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 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. 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 on TPM-residency*. 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

the following capitalized titles of TPM Manufacturer, OEM, and Owner are keywords that will be referenced throughout this paper

simplified version of the process for creating and distributing TPM-containing devices to the end user

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).

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

the provisioning of device identification in Steps 2 and 3 is the subject of this paper

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 [1]. 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. Enhanced authorization (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. 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. Public key 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 in the scenario that the device's hard drive gets wiped.

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 key 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 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. 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 [4]. 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 in which the key itself resides. 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 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.

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 [5]. 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), a certificate's identity field contains non-TPM device information. This information should be globally unique per device [2]. In this model, certificates are abstractly defined as the inductive signedCert type. A signedCert requires a public key, an identifier, and a private key. An identifier may include 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 2.2: Model of Certificates

2.2 Logical Foundation of Coq

The Calculus of Inductive Constructions (CIC) is the underlying formalism of the interactive proof assistant Coq [3]. 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 2.3: 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.

Chapter 3

Secure Device Identity

A secure device identifier (DevID) is defined as an identifier that is cryptographically bound to a device [2]. A device with DevID capability includes at least one Initial Device Identifier (IDevID). An IDevID must not be transferable from one device to another and must be stored in a way that protects it from modification. Furthermore, an IDevID is intended to be long-lived and usable for the lifetime of the device. Since the TPM is a secure Root of Trust for Storage and protects keys against compromise, TPM keys are an ideal choice for IDevIDs. The TCG specifically makes this recommendation as well. IDevIDs are installed by OEMs in TPM-containing devices at manufacturing time. Additionally, a device with TPM-based DevID capability may support the creation of Locally Significant Device Identifiers (LDevIDs) by the end user (i.e., the Owner). An LDevID must not be transferable 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. The `Restricted` attribute may optionally be 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. This idea is discussed in later sections. The acronym AK is prefixed by the letter I or L denoting Initial or Locally Significant respectively. The `FixedTPM` attribute must always be set because it is of paramount importance that a DevID never be duplicated, transferred, or copied. OEM-installed DevIDs (i.e., IAKs and IDevIDs) should be Primary keys so that they may be recoverable by the Owner. Since an Owner cannot provision new IAKs or IDevIDs on their device, these keys should

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

be able to be recreated during the lifetime of the TPM or more precisely during the lifetime of the TPM's Primary Seed to avoid the problematic loss of these essential identifiers. An IAK is intended to be used only for the certification of new DevIDs. On the other hand, an LAK may be used for attestation as well as for the certification of new DevIDs. An IDevID or LDevID may be used for general device authentication purposes.

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'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. The Owner's CA should ideally also verify the attributes and TPM residency of a key before signing a certificate. 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. If a CA issues a certificate, then it should be the case that applying the corresponding function on the DevID results in the value `True`. All CAs should support a standard certificate transport protocol that provides confidentiality, integrity, and protection from replay attacks [6]. These transport protocols are outside the scope of this paper. We assume CAs to be following this recommendation precisely.

```
Definition endorsementKey (k : pubKey) : Prop :=
  match k with
  | Public _ Restricting NonSigning Decrypting Fixing ⇒ True
  | _ ⇒ False
  end.

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

Definition devidKey (k : pubKey) : Prop :=
  match k with
  | Public _ NonRestricting Signing NonDecrypting Fixing ⇒ True
  | _ ⇒ False
  end.
```

Figure 3.1: Model of Key Attribute Requirements

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. In issuing an IAK certificate, the OEM's CA makes an assertion that is a primary security dependency for future enrollment of all DevIDs [6]. Since only AKs (i.e., DevIDs with the `Restricted` attribute set) may be used for key certification, enrollment of all DevIDs is reliant, either directly or indirectly, on an IAK certificate. Figure 3.2 shows that the IDevID, LAK, and LDevID may all be linked back to the IAK. The creation of an IAK certificate relies on the EK certificate. The TPM Manufacturer's method for creation of an EK certificate is outside the scope of this paper since an EK is not a DevID. We trust that an EK certificate provides definitive evidence that the EK resides within the specific TPM.

