TPM Specification Design

Perry Alexander Brigid Halling

August 6, 2012

Contents

List of Figures

```
tpmAbsState : TYPE = [#
                     memory : mem,
                     postInit : bool,
                     srk : (asymKey?),
                     ek : (asymKey?),
                     keys : KEYSET,
                     pcrs : PCRS,
                     locality : LOCALITY
                     #];
```

Figure 1: Abstract TPM and system state record data structure.

List of Tables

Abstract

The abstract goes here...

1 Introduction

2 Modeling Approach

2.1 TPM Abstract State

Figure 1 is the PVS record structure used to represent the internal state of the TPM and necessary external state of the environment it is running in.

2.2 TPM Abstract Command Definitions

Figure 2 is the PVS data type used to represent the abstract syntax of the TPM command set.

Table 1 maps TPM concrete commands to their abstract PVS representations.

Table 2 maps external commands that interact with the TPM to their PVS representations.

```
tpmInput : DATATYPE
BEGIN
%% Startup commands
  ABS_Init : ABS_Init?
  ABS_Startup : ABS_Startup? % Only clear implemented
  ABS_SaveState : ABS_SaveState? % unimplemented
%% PCRs, seals and keys
  ABS_Extend(h:HV,i:PCRINDEX) : ABS_Extend?
  ABS_Unseal(s:(sealBlob?),uk:(asymKey?)) : ABS_Unseal?
  ABS_Seal(sk:(asymKey?),data:BLOB) : ABS_Seal?
  ABS_LoadKey2(lk:(wrapKey?)): ABS_LoadKey2?
  ABS_CreateWrapKey(wk,parentk:(asymKey?)): ABS_CreateWrapKey?
%% Quotes and Identities
  ABS_Quote(aik:(wrapKey?),nonce:BLOB,pm:PCRMASK) : ABS_Quote?
  ABS_MakeIdentity(naik:(asymKey?),k:(symKey?)) : ABS_MakeIdentity?
  ABS_ActivateIdentity(caik:(wrapKey?),k:(symKey?)) : ABS_ActivateIdentity?
%% Ownership management
  ABS_TakeOwnership : ABS_TakeOwnership?
  ABS_OwnerClear : ABS_OwnerClear? % unimplemented
  ABS_ForceClear : ABS_ForceClear? % unimplemented
  ABS_DisableOwnerClear : ABS_DisabelOwnerClear? % unimplemented
%% Software Commands
  ABS_senter : ABS_senter? % implemented all actions as one senter
  ABS_sinit : ABS_sinit? % partially implemented
  ABS_Save(i:nat,v:tpmAbsOutput) : ABS_Save?
  ABS_Read(i:nat) : ABS_Read?
%% CA Commands
  ABS_certify(aik:(wrapKey?),ek:(asymKey?),freshk:(symKey?)) : ABS_certify?
%% Invented, imaginary Commands
  noopCom : noopCom?
END tpmInput;
```

Figure 2: TPM command data type.

```
tpmAbsOutput : DATATYPE
BEGIN
  outNothing : outNothing?
  outError(errorVal:nat) : outError?
  outQuote(oqk:KEY,oqnon:BLOB,oqpcrs:list[PCR]) : outQuote?
  outWrapKey(owk:KEY) : outWrapKey?
  outAsymKey(oask:KEY) : outAsymKey?
  outSymKey(osk:KEY) : outSymKey?
  outBlob(obl:BLOB) : outBlob?
  outCertReq(ocertaik:(wrapKey?),ocertek:(asymKey?),ofreshk:(symKey?)) : outCertReq?
  outIdentity(oidentaik:(wrapKey?),oidentc:(outCertReq?)) : outIdentity?
  outIdentActivation(oactc:(certBlob?),osessk:(symKey?),oactek:(asymKey?)) : outIdentActivation?
  outFullQuote(ofullqc:(certBlob?),ofullqsml:SML,ofullqq:(outQuote?)) : outFullQuote?
  outPCR(pcr:PCR) : outPCR?
END tpmAbsOutput;
```

