

Pasture Node State Specification

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

Ramakrishna Kotla
Microsoft Research, Silicon Valley

Aug 17, 2012

Pasture [5] is a secure messaging and logging library that enables secure off-line data access on untrusted user devices by leveraging commodity trusted hardware. Pasture does not trust the application, OS, or hypervisor and even admits hardware snooping attacks, while providing two important safety properties: *access-undeniability* (a user cannot deny any off-line data access obtained by his device without failing an audit) and *verifiable-revocation* (a user who generates a verifiable proof of revocation of unaccessed data can never access that data in the future). Each node running Pasture uses its Trusted Platform Module to protect and log access to encryption keys that shield the data. Permanently forfeiting the ability to access an unused decryption key is the basis of revocation.

This report presents a formal specification, written in TLA+ [7], for the actions of a Pasture node, along with the results of checking the specification using the TLC model checker. We also present a formal proof of the Pasture safety properties, which has been mechanically checked using the TLA+ proof system [2]. These results have been presented at a TLA Workshop [11].

This report also presents a formal specification and correctness proof of an optimized version of Pasture, in which the functions of two of Pasture's Program Configuration Registers (PCRs) have been combined into one PCR. In spite of the slight change in the specification, the formal proof of correctness of optimized Pasture is almost identical to the original proof.

Appendices contain the full texts of the formal specifications and formal proofs.

1 Introduction

Once a formal specification for a system is written, it is usually desirable to check that the specification conforms to some concept of correctness. One way to do this is to prove that the specification is a refinement of a simpler specification whose correctness is more obvious. This refinement approach is used in two stages in proving that the Memoir system is correct [9, 3]. Another way is to write invariants, or safety properties, and then directly prove that the specification maintains the invariants.

Of course, in either case there could be oversights in the proof, so to be certain that the proof is correct, it must be a formal proof checked by a mechanical proof checker. Unfortunately, such formal proofs tend to be lengthy and tedious, since the limited deductive power of current mechanical proof checkers requires detailed proof steps. So it is a good idea to be fairly confident that the specification is correct before investing the effort in writing a formal proof.

To get confidence that a specification maintains its invariants, one can use a model checker to explore the state space. Unfortunately, a model checker is limited to exploring a finite number of states, and the state space usually explodes rapidly as the model configuration parameters are increased. Although model checking may have found no errors in the specification, there is always the question of whether, if there were an error, would the model checker have been able to find it within the configurations that are feasible to check? To address this question, we propose introducing some errors on purpose and seeing if the model checker can find them.

We present the results of model checking, with inserted errors, a TLA+ [7] specification for a node in Pasture, a

messaging library that provides secure off-line access to data using a TPM. The state space of even small configurations of the Pasture specification is too large to gain much confidence by direct model checking, but observing that intentional bugs can be found within the configurations that can be checked gives a reasonable confidence that the specification is correct. We then go on to describe a formal proof that the specification maintains its invariants. The formal proof has been checked using the TLA+ Proof System [2].

After writing, checking, and proving the correctness of the Pasture specification, we realized that the implementation could be optimized by combining the functions of two of Pasture’s Program Configuration Registers (PCRs) into one PCR. Surprisingly enough, it turned out to be a simple matter to edit our initial formal specification and proof to produce a formal specification and proof of optimized Pasture.

2 Overview of Pasture

Pasture [5] is a messaging library that provides secure off-line access to data. When on-line, the *receiver* downloads an encrypted copy of the data from a *sender*. Later, when off-line, the receiver makes a decision either (1) to obtain access to the decryption key and thus to the data, or (2) to revoke access to the decryption key and thus effectively delete the data without reading it.

Pasture provides two safety properties: *access undeniability* and *verifiable revocation*. Access undeniability means that a receiver cannot deny any decision it made to obtain access to data and still survive an audit. Verifiable revocation means that a receiver can provide a *proof of revocation* for any decision it made to revoke access to data. This proof establishes that the receiver never did and never will be able to access that data.¹

These properties could be used, for example, by a video rental service. The receiver could pay for and download an encrypted video from the sender. Later, the receiver could decide whether to obtain access to the video and watch it, or revoke access and never watch it. Afterwards,

¹The properties apply only when the sender is correct. A faulty sender could just send the data in the clear to a receiver, and there could be no guarantee about whether the receiver accessed the data or not. The intent is to protect a correct sender against a faulty receiver.

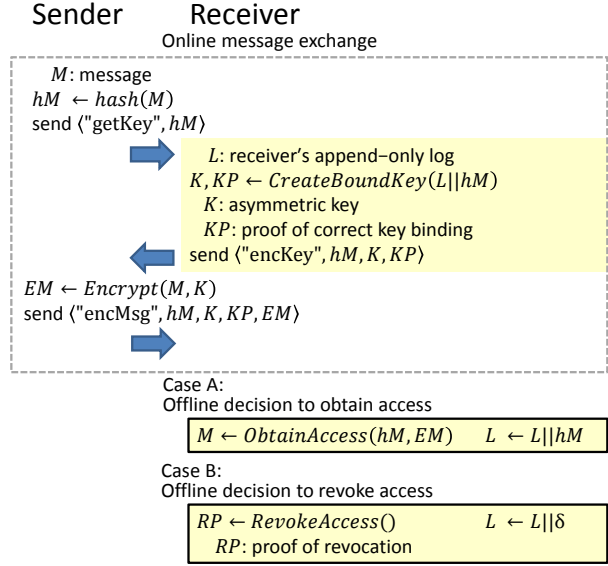


Figure 1: Pasture protocol.

if access was revoked, the receiver could present the proof of revocation to the sender and get a refund.

