

Formal Analysis of TPM Key Certification Protocols

©2023

Sarah Lavinia Johnson

B.S. in Computer Science, University of Kansas, 2021

B.S. in Mathematics, University of Kansas, 2021

Submitted to the graduate degree program in Department of Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Perry Alexander, Chairperson

Committee members

Michael Branicky

Emily Witt

Date defended: TODO

The Thesis Committee for Sarah Lavinia Johnson certifies
that this is the approved version of the following thesis:

Formal Analysis of TPM Key Certification Protocols

Perry Alexander, Chairperson

Date approved: TBD

Abstract

Abstract

Acknowledgements

Acknowledgements go here.

Contents

1	Introduction	1
2	Background	2
2.1	TPM 2.0	2
2.1.1	Keys	3
2.2	Inductive Propositions	6
3	Secure Device Identity	7
3.1	Certificate Chain	8
4	Execution Model	10
5	Identity Provisioning	16
5.1	Owner/Administrator Creation of LAK Certificate based on IAK Certificate	16
5.2	OEM Creation of IAK Certificate based on EK Certificate	21
6	Conclusion	23
6.1	Conclusion	23
6.2	Future Work	23
A	Model of Inference	25
B	Misc Stuff 2	27

List of Figures

2.1	Model of Keys	4
2.2	Model of Certificates	6
3.1	Model of Key Attribute Requirements	8
3.2	Key and Certificate Relationships [Trusted Computing Group (2021b)]	9
4.1	Model of Messages	10
4.2	Type Signature of Execute Relation	11
4.3	Model of Commands	11
4.4	Model of Command Sequences	14
4.5	Theorems on Sequential Execution	15
5.1	Parameters of LAK Certification Protocol	18
5.2	Model of Correct Steps of Owner	19
5.3	Minimal Initial State of Owner	19

List of Tables

3.1	Key Requirements and Recommendations	7
-----	--	---

Chapter 1

Introduction

Chapter 2

Background

2.1 TPM 2.0

A Trusted Platform Module (TPM) is a microcontroller that complies with the ISO/IEC 11889:2015 international standard. The TPM and its specification were designed by the Trusted Computing Group (TCG) to act as a hardware anchor for PC system security [Arthur et al. (2015)]. To this end, TPMs have the abilities necessary for secure generation of keys, algorithm agility, secure storage of keys, enhanced authorization, device health attestation, device identification, NVRAM storage and more.

The TPM's key generator is based on its own random number generator (RNG) so that it does not rely on external sources of randomness. These keys can be used for a multitude of purposes and may be created or destroyed as often as needed. Due to algorithm agility, the TPM can use nearly any cryptographic algorithm. As a result, keys may utilize asymmetric algorithms such as RSA or ECC, or they may utilize symmetric algorithms such as AES or DES. Additionally, a variety of key strengths (i.e., key sizes) and hash algorithms may be used. By design, keys stored within the TPM are protected against software attacks. Keys may optionally be further protected using enhanced authorization (EA). EA allows a key or other TPM entity to be authorized using a password, HMAC, or policy. This flexibility allows for varying complexities in the requirements for accessing an entity.

Device health attestation data provided by a TPM offers cryptographic proof of software state. Attestation data comes in the form of a quote which is a signed hash over a selection of platform configuration registers (PCRs). PCRs store the results from a chain of boot time measurements

in a way that guarantees integrity. In particular, a PCR cannot be rolled back to a previous value resulting in a measurement being undone. To tie attestation data to a specific device, the key that performed the signing operation must be cryptographically bound to that device. In order to understand this idea of device identification, we must look at a simplified version of the process for creating and distributing TPMs to the end user. The following capitalized titles of TPM Manufacturer, OEM, and Owner/Administrator are keywords that will be referenced throughout this paper.

1. TPM Manufacturers produce TPM chips according to the international standard. They provision each TPM chip with one or more certificates which bind a key to that specific TPM. These chips are then distributed to the original equipment manufacturers (OEMs).
2. OEMs produce devices (e.g., PCs) with these TPM chips integrated. They provision each TPM chip with one or more certificates which bind a key to that specific device. These devices are then distributed to the end users (Owners/Administrators).
3. Owners/Administrators may optionally provision their TPM chip(s) with one or more certificates which bind a key to that specific device.

These certificates are stored within the TPM's NVRAM providing protection from accidental erasure in the scenario that the device's hard drive gets wiped. The provisioning of device identification in Steps 2 and 3 is the subject of this paper.

2.1.1 Keys

All keys discussed in this document are TPM keys which utilize asymmetric cryptography. Although the TPM 2.0 has capabilities for utilizing symmetric keys, it is outside the scope of this paper. A key may be created using one of two commands.

