Formal Analysis of TPM Key Certification Protocols

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: 03 May 2023

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

Development and deployment of trusted systems often require definitive identification of devices. A remote entity should have confidence that a device is as it claims to be. An ideal method for fulfilling this need is through the utilization of TPM keys as secure device identitifiers. A secure device identifier (DevID) is defined as an identifier that is cryptographically bound to a device. The TCG's specification "TPM 2.0 Keys for Device Identity and Attestation" describes several methods for remotely proving a key to be resident in a specific device's TPM. These methods are carefully constructed protocols which are intended to be performed by a trusted Certificate Authority (CA) in communication with a certificate-requesting device. DevID certificates provisioned by an OEM at device manufacturing time may be used to provide definitive evidence to a remote entity that a key belongs to a specific device. Whereas DevID certificates provisioned by a device owner require a chain of certificates in order to verify a chain of trust to an OEM-provided root certificate. This distinction is due to the differences in the respective protocols prescribed by the TCG's specification. *I aim to abstractly model these DevID-provisioning protocols and formally verify their resulting assurances*. I choose this goal since the TCG themselves do not provide any proofs or clear justifications for how the protocols might provide these assurances. The resulting command library and execution relation modeled in Coq may easily be expanded upon to become useful in verifying a wide range of properties regarding DevIDs and TPMs.

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Development and deployment of trusted systems often require definitive identification of devices. A remote entity should have confidence that a device is as it claims to be. An ideal method for fulfulling this need is through the utilization of TPM keys as secure device identitifiers. A secure device identifier (DevID) is defined as an identifier that is cryptographically bound to a device [7]. A DevID must not be transferable from one device to another as that would allow distinct devices to be identified as the same. Since the Trusted Platform Module (TPM) is a secure Root of Trust for Storage, it provides the necessary protections for storing these identifiers and enforcing this constraint.

The TCG's specification "TPM 2.0 Keys for Device Identity and Attestation" describes several methods for remotely proving a key to be resident in a specific device's TPM. These methods are carefully constructed protocols which are intended to be performed by a trusted Certificate Authority (CA) in communication with a certificate-requesting device. These protocols are designed to maintain a cryptographic evidentiary chain linking a DevID to a specific TPM [13]. DevID certificates provisioned by an OEM at device manufacturing time provide definitive evidence that a key belongs to a specific device. Whereas DevID certificates provisioned by a device owner require a chain of certificates to prove that a key belongs to a specific device. This distinction is due to differences in the respective protocols prescribed by the TCG's specification. For each provisioning protocol described in the specification, the TCG outlines steps for the CA and the certificate-requesting entity to perform. Furthermore, the TCG claims that each protocol provides certain assurances. These assurances are the basis for the resulting cryptographic evidentiary chain. Each assurance manifests as an assertion regarding either TPM-residency, key attributes, or

previously-issued certificates.

This work places special emphasis on two of the DevID-provisioning protocols, namely OEM Creation of an IAK Certificate based on an EK Certificate and Owner Creation of an LAK Certificate based on an IAK Certificate. I select these protocols due to their especially significant security implications. The primary goal of this work is to abstractly model these two protocols and formally verify their resulting assurances. I choose this goal since the TCG themselves do not provide any proofs or clear justifications for how the protocols might provide these assurances.

1.1 Related Works

My work has many parallels to the work of Halling et al. [5, 6]. They abstractly model a large subset of the TPM 1.2 commands in the PVS specification language. TPM command execution is modeled as a transition system over an abstract system state. I implement this same technique in my own work but over a small subset of TPM 2.0 commands as well as several non-TPM commands. Their work attempts verification of the Privacy CA protocol. In their verification, they consider the functional correctness of the protocol and specifically examine the protocol implementation to ensure that it produces the structure that it should. They do not consider specific attacks on this structure. Similarly, my work attempts verification of protocols involving a certificate authority and focuses on determining whether the protocol implementations result in the assurances that they should provide.

The work of Delaune et al. uses the ProVerif tool to examine Authentication and PCRs in the TPM 1.2 [3, 4]. Their work focuses on verifying cryptographic properties over these elements of the TPM.

3,4 Related Works [14] Tamarin prover

[10] stateful applied pi calculus SAPIC tool and Tamarin prover

[9]

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. 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 [1].

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. Enhanced authorization (EA) allows a key or other TPM object 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. Platform configuration registers (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, that is, the key must be a secure device identifier. There exist a variety of additional applications, such as network authentication, which similarly require a device to be definitively identified. DevID certificates may be distributed to remote entities in order to prove that the corresponding private key resides in a specific TPM-containing device. These certificates are stored within the TPM's NVRAM providing protection from accidental erasure.

2.1.1 Keys

Since the focus of this paper is on the usage of TPM keys for secure device identifiers, the discussion on cryptographic keys in this section is limited to TPM keys which utilize asymmetric cryptography. A key may then 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`, and `Restricted`. The `FixedTPM` attribute indicates that the private key cannot be duplicated. 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

4

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 [2]. Furthermore, 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 : keyIdType → Restricted → Sign → Decrypt → FixedTPM → pubKey.