Figure 3: Abstract TPM outputs

TPM Command	*Abstract PVS Command*	*Concrete PVS Command*
Startup		
TPM_Init	`ABS_Init`	
TPM_Startup	`ABS_Startup`	
TPM_SaveState	`ABS_SaveState`	
PCRs, seals and keys		
TPM_Extend	`ABS_Extend(h,i)`	
TPM_Seal	`ABS_Seal(sk,data)`	
TPM_Bind		
TPM_Unbind		
TPM_Unseal	`ABS_Unseal(s,uk)`	
TPM_CreateWrapKey	`ABS_CreateWrapKey(wk,parentk)`	
TPM_LoadKey2	`ABS_LoadKey2(k)`	
Quotes and Identities		
TPM_Quote	`ABS_Quote(aik,b,pm)`	
TPM_MakeIdentity	`ABS_MakeIdentity(naik,k)`	
TPM_ActivateIdentity	`ABS_ActivateIdentity(caik,k)`	
Ownership		
TPM_TakeOwnership	`ABS_TakeOwnership`	
TPM_OwnerClear		
TPM_ForceClear		
TPM_DisableOwnerClear		

Table 1: TPM command mapping to PVS command representation.

External Command	*Abstract PVS Command*	*Concrete PVS Command*
senter	`ABS_senterCom`	
sinit	`ABS_sinitCom`	
Save to memory	`ABS_Save(i,v)`	
Read from memory	`ABS_Read(i)`	
CA certification	`ABS_certify(aik,ek,freshk)`	
Power On	`powerCom`	
Power Off	`offCom`	

Table 2: System commands interacting with TPM.

2.3 TPM Abstract Outputs

2.4 Defining Abstract TPM Command Execution

The technique for specifying TPM command execution is to define a transition from `tmpAbsState` (figure 1) and `tpmInput` (figure 2) to `tpmAbsState`:

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

and a transition from `tmpAbsState` (figure 1) and `tpmInput` (figure 2) to `tpmAbsOutput`:

```
outputCom : tpmAbsState → tpmInput → tpmAbsOutput
```

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

```
(outputCom(s,c),executeCom(s,c))
```

This is a standard technique for defining state transition and output functions for any transition system.

As one would expect, `executeCom` and `outputCom` are defined by cases over `tpmAbsInput`. 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.

As a concrete example, consider the `ABS_ActivateIdentity` input. The function `activateIdentityState` defines how the TPM state is modified:

```
activateIdentityState(s:tpmAbsState,a:(wrapKey?),k:(symKey?)) : tpmAbsState =
loadKey2State(s,a);
```

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

```
activateIdentityOut(s:tpmAbsState,a:(wrapKey?),k:(symKey?)) : tpmAbsOutput =
  IF checkKeyRoot(a,srk(s)) THEN outSymKey(k) ELSE outNothing ENDIF;
```

Note that many commands in the current abstract model either modify state or generate output. In such cases only the pertinent function is defined with the `CASES` construct used to assemble the functions defaulting to not modifying the state and generating a null output.

2.5 Sequencing TPM Commands

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 outputs to the next state. Using a `LET` form, to execute `i;i'` would look like the following:[1]

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

where the tick denotes next.

The notation typically used does not use the `LET`, performing calls directly:

```
(executeCom(executeCom(s,i),outputCom(s,i)),
 outputCom(executeCom(s,i),outputCom(s,i)))
```

TPM command execution is inherently sequential, making this notation exceptionally obtuse and difficult manage. Thus, we choose to use an alternative approach that uses a *state monad* to model sequential execution. In effect, the state monad threads the state through sequential execution in the background.

To understand the state monad, lets first define a simple data type, `State`, having a single field called `state` that holds a function from an abstract state to an abstract state, abstract output 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.[2]

Two functions must be defined for any instance of a monad – unit and bind (`>>=`). We will also define sequence (`>>`) command that is a special case of bind commonly defined in most state monad implementation.

First we define `unit`, frequently called `return` in the literature. It's form is:

[1] We use the semicolon in the canonical style to represent sequential execution.
[2] The notation used here is consistent with the literature.

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

What `unit` does is lift a member of `tmpAbsState` into `State` – given a `tpmAbsState` it produces a `State`. Specifically, when `unit` is provided with a `tpmAbsState` it should give back a `State` whose `runState` field should simply produce it.

`runState` is a function from `tpmAbsState` to a `tpmAbsOutput`, `tpmAbsState` pair. Clearly, the state part of the output should be the state lifted by `unit`. 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.