Pasture works by implementing a tamper-evident append-only log of decisions on the receiver. Figure 1 shows the protocol.² For this paper we concentrate on the implementation of the tamper-evident append-only log and how it relates to the use of a decryption key and the production of a proof of revocation.

Pasture uses a Trusted Platform Module (TPM) [12, 1] in the receiver to maintain a cryptographic summary of the receiver’s log and to protect decryption keys. We assume that the reader is generally familiar with how TPMs work.

The cryptographic summary of the receiver’s log is maintained in a Platform Configuration Register (PCR) inside the receiver’s TPM. A PCR can be updated only via the TPM primitive TPM_Extend, which corresponds to the action of appending a value (called a *measurement* in the TPM literature) to the log. A given PCR value serves as a cryptographically unique representation of the sequence of measurements used to produce it, since it is

²Certain details related to preventing spoofing have been omitted, such as signatures on the messages. Also, we omit showing that the sender should verify the proof KP before encrypting the message.

cryptographically impossible to determine any other sequence that would produce the same result. Pasture uses PCR_{APP} to hold the log summary.

The TPM primitive `TPM_CreateWrapKey` is used to bind the decryption key to a potential future state of the log, in which the current log has been extended by the decision to obtain access to the key. This decision is represented in the log as the cryptographic hash hM of the message. To revoke access to the key, the receiver instead extends its log by $\delta \neq hM$. Assuming that the cryptography is unbreakable, this extension makes it impossible to reach the log state to which the decryption key is bound, and thus makes it impossible ever to use the decryption key.

Since the TPM's PCRs are volatile, and are reset to their initial values on reboot, the main difficulty faced by Pasture is how to preserve its state across reboots. If an adversary could rollback Pasture state to an earlier point, arranging to violate Pasture's safety properties of access undeniability and verifiable revocation would be easy.

Memoir [9] presented a solution to this problem for any deterministic application. Memoir maintains a cryptographic log summary of application states via a PCR much like Pasture's PCR_{APP} . The optimized Memoir solution adds a checkpoint routine to the system shutdown sequence and a recovery routine to the system boot sequence. The checkpoint routine copies the PCR to an NVRAM location and then sets an NVRAM flag indicating that the copy is current. The recovery routine checks that the NVRAM value is marked as current and if so plays back measurements from the full log, re-extending the PCR until its contents matches the value saved in the NVRAM. Memoir uses an *ExtensionSecret* to prevent an adversary from duplicating a prefix of the re-extension process. Any time that the log is extended, when the measurement is extended on the PCR the NVRAM flag is cleared indicating that the NVRAM copy of the PCR is no longer current.

Memoir exploits Secure Execution Mode (SEM) in the manner developed by Flicker [8]. SEM enables a routine to run in a protected environment, with interrupts, other cores, and DMA disabled, and with a special PCR_{SEM} set to a value (otherwise cryptographically unreachable) based on a cryptographic hash of the routine.

Pasture adopts most of the Memoir approach, with a few modifications so that the normal Pasture operations of

CreateBoundKey(hM):

```

 $R_t \leftarrow \text{TPM\_Read}(PCR_{APP})$ 
 $R_{t+1} \leftarrow \text{SHA1}(R_t \parallel hM)$ 
transport session
┌ BINDKEY
│    $K \leftarrow \text{TPM\_CreateWrapKey}(\{$ 
│      $PCR_{APP} = R_{t+1} \ \&\&$ 
│      $PCR_{SEM} = \text{SemHappy} \ \&\&$ 
│      $PCR_{SEAL} = \text{SealReboot} \ \}$ 
│    $\alpha \leftarrow$ 
└    $KP \leftarrow \langle \text{"CreateBoundKey"}, hM, R_t, R_{t+1}, \alpha \rangle$ 

```

ObtainAccess(hM, EM):

```

append  $hM$  to full log
 $\text{TPM\_Extend}(PCR_{APP}, hM)$ 
 $M \leftarrow \text{TPM\_Unbind}(EM)$ 

```

RevokeAccess():

```

 $R_t \leftarrow \text{TPM\_Read}(PCR_{APP})$ 
append  $\delta$  to full log
 $\text{TPM\_Extend}(PCR_{APP}, \delta)$ 
 $R'_{t+1}, S'_{t+1}, A'_{t+1}, \alpha \leftarrow$ 
   $\text{TPM\_Quote}(PCR_{APP}, PCR_{SEM}, PCR_{SEAL})$ 
 $RP \leftarrow \langle \text{"RevokeAccess"}, \delta, R_t, R'_{t+1}, S'_{t+1}, A'_{t+1}, \alpha \rangle$ 

```

Audit(*nonce*):

```

 $R_t, S_t, A_t, \alpha \leftarrow$ 
   $\text{TPM\_Quote}(PCR_{APP}, PCR_{SEM}, PCR_{SEAL}, \text{nonce})$ 
 $AP \leftarrow \langle \text{"Audit"}, \text{full log}, R_t, S_t, A_t, \text{nonce}, \alpha \rangle$ 