Inductive privKey : Type :=
| Private : keyIdType → Restricted → Sign → Decrypt → FixedTPM → privKey.
```

Figure 2.1: Model of Keys

The `Restricted` attribute provides 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` [11]. All structures that the TPM constructs from internal data begins with this value. Such structures include keys and PCRs. These structures result in several interesting and significant uses for restricted signing keys, namely key certification and attestation. In particular, a restricted signing key is used during key certification to prove that a new key resides in the same TPM as itself. And during attestation, a restricted signing key is used to prove that a quote is the result of the PCR values within the same TPM as itself. Additionally, a restricted signing key has the ability to sign data supplied to the TPM externally by using the `TPM2_Hash` command. When used for this purpose, the hash operation is called a signature hash. 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 only produces a

5

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 [1]. 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.

2.2 Logical Foundation of Coq

The Calculus of Inductive Constructions (CIC) is the underlying formalism of the interactive proof assistant Coq [8]. This section shall not discuss this logical system in great detail and instead focuses only on the aspects which are relevant to this work. In particular, this section examines the sort `Prop` and the interesting ways in which it is utilized. The sort `Prop` is the universe of logical propositions. Inductive definitions may be used together with the sort `Prop` to define relations. These relations are called inductive propositions. Some inductive propositions are defined in Coq's standard library while others must be defined by the user. One simple yet important inductive proposition in Coq's standard library is the definition of equality. Equality has only one constructor,

```
Inductive eq {A : Type} (x : A) : A → Prop :=
| eq_refl : eq x x.
```

Figure 2.2: Equality in Coq

namely `eq_refl` which corresponds with reflexivity. The constructors of an inductive proposition act as introduction rules; for an element to satisfy a particular inductive proposition, it must be built from its constructors. Using `eq`'s single constructor, one can prove nearly all of the fundamental properties of equality (e.g., symmetry and transitivity), though there is one important property of equality which cannot be proven to hold in general, that is, decidability. A type has decidable equality if any two elements of that type are either equal or not equal. Due to the intuitionistic

nature of CIC, not all types in Coq have this property. Intuitionistic logic differs from classical logic in that it rejects the law of excluded middle and double negation elimination. Therefore, decidability is not guaranteed over any proposition: equality included. If decidable equality over a specific type is needed, one must declare that type an instance of the DecEq class and explicitly prove that this property holds.

```
Class DecEq (A : Type) :=
{
  decEq : ∀ x1 x2 : A, {x1 = x2} + {x1 ≠ x2}
}.
```

Figure 2.3: Decidable Equality

Chapter 3

Secure Device Identity

A secure device identifier (DevID) is defined as an identifier that is cryptographically bound to a device [7]. A DevID must not be transferable from one device to another and must be stored in a way that protects it from modification. Since the TPM is a secure Root of Trust for Storage and protects keys against compromise, TPM keys are an ideal choice for DevIDs. The binding of a TPM key to a specific device instance is represented by a device identity certificate. To know when and by whom DevIDs are typically provisioned, it is useful to briefly consider a simplified version of the process for creating and distributing TPM-containing devices to end users.

1. TPM Manufacturers produce TPM chips according to the international standard. They provision each TPM chip with an EK certificate which binds the EK 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 DevID certificates which bind a key to a specific device. These devices are then distributed to the end users (Owners).

3. Owners may optionally provision their TPM chip(s) with one or more DevID certificates which bind a key to their specific device.

The provisioning of device identification in Steps 2 and 3 is the primary subject of this work. It is important to note that an EK is not a DevID because its associated certificate identifies a TPM not device. 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 `Sign` attribute must be

8

set and the `Decrypt` attribute not set. This is in fact another reason that an EK cannot be a DevID
— recall from Section 2.1.1 that EKs are storage keys and hence have the `Decrypt` attribute set.
Furthermore, a DevID must have the `FixedTPM` attribute set because it is of paramount importance
that it never be duplicated, transferred, or copied. On the other hand, the `Restricted` attribute is
optional in a DevID. 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 prove that some object or data
is contained within the same TPM as itself. The acronyms DevID and AK are prefixed by the letter
I or L denoting Initial or Locally Significant respectively. Initial device identifiers are installed by
OEMs in TPM-containing devices at manufacturing time; IDevIDs and IAKs correspond with the
keys referenced in Step 2. Initial device identifiers are intended to be long-lived and usable for the
lifetime of the device. Since an Owner cannot provision new Initial device identifiers, these keys
should be recoverable to avoid the problematic loss of these essential identifiers. Primary keys are
able to be recreated during the lifetime of the TPM or more precisely during the lifetime of the
TPM's Primary Seed. Therefore IDevIDs and IAKs should be Primary keys. Locally Significant
device identifiers are installed by Owners; LDevIDs and LAKs correspond with the keys referenced
in Step 3. Locally Significant device identifiers are not expected to be long-lived because an Owner
may install new ones at any time.

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
LDevID	Ordinary	X	X			Owner

Table 3.1: Key Requirements and Recommendations

An IAK should be used only for the certification of new DevIDs and even then should only be
used sparingly to limit the chances of compromise. Similarly, an IDevID should be used sparingly
but instead for device authentication in an enterprise network. Since an Owner may create LAKs
and LDevIDs as often as needed, these DevIDs may be used freely. Although, it is still recom-

9

mended that any given DevID—LAK and LDevID included—be used in only a single application. The reason for this recommendation is that it might be possible to gather a lot of data about a device and/or user when the same identity is used for multiple applications [13]. An LAK is intended to be used for the purposes of attestation as well as for the certification of new DevIDs. While an LDevID is intended to be used for any general device authentication purposes.

The issuers of 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 EKs is known as the TPM Manufacturer's CA, 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's CA). All CAs should support a standard certificate transport protocol that provides confidentiality, integrity, and protection from replay attacks [13]. These transport protocols are outside the scope of this paper, and so this work assumes CAs to be following this recommendation precisely. All CAs should have a Certificate Practice Statement (CPS) which states the CA's enrollment procedures. The TPM Manufacturer's CA and OEM's CA must both be particularly scrupulous due to the important security and identity implications provided by the certificates they issue. The trustworthiness of a certificate is directly reliant on the diligency and rigor of the CA which issued the certificate. A CA's CSP is vital for making an informed trust decision regarding a given certificate. This work chooses to trust that the TPM Manufacturer's CA implements appropriate practices which provide this necessary diligence and rigor. I choose not to investigate further into this matter primarily because the EK is not a DevID and thus not the focus of this work.

3.1 Certificate Chain

The term certificate specifically refers to an X.509 v3 digital certificate. A certificate contains a public key and identifying information and is signed by a Certificate Authority. This identifying information is contained in the Subject field of the certificate structure. In particular, a certificate binds a key to the identity represented by the Subject field using a signature. Each certificate also has field which contains its associated certificate serial number. This number is unique (per CA)

and may be used to definitively identify a certificate. Certificates contain a variety of other fields as well. In this model, certificates are abstractly defined as the inductive `signedCert` type. This type greatly simplifies the true implementation of a certificate and thus includes only a select few of the associated fields. A `signedCert` requires a public key, an identifier, and a private key. An identifier is an abstract representation of a certificate's Subject field and 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 identifier : Type :=
| TPM_info : tpmInfoType → identifier
| Device_info : deviceInfoType → identifier.

Inductive signedCert : Type :=
| Cert : pubKey → identifier → privKey → signedCert.
```

