Verifying TPM Protocols Using a State Monad*

Brigid Halling and Perry Alexander

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

Abstract. This work presents continuing results of an effort to verify trusted computing protocols using a state monad. The protocols addressed originate in work verifying the Trusted Platform Module (TPM) that serves as the hardware root-of-trust for storage and reporting in building a trusted computing base. Using PVS, we specify TPM commands as state transformations and use a state monad to sequence them to represent protocols. Protocols presented here include the Privacy CA Attestation Protocol and two variants of a Migration Protocol. We verify correctness of each protocol and demonstrate the impacts of weakening the CA Protocol on security properties. All protocols are specified and verified automatically using PVS decision procedures and rewriting system.

1 Introduction

The Trusted Platform Module (TPM) is essential for trusted computing serving as both the root of trust for storage and reporting [3]. As such, it warrants formal verification to ensure correct behavior of both the device and protocols that use it. In our efforts to formalize and verify TPM-centered protocols, we have adopted the use of a state monad to sequence command execution and model state. Using PVS [10], we developed a general purpose TPM model and verified a monadic model of a Privacy CA centered remote attestation protocol [6]. Here we report on generalizing the approach to examine other trusted computing protocols including a weakened remote attestation protocol and migration protocols for PVS data.

Using a state monad to model command execution, we have previously verified the correctness of a remote attestation protocol known as the CA Protocol [6]. This protocol allows an external appraiser to assess system state by communicating with its TPM. In examining the protocol, questions arise concerning its complexity. To justify design decisions we have weakened the CA protocol to demonstrate necessity of protocol elements. We use this weakened model to analyze an attack on the protocol demonstrating the impacts of one TPM falsely assuming the identity of another.

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

As a root of trust for storage and reporting, the TPM must protect its data. However, at times it becomes necessary to migrate data—primarily keys—from one TPM to another for backup, data migration or key sharing within a network. To do this, the TPM uses a *migration protocol* to guarantee confidentiality and integrity. Because data is potentially exposed during migration, protocol verification is critical. From a verification perspective, modeling such a protocol is interesting due to the need to model two TPMs involved in a single protocol.

2 Background

The Trusted Platform Module (TPM) [12] is a secure cryptoprocessor at the heart of establishing and maintaining a trusted computing infrastructure [3]. The main capabilities of the TPM are threefold: establishing and protecting a main identifier; storing and securely reporting system measurements; and binding secrets to a specific platform.

The *endorsement key* (EK) is an asymmetric key that serves as the unique identifier of the TPM. The EK^{-1} is maintained confidentially, while EK serves as the root-of-trust for reporting. The *storage root key* (SRK), also an asymmetric key, provides a root key for chaining *wrapped keys*—asymmetric keys whose private key is encrypted by another asymmetric key. Wrapped keys can be stored outside of the TPM (through the process of key migration), but may only be used if its wrapping key is also installed. The use of the SRK as the root of these chains of keys then binds the keys to its associated TPM.

The TPM uses special purpose registers for storing and extending hashes called *platform configuration registers* (PCRs) to store configuration information. A PCR records hashes of a platform's components during boot or at runtime in a manner preserving value and order. Each component is hashed as it is started and that hash used to extend the hash presenting boot thus far. The integrity of PCRs is ensured by controlling access through locality and the TPM software stack. A quote mechanism providing cryptographic evidence of integrity delivers PCRs to an external appraiser via an attestation process.

2.1 The CA-Based Remote Attestation Protocol

Among the most important functions of the TPM is reporting system measurements via attestation. The TPM does this through the use of Remote Attestation where at the request of an external appraiser, the TPM gathers PCRs and delivers them in a trusted fashion [5]. The appraiser then determines whether or not it trusts the received PCR values and subsequently measured the system based on PCR contents. To achieve this while preserving the identity of the TPM, a trusted Privacy Certificate Authority (CA or Privacy CA) is used. The CA verifies that an attestation identity key (AIK) belongs to a specific TPM. Figure 1 shows the Privacy CA-based attestation protocol.

The TPM command `TPM_MakeIdentity` creates an AIK and wraps it with the SRK, allowing it to be used only by the TPM that created it. `TPM_MakeIdentity`