```

Recover():

```

FOR EACH entry  $\Delta$  on full log:  $\text{TPM\_Extend}(PCR_{APP}, \Delta)$ 
secure execution mode
┌ IF  $nv.\text{current} \ \&\& \ \text{nv}.R = \text{TPM\_Read}(PCR_{APP})$ 
│ THEN
│    $nv.\text{current} \leftarrow \text{FALSE}$ 
│    $\text{TPM\_Extend}(PCR_{SEM}, \text{Happy})$ 
│ ELSE
│    $\text{TPM\_Extend}(PCR_{SEM}, \text{Unhappy})$ 
└

```

Checkpoint():

```

transport session
┌ SEAL
│    $R_t \leftarrow \text{TPM\_Read}(PCR_{APP})$ 
│    $S_t \leftarrow \text{TPM\_Read}(PCR_{SEM})$ 
│    $A_t \leftarrow \text{TPM\_Read}(PCR_{SEAL})$ 
│    $C_t \leftarrow \text{TPM\_ReadCounter}(CTR)$ 
│    $\alpha \leftarrow \text{TPM\_Extend}(PCR_{SEAL}, \text{Seal})$ 
└
secure execution mode
┌  $nv.R \leftarrow R_t$ 
│ IF  $\text{Valid}_{SEAL}(\alpha, R_t, S_t, A_t, C_t)$ 
│   &&  $S_t = \text{SemHappy}$ 
│   &&  $A_t = \text{SealReboot}$ 
│   &&  $C_t = \text{TPM\_ReadCounter}(CTR)$ 
│ THEN
│    $\text{TPM\_IncrementCounter}(CTR)$ 
│    $nv.\text{current} \leftarrow \text{TRUE}$ 
└    $\text{TPM\_Extend}(PCR_{SEM}, \text{Unhappy})$ 

```

Figure 2: Pasture operations.

CreateBoundKey, ObtainAccess, and RevokeAccess do not need to run in SEM. In this way Pasture exploits the specific nature of its application to obtain a solution with much less overhead in its particular case. Figure 2 shows the implementation of Pasture operations.

Pasture’s Recover operation re-extends PCR_{APP} from the full log, then enters SEM to verify that PCR_{APP} matches the value saved in the NVRAM (and that the value in NVRAM is current). If so, PCR_{SEM} is extended by *Happy*, to produce a value *SemHappy* that can be reached in no other way, and the NVRAM flag is cleared to indicate that the NVRAM value is no longer current. Otherwise, PCR_{SEM} is extended by *Unhappy*, producing a different value.

CreateBoundKey requires both that PCR_{APP} contain the proposed future log summary and that PCR_{SEM} contain *SemHappy* in order for decryption to be possible. The adversary could reboot the system and re-extend PCR_{APP} , but cannot arrange for PCR_{SEM} to contain *SemHappy* and so cannot rollback and access a decryption key.

Likewise, RevokeAccess and Audit quote both PCR_{APP} and PCR_{SEM} in order to prove that PCR_{APP} was quoted at a time when PCR_{SEM} contained *SemHappy*.

Pasture’s Checkpoint operation has a difficulty. It needs to verify that PCR_{SEM} contains *SemHappy* so that it can trust the current contents of PCR_{APP} , but it has to enter SEM in order to protect its actions from interference by the adversary. Its solution is to use a transport session SEAL to get an attestation α of the contents of PCR_{APP} and PCR_{SEM} before entering SEM. A TPM monotonic counter CTR is used to prevent the adversary from replaying an earlier SEAL attestation.

Taking the SEAL also has to destroy the usefulness of PCR_{APP} , or else an adversary could take the SEAL, extend PCR_{APP} to obtain access to a key or generate a verifiable proof of revocation, and then pass the SEAL to the checkpoint SEM routine and continue with a normal reboot and recovery, which would rollback the actions that the adversary performed after taking the SEAL. For this purpose, PCR_{SEAL} is used. PCR_{SEAL} normally contains its initial value *SealReboot*, which is checked in CreateBoundKey and quoted in RevokeAccess and Audit. The SEAL transport session extends PCR_{SEAL} thus rendering PCR_{APP} useless until the next reboot.

3 The specification

Appendix A gives a TLA+ [7] specification of the state within a Pasture node. The specification closely tracks the Pasture operations shown in Figure 2 and also models the actions of an adversary who has the power to extend PCRs, to observe whatever attestations are created, to invoke Pasture’s secure execution mode routines with any parameters known to the adversary, and to reboot the node at arbitrary times. Since we assume that cryptography cannot be broken, the adversary does not have the power to forge attestations or to set PCRs to an arbitrary value.³

The specification abstracts the Pasture node in the following ways:

- *Only one hash.* The specification models the hash hM as just one value, PcrxOBTAIN. The revoke measurement δ is modeled as PcrxREVOKE. Note that if multiple, distinct hash values were modeled, the specification would be symmetric over permutations of the hash values. Modeling all hash values as just the one value PcrxOBTAIN eliminates this symmetry from the specification. No descriptive power is lost, because the specification does not admit any actions that compare isolated hash values.
- *Potential key bindings.* The specification assumes that a key may be bound to any current state of the log extended by PcrxOBTAIN. Extending the log by PcrxOBTAIN obtains access to this key and extending the log by PcrxREVOKE revokes access to this key.
- *Recovery.* In Pasture, recovery first re-extends PCR_{APP} from measurements recorded in the full log and then enters SEM to verify that the resulting value in PCR_{APP} is current. The specification accomplishes the re-extension by allowing any possible sequence of extensions of PCR_{APP} , since this is within the power of the adversary. The specification models recovery as the re-extension sequence that happens to be the correct one.

