**TPM 2.0 Cloudproxy prototype and protocol**

manferdelli@, tmroeder@

**Introduction**

This document describes Tpm 2.0 and a series of C++ demonstration programs that show how Tpm 2.0 is used to support Cloudproxy. For a description of Cloudproxy, see <http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf>.

Cloudproxy requires a hardware root of trust to measure booted images, store secrets for those images and “attest” or prove the validity and binding of keys to program identity and associated protection properties for system software. These capabilities were originally provided in Cloudproxy by Tpm 1.2. The wide-scale availability of Tpm 2.0 on newer machines along with Intel’s deployment of a “soft tpm 2.0” on Haswell chipsets makes it a “must support” for Cloudproxy.

Although Tpm 2.0 provides similar base functionality as Tpm 1.2 provides, and, in the case of Cloudproxy, ultimately identical capabilities, the interface is significantly different. Differences include:

* Support for additional modern algorithms like ECC and SHA-2.
* A more sophisticated authorization model for features and functions.
* Unified key migration between Tpm’s.
* A simplified and more powerful mechanism for attestation and “AIK” certification.

The machine level I/O interface for Tpm 1.2 and Tpm 2.0 is similar: Tpm 2.0 is a character device supporting reads and writes from /dev/tpm0 but the commands are quite different and incompatible. As with Tpm1.2, our interface uses no external code or libraries and implements commands directly by writing to /dev/tpm0 using our tpm2\_lib library. In Cloudproxy, the base system, either the OS or a hypervisor, “owns” the tpm device. The Cloudproxy interface virtualizes interactions with the Tpm for all software in the stack. Note that in Linux, the required Tpm 2.0 driver is in version 4 kernels and later.

The four voume TPM 2.0 specification is available at <http://www.trustedcomputinggroup.org/resources/tpm_library_specification>. It is voluminous. A novel feature of the Tpm 2.0 specification is that the formal behavioral model appears in the form of an executable “C” like language and all required “.h” files can be obtained directly from the specification. In fact, this specification code can be used to build a Tpm simulator. The TCG group has such a simulator and makes it available to TCG members; the simulator runs only on Windows but is accessible over a network using a TLS based protocol.

A book entitled “A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security,” by [Will Arthur](http://www.apress.com/author/author/view/id/7589), [David Challener is also available, although I did not find it that useful. Trousers does not support Tpm 2.0 but there is open source code, from Intel, at](http://www.apress.com/author/author/view/id/7590) <https://github.com/01org/TPM2.0-TSS>. Microsoft also has some nice C++ interface code, written in part by Paul England, at <https://tpm2lib.codeplex.com/> but it is currently available under a restricted license. I am told Microsoft plans to release this code under a UCB license in the future. One useful piece of documentation that was available for the Tpm 1.2 was a code primer in how to use the Tpm (using the /dev/tpm0 character interface). This was available in the form of some powerpoint slides presented by David Challener at CMU. That presentation also has a simplified “theory of operation.” I have not found a correspondingly simple description for TPM 2.0.

Our code for TPM 2.0 is available in the Cloudproxy repository ( <http://github.com/jlmucb/cloudproxy> ). The prototype code is in the directory cloudproxy/src/tpm2.

The prototype code runs under Linux *using a version 4 kernel or lat*er. Instructions for enabling and configuring the TPM are also in the Cloudproxy/src/tpm2 directory.

**Overall Description of Tpm 2.0**

Much of Tpm 2.0’s architecture is similar to tpm1.2. Programmatic interfaces are specified in tpm12.h and tpm20.h in the cloudproxy/src/tpm2 directory.

As with Tpm 1.2, all data passed to and retrieved from the Tpm is big endian format and, indeed, much of the interface code involves marshalling between tpm formats and native formats. Each Tpm object (keys, context, nvram) has an associated handle (a 32 bit identifier).

Commands to the TPM all have the same prefix format naming the authentication method, command-id and size of the command buffer; after the prefix, all commands have per command formats although there are common authentication data formats for commands that require authentication. Similarly, responses have a common prefix with authentication information, error code and size of response followed by command specific output. Once an object has an associated open handle, it can be closed using Tpm2\_FlushContext on the handle. The per command data transmitted to and returned by the Tpm are in specific marshallable structures (like, TPM2B\_DIGEST, TPM2B\_NAME, …) described in tpm20.h.

