Supplement to ANSI X9.24-3-2017

Python Source Code—

Accredited Standards Committee X9, Incorporated
Financial Industry Standards

Contents **Page**

1 Scope

This document is a supplement to ANSI X9.24-3-2017 and describes a set of source code that can be used as a reference implementation of the AES DUKPT algorithm and to support validation of an implementation of the AES DUKPT algorithm on a transaction-originating SCD or a receiving SCD. AES DUKPT is used to derive transaction key(s) from an initial terminal DUKPT key based on the transaction number. Keys that can be derived include symmetric encryption/decryption keys, authentication keys, and HMAC (keyed hash message authentication code) keys. AES DUKPT supports the derivation of AES-128, AES-192, AES-256, double length TDEA, and triple length TDEA keys from AES-128, AES-192, and AES-256 initial keys.

While the included source code contains a reference implementation of the AES DUKPT algorithm, in no way should the included source code be considered an implementation of the entirety of the requirements of the ANSI X9.24 Part 3 standard. Care must be taken to follow all requirements when deploying a complete implementation of the standard.

The included source code contains no warranty or guarantees and is considered open source.

2 Python Reference Implementation

2.1 General

This Annex gives the Python source code that was used to generate the test vectors in Annex B. In the event that it disagrees with the pseudo code in the main body of the standard, the text in the main body of the standard is considered normative.

It was developed using Python 3.4 and PyCrypto version 2.6.1, but should work with any version of Python 3. Information about PyCrypto can be found at https://pypi.python.org/pypi/pycrypto.

The original Python source files for this AES DUKPT reference implementation can be found at http://x9.org/standards/x9-24-part-3-test-vectors/, and it is recommended that the Python files be used rather than trying to copy the python source out of this document.

