Twofish Encryption Algorithm

Twofish Source Code

- **Kotlin code and unit test code (dv1 on Github)**

```
package info.nightscout.comboctl.base
/**
 * Implementation of the Two-Fish symmetric block cipher.
 *
 * This is based on the Java Twofish code from the Android Jobb tool:
 *
 *   https://android.googlesource.com/platform/tools/base/+/master/jobb/src/main/java/Twofish
 *
 * which in turn is based on the Twofish implementation from Bouncy Castle.
 *
 * The three public API functions of interest are [processKey], [blockEncrypt],
 * and [blockDecrypt]. Note that the latter two always process 16-byte blocks.
 */
object Twofish {
    /*********************
     * INTERNAL CONSTANTS *
     *********************/

    private const val BLOCK_SIZE = 16 // bytes in a data-block
    private const val MAX_ROUNDS = 16 // max # rounds (for allocating subkeys)

   // Subkey array indices
    private const val INPUT_WHITEN = 0
    private const val OUTPUT_WHITEN = INPUT_WHITEN + BLOCK_SIZE / 4
    private const val ROUND_SUBKEYS = OUTPUT_WHITEN + BLOCK_SIZE / 4 // 2*(#
rounds)

    private const val TOTAL_SUBKEYS = ROUND_SUBKEYS + 2 * MAX_ROUNDS

    private const val SK_STEP = 0x02020202
    private const val SK_BUMP = 0x01010101
    private const val SK_ROTL = 9

    // Fixed 8x8 permutation S-boxes
    private val P = arrayOf(
        intArrayOf(
            // p0
            0xA9, 0x67, 0xB3, 0xE8,
            0x04, 0xFD, 0xA3, 0x76,
            0x9A, 0x92, 0x80, 0x78,
            0xE4, 0xDD, 0xD1, 0x38,
```

0x0D, 0xC6, 0x35, 0x98,
0x18, 0xF7, 0xEC, 0x6C,
0x43, 0x75, 0x37, 0x26,
0xFA, 0x13, 0x94, 0x48,
0xF2, 0xD0, 0x8B, 0x30,
0x84, 0x54, 0xDF, 0x23,
0x19, 0x5B, 0x3D, 0x59,
0xF3, 0xAE, 0xA2, 0x82,
0x63, 0x01, 0x83, 0x2E,
0xD9, 0x51, 0x9B, 0x7C,
0xA6, 0xEB, 0xA5, 0xBE,
0x16, 0x0C, 0xE3, 0x61,
0xC0, 0x8C, 0x3A, 0xF5,
0x73, 0x2C, 0x25, 0x0B,
0xBB, 0x4E, 0x89, 0x6B,
0x53, 0x6A, 0xB4, 0xF1,
0xE1, 0xE6, 0xBD, 0x45,
0xE2, 0xF4, 0xB6, 0x66,
0xCC, 0x95, 0x03, 0x56,
0xD4, 0x1C, 0x1E, 0xD7,
0xFB, 0xC3, 0x8E, 0xB5,
0xE9, 0xCF, 0xBF, 0xBA,
0xEA, 0x77, 0x39, 0xAF,
0x33, 0xC9, 0x62, 0x71,
0x81, 0x79, 0x09, 0xAD,
0x24, 0xCD, 0xF9, 0xD8,
0xE5, 0xC5, 0xB9, 0x4D,
0x44, 0x08, 0x86, 0xE7,
0xA1, 0x1D, 0xAA, 0xED,
0x06, 0x70, 0xB2, 0xD2,
0x41, 0x7B, 0xA0, 0x11,
0x31, 0xC2, 0x27, 0x90,
0x20, 0xF6, 0x60, 0xFF,
0x96, 0x5C, 0xB1, 0xAB,
0x9E, 0x9C, 0x52, 0x1B,
0x5F, 0x93, 0x0A, 0xEF,
0x91, 0x85, 0x49, 0xEE,
0x2D, 0x4F, 0x8F, 0x3B,
0x47, 0x87, 0x6D, 0x46,
0xD6, 0x3E, 0x69, 0x64,
0x2A, 0xCE, 0xCB, 0x2F,
0xFC, 0x97, 0x05, 0x7A,
0xAC, 0x7F, 0xD5, 0x1A,
0x4B, 0x0E, 0xA7, 0x5A,
0x28, 0x14, 0x3F, 0x29,
0x88, 0x3C, 0x4C, 0x02,
0xB8, 0xDA, 0xB0, 0x17,
0x55, 0x1F, 0x8A, 0x7D,

```
        0x57, 0xC7, 0x8D, 0x74,
        0xB7, 0xC4, 0x9F, 0x72,
        0x7E, 0x15, 0x22, 0x12,
        0x58, 0x07, 0x99, 0x34,
        0x6E, 0x50, 0xDE, 0x68,
        0x65, 0xBC, 0xDB, 0xF8,
        0xC8, 0xA8, 0x2B, 0x40,
        0xDC, 0xFE, 0x32, 0xA4,
        0xCA, 0x10, 0x21, 0xF0,
        0xD3, 0x5D, 0x0F, 0x00,
        0x6F, 0x9D, 0x36, 0x42,
        0x4A, 0x5E, 0xC1, 0xE0
    ),
    intArrayOf(
        // p1
        0x75, 0xF3, 0xC6, 0xF4,
        0xDB, 0x7B, 0xFB, 0xC8,
        0x4A, 0xD3, 0xE6, 0x6B,
        0x45, 0x7D, 0xE8, 0x4B,
        0xD6, 0x32, 0xD8, 0xFD,
        0x37, 0x71, 0xF1, 0xE1,
        0x30, 0x0F, 0xF8, 0x1B,
        0x87, 0xFA, 0x06, 0x3F,
        0x5E, 0xBA, 0xAE, 0x5B,
        0x8A, 0x00, 0xBC, 0x9D,
        0x6D, 0xC1, 0xB1, 0x0E,
        0x80, 0x5D, 0xD2, 0xD5,
        0xA0, 0x84, 0x07, 0x14,
        0xB5, 0x90, 0x2C, 0xA3,
        0xB2, 0x73, 0x4C, 0x54,
        0x92, 0x74, 0x36, 0x51,
        0x38, 0xB0, 0xBD, 0x5A,
        0xFC, 0x60, 0x62, 0x96,