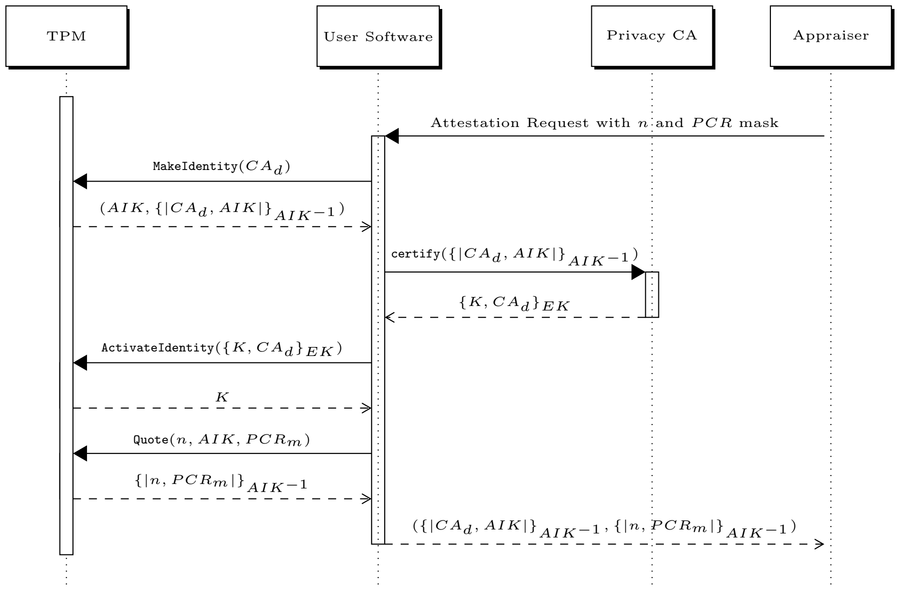
TPM
User Software
Privacy CA
Appraiser
Attestation Request with n and PCR mask
MakeIdentity(CA_d)
$(AIK, \{|CA_d, AIK|\}_{AIK^{-1}})$
certify($\{|CA_d, AIK|\}_{AIK^{-1}}$)
$\{K, CA_d\}_{EK}$
ActivateIdentity($\{K, CA_d\}_{EK}$)
K
Quote(n, AIK, PCR_m)
$\{|n, PCR_m|\}_{AIK^{-1}}$
$(\{|CA_d, AIK|\}_{AIK^{-1}}, \{|n, PCR_m|\}_{AIK^{-1}})$

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

also outputs a CA label digest that identifies the CA certifying the AIK and an AIK signed with AIK^{-1}. Since this TPM is the only entity that can sign with the AIK, this signature ensures the AIK came from the right TPM [12].

Our focus is the TPM, but for this protocol we most model the behavior of the Privacy CA, albeit abstractly. The command `CA_certify` models the interaction between the CA and the system where the TPM resides. The Privacy CA provides a session key (K) encrypted by the public EK associated with the TPM that it believes has requested the certificate. This allows only that TPM to decrypt the new session key.

`TPM_ActivateIdentity` ensures the Privacy CA is communicating with the correct TPM by attempting to decrypt K with the EK^{-1} of the TPM [12]. The resulting symmetric key is used by the CA to encrypt the identity credential of the TPM. Later we weaken this protocol deliver the new session key without encrypting in response to questions from a team attempting to synthesize the TPM. We demonstrate formally how a session key for one TPM cannot be used by another and that removing encryption removes this protection.

We next create a quote—signing PCR values and a nonce with the AIK—using the command `TPM_Quote}` [12]. This quote can then be returned to the user who may send it on to the appraiser where it will check the PCRs to know whether or not to trust the system.

2.2 Migration Protocols

The TPM must protect its identity and the data it stores from unauthorized access and modification. Situations arise where it becomes useful for keys to be exported out of one TPM and imported into another using a *migration protocol*. Migration is critical for TPM usage, but represents potential source of data leakage. If data outside the TPM is not appropriately protected and authenticated, migration can become a liability.

The two main TPM keys, EK and SRK, are tied to a specific TPM when created. These keys are non-migratable—they are unique to the TPM and cannot be migrated or exported from the TPM. Migratable keys, on the other hand, are not bound to any specific TPM, and can be used or even created outside of a TPM. With appropriate authorization, these keys can be moved between TPMs [7]. A third type of key, Certified Migratable Keys were developed as an upgrade to migratable keys, but due to the complexity that these keys and their related commands add, we will focus here on migratable keys.