2.2 Test Vectors

```
from Crypto.Cipher import AES
from enum import Enum
import binascii

# print out a hexadecimal number in groups of 8 capitalized digits.  Only for
debugging.
def ToGroups(x):
    n = len(x)//8
    for i in range(0, n):
        print(x[i*8:(i+1)*8].upper(), end="")
        if i != n-1:
            print(" ", end="")
```

```
# Convert a 32-bit integer to a list of bytes in big-endian order.  Used to convert
counter values to byte lists.
def IntToBytes(x):
    return [((x >> i) & 0xff) for i in (24,16,8,0)]

# Print out a debug message at depth d.  Takes an arbitrary list of strings and byte
lists or bytearrays.
# Lists of bytes are pretty-printed in hex.
def D(d, *args):
    global Debug
    if Debug:
        print(".", end="")
        for i in range(0,2*d):
            print(" ", end="")
        for x in args:
            if type(x) is str:
                print(x, " ", end="")
            elif type(x) is list:
                value = binascii.hexlify(bytearray(x)).decode("utf-8")
                ToGroups(value)
            elif type(x) is bytearray:
                value = binascii.hexlify(x).decode("utf-8")
                ToGroups(value)
            elif type(x) is bytes:
                value = binascii.hexlify(x).decode("utf-8")
                ToGroups(value)
            else:
                print(x, " ", end="")
        print("")

def P(*args):
    for x in args:
        if type(x) is str:
            print(x, " ", end="")
        elif type(x) is list:
            value = binascii.hexlify(bytearray(x)).decode("utf-8")
            ToGroups(value)
        elif type(x) is bytearray:
            value = binascii.hexlify(x).decode("utf-8")
            ToGroups(value)
        elif type(x) is bytes:
            value = binascii.hexlify(x).decode("utf-8")
            ToGroups(value)
        else:
            print(x, " ", end="")
    print("")

# B.3.1. Enumerations
class DerivationPurpose(Enum):
    _InitialKey = 0
    _DerivationOrWorkingKey = 1

class KeyType(Enum):
    _2TDEA = 0
    _3TDEA = 1
    _AES128 = 2
    _AES192 = 3
    _AES256 = 4

class KeyUsage(Enum):
    _KeyEncryptionKey = 0x0002
    _PINEncryption = 0x1000
    _MessageAuthenticationGeneration = 0x2000
    _MessageAuthenticationVerification = 0x2001
    _MessageAuthenticationBothWays = 0x2002
    _DataEncryptionEncrypt = 0x3000
    _DataEncryptionDecrypt = 0x3001
    _DataEncryptionBothWays = 0x3002
    _KeyDerivation = 0x8000
    _KeyDerivationInitialKey = 9

# Count the number of 1 bits in a counter value.  Readable, but not efficient.
def Count_One_Bits(x):
    bits = 0
    mask = 1 << (NUMREG-1)
    while mask > 0:
        if x & mask:
            bits = bits + 1
        mask = mask >> 1
    return bits

# B.3.2. Key Length function
# Length of an algorithm's key, in bits.
def Key_Length( keyType ):
    if ( keyType == KeyType._2TDEA):
        return 128
    if ( keyType == KeyType._3TDEA):
        return 192
    if ( keyType == KeyType._AES128):
        return 128
    if ( keyType == KeyType._AES192):
        return 192
    if ( keyType == KeyType._AES256):
        return 256
    assert False

# Encrypt plaintext with key using AES.
def AES_Encrypt_ECB( d, key, plaintext ):
    D(d, "AES_Encrypt_ECB:", key, "data:", plaintext)
    encobj = AES.new(bytes(key), AES.MODE_ECB)
    result = encobj.encrypt(bytes(plaintext))
    D(d+1, "result = ", result)
    return result

# Compute the XOR of two 128-bit numbers
def XOR(a, b):
    result = bytearray(16)
    for i in range(0,16):
        result[i] = a[i] ^ b[i]
    return result
```

```
# B.4.1. Derive Key algorithm
# AES DUKPT key derivation function.
def Derive_Key( d, derivationKey, keyType, derivationData, deriveType ):
   D(d, "Derive_Key:", derivationKey, keyType, derivationData, deriveType)
   L = Key_Length(keyType)
   n = -(-L // 128)

   result = bytearray(n*16)
   for i in range(1, n+1):
      derivationData[1] = i
      result[(i-1)*16:i*16] = AES_Encrypt_ECB(d+2, derivationKey, derivationData)

   derivedKey = result[0:(L//8)]
   D(d+1, "derivedKey:", derivedKey)
   return derivedKey

# B.4.3. Create Derivation Data
# Compute derivation data for an AES DUKPT key derivation operation.
def Create_Derivation_Data( derivationPurpose, keyUsage, derivedKeyType,
initialKeyID, counter):
   derivationData = bytearray(16)
   derivationData[0] = 1
   derivationData[1] = 1

   if (keyUsage == KeyUsage._KeyEncryptionKey):
      derivationData[2:4] = [0,2]
   elif (keyUsage == KeyUsage._PINEncryption):
      derivationData[2:4] = [16,0]
   elif (keyUsage == KeyUsage._MessageAuthenticationGeneration):
      derivationData[2:4] = [32,0]
   elif (keyUsage == KeyUsage._MessageAuthenticationVerification):
      derivationData[2:4] = [32,1]
   elif (keyUsage == KeyUsage._MessageAuthenticationBothWays):
      derivationData[2:4] = [32,2]
   elif (keyUsage == KeyUsage._DataEncryptionEncrypt):
      derivationData[2:4] = [48,0]
   elif (keyUsage == KeyUsage._DataEncryptionDecrypt):
      derivationData[2:4] = [48,1]
   elif (keyUsage == KeyUsage._DataEncryptionBothWays):
      derivationData[2:4] = [48,2]
   elif (keyUsage == KeyUsage._KeyDerivation):
      derivationData[2:4] = [128,0]
   elif (keyUsage == KeyUsage._KeyDerivationInitialKey):
      derivationData[2:4] = [128,1]
   else:
      assert False

   if (derivedKeyType == KeyType._2TDEA):
      derivationData[4:6] = [0,0]
   elif (derivedKeyType == KeyType._3TDEA):
      derivationData[4:6] = [0,1]
   elif (derivedKeyType == KeyType._AES128):
      derivationData[4:6] = [0,2]
   elif (derivedKeyType == KeyType._AES192):
        derivationData[4:6] = [0,3]
    elif (derivedKeyType == KeyType._AES256):
        derivationData[4:6] = [0,4]
    else:
        assert False

    if (derivedKeyType == KeyType._2TDEA):
        derivationData[6:8] = [0,128]
    elif (derivedKeyType == KeyType._3TDEA):
        derivationData[6:8] = [0,192]
    elif (derivedKeyType == KeyType._AES128):
        derivationData[6:8] = [0,128]
    elif (derivedKeyType == KeyType._AES192):
        derivationData[6:8] = [0,192]
    elif (derivedKeyType == KeyType._AES256):
        derivationData[6:8] = [1,0]
    else:
        assert False

    if (derivationPurpose == DerivationPurpose._InitialKey):
        derivationData[8:16] = initialKeyID[0:8]
    elif (derivationPurpose == DerivationPurpose._DerivationOrWorkingKey):
        derivationData[8:12] = initialKeyID[4:8]
        derivationData[12:16] = IntToBytes(counter)
    else:
        assert False

    return derivationData

# B.5. Derive Initial Key
# Derive the initial key for a particular initialKeyID from a BDK.
def Derive_Initial_Key( d, BDK, keyType, initialKeyID ):
    D(d, "Derive_Initial_Key:", BDK, keyType, initialKeyID)
    derivationData = Create_Derivation_Data(DerivationPurpose._InitialKey,
KeyUsage._KeyDerivationInitialKey, keyType, initialKeyID, 0)
    D(d+1, "derivationData:", derivationData)
    initialKey = Derive_Key(d+2, BDK, keyType, derivationData, keyType)