        0x6C, 0x42, 0xF7, 0x10,
        0x7C, 0x28, 0x27, 0x8C,
        0x13, 0x95, 0x9C, 0xC7,
        0x24, 0x46, 0x3B, 0x70,
        0xCA, 0xE3, 0x85, 0xCB,
        0x11, 0xD0, 0x93, 0xB8,
        0xA6, 0x83, 0x20, 0xFF,
        0x9F, 0x77, 0xC3, 0xCC,
        0x03, 0x6F, 0x08, 0xBF,
        0x40, 0xE7, 0x2B, 0xE2,
        0x79, 0x0C, 0xAA, 0x82,
        0x41, 0x3A, 0xEA, 0xB9,
        0xE4, 0x9A, 0xA4, 0x97,
        0x7E, 0xDA, 0x7A, 0x17,
        0x66, 0x94, 0xA1, 0x1D,
```

```
        0x3D, 0xF0, 0xDE, 0xB3,
        0x0B, 0x72, 0xA7, 0x1C,
        0xEF, 0xD1, 0x53, 0x3E,
        0x8F, 0x33, 0x26, 0x5F,
        0xEC, 0x76, 0x2A, 0x49,
        0x81, 0x88, 0xEE, 0x21,
        0xC4, 0x1A, 0xEB, 0xD9,
        0xC5, 0x39, 0x99, 0xCD,
        0xAD, 0x31, 0x8B, 0x01,
        0x18, 0x23, 0xDD, 0x1F,
        0x4E, 0x2D, 0xF9, 0x48,
        0x4F, 0xF2, 0x65, 0x8E,
        0x78, 0x5C, 0x58, 0x19,
        0x8D, 0xE5, 0x98, 0x57,
        0x67, 0x7F, 0x05, 0x64,
        0xAF, 0x63, 0xB6, 0xFE,
        0xF5, 0xB7, 0x3C, 0xA5,
        0xCE, 0xE9, 0x68, 0x44,
        0xE0, 0x4D, 0x43, 0x69,
        0x29, 0x2E, 0xAC, 0x15,
        0x59, 0xA8, 0x0A, 0x9E,
        0x6E, 0x47, 0xDF, 0x34,
        0x35, 0x6A, 0xCF, 0xDC,
        0x22, 0xC9, 0xC0, 0x9B,
        0x89, 0xD4, 0xED, 0xAB,
        0x12, 0xA2, 0x0D, 0x52,
        0xBB, 0x02, 0x2F, 0xA9,
        0xD7, 0x61, 0x1E, 0xB4,
        0x50, 0x04, 0xF6, 0xC2,
        0x16, 0x25, 0x86, 0x56,
        0x55, 0x09, 0xBE, 0x91
    )
)

// Define the fixed p0/p1 permutations used in keyed S-box lookup.
// By changing the following constant definitions, the S-boxes will
// automatically get changed in the Twofish engine.

private const val P_00 = 1
private const val P_01 = 0
private const val P_02 = 0
private const val P_03 = P_01 xor 1
private const val P_04 = 1

private const val P_10 = 0
private const val P_11 = 0
private const val P_12 = 1
private const val P_13 = P_11 xor 1
```

```
private const val P_14 = 0

private const val P_20 = 1
private const val P_21 = 1
private const val P_22 = 0
private const val P_23 = P_21 xor 1
private const val P_24 = 0

private const val P_30 = 0
private const val P_31 = 1
private const val P_32 = 1
private const val P_33 = P_31 xor 1
private const val P_34 = 1

// Primitive polynomial for GF(256)
private const val GF256_FDBK: Int = 0x169
private const val GF256_FDBK_2: Int = 0x169 / 2
private const val GF256_FDBK_4: Int = 0x169 / 4

private val MDS = Array(4) { IntArray(256) { 0 } }

private const val RS_GF_FDBK = 0x14D // field generator

/************************
 * INTERNAL FUNCTIONS *
 ************************/

private fun LFSR1(x: Int): Int =
    (x shr 1) xor (if ((x and 0x01) != 0) GF256_FDBK_2 else 0)

private fun LFSR2(x: Int): Int =
    (x shr 2) xor
    (if ((x and 0x02) != 0) GF256_FDBK_2 else 0) xor
    (if ((x and 0x01) != 0) GF256_FDBK_4 else 0)

private fun Mx_1(x: Int): Int = x
private fun Mx_X(x: Int): Int = x xor LFSR2(x) // 5B
private fun Mx_Y(x: Int): Int = x xor LFSR1(x) xor LFSR2(x) // EF

// Reed-Solomon code parameters: (12, 8) reversible code:<p>
// <pre>
//   g(x) = x**4 + (a + 1/a) x**3 + a x**2 + (a + 1/a) x + 1
// </pre>
// where a = primitive root of field generator 0x14D
private fun RS_rem(x: Int): Int {
    val b = (x ushr 24) and 0xFF
    val g2 = ((b shl 1) xor (if ((b and 0x80) != 0) RS_GF_FDBK else 0)) and 0xFF
    val g3 = (b ushr 1) xor (if ((b and 0x01) != 0) (RS_GF_FDBK ushr 1) else 0) xor g2
```

```
    return (x shl 8) xor (g3 shl 24) xor (g2 shl 16) xor (g3 shl 8) xor b
}

// Use (12, 8) Reed-Solomon code over GF(256) to produce a key S-box
// 32-bit entity from two key material 32-bit entities.