Given an output value `a`, `unit(s)` evaluates asL

```
unit(a) = state(LAMBDA (s:tpmAbsState) : (a,s))
```

Recognize that the resulting value is of type `State`. This is not a state, but a state monad that can produce a state. Specifically, if we extract the `runState` function and apply it to a `tpmAbsState` we will get the output, state pair that we need. Specifically:

```
runState(unit(a))(st) = runState(state(LAMBDA (s:tpmAbsState)) :  (a,s))(st);
                      = (LAMBDA (s:tpmAbsState) : (a,s))(st)
                      = (a,st)
```

The output specified when the monad was created, `a`, is the output included in the application. No matter what `tpmAbsState` this `runState` is applied to the output will always be `a`. However, the state element of the output will always be the state input to `runState`. So, we have a function that will always produce a pair whose output value is `a` and whose state value is the state passed to it. This is what we mean by lifting a value into the state monad.

Before moving on to bind and sequence, a quick note about how `unit` works may simplify things. Applying `unit` to a `tpmAbsState` value produces a state monad value. In that value, the output is hard wired by the application of `unit` – it is provided as an input and the resulting monad carries it along. This is precisely how the monad will handle state. With a bit more machinery, we will use the state generated as a value in the monad representing the next state.

The second function defined for all monads 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 as to 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));
```

Keep in mind that bind does not produce a state. Instead it produces a state monad that given a state will produce and output state pair. That state is the parameter `s0` in the above implementation.

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 evalutes that function with `s1`, the intermediate state. So, the state is threaded throught the evaluation with the user providing only `s0`, the initial state. This initial state is the primary use of `unit` – a state is lifted into the monad and then pushed through a collection of bind operations to caculuate the state resulting from many command applications.

Another common operation is a specialization of bind that we refer to as *sequence* (`m << n`) shown here:

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

Sequence works exactly like bind except that the previuos output is ignored. Most TPM commands use sequence because technically they do not return anything, but simply modify state.

3 Verification Approach

The approach taken for verification is establishing a *weak bisimulation* [Sangiorgi, 2012] relation between an abstract requirements model and a model derived from the TPM specification as shown in figure 4.

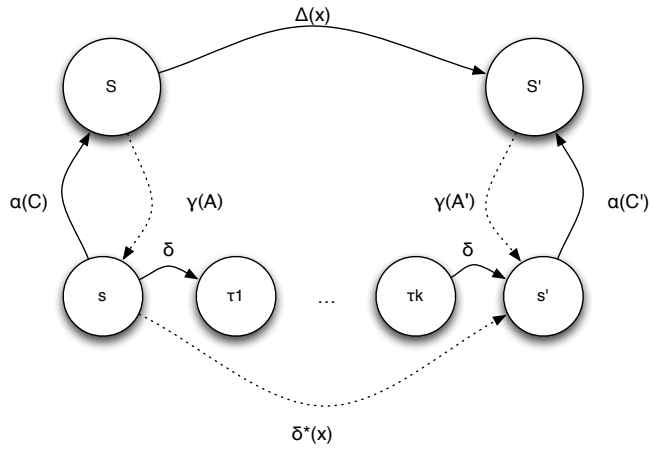

Figure 4: Weak bisimulation relation between an abstract transition system $A = (S, \Sigma, \Delta)$ and a concrete transition system $C = (s, \sigma, \delta)$.

We say that $A = (S, \Sigma, \Delta)$ is an *abstract model* where S is a set of abstract states, Σ is a set of actions on states and input, and $\Delta : S \times \Sigma \rightarrow \Sigma$ is a transitions on state and action. Similarly, we say that $C = (s, \sigma, \delta)$ is a *concrete state* where s is a set of concrete states, σ is a set of actions on states and input, and $\delta : s \times \sigma \rightarrow \sigma$ is a transition function.

We relate the abstract and concrete models through an *abstraction function*, $\alpha : s \rightarrow S$, and *concretization function*, $\gamma : S \rightarrow 2^s$. The abstraction and concretization functions must form a Galois Connection such that:

$$s \in \gamma(\alpha(s))$$

Specifically, when making the result of an abstraction concrete, the original state must be in the resulting set. Note that the concretization function may result in multiple states due to the necessity of specifying unknown detail.

We say that A and C are weakly bisimilar ($A \sim C$) if when $\alpha(s) = S$ then $\alpha(\delta^*(s)) = \Delta(S)$ for all inputs to s.

Definition 1

$$A \sim C \equiv \forall s_0 : s \cdot \exists S_0 : S \cdot \alpha(s_0) = S_0 \Rightarrow \alpha(\delta^*(s_0)) = \Delta(S_0)$$

In the formal TPM model `tpmAbsState` defines S while `tpmConcState` defines s.