- `TPM2_CreatePrimary`: A Primary key is produced based on the current Primary Seed. A Primary key may be persisted within the TPM. Otherwise it must be recreated after a TPM reset.

- `TPM2_Create`: An Ordinary key is produced based on a seed taken from the RNG. An Ordinary key is the child of another key; it is wrapped by that parent key. It may be persisted within the TPM or persisted external to the TPM in the form of an encrypted key blob. The blob is only loadable using the parent key’s authorization in the TPM that created it.

Keys have attributes that are set at creation-time. These attributes are permanent and include the following: `FixedTPM`, `Sign`, `Decrypt`, `Restricted`. The `FixedTPM` attribute indicates that the private key cannot be duplicated. All keys considered in this paper have this attribute set. A key pair with the `Sign` attribute set consists of a private signing key and a public signature-verification key. When properly handled, private signing keys can provide integrity, authenticity, and nonrepudiation. A key pair with the `Decrypt` attribute set consists of a public encryption key and a private decryption key. When properly handled, public encryption keys can provide confidentiality. A key with both the `Sign` and `Decrypt` attributes set is called a Combined key. US NIST SP800-57 disallows the use of Combined keys for the reason that it may weaken the security guarantees associated with one or both of the attributes. Moreover, a key pair may have the `Restricted` attribute set, limiting the operations of the private key to TPM generated data.

The Coq model inductively defines a `pubKey` and `privKey` type for public keys and private keys respectively. A key of either of these types requires a unique identifier and a sequence of boolean values describing whether a particular attribute is set or not set. A key pair consists of a `pubKey` and a `privKey` with the same identifier and attributes. The model does not differentiate between Primary and Ordinary keys.

```

Inductive pubKey : Type :=
| Public : keyID → Restricted → Sign → Decrypt → pubKey.

Inductive privKey : Type :=
| Private : keyID → Restricted → Sign → Decrypt → privKey.

```

Figure 2.1: Model of Keys

The `Restricted` attribute can provide important security implications. A restricted signing key may only sign a digest that has been produced by the TPM. Enforcement of this constraint is

reliant on a 4-byte magic value called `TPM_Generated` [Trusted Computing Group (2019)]. All structures that the TPM constructs from internal data begins with this value. Such structures include keys, platform configuration registers (PCRs), and audit digests. These structures contribute to two primary use cases for restricted signing keys: (1) key certification and (2) attestation. Use case 1 proves that a new key resides in the same TPM as some known restricted key. Use case 2 utilizes the `Restricted` attribute to provide assurance that a signature over PCRs or audit logs was in fact over a digest generated by that particular TPM. Additionally, a restricted signing key can sign data supplied to the TPM externally by using the `TPM2_Hash` command. In this case the `TPM2_Hash` command produces a ticket asserting that the TPM itself calculated this hash and will later sign it. A restricted signing key will not sign external data without this ticket. To prevent spoofing of another TPM's internal data as one's own, the `TPM2_Hash` command will only produce a ticket if the external data does not begin with the `TPM_Generated` value.

A restricted decryption key is called a storage key. Only storage keys can be used as parents to create or load child objects or to activate credentials [Arthur et al. (2015)]. All TPMs are shipped with an essential storage key: the endorsement key. The endorsement key (EK) is installed by the TPM Manufacturer and stored in a shielded location on the TPM. The corresponding EK certificate serves a significant role in the enrollment of secure device identifiers. This process will be discussed in further detail in later sections.

A certificate contains a public key and an identity and is signed by a trusted certificate authority. A certificate binds a public key to an identity. The term certificate specifically refers to an X.509 v3 digital certificate. The EK certificate includes the public part of the EK itself as well as various assertions regarding the security qualities and provenance of the TPM [Trusted Computing Group (2021a)]. The EK certificate binds the EK to a specific TPM. For keys created by entities other than the TPM manufacturer (i.e., the OEM and the Owner/Administrator), a certificate's identity field will contain non-TPM device information. This information should be globally unique per device [Institute of Electrical and Electronics Engineers (2018)]. In this model, certificates are defined as the `signedCert` type. A `signedCert` requires a public key, an identifier, and a private

key. An identifier may include information describing either the TPM or the device. The private key parameter denotes the key which performed the signature over the certificate.

```
Inductive signedCert : Type :=  
| Cert : pubKey → identifier → privKey → signedCert.  
  
Inductive identifier : Type :=  
| TPM_info : tpmInfoType → identifier  
| Device_info : deviceInfoType → identifier.
```

Figure 2.2: Model of Certificates

2.2 Inductive Propositions

[Pierce et al. (2022)]

Chapter 3

Secure Device Identity

A secure device identifier (DevID) is an identifier that is cryptographically bound to a device [Institute of Electrical and Electronics Engineers (2018)]. A device with DevID capability includes an Initial Device Identifier (IDevID) provided by the OEM. This IDevID must be stored in a way that protects it from modification. Since a TPM has capabilities to protect keys against compromise, it is an ideal choice for IDevID storage. Additionally, a device with DevID capability may support the creation of Locally Significant Device Identifiers (LDevIDs) by a device owner or network administrator (i.e., the Owner/Administrator). An LDevID cannot be transferred to a device with a different IDevID without knowledge of the private key used to produce the cryptographic binding.