Tpm 2.0 has several standard owner handles for platform owner, admin owner and endorsement owner. Unlike Tpm 1.2, initialization is done entirely by firmware when the Tpm is enabled. At that time, all the owner authorization is cleared so all access is granted pending changes. Tpm2\_Startup and Tpm2\_Shutdown are handled entirely by firmware. You cannot “reinit” the Tpm from the OS, the moral equivalent of clearing the Tpm must be done via the firmware interface.

Critical to TPM 2.0 is the key hierarchy. The “endorsement key” is a storage root key. As with other storage roots, this is created by Tpm2\_CreatePrimary with the designated endorsement key handle (TPM\_RH\_ENDORSEMENT). Unlike other keys the endorsement key is always generated from a (per Tpm but permanent) seed so every time you create the Endorsement Key you get the same values. There are two other kinds of keys under an Endorsement Key parent: signing keys and sealing keys. Quote keys are a type of signing key. These are created with Tpm2\_CreateKey and have different values every time they are created. With the exception of the Endorsement Key, keys need to be loaded (using Tpm2\_Load) before they can be used. The mechanism for authenticating a quote key for the Tpm (the “AIK” in Tpm 1.2 vernacular) based on the EK has changed. See the discussion below.

Keys, like other TpmPM objects, have authorization information associated with them. With the exception of a sealed object (which requires an authorization session), we always use password protection and currently, the passwords are hard coded in the tests.

Keys can be saved using Tpm2\_SaveContext and restored using Tpm\_LoadContext. It is important to close unused handles because the number of allowable open handles is very limited on Tpm 2.0.

There are quite a few new Tpm 2.0 commands but we only need a few others:

Tpm2\_Unseal is used to unseal data objects sealed with CreateKey.

Tpm2\_Quote is used to “quote” data. A quote signs a statement naming some data to be quoted as well as the value of specified Program Configuration Registers.

Tpm2\_DefineNVSpace defines a name space for NvRam values.

Tpm2\_UndefineNVSpace clears a name space for NvRam values.

Tpm2\_ReadNvRam reads an NV register under authorization control.

Tpm2\_WriteNvRam writes an NV register under authorization control.

Tpm2\_EvictControl permanently allocates a TPM handle. Unless you call EvictControl, a handle does not survive reboot.

Tpm2\_GetRandom gets random bits.

Tpm2\_GetCapabilities retrieves Tpm capabilities and handles. We currently use this to Flushall handles in tpm2\_util.

Tpm2\_MakeCredential and Tpm2\_ActivateCredential are used in the Tpm 2.0 protocol that replaces CertifyAIK in Tpm 1.2. MakeCredential, takes a secret (credential) and encrypts it to a key (typically, the endorsement key), it then associates the protected secret with a non-exportable TPM object; the non-exportable object is identified with a non-spoofable name that is included in the MakeCredential object. MakeCredential unseals the object and if the named associated object is loaded, returns the value. We use this to certify a Program key in the following way: We Make a credential using the Endorsement key that is associated with a non-exportable Quote key. The quote key signs the named public key along with the attendant PCRs. KeyNegoServer verifies the quoted values and using data provided with a request naming the Quote key, quote endorsement key, does an offline MakeCredential on a random 16 byte value (the “protected credential”). It then signs the program key and encrypts the certificate under the protected credential value. It then returns all this to the requester. ActivateCredential (which will only work on the original machine with the specific non-exportable Quote object that quoted the program key request) unseals the credential and the unsealed credential is used to decrypt the signed Program key.

Note that Tpm2\_MakeCredential[[1]](#footnote-0) is not actually called in the Cloudproxy protocol described below, KeyNegoServer can carry out the MakeCredential procedure given the Endorsement Key, QuotePublic key and key name, and the quoted values. Of course, before signing the Program key, KeyNegoServer verifies the PCRs correspond to a known, trusted program. The PCR’s which should be named are not generally dictated by the spec. As with Tpm 1.2, conventionally, the PCR’s KeyNegoServer should check are PCR 17 and 18 and the BIOS PCR’s.

tmp2\_lib implements several other Tpm 2.0 commands but they are not currently required for Cloudproxy.