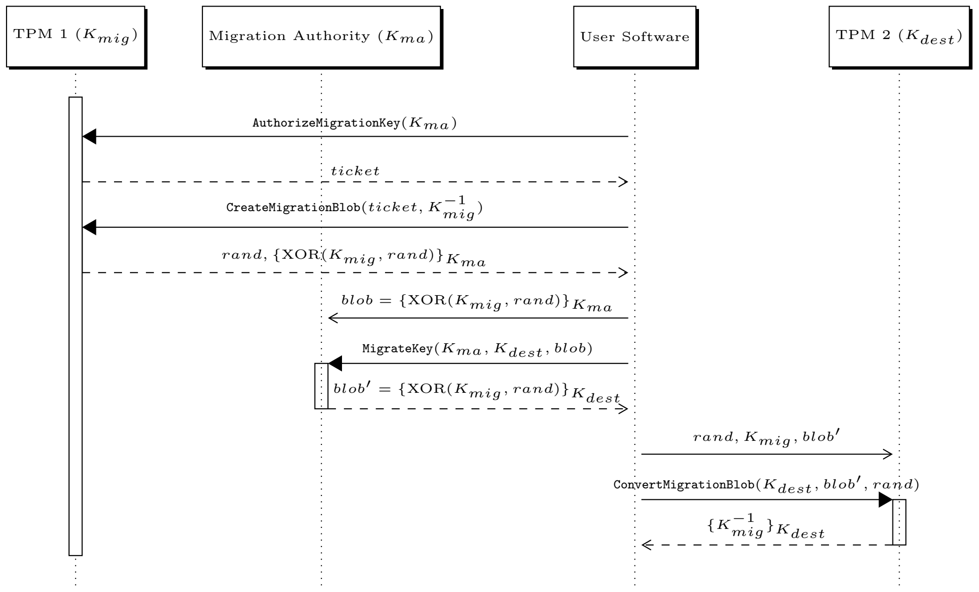
TPM 1 (K_{mig})
Migration Authority (K_{ma})
User Software
TPM 2 (K_{dest})
AuthorizeMigrationKey(K_{ma})
$ticket$
CreateMigrationBlob($ticket$, K_{mig}^{-1})
$rand$, $\{\mathrm{XOR}(K_{mig}, rand)\}_{K_{ma}}$
$blob = \{\mathrm{XOR}(K_{mig}, rand)\}_{K_{ma}}$
MigrateKey(K_{ma}, K_{dest}, $blob$)
$blob' = \{\mathrm{XOR}(K_{mig}, rand)\}_{K_{dest}}$
$rand$, K_{mig}, $blob'$
ConvertMigrationBlob(K_{dest}, $blob'$, $rand$)
$\{K_{mig}^{-1}\}_{K_{dest}}$

Fig. 2. Sequence diagram for the Migration Protocol.

Figure 2 shows the sequence of events that define the Migration Protocol. In this diagram, TPM 1 is the source of the key to be migrated (K_{mig}) and TPM 2 is the destination of the key. The use of a middleman, the Migration Authority (MA), is required to coordinate the migration, as K_{mig} temporarily passes through. The MA is authorized using the command `TPM_AuthorizeMigrationKey` that returns a migration ticket that specifies the MA. This ticket is used in the creation of a migration blob with the command `TPM_CreateMigrationBlob`. `TPM_CreateMigrationBlob` creates a blob (*blob*) and a random number (*rand*). *blob* consists of an XOR encryption of the key being migrated and *rand* encrypted by the key of the MA. With the command `TPM_MigrateKey`, the MA rewraps *blob* with K_{dest}, forming *blob'* and allowing only TPM 2 access. *rand* is passed directly to TPM 2 to prevent the MA from accessing K_{mig}. TPM 2 can now install the *blob* using `TPM_ConvertMigrationBlob`, that performs the necessary decryptions and essentially results in K_{mig} wrapped with K_{dest}, which can then be loaded into TPM 2 [11, 12].

A similar but simpler and less secure protocol that cuts out the MA portion of the protocol may be used. This directly authorizes the use of K_{dest} as the migration key and requires no random number, allowing the protocol to be more of a direct rewrapping of the K_{mig} using K_{dest}. Therefore, the blob output by `TPM_CreateMigrationBlob` can be sent directly to TPM 2, and loaded without needing `TPM_ConvertMigrationBlob`. This protocol is much more efficient, and therefore is preferred. If, however, a third party is facilitating the migration, the MA should be used [12, 8].

3 Modeling TPM Protocols

We define an abstract model of the TPM as a transition system by defining state transitions and outputs over an abstract state for each command. We can sequence command execution using the state monad. Commands define single state transformations while command sequences thread state between transformations. The model allows us to explore both the effects of single commands and the cumulative effects of protocol execution.

3.1 System State

To model TPM data, we define a PVS data type called `tpmData` whose constructors closely resemble the data definitions specified in the TPM standard. We avoid the complexity of bit-level representations specified capturing the essence of TPM functionality.

We define a PVS data type called `tpmAbsInput` to represent the abstract syntax of the TPM command set. Each TPM command specified in the TPM standard (`TPM_`*Command*) has an associated constructor in the `tpmAbsInput` data structure, giving an induction principle for the command set which is automatically usable by PVS to quantify over all possible TPM inputs. Similarly, we define the data type `tpmAbsOutput`, which includes the correct parameters as well as a return code that is returned by each TPM command as output.