When using a TPM key for secure device identity, there are restrictions on the attributes that it can have in order to enforce the best security practices; the key must have the `FixedTPM` and `Sign` attributes set and the `Decrypt` attribute not set. Furthermore the key may optionally have the `Restricted` attribute set. When the `Restricted` attribute is set, such a key is called an attestation key (AK). This DevID gets its special name due to its unique ability to be used as a parent node in a chain of certificates. The acronym AK is prefixed by the letter I or L denoting Initial or Locally Significant respectively.

Key	Type	FixedTPM	Signing	Decrypting	Restricted	Creator
EK	Primary	X		X	X	TPM Manufacturer
IAK	Primary	X	X		X	OEM
IDevID	Primary	X	X			OEM
LAK	Ordinary	X	X		X	Owner/Admin
LDevID	Ordinary	X	X			Owner/Admin

Table 3.1: Key Requirements and Recommendations

The attribute requirements displayed in Table 3.1 is modeled as a collection of functions. Each function takes a public key as input and returns a proposition.

```

Definition endorsementKey (k : pubKey) : Prop :=
  match k with
  | Public _ Restricting NonSigning Decrypting  $\Rightarrow$  True
  | _  $\Rightarrow$  False
  end.

Definition attestationKey (k : pubKey) : Prop :=
  match k with
  | Public _ Restricting Signing NonDecrypting  $\Rightarrow$  True
  | _  $\Rightarrow$  False
  end.

Definition devidKey (k : pubKey) : Prop :=
  match k with
  | Public _ NonRestricting Signing NonDecrypting  $\Rightarrow$  True
  | _  $\Rightarrow$  False
  end.

```

Figure 3.1: Model of Key Attribute Requirements

The issuers of device identity certificates are known as Certificate Authorities (CAs). CAs are further identified by the creator of the keys that they certify (i.e., the CA that issues certificates for IAKs and IDevIDs is known as the OEM’s CA and the CA that issues certificates for LAKs and LDevIDs is known as the Owner/Administrator’s CA). The OEM’s CA must carefully verify the attributes and TPM residency of a key before signing a certificate due to the important security and identity implications provided by these certificates. All CAs should support a standard certificate transport protocol that provides confidentiality, integrity, and protection from replay attacks [Trusted Computing Group (2021b)]. These transport protocols are outside the scope of this paper. We will assume CAs to be following this recommendation precisely.

3.1 Certificate Chain

A chain of certificates can be used to verify a chain of trust to some trust anchor. The IAK certificate typically acts as this trust anchor.

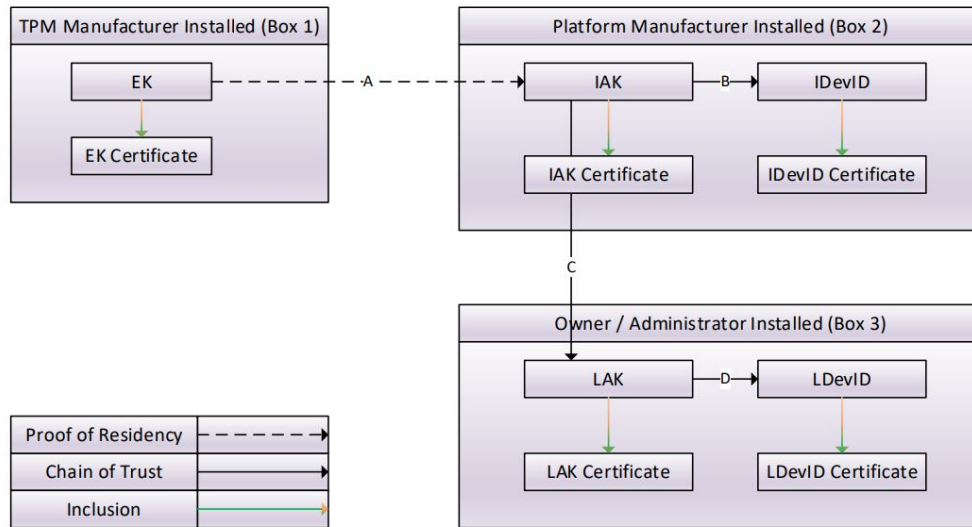


Figure 3.2: Key and Certificate Relationships [Trusted Computing Group (2021b)]