³The protection of Pasture’s NVRAM depends on the assumption that the value *SemProtect* is present in PCR_{SEM} only during Pasture’s secure execution mode routines, which is the subject of the invariant *InvNvProtection*.

- *Checkpoint*. In Pasture, checkpoint first performs a SEAL transport session and gets the attestation, and then enters SEM to verify the attestation and then record the log summary in NVRAM. The specification collects a knowledge of all SEAL attestations that have ever been generated and allows choosing any known one for the SEM routine to verify, since this is within the power of the adversary. The specification models checkpoint as choosing the correct one.

The specification starts with a series of *Bug* definitions all set to FALSE. Overriding one of these definitions with TRUE introduces a bug into the specification as discussed later in Section 5.

Next the specification introduces definitions for PCRs. A PCR is modeled as an initial value in *Pcri* combined with a sequence of extensions in *Pcrx*.

Next the specification introduces definitions for PC values within Secure Execution Mode (SEM). When the node is in SEM the adversary cannot interpose any actions except to reboot the node.

Next the specification introduces definitions for Pasture’s protected NVRAM, the SEAL transport session, and then finally the state of the entire node.

There are two “fiduciary” variables which are used for expressing invariants: *obtains* and *revokes*.

The variable *obtains* is a set that contains all application PCR values that have been used to obtain a key. Since the last decision logged must be the decision to obtain the key, these PCR values all have *PcrxOBTAIN* as their final extension.

The variable *revokes* is a set that contains all application PCR values have been used for a proof of revocation. Since the last decision logged must be the decision to revoke a key, these PCR values all have *PcrxREVOKE* as their final extension.

Next the specification introduces the next state relation decomposed as a long series of actions. Then the actual *Init* and *Next* definitions of the specification are presented, followed by the complete specification *Spec*.

Finally, the specification introduces a list of invariants. The invariant *InvType* asserts that all variables always contain values of the correct type. The invariant *InvNv-Protection* asserts that access to Pasture’s NVRAM region (which is controlled by the value contained in the

secure execution mode PCR) is permitted precisely when the node is in secure execution mode. The invariants *InvAccessUndeniability* and *InvVerifiableRevocation* correspond to the main safety properties of Pasture.

Access undeniability is equivalent to saying that whenever the node is auditable, every element in *obtains* is a prefix of the current application PCR. This means that whenever a node is auditable, it must provide a full log that lists every decision it made to obtain access to a key.

Verifiable revocation is equivalent to saying that there is no PCR $o \in obtains$ and PCR $r \in revokes$ such that everything in o except the last decision (which must be OBTAIN) matches everything in r except the last decision (which must be REVOKE). If it were possible to have such an o and r , it would mean that there would be a key for which both access was obtained and also a proof of revocation was generated.

4 Model checking

Appendix B shows a TLA+ specification for model checking the Pasture node specification. The model specification creates an instance of PastureNode with constants for the initial values and extensions of PCRs. The constants are carefully chosen to be a minimal set that satisfies the required assumptions.

The model specification also introduces a parameterized constraint to limit the number of states to a finite number. The parameters of the constraint are as follows:

- *MaxAppPcrLen*. The maximum number of extensions of PCR_{APP} ; and therefore the maximum number of entries in the log and the maximum number of keys for which access can be obtained or revoked.
- *MaxSemPcrLen*. The maximum number of extensions of PCR_{SEM} . Pasture requires at least one, so that the Pasture SEM routines can extend PCR_{SEM} before exiting, which is required to remove access privileges from Pasture’s NVRAM. Note that the specification does not count entering a SEM routine as requiring an extension to PCR_{SEM} , but merely initializes PCR_{SEM} with *SemProtected* which represents the result of resetting and then extending with the cryptographic module hash. The TPM semantics

of resetting PCR_{SEM} ensures that it is cryptographically impossible to reach *SemProtected* in any other way.

- **MaxSealPcrLen.** The maximum number of extensions of PCR_{SEAL} . Pasture requires at least one, so that the SEAL transport session can extend PCR_{SEAL} .
- **MaxTsValues.** The maximum number of SEAL attestations that can be known at any one time. Pasture requires at least one, so that the most recent SEAL attestation can be provided to the Checkpoint SEM routine. Note that the specification admits of forgetting a SEAL attestation that once was known. This permits model checking a Pasture configuration through multiple reboots with only one SEAL attestation known at a time, since only the most recent one needs to be remembered for Pasture to continue to function.
- **MaxBootCtr.** The maximum value of the boot counter. Pasture increments this counter once each time through the Checkpoint routine.

We used the TLA+ toolbox [6] with TLC2 version 2.05 to model check the specification for various configurations. For each configuration, TLC determined the maximum depth of the state space graph as well as the total number of distinct states. No violations were found. Table 1 shows the results.

For brevity, we refer to a specific configuration by listing the parameter values left-to-right in the order shown in Table 1. For example, the (2,1,2,2,2) configuration is the last configuration listed in the table.

We started by model checking configurations on an Intel Core™ i7 M620 laptop with 4 GB of memory and 4 cores @ 2.67 GHz. As expected, the number of distinct states and consequently the model checking run time increased enormously as the configuration parameters were increased. Figure 3 shows TLC’s agonizing plot of queue size over time for the largest configuration we model checked using the laptop. The rate of next state exploration became particularly slow after about two hours of run time as TLC was completely disk-bound.