Figure 3.1: Model of Certificates

An EK certificate's Subject field in combination with other fields imparts various assertions regarding the security qualities and provenance of the TPM [12]. Although the EK is not a DevID, these assertions allow the EK to serve a significant role in the enrollment of DevIDs. Specifically, the EK is used in the creation of an IAK certificate, the first and arguably most important certificate which identifies a device. Proving that an IAK belongs to a specific device requires first binding the IAK to a specific TPM using the EK and then binding the TPM to a specific device. Due to this reliance, an IAK certificate even includes a field which references the associated EK certificate's serial number.

Trust for enrolling a new DevID certificate is based on an existing certificate. Therefore, a chain of certificates can be used to verify a chain of trust to some trust anchor [13]. Since IAK certificates provide definitive evidence that a key belongs to a specific device, an IAK certificate typically acts as this trust anchor and thus typically is the root node in a chain of certificates. In issuing an IAK certificate, the OEM's CA makes an assertion that is a primary security dependency for all future enrollment of DevIDs. The Subject field of DevID certificates contains non-TPM device

information such as device model and serial numbers. This information should be globally unique per device [7]. All certificates in a chain should have the same Subject field as the IAK certificate.

Since a key with the `Restricted` attribute set has the ability to prove that some unknown key is resident in the same TPM as itself, AK certificates are used as the requisite existing certificate in the creation of new DevIDs. Specifically this means that AK certificates may be parent nodes in a chain of certificates. Due to this result, a chain of certificates exhibits a tree structure. The underlying security implications provided by a chain of certificates is formed primarily by the protocols which provision these certificates. Specifically, the Proof of Residency and Chain of Trust lines in Figure 3.2 are a direct consequence of the assurances provided by the respective provisioning protocols.
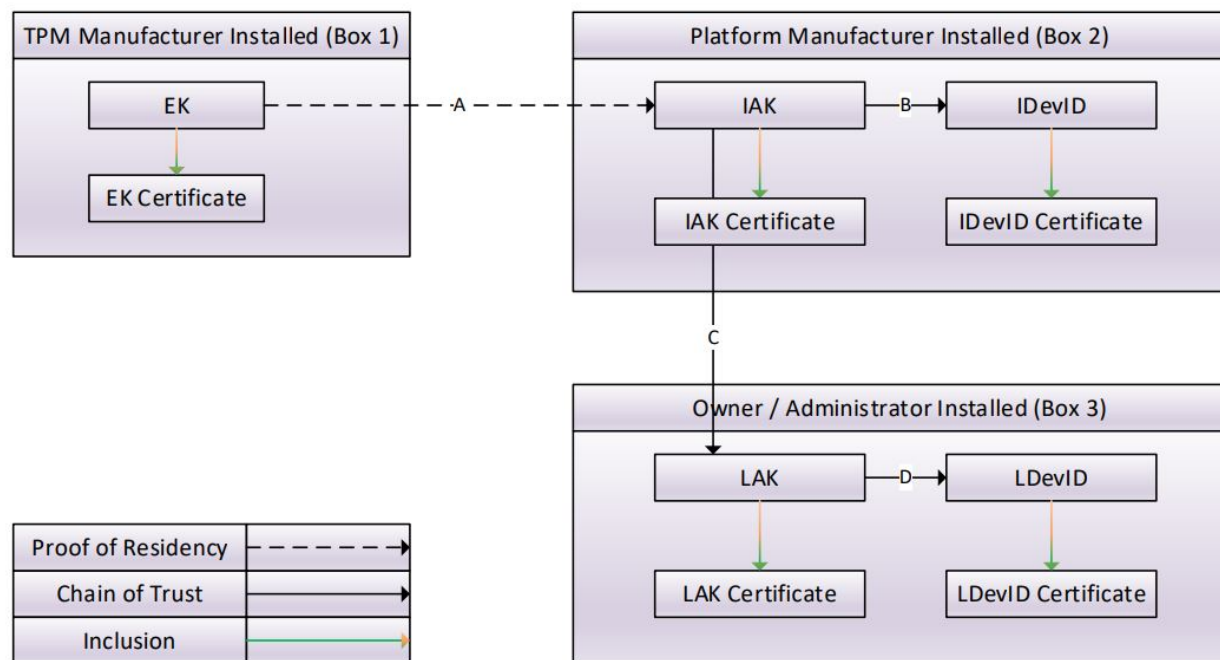

Figure 3.2: Key and Certificate Relationships [13]

- 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.

12

- 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'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'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's CA.

Although, Figure 3.2 shows the relationships between keys and certificates for exactly one of each type of DevID, in practice, there often exists multiple of each type of DevID. Due to algorithm agility, an OEM usually installs multiple IAKs and IDevIDs with varying algorithms and sizes. IAKs are always directly linked to an EK, while IDevIDs are always directly linked to an IAK. On the other hand, LAKs and LDevIDs may be directly linked to any attestation key (i.e., an IAK or LAK). This means that an LAK certificate may be used in the creation of LDevIDs or additional LAKs producing an arbitrarily long chain of certificates.