- Box 1: The EK certificate is signed by the TPM Manufacturer’s CA and binds the EK to a specific TPM.
- Line A: The IAK is verified by the OEM’s CA to have the correct key properties and to be resident in the same TPM as the EK.
- Line B: The IDevID is verified by the OEM’s CA to have the correct key properties and to be resident in the same TPM as the IAK.
- Box 2: The IAK certificate and IDevID certificate is signed by the OEM’s CA and binds the IAK and IDevID to a specific device.
- Line C: The LAK is verified by the Owner/Administrator’s CA to have the correct key properties and to be resident in the same TPM as the IAK.
- Line D: The LDevID is verified by the Owner/Administrator’s CA to have the correct key properties and to be resident in the same TPM as the LAK.
- Box 3: The LAK certificate and LDevID certificate is signed by the Owner/Administrator’s CA.

Chapter 4

Execution Model

The protocols used to enroll DevID certificates require both TPM and non-TPM commands to be performed by the CA and the requesting device. A command may rely on a variety of parameters such as keys, nonces, certificates, as well as other messages. A message includes all of the structures that an entity may use or produce. The message type is an abstract representation of these

```
Inductive message : Type :=  
| publicKey : pubKey → message  
| privateKey : privKey → message  
| hash : message → message  
| signature : message → privKey → message  
| TPM2B_Attest : pubKey → message  
| encryptedCredential : message → randType → pubKey → message  
| randomNum : randType → message  
| TCG_CSR_IDevID : identifier → signedCert → pubKey → message  
| TCG_CSR_LDevID : message → signedCert → message  
| signedCertificate : signedCert → message  
| pair : message → message → message.
```

Figure 4.1: Model of Messages

structures. From a message, additional messages may be inferred. For example, given a message `signature m k`, `m` may be deduced. Whereas given a message `encryptedCredential m g k`, no messages may be deduced. This concept is modeled in two ways: as a recursive function `inferFrom` and as an inductive proposition `inferred`. These two definitions are proven to be equivalent (see Appendix A for definitions and proof). In particular, additional information may be gained from signatures, TPM2B_Attest structures, certificate signing requests (CSRs), public key certificates, and pairs of messages. All other messages either contain no additional information

(i.e., keys and random numbers) or the information is concealed (i.e., hash digests and encryptions).

Each command and its execution is modeled abstractly. We do not attempt to model the computational intricacies of true cryptography. Command execution is defined as an inductive proposition relating an initial state, a command, and a final state. The `tpm_state` and `state` types are aliases

Inductive `execute : tpm_state * state → command → tpm_state * state → Prop`

Figure 4.2: Type Signature of Execute Relation

for a list of messages. These types are implemented as a list only for convenience; they are treated as a set in all practical aspects (i.e., ordering and duplicates are ignored). TPM state contains all messages produced by a command performed on an entity's own TPM. Whereas the general state contains all messages known to an entity and is correspondingly a superset of the TPM state.

Inductive `command : Type :=`
`| CheckAttributes : pubKey → Restricted → Sign → Decrypt → command`
`| TPM2_Hash : message → command`
`| CheckHash : message → message → command`
`| TPM2_Sign : message → privKey → command`
`| TPM2_Certify : pubKey → privKey → command`
`| CheckSig : message → pubKey → command`
`| MakeCSR_IDevID : identifier → signedCert → pubKey → command`
`| MakeCSR_LDevID : message → signedCert → command`
`| CheckCert : signedCert → pubKey → command`
`| MakePair : message → message → command`
`| CheckRandom : message → randType → command`
`| TPM2_ActivateCredential : message → privKey → privKey → command.`

Figure 4.3: Model of Commands

The `CheckAttributes` command verifies that a public key has all of the provided attributes. In practice, this is done by checking the `TPMA_Object` bits. In the model, these values are stored within the `Restricted`, `Sign`, and `Decrypt` fields of the `pubKey` type. In order to check the attributes of a particular key, one must have knowledge of that key. Specifically the public key to be checked must be in its state.

The TPM2_Hash command performs a cryptographic hash operation on a piece of data. This data may be any message that is known to the entity performing the command. The result of the operation is abstractly defined using the opaque hash constructor. In fact only one command may be used to determine the contents of a hash digest, that is, the CheckHash command which verifies that the contents of a hash digest match a particular plaintext message. The hash digest and the plaintext message must both be in state.

