

可信计算

实验报告

姓名：

学号：

教师：

2020 年 12 月

**实验二 TPM综合实验**

**一、实验目的与配置**

1. 实验目的
2. 了解TPM的核心原理
3. 掌握TPM的几种加密方式及其加密原理
4. 掌握NV和PCR机制
5. 建议实验环境

Win7或Win10系统

1. 建议实验工具

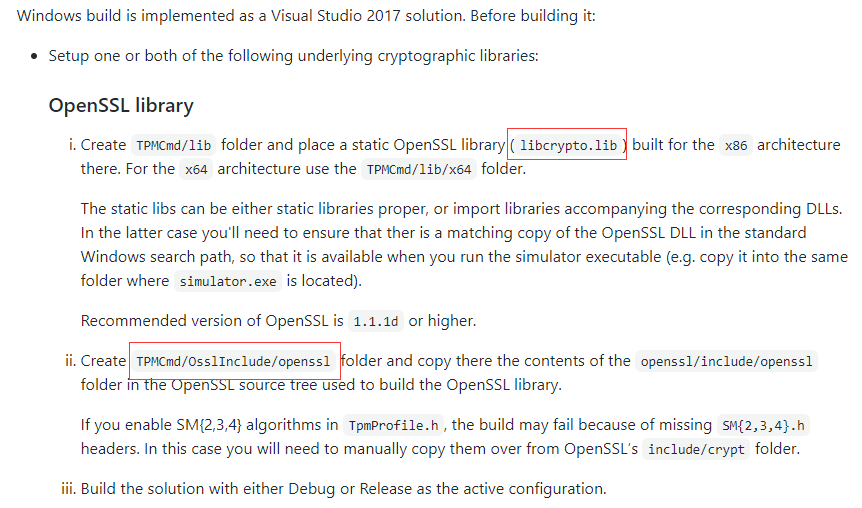
Visual Studio 2017

**二、实验过程与步骤**

**1、下载源码编译TPM Simulator模拟器**

地址：<https://github.com/microsoft/ms-tpm-20-ref>

这里还需要添加几个库到项目文件夹中

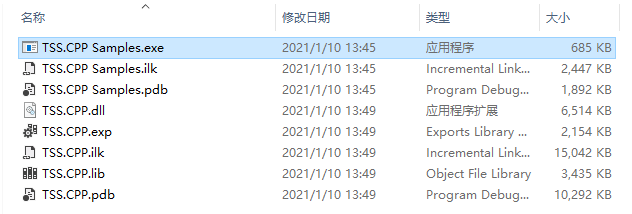


编译完成后生成simulator.exe模拟器，双击打开，等待TSS软件栈的运行



**2、编译TSS软件栈**

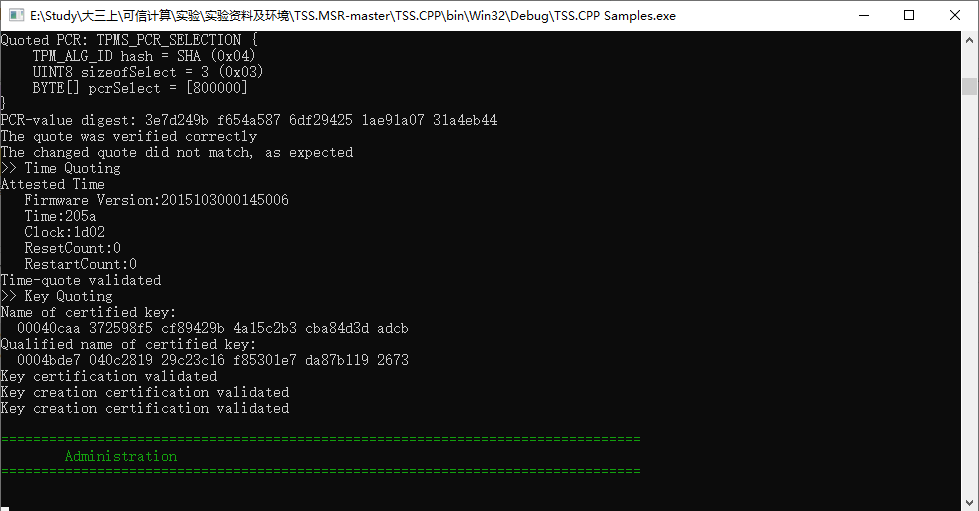
编译完成后运行TSS.CPP Samples.exe



Simulator监听到客户端的开启



TSS软件栈运行中



**3、分析几项加密模块的实现**

**（1）Encrypt Decrypt Sample()**

功能：实现AES加密

过程：

1. 首先建立一个初始化存储空间的TPM密钥prim，然后再利用创建一个公钥
2. 输入待加密的数组toEncrypt
3. 利用参数aesHandle，CFB，iv和待加密数组toEncrypt进行AES运算
4. 进行解密
5. 在控制台输出明文以及其加密解密结果

void Samples::EncryptDecryptSample()

{

Announce("EncryptDecryptSample");

TPM\_HANDLE prim = MakeStoragePrimary();

// Make an AES key

TPMT\_PUBLIC inPublic(TPM\_ALG\_ID::SHA256,

TPMA\_OBJECT::decrypt | TPMA\_OBJECT::sign | TPMA\_OBJECT::userWithAuth

| TPMA\_OBJECT::sensitiveDataOrigin,

null,

TPMS\_SYMCIPHER\_PARMS(Aes128Cfb),

TPM2B\_DIGEST\_SYMCIPHER());

auto aesKey = tpm.Create(prim, null, inPublic, null, null);

TPM\_HANDLE aesHandle = tpm.Load(prim, aesKey.outPrivate, aesKey.outPublic);

ByteVec toEncrypt { 1, 2, 3, 4, 5, 4, 3, 2, 12, 3, 4, 5 };

ByteVec iv(16);

auto encrypted = tpm.EncryptDecrypt(aesHandle, (BYTE)0, TPM\_ALG\_ID::CFB, iv, toEncrypt);