    return initialKey

# B.5. Host Derive Working Key
# Derive a working key for a particular transaction based on a initialKeyID and
counter.
def Host_Derive_Working_Key( d, BDK, deriveKeyType, workingKeyUsage, workingKeyType,
initialKeyID, counter):
    D(d, "Host_Derive_Working_Key:", BDK, workingKeyUsage, workingKeyType,
initialKeyID, counter)
    initialKey = Derive_Initial_Key(d+2, BDK, deriveKeyType, initialKeyID)
    D(d+1, "initialKey:", initialKey)

    mask = 0x80000000
    workingCounter = 0
    derivationKey = initialKey

    while mask > 0:
       if (mask & counter) != 0:
           D(d+1, "BIT FOUND:", mask)
           workingCounter = workingCounter | mask
           derivationData =
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey,
KeyUsage._KeyDerivation, deriveKeyType, initialKeyID, workingCounter)
           D(d+1, "derivationData:", derivationData)
           derivationKey = Derive_Key(d+2, derivationKey, deriveKeyType,
derivationData, deriveKeyType)
           D(d+1, "derivationKey:", derivationKey)
       mask = mask >> 1

   D(d+1, "FINAL DERIVATION:")
   derivationData =
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, workingKeyUsage,
workingKeyType, initialKeyID, counter)
   D(d+1, "derivationData", derivationData)
   workingKey = Derive_Key(d+2, derivationKey, workingKeyType, derivationData,
deriveKeyType)
   D(d+1, "workingKey", workingKey)

   return derivationKey, derivationData, workingKey

# B.6.3. Processing Routines
# Load an initial key for computing terminal transaction keys in sequence.
def Load_Initial_Key( d, initialKey, deriveKeyType, initialKeyID ):
   global NUMREG
   global gIntermediateDerivationKeyRegister
   global gIntermediateDerivationKeyInUse
   global gCurrentKey
   global gDeviceID
   global gCounter
   global gShiftRegister
   global gDeriveKeyType

   D(d, "Load_Initial_Key:", initialKey, deriveKeyType, initialKeyID)

   gIntermediateDerivationKeyRegister = [None]*NUMREG
   gIntermediateDerivationKeyInUse = [False]*NUMREG

   gIntermediateDerivationKeyRegister[0] = initialKey
   gIntermediateDerivationKeyInUse[0] = True
   gDeviceID = initialKeyID
   gCounter = 0
   gShiftRegister = 1
   gCurrentKey = 0
   gDeriveKeyType = deriveKeyType

   Update_Derivation_Keys(d+1, NUMREG-1, deriveKeyType)
   gCounter = gCounter + 1

# B.6.3. Update Initial Key
# Load a new terminal initial key under a pre-existing terminal initial key.
def Update_Initial_Key( d, encryptedInitialKey, initialKeyType, newDeviceID ):
   D(d, "Update_Initial_Key:", encryptedInitialkey, initialKeyType, newDeviceID)
```

```
    if (gCounter > ((1 << NUMREG) - 1)):
        Cease_Operation()
        return False

    derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, 
KeyUsage._KeyEncryptingKey, initialKeyType, gDeviceID, gCounter)
    D(d+1, "derivationData:", derivationData)
    keyEncryptionKey = DeriveKey(d+2, FutureKeyRegister[CurrentKey], initialKeyType, 
derivationData)
    D(d, "keyEncryptionKey:", keyEncryptionKey)

    n = (Key_Length(initialKeyType)+127)//128
    for i in range(1,n):
        newInitialKey[(i-1)*16:i*16] = AES_Decrypt_ECB(d+1, keyEncryptionKey, 
encryptedInitialKey[(i-1)*16:i*16])

    Load_Initial_Key(d+1, newInitialKey, newDeviceID)

    return True

# B.6.3. Generate Working Keys
# Generate a transaction key from the intermediate derivation key registers, and 
update the state to prepare for the next transaction.
def Generate_Working_Keys( d, workingKeyUsage, workingKeyType ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Generate_Working_Keys", workingKeyUsage, workingKeyType)

    Set_Shift_Register(d+1)

    while not gIntermediateDerivationKeyInUse[gCurrentKey]:
        D(d+1, "Key", gCurrentKey, "not in use")
        gCounter = gCounter + gShiftRegister
        if gCounter > ((1 << NUMREG) - 1):
            Cease_Operation()
            return False
        Set_Shift_Register(d+1)

    D(d+1, "gCounter:", gCounter)

    derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, workingKeyUsage, 
workingKeyType, gDeviceID, gCounter)
    D(d+1, "derivationData:", derivationData)
    assert gIntermediateDerivationKeyInUse[gCurrentKey]
    workingKey = Derive_Key(d+2, gIntermediateDerivationKeyRegister[gCurrentKey], 
workingKeyType, derivationData, gDeriveKeyType)
```

```
    D(d+1, "workingKey:", workingKey)

    Update_State_for_next_Transaction(d+2)
    return workingKey

# B.6.3. Update State for next Transaction
# Move the counter forward, and derive new intermediate derivation keys for the next
transaction.
def Update_State_for_next_Transaction(d):
    global NUMREG
    global MAX_WORK
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Update_State_for_next_Transaction")

    oneBits = Count_One_Bits(gCounter)
    if oneBits <= MAX_WORK:
        Update_Derivation_Keys(d+2, gCurrentKey, gDeriveKeyType)
        gIntermediateDerivationKeyRegister[gCurrentKey] = 0
        gIntermediateDerivationKeyInUse[gCurrentKey] = False
        gCounter = gCounter + 1
    else:
        gIntermediateDerivationKeyRegister[gCurrentKey] = 0
        gIntermediateDerivationKeyInUse[gCurrentKey] = False
        gCounter = gCounter + gShiftRegister
    D(d+1, "gCounter:", gCounter)

    if gCounter > (1 << NUMREG) - 1:
        CeaseOperation()
        return False
    else:
        return True

# B.6.3. Update Derivation Keys
# Update all the intermediate derivation key registers below a certain point.
# This is used to:
#    1. Update all the intermediate derivation key registers below the shift
register after computing a transaction key.
#    2. Update all the intermediate derivation key registers when an initial key is
loaded.
def Update_Derivation_Keys( d, start, deriveKeyType ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    D(d, "Update_Derivation_Keys", deriveKeyType)

    i = start
    j = 1 << start

    D(d+1, "gCurrentKey:", gCurrentKey)
    baseKey = gIntermediateDerivationKeyRegister[gCurrentKey]
    while j != 0:
        D(d+1, "i:", i, "gShiftRegister:", j)
        derivationData =
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey,
KeyUsage._KeyDerivation, deriveKeyType, gDeviceID, gCounter | j)
        D(d+1, "derivationData:", derivationData)
        assert gIntermediateDerivationKeyInUse[gCurrentKey]
        gIntermediateDerivationKeyRegister[i] = Derive_Key(d+1, baseKey,
deriveKeyType, derivationData, deriveKeyType)
        gIntermediateDerivationKeyInUse[i] = True
        j = j >> 1
        i = i - 1

    return True

# B.6.3. Set Shift Register
# Set the shift register to the value of the rightmost '1' bit in the counter.
def Set_Shift_Register(d):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister

    gShiftRegister = 1
    gCurrentKey = 0

    if gCounter == 0:
        D(d, "Set_Shift_Register -> gShiftRegister:", gShiftRegister, "gCurrentKey:",
gCurrentKey)
        return True

    while (gShiftRegister & gCounter) == 0:
        gShiftRegister = gShiftRegister << 1