**Services provided by tpm2\_lib and tpm2\_util**

All the Tpm supporting code mentioned in the previous section is implemented in tpm2\_lib.cc along with some test and additional interface code set forth in Appendix 2. The library contains code that interprets and creates the quote structures as well as the MakeCredential structures so, KeyNegoserver can prepare the values required for ActivateCredential and verify PCRs from a Cloudproxy application requesting a Program Key signature.

tpm2\_util.cc implements some additional useful utility functions like GetRandom and is used to call several end-to-end tests involving a specific sequence of Tpm2 calls. These tests include:

bool Tpm2\_SealCombinedTest (..., int pcr\_num) which tests a seal-unseal sequence.

bool Tpm2\_QuoteCombinedTest(..., int pcr\_num) which implements a quote and verify quote sequence.

bool Tpm2\_KeyCombinedTest(..., int pcr\_num) which implements a sign-verify sequence.

bool Tpm2\_NvCombinedTest(...) which test the NvRam functions.

bool Tpm2\_ContextCombinedTest(...) which tests SaveContext and LoadContext.

bool Tpm2\_EndorsementCombinedTest(...) which tests Endorsement Key generation. Note that the generated EK is always the same value. As with Tpm 1.2, vendors don’t usually supply an Endorsement Cert (OEM’s charge a fee for this for enterprise customers). As with Tpm 1.2, the utilities below allow a cloud provider to sign an endorsement certificate and we continue to believe this will be the primary method of obtaining EK certs in clouds for Cloudproxy

**CloudProxy protocol**

The Cloudproxy protocol includes all the steps to provision a Cloudproxy application with a certified Program Key using an attest protocol. In the prototype code, each protocol step is implemented by a command line program for demonstration purposes but intermediate structures are saved in files rather than transmitted over a TCP channel as would be the case in actual use. Command line arguments for each utility are described in the code and sample arguments are provided in the test scripts testall.sh and prototest.sh.

There is one implemented and one proposed “helper” command utility related to the Cloudproxy protocol.

* The SigningInstructions utility specifies policy related to signing certs like endorsement certs and program certs specifying things like CA name and cert durations.
* The (to be written) PolicyInstructions utility specifies the conditions under which a Program Key certificate should be signed; in particular, it names the PCRs and their values required by KeyNegoServer (the signing authority for an application domain) in order to certify Program Keys.

Preparing the key infrastructure for Cloudproxy for an application domain, happens in three phases[[2]](#footnote-1):

1. The policy key is generated, self-signed and provisioned to the KeyNegoServer along with the per-application-domain policy specifying what Program Key requests should be signed.
2. The endorsement key for each physical machine is retrieved and signed by some authority key (we use the Policy Key in our examples) producing the endorsement certificate required for each physical machine. This operation usually happens once as the machine is originally provisioned.
3. Each Cloudproxy program generates a public/private key pair for the *Program Key* and uses the *Program Key Certification Protocol* described below to communicate with KeyNegoServer. KeyNegoServer, based on information provided by the requesting Cloudproxy application, will sign the public portion of the program key with the Policy Key. Cloudproxy programs can use this certificate to prove identity, isolation and negotiate bi-directional encrypted, integrity protected channels with other Cloudproxy programs.

Steps one and two of the Cloudproxy key provisioning procedure, using Tpm 2.0 are illustrated by series of utilities, they are:

1. Generation and Signing of that Policy Key: There is one policy key for each application. A “self signed” cert naming the public portion of the policy key is embedded in every application program instance and serves as a “root” for all policy decisions enforced in the application domain. The utility GeneratePolicyKey generates a 2048-bit RSA signing key and the utility SelfSignPolicyCert self signs the generated request.
2. Retrieving the Endorsement Key and Endorsement Certificate: Every every machine running a Cloudproxy program must have an Endorsement Certificate, signed by the Policy Key in our examples,, naming the public endorsement key. It would be nice if every an Endorsement Certificate were provided by the (hardware) platform supplier but it usually isn’t. The utility GetEndorsementKey retrieves the public key and machine name and the utility CloudProxySignEndorsementKey signs the endorsement key (in our examples, the endorsement key is signed by the Policy Key).
3. The utility CreateAndSaveCloudProxyKeyHierarchy generates the key hierarchy for a Cloudproxy program and saves it in a form suitable for reloading by the program at initialization. The utility RestoreCloudProxyKeyHierarchy demonstrates reloading the hierarchy. Each Cloudproxy key hierarchy consists of three key contexts:
   1. A primary key, created by Tpm2\_CreatePrimaryKey, which is the storage root for the program.
   2. A sealing key, created and sealed by Tpm2\_CreateKey and unsealed by Tpm2\_Unseal. This key is used to encrypt program secrets.
   3. A quote key, which is used in the Cloudproxy key certification protocol. The quote key is a signing key used in the Tpm2\_Quote operation.