To check some larger configurations, we obtained unshared access to an AMD Opteron™ 6168 server with 128

			laptop	server					
			4 GB	128 GB					
			4 cores	48 cores					
	configuration	depth	distinct states	run time					
MaxAppPcrLen	1	1	1	1	31	47742	7s		
MaxSemPcrLen	1	1	1	2	32	106556	83s		
MaxSealPcrLen	1	1	1	2	1	36	966697	110s	
MaxTsValues	1	1	2	1	1	33	369750	41s	
MaxBootCtr	1	2	1	1	1	34	283760	83s	
	2	1	1	1	1	36	1062426	110s	
	1	1	2	1	2	34	853554	84s	
	1	1	1	2	2	42	3011870	293s	
	2	2	1	1	1	40	6800068	14m	6m
	2	2	2	1	1	42	68210216	157m	48m
	2	2	2	1	2	43	175125010	613m	122m
	2	1	1	2	2	47	162409454		106m
	2	1	1	2	3	48	379647806		224m
	2	1	2	2	2	53	4234887880		3596m

Table 1: Model checking results. Wall clock run time. Complete state space exploration.

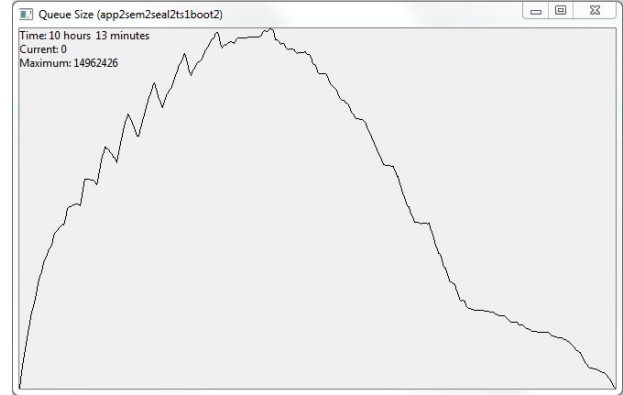


Figure 3: TLC queue size plot for the (2,2,2,1,2) configuration running on the laptop.

GB of memory and 48 cores @ 1.90 GHz. However, even using this large server machine, the enormous state space explosion of the Pasture node specification exposed some limitations in TLC.

The (2,1,1,2,3) configuration has over 379 million dis-

tinct states, so there is a non-trivial probability of fingerprint collision.⁴ Such a collision would cause TLC to fail to explore the complete state space. TLC reported a calculated collision probability of 0.058 and an observed collision probability of 0.027. We re-ran the configuration with a different fingerprint seed and four hours later were pleased to see that the second run explored the same number of distinct states, this time reporting an observed collision probability of 0.002. We performed a third run with yet another seed, and TLC again explored the same number of distinct states. At this point we decided that the TLC runs on this configuration were almost certainly not suffering from fingerprint collision.

The (2,1,2,2,2) configuration has over 4 billion distinct states. According to the birthday paradox, the probability of a 64-bit fingerprint collision among this many states is 0.38. TLC reported an observed collision probability of 1.0, for whatever that is worth. We re-ran the configuration with a different fingerprint seed and two and a half days later were pleased to see that the second run explored the same number of distinct states. We performed a third run with yet another seed, and this time TLC explored five fewer distinct states. The number of distinct states listed for this configuration in Table 1 is the number explored in the first and second runs. However, with these results it is not clear whether or not TLC is actually exploring the entire state space. None of the runs found any errors.

Clearly, the probability of a fingerprint collision would make the results of running TLC on any larger configuration fairly inconclusive, even if we wanted to wait for such a run to complete.

We did not apply SYMMETRY in our model checking runs because the specification as written has none. The use of the one value PcrxOBTAIN as a model for any hash value hM wrings out the one symmetry that would be present in a more detailed specification.

It was nice to see that none of the invariants were violated for the configurations that TLC could check. However, there is always the possibility that a bug lurks over the horizon. Normally, we would like to check configurations with parameter values up to at least three. In our

⁴Assuming all fingerprints are equally likely, the probability of a collision among k independent probes into a set of size H can be estimated as $1 - \exp(-k * (k - 1) / (2 * H))$. This is known as the birthday paradox. The formula calculates out to 0.0039 for the given number of distinct states using 64-bit fingerprints.

experience, a system will often have interesting behavior when there is the chance for three instances of something to interact. But in model checking the Pasture specification, it was not feasible to check a configuration in which all of the parameters were two, let alone three. This was disappointing.

5 Inserted bugs

To get more assurance that the specification was correct, we intentionally added bugs to the specification to see if the model checker could find violations within the small configurations that were feasible to check. The idea was to start with the smallest configuration and then carefully increase the configuration parameters until the model checker found a violation.