The return codes of `tpmAbsOutput` indicate either a non-fatal error has been found or the command has been executed successfully. A return code indicating success, however, does not necessarily imply that good data has been output. For example, in the case of encrypting and decrypting data, since data is handled at the bit level within the TPM, it is difficult to know if a decryption fails by looking at that data. Therefore, the command will still run successfully, but it will be using bad data. Therefore, if we know a decryption failure has occurred, we return `badData` (of type `tpmData`). This will become important as we discuss the weakening of the CA Protocol in Section 4.1.

```
tpmAbsState : TYPE =
        [# srk:(tpmKey?), ek:(tpmKey?), keys:KEYSET
         , pcrs:PCRVALUES, locality:LOCALITY, permFlags:PermFlags
         , permData:PermData, stanyFlags:StanyFlags, stanyData:StanyData
         , stclearFlags:StclearFlags, stclearData:StclearData
         , memory:mem, outData:set[tpmData], keyGenCnt:K
         , randCnt:K, restore:restoreStateData #];
```

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

Figure 3 shows the PVS record structure, `tpmAbsState`, used to represent an abstraction of the internal state of the TPM, the memory associated with the system where it is running, and record of previously output data. The asymmetric keys EK and SRK are easily recognizable within `tpmAbsState`. The `pcrs` field represents the array of hash sequences that define the value of a PCR. A list of all loaded keys is stored in the field `keys`. This loaded keys list is important to the Migration Protocol, as it provides an easy mechanism to ensure that a key has been migrated from one TPM to another. `keys` is a list of both the private key values (`vals`) of the loaded keys for easy identification and a list of the entire keys (`set`), that includes numerous fields related to the key.

The fields `memory` and `outData` are not part of the TPM itself, but represent the state of its associated system. `memory` represents system memory where TPM inputs and outputs are stored. `outData` plays a major part of the weakened CA Protocol, discussed in Section 4.1 by capturing all output generated by a protocol. Note that `outData` is not a part of either the TPM or its associated system and use used strictly for verification.

3.2 Command Execution

To specify TPM command execution, we define state transition and output functions as defined for a classical transition system. The `outputCom` function produces system output and is defined as a mapping from state and input to an output. Similarly, the `executeCom` function is the next-state function and is defined as a mapping from state and input to state. Given a state, s and input command, c, the output, state pair resulting from executing the command c in state s is defined as the pair $(\texttt{outputCom}(s,c), \texttt{executeCom}(s,c))$. Refer to earlier work [6] for the details of how individual TPM command execution is modeled.

3.3 Modeling State Transitions

Monads [9, 13] are well known in functional programming where they are used for implementing errors, state, and concurrency among many other computation elements. In our TPM model, command sequencing is modeled using a state monad to thread states through a sequence of TPM commands using a classic state monad. In effect, the state monad simulates the behavior of memory in the TPM and its associated system.

All monads are characterized by a type and the functions *return* and *bind*. Return lifts a data element into the monad, and bind is similar to a let expression [9, 13] that performs an operation and uses its result as input to a subsequent operation. In our monad, the type is `tpmAbsState`, the calculations performed are command executions, and bind is used to pass the output of a command to the next command in sequence. In our monad, the type is `tpmAbsState`, the calculations performed are command executions, and bind is used to pass the output of a command to the next command in sequence.

The rationale for using a monad is that a TPM executes commands in sequence much like assembly commands in a conventional microprocessor. The output state from one command is the input state to the next command. Modeling the command sequence `i;i'` conventionally using a `LET` form would look like the following:

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

where `(o',s')` is the output, state pair of running command `i`. We can see that the input state while running `i'` is `s'`, the state returned from running `i`. However, writing long sequences of let forms is impractical for our purposes.

The bind function of the monad is an encapsulation of this let form. It eliminates the necessity of manually passing the resulting state as input, passing state in the background. The result of using this bind is a pattern of modeling and execution that looks very similar to the execution pattern of TPM commands.

To define a state monad for TPM command execution, we first define a simple data type, `State`, having a single field called `state` that holds a function from an abstract state to an abstract output, abstract state pair:

```
State : DATATYPE
BEGIN
  state(runState:[tpmAbsState->[tpmAbsOutput,tpmAbsState]]):state?
END State;
```

Given s, a value of type `tpmAbsState`, and m, a `State`, the application `runState(`m`)(`s`)` will result in a `tpmAbsOutput`, `tpmAbsState` pair. This is precisely the output expected. Note that the use of `State` and `state` in this definition is somewhat misleading. Neither is actually a state, but a state monad that given a state will generate a new state. The data type should be viewed as a kind of state generation or next-state function, not a single state.

We now define `return`, *bind* (>>=) and sequence (>>). Sequence is a special case of bind commonly defined in most state monad implementations where the output of the previous execution is dropped. First we define `return` whose form is:

```
  return(x:tpmAbsOutput):State = state(LAMBDA (s:tpmAbsState) : (x,s));
```

`return` lifts a member of `tpmAbsState` into `State`—given a `tpmAbsState`, it returns a `State` that when run produces the original `tpmAbsState`.

`runState` is a function from `tpmAbsState` to a `tpmAbsOutput`, `tpmAbsState` pair. Clearly, the state part of the output should be the state lifted by `return`. But what about the output? If we are lifting `tpmAbsState` there is no way to extract an output. The `bind` function handles this by simply requiring an output be specified as a parameter.