//
// @param  k0  1st 32-bit entity.
// @param  k1  2nd 32-bit entity.
// @return  Remainder polynomial generated using RS code
private fun RS_MDS_Encode(k0: Int, k1: Int): Int {
    var r = k1
    for (i in 0 until 4) // shift 1 byte at a time
        r = RS_rem(r)
    r = r xor k0
    for (i in 0 until 4)
        r = RS_rem(r)
    return r
}

private fun calcb0(x: Int) = x and 0xFF
private fun calcb1(x: Int) = (x ushr 8) and 0xFF
private fun calcb2(x: Int) = (x ushr 16) and 0xFF
private fun calcb3(x: Int) = (x ushr 24) and 0xFF

private fun F32(k64Cnt: Int, x: Int, k32: IntArray): Int {
    var b0 = calcb0(x)
    var b1 = calcb1(x)
    var b2 = calcb2(x)
    var b3 = calcb3(x)

    val k0 = k32[0]
    val k1 = k32[1]
    val k2 = k32[2]
    val k3 = k32[3]

    var k64Cnt2LSB = k64Cnt and 3

    if (k64Cnt2LSB == 1) {
        return MDS[0][(P[P_01][b0] and 0xFF) xor calcb0(k0)] xor
               MDS[1][(P[P_11][b1] and 0xFF) xor calcb1(k0)] xor
               MDS[2][(P[P_21][b2] and 0xFF) xor calcb2(k0)] xor
               MDS[3][(P[P_31][b3] and 0xFF) xor calcb3(k0)]
    }

    if (k64Cnt2LSB == 0) { // same as 4
        b0 = (P[P_04][b0] and 0xFF) xor calcb0(k3)
        b1 = (P[P_14][b1] and 0xFF) xor calcb1(k3)
        b2 = (P[P_24][b2] and 0xFF) xor calcb2(k3)
```

```
            b3 = (P[P_34][b3] and 0xFF) xor calcb3(k3)
            k64Cnt2LSB = 3
        }

        if (k64Cnt2LSB == 3) {
            b0 = (P[P_03][b0] and 0xFF) xor calcb0(k2)
            b1 = (P[P_13][b1] and 0xFF) xor calcb1(k2)
            b2 = (P[P_23][b2] and 0xFF) xor calcb2(k2)
            b3 = (P[P_33][b3] and 0xFF) xor calcb3(k2)
            k64Cnt2LSB = 2
        }

        if (k64Cnt2LSB == 2) { // 128-bit keys (optimize for this case)
            return MDS[0][(P[P_01][(P[P_02][b0] and 0xFF) xor calcb0(k1)] and 0xFF) xor
calcb0(k0)] xor
                    MDS[1][(P[P_11][(P[P_12][b1] and 0xFF) xor calcb1(k1)] and 0xFF) xor
calcb1(k0)] xor
                    MDS[2][(P[P_21][(P[P_22][b2] and 0xFF) xor calcb2(k1)] and 0xFF) xor
calcb2(k0)] xor
                    MDS[3][(P[P_31][(P[P_32][b3] and 0xFF) xor calcb3(k1)] and 0xFF) xor
calcb3(k0)]
        }

        return 0
    }

    private fun Fe32(sBox: IntArray, x: Int, R: Int) =
        sBox[0x000 + 2 * _b(x, R + 0) + 0] xor
        sBox[0x000 + 2 * _b(x, R + 1) + 1] xor
        sBox[0x200 + 2 * _b(x, R + 2) + 0] xor
        sBox[0x200 + 2 * _b(x, R + 3) + 1]

    private fun _b(x: Int, N: Int) =
        when (N and 3) {
            0 -> calcb0(x)
            1 -> calcb1(x)
            2 -> calcb2(x)
            3 -> calcb3(x)
            // NOTE: This else-branch is only here to shut up build errors.
            // This case cannot occur because the bitwise AND above excludes
            // all values outside of the 0-3 range.
            else -> 0
        }

    /*************************
     * STATIC INITIALIZATION *
     *************************/
```

```
init {
   // precompute the MDS matrix
   val m1 = IntArray(2)
   val mX = IntArray(2)
   val mY = IntArray(2)

   for (i in 0 until 256) {
      // compute all the matrix elements

      val j0 = P[0][i] and 0xFF
      m1[0] = j0
      mX[0] = Mx_X(j0) and 0xFF
      mY[0] = Mx_Y(j0) and 0xFF

      val j1 = P[1][i] and 0xFF
      m1[1] = j1
      mX[1] = Mx_X(j1) and 0xFF
      mY[1] = Mx_Y(j1) and 0xFF

      MDS[0][i] = (m1[P_00] shl 0) or
                  (mX[P_00] shl 8) or
                  (mY[P_00] shl 16) or
                  (mY[P_00] shl 24)

      MDS[1][i] = (mY[P_10] shl 0) or
                  (mY[P_10] shl 8) or
                  (mX[P_10] shl 16) or
                  (m1[P_10] shl 24)

      MDS[2][i] = (mX[P_20] shl 0) or
                  (mY[P_20] shl 8) or
                  (m1[P_20] shl 16) or
                  (mY[P_20] shl 24)

      MDS[3][i] = (mX[P_30] shl 0) or
                  (m1[P_30] shl 8) or
                  (mY[P_30] shl 16) or
                  (mX[P_30] shl 24)
   }
}

/*******************
 * KEY PROCESSING *
 *******************/

/**
 * Class containing precomputed S-box and subkey values derived from a key.