auto decrypted = tpm.EncryptDecrypt(aesHandle, (BYTE)1, TPM\_ALG\_ID::CFB, iv, encrypted.outData);

cout << "AES encryption" << endl <<

"in: " << toEncrypt << endl <<

"enc: " << encrypted.outData << endl <<

"dec: " << decrypted.outData << endl;

\_ASSERT(decrypted.outData == toEncrypt);

tpm.FlushContext(prim);

tpm.FlushContext(aesHandle);

} // EncryptDecryptSample()

**（2）Hash()**

功能：进行Hash散列值运算

过程：

1. 输入待计算的数据data
2. 进行hash运算
3. 在控制台输出原数据值及其hash值

void Samples::Hash()

{

Announce("Hash");

vector<TPM\_ALG\_ID> hashAlgs = { TPM\_ALG\_ID::SHA1, TPM\_ALG\_ID::SHA256 };

ByteVec accumulator;

ByteVec data1 { 1, 2, 3, 4, 5, 6 };

cout << "Simple Hashing" << endl;

for (auto it = hashAlgs.begin(); it != hashAlgs.end(); it++)

{

auto hashResponse = tpm.Hash(data1, \*it, TPM\_RH\_NULL);

auto expected = Crypto::Hash(\*it, data1);

\_ASSERT(hashResponse.outHash == expected);

cout << "Hash:: " << EnumToStr(\*it) << endl;

cout << "Expected: " << expected << endl;

cout << "TPM generated: " << hashResponse.outHash << endl;

}

cout << "Hash sequences" << endl;

for (auto iterator = hashAlgs.begin(); iterator != hashAlgs.end(); iterator++) {

auto hashHandle = tpm.HashSequenceStart(null, \*iterator);

accumulator.clear();

for (int j = 0; j < 10; j++) {

// Note the syntax below. If no explicit sessions are provided then the

// library automatically uses PWAP with the authValue contained in the handle.

// If you want to mix PWAP and other sessions then you can use the psuedo-PWAP

// session as below.