Chapter 4

Command 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 in the form `signature m k`, the message `m` may be deduced. Whereas given a message in the form `encryptedCredential n g k`, no messages may be deduced. This concept is modeled in two ways: as a recursive function `inferFrom` and as an inductive proposition `inferrable`. These two definitions are proven to be equivalent. 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. I do not attempt to model the computational intricacies of true cryptography. Command execution is defined as an inductive proposition relating an initial state pair, a command, and a final state pair. This resulting model is in fact a labeled transition system. A labeled transition system is defined as a triple (S, L, T) where S is a set of states, L is a set of labels, and $T \subseteq S \times L \times S$ is a labeled transition relation. In this case, S is the set of `tpm_state * state` pairs; L is the set of elements in the `command` type; and T is the `execute` relation. The `tpm_state` and `state` types are aliases for a list of messages. These types

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

Figure 4.2: Type Signature of Execute Relation

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). The `tpm_state` type is intended to contain messages that a restricted signing key may operate on. Recall from Section 2.1.1 that these messages may be in one of two forms: (1) objects constructed from TPM-internal data or (2) digests produced by signature hash operations. Messages in form 1 include private keys and PCRs (although the latter are not yet included in this model).

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 I am unable to preserve this property in the model, I choose to eliminate all commands which generate a key or nonce. Therefore, a message of either of these types must be generated from or inferred from some other message or be in the initial state. Each command included in this model corresponds with exactly one constructor in the `execute` relation (excluding `TPM2_Sign` which corresponds with two). Each constructor possesses its own distinctive conditions which must be met for execution to be successful. These conditions are manifested in several ways: constructors pattern match on

```
Inductive command : Type :=
| TPM2_Hash : message → command
| CheckHash : message → message → command
| TPM2_Sign : message → privKey → command
| TPM2_Certify : pubKey → privKey → command
| CheckSig : message → pubKey → command
| TPM2_MakeCredential : message → randType → pubKey → command
| TPM2_ActivateCredential : message → privKey → privKey → command
| MakeCSR_IDevID : identifier → signedCert → pubKey → command
| MakeCSR_LDevID : message → signedCert → command
| CheckCert : signedCert → pubKey → command
| CheckAttributes : pubKey → Restricted → Sign → Decrypt → FixedTPM → command
| MakePair : message → message → command.
```

Figure 4.3: Model of Commands

the command's inputs, some of these inputs are required to be in the initial state, and the results must be added to the final 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; this message must be contained in the initial `state`. The result of the operation is abstractly defined using the opaque `hash` constructor and is stored in both the final `tpm_state` and `state`. 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. Both the hash digest and the plaintext message must be contained in the initial `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 (i.e., be contained in the initial `tpm_state`). The conditions then differ based on the status of the private key's `Restricted` attribute; the `execute` relation includes one constructor for each a restricted and nonrestricted key. If the key does not have the `Restricted` attribute set, then the message must simply be in the initial `state`. On the other hand, if the key does have the `Restricted` attribute set, then the message must be in the initial `tpm_state`. As discussed above, these messages may be TPM-internal objects or hash

16

digests. 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 its final `tpm_state` so that a subsequent signing operation may have the hash digest in its initial `tpm_state`.

The `TPM2_Certify` command proves than 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 recieving a request to execute the `TPM2_Certify` command, the TPM verifies 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 `TPM2_MakeCredential` command is used when a remote entity, especially a CA, desires to affirm that some private key resides in the same TPM as a particular EK. This command requires three inputs: the cryptographic name of a key to be credentialed, a secret, and a public EK. The cryptographic name of a key is produced by hashing its public area with its associated hash algorithm and prepending the Algorithm ID of the hashing algorithms. This command produces an encrypted credential. This command guarantees that the secret is only released from the credential blob if the credentialed key is loaded on the same TPM as the EK. The `TPM2_ActivateCredential` command is used by the recipient of an encrypted credential to release the secret. When executing the `TPM2_ActivateCredential` command, the TPM first decrypts the blob with the EK then verifies that the private key corresponding with the name field is loaded on the TPM as well. The secret is only release if both of these steps succeed. The secret value may then be returned to the remote CA so that they may validate the result.

The `MakeCSR_IDevID` command produces a `TCG_CSR_IDevID` structure containing the pro-

vided 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, I only include the fields necessary for creating an IAK certificate (i.e., device-identifying information, an 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, I only include the fields necessary for creating an LAK certificate (i.e., a signed `TPM2B_Attest` structure and an 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's CA.

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 of the key. In the model, these values are stored within the `Restricted`, `Sign`, `Decrypt`, and `FixedTPM` fields of the `pubKey` type. In order to check the attributes of a particular key, one must have have knowledge of that key. The key need not reside within one's own TPM though since this command is typically used to check the attributes of some external entity's key. The model requires specifically that the initial `state` contains the public key to be checked. The `MakePair` command combines two messages into a single message using the `pair` constructor.

Commands in the model are sequenced linearly by the `sequence` type which is identical in structure to the Coq type `list`. Sequential command execution is defined as an inductive proposition relating an initial state pair, a command sequence, and a final state pair. Sequential execution is done by first executing the command at the head of the list using the single command execution

18

```
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

relation execute followed by executing the sequence at the tail of the list using the sequential

execution relation seq_execute. The final state pair produced by executing the single command

is used as the initial state pair for the subsequent sequence. This recursive structure allows for

the convenient use of induction in many proofs. In fact, I prove several interesting and useful

facts about the seq_execute relation. Firstly, sequential execution is deterministic. Given an

initial state pair and a command sequence, there is at most one final state pair which satisfies the

seq_execute relation. This means that seq_execute is a partial function. Next, sequential exe-

cution is an expansion. Given a related initial state pair, command sequence, and final state pair,

the initial state is always a subset of the final state (i.e., seq_execute expands on the initial state).

This means that commands do not remove elements from the state.