 *
```

```
 * These values are computed by the [processKey] function.
 * [blockEncrypt] and [blockDecrypt] expect an instance of this class,
 * not a key directly.
 */
data class KeyObject(val sBox: IntArray, val subKeys: IntArray) {
    override fun equals(other: Any?): Boolean {
        if (this === other) return true
        if (other == null) return false
        if (this::class != other::class) return false

        other as KeyObject

        if (!sBox.contentEquals(other.sBox)) return false
        if (!subKeys.contentEquals(other.subKeys)) return false

        return true
    }

    override fun hashCode(): Int {
        var result = sBox.contentHashCode()
        result = 31 * result + subKeys.contentHashCode()
        return result
    }
}

/**
 * Processes a Two-fish key and stores the computed values in the returned object.
 *
 * Since the S-box and subkey values stay the same during en/decryption, it
 * makes sense to compute them once and store them for later reuse. This is
 * what this function does
 *
 * @param key 64/128/192/256-bit key for processing.
 * @return Object with the processed results.
 */
fun processKey(key: ByteArray): KeyObject {
    require(key.size in intArrayOf(8, 16, 24, 32))

    val k64Cnt = key.size / 8
    val k32e = IntArray(4) // even 32-bit entities
    val k32o = IntArray(4) // odd 32-bit entities
    val sBoxKey = IntArray(4)

    var offset = 0

    // split user key material into even and odd 32-bit entities and
    // compute S-box keys using (12, 8) Reed-Solomon code over GF(256)
    for (i in 0 until 4) {
```

```
    if (offset >= key.size)
        break

    val j = k64Cnt - 1 - i

    k32e[i] = ((key[offset++].toPosInt() and 0xFF) shl 0) or
            ((key[offset++].toPosInt() and 0xFF) shl 8) or
            ((key[offset++].toPosInt() and 0xFF) shl 16) or
            ((key[offset++].toPosInt() and 0xFF) shl 24)
    k32o[i] = ((key[offset++].toPosInt() and 0xFF) shl 0) or
            ((key[offset++].toPosInt() and 0xFF) shl 8) or
            ((key[offset++].toPosInt() and 0xFF) shl 16) or
            ((key[offset++].toPosInt() and 0xFF) shl 24)
    sBoxKey[j] = RS_MDS_Encode(k32e[i], k32o[i]) // reverse order
}

// compute the round decryption subkeys for PHT. these same subkeys
// will be used in encryption but will be applied in reverse order.
var q = 0
val subKeys = IntArray(TOTAL_SUBKEYS)
for (i in 0 until (TOTAL_SUBKEYS / 2)) {
    var A = F32(k64Cnt, q, k32e) // A uses even key entities
    var B = F32(k64Cnt, q + SK_BUMP, k32o) // B uses odd  key entities
    B = (B shl 8) or (B ushr 24)
    A += B
    subKeys[2 * i + 0] = A // combine with a PHT
    A += B
    subKeys[2 * i + 1] = (A shl SK_ROTL) or (A ushr (32 - SK_ROTL))
    q += SK_STEP
}

// fully expand the table for speed
val k0 = sBoxKey[0]
val k1 = sBoxKey[1]
val k2 = sBoxKey[2]
val k3 = sBoxKey[3]
val sBox = IntArray(4 * 256)

for (i in 0 until 256) {
    var b0 = i
    var b1 = i
    var b2 = i
    var b3 = i

    var k64Cnt2LSB = k64Cnt and 3

    if (k64Cnt2LSB == 1) {
        sBox[0x000 + 2 * i + 0] = MDS[0][(P[P_01][b0] and 0xFF) xor calcb0(k0)]
```

```
            sBox[0x000 + 2 * i + 1] = MDS[1][(P[P_11][b1] and 0xFF) xor calcb1(k0)]
            sBox[0x200 + 2 * i + 0] = MDS[2][(P[P_21][b2] and 0xFF) xor calcb2(k0)]
            sBox[0x200 + 2 * i + 1] = MDS[3][(P[P_31][b3] and 0xFF) xor calcb3(k0)]
            break
        }

        if (k64Cnt2LSB == 0) {
            b0 = (P[P_04][b0] and 0xFF) xor calcb0(k3)
            b1 = (P[P_14][b1] and 0xFF) xor calcb1(k3)
            b2 = (P[P_24][b2] and 0xFF) xor calcb2(k3)
            b3 = (P[P_34][b3] and 0xFF) xor calcb3(k3)
            k64Cnt2LSB = 3
        }

        if (k64Cnt2LSB == 3) {
            b0 = (P[P_03][b0] and 0xFF) xor calcb0(k2)
            b1 = (P[P_13][b1] and 0xFF) xor calcb1(k2)
            b2 = (P[P_23][b2] and 0xFF) xor calcb2(k2)
            b3 = (P[P_33][b3] and 0xFF) xor calcb3(k2)
            k64Cnt2LSB = 2
        }

        if (k64Cnt2LSB == 2) {
            sBox[0x000 + 2 * i + 0] = MDS[0][(P[P_01][(P[P_02][b0] and 0xFF) xor calcb0(k1)]
and 0xFF) xor calcb0(k0)]
            sBox[0x000 + 2 * i + 1] = MDS[1][(P[P_11][(P[P_12][b1] and 0xFF) xor calcb1(k1)]
and 0xFF) xor calcb1(k0)]
            sBox[0x200 + 2 * i + 0] = MDS[2][(P[P_21][(P[P_22][b2] and 0xFF) xor calcb2(k1)]
and 0xFF) xor calcb2(k0)]
            sBox[0x200 + 2 * i + 1] = MDS[3][(P[P_31][(P[P_32][b3] and 0xFF) xor calcb3(k1)]
and 0xFF) xor calcb3(k0)]
        }
    }

    return KeyObject(sBox = sBox, subKeys = subKeys)
}

/*****************************
 * EN/DECRYPTION FUNCTIONS *
 *****************************/

/**
 * Encrypts a block of 16 plaintext bytes with the given key object.