AUTH\_SESSION mySession = AUTH\_SESSION::PWAP();

tpm[mySession].SequenceUpdate(hashHandle, data1);

accumulator = Helpers::Concatenate(accumulator, data1);

}

accumulator = Helpers::Concatenate(accumulator, data1);

// Note that the handle is flushed by the TPM when the sequence is completed

auto hashVal = tpm.SequenceComplete(hashHandle, data1, TPM\_RH\_NULL);

auto expected = Crypto::Hash(\*iterator, accumulator);

\_ASSERT(hashVal.result == expected);

cout << "Hash:: " << EnumToStr(\*iterator) << endl;

cout << "Expected: " << expected << endl;

cout << "TPM generated: " << hashVal.result << endl;

}

// We can also do an "event sequence"

auto hashHandle = tpm.HashSequenceStart(null, TPM\_ALG\_NULL);

accumulator.clear();

// We can also do an "event sequence"

auto hashHandle = tpm.HashSequenceStart(null, TPM\_ALG\_NULL);

accumulator.clear();

for (int j = 0; j < 10; j++) {

tpm.SequenceUpdate(hashHandle, data1);

accumulator = Helpers::Concatenate(accumulator, data1);

}

accumulator = Helpers::Concatenate(accumulator, data1);

// Note that the handle is flushed by the TPM when the sequence is completed

auto initPcr = tpm.PCR\_Read({{TPM\_ALG\_ID::SHA1, 0}});

auto hashVal2 = tpm.EventSequenceComplete(TPM\_HANDLE::Pcr(0), hashHandle, data1);

auto expected = Crypto::Hash(TPM\_ALG\_ID::SHA1, accumulator);

auto finalPcr = tpm.PCR\_Read({{TPM\_ALG\_ID::SHA1, 0}});

// Is this what we expect?

TPM\_HASH expectedPcr(TPM\_ALG\_ID::SHA1, initPcr.pcrValues[0]);

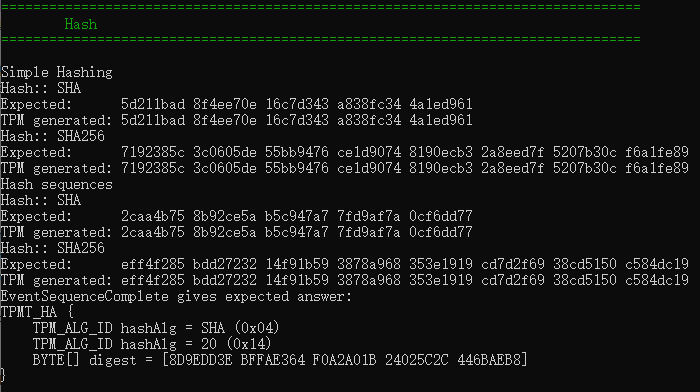
expectedPcr.Extend(expected);

if (expectedPcr == finalPcr.pcrValues[0])

cout << "EventSequenceComplete gives expected answer: " << endl << expectedPcr.ToString(false) << endl;

\_ASSERT(expectedPcr == finalPcr.pcrValues[0]);

} // Hash()



**（3）Rsa Encrypt Decrypt()**

功能：实现RSA加密

过程：

1. 输入待加密的明文dataToEncypt
2. 生成随机数，并利用随机数产生一对公私钥
3. 使用公钥加密，私钥解密
4. 在控制台输出明文和密文

void Samples::RsaEncryptDecrypt()

{

Announce("RsaEncryptDecrypt");

// This sample demostrates the use of the TPM for RSA operations.

// We will make a key in the "null hierarchy".

