Verifying a Privacy CA Remote Attestation Protocol*

Brigid Halling and Perry Alexander

Information and Telecommunication Technology Center
The University of Kansas
{bhalling,palexand}@ku.edu

Abstract. As the hardware root-of-trust in a trusted computing environment, the Trusted Platform Module (TPM) warrants formal specification and verification. This work presents results of an effort to specify and verify an abstract TPM 1.2 model using PVS that is useful for understanding the TPM and verifying protocols that utilize it. TPM commands are specified as state transformations and sequenced to represent protocols using a state monad. Postconditions and invariants are specified for individual commands and validated by verifying a Privacy CA attestation protocol. All specifications are written and verified automatically using the PVS decision procedures and rewriting system.

1 Introduction

At the heart of trusted computing [3] is the need to appraise a remote system in a trusted fashion. In this process – known as *remote attestation* [4, 5, 11] – an external appraiser sends an attestation request to an appraisal target and receives a quote used to assess the remote system's state. To achieve its goal, the appraiser must not only analyze the quote's contents, but also assess the trustworthiness of the information it contains.

The Trusted Platform Module (TPM) and its associated Trusted Software Stack (TSS) [1] provide core functionality for assembling and delivering a quote for appraisal with high integrity as well as binding confidential data to a specific platform. However, neither the TPM nor TSS have been formally specified or verified. Definitions of the over 90 current TPM commands as well as additional TSS commands are embedded in more than 700 pages of English documentation.

We formally specify and verify a remote attestation protocol – known as the Privacy CA Protocol – using commands from TPM version 1.2. Our objective is to capture an abstract specification from the TPM specification, validate it, and use it to verify the correctness of the Privacy CA Protocol. We are not making an argument for the protocol itself, we are merely verifying this protocol as a part of verifying the TPM. We use PVS [14] for our work, however the results and approach generalize to other tools.

* This work was sponsored in part by the Battelle Memorial Institute under PO US001-0000328568

1.1 Trusted Platform Module

The Trusted Platform Module (TPM) [1] is a hardware co-processor that provides cryptographic functions at the heart of establishing and maintaining a trusted computing infrastructure [3]. The TPM's functionality can be distilled into three major capabilities: (i) establishing, maintaining, and protecting a unique identifier; (ii) storing and securely reporting system measurements; and (iii) binding secrets to a specific platform.

The *endorsement key* (EK) and *storage root key* (SRK) are persistent asymmetric keys maintained by the TPM. EK uniquely identifies the TPM and EK^{-1} is maintained confidentially while EK encrypts secrets for use by TPM. EK^{-1} could theoretically sign TPM data, but is never used for this purpose to avoid unintended information aggregation. Instead, it provides a root-of-trust for reporting used in the attestation process. The SRK provides a root key for chaining *wrapped keys*. A wrapped key is an asymmetric key pair whose private key is encrypted by another asymmetric key. The resulting wrapped key can be safely stored outside the TPM and may only be installed and used if its wrapping key is installed. Using the SRK as the root of these chains binds information to its associated TPM.

A *platform configuration register* (PCR) is a special purpose register for storing and extending hashes within the TPM. As its name implies, a PCR records a platform's configuration during boot or at run time. The TPM ensures the integrity of PCRs and uses a quote mechanism to deliver them with integrity to an external appraiser. Rather than being set to a specific value, PCRs are extended using the formula $pcr \parallel h = SHA1(pcr +\!\!+ h)$. These hashes – called *measurements* – are gathered in PCRs at various points during system operation, but the most common use is to ensure trusted boot. As each system component boots, images and data are hashed, and each hash is used to extend a PCR. The nature of extension implies that at the conclusion of the boot process, the hashes in PCRs indicate whether the right parts were used in the right order during boot. Specifically, ideal PCR extension exhibits the property that $h_0 \parallel h_1 = h_1 \parallel h_0 \Leftrightarrow h_0 = h_1$. The only way to change a PCR value is with a platform reboot or by using the command `TPM_Extend`.

1.2 Privacy CA Protocol

Remote Attestation using a TPM is the process of gathering PCRs and delivering them to an external appraiser in a trusted fashion [9]. By examining the reported contents of PCRs, the appraiser can determine whether it trusts the system described. Using hashes guarantees the appraiser only learns whether the right system is running and nothing more. Our remote attestation method is to use a *Privacy Certificate Authority* (CA or Privacy CA) that produces an identity certificate verifying that an *attestation identity key* (AIK) public key belongs to a certain TPM using its EK. The Privacy CA is so named because it protects the EK while assuring the AIK belongs to the right EK. This protocol is shown in figure 1.