The second function defined is bind, typically represented by the infix operator >>=. The bind operation takes a monad and a function from `tpmAbsOutput` to a monad and produces a new monad. The signature is of the form:

```
  m:State >>= f:[A->State] : State
```

Unfortunately, the signature alone provides little insight into the actual function of bind. The implementation is:

```
 >>= (m:State,f:[A->State]):State =
   state(LAMBDA(s0:S):
     LET (a,s1) = runState(m)(s0) IN
       runState(f(a))(s1));
```

The biggest clue to the behavior of bind comes in the `LET` form where `(a,s1)` is bound to running `m`—the first argument to bind—on state `s0`. `a` is bound to the output and `s1` to the state resulting from running `m` on `s0`. If we were doing this outside the monad, we would refer to this as the intermediate state between the two executions.

The result of the bind is `runState(f(a))(s1)`. First consider `runState(f(a))`. Looking at the signature, `f(a)` is a mapping from something of type `A` to a monad of type `State`. What we get is a next state monad with the previous output bound to `a`—the previous output is available to the calculation of the next state. `runState` pulls the state function out of the new state monad and evaluates that function with `s1`, the intermediate state. So, the state is threaded through the evaluation with the user providing only `s0`, the initial state. Always remember that bind does not produce a state. Instead it produces a state monad that given a state will produce an output, state pair.

Another common operation is a specialization of bind called sequence, represented by the infix operator >>. It is similar to bind, sequencing operations, but it drops the output of previous command execution and simply passes state:

```
 >> (m:State,f:State):State =
   state(LAMBDA(s0:S):
     LET (a,s1) = runState(m)(s0) IN
       runState(f)(s1));
```

3.4 Memory and the Monad

The memory field of tpmAbsState is not a part of the TPM itself, but represents memory use by the TPM's environment for storing values. For example, the command TPM_MakeIdentity generates a new identity key, called *aik* that the Privacy CA and the TPM_Quote command use. Commands such as CA_certify, TPM_ActivateIdentity, and TPM_Quote require *aik* as input. Furthermore, we require the *aik* to verify command preconditions.

The model fragment below shows a portion of the CA Protocol model where the TPM_MakeIdentity command is followed by the CA_certify command. The output from TPM_MakeIdentity can easily be passed to CA_certify using bind (>>=). We can see the outputs aik and idBind from TPM_MakeIdentity used as inputs to CA_certify.

```
TPM_MakeIdentity(e,d,k)
>>= (LAMBDA (a:tpmAbsOutput) :
     CASES a OF
       OUT_MakeIdentity(aik,idBind,m) : CA_certify(aik,idBind)
       ELSE TPM_Noop(a)
     ENDCASES)
```

However, to use that same data in a subsequent command that is not run sequentially with TPM_MakeIdentity requires storing data. We use memory and the CPU_saveOutput and CPU_read commands. These are not TPM commands, but simulate the behavior of the system associated with the TPM. The field memory is an array mapping a natural number to a tpmAbsOutput. The output of a command (of type tpmAbsOutput) can be saved using CPU_saveOutput, and later read using the CPU_read. The memory field allows any number of outputs to be saved to the tpmAbsState, and therefore to be passed from command to command.

To add a TPM_Quote to the end of our previous example, we must employ the use of memory. To be able to use the output of TPM_MakeIdentity, we save it into memory at location 1 using CPU_saveOutput(1). Since we are using bind to sequence these commands, we are still passing along the output of TPM_MakeIdentity, and therefore do not need to read it in before calling CA_certify. However, after running CA_certify, we have a new output and must call CPU_read(1) to retrieve the output of TPM_MakeIdentity so we can use that output as the input for TPM_Quote.

```
TPM_MakeIdentity(e,d,k)
>>= CPU_saveOutput(1)
>>= (LAMBDA (a:tpmAbsOutput) :
     CASES a OF
       OUT_MakeIdentity(aik,idBind,m) : CA_certify(aik,idBind)
       ELSE TPM_Noop(a)
     ENDCASES)
>> CPU_read(1)
>>= (LAMBDA (a:tpmAbsOutput) :
     CASES a OF
       OUT_MakeIdentity(aik,idBind,m) : TPM_Quote(aik,n,p)
       ELSE TPM_Noop(a)
     ENDCASES)
```