IAK certificates provide definitive evidence to a remote entity that a key belongs to a specific device. To demonstrate that some other key belongs to a specific device, one must provide all of the certificates which link that key to an IAK. These dependencies are a direct result of the protocols performed to issue these certificates.

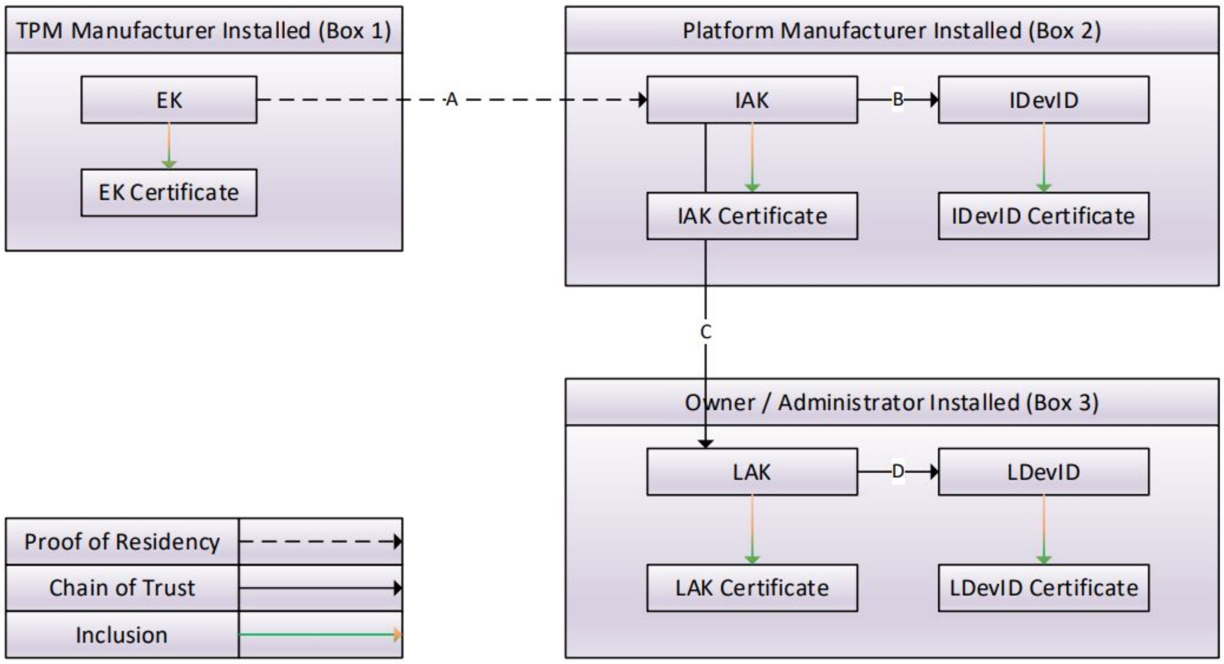

Figure 3.2: Key and Certificate Relationships [6]

- 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'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.

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. An Owner may install new LAKs and LDevIDs as often as need. While an IDevID certificate always relies directly on an IAK certificate, an LAK or LDevID certificate may be directly reliant on either an IAK or LAK certificate. In particular, an LAK certificate may be used to enroll LDevIDs or additional LAKs creating an arbitrary long chain of certificates. All DevIDs (excluding IAKs) must be directly linked to an AK because the `Restricted` attribute is necessary in proving that the new key resides in a specific TPM (i.e., the same TPM as the AK).

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 (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. 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 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 consid-

```
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

ered successful. These conditions are manifested in several ways: constructors pattern match on 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 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 name of a key to be credentialed, a secret, and a public EK. All three of these inputs must be contained in the caller's initial state.

The TPM2_ActivateCredential command

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, I only include the fields necessary for creating an IAK certificate (i.e., deviceidentifying 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 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