TPMT\_PUBLIC primTempl(TPM\_ALG\_ID::SHA1,

TPMA\_OBJECT::decrypt | TPMA\_OBJECT::userWithAuth | TPMA\_OBJECT::sensitiveDataOrigin,

null, // No policy

TPMS\_RSA\_PARMS(null, TPMS\_SCHEME\_OAEP(TPM\_ALG\_ID::SHA1), 2048, 65537),

TPM2B\_PUBLIC\_KEY\_RSA());

// Create the key

auto storagePrimary = tpm.CreatePrimary(TPM\_RH\_NULL, null, primTempl, null, null);

TPM\_HANDLE& keyHandle = storagePrimary.handle;

ByteVec dataToEncrypt = TPM\_HASH::FromHashOfString(TPM\_ALG\_ID::SHA1, "secret");

cout << "Data to encrypt: " << dataToEncrypt << endl;

auto enc = tpm.RSA\_Encrypt(keyHandle, dataToEncrypt, TPMS\_NULL\_ASYM\_SCHEME(), null);

cout << "RSA-encrypted data: " << enc << endl;

auto dec = tpm.RSA\_Decrypt(keyHandle, enc, TPMS\_NULL\_ASYM\_SCHEME(), null);

cout << "decrypted data: " << dec << endl;

if (dec == dataToEncrypt)

cout << "Decryption worked" << endl;

\_ASSERT(dataToEncrypt == dec);

// Now encrypt using TSS.C++ library functions

ByteVec mySecret = Helpers::RandomBytes(20);

enc = storagePrimary.outPublic.Encrypt(mySecret, null);

dec = tpm.RSA\_Decrypt(keyHandle, enc, TPMS\_NULL\_ASYM\_SCHEME(), null);

cout << "My secret: " << mySecret << endl;

cout << "My decrypted secret: " << dec << endl;

\_ASSERT(mySecret == dec);

// Now with padding

ByteVec pad { 1, 2, 3, 4, 5, 6, 0 };

enc = storagePrimary.outPublic.Encrypt(mySecret, pad);

dec = tpm.RSA\_Decrypt(keyHandle, enc, TPMS\_NULL\_ASYM\_SCHEME(), pad);

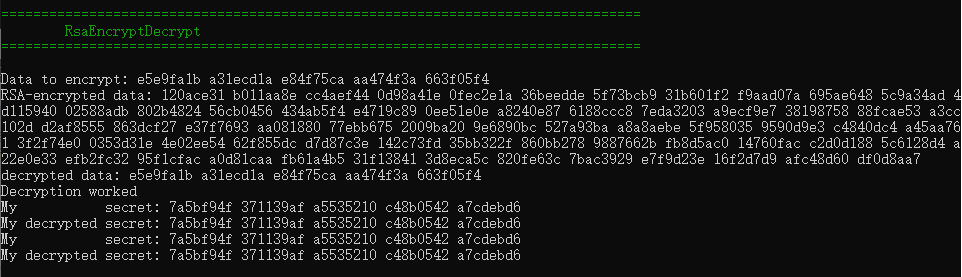
cout << "My secret: " << mySecret << endl;

cout << "My decrypted secret: " << dec << endl;

\_ASSERT(mySecret == dec);

tpm.FlushContext(keyHandle);

} // RsaEncryptDecrypt()



**（4）HMAC()**

功能：密钥相关的Hash运算消息认证码

原理：HMAC的定义中用到了一个密码散列函数H和一个密钥K，利用Hash算法，以一个消息M和一个密钥K作为输入，生成一个定长的消息摘要作为输出。

过程：

1. 在密钥K候面填充0，使其长度达到指定byte长度
2. 将填充好0的密钥K与ipad逐比特异或
3. 将数据流text附加到异或的结果后面
4. 对附加了text的数据流计算Hash摘要
5. 将填充好0的密钥K与opad逐比特异或
6. 将（4）中生成的消息摘要附加到（5）生成的数据流后
7. 对（6）中的数据流计算Hash摘要，输出摘要

void Samples::HMAC()

{

Announce("HMAC");

// Key and data to be HMACd

ByteVec key { 5, 4, 3, 2, 1, 0 };

ByteVec data1 { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };

auto hashAlg = TPM\_ALG\_ID::SHA1;

// To do an HMAC we need to load a key into the TPM. A primary key is easiest.

