q がそれぞれ等しいこと．
- $g^q \equiv 1 \bmod p$ であること．
- IUT が生成した複数 (別途規定する数) のドメインパラメータ (p, q, g) が全て異なるものであること．

試験 2(任意で実施する試験)

- IUT が生成した $seed$ および $pgenCounter$ を JCATT に入力し，JCATT は ANSI X9.42 Annex B.1.3 記載のアルゴリズムに従って 2 つの素数 p'，q' を JCATT が計算する．この p'，q' と，IUT が生成した p，q がそれぞれ等しいこと．
- $1 < h < p - 1$ であること．
- $g \equiv h^{(p-1)/q} \bmod p$ であること．
- IUT が生成した複数 (別途規定する数) のドメインパラメータ (p, q, g) が全て異なるものであること．

5.1.2 ドメインパラメータ検証機能試験

ドメインパラメータ検証機能試験の試験項目は次の通りである.

- JCATT が与えた前節に記述したドメインパラメータ生成機能に対する試験に適合するようなドメインパラメータに対して，IUT が “合格” と判定すること.
- JCATT が与えた前節に記述したドメインパラメータ生成機能に対する試験に違反するようなドメインパラメータに対して，IUT が “不正” と判定すること.

5.1.3 鍵ペア生成機能試験

鍵ペア生成機能試験の試験項目は次の通りである.

- $y \equiv g^x \bmod p$ であること
- $1 \leq x \leq q-1, 2 \leq y \leq p-2$ であること
- $y^q \equiv 1 \bmod p$ であること
- IUT が生成した複数 (別途規定する数) の鍵ペアが全て異なるものであること.

5.1.4 公開鍵検証機能試験

公開鍵検証機能試験の試験項目は次の通りである.

- 公開鍵 y およびドメインパラメータ (p, q, g) が以下の条件全てを満たしている時には，“合格” と判定し, そうでなければ “不正” と判定すること.
  - $2 \leq y \leq p-2$ であること
  - $y^q \equiv 1 \bmod p$ であること

5.1.5 鍵共有機能試験

鍵共有機能試験の試験項目は次の通りである.

- JCATT が与えた (複数の) プライベート鍵と公開鍵に対して，IUT が正しい共有鍵を生成すること.

KDF は，ANSI X9.42 に記載された次の KDF から指定する.

- SHA-1 を用いた Concatenation に基づく KDF
- SHA-224 を用いた Concatenation に基づく KDF
- SHA-256 を用いた Concatenation に基づく KDF
- SHA-384 を用いた Concatenation に基づく KDF
- SHA-512 を用いた Concatenation に基づく KDF

5.2 Elliptic Curve (Cofactor) Diffie-Hellman (ECDH)

ECDH の試験対象機能は次の通りである．

- ドメインパラメータ生成機能
- ドメインパラメータ検証機能
- 鍵ペア生成機能
- 公開鍵検証機能
- 鍵共有機能

5.2.1 ドメインパラメータ生成機能試験

第 3.2 節に記述した ECDSA のドメインパラメータ生成機能試験項目と同じである．

5.2.2 ドメインパラメータ検証機能試験

第 3.2 節に記述した ECDSA のドメインパラメータ検証機能試験項目と同じである．

5.2.3 鍵ペア生成機能試験

第 3.2 節に記述した ECDSA の鍵ペア生成機能試験項目と同じである．

5.2.4 公開鍵検証機能試験

第 3.2 節に記述した ECDSA の公開鍵検証機能試験項目と同じである．

5.2.5 鍵共有機能試験

鍵共有機能試験の試験項目は次の通りである．

- JCATT が与えた (複数の) プライベート鍵と公開鍵に対して，IUT が正しい共有鍵を生成すること．

KDF は，ANSI X9.63 に記載された次の中の KDF を指定する．KDF で用いるハッシュ関数は，SHA-1，SHA-224，SHA-256，SHA-384，SHA-512 の中から指定する．

5.3 PSEC-KEM

PSEC-KEM の試験対象機能は次の通りである．

- 鍵ペア生成機能
- セッション鍵暗号化機能
- セッション鍵復号機能

それぞれの機能の試験項目を以下に記述する．

5.3.1 鍵ペア生成機能試験

第 3.2 節に記述した ECDSA の鍵ペア生成機能試験項目と同じである．

5.3.2 セッション鍵暗号化機能試験

セッション鍵暗号化機能試験の試験項目は次の通りである．

- JCATT が与えた公開鍵 Q に対して，IUT が生成した暗号文を，JCATT で復号化した時に，暗号文正当性の検証が合格となること．
- 同じ公開鍵に対して複数 (別途規定する数) の暗号文を生成させた時，IUT が同じ暗号文を生成しないこと．

KDF は，ISO/IEC 18033-2 に記載された KDF1 から指定する．KDF1 で用いるハッシュ関数は，SHA-1，SHA-224，SHA-256，SHA-384，SHA-512 の中から指定する．

5.3.3 セッション鍵復号機能試験

セッション鍵復号機能試験の試験項目は次の通りである．

- JCATT が与えた鍵ペア (d, Q) および暗号文に対して，IUT が正しく復号すること．
- JCATT が改竄した暗号文に対して，IUT が正しく “棄却”すること．

KDF は，ISO/IEC 18033-2 に記載された KDF1 から指定する．KDF1 で用いるハッシュ関数は，SHA-1，SHA-224，SHA-256，SHA-384，SHA-512 の中から指定する．

6 ブロック暗号

64 ビットブロック暗号アルゴリズム，CIPHERUNICORN-E，Hierocrypt-L1，MISTY1，3-key Triple DES，ならびに 128 ビットブロック暗号アルゴリズム，AES，Camellia，CIPHERUNICORN-A，Hierocrypt-3，SC2000 の試験対象機能および各試験対象機能に対する試験項目を記述する．

ブロック暗号の試験対象機能は次の通りである．

- ECB モード暗号化/復号機能
- CBC モード暗号化/復号機能
- CFB モード暗号化/復号機能
- OFB モード暗号化/復号機能
- CTR モード暗号化/復号機能

ここで，OFB モードと CFB モードのフィードバックビット幅はブロック幅とする．ただし，AES と 3-key Triple DES に対しては，次の機能に対しても試験を実施することができる．