Glossary

This glossary is intended to document some common acronyms as well as define some common terms. It is currently a bit haphazard and a number of elements are missing.

A Trusted Platform Definitions

Trusted Platform Module (TPM) – Hardware Trusted Platform Module as defined by TCG.

Process Configuration Register (PCR) – Registers defined in the TPM. In the TPM there are at least 16 and they are 20 bytes wide.

Bound to PCR – Encryption including PCR values that must match TPM PCR values before decryption is performed.

Sealed to State – Operation performed by the TPM where data is encrypted and bound to PCRs or a PCR composite that must be checked before unsealing. Only data is sealed.

Wrapped Key – An asymmetric key with its private key encrypted and its public key visible. Only keys are wrapped.

Virtual TPM (vTPM) – A virtual Trusted Platform Module

Certified Key (CK) - Asymmetric key with private key signed by the AIK private key.

Endorsement Key (EK) - Asymmetric key who's private key is in the TPM hardware. Private key is used to sign data from TPM while public key is used to encrypt sensitive data sent to the TPM. The EK is set by the TPM factory and may not be reset.

Attestation Identity Key (AIK) - Asymmetric key whose private key is used for only two purposes: (i) sign (or attest to) TPM internal state; (ii) sign (or certify) other general purpose keys. AIKs are generated by a TPM, certified by a trusted third party, and used for signing in lieu of the EK.

Storage Root Key (SRK) - Root of secure storage hierarchy. Used to encrypt storage keys that exist outside the TPM. The SRK may be reset.

Attestation Identity Certificate (AIC) - Certificate provided by a trusted third party binding an AIK to a specific trusted platform.

Digest - 20-byte Value contained in a PCR.

PCR Composite - Single digest value generated from a collection of PCR values.

Quote - A value along with a set of PCR values or PCR composite signed by a TPM using an AIK.

Root of Trust for Measurement (RTM) - the "place to stand" for measurement. Effectively a hardware-based trusted launch point that will

faithfully measure, start and pass control to its target without trusting other components.

Root of Trust for Storage (RTS) - the "place to stand" for measurement storage. Effectively a trusted hardware store that will store measurements with integrity without trusting other components.

Root of Trust for Reporting (RTR) - the "place to stand" for generating quotes and providing evidence with integrity and authenticity. Effectively a trusted hardware component that generates and signs quotes.

B Cryptography Notations

Hash Notation $\#data$ - The hash of $data$.

Certificate Notation $[[cert]]_{key}$ - Certificate, $cert$, signed by key.

Signed Data Notation $\{|data|\}_{key^{-1}}$ - $data$, signed by key.

Encrypted Data Notation $\{data\}_{key}$ - $data$ encrypted with key.

Sealed Data Notation $seal(data, \{pcrs\})$ - $data$ sealed to $pcrs$.

Wrapped Key Notation $wrap(k, \{pcrs\})$ - Equivalent to the key pair $(seal(k^{-1}, \{pcrs\}), k)$

C Intel Secure Boot

Trusted eXecution Technology (TXT) - Intel's trusted boot support.

Measured Launch Environment (MLE) - Run time environment providing a measured boot. Initialized and started by the SINIT command execution.

SENTER or GETSEC - Intel's trusted boot command. Provides synchronization, special bus cycles, and a special environment residing on the CPU (ACEA).

Secure INITialization Instructions (SINIT) - Code for performing secure initialization. Loaded by SENTER and validated by the ACM.

Authenticated Code Execution Area (ACEA) - CPU resident environment for executing code known as the Authenticated Code Module (ACM). Boot sequence involving SENTER and SINIT is as follows:

1. Load SINIT and MLE into memory
2. Invoke SENTER (GETSEC)
3. Establish special environment (ACEA)
4. Load SINIT into ACEA
5. Validate SINIT digital signature and store SINIT identity in TPM
6. SINIT measures MLE in memory and stores MLE identity in TPM
7. SINIT passes control to MLE

Authenticated Code Module (ACM) - Code running in the ACEA. May be used for validating platform configuration, measuring the measured launch environment, cleaning up after crashes.

References

D. Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2012.