// template for signing/symmetric HMAC key with data originating externally

TPMT\_PUBLIC templ(TPM\_ALG\_ID::SHA256, TPMA\_OBJECT::sign | TPMA\_OBJECT::fixedParent |

TPMA\_OBJECT::fixedTPM | TPMA\_OBJECT::userWithAuth,

null,

TPMS\_KEYEDHASH\_PARMS(TPMS\_SCHEME\_HMAC(hashAlg)),

TPM2B\_DIGEST\_KEYEDHASH());

// The key is passed in in the SENSITIVE\_CREATE structure

TPMS\_SENSITIVE\_CREATE sensCreate(null, key);

// "Create" they key based on the externally provided keying data

auto newPrimary = tpm.CreatePrimary(TPM\_RH::OWNER, sensCreate, templ, null, null);

TPM\_HANDLE keyHandle = newPrimary.handle;

TPM\_HANDLE hmacHandle= tpm.HMAC\_Start(keyHandle, null, TPM\_ALG\_ID::SHA1);

tpm.SequenceUpdate(hmacHandle, data1);

auto hmacDigest = tpm.SequenceComplete(hmacHandle, data1, TPM\_RH\_NULL);

auto data = Helpers::Concatenate(data1, data1);

auto expectedHmac = Crypto::HMAC(hashAlg, key, data);

\_ASSERT(expectedHmac == hmacDigest.result);

cout << "HMAC[SHA1] of " << data << endl <<

"with key " << key << endl <<

" = " << hmacDigest.result << endl;

// We can also just TPM2\_Sign() with an HMAC key

auto sig = tpm.Sign(keyHandle, data, TPMS\_NULL\_SIG\_SCHEME(), null);

TPM\_HASH \*sigIs = dynamic\_cast<TPM\_HASH\*>(&\*sig);

cout << "HMAC[SHA1] of " << data << endl <<

"with key " << key << endl <<

" = " << sigIs->digest << endl;

// Or use the HMAC signing command

ByteVec sig3 = tpm.HMAC(keyHandle, data, TPM\_ALG\_ID::SHA1);

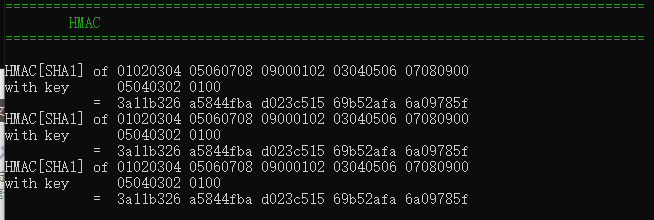
cout << "HMAC[SHA1] of " << data << endl <<

"with key " << key << endl <<

" = " << sig3 << endl;

tpm.FlushContext(keyHandle);

} // HMAC()



**（5）NV（）**

功能：非易失性存储器

介绍：物理性的非易失性存储器，当断开电源后，其内部所存储的数据不会消失。这里使用NV来模拟非易失性存储器的功能。

代码中定义了一些数据结构，来保证在平台“断电”后其内部的数据不会消失

void Samples::NV()

{

Announce("NV");

// Several types of NV-slot use are demonstrated here: simple, counter, bitfield, and extendable

ByteVec nvAuth = Helpers::RandomBytes(20);

TPM\_HANDLE nvHandle = RandomNvHandle();

// Try to delete the slot if it exists

tpm.\_AllowErrors()

.NV\_UndefineSpace(TPM\_RH::OWNER, nvHandle);

// CASE 1 - Simple NV-slot: Make a new simple NV slot, 16 bytes, RW with auth

TPMS\_NV\_PUBLIC nvTemplate(nvHandle, // Index handle

TPM\_ALG\_ID::SHA256, // Name-alg

TPMA\_NV::AUTHREAD | // Attributes

TPMA\_NV::AUTHWRITE,

null, // Policy

16); // Size in bytes

tpm.NV\_DefineSpace(TPM\_RH::OWNER, nvAuth, nvTemplate);

// We have set the authVal to be nvAuth, so set it in the handle too.

nvHandle.SetAuth(nvAuth);