```
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

seq_execute relation. This means that seq_execute is a partial function. Next, sequential execution 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). 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. Furthermore, sequential execution cannot generate encrypted credential blobs without a nonce in the initial state. The proof statements are recorded below (See Appendix C for the proofs themselves). I use the contrapositive statements for the theorems regarding nonce and credential generation for convenient use in later proofs.

```
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

I define a recursive function which determines whether a provided command is an element of a provided sequence. Not only is the command itself required to match but all of its inputs must as well.

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*. We consider in detail two of these protocols: OEM creation of an IAK certificate based on an EK certificate and Owner creation of an LAK certificate based on an IAK certificate. We select 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 specification outlines steps for each the CA and the certificate-requesting entity to perform. Furthermore, the specification claims that each protocol provides certain assurances. The TCG themselves do not present any proofs or clear justifications to support these claims. Therefore, our goal is to abstractly model these protocols using the definitions from Section 4 and then formally verify that the corresponding assurances do in fact hold.

We 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, we 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 that the associated assurances hold. For convenience and clarity, we inspect each of these protocols in the reverse order that their dependencies entails. We 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 following procedure guarantees that the new LAK is resident in the same TPM as the IAK.

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

We begin to model this protocol 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 parameters; we store these values in the `privLAK`, `privIAK`, and `privCA` variables. To enforce the randomness of cryptographic keys, we 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, we can abstractly define each of these parts in our model. We use the parameters defined above to construct an object with `sequence` type for the Owner and an object with function type for the CA. Constructing the Owner's steps is straightforward. Since we disallow the arbitrary creation of keys, we assume step 0 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.

```
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 m ++ 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 ;;
                     Done)
                    (iniTPM_CA, inferFrom m ++ ini_CA)
  | _ ⇒ False
  end.
```

Figure 5.2: Model of LAK Certification Protocol

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 definition, 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). We 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 abstractly modeling the conditions of each scenario. Thus let us now attempt formal verification of our goal under the conditions of scenario 1: the Owner and the CA are both trusted to execute their steps correctly. Recall first that our goal is to show that this certification procedure guarantees that the new LAK is resident in the same TPM as the IAK. Also recall that we trust that the IAK certification procedure guarantees its associated assurances, primarily that the IAK has good attributes.

We begin by examining the Owner's steps and its requirements. We can quantitatively describe these requirements 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. We build a minimal initial state pair for `steps1to5_Owner` using the following intuition. 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 known to the Owner since it is included in the CSR. The lower bound property is proven to hold in general over this state pair. We use the intuition behind building this minimal initial state pair to guide this proof. On the other hand, the proof of the sufficiency property requires several preconditions: (1) the IAK has good attributes and (2) the Owner's CA decides to issue a certificate for the LAK. The first precondition is safe to assume since we trust the OEM's CA to have checked the attributes on the IAK when issuing its respective certificate. And the second precondition is clearly safe to assume as it is a direct consequence of scenario 1. See Appendix D for the completed proofs of lemmas `ini_Owner_lowerBound` and `ini_Owner_sufficient`. This analysis of the Owner's steps' requirements in the form of a minimal initial state conveniently

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

Figure 5.3: Minimal Initial State of Owner

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 the claimed assurance is in fact guaranteed when we assume that both the Owner and the CA are trusted to execute their steps correctly.

Now let us 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, we consider the Owner to be this entity. To this end, we 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 `m` 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. We argue that these assumptions are reason-

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

Figure 5.4: Assumptions on the Untrusted Owner

able 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 contstraint. The `needsGeneratedTPM` function restricts the Owner's initial `tpm_state` to the inclusion of previously created private keys. While the `needsGenerated` function restricts the 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 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 super sequence of the correct steps of the Owner (i.e., `steps1to5_Owner`). A list is a super sequence of another list if all the elements of the second list occur, in order, in the first — the elements do not need to occur consecutively. We define a cascading collection of recursive functions to determine whether a given sequence of commands is a super sequence of `steps1to5_Owner` (See Appendix D for definitions). Specifically, we prove that if we make all of the initial assumptions regarding the Owner and the CA (i.e., those in Figure 5.4 plus `steps_CA m iak lak cert`) then the Owner's unknown sequence `s` is actually a super sequence of `steps1to5_Owner`.

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. Since we require all of the commands inputs to match exactly, we use the recursive funtion `command_in_sequence` to accurately describe this situation. Then we need only assume the conclusion obtained above to complete this helpful lemma. And finally we prove one last set of lemmas which states that the private IAK and private LAK must reside in the same TPM if one executes a sequence which contains that particular command. Now it is apparent that the composition of all of these small proofs leads to our end goal.

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

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

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

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

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

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

Appendix A

Model of Inference