- CFB-1 モード暗号化/復号機能
- CFB-8 モード暗号化/復号機能

また，CTR モードは，NIST SP 800-38A の Appendix B.1[5] に記述されているインクリメンタルカウンタ，$x \leftarrow x + 1 \bmod 2^m$，により実装されている場合を試験の対象とする．

128 ビットブロック暗号 (Camellia，AES，Hierocrypt-3，CIPHERUNICORN-A，SC2000) の場合，鍵長は 128，192，256 ビットの 3 種類である．したがって，上記の各機能が 3 種類ずつに増えることに注意．

ブロック暗号の各機能に対する試験項目を次に示す．

AES

- 種々の平文 (暗号文) に対する既知入出力試験 (KAT-Text)
- 種々の鍵に対する既知入出力試験 (KAT-key)
- マルチブロックメッセージ試験 (MMT)
- モンテカルロ試験 (MCT)
- GFSbox 既知入出力試験 (KAT-GFSbox)
- KeySbox 既知入出力試験 (KAT-KeySbox)

3-key Triple DES

- 種々の平文 (暗号文) に対する既知入出力試験 (KAT-Text)
- 種々の鍵に対する既知入出力試験 (KAT-Key)
- マルチブロックメッセージ試験 (MMT)
- モンテカルロ試験 (MCT)
- 逆転置既知入出力試験 (KAT-IP)
- 置換既知入出力試験 (KAT-PO)
- Sbox 既知入出力試験 (KAT-ST)

他のブロック暗号

- 種々の平文 (暗号文) に対する既知入出力試験 (KAT-Text)
- 種々の鍵に対する既知入出力試験 (KAT-Key)
- マルチブロックメッセージ試験 (MMT)
- モンテカルロ試験 (MCT)
- Sbox 既知入出力試験 (KAT-Sbox)

AES の試験項目は AESAVS[1]，3-key Triple DES の試験項目は TMOVS[4] に準拠する．ただし，各検定項目のパラメータ (MCT のループ数など) は別に定める規定値とする．また，AES と 3-key Triple DES に対する各既知入出力試験用のテストベクタは，それぞれ AESAVS および TMOVS に記述されているものを使用する．

AES と 3-key Triple DES 以外の暗号モジュールに対する試験項目は，AESAVS[1] に準拠する．

ブロック暗号の試験項目の詳細を以下に記述する．

6.1 種々の平文 (暗号文) に対する既知入出力試験 (KAT-Text)

KAT-Text では，平文 (または暗号文) を様々に変化させ，暗号文 (または平文) が期待値と一致するかどうかを試験する．

暗号化機能に対する入力平文 (または IV，カウンタ) には，次のように 1 ビットずつ変化させたデータを用いる．

AESAVS 記載の KAT-Text 用入力平文 (または IV，カウンタ):

```
0x80000000 00000000 00000000 00000000
0xc0000000 00000000 00000000 00000000
0xe0000000 00000000 00000000 00000000
0xf0000000 00000000 00000000 00000000
0xf8000000 00000000 00000000 00000000
0xfc000000 00000000 00000000 00000000
0xfe000000 00000000 00000000 00000000
0xff000000 00000000 00000000 00000000
0xff800000 00000000 00000000 00000000
0xffc00000 00000000 00000000 00000000
0xffe00000 00000000 00000000 00000000
0xfff00000 00000000 00000000 00000000
...
0xffffffff ffffffff ffffffff fffffffe
0xffffffff ffffffff ffffffff ffffffff
```

TMOVS 記載の KAT-Text 用入力平文 (または IV，カウンタ):

```
0x80000000 00000000
0x40000000 00000000
```

0x20000000 00000000
0x10000000 00000000
0x08000000 00000000
0x04000000 00000000
0x02000000 00000000
0x01000000 00000000
0x00800000 00000000
0x00400000 00000000
0x00200000 00000000
0x00100000 00000000
...
0x00000000 00000002
0x00000000 00000001

各モードにおけるテストベクタを表 4，5，6 に示す．CTR モードは，OFB モードと同様とした．また，AESAVS において復号機能に対する KAT-Text では，入力暗号文として用いる暗号文は，KAT-Text 暗号化の際の期待値暗号文である．

表 1: AESAVS のテストベクタ：KAT-Text 暗号化

モード	平文	IV またはカウンタ	鍵
ECB	上記テストベクタ	—	全ゼロ
CBC	上記テストベクタ	全ゼロ	全ゼロ
CFB128	全ゼロ	上記テストベクタ	全ゼロ
CFB1	全ゼロ	上記テストベクタ	全ゼロ
CFB8	全ゼロ	上記テストベクタ	全ゼロ
OFB	全ゼロ	上記テストベクタ	全ゼロ
CTR	全ゼロ	上記テストベクタ	全ゼロ

表 2: TMOVS テストベクタ：KAT-Text 暗号化

モード	平文	IV またはカウンタ	鍵
ECB	上記テストベクタ	—	0101 ··· 01(奇数パリティ)
CBC	上記テストベクタ	全ゼロ	0101 ··· 01(奇数パリティ)
CFB64	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
CFB1	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
CFB8	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
OFB	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
CTR	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)

表 3: TMOVS テストベクタ：KAT-Text 復号

モード	暗号文	IV またはカウンタ	鍵
ECB	KAT-Text 暗号化の際の暗号文	—	0101 ··· 01(奇数パリティ)
CBC	KAT-Text 暗号化の際の暗号文	全ゼロ	0101 ··· 01(奇数パリティ)
CFB64	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
CFB1	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
CFB8	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
OFB	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)
CTR	全ゼロ	上記テストベクタ	0101 ··· 01(奇数パリティ)

6.2 種々の鍵に対する既知入出力試験 (KAT-Key)

KAT-Key では，鍵を様々に変化させ，暗号文 (または平文) が期待値と一致するかどうかを試験する．

暗号化機能に対する鍵には，次のように 1 ビットずつ変化させたデータを用いる．

AESAVS 記載の KAT-Key 用の入力鍵:

```
0x80000000 00000000 00000000 00000000
0xc0000000 00000000 00000000 00000000
0xe0000000 00000000 00000000 00000000
0xf0000000 00000000 00000000 00000000
0xf8000000 00000000 00000000 00000000
0xfc000000 00000000 00000000 00000000
0xfe000000 00000000 00000000 00000000
0xff000000 00000000 00000000 00000000
0xff800000 00000000 00000000 00000000
0xffc00000 00000000 00000000 00000000
0xffe00000 00000000 00000000 00000000
0xfff00000 00000000 00000000 00000000
...
0xffffffff ffffffff ffffffff fffffffe
0xffffffff ffffffff ffffffff ffffffff
```