// Write some data

ByteVec toWrite { 1, 2, 3, 4, 5, 4, 3, 2, 1 };

tpm.NV\_Write(nvHandle, nvHandle, toWrite, 0);

// And make sure that it's good

\_ASSERT(equal(toWrite.begin(), toWrite.end(), dataRead.begin()));

// We can also read the public area

auto nvPub = tpm.NV\_ReadPublic(nvHandle);

cout << "NV Slot public area:" << endl << nvPub.ToString(false) << endl;

// And then delete it

tpm.NV\_UndefineSpace(TPM\_RH::OWNER, nvHandle);

// CASE 2 - Counter NV-slot

TPMS\_NV\_PUBLIC nvTemplate2(nvHandle, // Index handle

TPM\_ALG\_ID::SHA256, // Name-alg

TPMA\_NV::AUTHREAD | // Attributes

TPMA\_NV::AUTHWRITE |

TPMA\_NV::COUNTER,

null, // Policy

8); // Size in bytes

tpm.NV\_DefineSpace(TPM\_RH::OWNER, nvAuth, nvTemplate2);

// We have set the authVal to be nvAuth, so set it in the handle too.

nvHandle.SetAuth(nvAuth);

// Should not be able to write (increment only)

tpm.\_ExpectError(TPM\_RC::ATTRIBUTES)

.NV\_Write(nvHandle, nvHandle, toWrite, 0);

// Should not be able to read before the first increment

tpm.\_ExpectError(TPM\_RC::NV\_UNINITIALIZED)

.NV\_Read(nvHandle, nvHandle, 8, 0);

// First increment

tpm.NV\_Increment(nvHandle, nvHandle);

// Should now be able to read

ByteVec beforeIncrement = tpm.NV\_Read(nvHandle, nvHandle, 8, 0);

cout << "Initial counter data: " << beforeIncrement << endl;

// Should be able to increment

for (int j = 0; j < 5; j++)

tpm.NV\_Increment(nvHandle, nvHandle);

// And make sure that it's good

ByteVec afterIncrement = tpm.NV\_Read(nvHandle, nvHandle, 8, 0);

cout << "After 5 increments: " << afterIncrement << endl;

// And then delete it

tpm.NV\_UndefineSpace(TPM\_RH::OWNER, nvHandle);

// CASE 3 - Bitfield

TPMS\_NV\_PUBLIC nvTemplate3(nvHandle, // Index handle

TPM\_ALG\_ID::SHA256, // Name-alg

TPMA\_NV::AUTHREAD | // Attributes

TPMA\_NV::AUTHWRITE |

TPMA\_NV::BITS,

null, // Policy

8); // Size in bytes

tpm.NV\_DefineSpace(TPM\_RH::OWNER, nvAuth, nvTemplate3);

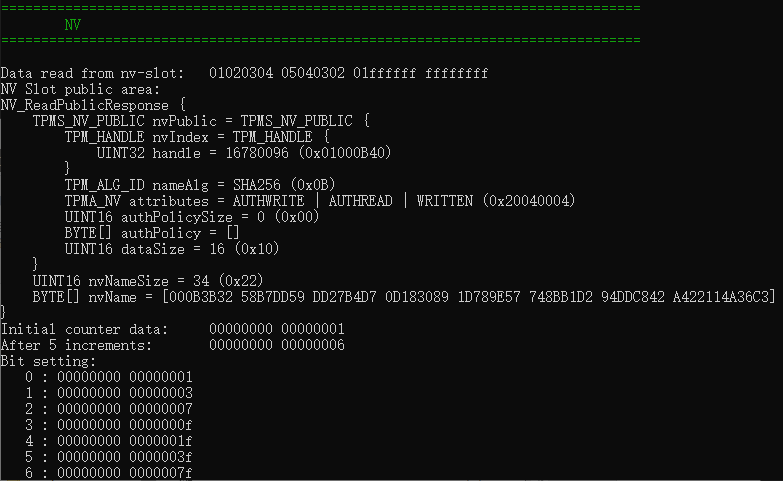
// We have set the authVal to be nvAuth, so set it in the handle too.