An AIK, wrapped by the SRK, is created using the TPM's `TPM_MakeIdentity` command and can only be used by the TPM that generated it. The command also returns a CA label digest identifying the CA certifying the AIK, and the public AIK signed with AIK^{-1}. The AIK signature tells us that the AIK came from the right TPM since the TPM that generated the AIK is the only entity with access to its private key. Using the public key embedded in the certificate, the CA can determine if the entire certificate did indeed come from the TPM associated with the AIK.

Although we are modeling the TPM, we also need to model the role of the Privacy CA. This interaction between the CA and the User is modeled by `CA_certify`. The CA returns a session key (identified as `K` with figure 1) encrypted by the public EK associated with the TPM that claims to have requested the certificate. `TPM_ActivateIdentity` attempts to decrypt `K` using the TPM's EK^{-1} and releases if it decrypts successfully. Finally, we are able to use the AIK to sign PCR values using the TPM command `TPM_Quote` [1]. This quote is returned to the User who can then send back to the appraiser the information that it needs. The command `CPU_BuildQuoteFromMemory` simulates this final step generating for the appraiser an evidence package of the form:

$$(\{|\{|AIK|\}_{CA^{-1}}|\}_{AIK^{-1}}, \{|n, PCR|\}_{AIK^{-1}}) \tag{1}$$

where: $\{|n, PCR|\}_{AIK^{-1}}$ is the nonce from the appraiser's request and desired PCR values; $\{|\{|AIK|\}_{CA^{-1}}|\}_{AIK^{-1}}$ is the certificate from a Privacy CA and public AIK; and both are signed by the AIK.

2 System Model

The overall approach we take for verifying the TPM is to establish a *weak bisimulation* [17] relation between an abstract requirements model and a concrete model derived from the TPM specification. Both the abstract and concrete models define transition systems in terms of system state and transitions over that system state. Here we address only the abstract model, useful in its own right for modeling protocols and verifying operations. Here we describe our abstract model of the TPM, including data structures and command execution.

2.1 Data Model

Our abstraction of data relevant to the TPM is defined in the PVS data type `tpmData`. Figure 2 shows a subset of this data that is relevant to verifying the remote attestation protocol. It may be noted that most elements of our `tpmData` data type include a tag that shows what cryptographic operations have been performed on data using the `CRYPTOSTATUS` type. These functions include encryption, signing, and sealing. For example, a symmetric key identified as `k:KVAL` and signed with the private key of `idKey:(tpmKey?)` is expressed as:

$$\texttt{tpmSessKey}(\texttt{k}, \texttt{signed}(\texttt{private}(\texttt{idKey}), \texttt{clear}))$$