In order to insert a bug, we identified a place in the specification where it seemed likely that omitting a check or an action would prove harmful to correct behavior. Since the intent of a specification is to capture what is necessary for correct behavior, such *bugs of omission* could be inserted at almost any point. Table 2 shows the results. The 16 different bugs we investigated are as follows:

- *BugObtainAccessNoCheckHappy* models what happens if Pasture fails to bind the key such that it can be used for decryption only when the secure execution mode PCR is happy.
- *BugObtainAccessNoCheckSeal* models what happens if Pasture fails to bind the key such that it can be used for decryption only when the seal PCR contains its reboot value.
- *BugProveRevokeNoCheckHappy* models what happens if Pasture fails to check in a proof of revocation that the application PCR was quoted at a time when simultaneously the secure execution mode PCR was happy.
- *BugProveRevokeNoCheckSeal* models what happens if Pasture fails to check in a proof of revocation that the application PCR was quoted at a time when simultaneously the seal PCR contained its reboot value.

bug	configuration	MaxAppPcrLen	MaxSemPcrLen	MaxSealPcrLen	MaxTsValues	MaxBootCtr	counterexample (if any) found at			invariant violated
							depth	distinct states	run time	
<i>BugObtainAccessNoCheckHappy</i>	1 1 1 1 1	1	1	1	1	1	8	1969	4s	InvAccessUndeniability
<i>BugObtainAccessNoCheckSeal</i>	1 1 1 1 1	1	1	1	1	1	19	29259	7s	InvAccessUndeniability
<i>BugProveRevokeNoCheckHappy</i>	1 1 1 1 1	1	1	1	1	1	10	5836	3s	InvVerifiableRevocation
<i>BugProveRevokeNoCheckSeal</i>	1 1 1 1 1	1	1	1	1	1	21	37812	7s	InvVerifiableRevocation
<i>BugRecovNoCheckApp</i>	1 1 1 1 1	1	1	1	1	1	19	28013	6s	InvAccessUndeniability
<i>BugRecovNoCheckCur</i>	1 1 1 1 1	1	1	1	1	1	12	9029	5s	InvAccessUndeniability
<i>BugRecovNoClrCur</i>	1 1 1 1 1	1	1	1	1	1	12	6368	4s	InvAccessUndeniability
<i>BugSealNoExt</i>	1 1 1 1 1	1	1	1	1	1	19	109599	15s	InvAccessUndeniability
<i>BugChkptNoCheckTsHappy</i>	1 1 1 1 1	1	1	1	1	1	20	42021	8s	InvAccessUndeniability
<i>BugChkptNoCheckTsSeal</i>	1 1 2 1 2	1	1	2	1	2	34	874078	165s	—none—
<i>BugChkptNoCheckTsCtr</i>	1 1 1 1 2	1	1	1	1	2	29	107183	16s	InvAccessUndeniability
<i>BugChkptSaveCurApp</i>	1 1 1 1 1	1	1	1	1	1	20	32826	8s	InvAccessUndeniability
<i>BugChkptNoIncCtr</i>	1 1 1 1 1	1	1	1	1	1	29	66215	11s	InvAccessUndeniability
<i>BugChkptNoSetCur</i>	1 1 1 2 2	1	1	1	2	2	32	198270	250s	—none—
<i>BugAuditNoCheckHappy</i>	1 1 1 1 1	1	1	1	1	1	9	2490	4s	InvAccessUndeniability
<i>BugAuditNoCheckSeal</i>	1 1 2 1 2	1	1	2	1	2	34	853554	166s	—none—

Table 2: Model checking results for inserted bugs. All runs performed on laptop.

- *BugRecovNoCheckApp* models what happens if secure execution mode within recovery fails to check that the application PCR was restored to the value saved in the NVRAM.
- *BugRecovNoCheckCur* models what happens if secure execution mode within recovery fails to check that the value saved in the NVRAM is marked as current.
- *BugRecovNoClrCur* models what happens if secure execution mode within recovery fails to clear the *current* flag in the NVRAM.
- *BugSealNoExt* models what happens if the seal transport session within checkpoint fails to extend the seal PCR.
- *BugChkptNoCheckTsHappy* models what happens if secure execution mode within checkpoint fails to check that the seal attestation recorded that the secure execution mode PCR was happy.
- *BugChkptNoCheckTsSeal* models what happens if secure execution mode within checkpoint fails to check that the seal attestation recorded that the seal PCR contained its reboot value.
- *BugChkptNoCheckTsCtr* models what happens if secure execution mode within checkpoint fails to check that the seal attestation recorded the same value of the boot counter as it currently contains.
- *BugChkptSaveCurApp* models what happens if secure execution mode within checkpoint saves in NVRAM the current application PCR rather than the value of the application PCR recorded in the seal attestation.
- *BugChkptNoIncCtr* models what happens if secure execution mode within checkpoint fails increment the boot counter.
- *BugChkptNoSetCur* models what happens if secure execution mode within checkpoint fails set the *current* flag in the NVRAM.

- *BugAuditNoCheckHappy* models what happens if the verifier of an audit fails to check that the application PCR was quoted at a time when simultaneously the secure execution mode PCR was happy.
- *BugAuditNoCheckSeal* models what happens if the verifier of an audit fails to check that the application PCR was quoted at a time when simultaneously the seal PCR contained its reboot value.

5.1 Rapid finding of counterexamples

For all but three bugs the model checker found, within a few seconds of wall clock run time in a very small configuration, a counterexample execution trace that exhibited a violation of an invariant.

For example, consider *BugChkptNoCheckTsCtr*. In this bug, the secure execution mode routine within Checkpoint neglects to check that the SEAL attestation quotes a boot counter value that is the same as the current boot counter value.