TMOVS KAT-Key の鍵:

```
0x8001010101010101 8001010101010101 8001010101010101
0x4001010101010101 4001010101010101 4001010101010101
0x2001010101010101 2001010101010101 2001010101010101
0x1001010101010101 1001010101010101 1001010101010101
0x0801010101010101 0801010101010101 0801010101010101
0x0401010101010101 0401010101010101 0401010101010101
0x0201010101010101 0201010101010101 0201010101010101
0x0180010101010101 0180010101010101 0180010101010101
0x0140010101010101 0140010101010101 0140010101010101
```

```
0x0120010101010101 0120010101010101 0120010101010101
0x0110010101010101 0110010101010101 0110010101010101
0x0108010101010101 0108010101010101 0108010101010101
...
0x0101010101010104 0101010101010104 0101010101010104
0x0101010101010102 0101010101010102 0101010101010102
```

各モードにおけるテストベクタを表 4，5，6 に示す．CTR モードは，OFB モードと同様とした．また，AESAVS において復号機能に対する KAT-Key では，入力暗号文として用いる暗号文は，KAT-Key 暗号化の際の期待値暗号文である．

表 4: AESAVS テストベクタ：KAT-Key 暗号化

モード	平文	IV またはカウンタ	鍵
ECB	全ゼロ	—	上記テストベクタ
CBC	全ゼロ	全ゼロ	上記テストベクタ
CFB128	全ゼロ	全ゼロ	上記テストベクタ
CFB1	全ゼロ	全ゼロ	上記テストベクタ
CFB8	全ゼロ	全ゼロ	上記テストベクタ
OFB	全ゼロ	全ゼロ	上記テストベクタ
CTR	全ゼロ	全ゼロ	上記テストベクタ

表 5: TMOVS テストベクタ：KAT-Key 暗号化

モード	平文	IV またはカウンタ	鍵
ECB	全ゼロ	—	上記テストベクタ
CBC	全ゼロ	全ゼロ	上記テストベクタ
CFB64	全ゼロ	全ゼロ	上記テストベクタ
CFB1	全ゼロ	全ゼロ	上記テストベクタ
CFB8	全ゼロ	全ゼロ	上記テストベクタ
OFB	全ゼロ	全ゼロ	上記テストベクタ
CTR	全ゼロ	全ゼロ	上記テストベクタ

6.3 マルチブロックメッセージ試験 (MMT)

MMT ではランダムに与えられた複数ブロック分の平文 (または暗号文)，IV，カウンタに対する暗号文 (または平文) が期待値と一致するかどうかで検証を行う．ただし，MMT ブロック数は別に定める規定値とする．

6.4 モンテカルロ試験 (MCT)

MCT では，ランダムに与えられた 1 ブロック分の初期平文 (または暗号文)，初期鍵，初期 IV に対して，次のアルゴリズムで計算される暗号文 (または平文)`CT[i]` が期待値と一致するかどう

表 6: TMOVS テストベクタ：KAT-Key 復号

モード	暗号文	IV またはカウンタ	鍵
ECB	KAT-Key 暗号化の際の暗号文	—	上記テストベクタ
CBC	KAT-Key 暗号化の際の暗号文	全ゼロ	上記テストベクタ
CFB64	全ゼロ	全ゼロ	上記テストベクタ
CFB1	全ゼロ	全ゼロ	上記テストベクタ
CFB8	全ゼロ	全ゼロ	上記テストベクタ
OFB	全ゼロ	全ゼロ	上記テストベクタ
CTR	全ゼロ	全ゼロ	上記テストベクタ

かで検証を行う．ただし，内側ループ回数 innerloop および外側ループ回数 outerloop は別途定める規定値とする．

なお，3-key Triple DES に対する MCT は第 6.4.3 節に記述する．

6.4.1 64 ビットブロック暗号に対する MCT

6.4.1.1 ECB モード

ECB モード暗号化

```
Key[0] = Key // 初期鍵
PT = PT_0    // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], PT) // ECB モード暗号化
    PT = CT[j]
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
}
```

ECB モード復号

```
Key[0] = Key // 初期鍵
CT = CT_0    // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
```

```
  {
    PT[j] = Decryption(Key[i], CT) // ECB モード復号
    CT = PT[j]
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文


  Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])
}
```

6.4.1.2 CBC モード

CBC モード暗号化

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
PT = PT_0    // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // CBC モード暗号化
    If( j==0 )
    {
      PT = IV
    }
    else
    {
      PT = CT[j-1]
    }
    IV = CT[j]
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文


  Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
}
```

CBC モード復号

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
CT = CT_0    // 初期暗号文
```

```
for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // CBC モード復号
    If( j==0 )
    {
      tmp = CT
      CT = IV
      IV = tmp
    }
    else
    {
      IV = CT
      CT = PT[j-1]
    }
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文


  Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])

  IV = PT[innerloop-1]
}
```

6.4.1.3 CFB モード

CFB モード暗号化

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
PT = PT_0    // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // CFB モード暗号化
    If( j==0 )
    {
      PT = IV
    }
    else
```

```
    {
      PT = CT[j-1]
    }
    IV = CT[j]
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文


  Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
}
```

CFB モード復号

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
CT = CT_0    // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // CFB モード復号
    If( j==0 )
    {
      tmp = IV
      IV = CT
      CT = tmp
    }
    else
    {
      IV = CT
      CT = PT[j-1]
    }
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])

  IV = PT[innerloop-1]
}
```

6.4.1.4 OFB モード

OFB モード暗号化

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
PT = PT_0    // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // OFB モード暗号化
    If( j==0 )
    {
      PT = IV
    }
    else
    {
      PT = CT[j-1]
    }
    IV = O[j] // O[j]=Encryption() 内で平文を xor する直前の値
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])

  IV = CT[innerloop-1]
}
```

OFB モード復号

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
CT = CT_0    // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // OFB モード復号
    If( j==0 )
    {
      CT = IV
    }
    else
    {
      CT = PT[j-1]
```

```
    }
    IV = O[j] // O[j]=Decryption() 内で暗号文を xor する直前の値
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])

  IV = PT[innerloop-1]
}
```