```
Theorem seq_exec_deterministic : ∀ ini s fin1 fin2,
  seq_execute ini s fin1 →
  seq_execute ini s fin2 →
  fin1 = fin2.


Theorem seq_exec_expansion : ∀ iniTPM ini s finTPM fin,
  seq_execute (iniTPM,ini) s (finTPM,fin) →
  (iniTPM ⊆ finTPM) ∧ (ini ⊆ fin).
```

Figure 4.5: Properties of Sequential Execution

It is useful to know whether a particular command is contained within a sequence. Therefore, I define a function `command_in_sequence` which determines whether a provided command is an element of a provided sequence. This function is implemented by checking the command at the head of the list against the provided command. If they match, return True. Otherwise, recursively call the function on the tail of the sequence. For two commands to match, not only must the commands itself match but all of their inputs as well. To precisely describe this situation, this function utilizes the decidable equality property over the `command` type. This requires declaring and proving that the `command` type and all of the types it relies on (e.g., message, pubKey, signedCert) a member of the `DecEq` class. These proofs may be conveniently automated.

Chapter 5

Identity Provisioning

To mantain 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". Trust for enrolling a new certificate is based on an existing certificate.

This chapter considers two of these protocols in detail: OEM creation of an IAK certificate based on an EK certificate and Owner creation of an LAK certificate based on an IAK certificate. I select these two protocols since they bear the most significance in enrollment of additional DevIDs (recall that IAK certificates may be root nodes in a chain of certificates and LAK certificates may be parent nodes). For each protocol, the specification outlines steps for the CA and the certificate-requesting entity to perform. Furthermore, the specification claims that each protocol provides certain assurances. Each assurance manifests as an assertion regarding either TPM-residency, key attributes, or previously-issued certificates which provide the basis for the resulting cryptographic evidentiary chain formed by a chain of certificates. Therefore, it is of utmost importance to verify that each protocol can guarantee its associated assurances. Since the TCG themselves do not present any proofs or clear justifications to support these claims, the goal of this work is to abstractly model these DevID-provisioning protocols and formally verify their resulting assurances.

I model these protocol within Coq's `Module Type` mechanism. This mechanisms allows for the inclusion of parameters which provides the necessary flexibility to describe each protocol generally. Additionally, this mechanism allows for axioms to be defined. When instantiating a Module Type, one must provide concrete values for all parameters and prove that all axioms hold.

In conducting these verifications, I consider two scenarios: (1) the certificate-requesting entity

and the CA are both trusted to execute their steps of the protocol correctly and (2) only the CA is trusted to execute its steps correctly. The specification does not state which of these assumptions they reason under. Both of these scenarios include the presupposition that previously issued certificates imply the associated assurances of their provisioning protocols . For convenience and clarity, this chapter inspects each of these protocols in the reverse order that their dependencies entails. I proceed with this ordering because the LAK certification protocol is approximately contained within the IAK's protocol.

5.1   Owner Creation of LAK Certificate based on IAK Certificate

The TCG's specification claims that the below procedure provides the following assurances: (A) the LAK has good attributes (B) the new LAK is resident in the same TPM as the IAK. These assurances correspond exactly with the Chain of Trust line C in Figure 3.2. The CA should use the device identifying information from the IAK certificate in the Subject field of the LAK certificate.

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 public LAK
    (c) The signature on the `TPM2B_Attest` structure with the public IAK
    (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

Modeling this protocol begins by defining parameters for each the Owner and CA. These parameters correspond with the elements required by the Owner and the CA to perform their respective parts of the protocol. However, elements intended to be received during the communication phases of the protocol are excluded. Specifically, these parameters intend to represent the elements which must be known by each entity prior to execution of the protocol. Therefore, the Owner has its LAK, IAK, and IAK certificate, and the CA has its own key and the public key of the OEM's CA. The parameters only explicitly include the public key values of those listed keys pairs. Private key values are computed by taking the inverse of the corresponding public key; these values are stored in the `privLAK`, `privIAK`, and `privCA` variables. To enforce the randomness of cryptographic keys, I define an axiom which requires all key parameters to be pairwise distinct.

```
(* 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

Similar to how the parameters are separated by ownership, the procedure itself may be separated as well. That is, the procedure may be regarded as the composition 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, each part

of the procedure may be abstractly modeled using the parameters defined above and the sequential command construction defined in Chapter 4. I construct an object of `sequence` type for the Owner and an object of function type for the CA. Constructing the Owner's steps is straightforward. Since this model disallows the arbitrary creation of keys, Step 0 is assumed to have been performed prior; the results of Step 0 are in fact already encapsulated in the `pubLAK` parameter and `privLAK` variable. Then each remaining Step of the Owner corresponds with exactly one command in the model, namely `TPM2_Certify` for Step 1, `MakeCSR_LDevID` for Step 2, `TPM2_Hash` for Step 3, `TPM2_Sign` for Step 4, and `MakePair` for Step 5.

Constructing the CA's steps is more complex as it relies on external input (i.e., the certification request produced by the Owner's steps). Although this complexity leads to a convoluted function, there is still a straightforward correspondence between the function definition and the real-life protocol. First, the CA waits to receive a certification request from the Owner (see the `msg` input). The request must be in a specific format to be considered valid (see the match statement on `msg`). The CA then executes Step 6 of the procedure (see the sequence within `seq_execute`). If execution succeeds, the CA issues the LAK certificate to the Owner (see the `Prop` return type). I include several additional parameters and criteria to serve as a method for referencing certain elements of the certification request within proof statements.

These definitions provide the framework necessary for describing the conditions of each scenario. In fact, using only the CA's function, it is trivial to prove Assurance A. The command `CheckAttributes k Restricting Signing NonDecrypting Fixing` in the CA's function corresponds with Step 6e of the certification procedure (see that the CA's function binds the LAK to the variable `k`). Then it is clear to see that succesful execution of this command directly implies that the LAK has all of the attributes required by the `attestationKey` function defined in Chapter 3.