nvHandle.SetAuth(nvAuth);

// Should not be able to write (bitfield)

tpm.\_ExpectError(TPM\_RC::ATTRIBUTES)

.NV\_Write(nvHandle, nvHandle, toWrite, 0);



**（6）PRC（）**

功能：平台配置寄存器(PCR,Platform Con-figurationRegister)

介绍：TPM利用此功能实现可信计算的“可信传递”。在可信传递过程中,TPM 对影响平台完整性的实体进行度量,并将度量事件记入存储度量日志,然后通过“扩展(Extend)操作”将度量值存储到其内部的平台配置寄存器。实体询问时,TPM 忠实报告PCR 中的值。此功能允许平台进入任何状态,但平台不能对其进入或退出这种状态进行隐瞒或修改。

void Samples::PCR()

{

Announce("PCR");

// Modify PCR0 via a PCR\_Event, and print out the value

ByteVec toEvent { 1, 2, 3 };

auto afterEvent = tpm.PCR\_Event(TPM\_HANDLE::Pcr(0), toEvent);

cout << "PCR after event:" << endl << afterEvent[0].ToString() << endl;

vector<TPMS\_PCR\_SELECTION> toReadArray = { {TPM\_ALG\_ID::SHA1, 0},

{TPM\_ALG\_ID::SHA256, 1} };

// Get the initial values of two PCRs: one SHA1, and one SHA256

auto initVals = tpm.PCR\_Read(toReadArray);

cout << "Initial value:" << endl << initVals.ToString(false) << endl;

// Used by PCR\_Read to read PCR0 in the SHA1 bank

vector<TPMS\_PCR\_SELECTION> toReadPcr0 = { {TPM\_ALG\_ID::SHA1, 0} };

// Modify PCR0 via event

auto newVals = tpm.PCR\_Event(TPM\_HANDLE::Pcr(0), toEvent);

auto pcrVals = tpm.PCR\_Read(toReadPcr0);

cout << "SHA1 After Event:" << endl << pcrVals.pcrValues[0].ToString() << endl;

// Now modify the SHA1 PCR0 via extend

TPM\_HASH toExtend = TPM\_HASH::FromHashOfString(TPM\_ALG\_ID::SHA1, "abc");

tpm.PCR\_Extend(TPM\_HANDLE::Pcr(0), {toExtend});

// Now read SHA1:PCR0 again

pcrVals = tpm.PCR\_Read(toReadPcr0);

cout << "SHA1 After Extend:" << endl << pcrVals.pcrValues[0].ToString() << endl;

TPM\_HASH pcrAtEnd(TPM\_ALG\_ID::SHA1, pcrVals.pcrValues[0]);

//Check that this answer is what we expect

TPM\_HASH pcrSim(TPM\_ALG\_ID::SHA1, initVals.pcrValues[0]);

pcrSim.Event(toEvent);

pcrSim.Extend(toExtend);

if (pcrSim == pcrAtEnd)

cout << "PCR values correct" << endl;

else {

cout << "Error: PCR values NOT correct" << endl;

\_ASSERT(FALSE);

}

// Extend a resettable PCR

UINT32 resettablePcr = 16;

tpm.PCR\_Event(TPM\_HANDLE::Pcr(resettablePcr), { 1, 2, 3 });

auto resettablePcrVal = tpm.PCR\_Read({{TPM\_ALG\_ID::SHA1, resettablePcr}});

cout << "Resettable PCR before reset: " << resettablePcrVal.pcrValues[0] << endl;

tpm.PCR\_Reset(TPM\_HANDLE::Pcr(resettablePcr));

resettablePcrVal = tpm.PCR\_Read({{TPM\_ALG\_ID::SHA1, resettablePcr}});

cout << "After reset: " << resettablePcrVal.pcrValues[0] << endl;

// Check it really is all zeros

\_ASSERT(resettablePcrVal.pcrValues[0].buffer == ByteVec(20));

} // PCR()