6.4.1.5 CTR モード

CTR モード暗号化

```
Key[0] = Key     // 初期鍵
CTR = CTR_0      // 初期カウンタ
PT = PT_0        // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], CTR, PT) // CTR モード暗号化
    CRT = (CTR + 1) mod 2^64
    PT = CT[j]
  }

  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
}
```

CTR モード復号

```
Key[0] = Key     // 初期鍵
CTR = CTR_0      // 初期カウンタ
CT = CT_0        // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], CTR, CT) // CTR モード復号
    CRT = (CTR + 1) mod 2^64
```

```
    CT = PT[j]
  }

  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])
}
```

6.4.2 128 ビットブロック暗号に対する MCT

6.4.2.1 ECB モード

ECB モード暗号化

```
Key[0] = Key // 初期鍵
PT = PT_0    // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], PT) // ECB モード暗号化
    PT = CT[j]
  }
  Output CT[innerloop-1]

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of CT[innerloop-2] || CT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
  }
}
```

ECB モード復号

```
Key[0] = Key // 初期鍵
CT = CT_0    // 初期暗号文
```

```
for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], CT) // ECB モード復号
    CT = PT[j]
  }
  Output PT[innerloop-1]

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of PT[innerloop-2] || PT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])
  }
}
```

6.4.2.2 CBC モード

CBC モード暗号化

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
PT = PT_0    // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // CBC モード暗号化
    If( j==0 )
    {
      PT = IV
    }
    else
    {
      PT = CT[j-1]
```

```
    }
    IV = CT[j]
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of CT[innerloop-2] || CT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
  }
}
```

CBC モード復号

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
CT = CT_0    // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // CBC モード復号
    If( j==0 )
    {
      tmp = CT
      CT = IV
      IV = tmp
    }
    else
    {
      IV = CT
      CT = PT[j-1]
    }
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文
```

```
  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of PT[innerloop-2] || PT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])
  }

  IV = PT[innerloop-1] // 内側ループで計算された最後の平文
}
```

6.4.2.3 CFB モード

CFB モード暗号化

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV(128 ビット)
PT = PT_0    // 初期平文 (128 ビット)

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // CFB モード暗号化
    If( j==0 )
    {
      PT = IV
    }
    else
    {
      PT = CT[j-1]
    }
    IV = CT[j]
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-1]
```

```
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of CT[innerloop-2] || CT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
  }
  IV = CT[innerloop-1]
}
```

CFB モード復号

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV(128 ビット)
CT = CT_0    // 初期暗号文 (128 ビット)

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // CFB モード復号
    If( j==0 )
    {
      tmp = IV
      IV = CT
      CT = tmp
    }
    else
    {
      IV = CT
      CT = PT[j-1]
    }
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of PT[innerloop-2] || PT[innerloop-1])
```

```
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])
  }
  IV = PT[innerloop-1]
}
```

6.4.2.4 CFB-1 モード

CFB-1 モード暗号化

```
Key[0] = Key // 初期鍵
IV[0] = IV_0 // 初期 IV(128 ビット)
PT = PT_0    // 初期平文 (1 ビット)

for (i=0; i<outerloop; i++)
{
  IV = IV[i]
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // CFB-1 モード暗号化
    IV = IV の右 127 ビット||CT[j]
    if( j<128 )
    {
      PT = IV[i] の左 j 番目 1 ビット
    }
    else
    {
      PT = CT[j-128]
    }
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-128]||CT[innerloop-127]||...||CT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-192]||CT[innerloop-191]||...||CT[innerloop-1]
  }
  If ( 鍵長==256 ビット )
  {
```

```
    Key[i+1] = Key[i] xor CT[innerloop-256]||CT[innerloop-255]||...||CT[innerloop-1]
  }
  IV[i+1] = CT[innerloop-128] || CT[innerloop-127] ||...|| CT[innerloop-1]
}
```

CFB-1 モード復号

```
Key[0] = Key // 初期鍵
IV[0] = IV_0 // 初期 IV(128 ビット)
CT = CT_0    // 初期暗号文 (1 ビット)

for (i=0; i<outerloop; i++)
{
  IV = IV[i]
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // CFB-1 モード復号
    IV = IV の右 127 ビット||CT
    if( j<128 )
    {
      CT = IV[i] の左 j 番目 1 ビット
    }
    else
    {
      CT = PT[j-128]
    }
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-128]||PT[innerloop-127]||...||PT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-192]||PT[innerloop-191]||...||PT[innerloop-1]
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-256]||PT[innerloop-255]||...||PT[innerloop-1]
  }
  IV[i+1] = PT[innerloop-128] || PT[innerloop-127] ||...|| PT[innerloop-1]
}
```

6.4.2.5 CFB-8 モード

CFB-8 モード暗号化

```
Key[0] = Key // 初期鍵
IV[0] = IV_0 // 初期 IV(128 ビット)
PT = PT_0    // 初期平文 (8 ビット)

for (i=0; i<outerloop; i++)
{
  IV = IV[i]
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // CFB-8 モード暗号化
    IV = IV の右 120 ビット||CT[j]
    if( j<16 )
    {
      PT = IV[i] の左 j 番目 1 バイト
    }
    else
    {
      PT = CT[j-16]
    }
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-16]||CT[innerloop-15]||...||CT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-24]||CT[innerloop-23]||...||CT[innerloop-1]
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-32]||CT[innerloop-31]||...||CT[innerloop-1]
  }
  IV[i+1] = CT[innerloop-16] || CT[innerloop-15] ||...|| CT[innerloop-1]
}
```

CFB-8 モード復号

```
Key[0] = Key // 初期鍵
```

```
IV[0] = IV_0 // 初期 IV(128 ビット)
CT = CT_0    // 初期暗号文 (8 ビット)

for (i=0; i<outerloop; i++)
{
  IV = IV[i]
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // CFB-8 モード復号
    IV = IV の右 120 ビット||CT
    if( j<16 )
    {
      CT = IV[i] の左 j 番目 1 バイト
    }
    else
    {
      CT = PT[j-16]
    }
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-16]||PT[innerloop-15]||...||PT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-24]||PT[innerloop-23]||...||PT[innerloop-1]
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-32]||PT[innerloop-31]||...||PT[innerloop-1]
  }
  IV[i+1] = PT[innerloop-16] || PT[innerloop-15] ||...|| PT[innerloop-1]
}
```