With that complete, let us now attempt formal verification of Assurance B first under the conditions of scenario 1: the Owner and the CA are both trusted to execute their steps correctly. Recall that this verification trusts that the previously-performed IAK certification procedure guarantees

```
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.



Definition steps_CA (msg : message) (iak lak : pubKey) (cert : signedCert) : Prop :=
  match msg with
  | (pair (TCG_CSR_LDevID (signature (TPM2B_Attest k) k0') (Cert k0 id k_ca')) (signature m k')) ⇒
        iak = k0 ∧ lak = k ∧ cert = (Cert k0 id k_ca') ∧
        seq_execute (iniTPM_CA, inferFrom msg ++ ini_CA)
                    (CheckHash
                        m
                       (TCG_CSR_LDevID (signature (TPM2B_Attest k) k0') (Cert k0 id k_ca')) ;;
                     CheckSig
                       (signature m k')
                        k ;;
                     CheckSig
                       (signature (TPM2B_Attest k) k0')
                        k0 ;;
                     CheckCert
                       (Cert k0 id k_ca')
                        pubOEM ;;
                     CheckAttributes
                        k
                        Restricting Signing NonDecrypting Fixing ;;
                     Done)
                    (iniTPM_CA, inferFrom msg ++ ini_CA)
  | _ ⇒ False
  end.
```

Figure 5.2: Model of LAK Certification Protocol

its associated assurances. Specifically this verification uses the assertion that the IAK has good attributes.

We begin by examining the Owner's steps and its requirements. These requirements can be quantitatively described by a minimal initial state pair. Given a sequence, a minimal initial state pair is defined as the smallest `tpm_state` and `state` which allows for successful execution of the sequence. The proof statement describing this property is constructed by two parts. First, the minimal initial state is a lower bound on the set of possible initial states. And second, the minimal initial state is sufficient for successful execution. I build a minimal initial state pair for `steps1to5_Owner` using the following intuition: (i) the private LAK and private IAK reside in the same TPM because the LAK is certified by the IAK and (ii) the IAK certificate is known to the Owner because it is included in the CSR. This intuition is used to guide the proof of the lower bound property. On the other hand, proving the sufficiency property uses several preconditions, namely that the CA decides to issue the LAK certificate and that the IAK has good attributes. The first precondition is clearly safe to assume as it is a direct consequence of scenario 1. And the second precondition is also safe to assume since the OEM's CA is trusted to have checked the attributes on the IAK when issuing its respective certificate. This analysis of the Owner's steps'

```
Definition iniTPM_Owner : tpm_state :=
[ privateKey privLAK ;
  privateKey privIAK ].

Definition ini_Owner : state :=
[ signedCertificate certIAK ].

Lemma ini_Owner_lowerBound : ∀ iniTPM ini fin,
  seq_execute (iniTPM, ini) steps1to5_Owner fin →
  ( iniTPM_Owner ⊆ iniTPM) ∧
  ( ini_Owner ⊆ ini).

Lemma ini_Owner_sufficient : ∀ msg,
  attestationKey pubIAK →
  steps_CA msg pubIAK pubLAK certIAK →
  ∃ fin, seq_execute (iniTPM_Owner, ini_Owner) steps1to5_Owner fin.
```

Figure 5.3: Minimal Initial State of Owner

requirements in the form of a minimal initial state conveniently leads to the conclusion that the new LAK and the IAK must be resident in the same TPM, namely the TPM on the Owner's device. This conclusion is manifested in the `iniTPM_Owner` variable which contains both the private LAK and private IAK. In conclusion, we have now confirmed that Assurance B is in fact guaranteed by the protocol when we assume that both the Owner and the CA are trusted to execute their steps correctly.

Therefore, let us now attempt formal verification of this same goal under the conditions of scenario 2: only the CA is trusted to execute its steps correctly. This proof is troublesome and likely impossible if we make no assumptions regarding the Owner, but since the certification request and its contents must have been produced by some entity, I consider the Owner to be this entity. To this end, I describe the Owner and its characteristics as a series of assumptions: the Owner executes some unknown sequence of commands `s`, this sequence produces some message `msg` in the Owner's final `state`, the Owner's initial `tpm_state` may only contain private keys, and the Owner's initial `state` may only contain public keys and certificates. I argue that these assumptions

```
∀ s iniTPM ini finTPM fin msg iak lak cert,
seq_execute (iniTPM, ini) s (finTPM, fin) →
In msg fin →
(∀ m', needsGeneratedTPM m' → ¬In m' iniTPM) →
(∀ m', needsGenerated m' → ¬In m' ini) →
```

Figure 5.4: Assumptions on the Untrusted Owner

are reasonable and do not corrupt the conditions regarding the trustworthiness of the Owner. In particular, these assumptions aim only to constrain the production of the CSR and its contents to the Owner. The restrictions on the Owner's initial state pair are the main contributors to enforcement of this constraint. The `needsGeneratedTPM` function restricts the Owner's initial `tpm_state` to the inclusion of previously created private keys. While the `needsGenerated` function restricts the Owner's initial `state` to the inclusion of public keys as well as previously issued certificates — the subject of these certificates may be the Owner itself or any other entity. Although realistically the Owner has many other messages in its knowledge, these restrictions simply aim to disallow

27

them from being used to build the CSR.