The TPM2_Sign command generates a signature over a message using the specified private key. There are several conditions for successful execution. For one, the key must have the Sign attribute set. Additionally, the key must reside in the TPM. If the key has the Restricted attribute not set, then the message must simply be in state. On the other hand, if the key does have the Restricted attribute set, then the message must have been produced by the TPM and be in the TPM state. As described in Section 2.1.1, a restricted signing key may sign external data if and only if the TPM first performed a hash operation on the data. We call the hash operation a signature hash when used for this purpose. In practice, the TPM2_Hash command would produce a ticket containing a validation structure which indicates that the resulting hash was produced by the TPM and is safe to sign. This ticket is then passed to the TPM2_Sign command. In the model, these tickets are handled implicitly: the hash digest produced by TPM2_Hash is added to the TPM state so that the restricting sign key may then sign the external data.

The TPM2_Certify command proves that an object is loaded in the TPM by producing a signed TPM2B_Attest structure. The command requires two inputs: a public key to be certified and a private key to sign the attestation structure. The private key must have the Sign attribute set and must reside in the TPM. Upon receiving a request to execute the TPM2_Certify command, the TPM will verify that the inverse of the public key parameter resides in the TPM as well. Messages produced by the TPM2_Sign and TPM2_Certify commands are defined using the signature constructor. A signature may be verified against a public key using the CheckSig command. If the provided public key is the inverse of the private key which performed the signature, then the check succeeds.

The `MakeCSR_IDevID` command produces a `TCG_CSR_IDevID` structure containing the provided inputs. A `TCG_CSR_IDevID` is a certificate signing request (CSR) which contains the data required to couple an IAK to a TPM-containing device. Additionally, it may include the certification information for an IDevID if one wishes to produce both the IAK and IDevID certificates in a single pass. In particular, this structure is used any time an enrollment process uses the EK certificate. The Trusted Computing Group (TCG) defines a C typedef structure to group all of the fields. In the model, we only include the fields necessary for creating an IAK certificate (i.e., device-identifying information, the EK certificate, and a public key to be certified). The `MakeCSR_LDevID` command is very similar to the `MakeCSR_IDevID` command except it produces a `TCG_CSR_LDevID` structure which includes the certification information for an LAK or LDevID. In the model, we only include the fields necessary for creating an LAK certificate (i.e., a signed `TPM2B_Attest` structure and the IAK certificate).

The `CheckCert` command verifies a signature over a certificate against a public key. One should check an EK certificate against the public key of the TPM Manufacturer's CA, an IAK or IDevID certificate against the public key of the OEM's CA, and an LAK or LDevID certificate against the public key of the Owner/Administrator's CA.

The `MakePair` command combines two messages into a single message using the `pair` constructor.

Due to the abstract, symbolic nature of this model, several TPM commands are intentionally excluded from the `command` type. This results specifically from an inability to truly capture the cryptographic properties of randomness. Randomness plays a vital role in the real-life implementation of keys and nonces, that is, it prevents a key or nonce from being guessed. Since we are unable to preserve this property in our model, we choose to eliminate all commands which generate a key or nonce. Therefore, a message of either of these types must be inferred from some other message or be in the initial state.

The `TPM2_GetRandom` command retrieves random bytes from the TPM. These resulting bytes may be used as a nonce. Due to the reason described above, we do not include this command

explicitly in our model, although we do assume it is used by the CA in some protocols.

The TPM2_MakeCredential command

The TPM2_ActivateCredential command

Commands in the model are sequenced linearly by the `sequence` type. In fact, this type is identical in structure to the Coq type `list command`. Sequential command execution is defined as an inductive proposition relating an initial state, a command sequence, and a final state. We

```

Inductive sequence : Type :=
| Sequence : command → sequence → sequence
| Done : sequence.

Infix ";;" := Sequence (at level 60, right associativity).

Inductive seq_execute : tpm_state * state → sequence → tpm_state * state → Prop :=
| SE_Seq : ∀ ini mid fin c s,
  execute ini c mid →
  seq_execute mid s fin →
  seq_execute ini (Sequence c s) fin
| SE_Done : ∀ ini,
  seq_execute ini Done ini.

```

Figure 4.4: Model of Command Sequences

can prove several interesting and useful facts about the `seq_execute` relation. Firstly, sequential execution is deterministic. Given an initial state and a command sequence, there is at most one final state which satisfies the `seq_execute` relation. This means that `seq_execute` is a partial function. Next, sequential execution is an expansion. Given a related initial state, command sequence, and final state, the initial state is always a subset of the final state (i.e., `seq_execute` expands on the initial state). In particular, commands do not remove elements from the state. Finally, sequential execution cannot generate keys or nonces. This feature is due to the deliberate exclusion of certain TPM commands from the language. The proof statements are recorded below (See Appendix B for the proofs themselves).