 *
 * The 16 bytes are read from the given array at the given offset.
 * This function always reads exactly 16 bytes.
 *
 * The key object is generated from a key by using [processKey].
```

```
 *
 * @param input Byte array with the input bytes of plaintext to encrypt.
 * @param offset Offset in the input byte array to start reading bytes from.
 * @param keyObject Key object to use for encryption.
 * @return Byte array with the ciphertext version of the 16 input bytes.
 */
fun blockEncrypt(input: ByteArray, offset: Int, keyObject: KeyObject): ByteArray {
    var inputOffset = offset

    var x0 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 24)
    var x1 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 24)
    var x2 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 24)
    var x3 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
            ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
            ((input[inputOffset].toPosInt() and 0xFF) shl 24)

    val sBox = keyObject.sBox
    val subKeys = keyObject.subKeys

    x0 = x0 xor subKeys[INPUT_WHITEN + 0]
    x1 = x1 xor subKeys[INPUT_WHITEN + 1]
    x2 = x2 xor subKeys[INPUT_WHITEN + 2]
    x3 = x3 xor subKeys[INPUT_WHITEN + 3]

    var k = ROUND_SUBKEYS

    for (R in 0 until MAX_ROUNDS step 2) {
        var t0: Int
        var t1: Int

        t0 = Fe32(sBox, x0, 0)
        t1 = Fe32(sBox, x1, 3)
        x2 = x2 xor (t0 + t1 + subKeys[k++])
        x2 = (x2 ushr 1) or (x2 shl 31)
        x3 = (x3 shl 1) or (x3 ushr 31)
        x3 = x3 xor (t0 + 2 * t1 + subKeys[k++])

        t0 = Fe32(sBox, x2, 0)
```

```
            t1 = Fe32(sBox, x3, 3)
            x0 = x0 xor (t0 + t1 + subKeys[k++])
            x0 = (x0 ushr 1) or (x0 shl 31)
            x1 = (x1 shl 1) or (x1 ushr 31)
            x1 = x1 xor (t0 + 2 * t1 + subKeys[k++])
        }

        x2 = x2 xor subKeys[OUTPUT_WHITEN + 0]
        x3 = x3 xor subKeys[OUTPUT_WHITEN + 1]
        x0 = x0 xor subKeys[OUTPUT_WHITEN + 2]
        x1 = x1 xor subKeys[OUTPUT_WHITEN + 3]

        return byteArrayOfInts(
            x2, x2 ushr 8, x2 ushr 16, x2 ushr 24,
            x3, x3 ushr 8, x3 ushr 16, x3 ushr 24,
            x0, x0 ushr 8, x0 ushr 16, x0 ushr 24,
            x1, x1 ushr 8, x1 ushr 16, x1 ushr 24
        )
    }

    /**
     * Decrypts a block of 16 ciphertext bytes with the given key object.
     *
     * The 16 bytes are read from the given array at the given offset.
     * This function always reads exactly 16 bytes.
     *
     * The key object is generated from a key by using [processKey].
     *
     * @param input Byte array with the input bytes of ciphertext to decrypt.
     * @param offset Offset in the input byte array to start reading bytes from.
     * @param keyObject Key object to use for decryption.
     * @return Byte array with the plaintext version of the 16 input bytes.
     */
    fun blockDecrypt(input: ByteArray, offset: Int, keyObject: KeyObject): ByteArray {
        var inputOffset = offset

        var x2 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 24)
        var x3 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 24)
        var x0 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 24)
```

```
        var x1 = ((input[inputOffset++].toPosInt() and 0xFF) shl 0) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 8) or
                ((input[inputOffset++].toPosInt() and 0xFF) shl 16) or
                ((input[inputOffset].toPosInt() and 0xFF) shl 24)

        val sBox = keyObject.sBox
        val subKeys = keyObject.subKeys

        x2 = x2 xor subKeys[OUTPUT_WHITEN + 0]
        x3 = x3 xor subKeys[OUTPUT_WHITEN + 1]
        x0 = x0 xor subKeys[OUTPUT_WHITEN + 2]
        x1 = x1 xor subKeys[OUTPUT_WHITEN + 3]

        var k = TOTAL_SUBKEYS - 1

        for (R in 0 until MAX_ROUNDS step 2) {
            var t0: Int
            var t1: Int

            t0 = Fe32(sBox, x2, 0)
            t1 = Fe32(sBox, x3, 3)
            x1 = x1 xor (t0 + 2 * t1 + subKeys[k--])
            x1 = (x1 ushr 1) or (x1 shl 31)
            x0 = (x0 shl 1) or (x0 ushr 31)
            x0 = x0 xor (t0 + t1 + subKeys[k--])

            t0 = Fe32(sBox, x0, 0)
            t1 = Fe32(sBox, x1, 3)
            x3 = x3 xor (t0 + 2 * t1 + subKeys[k--])
            x3 = (x3 ushr 1) or (x3 shl 31)
            x2 = (x2 shl 1) or (x2 ushr 31)
            x2 = x2 xor (t0 + t1 + subKeys[k--])
        }

        x0 = x0 xor subKeys[INPUT_WHITEN + 0]
        x1 = x1 xor subKeys[INPUT_WHITEN + 1]
        x2 = x2 xor subKeys[INPUT_WHITEN + 2]
        x3 = x3 xor subKeys[INPUT_WHITEN + 3]

        return byteArrayOfInts(
            x0, x0 ushr 8, x0 ushr 16, x0 ushr 24,
            x1, x1 ushr 8, x1 ushr 16, x1 ushr 24,
            x2, x2 ushr 8, x2 ushr 16, x2 ushr 24,
            x3, x3 ushr 8, x3 ushr 16, x3 ushr 24
        )
    }
}
```

```
package info.nightscout.comboctl.base

import kotlin.test.Test
import kotlin.test.assertEquals

class TwofishTest {

    // The test datasets in the unit tests below originate from the
    // Two-Fish known answers test dataset (the twofish-kat.zip
    // archive). It can be downloaded from:
    // https://www.schneier.com/academic/twofish/

    @Test
    fun checkProcessedSubkeys() {
        // From the ecb_ival.txt test file in the twofish-kat.zip
        // Two-Fish known answers test dataset. 128-bit keysize.