Our next step is to use this series of initial assumptions to glean further information on the Owner. We cannot directly obtain the conclusion that the new LAK is resident on the same TPM as the IAK, but we are able to make an important conclusion regarding the sequence which the Owner executes. That is, the sequence `s` is a supersequence of the correct steps of the Owner (i.e., `steps1to5_Owner`). A list is a supersequence of another list if all the elements of the second list occur, in order, in the first — the elements need not occur consecutively. I define a cascading collection of recursive functions to determine whether a given sequence of commands is a supersequence of `steps1to5_Owner`. Then using the initial assumption regarding the Owner and the CA (i.e., those in Figure 5.4 plus `steps_CA msg iak lak cert`), I prove that the Owner's unknown sequence `s` satisfies this function.

Our overall proof hinges on this conclusion. In fact, the proof proceeds fairly naturally from this point on. Recall our musings in scenario 1 which reason that the LAK and IAK must reside in the same TPM if one certifies the LAK with the IAK. Therefore, our next step is to demonstrate that the Owner must in fact have executed `TPM2_Certify` on the public LAK and private IAK. Using the conclusion obtained above it is trivial to prove that this command is contained within the Owner's sequence `s`. I use the function `command_in_sequence` to accurately describe this situation because all of the command inputs must match exactly. Then finally I prove one last set of intermediate lemmas which authoritatively state that the LAK and IAK are resident in the same TPM whenever one executes any sequence which contains that command. Now it is apparent that the composition of these small proofs leads to our end goal. In conclusion, we have now confirmed that Assurance B is in fact guaranteed by the protocol when we assume that only the CA is trusted to execute its steps correctly.

5.2 OEM Creation of IAK Certificate based on EK Certificate

The TCG's specification claims that the below procedure provides the following assurances: (A) the IAK has good attributes, (B) the new IAK is resident in the same TPM as the EK, and (C) the EK certificate is valid. These assurances correspond exactly with the Proof of Residency line A in Figure 3.2. The CA should use the device identify information from the CSR in the Subject field of the IAK 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

This procedure is very similar to the one described in the previous section. In fact, nearly all steps of the LAK certification protocol—specifically all steps except for Steps 1 and 6c—are roughly included in this procedure. Therefore, this section need only expound on the details which differ from the previous verification process.

Modeling this protocol begins by defining parameters for each the OEM and CA. The OEM has its IAK, EK, EK certificate, and device identifying information and the CA has its own key, the public key of the TPM Manufacturer's CA, and a secret nonce. The procedure may be regarded

```
(* OEM parameters *)
Parameter pubIAK : pubKey.
Parameter pubEK : pubKey.
Parameter certEK : signedCert.
Parameter devInfo : deviceInfoType.

(* CA parameters *)
Parameter pubCA : pubKey.
Parameter pubTM : pubKey.
Parameter nonce : randType.
```

Figure 5.5: Parameters of IAK Certification Protocol

as the composition of four parts: the OEM's initial steps (i.e., Steps 0-4) followed by the CA's initial steps (i.e., Steps 5-6) followed the OEM's final steps (i.e., Step 7) followed by the CA's final steps (i.e., Steps 8-9). The initial steps of the OEM and the OEM's CA are constructed similarly to the steps of the Owner and the Owner's CA respectively. In any case, the OEM's final steps are naturally constructed as a simple function; first the OEM waits to receive a challenge blob from the CA, then the OEM executes Step 7 of the procedure. As a result of these definition, the CA's final steps are implicit in the proof statements and do not require an explicit definition.

Now proving Assurance A is trivial and proceeds identically to the corresponding proof in the previous section. Proving Assurance C is in fact quite similar to this proof as well. The comand `CheckCert (Cert k0 id0 k_ca')` pubTM in the CA's function corresponds with Step 5c of the

```
Definition steps_CA (msg : message) (ident : identifier) (ek iak : pubKey) (cert : signedCert) : Prop :=
  match msg with
  | (pair (TCG_CSR_IDevID (Device_info id) (Cert k0 id0 k_ca') k) (signature m k')) ⇒
        ident = (Device_info id) ∧ ek = k0 ∧ iak = k ∧ cert = (Cert k0 id0 k_ca') ∧
        seq_execute (iniTPM_CA, inferFrom msg ++ ini_CA)
                        (CheckHash
                           msg
                           (TCG_CSR_IDevID (Device_info id) (Cert k0 id0 k_ca') k) ;;
                         CheckSig
                           (signature m k')
                            k ;;
                         CheckCert
                           (Cert k0 id0 k_ca')
                            pubTM ;;
                         CheckAttributes
                            k
                            Restricting Signing NonDecrypting Fixing ;;
                         TPM2_Hash
                           (publicKey k);;
                         TPM2_MakeCredential
                           (hash (publicKey k))
                            nonce
                            k0 ;;
                         Done)
                        (hash (publicKey k) :: iniTPM_CA,
                         encryptedCredential (hash (publicKey k)) nonce k0 :: hash (publicKey k)
                         :: inferFrom msg ++ ini_CA)
  | _ ⇒ False
  end.



Definition step7_OEM (msg : message) : sequence :=
TPM2_ActivateCredential
    msg
    privEK
    privIAK ;;
Done.
```

Figure 5.6: Model of IAK Certification Protocol

certification procedure (see that the CA's function binds the EK to the variable `k0` and the EK certificate to the message `Cert k0 id0 k_ca'`). Then it is clear that successful execution of this command directly implies that the EK certificate is valid.

With that complete, let us now attempt formal verification of Assurance B under the conditions of scenario 1: the OEM and the CA are both trusted to execute their steps correctly. I implement the same strategy as before, that is I aim to show that the private IAK and private EK are contained in the OEM's minimal initial `tpm_state`. I build a minimal initial state pair for the composition of `steps1to4_OEM` and `step7_OEM` using the following intuition: (i) the EK certificate and public IAK are known to the Owner because they are included in the CSR and (ii) the private IAK and private EK reside in the same TPM because the IAK is credentialed by the EK. This intuition is used to guide the proof of the lower bound property. While the sufficiency property uses the preconditions that the CA decides to issue the IAK certificate and that the EK has good attributes. The completion of these proofs confirm that Assurance B is in fact guaranteed by the protocol when we assume that both the OEM and the CA are trusted to execute their steps correctly.