Theorem `seq_exec_deterministic` : $\forall \text{ini } s \text{ fin1 fin2},$
 $\text{seq_execute } \text{ini } s \text{ fin1} \rightarrow$
 $\text{seq_execute } \text{ini } s \text{ fin2} \rightarrow$
 $\text{fin1} = \text{fin2}.$

Theorem `seq_exec_expansion` : $\forall \text{iniTPM } \text{ini } s \text{ finTPM } \text{fin},$
 $\text{seq_execute } (\text{iniTPM}, \text{ini}) s (\text{finTPM}, \text{fin}) \rightarrow$
 $(\text{iniTPM} \subseteq \text{finTPM}) \wedge (\text{ini} \subseteq \text{fin}).$

Figure 4.5: Theorems on Sequential Execution

Chapter 5

Identity Provisioning

To maintain a cryptographic evidentiary chain linking a DevID to a specific TPM and device, the CA should follow certain provisioning protocols. The TCG describes several such protocols in their specification *TPM 2.0 Keys for Device Identity and Attestation*. We will consider in detail two of these protocols: OEM creation of an IAK certificate based on an EK certificate and Owner/Administrator creation of an LAK certificate based on an IAK certificate. We choose these two protocols since they bear the most significance in enrollment of additional DevIDs (recall that AK certificates may be used as parent nodes in a chain of certificates). For each protocol, the TCG's specification not only outlines its steps but also claims it provides certain assurances.

For convenience and clarity, we will inspect each of these protocols in the reverse order that their dependencies entails.

5.1 Owner/Administrator Creation of LAK Certificate based on IAK Certificate

In this section, we will shorten the term Owner/Administrator to Owner. The Administrator may still be a participant in this protocol. We only mean for this to act as an abbreviation. Now, we begin by provided a description of the recommended procedure.

0. The Owner creates and loads the LAK
1. The Owner certifies the LAK with the IAK
2. The Owner builds the CSR containing:
 - (a) The signed TPM2B_Attest structure

- (b) The IAK certificate
- 3. The Owner takes a signature hash of the CSR
- 4. The Owner signs the resulting hash digest with the LAK
- 5. The Owner sends the CSR paired with the signed hash to the CA
- 6. The CA verifies the recieved data by checking:
 - (a) The hash digest against the CSR
 - (b) The signature on the hash digest with the LAK public key
 - (c) The signature on the TPM2B_Attest structure with the IAK public key
 - (d) The signature on the IAK certificate with the public key of the OEM's CA
 - (e) The attributes of the LAK
- 7. If all of the checks succeed, the CA issues the LAK certificate to the Owner

The TCG's specification claims that this procedure provides the following assurances: (A) The new LAK is resident in the same TPM as the IAK and (B) The LAK has the correct key properties. Our goal is to prove that these assurances do in fact hold. Going forward, we will consider two cases: (1) the Owner and the CA are both trusted to execute their steps correctly and (2) only the CA is trusted to execute its steps correctly — the Owner is NOT trusted. The TCG's specification does not state which of these assumptions they are reasoning under. Additionally, it does not provide a proof or even any clear justification for how the protocol provides these assurances.

We model this protocol within Coq's `Module Type` mechanism. This mechanisms allows for the inclusion of parameters which provides the necessary flexibility to describe the protocol generally. Each participating entity has its own parameters. The Owner has a key to be certified (i.e., the LAK), its IAK, and IAK certificate. The CA has its own key and the public key of the OEM's CA. The `Module Type` mechanism additionally allows for axioms to be defined. When instantiating a `Module Type` with concrete values, one must prove all of the axioms. The first axiom we define is straightforward and only attempts to enforce the randomness of cryptographic keys, that is, all key parameters must be pairwise distinct.

First let us assume that statement 1 holds: the Owner and the CA are both trusted to execute


```

(* Owner parameters *)
Parameter pubLAK : pubKey.
Parameter pubIAK : pubKey.
Parameter certIAK : signedCert.

(* CA parameters *)
Parameter pubCA : pubKey.
Parameter pubOEM : pubKey.

(* All keys are pairwise distinct *)
Axiom keys_distinct :
  pubLAK <> pubIAK ∧
  pubLAK <> pubCA ∧
  pubLAK <> pubOEM ∧
  pubIAK <> pubCA ∧
  pubIAK <> pubOEM ∧
  pubCA <> pubOEM.