        val key = byteArrayOfInts(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00, 0x00)
        val expectedSubKeys = intArrayOf(
            0x52C54DDE.toInt(), 0x11F0626D.toInt(), // Input whiten
            0x7CAC9D4A.toInt(), 0x4D1B4AAA.toInt(),
            0xB7B83A10.toInt(), 0x1E7D0BEB.toInt(), // Output whiten
            0xEE9C341F.toInt(), 0xCFE14BE4.toInt(),
            0xF98FFEF9.toInt(), 0x9C5B3C17.toInt(), // Round subkeys
            0x15A48310.toInt(), 0x342A4D81.toInt(),
            0x424D89FE.toInt(), 0xC14724A7.toInt(),
            0x311B834C.toInt(), 0xFDE87320.toInt(),
            0x3302778F.toInt(), 0x26CD67B4.toInt(),
            0x7A6C6362.toInt(), 0xC2BAF60E.toInt(),
            0x3411B994.toInt(), 0xD972C87F.toInt(),
            0x84ADB1EA.toInt(), 0xA7DEE434.toInt(),
            0x54D2960F.toInt(), 0xA2F7CAA8.toInt(),
            0xA6B8FF8C.toInt(), 0x8014C425.toInt(),
            0x6A748D1C.toInt(), 0xEDBAF720.toInt(),
            0x928EF78C.toInt(), 0x0338EE13.toInt(),
            0x9949D6BE.toInt(), 0xC8314176.toInt(),
            0x07C07D68.toInt(), 0xECAE7EA7.toInt(),
            0x1FE71844.toInt(), 0x85C05C89.toInt(),
            0xF298311E.toInt(), 0x696EA672.toInt()
        )

        val keyObject = Twofish.processKey(key)

        assertEquals(expectedSubKeys.toList(), keyObject.subKeys.toList())
    }

    @Test
```

```
fun checkPermutationTablesAndMDSMatrixMultiplyTables() {
    // From the ecb_tbl.txt test file in the twofish-kat.zip
    // Two-Fish known answers test dataset. 128-bit keysize.

    class TestVector(key: String, plaintext: String, ciphertext: String) {
        val keyArray: ByteArray
        val plaintextArray: ByteArray
        val ciphertextArray: ByteArray

        init {
            keyArray = hexstringToByteArray(key)
            plaintextArray = hexstringToByteArray(plaintext)
            ciphertextArray = hexstringToByteArray(ciphertext)
        }

        private fun hexstringToByteArray(hexstring: String): ByteArray {
            val array = ByteArray(hexstring.length / 2)

            for (i in array.indices) {
                val hexcharStr = hexstring.substring(IntRange(i * 2, i * 2 + 1))
                array[i] = Integer.parseInt(hexcharStr, 16).toByte()
            }

            return array
        }
    }

    val testVectors = arrayOf(
        TestVector(
            key = "00000000000000000000000000000000",
            plaintext = "00000000000000000000000000000000",
            ciphertext = "9F589F5CF6122C32B6BFEC2F2AE8C35A"
        ),
        TestVector(
            key = "00000000000000000000000000000000",
            plaintext = "9F589F5CF6122C32B6BFEC2F2AE8C35A",
            ciphertext = "D491DB16E7B1C39E86CB086B789F5419"
        ),
        TestVector(
            key = "9F589F5CF6122C32B6BFEC2F2AE8C35A",
            plaintext = "D491DB16E7B1C39E86CB086B789F5419",
            ciphertext = "019F9809DE1711858FAAC3A3BA20FBC3"
        ),
        TestVector(
            key = "D491DB16E7B1C39E86CB086B789F5419",
            plaintext = "019F9809DE1711858FAAC3A3BA20FBC3",
            ciphertext = "6363977DE839486297E661C6C9D668EB"
        ),
```

```
TestVector(
   key = "019F9809DE1711858FAAC3A3BA20FBC3",
   plaintext = "6363977DE839486297E661C6C9D668EB",
   ciphertext = "816D5BD0FAE35342BF2A7412C246F752"
),
TestVector(
   key = "6363977DE839486297E661C6C9D668EB",
   plaintext = "816D5BD0FAE35342BF2A7412C246F752",
   ciphertext = "5449ECA008FF5921155F598AF4CED4D0"
),
TestVector(
   key = "816D5BD0FAE35342BF2A7412C246F752",
   plaintext = "5449ECA008FF5921155F598AF4CED4D0",
   ciphertext = "6600522E97AEB3094ED5F92AFCBCDD10"
),
TestVector(
   key = "5449ECA008FF5921155F598AF4CED4D0",
   plaintext = "6600522E97AEB3094ED5F92AFCBCDD10",