6.4.2.6 OFB モード

OFB モード暗号化

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
PT = PT_0    // 初期平文
```

```
for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], IV, PT) // OFB モード暗号化
    If( j==0 )
    {
      PT = IV
    }
    else
    {
      PT = CT[j-1]
    }
    IV = O[j] // O[j]=Encryption() 内で平文を xor する直前の値
  }
  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of CT[innerloop-2] || CT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
  }
  IV = CT[innerloop-1]
}
```

OFB モード復号

```
Key[0] = Key // 初期鍵
IV = IV_0    // 初期 IV
CT = CT_0    // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], IV, CT) // OFB モード復号
    If( j==0 )
```

```
    {
      CT = IV
    }
    else
    {
      CT = PT[j-1]
    }
    IV = O[j] // O[j]=Decryption() 内で暗号文を xor する直前の値
  }
  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of PT[innerloop-2] || PT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])
  }
  IV = PT[innerloop-1]
}
```

6.4.2.7 CTR モード

CTR モード暗号化

```
Key[0] = Key     // 初期鍵
CTR = CTR_0      // 初期カウンタ
PT = PT_0        // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key[i], CTR, PT) // CTR モード暗号化
    CTR = (CTR + 1) mod 2^128
    PT = CT[j]
  }

  Output CT[innerloop-1] // 内側ループで計算された最後の暗号文
```

```
  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor CT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of CT[innerloop-2] || CT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (CT[innerloop-2] || CT[innerloop-1])
  }
}
```

CTR モード復号

```
Key[0] = Key     // 初期鍵
CTR = CTR_0      // 初期カウンタ
CT = CT_0        // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key[i], CTR, CT) // CTR モード復号
    CTR = (CTR + 1) mod 2^128
    CT = PT[j]
  }

  Output PT[innerloop-1] // 内側ループで計算された最後の平文

  If ( 鍵長==128 ビット )
  {
    Key[i+1] = Key[i] xor PT[innerloop-1]
  }
  If ( 鍵長==192 ビット )
  {
    Key[i+1] = Key[i] xor (last 64-bits of PT[innerloop-2] || PT[innerloop-1])
  }
  If ( 鍵長==256 ビット )
  {
    Key[i+1] = Key[i] xor (PT[innerloop-2] || PT[innerloop-1])
  }
```

```
}
```

6.4.3 3-key Triple DES に対する MCT

3-key Triple DES に対する MCT を以下に記述する．TMOVS に記述されているアルゴリズムと同じである．ただし，CTR モード (インクリメンタルカウンタ) は TMOVS に記述されていないため，ECB ードに対する MCT に準拠した．なお，TMOVS に記述されているループ回数の規定値は，innerloop=10,000，outerloop=400 である．

6.4.3.1 ECB モード

ECB モード暗号化

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
PT = PT_0      // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key1[i], Key2[i], Key3[i], PT) // ECB モード暗号化
    PT = CT[j]
  }
  Output CT[innerloop-1] // 内側ループ内で計算された最後の暗号文

  Key1[i+1] = Key1[i] xor CT[innerloop-1]
  Key2[i+1] = Key2[i] xor CT[innerloop-2]
  Key3[i+1] = Key3[i] xor CT[innerloop-3]
}
```

ECB モード復号

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
CT = CT_0      // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key1[i], Key2[i], Key3[i], CT) // ECB モード復号
```

```
    CT = PT[j]
  }
  Output PT[innerloop-1] // 内側ループ内で計算された最後の平文

  Key1[i+1] = Key1[i] xor PT[innerloop-1]
  Key2[i+1] = Key2[i] xor PT[innerloop-2]
  Key3[i+1] = Key3[i] xor PT[innerloop-3]
}
```

6.4.3.2 CBC モード

CBC モード暗号化

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV
PT = PT_0      // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key1[i], Key2[i], Key3[i], IV, PT) // CBC モード暗号化
    If( i==0 )
    {
      PT = IV
    }
    else
    {
      PT = CT[j-1]
    }
    IV = CT[j]
  }
  Output CT[innerloop-1] // 内側ループ内で計算された最後の暗号文

  Key1[i+1] = Key1[i] xor CT[innerloop-1]
  Key2[i+1] = Key2[i] xor CT[innerloop-2]
  Key3[i+1] = Key3[i] xor CT[innerloop-3]
}
```

CBC モード復号

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
```

```
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV
CT = CT_0      // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key1[i], Key2[i], Key3[i], IV, CT) // CBC モード復号
    IV = CT
    CT = PT[j]
  }
  Output PT[innerloop-1] // 内側ループ内で計算された最後の平文

  Key1[i+1] = Key1[i] xor PT[innerloop-1]
  Key2[i+1] = Key2[i] xor PT[innerloop-2]
  Key3[i+1] = Key3[i] xor PT[innerloop-3]
}
```

6.4.3.3 CFB モード

CFB モード暗号化

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV(64 ビット)
PT = PT_0      // 初期平文 (64 ビット)

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key1[i], Key2[i], Key3[i], IV, PT) // CFB モード暗号化
    PT = IV
    IV = CT[j]
  }
  Output CT[innerloop-1] // 内側ループ内で計算された最後の暗号文

  Key1[i+1] = Key1[i] xor CT[innerloop-1]
  Key2[i+1] = Key2[i] xor CT[innerloop-2]
  Key3[i+1] = Key3[i] xor CT[innerloop-3]
}
```

CFB モード復号

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV(64 ビット)
CT = CT_0      // 初期暗号文 (64 ビット)

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key1[i], Key2[i], Key3[i], IV, CT) // CFB モード復号
    IV = CT
    CT = O[j] // O[j]=Decryption() 内で暗号文を xor する直前の値
  }
  Output PT[innerloop-1] // 内側ループ内で計算された最後の平文

  Key1[i+1] = Key1[i] xor PT[innerloop-1]
  Key2[i+1] = Key2[i] xor PT[innerloop-2]
  Key3[i+1] = Key3[i] xor PT[innerloop-3]
}
```

6.4.3.4 CFB-1 モード

CFB-1 モード暗号化

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV(64 ビット)
PT = PT_0      // 初期平文 (1 ビット)

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key1[i], Key2[i], Key3[i], IV, PT) // CFB-1 モード暗号化
    PT = IV の左 1 ビット
    IV = IV の右 63 ビット||CT[j]
  }
  Output CT[innerloop-1] // 内側ループ内で計算された最後の暗号文

  // 暗号文を連結して 192 ビットにする
```

```
  C = CT[innerloop-192]||...||CT[innerloop-2]||CT[innerloop-1]

  Key1[i+1] = Key1[i] xor "C の右 64 ビット"
  Key2[i+1] = Key2[i] xor "C の中 64 ビット"
  Key3[i+1] = Key3[i] xor "C の左 64 ビット"
}
```