        gCurrentKey = gCurrentKey + 1

    D(d, "Set_Shift_Register -> gShiftRegister:", gShiftRegister, "gCurrentKey:",
gCurrentKey)
    return True

# Generate the test vectors for a particular BDK, initial key ID, and transaction
key type
def GenerateTestVectors(bdk, bk, initialKeyID, k):
    P("")
    P("Test Vectors for generating ", k, " from ", bk, " Base Derivation Key")
    initialKey = Derive_Initial_Key(0, bdk, bk, initialKeyID)
    P("")
    P("Initial Key:")
    P(initialKey)
    P("")
    Load_Initial_Key(0, initialKey, bk, initialKeyID )

    max = (1 << NUMREG) - 1
    for i in [1,2,3,4,5,6,7,8,
             0x0001FFFE,0x0001FFFF,0x00020000,0x00020001,
             0x00845FED,
             0xFFFE2000, 0xFFFE4000, 0xFFFE8000, 0xFFFF0000,
             max]:
        P("")
        if i == max:
            P("DUKPT Update Key (counter = ", hex(i), ")")
        else:
            P("Counter: ", i, "   (", hex(i), ")")

        keyPIN = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._PINEncryption, k,
initialKeyID, i)
        keyMAC = Host_Derive_Working_Key(0, bdk, bk,
KeyUsage._MessageAuthenticationGeneration, k, initialKeyID, i)
        keyDEE = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._DataEncryptionEncrypt,
k, initialKeyID, i)
        keyKEK = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._KeyEncryptionKey, k,
initialKeyID, i)

        P("")
        if i == max:
            P("Derivation Key:")
            P(keyKEK[0])
            P("Derivation Data:")
            P(keyKEK[1])
            P("Key Encryption Key:")
            P(keyKEK[2])
        else:
            P("Derivation Key:")
            P(keyPIN[0])
            P("PIN Encryption Derivation Data:")
            P(keyPIN[1])
            P("PIN Encryption Key:")
            P(keyPIN[2])
            P("MAC Derivation Data:")
            P(keyMAC[1])
            P("Message Authentication, Generation:")
            P(keyMAC[2])
            P("Encryption Derivation Data:")
            P(keyDEE[1])
            P("Data Encryption, Encrypt:")
            P(keyDEE[2])

# Generate all the valid key types for a few transactions.
# Also demonstrate the calculation of a AES PIN Block (Format 4)
def GenerateAllKeys(bdk, bk, initialKeyID, k):
    for i in [1,2,3,4,5,6,7,8, 8675309]:
        P("")
        P(" All Key Usages for Transaction ",i," (AES-128 under AES-128 BDK)")
        initialKey = Derive_Initial_Key(0, bdk, bk, initialKeyID)
        P("")
        P("   Initial Key:")
        P(initialKey)
        P("")
        Load_Initial_Key(0, initialKey, bk, initialKeyID )

        P("   Counter: ", i, "   (", hex(i), ")")

        keyKEK = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._KeyEncryptionKey, k,
initialKeyID, i)
        keyPIN = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._PINEncryption, k,
initialKeyID, i)
        keyMACG = Host_Derive_Working_Key(0, bdk, bk,
KeyUsage._MessageAuthenticationGeneration, k, initialKeyID, i)
        keyMACV = Host_Derive_Working_Key(0, bdk, bk,
KeyUsage._MessageAuthenticationVerification, k, initialKeyID, i)
        keyMACB = Host_Derive_Working_Key(0, bdk, bk,
KeyUsage._MessageAuthenticationBothWays, k, initialKeyID, i)
        keyDEE = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._DataEncryptionEncrypt,
k, initialKeyID, i)
        keyDED = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._DataEncryptionDecrypt,
k, initialKeyID, i)
        keyDEB = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._DataEncryptionBothWays,
k, initialKeyID, i)
        keyKD = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._KeyDerivation, k,
initialKeyID, i)

        P("")
        P("Derivation Key:")
        P(keyKEK[0])
        P("Key Encryption Key Derivation Data:")
        P(keyKEK[1])
        P("Key Encryption Key:")
        P(keyKEK[2])
        P("")
        P("PIN Encryption Derivation Data:")
        P(keyPIN[1])
        P("PIN Encryption Key:")
        P(keyPIN[2])
        P("")
        P("MAC Generation Derivation Data:")
        P(keyMACG[1])
        P("Message Auth, Generation:")
        P(keyMACG[2])
        P("")
        P("MAC Verification Derivation Data:")
        P(keyMACV[1])
        P("Message Auth, Verification:")
        P(keyMACV[2])
        P("")
        P("MAC Both Ways Derivation Data:")
        P(keyMACB[1])
```

```
	P("Message Auth, Both Ways:")
	P(keyMACB[2])
	P("")
	P("DE Encrypt Derivation Data:")
	P(keyDEE[1])
	P("Data Encryption, Encrypt:")
	P(keyDEE[2])
	P("")
	P("DE Decrypt Derivation Data:")
	P(keyDED[1])
	P("Data Encryption, Decrypt:")
	P(keyDED[2])
	P("")
	P("DE Both Ways Derivation Data:")
	P(keyDEB[1])
	P("Data Encryption, Both Ways:")
	P(keyDEB[2])
	P("")
	P("Key Derivation Derivation Data:")
	P(keyKD[1])
	P("Key Derivation Key:")
	P(keyKD[2])

	P("")
	P("Calculation of AES PIN Block (Format 4)")
	P("")
	P("PAN = 4111111111111111")
	P("PIN = 1234")
	P("Random Number = 2F69ADDE2E9E7ACE")
	P("")
	P("Plaintext PIN field:")
	P("441234AA  AAAAAAAA  2F69ADDE  2E9E7ACE")
	P("Plaintext PAN field:")
	P("44111111  11111111  10000000  00000000")
	P("PIN Encryption Key:")
	P(keyPIN[2])

	pinField = [ 0x44, 0x12, 0x34, 0xAA, 0xAA, 0xAA, 0xAA, 0xAA, 0x2F, 0x69, 0xAD,
0xDE, 0x2E, 0x9E, 0x7A, 0xCE ]
	panField = [ 0x44, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x11, 0x10, 0x00, 0x00,
0x00, 0x00, 0x00, 0x00, 0x00 ]
	blockA = AES_Encrypt_ECB(0, keyPIN[2], pinField)
	blockB = XOR(blockA, panField)
	pinBlock = AES_Encrypt_ECB(0, keyPIN[2], blockB)

	P("")
	P("Intermediate Block A:")
	P(blockA)
	P("Intermediate Block B:")
	P(blockB)
	P("Encrypted PIN Block:")
	P(pinBlock)

######################## MAIN ##################################
```

```
Debug = False
NUMREG = 32
MAX_WORK = 16

bdk = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1 ]
bdk192 = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1,
         0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10 ]
bdk256 = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1,
         0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1]

initialKeyID = [ 0x12, 0x34, 0x56, 0x78, 0x90, 0x12, 0x34, 0x56 ]

derivationData = Create_Derivation_Data( DerivationPurpose._InitialKey,
KeyUsage._KeyDerivationInitialKey, KeyType._AES128, initialKeyID, 0 )

initialKey = Derive_Initial_Key(0, bdk, KeyType._AES128, initialKeyID)

P("BDK-128:")
P(bdk)
P("")

P("BDK-256:")
P(bdk256)
P("")

P("InitialKeyID:")
P(initialKeyID)
P("")

P("Derivation Data:")
P(derivationData)
P("")

P("Initial Key:")
P(initialKey)
P("")
P("")

GenerateTestVectors(bdk, KeyType._AES128, initialKeyID, KeyType._AES128)
GenerateTestVectors(bdk256, KeyType._AES256, initialKeyID, KeyType._AES128)
GenerateTestVectors(bdk256, KeyType._AES256, initialKeyID, KeyType._AES256)
GenerateTestVectors(bdk, KeyType._AES128, initialKeyID, KeyType._2TDEA)
GenerateTestVectors(bdk, KeyType._AES128, initialKeyID, KeyType._3TDEA)

GenerateAllKeys(bdk, KeyType._AES128, initialKeyID, KeyType._AES128)
```