```
Lemma inferFrom_iff_inferrable : ∀ m st,
  inferFrom m = st ↔ inferrable m st.
Proof.
  intros m st; split; intros H.
  - generalize dependent m; assert (HI : ∀ 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 crt; reflexivity.
Qed.
```

```
Fixpoint inferFrom (msg : message) : state :=
  match msg with
  | signature m k ⇒
      (signature m k ::  inferFrom m)
  | TPM2B_Attest k ⇒
      [TPM2B_Attest k ; publicKey k]
  | TCG_CSR_IDevID id0 (Cert k id k_ca) k0 ⇒
      [TCG_CSR_IDevID id0 (Cert k id k_ca) k0 ; publicKey k0 ;
       signedCertificate (Cert k id k_ca) ; publicKey k]
  | TCG_CSR_LDevID m (Cert k id k_ca) ⇒
      (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) ⇒
      [signedCertificate (Cert k id k_ca) ; publicKey k]
  | pair m1 m2 ⇒
      (pair m1 m2 ::   inferFrom m1 ++ inferFrom m2)
  | _ ⇒
      [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 crt k st,
    inferrable (signedCertificate crt) st →
    inferrable (TCG_CSR_IDevID id crt k)
               (TCG_CSR_IDevID id crt k :: publicKey k :: st)
| I_CSR_LDevID : ∀ m crt st1 st2,
    inferrable m st1 →
    inferrable (signedCertificate crt) st2 →
    inferrable (TCG_CSR_LDevID m crt)
               (TCG_CSR_LDevID m crt :: 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