```

Figure 5.1: Parameters of LAK Certification Protocol

their steps correctly. The protocol can then be regarded as being composed of two parts: the Owner's steps (i.e., steps 0-5) followed by the CA's steps (i.e., steps 6-7). With that in mind, we can construct an object of type sequence for each the Owner and the CA. Since the command type does not include a method for creating and loading keys, we assume step 0 to have been performed prior to the sequence. A minimal initial state for the Owner can be constructed intuitively based on this sequence. In particular, given a sequence, a minimal initial state is defined as the smallest possible TPM state and general state which allows for successful execution of the sequence. Note that this definition does not constrain the general state to be a superset of the TPM state; in fact, it is the case that most often it is not. We provide a candidate initial state which satisfies this minimality constraint for the Owner's steps. The intuition behind this candidate state proceeds as follows. First, the private LAK and private IAK must reside in the same TPM because the Owner certifies the LAK with the IAK. Next, the IAK certificate must be in state because the Owner includes it in the CSR. The intuition behind building the minimal initial state is useful in proving that it is in fact minimal. The proof statement is constructed by two parts. First, the minimal initial state is a lower bound on the set of possible initial states (i.e., the minimal initial state is a subset

```

Definition steps1to5_Owner : sequence :=
  TPM2_Certify
    pubLAK
    privIAK ;;
  MakeCSR_LDevID
    (signature (TPM2B_Attest pubLAK) privIAK)
    certIAK ;;
  TPM2_Hash
    (TCG_CSR_LDevID (signature (TPM2B_Attest pubLAK) privIAK) certIAK) ;;
  TPM2_Sign
    (hash (TCG_CSR_LDevID (signature (TPM2B_Attest pubLAK) privIAK) certIAK))
    privLAK ;;
  MakePair
    (TCG_CSR_LDevID (signature (TPM2B_Attest pubLAK) privIAK) certIAK)
    (signature (hash (TCG_CSR_LDevID (signature (TPM2B_Attest pubLAK) privIAK) certIAK)) privLAK) ;;
  Done.

```

Figure 5.2: Model of Correct Steps of Owner

of every initial state which allows for successful execution of the steps). And second, the minimal initial state is sufficient for successful execution (i.e., the minimal initial state itself allows for successful execution of the steps). The lower bound property is proven to hold in general (see

```

Definition min_iniTPM_Owner : tpm_state :=
  [ privateKey privLAK ;
    privateKey privIAK ].

```

```

Definition min_ini_Owner : state :=
  [ signedCertificate certIAK ].

```

```

Lemma ini_Owner_lowerBound :  $\forall$  iniTPM ini fin,
  seq_execute (iniTPM, ini) steps1to5_Owner fin  $\rightarrow$ 
  (min_iniTPM_Owner  $\subseteq$  iniTPM)  $\wedge$ 
  (min_ini_Owner  $\subseteq$  ini).

```

```

Axiom ini_Owner_sufficient :  $\exists$  fin,
  seq_execute (min_iniTPM_Owner, min_ini_Owner) steps1to5_Owner fin.

```

Figure 5.3: Minimal Initial State of Owner

Appendix for proof). On the other hand, the sufficiency property must be proven for each specific instantiation of the Module Type since it is reliant on the parameters.

We provide a proof that this intuition led us to a correct minimum

5.2 OEM Creation of IAK Certificate based on EK Certificate

0. The OEM creates and loads the IAK
1. The OEM builds the CSR containing:
 - (a) Device identity information including the device model and serial number
 - (b) The EK certificate
 - (c) The IAK public area
2. The OEM takes a signature hash of the CSR
3. The OEM signs the resulting hash digest with the IAK
4. The OEM sends the CSR paired with the signed hash to the CA
5. The CA verifies the recieved data by checking:
 - (a) The hash digest against the CSR
 - (b) The signature on the hash digest with the IAK public key
 - (c) The signature on the EK certificate with the public key of the TPM Manufacturer's CA
 - (d) The attributes of the IAK
6. If all of the checks succeed, the CA issues a challenge blob to the OEM by:
 - (a) Calculating the cryptographic name of the IAK
 - (b) Generating a nonce
 - (c) Building the encrypted credential structure using the name of the IAK, the nonce, and the EK public key
7. The OEM releases the secret nonce by verifying the name of the IAK and decrypting the challenge blob
8. The CA checks the returned nonce against the one generated in step 6b
9. If the check succeeds, the CA issues the IAK certificate to the OEM

We model this protocol within Coq's Module Type mechanism. This mechanisms allows for the inclusion of parameters which provides the necessary flexibility to describe the protocol in the general case. Each participating entity has its own parameters. The OEM has a key to be certified

(i.e., the IAK), its EK certificate, and information to identify its device. The CA has its own key, the public key of the TPM Manufacturer's CA, and a nonce. The nonce parameter may be removed if we include the `TPM2_GetRandom` command in the model's language. The `Module Type` mechanism additionally allows for axioms to be defined. When instantiated a `Module Type` with concrete values, one must prove all of the axioms. The first axiom we define is straightforward and only attempts to enforce the randomness of cryptographic keys, that is, all key parameters must be pairwise distinct.