   ciphertext = "34C8A5FB2D3D08A170D120AC6D26DBFA"
),
TestVector(
   key = "6600522E97AEB3094ED5F92AFCBCDD10",
   plaintext = "34C8A5FB2D3D08A170D120AC6D26DBFA",
   ciphertext = "28530B358C1B42EF277DE6D4407FC591"
),
TestVector(
   key = "34C8A5FB2D3D08A170D120AC6D26DBFA",
   plaintext = "28530B358C1B42EF277DE6D4407FC591",
   ciphertext = "8A8AB983310ED78C8C0ECDE030B8DCA4"
),
TestVector(
   key = "28530B358C1B42EF277DE6D4407FC591",
   plaintext = "8A8AB983310ED78C8C0ECDE030B8DCA4",
   ciphertext = "48C758A6DFC1DD8B259FA165E1CE2B3C"
),
TestVector(
   key = "8A8AB983310ED78C8C0ECDE030B8DCA4",
   plaintext = "48C758A6DFC1DD8B259FA165E1CE2B3C",
   ciphertext = "CE73C65C101680BBC251C5C16ABCF214"
),
TestVector(
   key = "48C758A6DFC1DD8B259FA165E1CE2B3C",
   plaintext = "CE73C65C101680BBC251C5C16ABCF214",
   ciphertext = "C7ABD74AA060F78B244E24C71342BA89"
),
TestVector(
   key = "CE73C65C101680BBC251C5C16ABCF214",
   plaintext = "C7ABD74AA060F78B244E24C71342BA89",
```

```
    ciphertext = "D0F8B3B6409EBCB666D29C916565ABFC"
),
TestVector(
    key = "C7ABD74AA060F78B244E24C71342BA89",
    plaintext = "D0F8B3B6409EBCB666D29C916565ABFC",
    ciphertext = "DD42662908070054544FE09DA4263130"
),
TestVector(
    key = "D0F8B3B6409EBCB666D29C916565ABFC",
    plaintext = "DD42662908070054544FE09DA4263130",
    ciphertext = "7007BACB42F7BF989CF30F78BC50EDCA"
),
TestVector(
    key = "DD42662908070054544FE09DA4263130",
    plaintext = "7007BACB42F7BF989CF30F78BC50EDCA",
    ciphertext = "57B9A18EE97D90F435A16F69F0AC6F16"
),
TestVector(
    key = "7007BACB42F7BF989CF30F78BC50EDCA",
    plaintext = "57B9A18EE97D90F435A16F69F0AC6F16",
    ciphertext = "06181F0D53267ABD8F3BB28455B198AD"
),
TestVector(
    key = "57B9A18EE97D90F435A16F69F0AC6F16",
    plaintext = "06181F0D53267ABD8F3BB28455B198AD",
    ciphertext = "81A12D8449E9040BAAE7196338D8C8F2"
),
TestVector(
    key = "06181F0D53267ABD8F3BB28455B198AD",
    plaintext = "81A12D8449E9040BAAE7196338D8C8F2",
    ciphertext = "BE422651C56F2622DA0201815A95A820"
),
TestVector(
    key = "81A12D8449E9040BAAE7196338D8C8F2",
    plaintext = "BE422651C56F2622DA0201815A95A820",
    ciphertext = "113B19F2D778473990480CEE4DA238D1"
),
TestVector(
    key = "BE422651C56F2622DA0201815A95A820",
    plaintext = "113B19F2D778473990480CEE4DA238D1",
    ciphertext = "E6942E9A86E544CF3E3364F20BE011DF"
),
TestVector(
    key = "113B19F2D778473990480CEE4DA238D1",
    plaintext = "E6942E9A86E544CF3E3364F20BE011DF",
    ciphertext = "87CDC6AA487BFD0EA70188257D9B3859"
),
TestVector(
```

```
    key = "E6942E9A86E544CF3E3364F20BE011DF",
    plaintext = "87CDC6AA487BFD0EA70188257D9B3859",
    ciphertext = "D5E2701253DD75A11A4CFB243714BD14"
),
TestVector(
    key = "87CDC6AA487BFD0EA70188257D9B3859",
    plaintext = "D5E2701253DD75A11A4CFB243714BD14",
    ciphertext = "FD24812EEA107A9E6FAB8EABE0F0F48C"
),
TestVector(
    key = "D5E2701253DD75A11A4CFB243714BD14",
    plaintext = "FD24812EEA107A9E6FAB8EABE0F0F48C",
    ciphertext = "DAFA84E31A297F372C3A807100CD783D"
),
TestVector(
    key = "FD24812EEA107A9E6FAB8EABE0F0F48C",
    plaintext = "DAFA84E31A297F372C3A807100CD783D",
    ciphertext = "A55ED2D955EC8950FC0CC93B76ACBF91"
),
TestVector(
    key = "DAFA84E31A297F372C3A807100CD783D",
    plaintext = "A55ED2D955EC8950FC0CC93B76ACBF91",
    ciphertext = "2ABEA2A4BF27ABDC6B6F278993264744"
),
TestVector(
    key = "A55ED2D955EC8950FC0CC93B76ACBF91",
    plaintext = "2ABEA2A4BF27ABDC6B6F278993264744",
    ciphertext = "045383E219321D5A4435C0E491E7DE10"
),
TestVector(
    key = "2ABEA2A4BF27ABDC6B6F278993264744",
    plaintext = "045383E219321D5A4435C0E491E7DE10",
    ciphertext = "7460A4CD4F312F32B1C7A94FA004E934"
),
TestVector(
    key = "045383E219321D5A4435C0E491E7DE10",
    plaintext = "7460A4CD4F312F32B1C7A94FA004E934",