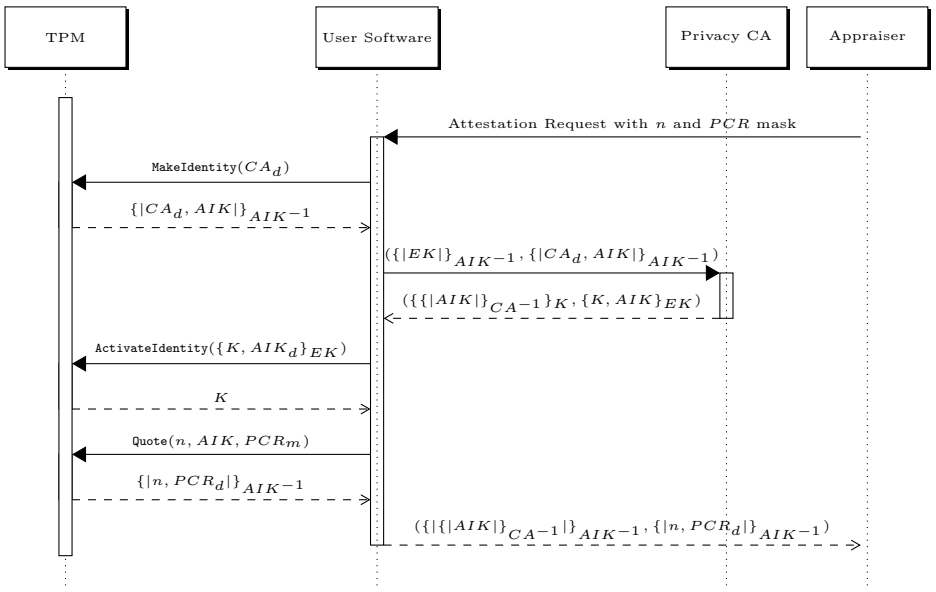

Fig. 1. Sequence diagram for the Privacy CA protocol.

```
tpmData : DATATYPE
BEGIN
 tpmDigest(digest:list[tpmData],crs:CRYPTOSTATUS) : tpmDigest?
 tpmNonce : tpmNonce?
 tpmSessKey(skey:KVAL,crs:CRYPTOSTATUS) : tpmSessKey?
 tpmKey(key:KVAL,usage:KEY_USAGE,flags:KEY_FLAGS,PCRInfo:list[PCR],
    wrappingKey:KVAL,crs:CRYPTOSTATUS) : tpmKey?
 tpmQuote(digest:list[PCR],nonce:(tpmNonce?),crs:CRYPTOSTATUS) : tpmQuote?
 tpmIdContents(digest:(tpmDigest?),aik:(tpmKey?),
    crs:CRYPTOSTATUS) : tpmIdContents?
 tpmAsymCAContents(sessK:(tpmSessKey?),idDigest:(tpmDigest?),
    crs:CRYPTOSTATUS) : tpmAsymCAContents?
 ...
END tpmData;
```

Fig. 2. Data structure for abstract TPM data.

A `tpmDigest` structure contains the list of things that are concatenated and then hashed to create the digest value – $SHA1(d_0 +\!\!+ d_1 +\!\!+ ... +\!\!+ d_n)$ while `tpmSessKey` is our representation of a symmetric key. Finally, the `tpmKey` structure represents an asymmetric key with additional properties used by the TPM. These include its usage, associated flags, and PCR information for wrapping. Virtually all asymmetric keys used by the TPM are created as wrapped keys. Thus, a reference to the wrapping key is part of the `tpmKey` structure. The type `KVAL` associated with all keys is an integer value that uniquely identifies the key.

2.2 Abstract State

The TPM manages state by maintaining several data fields and flags. We use a PVS record structure, referred to as `tpmAbsState` and shown in figure 3, to maintain an abstract view of this state as well as the memory associated with the environment where the TPM is being run. Elements key to the remote attestation protocol include `srk`, `ek`, `pcrs`, and `memory`.

```
tpmAbsState : TYPE =
  [# restore : restoreStateData, memory : mem, srk : (tpmKey?),
     ek : (tpmKey?), keyGenCnt : K, keys : KEYSET, pcrs : PCRS,
     locality : LOCALITY, permFlags : PermFlags, permData : PermData #];
```

Fig. 3. Abstract TPM and system state record data structure.

The `srk` and the `ek` represent the asymmetric keys SRK and EK used by the TPM as roots of trust previously discussed in section 1.1. `memory` is not part of the actual TPM, but represents the memory used by the TPMs environment for storing values. This is necessary for our model due to our method of command sequencing discussed in section 2.6.

`pcrs` is an array of hash sequences that define the value of a PCR. Rather than calculate the hash, a sequence of values used to create the PCR value is maintained. One unusual feature of PCRs is they can have one of two initial values. Resettable PCRs initialize to -1 (all 1s) while non-resettable PCRs reset to 0. This feature along together with PCR locality is used by the appraiser determine if the `senter` command is called during boot.

2.3 Abstract Command Definitions

Figure 4 shows the PVS data type `tpmAbsInput` that represents the abstract syntax of the TPM command set. Each `TPM_Command` will have a corresponding `ABS_Command` in the `tpmAbsInput` data structure. This approach gives us an induction principle for the command set automatically usable by PVS to quantify over all possible TPM inputs.

```
tpmAbsInput : DATATYPE
BEGIN
 ABS_MakeIdentity(CADigest:(tpmDigest?),aikParams:(tpmKey?))
    : ABS_MakeIdentity?
 ABS_ActivateIdentity(aik:(tpmKey?),blob:(tpmAsymCAContents?))
    : ABS_ActivateIdentity?
 ABS_Extend(pcrNum:PCRINDEX,d:HV) : ABS_Extend?
 ABS_Quote(aik:(tpmKey?),nonce:(tpmNonce?),pm:PCRMASK) : ABS_Quote?
 ABS_certify(aik:(tpmKey?),certReq:(tpmIdContents?)) : ABS_certify?
 ABS_save(i:nat,v:tpmAbsOutput) : ABS_save?
 ABS_read(i:nat) : ABS_read?
 ...
END tpmAbsInput;
```

Fig. 4. Representative elements from the TPM command data type.

Within the `tpmAbsInput` data structure, the arguments to each command are abstract representations of the actual TPM data formats and come from `tpmData` data type. This is appropriate for an abstract model such as ours where we are capturing functionality, not implementation. Some details are abstracted away when they do not contribute to verifying the basic functionality of the device.

2.4 Abstract Outputs

Like inputs to the TPM, outputs are modeled abstractly using an algebraic type. Again we avoid the complexity of bit-level representations specified in the TPM standard in favor of an abstract representation that captures the essence of TPM functionality. Figure 5 shows the representation of this type.