Step three of the Cloudproxy key provisioning procedure is implemented as an on-line protocol between a Cloudproxy program and a server, KeyNegoServer, in a real application. This protocol consists of a three step process.

To illustrate the protocol, we provide three command line utilities which implement the protocol steps. Communication between participants of the Cloudproxy key provisioning procedure takes place using protobufs which are defined in Appendix 1. Our utilities simply store the protobufs in designated files.

The utility ClientGenerateProgramKeyRequest implements step 1 of the Cloudproxy protocol which would run in a Cloudproxy application. ClientGenerateProgramKeyRequest generates a public/private key pair. The private portion of the key is sealed and stored for later use. It collects the public key along with the machine’s endorsement certificate and a Tpm 2.0 quote naming the PCR state and the newly generated program public key as well as the name and qualified name of the quote key along with its public parameters. All this information is packaged in a protobuf (the program\_cert\_request\_message message) for transmission to KeyNegoServer.

The utility ServerSignProgramKeyRequest implements the actions taken by KeynegoServer in the “live” protocol. These includes the computations performed by MakeCredential and ActivateCredential above. The responses are packaged in a protobuf (the program\_cert\_response\_message message) and transmitted to the requesting application.

The utility ClientGetProgramKeyCert implements the final step carried out by the Cloudproxy application upon receipt of a successful response from Keynegoserver. This consists of retrieving the protected credential consisting of the encryption/integrity keys used to encrypt the signed Program Certificate using the Tpm2\_ActivateCredential function and decrypting and storing the unencrypted Program Certificate for later use.

**Source code and tests**

You can download the Cloudproxy repository from <https://github.com/jlmucb/cloudproxy>, the tpm sample code is in cloudproxy/src/tpm2. To make the library and the utilities, type

make -f tpm2.mak

after setting up the object and binary locations. To run the test scripts, **as root**, type

./prototest.sh

./testall.sh

in the binary directory.

To build with cmake, assuming the source directory (cloudproxy/src/tpm2) is in ${SRC} and a convenient build directory is ${OUT}:

* cd ${OUT}
* cmake -G Ninja ${SRC}
* ninja
* cd bin
* ${SRC}/prototest.sh && ${SRC}/testall.sh

Depending on the current state of the test scripts, this last step might require changing the path to the binaries.

**Acknowledgement**

Thanks to Paul England for many helpful discussions.

**Appendix - Protobufs for Cloudproxy Protocol**

message private\_key\_blob\_message {

required string key\_type = 1;

optional string key\_name = 2;

optional bytes blob = 3;

}

message rsa\_public\_key\_message {

optional string key\_name = 1;

required int32 bit\_modulus\_size = 2;

required bytes exponent = 3;

required bytes modulus = 4;

}

message rsa\_private\_key\_message {

required rsa\_public\_key\_message public\_key = 1;

optional bytes d = 2;

optional bytes p = 3;

optional bytes q = 4;

optional bytes dp = 5;

optional bytes dq = 6;

}

message asymmetric\_key\_message {

optional rsa\_private\_key\_message key = 1;

}

message public\_key\_message {

optional string key\_type = 1;

optional rsa\_public\_key\_message rsa\_key = 2;

}

message endorsement\_key\_message {

optional string machine\_identifier = 1;

optional bytes tpm2b\_blob = 2;

optional bytes tpm2\_name = 3;

}

message signing\_instructions\_message {

optional string issuer = 1;

optional int64 duration = 2;

optional string purpose = 3;

optional string date = 4;

optional string time = 5;

optional string sign\_alg = 6;

optional string hash\_alg = 7;

optional bool isCA = 8;

optional bool can\_sign = 9;

}

message x509\_cert\_request\_parameters\_message {

required string common\_name = 1;

optional string country\_name = 2;

optional string state\_name = 3;

optional string locality\_name = 4;

optional string organization\_name = 5;

optional string suborganization\_name = 6;

optional public\_key\_message key = 7;