CFB-1 モード復号

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV(64 ビット)
CT = CT_0      // 初期暗号文 (1 ビット)

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key1[i], Key2[i], Key3[i], IV, CT) // CFB-1 モード復号
    IV = IV の右 63 ビット||CT
    CT = O[j] の左 1 ビット // O[j]=Decryption() 内で暗号文を xor する直前の値
  }
  Output PT[innerloop-1] // 内側ループ内で計算された最後の平文

  // 平文を連結して 192 ビットにする
  P = PT[innerloop-192]||...||PT[innerloop-2]||PT[innerloop-1]

  Key1[i+1] = Key1[i] xor "P の右 64 ビット"
  Key2[i+1] = Key2[i] xor "P の中 64 ビット"
  Key3[i+1] = Key3[i] xor "P の左 64 ビット"
}
```

6.4.3.5 CFB-8 モード

CFB-8 モード暗号化

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV(64 ビット)
PT = PT_0      // 初期平文 (8 ビット)

for (i=0; i<outerloop; i++)
```

```
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key1[i], Key2[i], Key3[i], IV, PT) // CFB-8 モード暗号化
    PT = IV の左 8 ビット
    IV = IV の右 56 ビット||CT[j]
  }
  Output CT[innerloop-1] // 内側ループ内で計算された最後の暗号文

  // 暗号文を連結して 192 ビットにする
  C = CT[innerloop-24]||...||CT[innerloop-2]||CT[innerloop-1]

  Key1[i+1] = Key1[i] xor "C の右 64 ビット"
  Key2[i+1] = Key2[i] xor "C の中 64 ビット"
  Key3[i+1] = Key3[i] xor "C の左 64 ビット"
}
```

CFB-8 モード復号

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV(64 ビット)
CT = CT_0      // 初期暗号文 (8 ビット)

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key1[i], Key2[i], Key3[i], IV, CT) // CFB-8 モード復号
    IV = IV の右 56 ビット||CT
    CT = O[j] の左 8 ビット // O[j]=Decryption() 内で暗号文を xor する直前の値
  }
  Output PT[innerloop-1] // 内側ループ内で計算された最後の平文

  // 平文を連結して 192 ビットにする
  P = PT[innerloop-24]|||...|PT[innerloop-2]||PT[innerloop-1]

  Key1[i+1] = Key1[i] xor "P の右 64 ビット"
  Key2[i+1] = Key2[i] xor "P の中 64 ビット"
  Key3[i+1] = Key3[i] xor "P の左 64 ビット"
}
```

6.4.3.6 OFB モード

OFB モード暗号化

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV
PT = PT_0      // 初期平文

for (i=0; i<outerloop; i++)
{
  INIT_PT = PT
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key1[i], Key2[i], Key3[i], IV, PT) // OFB モード暗号化
    PT = IV
    IV = O[j] // O[j]=Encryption() 内で平文を xor する直前の値
  }
  Output CT[innerloop-1] // 内側ループ内で計算された最後の暗号文

  Key1[i+1] = Key1[i] xor CT[innerloop-1]
  Key2[i+1] = Key2[i] xor CT[innerloop-2]
  Key3[i+1] = Key3[i] xor CT[innerloop-3]

  PT = PT xor INIT_PT
}
```

OFB モード復号

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
IV = IV_0      // 初期 IV
CT = CT_0      // 初期暗号文

for (i=0; i<outerloop; i++)
{
  INIT_CT = CT
  for (j=0; j<innerloop; j++)
  {
    PT[j] = Decryption(Key1[i], Key2[i], Key3[i], IV, CT) // OFB モード復号
    CT = IV
    IV = O[j] // O[j]=Decryption() 内で暗号文を xor する直前の値
  }
  Output PT[innerloop-1] // 内側ループ内で計算された最後の平文
```

```
  Key1[i+1] = Key1[i] xor PT[innerloop-1]
  Key2[i+1] = Key2[i] xor PT[innerloop-2]
  Key3[i+1] = Key3[i] xor PT[innerloop-3]

  CT = CT xor INIT_CT
}
```

6.4.3.7 CTR モード

CTR モード暗号化

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
CTR = CTR_0    // 初期カウンタ
PT = PT_0      // 初期平文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
  {
    CT[j] = Encryption(Key1[i], Key2[i], Key3[i], CTR, PT) // CTR モード暗号化
    CTR = (CTR + 1) mod 2^64
    PT = CT[j]
  }
  Output CT[innerloop-1] // 内側ループ内で計算された最後の暗号文

  Key1[i+1] = Key1[i] xor CT[innerloop-1]
  Key2[i+1] = Key2[i] xor CT[innerloop-2]
  Key3[i+1] = Key3[i] xor CT[innerloop-3]
}
```

CTR モード復号

```
Key1[0] = Key1 // 初期鍵 (64 ビット)
Key2[0] = Key2 // 初期鍵 (64 ビット)
Key3[0] = Key3 // 初期鍵 (64 ビット)
CTR = CTR_0    // 初期カウンタ
CT = CT_0      // 初期暗号文

for (i=0; i<outerloop; i++)
{
  for (j=0; j<innerloop; j++)
```

```
  {
    PT[j] = Decryption(Key1[i], Key2[i], Key3[i], CTR, CT) // CTR モード復号
    CTR = (CTR + 1) mod 2^64
    CT = PT[j]
  }
  Output PT[innerloop-1] // 内側ループ内で計算された最後の平文

  Key1[i+1] = Key1[i] xor PT[innerloop-1]
  Key2[i+1] = Key2[i] xor PT[innerloop-2]
  Key3[i+1] = Key3[i] xor PT[innerloop-3]
}
```

6.5 Sbox 既知入出力試験 (KAT-Sbox)

KAT-Sbox は，各暗号アルゴリズムの Sbox テーブルのエントリーを全て 1 回以上引くように作成された入力データ (平文，暗号文，鍵，IV，カウンタ) を用いる検証方法である．入力データは，各ブロック暗号ごとに作成されたものを用いる．