2.3 Debugging Trace (Host Algorithm)

```
from Crypto.Cipher import AES
from enum import Enum
import binascii

# print out a hexadecimal number in groups of 8 capitalized digits.  Only for
debugging.
def ToGroups(x):
   n = len(x)//8
   for i in range(0, n):
      print(x[i*8:(i+1)*8].upper(), " ", end="")

# Convert a 32-bit integer to a list of bytes in big-endian order.  Used to convert
counter values to byte lists.
def IntToBytes(x):
   return [((x >> i) & 0xff) for i in (24,16,8,0)]

# Print out a debug message at depth d.  Takes an arbitrary list of strings and byte
lists or bytearrays.
# Lists of bytes are pretty-printed in hex.
def D(d, *args):
   for i in range(0,2*d):
      print(" ", end="")
   for x in args:
      if type(x) is str:
         print(x, " ", end="")
      elif type(x) is list:
         value = binascii.hexlify(bytearray(x)).decode("utf-8")
         ToGroups(value)
      elif type(x) is bytearray:
         value = binascii.hexlify(x).decode("utf-8")
         ToGroups(value)
      elif type(x) is bytes:
         value = binascii.hexlify(x).decode("utf-8")
         ToGroups(value)
      else:
         print(x, " ", end="")
   print("")

# B.3.1. Enumerations
class DerivationPurpose(Enum):
   _InitialKey = 0
   _DerivationOrWorkingKey = 1

class KeyType(Enum):
   _2TDEA = 0
   _3TDEA = 1
   _AES128 = 2
   _AES192 = 3
   _AES256 = 4

class KeyUsage(Enum):
   _KeyEncryptionKey = 0x0002
   _PINEncryption = 0x1000
```

```
    _MessageAuthenticationGeneration = 0x2000
    _MessageAuthenticationVerification = 0x2001
    _MessageAuthenticationBothWays = 0x2002
    _DataEncryptionEncrypt = 0x3000
    _DataEncryptionDecrypt = 0x3001
    _DataEncryptionBothWays = 0x3002
    _KeyDerivation = 0x8000
    _KeyDerivationInitialKey = 9

# Count the number of 1 bits in a counter value.  Readable, but not efficient.
def Count_One_Bits(x):
    bits = 0
    mask = 1 << (NUMREG-1)
    while mask > 0:
        if x & mask:
            bits = bits + 1
        mask = mask >> 1
    return bits

# B.3.2. Key Length function
# Length of an algorithm's key, in bits.
def Key_Length( keyType ):
    if ( keyType == KeyType._2TDEA):
        return 128
    if ( keyType == KeyType._3TDEA):
        return 192
    if ( keyType == KeyType._AES128):
        return 128
    if ( keyType == KeyType._AES192):
        return 192
    if ( keyType == KeyType._AES256):
        return 256
    assert False

# Encrypt plaintext with key using AES.
def AES_Encrypt_ECB( d, key, plaintext ):
    D(d, "AES_Encrypt_ECB(key =", key, "data =", plaintext, ")")
    encobj = AES.new(bytes(key), AES.MODE_ECB)
    result = encobj.encrypt(bytes(plaintext))
    D(d+1, "result = ", result)
    return result

# B.4.1. Derive Key algorithm
# AES DUKPT key derivation function.
def Derive_Key( d, derivationKey, keyType, derivationData, deriveType ):
    D(d, "Derive_Key(derivationKey =", derivationKey, "keyType =", keyType,
"derivationData =", derivationData, "deriveType =", deriveType, ")")
    L = Key_Length(keyType)
    n = -(-L // 128)

    result = bytearray(n*16)
    for i in range(1, n+1):
        derivationData[1] = i
        result[(i-1)*16:i*16] = AES_Encrypt_ECB(d+2, derivationKey, derivationData)
    derivedKey = result[0:(L//8)]
    D(d+1, "derivedKey:", derivedKey)
    return derivedKey

# B.4.3. Create Derivation Data
# Compute derivation data for an AES DUKPT key derivation operation.
def Create_Derivation_Data( derivationPurpose, keyUsage, derivedKeyType,
initialKeyID, counter):
    derivationData = bytearray(16)
    derivationData[0] = 1
    derivationData[1] = 1

    if (keyUsage == KeyUsage._KeyEncryptionKey):
        derivationData[2:4] = [0,2]
    elif (keyUsage == KeyUsage._PINEncryption):
        derivationData[2:4] = [16,0]
    elif (keyUsage == KeyUsage._MessageAuthenticationGeneration):
        derivationData[2:4] = [32,0]
    elif (keyUsage == KeyUsage._MessageAuthenticationVerification):
        derivationData[2:4] = [32,1]
    elif (keyUsage == KeyUsage._MessageAuthenticationBothWays):
        derivationData[2:4] = [32,2]
    elif (keyUsage == KeyUsage._DataEncryptionEncrypt):
        derivationData[2:4] = [48,0]
    elif (keyUsage == KeyUsage._DataEncryptionDecrypt):
        derivationData[2:4] = [48,1]
    elif (keyUsage == KeyUsage._DataEncryptionBothWays):
        derivationData[2:4] = [48,2]
    elif (keyUsage == KeyUsage._KeyDerivation):
        derivationData[2:4] = [128,0]
    elif (keyUsage == KeyUsage._KeyDerivationInitialKey):
        derivationData[2:4] = [128,1]
    else:
        assert False

    if (derivedKeyType == KeyType._2TDEA):
        derivationData[4:6] = [0,0]
    elif (derivedKeyType == KeyType._3TDEA):
        derivationData[4:6] = [0,1]
    elif (derivedKeyType == KeyType._AES128):
        derivationData[4:6] = [0,2]
    elif (derivedKeyType == KeyType._AES192):
        derivationData[4:6] = [0,3]
    elif (derivedKeyType == KeyType._AES256):
        derivationData[4:6] = [0,4]
    else:
        assert False

    if (derivedKeyType == KeyType._2TDEA):
        derivationData[6:8] = [0,128]
    elif (derivedKeyType == KeyType._3TDEA):
        derivationData[6:8] = [0,192]
    elif (derivedKeyType == KeyType._AES128):
        derivationData[6:8] = [0,128]
    elif (derivedKeyType == KeyType._AES192):
        derivationData[6:8] = [0,192]
    elif (derivedKeyType == KeyType._AES256):
        derivationData[6:8] = [1,0]
    else:
        assert False

    if (derivationPurpose == DerivationPurpose._InitialKey):
        derivationData[8:16] = initialKeyID[0:8]
    elif (derivationPurpose == DerivationPurpose._DerivationOrWorkingKey):
        derivationData[8:12] = initialKeyID[4:8]
        derivationData[12:16] = IntToBytes(counter)
    else:
        assert False

    return derivationData

# B.5. Derive Initial Key
# Derive the initial key for a particular initialKeyID from a BDK.
def Derive_Initial_Key( d, BDK, keyType, initialKeyID ):
    D(d, "Derive_Initial_Key(BDK =", BDK, "keyType =", keyType, "initialKeyID =",
initialKeyID, ")")
    derivationData = Create_Derivation_Data(DerivationPurpose._InitialKey,
KeyUsage._KeyDerivationInitialKey, keyType, initialKeyID, 0)
    D(d+1, "derivationData: ", derivationData)
    initialKey = Derive_Key(d+2, BDK, keyType, derivationData, keyType)
    D(d+1, "initialKey: ", initialKey)

    return initialKey

# B.5. Host Derive Working Key
# Derive a working key for a particular transaction based on a initialKeyID and
counter.
def Host_Derive_Working_Key( d, BDK, deriveKeyType, workingKeyUsage, workingKeyType,
initialKeyID, counter):
    D(d, "Host_Derive_Working_Key(BDK =", BDK, "workingKeyUsage =", workingKeyUsage,
"workingKeyType =", workingKeyType, "initialKeyID =", initialKeyID, "counter =",
counter, ")")
    initialKey = Derive_Initial_Key(d+2, BDK, deriveKeyType, initialKeyID)
    D(d+1, "initialKey:", initialKey)

    mask = 0x80000000
    workingCounter = 0
    derivationKey = initialKey

    while mask > 0:
        if (mask & counter) != 0:
            D(d+1, "BIT FOUND:", mask)
            workingCounter = workingCounter | mask
            derivationData =
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey,
KeyUsage._KeyDerivation, deriveKeyType, initialKeyID, workingCounter)
            D(d+1, "derivationData:", derivationData)