Since a command like `CA_certify` has preconditions that require knowing exactly what is used as input, we also use saved memory as a means to retrieving that input data. This is shown in the following model fragment where `s` represents the final state of the protocol, we ensure that no other output has rewritten the saved output of `TPM_MakeIdentity`, and the function `certify?` checks the preconditions of the `CA_certify` command:

```
  OUT_MakeIdentity?(s‘memory(1)) AND
  certify?(idKey(s‘memory(1)),idBinding(s‘memory(1)))
```

```
CA_unsigned_outData : THEOREM
FORALL (state:(afterStartup?), e:(tpmEncAuth?), cad:(tpmDigest?),
        aik:(tpmKey?), n:(tpmNonce?), p:PCR_SELECTION, w,x,y,z:nat) :
  private(badEK)/=private(goodEK)
  AND w/=x AND w/=y AND w/=z AND x/=y AND x/=z AND y/=z  =>
  LET state1 = state WITH ['ek:=badEK] IN
    LET (a,s) = runState(TPM_MakeIdentity_noSign(e,cad,aik)
     >>= CPU_saveOutput(x)
     >>= (LAMBDA (a:tpmAbsOutput) :
          CASES a OF
            OUT_MakeIdentity(aik,idBind,m) :
              CA_certify_noSign(aik,signData(idBind))
            ELSE TPM_Noop(a)
          ENDCASES)
     >>= CPU_saveOutput(y)
     >>= (LAMBDA (a:tpmAbsOutput) :
          CASES a OF
            OUT_Certify(aik,data,m) : TPM_ActivateIdentity(aik,data)
            ELSE TPM_Noop(a)
          ENDCASES)
     >>= CPU_saveOutput(w)
     )(state1) IN
  member(private(state1'ek),vals(state1'keys))
  AND member(private(k),vals(state1'keys))
  AND wellFormedRestore?(s'restore) AND makeIdentity?(state1,k)  =>
  OUT_MakeIdentity?(s'memory(x)) AND
  LET key=idKey(s'memory(x)) IN
    certify_noSign?(key,signData(idBinding(s'memory(x)))) =>
  s'memory(y)=OUT_Certify(key,
           encrypted(tpmAsymCAContents(tpmSessKey(initSessKeyVal),
                           digest(signData(idBinding(s'memory(x))))),
                     key(goodEK)),CPU_SUCCESS) AND
  LET encr=dat(s'memory(y)) IN activateIdentity?(s,key,encr) =>
    OUT_ActivateIdentity?(s'memory(w))
    AND (a=OUT_ActivateIdentity(sKey(blob(badData)),TPM_SUCCESS)
         OR a=OUT_ActivateIdentity(sessK(badData),TPM_SUCCESS)
         OR a=OUT_ActivateIdentity(badData,TPM_SUCCESS))
    AND s'outData=add(symmKey(s'memory(w)),
                    add(key,add(signData(idBinding(s'memory(x))),
                    outData(state1))));
```

Fig. 4. Weakened CA Protocol.

4 Verifying Protocols

Two protocol analysis problems arose from our initial verification of the original Privacy CA protocol requiring us to generalize the monad-based technique. The original verification established that the Privacy CA protocol produces the correct result while ensuring trustworthiness. We were then asked to examine the impact of an attack on the original protocol and to perform similar analysis on TPM data migration protocols.

Figures 4 and 5 share a common structure that appears repeatedly in this work. A `LET` binding captures protocol execution and its associated body expresses the theorem to be verified. In Figure 4 commands comprising the protocol are in boldface. Commands are sequenced in the `LET` binding with bind and sequence operators. `CASES` account for erroneous output by skipping commands that will not execute. Following protocol evaluation, state and other data bound to `LET` variables becomes the subject of a correctness theorem forming the `LET` body. Correctness theorems take many forms, but most either compare an execution result with a golden value or check properties of the state. In Figure 5 important clauses defining correctness are in boldface.

4.1 CA Protocol with Weakening

In our main CA Protocol verification using PVS [6], we execute the protocol shown in Figure 1 using an EK that is assumed to be good and that all of the necessary preconditions are true. We are then able to verify the resulting output is as expected and that the final state includes all necessary updates. Such a verification is useful to us to show that our method of modeling the protocol was valid. However, the pragmatic impact of such a proof is minimal—it is simply proving the correct output results from an unimpeded run. In this effort we demonstrate resilience to a spoofing attack where one TPM system attempts to assume the identity of another.

To introduce an attacker to this model, we introduce TPM A whose EK we'll refer to as `badEK` masquerading as another TPM, TPM B with EK `goodEK`. TPM A will contact the Privacy CA as if it were TPM B. Normally, the Privacy CA will authenticate a TPM using the signature of the data it is sent. We will weaken this protocol and attempt to send unsigned data from TPM A to the Privacy CA and identify it as being from TPM B. Therefore, the CA will return the requested data to the user encrypted with `goodEK` ($\{K, CA_d\}_{goodEK}$). In Figure 4, this encryption is `data` output by `OUT_Certify`.

`TPM_ActivateIdentity` attempts to proceed as normal, but should never be able to properly decrypt `data` as it does cannot have access to $goodEK^{-1}$. It is important to note that `TPM_ActivateIdentity` has still run successfully and produces an output as expected. We could continue to run the remaining commands of the protocol, and continue to successfully run.

Looking at the output of the protocol in Figure 4 with `a` as the output, and `s` as the resulting state, we can see that in all resulting cases, the first parameter of `a` contains `badData` the result of a failed decryption using an incorrect key.