}

message x509\_cert\_issuer\_parameters\_message {

required string common\_name = 1;

optional string country\_name = 2;

optional string state\_name = 3;

optional string locality\_name = 4;

optional string organization\_name = 5;

optional string suborganization\_name = 6;

optional string purpose = 7;

optional public\_key\_message key = 8;

}

message cert\_parameters\_message {

optional x509\_cert\_request\_parameters\_message request = 1;

optional x509\_cert\_issuer\_parameters\_message signer = 2;

optional string not\_before = 3;

optional string not\_after = 4;

}

message credential\_info\_message {

// public key parameters of "active-key"

optional public\_key\_message public\_key = 1;

// Tpm2 name (hash) of the "active-key" info

optional bytes name = 2;

// objectAttributes of the "active key"

optional int32 properties = 3;

}

message program\_key\_parameters {

optional string program\_name = 1;

optional string program\_key\_type = 2;

optional int32 program\_bit\_modulus\_size = 3;

optional bytes program\_key\_exponent = 4;

optional bytes program\_key\_modulus = 5;

};

message program\_cert\_request\_message {

optional string request\_id = 1;

optional bytes endorsement\_cert\_blob = 2;

optional program\_key\_parameters program\_key = 3;

optional string active\_sign\_alg = 4;

optional int32 active\_sign\_bit\_size = 5;

optional string active\_sign\_hash\_alg = 6;

optional bytes active\_signature = 7;

optional credential\_info\_message cred = 8;

optional bytes quoted\_blob = 9;

}

message program\_cert\_response\_message {

optional string request\_id = 1;

optional string program\_name = 2;

optional string integrity\_alg = 3;

// outer HMAC, does not include size in buffer

// HMAC key is KDFa derived from seed and "INTEGRITY"

// This is a TPM2B\_DIGEST and has a size.

optional bytes integrityHMAC = 4;

// encIdentity, does not include size of encIdentity in buffer.

// encIdentity should be an encrypted correctly marshalled

// This is an encrypted TPM2B\_DIGEST and has a size.

// encIdentity is always CFB Aes-128 encrypted

// with KDFa derived key derived from the "seed," "STORAGE" and

// the name of the active key.

optional bytes encIdentity = 5;

// protector-key private-key encrypted seed || "IDENTITY" buffer

optional bytes secret = 6;

// Signed, der-encoded program cert CTR encrypted with

// secret in credential buffer. TODO(jlm): should also

// contain an HMAC.

optional bytes encrypted\_cert = 7;

optional bytes encrypted\_cert\_hmac = 8;

}

message certificate\_chain\_entry\_message {

optional string subject\_key\_name = 1;

optional string issuer\_key\_name = 2;

optional string cert\_type = 3;

optional bytes cert\_blob = 4;

}

message certificate\_chain\_message {

repeated certificate\_chain\_entry\_message entry = 1;

}

message quote\_certification\_information {

optional bytes magic = 1;

optional bytes type = 2;

optional bytes qualifiedsigner = 3;

optional bytes extraData = 4;

optional bytes clockinfo = 5;

optional int64 firmwareversion = 6;

optional bytes pcr\_selection = 7;

optional bytes digest = 8;

}

**Appendix 2 - tpm2\_lib functions**

int Tpm2\_SetCommand(TPM\_ST tag, uint32\_t cmd, byte\* buf, int size\_param, byte\* params);

void Tpm2\_IntepretResponse(int out\_size, byte\* out\_buf, int16\_t\* cap, uint32\_t\* responseSize, uint32\_t\* responseCode);

int Tpm2\_Set\_OwnerAuthHandle(int size, byte\* buf);

int Tpm2\_Set\_OwnerAuthData(int size, byte\* buf)

bool Tpm2\_Startup(LocalTpm& tpm);

bool Tpm2\_Shutdown(LocalTpm& tpm);

bool Tpm2\_GetCapability(LocalTpm& tpm, uint32\_t cap, int\* size, byte\* buf);

bool Tpm2\_GetRandom(LocalTpm& tpm, int numBytes, byte\* buf);

bool Tpm2\_ReadClock(LocalTpm& tpm, uint64\_t\* current\_time, uint64\_t\* current\_clock);

bool Tpm2\_ReadPcrs(LocalTpm& tpm, TPML\_PCR\_SELECTION pcrSelect, uint32\_t\* updateCounter, TPML\_PCR\_SELECTION\* pcrSelectOut, TPML\_DIGEST\* values);