Each TPM command returns an output, often just to return the message that the command was successfully run. The `tpmAbsOutput` constructs allow for each command to return the correct output parameters as well as a return code. These return codes either indicate success or a non-fatal error. Fatal errors from TPM commands are generated using the `OUT_Error` construct, while non-TPM-related fatal errors are generated using `OUT_CPUError`.

2.5 Abstract Command Execution

The technique for specifying TPM command execution is to define state transition and output functions in the canonical fashion for transition systems. Specifically, we define the `executeCom` function as a transition from `tpmAbsState` (figure 3) and `tpmAbsInput` (figure 4) to `tpmAbsState`:

```
executeCom : tpmAbsState → tpmInput → tpmAbsState
```

and the function `outputCom` to transform `tmpAbsState` and `tpmAbsInput` into a `tpmAbsOutput` (figure 5) value:

```
tpmAbsOutput : DATATYPE
BEGIN
 OUT_MakeIdentity(aik:(tpmKey?),idc:(tpmIdContents?),m:ReturnCode)
    : OUT_MakeIdentity?
 OUT_ActivateIdentity(symmKey:(tpmSessKey?),m:ReturnCode)
    : OUT_ActivateIdentity?
 OUT_Extend(outDigest:PCR,m:ReturnCode) : OUT_Extend?
 OUT_Quote(pcrData:list[PCR],sig:(tpmQuote?),m:ReturnCode) : OUT_Quote?
 OUT_FullQuote(q:(tpmQuote?),idc:(tpmIdContents?),m:cpuReturn)
    : OUT_FullQuote?
 OUT_Certify(data:(tpmAsymCAContents?),m:cpuReturn) : OUT_Certify?
 OUT_Error(m:ReturnCode) : OUT_Error?
 OUT_CPUError(m:cpuReturn) : OUT_CPUError?
 ...
END tpmAbsOutput;
```

Fig. 5. Abstract TPM output record data structure.

$$\texttt{outputCom} : \texttt{tpmAbsState} \rightarrow \texttt{tpmAbsInput} \rightarrow \texttt{tpmAbsOutput}$$

Given $s : tpmAbsState$ and $c : tpmAbsInput$, the output, state pair resulting from executing c is defined as:

$$(\texttt{outputCom}(s, c), \texttt{executeCom}(s, c))$$

As one would expect, `executeCom` and `outputCom` are defined by cases over `tpmAbsInput`. Specifically, for each command in `tpmAbsInput` a function is defined for generating the next state and for generating output. These commands are named within the specification using the suffix `State` and `Out` respectively for easy identification.

For example, consider the `ABS_MakeIdentity` input. At its core, the command `TPM_MakeIdentity` creates the AIK and returns the public AIK key for use in other operations as well as a `tpmIdContents` structure. This `tpmIdContents` structure, containing the identity of the privacy CA that the owner expects to certify the AIK and the AIK (see figure 5 for the `OUT_MakeIdentity` structure and figure 2 for the `tpmIdContents` structure), is signed by the private AIK [1].

The function `makeIdentityState` defines how the TPM state is modified (a new key value for the AIK is created):

```
makeIdentityState(s:tpmAbsState,CADigest:(tpmDigest?),
   aikParams:(tpmKey?)) : tpmAbsState =
  IF identity?(keyUsage(aikParams))
     AND not(migratable(keyFlags(aikParams)))
   THEN s WITH [`keyGenCnt := keyGenCnt(s)+1]
   ELSE s
  ENDIF;
```

while, the function `makeIdentityOut` defines the TPM output generated by the command:

```
makeIdentityOut(s:tpmAbsState,CADigest:(tpmDigest?),
     aikParams:(tpmKey?)) : tpmAbsOutput =
  IF identity?(keyUsage(aikParams))
     AND not(migratable(keyFlags(aikParams)))
   THEN LET aik:(tpmKey?) = tpmKey(keyGenCnt(s), keyUsage(aikParams),
                                   keyFlags(aikParams), pcrs(s),
                                   wrappingKey(srk(s)),clear) IN
        LET idBinding = tpmIdContents(CADigest, aik,
                                   signed(private(aik),clear)) IN
             OUT_MakeIdentity(aik,idBinding,TPM_SUCCESS)
    ELSE OUT_Error(TPM_INVALID_KEYUSAGE)
 ENDIF;
```

Functions like `makeIdentityState` and `makeIdentityOut` define the functionality associated with `ABS_MakeIdentity`. They are associated with the command in the `executeCom` and `outputCom` using a case structure defined over `tpmAbsInput`. Since all TPM commands return at least a success or error message, all abstract commands generate output, but not all commands modify state. In instances where the state is not modified, the `CASES` construct used to assemble the functions defaults to not modifying the state.