The counterexample found by TLC violated the invariant *InvAccessUndeniability* because execution reached a state in which (1) a key was present in the *obtains* fiduciary variable, meaning that at some point access had been obtained to the key, and (2) the node was auditable but the resulting audit log (based on PCR_{APP}) did not include this key. In the TLC counterexample, the state got this way as follows:

1. an initial Recovery sequence,
2. a normal Checkpoint sequence, which performed a SEAL transport session and passed the SEAL attestation to the secure execution mode routine within Checkpoint, which saved the initial, empty log in the NVRAM,
3. a reboot,
4. a normal Recovery sequence,
5. an extension of PCR_{APP} to obtain access to a key,
6. an adversarial entry to the secure execution mode routine within Checkpoint, passing it the SEAL attestation from the first Checkpoint sequence, and then performing the routine to save the initial, empty log in the NVRAM as current, (this is where the bug took effect)
7. a reboot, and finally

8. a normal Recovery sequence, which restored PCR_{APP} to the value of the empty log, while establishing $\text{PCR}_{\text{SEM}} = \text{SemHappy}$ and $\text{PCR}_{\text{SEAL}} = \text{SealReboot}$.

The counterexample requires the boot counter to be incremented twice, which is why the configuration that exhibits the counterexample requires $\text{MaxBootCtr} = 2$.

All of the other counterexamples were found in a minimal configuration that permitted at most one boot counter increment. Since the counterexamples were all found in such very small configurations, it would seem likely that the Pasture node specification does not have any “interesting” behavior that comes out only at higher configuration parameter values. This result gives a reasonable assurance that if there were a bug in the original specification, it would have been found in the original model checking runs.

In prior work [10] we also found that inserted bugs were detected in model checking runs much shorter than the runs required to model check the correct specification with “decent” configuration parameter values, although not nearly to the dramatic extent that we see in the Pasture node specification.

5.2 Bugs that were not safety violations

In three cases the bugs we introduced did not produce counterexamples in small configurations. We examined these bugs more closely and it turned out that they were not safety violations after all.

In the case of *BugChkptNoSetCur*, the checkpoint routine fails to set the *current* flag in the NVRAM after saving the application PCR. The consequence of this bug is that it will not be possible to recover after a reboot. Although this is a serious liveness problem, it is not a safety violation.

The other two cases, *BugChkptNoCheckTsSeal* and *BugAuditNoCheckSeal* are perhaps more interesting.

In *BugChkptNoCheckTsSeal* the checkpoint SEM routine fails to check that $\text{PCR}_{\text{SEAL}} = \text{SealReboot}$. We can see that this bug results in different execution behavior in the (1,1,2,1,2) configuration because the number of distinct states with the bug in Table 2 is different from the number listed for this configuration in Table 1.

With this bug, the adversary can run the transport ses-

sion to take a SEAL, then perform some additional extensions on PCR_{APP} , then perform the transport session a second time to take a second SEAL. Since the checkpoint SEM routine fails to check the value of PCR_{SEAL} in the SEAL, it will accept either of the two SEALS indiscriminately. So with this bug, the adversary can optionally either leave the additional extensions on the log by calling checkpoint with the second SEAL or retract them by calling checkpoint with the first SEAL, in either case afterwards rebooting and recovering in the normal way. This is strange behavior, because Pasture’s design is based on the idea of an append-only log, and this bug permits the adversary to retract some entries from the end of the log.

But although the behavior is strange, it turns out that there is no actual safety violation. The additional extensions performed by the adversary cannot be used to obtain access to any keys or generate any verifiable proofs of revocation, since PCR_{SEAL} will no longer contain its original *SealReboot* value after the first transport session runs. The entries that the adversary can retract from the log are merely “phantom” entries that do not correspond to any effective decision.

A similar situation exists in the case of *BugAudit-NoCheckSeal*. This bug permits the adversary to add entries to the end of the log as shown by one audit which a later audit will show as having been retracted. But the retractable entries are “phantom” entries that do not correspond to obtaining access to keys or to generating verifiable proofs of revocation.

Originally, when we placed bugs into the specification, we assumed that they all would lead to safety violations. But in three cases this assumption turned out to be mistaken. One benefit of model checking with known bugs is a better understanding of what actually makes the specification work.

6 Formal proof of correctness

Once we were confident that the Pasture node specification was correct, we proceeded to write a formal correctness proof and check it using the TLA+ Proof System [2].

Appendix C shows the proof. Since the TLA+ Proof System currently cannot handle temporal reasoning, we had to check manually the final step that proves that an invariant always holds. We also omitted numerous tedious

proofs about properties of sequences.

The proof is based on the idea that there is always at most one current log and the current log can be domiciled in at most one of three places:

- The current log can be domiciled in PCR_{APP} , when $\text{PCR}_{\text{SEM}} = \textit{SemHappy}$ and $\text{PCR}_{\text{SEAL}} = \textit{SealReboot}$. This is the situation when the Pasture node is operational and processing decisions to obtain access or revoke access to Pasture decryption keys.
- The current log can be domiciled in a SEAL attestation, when the SEAL quotes $\text{PCR}_{\text{SEM}} = \textit{SemHappy}$, $\text{PCR}_{\text{SEAL}} = \textit{SealReboot}$, and the current boot counter. This is the situation during shutdown after the SEAL transport session has been run but before the SEM checkpoint routine is invoked.
- The current log can be domiciled in the NVRAM, when the *current* flag is set. This is the situation after shutdown before the node reboots.

The proof wraps this idea up into one master invariant called *InvOneLog*.