Model of Execution

```
Inductive execute : tpm_state ∗ state → command → tpm_state ∗ state → Prop :=

| E_Hash : ∀ stTPM st m,
    In m st →
    execute (stTPM, st)
            (TPM2_Hash m)
            (hash m ::  stTPM, hash m ::  st)

| E_CheckHash : ∀ stTPM st m,
    In (hash m) st →
    In m st →
    execute (stTPM, st)
            (CheckHash (hash m) m)
            (stTPM, st)

| E_SignNR : ∀ stTPM st m i d f,
    In (privateKey (Private i NonRestricting Signing d f)) stTPM →
    In m st →
    execute (stTPM, st)
            (TPM2_Sign m (Private i NonRestricting Signing d f))
            (stTPM, signature m (Private i NonRestricting Signing d f) :: st)

| E_SignR : ∀ stTPM st m i d f,
    In (privateKey (Private i Restricting Signing d f)) stTPM →
    In m stTPM →
    execute (stTPM,st)
            (TPM2_Sign m (Private i Restricting Signing d f))
            (stTPM, signature m (Private i Restricting Signing d f) :: st)
```

```
| E_Certify : ∀ stTPM st i r s d f i0 r0 d0 f0,
    In (privateKey (Private i r s d f)) stTPM →
    In (privateKey (Private i0 r0 Signing d0 f0)) stTPM →
    execute (stTPM, st)
            (TPM2_Certify (Public i r s d f) (Private i0 r0 Signing d0 f0))
            (stTPM, signature (TPM2B_Attest (Public i r s d f)) (Private i0 r0 Signing d0 f0) :: st)

| E_CheckSig : ∀ stTPM st m i r s d f,
    In (signature m (Private i r s d f)) st →
    In (publicKey (Public i r s d f)) st →
    execute (stTPM, st)
            (CheckSig (signature m (Private i r s d f)) (Public i r s d f))
            (stTPM, st)

| E_MakeCredential : ∀ stTPM st n g i,
    In n st →
    In (randomNum g) st →
    In (publicKey (Public i Restricting NonSigning Decrypting Fixing)) st →
    execute (stTPM, st)
            (TPM2_MakeCredential n g (Public i Restricting NonSigning Decrypting Fixing))
            (stTPM, encryptedCredential n g (Public i Restricting NonSigning Decrypting Fixing) :: st)

| E_ActivateCredential : ∀ stTPM st n g i r s d f i0 r0 s0 d0 f0,
    In (encryptedCredential n g (Public i r s d f)) st →
    In (privateKey (Private i r s d f)) stTPM →
    In (privateKey (Private i0 r0 s0 d0 f0)) stTPM →
    execute (stTPM, n:: st) (CheckHash n (publicKey (Public i0 r0 s0 d0 f0))) (stTPM, n:: st) →
    execute (stTPM, st)
            (TPM2_ActivateCredential (encryptedCredential n g (Public i r s d f))
                                     (Private i r s d f)
                                     (Private i0 r0 s0 d0 f0))
            (stTPM, randomNum g :: st)
```

```
| E_MakeCSR_IDevID : ∀ stTPM st id crt k,
    In (signedCertificate crt) st →
    In (publicKey k) st →
    execute (stTPM, st)
            (MakeCSR_IDevID id crt k)
            (stTPM, TCG_CSR_IDevID id crt k :: st)

| E_MakeCSR_LDevID : ∀ stTPM st m crt,
    In m st →
    In (signedCertificate crt) st →
    execute (stTPM, st)
            (MakeCSR_LDevID m crt)
            (stTPM, TCG_CSR_LDevID m crt :: st)

| E_CheckCert : ∀ stTPM st k id i r s d f,
    In (signedCertificate (Cert k id (Private i r s d f))) st →
    In (publicKey (Public i r s d f)) st →
    execute (stTPM, st)
            (CheckCert (Cert k id (Private i r s d f)) (Public i r s d f))
            (stTPM, st)

| E_CheckAttributes : ∀ stTPM st i r s d f,
    In (publicKey (Public i r s d f)) st →
    execute (stTPM, st)
            (CheckAttributes (Public i r s d f) r s d f)
            (stTPM, st)

| E_MakePair : ∀ stTPM st m1 m2,
    In m1 st →
    In m2 st →
    execute (stTPM, st)
            (MakePair m1 m2)
            (stTPM, pair m1 m2 :: st).
```