2.6 Sequencing Command Execution

TPM commands are executed in sequence like assembly commands in a traditional microprocessor. To validate the abstract model as well as verify TPM protocols, a mechanism must be chosen to sequence command execution. Such sequencing of TPM commands is a matter of using the output state from one command as the input to the next command. The classical mechanism for doing this involves executing a command and manually feeding its resulting state to the next command in sequence. Using a `LET` form, to execute `i;i'` would look like the following:

```
LET (o',s') = (outputCom(s,i),executeCom(s,i)) IN
  (outputCom(s',i'),executeCom(s',i'))
```

We choose to use an alternative approach that uses a state monad [13, 18] to model sequential execution. The state monad threads the state through sequential execution in the background. The result is a modeling and execution pattern that closely resembles the execution pattern of TPM commands. Within PVS, we defined a state monad that gives us the traditional bind (`>>=`) and sequence (`>>`) operations. Examples of sequence and bind can be seen in figure 8.

3 Verification Results

To verify our requirements model we verify individual commands with respect to their postconditions and invariants. To provide a degree of validation, we use

those commands to model protocols and verify execution results. Some aspects of attacks are considered, but there is no attempt to be comprehensive at this time. We also assume the hash function is perfect, giving the property $SHA1(b_0) = SHA1(b_1) \Leftrightarrow b_0 = b_1$. This and the constructive specification of the `PCRS` type gives us the important property that bad hashes or bad extension ordering is detectable in the PCR value.

3.1 Verifying Individual Commands

In order to prove the validity of our abstract TPM model, we define and verify postconditions and invariants for each TPM command and verify that our abstract specifications meet those properties. We consider only partial correctness in the abstract model, as termination is meaningless at this level.

For each command, we must show that given any value for all parameters of a command, running that command produces an output, state pair that satisfies the postcondition while not violating any invariant. Note that we do not address preconditions, as TPM output for every command and for each state must be defined, therefore preconditions are always trivial. Returning to our example of the `TPM_MakeIdentity` command, we verify the postconditions of command execution with the theorem shown in figure 6.

```
make_identity_post: THEOREM
 FORALL (state:(afterStartup?),CADigest:(tpmDigest?),aikParams:(tpmKey?)):
  LET (a,s)=runState(TPM_MakeIdentity(CADigest,aikParams))(state) IN
      LET waik:(tpmKey?)=tpmKey(state`keyGenCnt, keyUsage(aikParams),
                                keyFlags(aikParams), state`pcrs,
                                state`srk`wrappingKey, clear) IN
          LET idBind=tpmIdContents(CADigest,waik,
                                   signed(private(waik),clear)) IN
IF identity?(keyUsage(aikParams))
   AND not(migratable(keyFlags(aikParams)))
 THEN a=OUT_MakeIdentity(waik,idBind,TPM_SUCCESS) AND
      s=state WITH [`keyGenCnt := keyGenCnt(state)+1]
 ELSE a=OUT_Error(TPM_INVALID_KEYUSAGE) AND
      s=state
ENDIF;
```

Fig. 6. Verifying postconditions of `TPM_MakeIdentity`.

The `LET` form runs the command starting from any state in the predicate subtype `(afterStartup?)`. This predicate ensures that the state is any valid `tpmAbsState` after the initialization commands have been run. The remainder of the theorem defines conditions on proper execution of `TPM_MakeIdentity` including both error and success cases.

State Field (Invariant)	*Abstract Commands That Change Field*
restore	ABS_Startup, ABS_Init, ABS_SaveState
memory	ABS_Startup, ABS_Init, ABS_save
srk	ABS_Startup, ABS_Init, ABS_TakeOwnership
ek	ABS_Startup, ABS_Init, ABS_CreateEndorsementKeyPair, ABS_CreateRevocableEK, ABS_RevokeTrust
keyGenCtr	ABS_Startup, ABS_Init, ABS_LoadKey2, ABS_CreateWrapKey ABS_MakeIdentity, ABS_certify
keys	ABS_Startup, ABS_Init, ABS_LoadKey2, ABS_ActivateIdentity ABS_OwnerClear, ABS_ForceClear, ABS_RevokeTrust
pcrs	ABS_Startup, ABS_Init, ABS_Extend ABS_sinit, ABS_senter
locality	ABS_Startup, ABS_Init
permFlags	ABS_Startup, ABS_Init, ABS_DisableOwnerClear, ABS_ForceClear, ABS_OwnerClear, ABS_TakeOwnership, ABS_CreateEndorsementKeyPair, ABS_CreateRevocableEK, ABS_RevokeTrust
permData	ABS_Startup, ABS_Init, ABS_CreateRevocableEK

Table 1. Invariant fields from tpmAbsState.