Chapter 6

Conclusion

6.1 Conclusion

6.2 Future Work

Include attestation variety of protocols which uses PCRs.

Include more TPM and TSS commands in the model to create a library. Find a way to add TPM2_Create and TPM2_GetRandom.

Include command branching, looping, etc

References

- Arthur, W., Challener, D., & Goldman, K. (2015). *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress.
- Institute of Electrical and Electronics Engineers (2018). IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity. *IEEE Std 802.1AR-2018*.
- Pierce, B. C., de Amorim, A. A., Casinghino, C., Gaboardi, M., Greenberg, M., Hrițcu, C., Sjöberg, V., & Yorgey, B. (2022). *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook.
- Trusted Computing Group (2019). *Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.59*.
- Trusted Computing Group (2021a). *TCG EK Credential Profile For TPM Family 2.0, Level 00, Version 2.4, Revision 3*.
- Trusted Computing Group (2021b). *TPM 2.0 Keys for Device Identity and Attestation*.

Appendix A

Model of Inference

```
Lemma inferFrom_iff_inferrable :  $\forall$  m st,  
  inferFrom m = st  $\leftrightarrow$  inferrable m st.
```

Proof.

```
  intros m st; split; intros H.  
  - generalize dependent m; assert (HI :  $\forall$  m, inferrable m (inferFrom m)); intros m.  
  -- induction m; simpl; try destruct c; try destruct s;  
    repeat constructor; assumption.  
  -- intros H; induction m; subst; apply HI.  
  - induction H; simpl; subst; try destruct cert; reflexivity.
```

Qed.

```
Fixpoint inferFrom (msg : message) : state :=  
  match msg with  
  | signature m k  $\Rightarrow$   
    (signature m k :: inferFrom m)  
  | TPM2B_Attest k  $\Rightarrow$   
    [TPM2B_Attest k ; publicKey k]  
  | TCG_CSR_IDevID id' (Cert k id k_ca) k'  $\Rightarrow$   
    [TCG_CSR_IDevID id' (Cert k id k_ca) k' ; publicKey k' ;  
     signedCertificate (Cert k id k_ca) ; publicKey k]  
  | TCG_CSR_LDevID m (Cert k id k_ca)  $\Rightarrow$   
    (TCG_CSR_LDevID m (Cert k id k_ca) :: inferFrom m ++  
     [signedCertificate (Cert k id k_ca) ; publicKey k])  
  | signedCertificate (Cert k id k_ca)  $\Rightarrow$   
    [signedCertificate (Cert k id k_ca) ; publicKey k]  
  | pair m1 m2  $\Rightarrow$   
    (pair m1 m2 :: inferFrom m1 ++ inferFrom m2)  
  | _  $\Rightarrow$   
    [msg]  
  end.
```



```

Inductive inferrable : message → state → Prop :=
| I_publicKey : ∀ k,
  inferrable (publicKey k)
  [publicKey k]
| I_privateKey : ∀ k,
  inferrable (privateKey k)
  [privateKey k]
| I_hash : ∀ m,
  inferrable (hash m)
  [hash m]
| I_signature : ∀ m k st,
  inferrable m st →
  inferrable (signature m k)
  (signature m k :: st)
| I_Attest : ∀ k,
  inferrable (TPM2B_Attest k)
  [TPM2B_Attest k ; publicKey k]
| I_encryptedCredential : ∀ n g k,
  inferrable (encryptedCredential n g k)
  [encryptedCredential n g k]
| I_randomNum : ∀ g,
  inferrable (randomNum g)
  [randomNum g]
| I_CSR_IDevID : ∀ id' cert k' st,
  inferrable (signedCertificate cert) st →
  inferrable (TCG_CSR_IDevID id' cert k')
  (TCG_CSR_IDevID id' cert k' :: publicKey k' :: st)
| I_CSR_LDevID : ∀ m cert st1 st2,
  inferrable m st1 →
  inferrable (signedCertificate cert) st2 →
  inferrable (TCG_CSR_LDevID m cert)
  (TCG_CSR_LDevID m cert :: st1 ++ st2)
| I_signedCertificate : ∀ k id k_ca,
  inferrable (signedCertificate (Cert k id k_ca))
  [signedCertificate (Cert k id k_ca) ; publicKey k]
| I_pair : ∀ m1 m2 st1 st2,
  inferrable m1 st1 →
  inferrable m2 st2 →
  inferrable (pair m1 m2)
  (pair m1 m2 :: st1 ++ st2).

```

Appendix B

Misc Stuff 2