            derivationKey = Derive_Key(d+2, derivationKey, deriveKeyType,
derivationData, deriveKeyType)
            D(d+1, "derivationKey:", derivationKey)
        mask = mask >> 1

    D(d+1, "FINAL DERIVATION:")
    derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, workingKeyUsage, 
workingKeyType, initialKeyID, counter)
    D(d+1, "derivationData", derivationData)
    workingKey = Derive_Key(d+2, derivationKey, workingKeyType, derivationData, 
deriveKeyType)
    D(d+1, "workingKey", workingKey)

    return derivationKey, derivationData, workingKey

# B.6.3. Processing Routines
# Load an initial key for computing terminal transaction keys in sequence.
def Load_Initial_Key( d, initialKey, deriveKeyType, initialKeyID ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Load_Initial_Key:", initialKey, deriveKeyType, initialKeyID)

    gIntermediateDerivationKeyRegister = [None]*NUMREG
    gIntermediateDerivationKeyInUse = [False]*NUMREG

    gIntermediateDerivationKeyRegister[0] = initialKey
    gIntermediateDerivationKeyInUse[0] = True
    gDeviceID = initialKeyID
    gCounter = 0
    gShiftRegister = 1
    gCurrentKey = 0
    gDeriveKeyType = deriveKeyType

    Update_Derivation_Keys(d+1, NUMREG-1, deriveKeyType)
    gCounter = gCounter + 1

# B.6.3. Update Initial Key
# Load a new terminal initial key under a pre-existing terminal initial key.
def Update_Initial_Key( d, encryptedInitialKey, initialKeyType, newDeviceID ):
    D(d, "Update_Initial_Key:", encryptedInitialkey, initialKeyType, newDeviceID)
    if (gCounter > ((1 << NUMREG) - 1)):
        Cease_Operation()
        return False

    derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, 
KeyUsage._KeyEncryptingKey, initialKeyType, gDeviceID, gCounter)
    D(d+1, "derivationData:", derivationData)
    keyEncryptionKey = DeriveKey(d+2, FutureKeyRegister[CurrentKey], initialKeyType, 
derivationData)
```

```
    D(d, "keyEncryptionKey:", keyEncryptionKey)

    n = (Key_Length(initialKeyType)+127)//128
    for i in range(1,n):
        newInitialKey[(i-1)*16:i*16] = AES_Decrypt_ECB(d+1, keyEncryptionKey, 
encryptedInitialKey[(i-1)*16:i*16])

    Load_Initial_Key(d+1, newInitialKey, newDeviceID)

    return True

# B.6.3. Generate Working Keys
# Generate a transaction key from the intermediate derivation key registers, and 
update the state to prepare for the next transaction.
def Generate_Working_Keys( d, workingKeyUsage, workingKeyType ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Generate_Working_Keys", workingKeyUsage, workingKeyType)

    Set_Shift_Register(d+1)

    while not gIntermediateDerivationKeyInUse[gCurrentKey]:
        D(d+1, "Key", gCurrentKey, "not in use")
        gCounter = gCounter + gShiftRegister
        if gCounter > ((1 << NUMREG) - 1):
            Cease_Operation()
            return False
        Set_Shift_Register(d+1)

    D(d+1, "gCounter:", gCounter)

    derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, workingKeyUsage, 
workingKeyType, gDeviceID, gCounter)
    D(d+1, "derivationData:", derivationData)
    assert gIntermediateDerivationKeyInUse[gCurrentKey]
    workingKey = Derive_Key(d+2, gIntermediateDerivationKeyRegister[gCurrentKey], 
workingKeyType, derivationData, gDeriveKeyType)
    D(d+1, "workingKey:", workingKey)

    Update_State_for_next_Transaction(d+2)
    return workingKey

# B.6.3. Update State for next Transaction
# Move the counter forward, and derive new intermediate derivation keys for the next 
transaction.
def Update_State_for_next_Transaction(d):
    global NUMREG
    global MAX_WORK
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Update_State_for_next_Transaction")

    oneBits = Count_One_Bits(gCounter)
    if oneBits <= MAX_WORK:
        Update_Derivation_Keys(d+2, gCurrentKey, gDeriveKeyType)
        gIntermediateDerivationKeyRegister[gCurrentKey] = 0
        gIntermediateDerivationKeyInUse[gCurrentKey] = False
        gCounter = gCounter + 1
    else:
        gIntermediateDerivationKeyRegister[gCurrentKey] = 0
        gIntermediateDerivationKeyInUse[gCurrentKey] = False
        gCounter = gCounter + gShiftRegister
    D(d+1, "gCounter:", gCounter)

    if gCounter > (1 << NUMREG) - 1:
        CeaseOperation()
        return False
    else:
        return True

# B.6.3. Update Derivation Keys
# Update all the intermediate derivation key registers below a certain point.
# This is used to:
#    1. Update all the intermediate derivation key registers below the shift
register after computing a transaction key.
#    2. Update all the intermediate derivation key registers when an initial key is
loaded.
def Update_Derivation_Keys( d, start, deriveKeyType ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister

    D(d, "Update_Derivation_Keys", deriveKeyType)

    i = start
    j = 1 << start

    D(d+1, "gCurrentKey:", gCurrentKey)
    baseKey = gIntermediateDerivationKeyRegister[gCurrentKey]
    while j != 0:
        D(d+1, "i:", i, "gShiftRegister:", j)
        derivationData =
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey,
KeyUsage._KeyDerivation, deriveKeyType, gDeviceID, gCounter | j)
        D(d+1, "derivationData:", derivationData)
        assert gIntermediateDerivationKeyInUse[gCurrentKey]
        gIntermediateDerivationKeyRegister[i] = Derive_Key(d+1, baseKey,
deriveKeyType, derivationData, deriveKeyType)
        gIntermediateDerivationKeyInUse[i] = True
        j = j >> 1
        i = i - 1

    return True

# B.6.3. Set Shift Register
# Set the shift register to the value of the rightmost '1' bit in the counter.
def Set_Shift_Register(d):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister

    gShiftRegister = 1
    gCurrentKey = 0

    if gCounter == 0:
        D(d, "Set_Shift_Register -> gShiftRegister:", gShiftRegister, "gCurrentKey:",
gCurrentKey)
        return True

    while (gShiftRegister & gCounter) == 0:
        gShiftRegister = gShiftRegister << 1
        gCurrentKey = gCurrentKey + 1

    D(d, "Set_Shift_Register -> gShiftRegister:", gShiftRegister, "gCurrentKey:",
gCurrentKey)
    return True

# Generate a trace of the first eight transactions
def GenerateTestData(bdk, bk, initialKeyID, k):
    print("")
    D(0, "Test Vectors for generating ", k, " from ", bk, " Base Derivation Key")

    max = (1 << NUMREG) - 1
    for i in [1,2,3,4,5,6,7,8]:
        print("")
        D(0, "Counter: ", i, "   (", hex(i), ")")

        keyPIN = Host_Derive_Working_Key(0, bdk, bk, KeyUsage._PINEncryption, k,
initialKeyID, i)

        print("")
        D(0, "PIN Encryption Key:              ", keyPIN[2])
```

```
######################## MAIN ##################################

NUMREG = 32
MAX_WORK = 16

bdk = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1 ]
bdk192 = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1,
         0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10 ]
bdk256 = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1,
         0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1]

initialKeyID = [ 0x12, 0x34, 0x56, 0x78, 0x90, 0x12, 0x34, 0x56 ]

derivationData = Create_Derivation_Data( DerivationPurpose._InitialKey,
KeyUsage._KeyDerivationInitialKey, KeyType._AES128, initialKeyID, 0 )

GenerateTestData(bdk, KeyType._AES128, initialKeyID, KeyType._AES128)
```