In addition to defining and verifying postconditions of each TPM command, we also verify that properties that we want to remain invariant over command execution. Invariants in the model take two forms – those that are explicitly defined and those that are captured in the abstract state type definitions. As was previously mentioned, the only way to change a PCR value is by rebooting the platform or using the TPM_Extend command. We can prove that this property holds in our model. With the following theorem, we show that along with ABS_Extend, the startup (after reboot) commands – ABS_Startup and ABS_Init, ABS_sinit and ABS_senter – are the only commands that change the state field pcrs:

```
pcrs_unchanged: THEOREM
 FORALL (s:tpmAbsState,c:tpmAbsInput) :
  not(ABS_Startup?(c) OR ABS_Init?(c) OR
      ABS_senter?(c) OR ABS_sinit?(c) OR
      ABS_Extend?(c)) =>
  pcrs(s) = pcrs(executeCom(s,c));
```

Note that while postconditions are associated with individual commands, invariants are typically proven over all commands simultaneously using the induction principle associated with the tpmAbsInput structure. The previous invariant is an example of one such theorem – note the universally quantified variable c : tpmAbsInput in the theorem signature.

The ABS_Startup and ABS_Init commands set up standard initial states following the startup command and hardware initialization, respectively. They reset all fields within tpmAbsState and are exceptions to most invariants. A list of invariants and the commands that modify them are shown in Table 1.

```
make_and_activate_identity: THEOREM
 FORALL (state:(afterStartup?),caDigest:(tpmDigest?),aikParams:(tpmKey?)):
  LET (a,s)=runState(
            TPM_MakeIdentity(caDigest,aikParams)
            >>= CPU_saveOutput(0)
            >>= (LAMBDA (a:tpmAbsOutput) :
                 CASES a OF
                   OUT_MakeIdentity(aik,idBind,m) : CA_certify(aik,idBind)
                   ELSE TPM_Noop(a)
                 ENDCASES)
            >>= CPU_saveOutput(1)
            >>= (LAMBDA (a:tpmAbsOutput) :
                 CASES a OF
                   OUT_Certify(data,m) : TPM_ActivateIdentity(aikParams,data)
                   ELSE TPM_Noop(a)
                 ENDCASES))
                  (state) IN
     identity?(keyUsage(aikParams)) AND not(migratable(keyFlags(aikParams)))
      AND private(aikParams)=key(idKey(memory(s)(0)))
      AND caDigest=idBinding(memory(s)(0)) =>
     a=OUT_ActivateIdentity(sessK(data(memory(s)(1))),TPM_SUCCESS)
      AND s=state WITH [`keyGenCnt:=keyGenCnt(state)+2]
```

Fig. 7. Protocol used to verify AIK support.

Possible invariants on the abstract state are captured in the subtype defined by the `wellFormed?` predicate. Specifically, the definition of instruction execution maps a state of type `(wellFormed?)` to another state of type `(wellFormed?)`. Conditions in the `wellFormed?` predicate include basic structural properties such as the integrity of data for restoring TPM state that will automatically be checked during type checking.

Verifying protocols involves using the state monad to sequence command execution to perform more complex tasks. Before a quote can be generated, the TPM internally creates an AIK. The public AIK is certified by a trusted Certificate Authority (CA) [16] . The protocol for generating and certifying this AIK is shown in figure 7. The function `runState` runs the monad by calling it on the initial state.

The use of bind (`>>=`) and lambda constructs allows one instruction to consume the output of the previous instruction. For example, `CA_Certify` uses the output of `TPM_MakeIdentity` after it is stored in memory for later use. The use of `CASE` constructs accounts for the possibility that the previous output is not of the correct type. We are working on mechanisms for eliminating this, thereby cleaning up the protocol representation.

The conditions for proper execution of this sequence of commands involve conditions for proper execution of the commands individually. For example, notice the conditions that the key be non-migratable and an identity key were

previously seen when discussing the verification of the single `TPM_MakeIdentity` command. The additional conditions in the antecedent are necessary to verify the `memory` was stored correctly within the `tpmAbsState`. In the consequent, we ensure that the output bound to `a` and the state bound to `s` correspond with the postconditions of `TPM_ActivateIdentity`, since it is the last command in the sequence. However, in doing so, we know that in order for these postconditions to be met, the previous commands were correctly executed.

3.2 Verifying Privacy CA Protocol

We are now ready to put all the moving parts together and verify the Privacy CA protocol. The PVS representation of the protocol from figure 1 that generates the output in equation 1 is shown in figure 8. To verify protocol execution, we first ensure that for all inputs the output bound by the `LET` form to `a` is the quote defined in equation 1 and that the state bound to `s` is the correct state following execution. This tells us the protocol generates the right output.