AES に対する KAT-GFSbox と KAT-KeySbox の入力データは，AESAVS[1] に記載の入力データを用いる．

3-key Triple DES に対する KAT-ST の入力データは，TMOVS[4] に記載の入力データを用いる．

6.6 3-key Triple DES に対するその他の KAT

3-key Triple DES に対する次の KAT の入力データは，TMOVS[4] に記載の入力データを用いる．

- 逆転置既知入出力試験 (KAT-IP)
- 置換既知入出力試験 (KAT-PO)

7　ストリーム暗号

ストリーム暗号アルゴリズム，MUGI，MULTI-S01 128-bit RC4，の暗号モジュール試験項目を記述する．平文を入力としてとるストリーム暗号アルゴリズム 128-bit RC4 および MULTI-S01 と異なり，MUGI は擬似乱数生成アルゴリズムである．すなわち，MUGI は平文を入力としない．

7.1　MUGI

MUGI の試験対象機能は，

- 擬似乱数生成機能

のみである．試験項目は次の通り．

- 種々の鍵に対する既知入出力試験 (KAT-Key)
- モンテカルロ試験 (MCT)

KAT のテストベクトルは，AESAVS に記載されているものに準拠する．また，MCT は，AESAVS に記述されたアルゴリズムに準拠する．詳細を以下に記述する．

7.1.1　種々の鍵に対する既知入出力試験 (KAT-Key)

128 ビット鍵 ECB モード AES に対する KAT-Key と同じ KAT-Key を行う．ユニット数は別途定める規定値とする．

7.1.2　モンテカルロ試験 (MCT)

MCT では，ランダムに与えられた初期鍵および初期ベクトルに対して，次のアルゴリズムで計算される乱数 `CT[i]` が期待値と一致するかどうかで検証を行う．ループ回数 `innerloop` および `outerloop` は別途定める規定値とする．

MUGI に対する MCT

```
Key[0] = Key // 初期鍵 128 ビット
I[0] = I     // 初期ベクトル
n = 2        // ユニット数
for (j=0; j<outerloop; j++)
{
  for (i=0; i<innerloop; i++ )
  {
    CT[i] = MUGI(Key[i], I[i], n)
    I[i+1] = CT[i]
  }

  Output CT[i-1]
```

```
  Key[0] = Key[i-1] xor CT[i-1]

  I[0] = CT[i-1]
}
```

7.2 MULTI-S01

MULTI-S01 の試験対象機能は次の 2 つである．

- 暗号化機能
- 復号機能

暗号化機能試験の試験項目は次の通りである．

- 種々の平文に対する既知入出力試験 (KAT-Text)
- 種々の鍵に対する既知入出力試験 (KAT-Key)
- モンテカルロ試験 (MCT)

各 KAT のテストベクトルは，AESAVS に記載されているものと同様とする (6.1 節参照)．また，MCT は，AESAVS に記述されたアルゴリズムに準拠する．

復号機能試験の試験項目は次の通りである．

- 種々の暗号文に対する既知入出力試験 (KAT-Text)
- 改竄された暗号文に対してメッセージ認証子不正を検出すること．

各試験項目の詳細を以下に記述する．

7.2.1 種々の平文 (暗号文) に対する既知入出力試験 (KAT-Text)

128 ビット鍵 ECB モード AES に対する KAT-Text と同様な KAT-Text を行う．また，入力平文 (暗号文) のビット数は別途定める規定値とする．

7.2.2 種々の鍵に対する既知入出力試験 (KAT-Key)

入力平文を 128 ビットとし，128 ビット鍵 ECB モード AES に対する KAT-Key と同様な KAT-Key を行う．

7.2.3 モンテカルロ試験 (MCT)

MCT では，ランダムに与えられた 128 ビットの初期平文，初期鍵，冗長性 (R)，初期値 (IV) に対して，次のアルゴリズムで計算される暗号文 CT[i] が期待値と一致するかどうかで検証を行う．ループ回数 innerloop および outerloop は別途定める規定値とする．

MULTI-S01 暗号化機能に対する MCT

```
Key[0] = Key // 初期鍵 256 ビット
PT[0] = PT   // 初期平文 128 ビット
for (j=0; j<outerloop; j++)
{
  for (i=0; i<innerloop; i++ )
  {
    CT[i] = MULTI-S01(Key[i], PT[i], R, IV) // 128 ビット分の暗号化
    PT[i+1] = CT[i]
  }

  Output CT[i-1]

  Key[0] = Key[i-1] xor CT[i-1]

  PT[0] = CT[i-1]
}
```

7.3 128-bit RC4

128-bit RC4 の試験対象機能は次の 2 つである．

- 暗号化機能
- 復号機能

暗号化機能，復号機能共に，試験項目は次の通りである．

- 種々の平文 (暗号文) に対する既知入出力試験 (KAT-Text)
- 種々の鍵に対する既知入出力試験 (KAT-Key)
- モンテカルロ試験 (MCT)

各 KAT のテストベクトルは，AESAVS に記載されているものと同様とする (6.1 節参照)．また，MCT は，AESAVS に記述されたアルゴリズムに準拠する．詳細を以下に記述する．

7.3.1 種々の平文 (暗号文) に対する既知入出力試験 (KAT-Text)

128 ビット鍵 ECB モード AES に対する KAT-Text と同じ KAT-Text を行う．入力平文 (暗号文) のビット数は別途定める規定値とする．

7.3.2 種々の鍵に対する既知入出力試験 (KAT-Key)

入力平文 (暗号文) を 128 ビットとし，128 ビット鍵 ECB モード AES に対する KAT-Key と同じ KAT-Key を行う.

7.3.3 モンテカルロ試験 (MCT)

MCT では，ランダムに与えられた 128 ビットの初期平文 (または暗号文)，初期鍵に対して，次のアルゴリズムで計算される暗号文 (または平文)CT[i] が期待値と一致するかどうかで検証を行う．ループ回数 innerloop および outerloop は別途定める規定値とする.