Although not immediately obvious, the `symmKey(s'memory(w))` that is added to `s'outData` is this same parameter, and therefore also contains `badData`. This shows us that the CA does not need to receive signed data from the TPM in order to only allow a good TPM to be able to decrypt the key that unlocks the identity credential.

As mentioned in Section ??, `outData` is not a part of the TPM itself, but was added to aid in protocol analysis. For each TPM command run in a sequence, output data is added to the list `outData` allowing Dolev-Yao style attack models where the enemy has access to all communication [4]. At the end of protocol execution, we can then check this list for bad data or data that should not be available to an attacker. This allows us to determine if something went wrong with any part of the protocol execution, even if it completed without incident. The presence of `badData` in this `outData` list indicates that something went wrong during protocol execution.

When comparing the protocol shown in Figure 1 to the protocol shown in Figure 4 that there are four different entities in the diagram (the TPM, the user, the CA, and the appraiser), yet in our model of the same system, these entities are not distinct. The only difference we can see is the difference in the command prefixes `TPM` and `CA`. In essence, this is the only distinction necessary. There is an implied user calling the commands at the implied request of an appraiser. Since the CA and TPM are communicating with the same user, these commands are run in sequence.

By showing that running that command sequence results in bad data for every possible output, we are able to prove that a TPM attempting to identify itself with the wrong EK value results in an inability to correctly run a CA Protocol. We were able to prove this theorem using PVS decision procedures and rewriting system. However, we were not able to completely discharge all type check conditions (TCCs) associated with this proof. The unprovable TCCs all relate to `badData`, which has a type that is more generic than PVS accepts. Based on the nature of `badData`, we have found that such cases are acceptable within our model due to the inherent nature of `badData` being of an unspecific type.

4.2 Migration Protocol

The migration protocol example is interesting because it involves two TPMs communicating to move data from one to the other while maintaining confidentiality and integrity. The sequence diagram depicting the Migration Protocol seen in Figure 2 includes four distinct entities—TPM 1, TPM 2, a user, and Migration Authority (MA). However, unlike the CA Protocol, we must use more than command prefixes to distinguish between entities due to the existence of two distinct TPMs. Since the state monad threads a single state through the sequence of commands, we must have one state monad for each TPM.

Notice from the protocol modeled using PVS shown in Figure 5 the two main `LET` statements that name the outputs of running the command sequences—the pairs `(a1,s1)` and `(a2,s2)`. These output, state pairs correspond with the

```
migrate_scheme : THEOREM
  FORALL(state1,state2:(afterStartup?), dest,par,mig,ma:(tpmKey?), w,x,y:nat) :
migrateKey?(ma) AND w/=x AND w/=y AND x/=y
AND member(private(ma),vals(state1'keys)) AND member(private(mig),vals(state1'keys))
AND member(private(par),vals(state1'keys)) AND member(private(dest),vals(state2'keys))=>
LET (a1,s1) =  runState(
     TPM_AuthorizeMigrationKey(ma,migrate)
     >>= CPU_saveOutput(w)
     >>= LAMBDA (a:tpmAbsOutput) :
         CASES a OF
           OUT_AuthorizeMigrationKey(t,r) :
              TPM_CreateMigrationBlob(par,m,t,encDat(mig))
           ELSE TPM_Noop(a)
         ENDCASES
     >>= CPU_saveOutput(x)
     >>= LAMBDA (a:tpmAbsOutput) :
         CASES a OF
           OUT_CreateMigrationBlob(rand,mb,r) : TPM_MigrateKey(ma,dest,mb)
           ELSE TPM_Noop(a)
         ENDCASES
     )(state1),
    d=encData(encDat(mig)) IN
createMigBlob?(state1,par,m,autData(s1'memory(w)),encDat(mig)) =>
  a1=OUT_MigrateKey(encrypted(tpmXOR(
           OAEP(tpmMigrateAsymkey(usageAuth(d),pubDataDigest(d),privKey(d)),
                migrationAuth(d),privKey(d)),RNG(s1'randCnt-1)),key(dest))
           ,TPM_SUCCESS)
AND OUT_CreateMigrationBlob?(s1'memory(x)) AND
LET (a2,s2) = runState(
     TPM_ConvertMigrationBlob(dest,migData(a1),random(s1'memory(x)))
     >>= CPU_saveOutput(y)
     >>= LAMBDA (a:tpmAbsOutput) :
         CASES a OF
           OUT_ConvertMigrationBlob(kb,r) : TPM_LoadKey2(dest,makeKey(kb))
           ELSE TPM_Noop(a)
         ENDCASES
     )(state2) IN
convertMigBlob?(state2,dest,migData(a1),random(s1'memory(x)))
AND OUT_ConvertMigrationBlob?(s2'memory(y))
AND loadKey2?(state2,dest,makeKey(convertData(s2'memory(y)))) =>
  a2=OUT_LoadKey2(makeKey(convertData(s2'memory(y))),TPM_SUCCESS) AND
  s2=state2 WITH
        ['keys:=(#vals:=add(private(mig),vals(keys(state2)))
                 ,keys:=add(makeKey(convertData(s2'memory(y))),keys(keys(state2)))#)
        ,'memory:=s2'memory, 'outData:=s2'outData];
```

Fig. 5. Migration Protocol.

final outputs and states of TPM 1 and TPM 2 from Figure 2, respectively. The functionality of the MA is included in the TPM 1 sequence.