Appendix C

Theroems on Sequential Execution

```
Theorem seq_exec_deterministic : ∀ ini s fin1 fin2,
  seq_execute ini s fin1 →
  seq_execute ini s fin2 →
  fin1 = fin2.
Proof.
  intros ini s fin1 fin2 E1 E2; generalize dependent fin2;
  induction E1; intros fin2 E2; inversion E2; subst.
  – assert (mid = mid0) as EQ_mid;
    [ apply exec_deterministic with (ini := ini) (c := c)
    | apply IHE1; rewrite ← EQ_mid in H5 ]; assumption.
  – reflexivity.
Qed.

Theorem seq_exec_expansion : ∀ iniTPM ini s finTPM fin,
  seq_execute (iniTPM,ini) s (finTPM,fin) →
  (iniTPM ⊆ finTPM) ∧ (ini ⊆ fin).
Proof.
  intros iniTPM ini s finTPM fin E; split;
  generalize dependent fin; generalize dependent finTPM;
  generalize dependent ini; generalize dependent iniTPM;
  induction s; intros;
  inversion E; subst; try (intros m' I; assumption);
  destruct mid as [midTPM mid];
  assert (Inc : (iniTPM ⊆ midTPM) ∧ (ini ⊆ mid));
  try (apply exec_expansion with (c := c); assumption).
  – apply Included_transitive with (st2 := midTPM);
    [ apply Inc
    | apply IHs with (ini := mid) (fin := fin); assumption ].
  – apply Included_transitive with (st2 := mid);
    [ apply Inc
    | apply IHs with (iniTPM := midTPM) (finTPM := finTPM); assumption ].
Qed.
```

```
Theorem seq_exec_cannotGenerateKey : ∀ s iniTPM ini finTPM fin k,
  seq_execute (iniTPM, ini) s (finTPM, fin) →
  In (privateKey k) finTPM →
  In (privateKey k) iniTPM.
Proof.
  induction s; intros iniTPM ini finTPM fin k E I;
  inversion E; subst; try assumption; destruct mid as [midTPM mid];
  eapply exec_cannotGenerateKey; eauto.
Qed.

Theorem seq_exec_cannotGenerateRand_contrapositive : ∀ s iniTPM ini finTPM fin g,
  seq_execute (iniTPM, ini) s (finTPM, fin) →
  ¬ In (randomNum g) ini →
  (∀ n k, ¬ In (encryptedCredential n g k) ini) →
  ¬ In (randomNum g) fin.
Proof.
  induction s; intros iniTPM ini finTPM fin g E Ng N.
  - destruct c; inversion E; inversion H2;
    subst; eapply IHs; try fdeq;
    try (intros n0 k HC; inversion HC; try congruence; destruct (N n0 k); assumption);
    intros HC; inversion HC;
    [ inversion H; subst; destruct (N n (Public i r s1 d f)); assumption
    | congruence ].
  - inversion E; subst; congruence.
Qed.

Theorem seq_exec_cannotGenerateCred_contrapositive : ∀ s iniTPM ini finTPM fin n g k,
  seq_execute (iniTPM, ini) s (finTPM, fin) →
  (∀ n k, ¬ In (encryptedCredential n g k) ini) →
  ¬ In (randomNum g) ini →
  ¬ In (encryptedCredential n g k) fin.
Proof.
  induction s; intros iniTPM ini finTPM fin n g k E Ne N;
  inversion E; subst;
  [ destruct mid as [midTPM mid]; eapply IHs; eauto
  | intros HC; destruct (Ne n k); assumption];
  [ intros n0 k0; eapply exec_cannotGenerateCred_contrapositive; eauto
  | eapply exec_cannotGenerateRand_contrapositive; eauto].
Qed.
```

```
Lemma exec_deterministic : ∀ ini c fin1 fin2,
  execute ini c fin1 →
  execute ini c fin2 →
  fin1 = fin2.
Proof.
  intros ini c fin1 fin2 E1 E2;
  destruct E1; inversion E2; subst; reflexivity.
Qed.

Lemma exec_expansion : ∀ iniTPM ini c finTPM fin,
  execute (iniTPM,ini) c (finTPM,fin) →
  (iniTPM ⊆ finTPM) ∧
  (ini ⊆ fin).
Proof.
  intros iniTPM ini c finTPM fin E; split;
  destruct c; inversion E; subst;
  intros m' I;  try (repeat apply in_cons); assumption.
Qed.

Lemma exec_cannotGenerateKey : ∀ c iniTPM ini finTPM fin k,
  execute (iniTPM, ini) c (finTPM, fin) →
  In (privateKey k) finTPM →
  In (privateKey k) iniTPM.
Proof.
  destruct c; intros iniTPM ini finTPM fin k E I;
  inversion E; subst; try inversion I as [EQ_false | I'];
  try inversion EQ_false; assumption.
Qed.

Lemma exec_cannotGenerateRand_contrapositive : ∀ c iniTPM ini finTPM fin g,
  execute (iniTPM, ini) c (finTPM, fin) →
  ¬ In (randomNum g) ini →
  (∀  n k,  ¬ In (encryptedCredential n g k) ini) →
  ¬ In (randomNum g) fin.
Proof.
  induction c; intros iniTPM ini finTPM fin g E Ng N;
  inversion E; subst; try assumption;
  intros HC; inversion HC; try inversion H; subst; try congruence;
  destruct (N n (Public i r s  d  f));   assumption.
Qed.

Lemma exec_cannotGenerateCred_contrapositive : ∀ c iniTPM ini finTPM fin n g k,
  execute (iniTPM, ini) c (finTPM, fin) →
  ¬ In (encryptedCredential n g k) ini →
  ¬ In (randomNum g) ini →
  ¬ In (encryptedCredential n g k) fin.
Proof.
  induction c;  intros iniTPM ini finTPM fin n g k E Ne N;
  inversion E; subst; try assumption;
  intros HC; inversion HC; try inversion H; congruence.
Qed.
```