A collection of additional theorems verify detection of replay attacks, spoofed quotes and nonces, and bad signatures. For example, we can show that a bad nonce indicating potential replay is detectable in the quote:

```
bad_nonce: THEOREM
 FORALL (s:tpmAbsState,k:(tpmKey?), n1,n2:(tpmNonce?), pm:PCRMASK) :
    n1/=n2 =>
    runState(TPM_Quote(k,n1,pm))(s) /=
      runState(TPM_Quote(k,n2,pm))(s);
```

Additionally, we confirm that a bad AIK results in a bad quote recognizable in the quote returned by the protocol:

```
bad_signing_key: THEOREM
 FORALL (s:(afterStartup?),n:(tpmNonce?),pm:PCRMASK,k0,k1:(tpmKey?)) :
  LET (a0,s0)=runState(TPM_Quote(k0,n,pm))(s),
      (a1,s1)=runState(TPM_Quote(k1,n,pm))(s) IN
    private(k0)/=private(k1) =>
      a0/=a1;
```

These and similarly formed theorems verify that: (i) bad nonces, AIK signatures and PCR values are detectable; (ii) PCRs record measurement order as well as values; and (iii) `senter` was called to initiate the secure session. These are not properties of individual commands, but of the protocol run's output.

4 Related Work

Most verification work involving the TPM examines systems that use the TPM API [12, 6], not the command set itself. Noteworthy exceptions are works by Delaune et. al. [8, 7] and Gürgens et. al. [10]. Delaune's work examines properties

```
cert_and_quote_with_prev_key : THEOREM
 FORALL (state:(afterStartup?),n:(tpmNonce?),pm:PCRMASK,idKey:(tpmKey?),
         caDig:(tpmDigest?)) :
  LET (a,s)=runState(
                 TPM_MakeIdentity(caDig,idKey)
                 >>= CPU_saveOutput(0)
                 >>= (LAMBDA (a:tpmAbsOutput) :
                      CASES a OF
                        OUT_MakeIdentity(aik,idBind,m) :
                           CA_certify(aik,idBind)
                        ELSE TPM_Noop(a)
                      ENDCASES)
                 >>= (LAMBDA (a:tpmAbsOutput) :
                      CASES a OF
                        OUT_Certify(data,m) :
                           TPM_ActivateIdentity(idKey,data)
                        ELSE TPM_Noop(a)
                      ENDCASES)
                 >> CPU_read(0)
                 >>= (LAMBDA (a:tpmAbsOutput) :
                      CASES a OF
                        OUT_MakeIdentity(aik,idBind,m) :
                           TPM_Quote(aik,n,pm)
                        ELSE TPM_Noop(a)
                      ENDCASES)
                 >>= CPU_saveOutput(2)
                 >> CPU_BuildQuoteFromMem(2,0))
                 (state) IN
            identity?(keyUsage(aikParams))
              AND not(migratable(keyFlags(aikParams)))
              AND OUT_MakeIdentity?(memory(s)(0))
              AND OUT_Quote?(memory(s)(2))
              AND private(idKey)=key(idKey(memory(s)(0)))
              AND caDig=idBinding(memory(s)(0)) =>
            LET pcrs=getPCRs(s`pcrs,pm) IN
              a=OUT_FullQuote(tpmQuote(pcrs,n,signed(private(idKey),clear)),
                    tpmIdContents(caDig,idKey,signed(private(idKey),clear)),
                    CPU_SUCCESS);
```

Fig. 8. Protocol used to generate full quote for an external appraiser.

of functions performed within the TPM using ProVerif for their analysis. While we are attempting to develop an abstract requirements model for the TPM, they focus on verifying cryptographic properties of TPM functions. Their work deals with verifying authentication [8] where they examine a command subset responsible for authentication. Two major differences are their inclusion of session management commands and their decision not to explicitly model state change. We have chosen to defer session management thus far and explicitly model state change using the state monad described earlier. In their analysis of Microsoft Bitlocker and the envelope protocol [7], they include an attacker while we are looking at functional correctness. These distinctions aside, the abstractions they choose are quite similar to ours even though we are working in higher-order logic in contrast to their use of horn clauses. This is encouraging and suggests that developing a common TPM requirements model may be feasible. It is also worth mentioning here that Ryan's unpublished work [15] is an excellent general introduction to the TPM and its use.

Gürgens and colleagues [10] develop a TPM model using asynchronous product automata (APA) and analyze models using the SH-Verification Tool (SHVT). Their work shares several protocols of interest with ours – secure boot, secure storage, remote attestation, and data migration – with only remote attestation being described in detail. Like our work they analyze interaction with a Privacy CA, but unlike our work and similar to Delaune, Gürgens includes various kinds of attackers in examining the protocol. Considering multiple attackers with multiple intents is the most interesting contribution of this work. By using a automata model, Gürgens also models state transition explicitly as we do, in contrast with Delaune.