bool Tpm2\_ReadPcr(LocalTpm& tpm, int pcrNum, uint32\_t\* updateCounter, TPML\_PCR\_SELECTION\* pcrSelectOut, TPML\_DIGEST\* digest);

bool Tpm2\_CreatePrimary(LocalTpm& tpm, TPM\_HANDLE owner, string& authString, TPML\_PCR\_SELECTION& pcr\_selection, TPM\_ALG\_ID enc\_alg, TPM\_ALG\_ID int\_alg, TPMA\_OBJECT& flags, TPM\_ALG\_ID sym\_alg, TPMI\_AES\_KEY\_BITS sym\_key\_size, TPMI\_ALG\_SYM\_MODE sym\_mode, TPM\_ALG\_ID sig\_scheme, int mod\_size, uint32\_t exp, TPM\_HANDLE\* handle, TPM2B\_PUBLIC\* pub\_out);

bool Tpm2\_Load(LocalTpm& tpm, TPM\_HANDLE parent\_handle, string& parentAuth, int size\_public, byte\* inPublic, int size\_private, byte\* inPrivate, TPM\_HANDLE\* new\_handle, TPM2B\_NAME\* name);

bool Tpm2\_PolicyPassword(LocalTpm& tpm, TPM\_HANDLE handle);

bool Tpm2\_PCR\_Event(LocalTpm& tpm, int pcr\_num, uint16\_t size, byte\* eventData);

bool Tpm2\_PolicyGetDigest(LocalTpm& tpm, TPM\_HANDLE handle, TPM2B\_DIGEST\* digest\_out);

bool Tpm2\_StartAuthSession(LocalTpm& tpm, TPM\_RH tpm\_obj, TPM\_RH bind\_obj, TPM2B\_NONCE& initial\_nonce, TPM2B\_ENCRYPTED\_SECRET& salt, TPM\_SE session\_type, TPMT\_SYM\_DEF& symmetric, TPMI\_ALG\_HASH hash\_alg, TPM\_HANDLE\* session\_handle, TPM2B\_NONCE\* nonce\_obj);

bool Tpm2\_PolicyPcr(LocalTpm& tpm, TPM\_HANDLE session\_handle, TPM2B\_DIGEST& expected\_digest, TPML\_PCR\_SELECTION& pcr);

bool Tpm2\_PolicySecret(LocalTpm& tpm, TPM\_HANDLE handle,TPM2B\_DIGEST\* policy\_digest, TPM2B\_TIMEOUT\* timeout, TPMT\_TK\_AUTH\* ticket);

bool Tpm2\_CreateSealed(LocalTpm& tpm, TPM\_HANDLE parent\_handle,

int size\_policy\_digest, byte\* policy\_digest, string& parentAuth, int size\_to\_seal, byte\* to\_seal, TPML\_PCR\_SELECTION& pcr\_selection, TPM\_ALG\_ID int\_alg, TPMA\_OBJECT& flags, TPM\_ALG\_ID sym\_alg, TPMI\_AES\_KEY\_BITS sym\_key\_size, TPMI\_ALG\_SYM\_MODE sym\_mode, TPM\_ALG\_ID sig\_scheme, int mod\_size, uint32\_t exp, int\* size\_public, byte\* out\_public, int\* size\_private, byte\* out\_private, TPM2B\_CREATION\_DATA\* creation\_out, TPM2B\_DIGEST\* digest\_out, TPMT\_TK\_CREATION\* creation\_ticket);

bool Tpm2\_CreateKey(LocalTpm& tpm, TPM\_HANDLE parent\_handle,

string& parentAuth, string& authString, TPML\_PCR\_SELECTION& pcr\_selection, TPM\_ALG\_ID enc\_alg, TPM\_ALG\_ID int\_alg, TPMA\_OBJECT& flags, TPM\_ALG\_ID sym\_alg,TPMI\_AES\_KEY\_BITS sym\_key\_size, TPMI\_ALG\_SYM\_MODE sym\_mode, TPM\_ALG\_ID sig\_scheme, int mod\_size, uint32\_t exp,int\* size\_public, byte\* out\_public, int\* size\_private, byte\* out\_private, TPM2B\_CREATION\_DATA\* creation\_out, TPM2B\_DIGEST\* digest\_out, TPMT\_TK\_CREATION\* creation\_ticket);