2.4 Debugging Trace (Terminal Algorithm)

```
from Crypto.Cipher import AES
from enum import Enum
import binascii

# print out a hexadecimal number in groups of 8 capitalized digits.  Only for
debugging.
def ToGroups(x):
   for i in range(0, len(x)//8):
      print(x[i*8:(i+1)*8].upper(), " ", end="")

# Convert a 32-bit integer to a list of bytes in big-endian order.  Used to convert
counter values to byte lists.
def IntToBytes(x):
   return [((x >> i) & 0xff) for i in (24,16,8,0)]

# Print out a debug message at depth d.  Takes an arbitrary list of strings and byte
lists or bytearrays.
# Lists of bytes are pretty-printed in hex.
def D(d, *args):
   for i in range(0,2*d):
      print(" ", end="")
   for x in args:
      if type(x) is str:
         print(x, " ", end="")
      elif type(x) is list:
         value = binascii.hexlify(bytearray(x)).decode("utf-8")
         ToGroups(value)
      elif type(x) is bytearray:
         value = binascii.hexlify(x).decode("utf-8")
         ToGroups(value)
      elif type(x) is bytes:
            value = binascii.hexlify(x).decode("utf-8")
            ToGroups(value)
        else:
            print(x, " ", end="")
    print("")

# B.3.1. Enumerations
class DerivationPurpose(Enum):
    _InitialKey = 0
    _DerivationOrWorkingKey = 1

class KeyType(Enum):
    _2TDEA = 0
    _3TDEA = 1
    _AES128 = 2
    _AES192 = 3
    _AES256 = 4

class KeyUsage(Enum):
    _KeyEncryptionKey = 0x0002
    _PINEncryption = 0x1000
    _MessageAuthenticationGeneration = 0x2000
    _MessageAuthenticationVerification = 0x2001
    _MessageAuthenticationBothWays = 0x2002
    _DataEncryptionEncrypt = 0x3000
    _DataEncryptionDecrypt = 0x3001
    _DataEncryptionBothWays = 0x3002
    _KeyDerivation = 0x8000
    _KeyDerivationInitialKey = 9

# Count the number of 1 bits in a counter value.  Readable, but not efficient.
def Count_One_Bits(x):
    bits = 0
    mask = 1 << (NUMREG-1)
    while mask > 0:
        if x & mask:
            bits = bits + 1
        mask = mask >> 1
    return bits

# B.3.2. Key Length function
# Length of an algorithm's key, in bits.
def Key_Length( keyType ):
    if ( keyType == KeyType._2TDEA):
        return 128
    if ( keyType == KeyType._3TDEA):
        return 192
    if ( keyType == KeyType._AES128):
        return 128
    if ( keyType == KeyType._AES192):
        return 192
    if ( keyType == KeyType._AES256):
        return 256
    assert False
```

```
# Encrypt plaintext with key using AES.
def AES_Encrypt_ECB( d, key, plaintext ):
    D(d, "AES_Encrypt_ECB(key =", key, "data =", plaintext, ")")
    encobj = AES.new(bytes(key), AES.MODE_ECB)
    result = encobj.encrypt(bytes(plaintext))
    D(d+1, "result = ", result)
    return result

# B.4.1. Derive Key algorithm
# AES DUKPT key derivation function.
def Derive_Key( d, derivationKey, keyType, derivationData, deriveType ):
    D(d, "Derive_Key(derivationKey =", derivationKey, "keyType =", keyType,
"derivationData =", derivationData, "deriveType =", deriveType, ")")
    L = Key_Length(keyType)
    n = -(-L // 128)

    result = bytearray(n*16)
    for i in range(1, n+1):
        derivationData[1] = i
        result[(i-1)*16:i*16] = AES_Encrypt_ECB(d+2, derivationKey, derivationData)

    derivedKey = result[0:(L//8)]
    D(d+1, "derivedKey:", derivedKey)
    return derivedKey

# B.4.3. Create Derivation Data
# Compute derivation data for an AES DUKPT key derivation operation.
def Create_Derivation_Data( derivationPurpose, keyUsage, derivedKeyType,
initialKeyID, counter):
    derivationData = bytearray(16)
    derivationData[0] = 1
    derivationData[1] = 1

    if (keyUsage == KeyUsage._KeyEncryptionKey):
        derivationData[2:4] = [0,2]
    elif (keyUsage == KeyUsage._PINEncryption):
        derivationData[2:4] = [16,0]
    elif (keyUsage == KeyUsage._MessageAuthenticationGeneration):
        derivationData[2:4] = [32,0]
    elif (keyUsage == KeyUsage._MessageAuthenticationVerification):
        derivationData[2:4] = [32,1]
    elif (keyUsage == KeyUsage._MessageAuthenticationBothWays):
        derivationData[2:4] = [32,2]
    elif (keyUsage == KeyUsage._DataEncryptionEncrypt):
        derivationData[2:4] = [48,0]
    elif (keyUsage == KeyUsage._DataEncryptionDecrypt):
        derivationData[2:4] = [48,1]
    elif (keyUsage == KeyUsage._DataEncryptionBothWays):
        derivationData[2:4] = [48,2]
    elif (keyUsage == KeyUsage._KeyDerivation):
        derivationData[2:4] = [128,0]
    elif (keyUsage == KeyUsage._KeyDerivationInitialKey):
        derivationData[2:4] = [128,1]
    else:
        assert False
    if (derivedKeyType == KeyType._2TDEA):
        derivationData[4:6] = [0,0]
    elif (derivedKeyType == KeyType._3TDEA):
        derivationData[4:6] = [0,1]
    elif (derivedKeyType == KeyType._AES128):
        derivationData[4:6] = [0,2]
    elif (derivedKeyType == KeyType._AES192):
        derivationData[4:6] = [0,3]
    elif (derivedKeyType == KeyType._AES256):
        derivationData[4:6] = [0,4]
    else:
        assert False

    if (derivedKeyType == KeyType._2TDEA):
        derivationData[6:8] = [0,128]
    elif (derivedKeyType == KeyType._3TDEA):
        derivationData[6:8] = [0,192]
    elif (derivedKeyType == KeyType._AES128):
        derivationData[6:8] = [0,128]
    elif (derivedKeyType == KeyType._AES192):
        derivationData[6:8] = [0,192]
    elif (derivedKeyType == KeyType._AES256):
        derivationData[6:8] = [1,0]
    else:
        assert False

    if (derivationPurpose == DerivationPurpose._InitialKey):
        derivationData[8:16] = initialKeyID[0:8]
    elif (derivationPurpose == DerivationPurpose._DerivationOrWorkingKey):
        derivationData[8:12] = initialKeyID[4:8]
        derivationData[12:16] = IntToBytes(counter)
    else:
        assert False

    return derivationData

# B.5. Derive Initial Key
# Derive the initial key for a particular initialKeyID from a BDK.
def Derive_Initial_Key( d, BDK, keyType, initialKeyID ):
    D(d, "Derive_Initial_Key(BDK =", BDK, "keyType =", keyType, "initialKeyID =",
initialKeyID, ")")
    derivationData = Create_Derivation_Data(DerivationPurpose._InitialKey,
KeyUsage._KeyDerivationInitialKey, keyType, initialKeyID, 0)
    D(d+1, "derivationData: ", derivationData)
    initialKey = Derive_Key(d+2, BDK, keyType, derivationData, keyType)
    D(d+1, "initialKey: ", initialKey)

    return initialKey

# B.5. Host Derive Working Key
# Derive a working key for a particular transaction based on a initialKeyID and
counter.
def Host_Derive_Working_Key( d, BDK, deriveKeyType, workingKeyUsage, workingKeyType,
initialKeyID, counter):
    D(d, "Host_Derive_Working_Key:", BDK, workingKeyUsage, workingKeyType, 
initialKeyID, counter)
    initialKey = Derive_Initial_Key(d+2, BDK, deriveKeyType, initialKeyID)
    D(d+1, "initialKey:", initialKey)

    mask = 0x80000000
    workingCounter = 0
    derivationKey = initialKey

    while mask > 0:
        if (mask & counter) != 0:
            D(d+1, "BIT FOUND:", mask)
            workingCounter = workingCounter | mask
            derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, 
KeyUsage._KeyDerivation, deriveKeyType, initialKeyID, workingCounter)
            D(d+1, "derivationData:", derivationData)
            derivationKey = Derive_Key(d+2, derivationKey, deriveKeyType, 
derivationData, deriveKeyType)
            D(d+1, "derivationKey:", derivationKey)
        mask = mask >> 1

    D(d+1, "FINAL DERIVATION:")
    derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, workingKeyUsage, 
workingKeyType, initialKeyID, counter)
    D(d+1, "derivationData", derivationData)
    workingKey = Derive_Key(d+2, derivationKey, workingKeyType, derivationData, 
deriveKeyType)
    D(d+1, "workingKey", workingKey)

    return derivationKey, derivationData, workingKey