5 Conclusions and Future Work

We have successfully verified about 40% of the TPM command set and the CA Protocol using TPM commands. As the TPM currently has no other formal verification, this is an important step to ensuring the validity of the TPM and its commands. Our CA Protocol steps through the role of the TPM in remote attestation and proves that the commands return what they are intended to return. Additional theorems verify invariants, postconditions, and detectability of various attacks. All models defined in this paper are available through the authors.

Immediate plans are continuing to specify the abstract TPM model while starting on the concrete model and bisimulation specification. In the abstract model, we are focusing now on data migration among TPMs and on direct anonymous attestation (DAA) [2] protocols while continuing to verify the full TPM command set. We also plan to extend our work to include virtual TPMs.

References

1. —: TCG TPM Specification. Trusted Computing Group, 3885 SW 153rd

Drive, Beaverton, OR 97006, version 1.2 revision 103 edn. (July 2007), https://www.trustedcomputinggroup.org/resources/tpm_main_specification/
2. Brickell, E., Camenisch, J., Chen, L.: Direct anonymous attestation. In: Proceedings of the 11th ACM conference on Computer and communications security. pp. 132–145. ACM (2004)
3. Challener, D., Yoder, K., Catherman, R., Stafford, D., Doorn, L.V.: A Practical Guide to Trusted Computing. IBM Press (2007)
4. Coker, G., Guttman, J., Loscocco, P., Herzog, A., Millen, J., O'Hanlon, B., Ramsdell, J., Segall, A., Sheehy, J., Sniffen, B.: Principles of remote attestation. International Journal of Information Security 10(2), 63–81 (June 2011)
5. Coker, G.S., Guttman, J.D., Loscocco, P.A., Sheehy, J., Sniffen, B.T.: Attestation: Evidence and trust. In: Proceedings of the International Conference on Information and Communications Security. vol. LNCS 5308 (2008)
6. Datta, A., Franklin, J., Garg, D., Kaynar, D.: A logic of secure systems and its application to trusted computing. In: Security and Privacy, 2009 30th IEEE Symposium on. pp. 221–236. IEEE (2009)
7. Delaune, S., Kremer, S., Ryan, M., Steel, G.: Formal analysis of protocols based on tpm state registers. In: Proceedings of the 24th IEEE Computer Security Foundations Workshop (CSF 2011). pp. 66–82 (2011)
8. Delaune, S., Kremer, S., Ryan, M.D., Steel, G.: A formal analysis of authentication in the tpm. In: Proceedings of the Seventh International Workshop on Formal Aspects in Security and Trust (FAST'10). Springer (2010)
9. Goldreich, O., Oren, Y.: Definitions and properties of zero-knowledge proof systems. Journal of Cryptology 7, 1–32 (1994), http://dx.doi.org/10.1007/BF00195207, 10.1007/BF00195207
10. Gürgens, S., Rudolph, C., Scheuermann, D., Atts, M., Plaga, R.: Security evaluation of scenarios based on the tcgs tpm specification. Computer Security–ESORICS 2007 pp. 438–453 (2007)
11. Haldar, V., Chandra, D., Franz, M.: Semantic remote attestation – a virtual machine directed approach to trusted computing. In: Proceedings of the Third Virtual Machine Research and Technology Symposium. San Jose, CA (May 2004)
12. Lin, A.H.: Automated analysis of security APIs. Ph.D. thesis, Massachusetts Institute of Technology (2005)
13. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991), citeseer.nj.nec.com/moggi89notions.html
14. Owre, S., Rushby, J., Shankar, N.: PVS: A Prototype Verification System. In: Kapur, D. (ed.) Proc. of 11th International Conference on Automated Deduction. Lecture Notes in Artificial Intelligence, vol. 607, pp. 748–752. Springer–Verlag, Saratoga, NY (June 1992)
15. Ryan, M.: Introduction to the tpm 1.2 (March 2009), ftp://ftp.cs.bham.ac.uk/pub/authors/M.D.Ryan/08-intro-TPM.pdf, draft Report
16. Sailer, R., Zhang, X., Jaeger, T., van Doorn, L.: Design and implementatation of a tcg-based integrity measurement architecture. In: Proceedings of the 13th USENIX Security Symposium. USENIX Association, Berkeley, CA (2004)
17. Sangiorgi, D.: Introduction to Bisimulation and Coinduction. Cambridge University Press (2012)
18. Wadler, P.: The essence of functional programming. In: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 1–14. Albequerque, New Mexico (1992), citeseer.nj.nec.com/wadler92essence.html