bool Tpm2\_Unseal(LocalTpm& tpm, TPM\_HANDLE item\_handle, string& parentAuth, TPM\_HANDLE session\_handle, TPM2B\_NONCE& nonce, byte session\_attributes, TPM2B\_DIGEST& hmac\_digest, int\* out\_size, byte\* sealed);

bool Tpm2\_Quote(LocalTpm& tpm, TPM\_HANDLE signingHandle, string& parentAuth, int quote\_size, byte\* toQuote, TPMT\_SIG\_SCHEME scheme, TPML\_PCR\_SELECTION& pcr\_selection, TPM\_ALG\_ID sig\_alg, TPM\_ALG\_ID hash\_alg, int\* attest\_size, byte\* attest, int\* sig\_size, byte\* sig);

bool Tpm2\_LoadContext(LocalTpm& tpm, int size, byte\* saveArea, TPM\_HANDLE\* handle);

bool Tpm2\_SaveContext(LocalTpm& tpm, TPM\_HANDLE handle, int\* size, byte\* saveArea);

bool Tpm2\_FlushContext(LocalTpm& tpm, TPM\_HANDLE handle);

bool Tpm2\_ReadNv(LocalTpm& tpm, TPMI\_RH\_NV\_INDEX index, string& authString, uint16\_t size, byte\* data);

bool Tpm2\_WriteNv(LocalTpm& tpm, TPMI\_RH\_NV\_INDEX index, string& authString,uint16\_t size, byte\* data);

bool Tpm2\_DefineSpace(LocalTpm& tpm, TPM\_HANDLE owner, TPMI\_RH\_NV\_INDEX index, string& authString, uint16\_t size\_data);

bool Tpm2\_UndefineSpace(LocalTpm& tpm, TPM\_HANDLE owner, TPMI\_RH\_NV\_INDEX index);

bool Tpm2\_Flushall(LocalTpm& tpm);

bool Tpm2\_MakeCredential(LocalTpm& tpm, TPM\_HANDLE keyHandle, TPM2B\_DIGEST& credential, TPM2B\_NAME& objectName, TPM2B\_ID\_OBJECT\* credentialBlob, TPM2B\_ENCRYPTED\_SECRET\* secret);

bool Tpm2\_ActivateCredential(LocalTpm& tpm, TPM\_HANDLE activeHandle, TPM\_HANDLE keyHandle, string& activeAuth, string& keyAuth, TPM2B\_ID\_OBJECT& credentialBlob, TPM2B\_ENCRYPTED\_SECRET& secret, TPM2B\_DIGEST\* certInfo);

bool Tpm2\_Certify(LocalTpm& tpm, TPM\_HANDLE signedKey, TPM\_HANDLE signingKey, string& auth\_signed\_key, string& auth\_signing\_key, TPM2B\_DATA& qualifyingData, TPM2B\_ATTEST\* attest, TPMT\_SIGNATURE\* sig);

bool Tpm2\_ReadPublic(LocalTpm& tpm, TPM\_HANDLE handle, uint16\_t\* pub\_blob\_size, byte\* pub\_blob, TPM2B\_PUBLIC& outPublic, TPM2B\_NAME& name, TPM2B\_NAME& qualifiedName);

bool Tpm2\_Rsa\_Encrypt(LocalTpm& tpm, TPM\_HANDLE handle, string& authString, TPM2B\_PUBLIC\_KEY\_RSA& in, TPMT\_RSA\_DECRYPT& scheme, TPM2B\_DATA& label, TPM2B\_PUBLIC\_KEY\_RSA\* out);

bool Tpm2\_EvictControl(LocalTpm& tpm, TPMI\_RH\_PROVISION owner, TPM\_HANDLE handle, string& authString, TPMI\_DH\_PERSISTENT\* persistantHandle);

bool Tpm2\_DictionaryAttackLockReset(LocalTpm& tpm);

1. First aid: The padding method that should be used by an endorsement key to encrypt the credential is not terribly obvious but is in the sample code in tpm2\_lib. [↑](#footnote-ref-0)
2. Readers should consult the Cloudproxy Tao for Trusted Computing available at <http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf>

   to familiarize themselves with the operation of Cloudproxy and the purposes of each of the keys mentioned in this section. [↑](#footnote-ref-1)