# B.6.3. Processing Routines
# Load an initial key for computing terminal transaction keys in sequence.
def Load_Initial_Key( d, initialKey, deriveKeyType, initialKeyID ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Load_Initial_Key(initialKey =", initialKey, "deriveKeyType =", 
deriveKeyType, "initialKeyID =", initialKeyID, ")")

    gIntermediateDerivationKeyRegister = [None]*NUMREG
    gIntermediateDerivationKeyInUse = [False]*NUMREG

    gIntermediateDerivationKeyRegister[0] = initialKey
    D(d+1, "gIntermediateDerivationKeyRegister[0] <-", initialKey)
    gIntermediateDerivationKeyInUse[0] = True
    gDeviceID = initialKeyID
    D(d+1, "gDeviceID <-", gDeviceID)
    gCounter = 0
    D(d+1, "gCounter <-", gCounter)
    gShiftRegister = 1
    D(d+1, "gShiftRegister <-", gShiftRegister)
    gCurrentKey = 0
    gDeriveKeyType = deriveKeyType
    D(d+1, "gDeriveKeyType <-", gDeriveKeyType)

    Update_Derivation_Keys(d+1, NUMREG-1, deriveKeyType)
    gCounter = gCounter + 1
    D(d+1, "gCounter <-", gCounter)

# B.6.3. Update Initial Key
# Load a new terminal initial key under a pre-existing terminal initial key.
def Update_Initial_Key( d, encryptedInitialKey, initialKeyType, newDeviceID ):
    D(d, "Update_Initial_Key:", encryptedInitialkey, initialKeyType, newDeviceID)
    if (gCounter > ((1 << NUMREG) - 1)):
        Cease_Operation()
        return False

    derivationData =
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey,
KeyUsage._KeyEncryptingKey, initialKeyType, gDeviceID, gCounter)
    D(d+1, "derivationData:", derivationData)
    keyEncryptionKey = DeriveKey(d+2, FutureKeyRegister[CurrentKey], initialKeyType,
derivationData)
    D(d, "keyEncryptionKey:", keyEncryptionKey)

    n = (Key_Length(initialKeyType)+127)//128
    for i in range(1,n):
        newInitialKey[(i-1)*16:i*16] = AES_Decrypt_ECB(d+1, keyEncryptionKey,
encryptedInitialKey[(i-1)*16:i*16])

    Load_Initial_Key(d+1, newInitialKey, newDeviceID)

    return True

# B.6.3. Generate Working Keys
# Generate a transaction key from the intermediate derivation key registers, and
update the state to prepare for the next transaction.
def Generate_Working_Keys( d, workingKeyUsage, workingKeyType ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Generate_Working_Keys(workingKeyUsage =", workingKeyUsage, "workingKeyType
=", workingKeyType, ")")

    Set_Shift_Register(d+1)
    while not gIntermediateDerivationKeyInUse[gCurrentKey]:
        D(d+1, "Key", gCurrentKey, "not in use")
        gCounter = gCounter + gShiftRegister
        D(d+1, "gCounter <-", gCounter)
        if gCounter > ((1 << NUMREG) - 1):
            Cease_Operation()
            return False
        Set_Shift_Register(d+1)

    D(d+1, "gCounter:", gCounter)

    derivationData = 
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey, workingKeyUsage, 
workingKeyType, gDeviceID, gCounter)
    D(d+1, "derivationData:", derivationData)
    assert gIntermediateDerivationKeyInUse[gCurrentKey]
    workingKey = Derive_Key(d+2, gIntermediateDerivationKeyRegister[gCurrentKey], 
workingKeyType, derivationData, gDeriveKeyType)
    D(d+1, "workingKey:", workingKey)

    Update_State_for_next_Transaction(d+2)
    return workingKey

# B.6.3. Update State for next Transaction
# Move the counter forward, and derive new intermediate derivation keys for the next 
transaction.
def Update_State_for_next_Transaction(d):
    global NUMREG
    global MAX_WORK
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister
    global gDeriveKeyType

    D(d, "Update_State_for_next_Transaction()")

    oneBits = Count_One_Bits(gCounter)
    if oneBits <= MAX_WORK:
        Update_Derivation_Keys(d+2, gCurrentKey, gDeriveKeyType)
        gIntermediateDerivationKeyRegister[gCurrentKey] = 0
        D(d+1, "gIntermediateDerivationKeyRegister[", gCurrentKey, "] <-", 0, ")")
        gIntermediateDerivationKeyInUse[gCurrentKey] = False
        gCounter = gCounter + 1
    else:
        gIntermediateDerivationKeyRegister[gCurrentKey] = 0
        D(d+1, "gIntermediateDerivationKeyRegister[", gCurrentKey, "] <-", 0, ")")
        gIntermediateDerivationKeyInUse[gCurrentKey] = False
        gCounter = gCounter + gShiftRegister
    D(d+1, "gCounter <-", gCounter)

    if gCounter > (1 << NUMREG) - 1:
        CeaseOperation()
        return False
    else:
        return True

# B.6.3. Update Derivation Keys
# Update all the intermediate derivation key registers below a certain point.
# This is used to:
#    1. Update all the intermediate derivation key registers below the shift
register after computing a transaction key.
#    2. Update all the intermediate derivation key registers when an initial key is
loaded.
def Update_Derivation_Keys( d, start, deriveKeyType ):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister

    D(d, "Update_Derivation_Keys(deriveKeyType =", deriveKeyType, ")")

    i = start
    j = 1 << start

    D(d+1, "gCurrentKey:", gCurrentKey)
    baseKey = gIntermediateDerivationKeyRegister[gCurrentKey]
    D(d+1, "baseKey:", baseKey)
    while j != 0:
        D(d+1, "i:", i, "gShiftRegister:", j)
        derivationData =
Create_Derivation_Data(DerivationPurpose._DerivationOrWorkingKey,
KeyUsage._KeyDerivation, deriveKeyType, gDeviceID, gCounter | j)
        D(d+1, "derivationData:", derivationData)
        assert gIntermediateDerivationKeyInUse[gCurrentKey]
        gIntermediateDerivationKeyRegister[i] = Derive_Key(d+1, baseKey,
deriveKeyType, derivationData, deriveKeyType)
        D(d+1, "gIntermediateDerivationKeyRegister[", i, "] <- ",
gIntermediateDerivationKeyRegister[i])
        gIntermediateDerivationKeyInUse[i] = True
        j = j >> 1
        i = i - 1

    return True

# B.6.3. Set Shift Register
# Set the shift register to the value of the rightmost '1' bit in the counter.
def Set_Shift_Register(d):
    global NUMREG
    global gIntermediateDerivationKeyRegister
    global gIntermediateDerivationKeyInUse
    global gCurrentKey
    global gDeviceID
    global gCounter
    global gShiftRegister

    gShiftRegister = 1
    gCurrentKey = 0

    if gCounter == 0:
        D(d, "Set_Shift_Register -> gShiftRegister:", gShiftRegister, "gCurrentKey:",
gCurrentKey)
        return True

    while (gShiftRegister & gCounter) == 0:
        gShiftRegister = gShiftRegister << 1
        gCurrentKey = gCurrentKey + 1

    D(d, "Set_Shift_Register -> gShiftRegister:", gShiftRegister, "gCurrentKey:",
gCurrentKey)
    return True

# Generate a full trace of the first eight transactions
def GenerateTestData(bdk, bk, initialKeyID, k):
    print("")
    D(0, "Test Vectors for generating ", k, " from ", bk, " Base Derivation Key")
    initialKey = Derive_Initial_Key(0, bdk, bk, initialKeyID)
    print("")
    D(0, "Initial Key:                     ", initialKey)
    print("")
    Load_Initial_Key(0, initialKey, bk, initialKeyID )

    max = (1 << NUMREG) - 1
    for i in [1,2,3,4,5,6,7,8]:
        print("")
        D(0, "Counter: ", i, "   (", hex(i), ")")

        keyPIN = Generate_Working_Keys( 0, KeyUsage._PINEncryption, k)

        print("")
        D(0, "PIN Encryption Key:              ", keyPIN)

######################## MAIN ##################################

NUMREG = 32
MAX_WORK = 16

bdk = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1 ]
bdk192 = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1,
         0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10 ]
bdk256 = [ 0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1,
         0xFE, 0xDC, 0xBA, 0x98, 0x76, 0x54, 0x32, 0x10, 0xF1, 0xF1, 0xF1, 0xF1,
0xF1, 0xF1, 0xF1, 0xF1]

initialKeyID = [ 0x12, 0x34, 0x56, 0x78, 0x90, 0x12, 0x34, 0x56 ]
```

```
derivationData = Create_Derivation_Data( DerivationPurpose._InitialKey,
KeyUsage._KeyDerivationInitialKey, KeyType._AES128, initialKeyID, 0 )

GenerateTestData(bdk, KeyType._AES128, initialKeyID, KeyType._AES128)
```