We pass the pieces of both the output (`a1`) and state (`s1`) of TPM 1 as input parameters to the `TPM_ConvertMigrationBlob` command of TPM 2. This process of using the output from one thing as the input to the next is extremely familiar. It is the entire reason we incorporate the state monad into our model. Therefore, it would be entirely possible to set up this protocol using nested monads. However, that adds a layer of complexity that is unnecessary for this protocol. Perhaps if we modeled a protocol that included the use of more than two TPMs this would be worth the effort. Such a scenario using the TPM, however, is not likely.

To verify the Migration Protocol we show that the same key that was initially a part of TPM 1, K_{mig}, is also loaded into TPM 2 after running the protocol (as it is part of the `keys` list within the state). Using the decision procedures and rewriting rules of PVS as well as some lemmas clarifying properties of keys, we were able to prove the theorem shown in Figure 5. This assures us that K_{mig} is a part of TPM 2. We have additionally proved other migration protocols of the TPM, such as the simpler rewrapping protocol mentioned in Section 2.2, but they are proved in the same way as the protocol we have thoroughly discussed here.

4.3 Proof Complexity

A project goal is building models that can be reused by non-experts with modest training. Thus, proof automation is important. Proofs of theorems shown are accomplished using the PVS automatic rewriting system and the `bash` strategy with some intervention where structural equivalence is applied. Run time for the large proofs documented here is approximately 2 hours on a Macintosh Pro with 8GB of memory. This is a significant improvement over our original proofs of the Privacy CA attestation protocol used `grind` that ran for multiple days.

Given the nature of our specifications, rewriting is a natural approach. We allow PVS to construct rewrite rules without user input. The use of `bash` adds simplification using BDDs and disjunctive flattening to discharge logical terms. Structural equivalence decomposes equality checks for data types into equivalence of their fields and is not included in the automated commands used.

5 Conclusions and Future Work

We have successfully demonstrated that the monadic approach used in earlier attestation protocol verification efforts generalizes to other protocols and verification goals. Specifically, we have reported on verification of properties for a simple attack, a weakened protocol, and two migration protocols. We have identified a common structure for protocols and proofs that should be applicable in our continuing efforts to verify TPM protocols.

As we continue to develop models of TPM-related protocols, our representation of the TPM continues to grow both in terms of implemented commands and detail of existing commands. As the complexity of the model grows, so does the complexity of our proofs. We see great promise in the use of the newly added `outData` construct in modeling attacks on TPM protocols as well as developing new approaches for discovering other ways to model attacks. Ongoing work is extending the current model to virtual TPMs (vTPMs) [1] and modeling direct anonymous attestation (DAA) [2] protocols.

References

1. Berger, S., Caceres, R., Goldman, K., Perez, R., Sailer, R., van Doorn, L.: vTPM: Virtualizing the Trusted Platform Module (2006), http://www.kiskeya.net/ramon/work/pubs/security06.pdf, IBM T. J. Watson Research Center, Hawthorne, NY 10532 USA
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. Dolev, D., Yao, A.: On the security of public key protocols. IEEE Transactions on Information Theory 29(2), 198 – 208 (March 1983)
5. 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
6. Halling, B., Alexander, P.: Verifying A Privacy CA Remote Attestation Protocol. In: Proceedings of NASA Formal Methods (NFM 2013). pp. 398 – 412. No. 7871 in Lecture Notes in Computer Science, Moffett Field, CA, USA (May 2013)
7. Hardjono, T., Kazmierczak, G.: Overview of the TPM key management standard (2005), https://www.trustedcomputinggroup.org/news
8. Kinney, S.L.: Trusted Platform Module Basics: Using TPM in Embedded Systems (Embedded Technology). Newnes (2006)
9. Moggi, E.: Notions of computation and monads. Information and Computation 93(1), 55–92 (1991), citeseer.nj.nec.com/moggi89notions.html
10. 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)
11. Trusted Computing Group: Interoperability Specification for Backup and Migration Services. Trusted Computing Group, version 1.0 revision 1.0 edn. (June 2005), https://www.trustedcomputinggroup.org
12. Trusted Computing Group: TCG Trusted Platform Module Specification. Trusted Computing Group, 3885 SW 153rd Drive, Beaverton, OR 97006, version 1.2 revision 103 edn. (July 2007), https://www.trustedcomputinggroup.org
13. 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