Therefore, let us now attempt formal verification of this same goal under the conditions of scenario 2: only the CA is trusted to execute its steps correctly. I describe the OEM and its characteristics as a series of assumptions: the OEM executes some unknown sequence of commands `s1`, this sequence produces some message `msg` in the OEM's intermediate `state`, the OEM's initial `tpm_state` may only contain private keys, the OEM's initial `state` may only contain public keys and certificates, the CA executes its prescribed sequence on the message and sends the challenge blob `encryptedCredential (hash (publicKey iak)) g ek` to the OEM, the OEM executes some unknown sequence of commands `s2`, this sequence releases the secret nonce value g into the OEM's final `state`. The OEM's final steps are implicit in the fact that the nonce in the challenge blob is the same as the nonce in the OEM's final `state`.

In order to complete this verification, no further knowledge on the OEM's initial sequence `s1` is needed. On the other hand, the OEM's final sequence `s2` must be analyzed. I prove that `s2` is a supersequence of the correct final steps of the OEM (i.e., `step7_OEM`). This proof relies on two important, related properties of sequential execution. The first being that one cannot produce a random number in its final `state` without an encrypted credential containing that number in its initial `state`. The next property states approximately the converse of the first, that is, one cannot

```
∀ s2 s1 iniTPM ini midTPM mid finTPM fin msg ident ek iak cert g,
seq_execute (iniTPM, ini) s1 (midTPM, mid) →
In msg mid →
(∀ m', needsGeneratedTPM m' → ¬In m' iniTPM) →
(∀ m', needsGenerated m' → ¬In m' ini) →
steps_CA msg ident ek iak cert →
seq_execute (midTPM, inferFrom (encryptedCredential (hash (publicKey iak)) g ek) ++ mid)
            s2
            (finTPM, fin) →
In (randomNum g) fin →
```

Figure 5.7: Assumptions on the Untrusted OEM

produce an encrypted credential containing a particular random number in its final `state` without

that number or an encrypted credential containing that number in its initial `state`.

Chapter 6

Conclusion

6.1 Conclusion

The research presented in this thesis has resulted in the development of a modeled command library and execution environment which are useful in verifying properties over certain key certification protocols.

6.2 Future Work

Although the modeled command library and execution environment are useful in verifying properties over some key certification protocols, further improvements are necessary to extend the applicability of this model to a broader range of problems. An initial step towards achieving this is to expand the command library to include the TPM commands necessary for modeling the attestation variation of these protocols. These attestation variations use PRCs to inform a remote CA of the internal state of a certificate-requesting entity. This variation is strongly encouraged to be performed during certification of an IAK so that the CA may be assured it is issuing a certificate to a device running trusted software. In addition to adding those particular PCR-related commands, it is useful to include more TPM and TSS commands in general. By expanding the range of commands supported by this model, we can describe a broad range of situations, even those unrelated to key certification protocols. Besides expanding the modeled TPM command library, we may also improve on the execution environment. Specifically, more complex control sequences such as branching and looping should be included so that we can describe more elaborate scenarios.

In conclusion, this research has revealed several areas for further exploration and improvement. These future areas of work will enhance the capabilities of this model and provide a formal system for verification of TPM-related properties.

References

[1] 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.

[2] Barker, E. (2020). Recommendation for key management. *NIST Special Publication 800-57 Part 1 Revision 5*.

[3] Delaune, S., Kremer, S., Ryan, M. D., & Steel, G. (2011a). A Formal Analysis of Authentication in the TPM. In P. Degano, S. Etalle, & J. Guttman (Eds.), *Formal Aspects of Security and Trust* (pp. 111–125). Berlin, Heidelberg: Springer Berlin Heidelberg.

[4] Delaune, S., Kremer, S., Ryan, M. D., & Steel, G. (2011b). Formal Analysis of Protocols Based on TPM State Registers. In *2011 IEEE 24th Computer Security Foundations Symposium* (pp. 66–80).

[5] Halling, B. (2013). Towards a Formal Verification of the Trusted Platform Module. Master's thesis, University of Kansas.

[6] Halling, B. & Alexander, P. (2013). Verifying a Privacy CA Remote Attestation Protocol. In G. Brat, N. Rungta, & A. Venet (Eds.), *NASA Formal Methods* (pp. 398–412). Berlin, Heidelberg: Springer Berlin Heidelberg.

[7] Institute of Electrical and Electronics Engineers (2018). IEEE Standard for Local and Metropolitan Area Networks - Secure Device Identity. *IEEE Std 802.1AR-2018*.

[8] Paulin-Mohring, C. (2014). Introduction to the Calculus of Inductive Constructions.

[9] Shao, J., Qin, Y., & Feng, D. (2018). Formal Analysis of HMAC Authorisation in the TPM 2.0 Specification. *IET Information Security*, 12(2), 133–140.

[10] Shao, J., Qin, Y., Feng, D., & Wang, W. (2015). Formal Analysis of Enhanced Authorization in the TPM 2.0. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15 (pp. 273–284). New York, NY, USA: Association for Computing Machinery.

[11] Trusted Computing Group (2019). *Trusted Platform Module Library Specification, Family 2.0, Level 00, Revision 01.59*.

[12] Trusted Computing Group (2021a). *TCG EK Credential Profile For TPM Family 2.0, Level 00, Version 2.4, Revision 3*.

[13] Trusted Computing Group (2021b). *TPM 2.0 Keys for Device Identity and Attestation*.

[14] Wesemeyer, S., Newton, C. J., Treharne, H., Chen, L., Sasse, R., & Whitefield, J. (2020). Formal Analysis and Implementation of a TPM 2.0-Based Direct Anonymous Attestation Scheme. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20 (pp. 784–798). New York, NY, USA: Association for Computing Machinery.