128-bit RC4 に対する MCT

```
Key[0] = Key // 初期鍵 128 ビット
PT[0] = PT   // 初期平文 128 ビット
for (j=0; j<outerloop; j++)
{
  for (i=0; i<innerloop; i++ )
  {
    CT[i] = RC4(Key[i], PT[i]) // 128 ビット分の暗号化 (または復号)
    Key[i+1] = key[i] xor CT[i]
    PT[i+1] = CT[i]
  }

  Output CT[i-1]

  Key[0] = Key[i-1] xor CT[i-1]

  PT[0] = CT[i-1]
}
```

8 ハッシュ関数

ハッシュ関数アルゴリズム，RIPEMD-160，SHA-1，SHA-224, SHA-256，SHA-384，SHA-512 の暗号モジュール試験項目を記述する．

また，本ツールの試験対象である上記 6 つのハッシュ関数は，ハッシュ値のサイズを除いて入出力は同じであるので，全てのハッシュ関数機能に対して試験項目は同じである．以下に試験項目を記述する．

- 短いメッセージに対する試験 (SMT)
- 選択された長いメッセージに対する試験 (SLMT)
- 擬似ランダムメッセージに対する試験 (PGMT)

ハッシュ関数モジュールが byte oriented 実装の場合の上記試験項目の詳細を以下に記述する．試験項目の詳細は SHAVS[2] に従っている．ただし，SLMT で行う試験におけるビット長は別に定める規定値とする．

8.1 短いメッセージに対する試験 (SMT)

ハッシュ関数のブロック長 (ビット数) を m とする．すなわち，

- RIPEMD-160: $m = 512$
- SHA-1: $m = 512$
- SHA-224: $m = 512$
- SHA-256: $m = 512$
- SHA-384: $m = 1,024$
- SHA-512: $m = 1,024$

となる．この試験項目では，$m/8 + 1$ 個のランダムに生成されたメッセージに対するハッシュ値に対する Known Answer Test を行う．各メッセージのビット長は $0, 8, 16, \ldots, m$ である．

8.2 選択された長いメッセージに対する試験 (SLMT)

第 8.1 節のように，ハッシュ関数のブロック長 (ビット数) を m とする．この試験項目では，ランダムに生成された $m/8$ 個の長いメッセージに対するハッシュ値に対する Known Answer Test を行う．各メッセージのビット長は，$m + 8 \times i \times$ (“Upperbound of SLMT” $- 1$)，$1 \leq i \leq m/8$, である．Upperbound of SLMT は別途定める規定値とする．

8.3 擬似ランダムメッセージに対する試験 (PGMT)

与えられた Seed，outerloop および innerloop から，次のアルゴリズムにより計算されるデータ MD[0]〜MD[outerloop-1] に対する Known Answer Test を行う．(注: 実際のプログラムでは配列 MD[ ] は outerloop 個の大きさを確保する必要はない)

```
for (j=0; j<outerloop; j++)
{
  MD[0] = Seed;
  MD[1] = Seed;
  MD[2] = Seed;
  for (i=3; i<innerloop+3; i++)
  {
    M[i] = MD[i-3]||MD[i-2]||MD[i-1]; // 記号||はデータの連結
    MD[i] = Hash(M[i]); // ハッシュ関数
  }
  MD[j] = MD[i-1];
  Seed  = MD[i-1];
  OUTPUT MD[j];
}
```

9　メッセージ認証

HMAC の暗号モジュール試験項目を記述する．HMAC の試験対象機能は次の通りである．

- メッセージ認証子生成機能

メッセージ認証機能の試験項目は次の通りである．

- 短いメッセージに対する試験 (SMT)
- 選択された長いメッセージに対する試験 (SLMT)
- 擬似ランダムメッセージに対する試験 (PGMT)

これらの試験方法は，第 8 節に記述した SMT, SLMT, PGMT とそれぞれ同じである．

10　擬似乱数生成関数

次の擬似乱数生成関数に対する試験項目を記述する．

- CTR_DRBG in ISO/IEC 18031
- OFB_DRBG in ISO/IEC 18031
- Hash_DRBG in ISO/IEC 18031
- PRNG based on SHA-1 in ANSI X9.42-2001 Annex C.1
- PRNG based on SHA-1 for general purpose in FIPS 186-2 (+ change notice 1) Appendix 3.1
- PRNG based on SHA-1 for general purpose in FIPS 186-2 (+ change notice 1) revised Appendix 3.1

試験項目はすべての擬似乱数生成関数について次の通りである．

- 種々のシードに対する試験
- モンテカルロ試験

これらは RNGVS[3] に従っている．以下に詳細を記述する．

10.1　種々のシードに対する試験

種々のシードに対する試験では，乱数シードを様々に変化させ，乱数値が期待値と一致するかどうかを試験する．乱数シード値は次のように変化させる．

種々のシードに対する試験の乱数シード値

```
0x80000000 00000000 ··· 00000000
0xc0000000 00000000 ··· 00000000
0xe0000000 00000000 ··· 00000000
0xf0000000 00000000 ··· 00000000
0xf8000000 00000000 ··· 00000000
0xfc000000 00000000 ··· 00000000
0xfe000000 00000000 ··· 00000000
0xff000000 00000000 ··· 00000000
0xff800000 00000000 ··· 00000000
0xffc00000 00000000 ··· 00000000
0xffe00000 00000000 ··· 00000000
0xfff00000 00000000 ··· 00000000
...
0xffffffff fffffffd ··· fffffffe
0xffffffff ffffffff ··· ffffffff
```

乱数シード以外の入力パラメータは全 0 とする．

10.2 モンテカルロ試験

モンテカルロ試験では，与えられた乱数シードに対して規定値10,000個の乱数列を生成し，10,000個目の乱数列に対して既知入出力試験を行う．

MCTのアルゴリズムを次に示す．なお，乱数シード以外の入力パラメータは全0とする．

```
Seed = 乱数シード
randLen = 乱数列1個あたりのビット長
for (i=0; i<loop; i++) // loopの規定値は10,000
{
  // 1ブロック分の乱数生成
  R[i] = PRNG(Seed, randLen)
  // PRNGの内部状態はPRNG()内で保持されているものとする．

}

Output R[i-1] // 最後の乱数randLenビット
```

参考文献

[1] L. E. Bassham III, “The Advanced Encryption Standard Algorithm Validation Suite (AESAVS),” NIST, 2002.

[2] L. E. Bassham III, “The secure hash algorithm validation system (SHAVS),” NIST, 2004.

[3] L. E. Bassham III, Shanon Keller, “The random number generator validation system (RNGVS),” NIST, 2005.

[4] NIST SP 800-20, “Modes of Operation Validation System for the Triple Data Encryption Algorithm (TMOVS) : Requirements and Procedures,” NIST, 2000.

[5] NIST SP 800-38A, “Recommendation for Block Cipher Modes of Operation : Methods and Techniques,” NIST, 2001.

[6] RSA Laboratories, “PKCS#1 v2.1: RSA Cryptography Standard, RSA Laboratories,” 2002.