    ciphertext = "6BBF9186D32C2C5895649D746566050A"
),
TestVector(
    key = "7460A4CD4F312F32B1C7A94FA004E934",
    plaintext = "6BBF9186D32C2C5895649D746566050A",
    ciphertext = "CDBDD19ACF40B8AC0328C80054266068"
),
TestVector(
    key = "6BBF9186D32C2C5895649D746566050A",
    plaintext = "CDBDD19ACF40B8AC0328C80054266068",
    ciphertext = "1D2836CAE4223EAB5066867A71B1A1C3"
```

```
),
TestVector(
    key = "CDBDD19ACF40B8AC0328C80054266068",
    plaintext = "1D2836CAE4223EAB5066867A71B1A1C3",
    ciphertext = "2D7F37121D0D2416D5E2767FF202061B"
),
TestVector(
    key = "1D2836CAE4223EAB5066867A71B1A1C3",
    plaintext = "2D7F37121D0D2416D5E2767FF202061B",
    ciphertext = "D70736D1ABC7427A121CC816CD66D7FF"
),
TestVector(
    key = "2D7F37121D0D2416D5E2767FF202061B",
    plaintext = "D70736D1ABC7427A121CC816CD66D7FF",
    ciphertext = "AC6CA71CBCBEDCC0EA849FB2E9377865"
),
TestVector(
    key = "D70736D1ABC7427A121CC816CD66D7FF",
    plaintext = "AC6CA71CBCBEDCC0EA849FB2E9377865",
    ciphertext = "307265FF145CBBC7104B3E51C6C1D6B4"
),
TestVector(
    key = "AC6CA71CBCBEDCC0EA849FB2E9377865",
    plaintext = "307265FF145CBBC7104B3E51C6C1D6B4",
    ciphertext = "934B7DB4B3544854DBCA81C4C5DE4EB1"
),
TestVector(
    key = "307265FF145CBBC7104B3E51C6C1D6B4",
    plaintext = "934B7DB4B3544854DBCA81C4C5DE4EB1",
    ciphertext = "18759824AD9823D5961F84377D7EAEBF"
),
TestVector(
    key = "934B7DB4B3544854DBCA81C4C5DE4EB1",
    plaintext = "18759824AD9823D5961F84377D7EAEBF",
    ciphertext = "DEDDAC6029B01574D9BABB099DC6CA6C"
),
TestVector(
    key = "18759824AD9823D5961F84377D7EAEBF",
    plaintext = "DEDDAC6029B01574D9BABB099DC6CA6C",
    ciphertext = "5EA82EEA2244DED42CCA2F835D5615DF"
),
TestVector(
    key = "DEDDAC6029B01574D9BABB099DC6CA6C",
    plaintext = "5EA82EEA2244DED42CCA2F835D5615DF",
    ciphertext = "1E3853F7FFA57091771DD8CDEE9414DE"
),
TestVector(
    key = "5EA82EEA2244DED42CCA2F835D5615DF",
```

```
            plaintext = "1E3853F7FFA57091771DD8CDEE9414DE",
            ciphertext = "5C2EBBF75D31F30B5EA26EAC8782D8D1"
        ),
        TestVector(
            key = "1E3853F7FFA57091771DD8CDEE9414DE",
            plaintext = "5C2EBBF75D31F30B5EA26EAC8782D8D1",
            ciphertext = "3A3CFA1F13A136C94D76E5FA4A1109FF"
        ),
        TestVector(
            key = "5C2EBBF75D31F30B5EA26EAC8782D8D1",
            plaintext = "3A3CFA1F13A136C94D76E5FA4A1109FF",
            ciphertext = "91630CF96003B8032E695797E313A553"
        ),
        TestVector(
            key = "3A3CFA1F13A136C94D76E5FA4A1109FF",
            plaintext = "91630CF96003B8032E695797E313A553",
            ciphertext = "137A24CA47CD12BE818DF4D2F4355960"
        ),
        TestVector(
            key = "91630CF96003B8032E695797E313A553",
            plaintext = "137A24CA47CD12BE818DF4D2F4355960",
            ciphertext = "BCA724A54533C6987E14AA827952F921"
        ),
        TestVector(
            key = "137A24CA47CD12BE818DF4D2F4355960",
            plaintext = "BCA724A54533C6987E14AA827952F921",
            ciphertext = "6B459286F3FFD28D49F15B1581B08E42"
        ),
        TestVector(
            key = "BCA724A54533C6987E14AA827952F921",
            plaintext = "6B459286F3FFD28D49F15B1581B08E42",
            ciphertext = "5D9D4EEFFA9151575524F115815A12E0"
        )
    )

    for (testVector in testVectors) {
        val keyObject = Twofish.processKey(testVector.keyArray)

        val computedCiphertext = Twofish.blockEncrypt(testVector.plaintextArray, 0,
keyObject)
        assertEquals(testVector.ciphertextArray.toList(), computedCiphertext.toList())

        val computedPlaintext = Twofish.blockDecrypt(testVector.ciphertextArray, 0,
keyObject)
        assertEquals(testVector.plaintextArray.toList(), computedPlaintext.toList())
    }
  }
}
```