In order to establish *InvOneLog*, the proof first establishes a number of preliminary invariants showing that all variables contain values of the correct type, that PCR_{SEM} and PCR_{SEAL} are managed properly, that SEAL attestations quote a reasonable boot counter value, and that the contents of the fiduciary variables *obtains* and *revokes* make sense. Once the master invariant *InvOneLog* is established, the Pasture safety invariants *InvAccessUndeniability* and *InvVerifiableRevocation* follow as corollaries.

Most of the proof is consumed with walking each invariant through all of the action alternatives. Although tedious, writing the proof was straightforward. Counting the time it took to learn how to use the TLA+ Proof System, the proof took two weeks to write. Interestingly, the seL4 microkernel verification project (a far larger effort) also found that invariant reasoning dominated their proof effort [4].

As a side note, the two “phantom” entry non-safety-violation bugs discussed in Section 5.2 each violate the invariant *InvOneLog* since they permit different versions of the current log to exist at the same time. This shows that the proof is stronger than strictly necessary to establish the correctness of the Pasture node specification.

However, weakening the proof to account for this seems like it would add considerable detail to an already tedious proof.

The Pasture node specification runs 19 pages and the formal proof 68 pages. Memoir also used TLA+ and the TLA+ Proof System and their specification runs 40 pages and formal proof 350 pages [3]. The seL4 project used Haskell and Isabelle and took about 2 person-years to create the specification and 11 person-years to create the proof [4]. So even though a formal specification can be somewhat lengthy, a formal proof of its correctness tends to be much more lengthy.

7 Optimized Pasture

After writing, checking, and proving the correctness of the Pasture specification, we realized that the only function of PCR_{SEAL} was to indicate that a SEAL transport session had been performed and therefore PCR_{APP} no longer contained the current log. However, it ought to be possible to use PCR_{SEM} to implement this indication. Extending PCR_{SEM} inside the SEAL transport session would destroy *SemHappy* and have the same effect as using PCR_{SEAL} .

The only question we had to answer was whether or not the TPM client interface permitted code outside of secure execution mode to extend PCR_{SEM} . TPMs have locality restrictions that prevent extending certain PCRs at certain times. Fortunately, according to the current TPM PC client interface specification [13], the secure execution mode PCR can be extended in hypervisor or operating system code.

Figure 4 shows the implementation of operations in optimized Pasture. The changes from the initial Pasture implementation are simple: merely delete all references to PCR_{SEAL} and make the SEAL transport session extend PCR_{SEM} instead of PCR_{SEAL} .

Although for simplicity we use the same names—such as *SemProtect*, *SemHappy*, *BINDKEY*, and *SEAL*—in optimized Pasture as in initial Pasture, the routines are slightly different and therefore any derived cryptographic hash values will be different. We do not think that it would be useful to try to mix Pasture implementations within the same system. The conference paper [5] limits its attention to optimized Pasture.

CreateBoundKey(*hM*):

```

 $R_t \leftarrow \text{TPM\_Read}(\text{PCR}_{\text{APP}})$ 
 $R_{t+1} \leftarrow \text{SHA1}(R_t \parallel hM)$ 
transport session
  BINDKEY
   $K \leftarrow \text{TPM\_CreateWrapKey}(\{$ 
     $\text{PCR}_{\text{APP}} = R_{t+1} \ \&\&$ 
     $\text{PCR}_{\text{SEM}} = \text{SemHappy} \})$ 
 $\alpha \leftarrow$ 
 $KP \leftarrow \langle \text{"CreateBoundKey"}, hM, R_t, R_{t+1}, \alpha \rangle$ 

```

ObtainAccess(*hM*, *EM*):

```

append  $hM$  to full log
 $\text{TPM\_Extend}(\text{PCR}_{\text{APP}}, hM)$ 
 $M \leftarrow \text{TPM\_Unbind}(EM)$ 

```

RevokeAccess():

```

 $R_t \leftarrow \text{TPM\_Read}(\text{PCR}_{\text{APP}})$ 
append  $\delta$  to full log
 $\text{TPM\_Extend}(\text{PCR}_{\text{APP}}, \delta)$ 
 $R'_{t+1}, S'_{t+1}, \alpha \leftarrow \text{TPM\_Quote}(\text{PCR}_{\text{APP}}, \text{PCR}_{\text{SEM}})$ 
 $RP \leftarrow \langle \text{"RevokeAccess"}, \delta, R_t, R'_{t+1}, S'_{t+1}, \alpha \rangle$ 

```

Audit(*nonce*):

```

 $R_t, S_t, \alpha \leftarrow$ 
 $\text{TPM\_Quote}(\text{PCR}_{\text{APP}}, \text{PCR}_{\text{SEM}}, \text{nonce})$ 
 $AP \leftarrow \langle \text{"Audit"}, \text{full log}, R_t, S_t, \text{nonce}, \alpha \rangle$ 

```

Recover():

```

FOR EACH entry  $\Delta$  on full log:  $\text{TPM\_Extend}(\text{PCR}_{\text{APP}}, \Delta)$ 
secure execution mode
  IF  $\text{nv.current} \ \&\& \ \text{nv.R} = \text{TPM\_Read}(\text{PCR}_{\text{APP}})$ 
  THEN
     $\text{nv.current} \leftarrow \text{FALSE}$ 
     $\text{TPM\_Extend}(\text{PCR}_{\text{SEM}}, \text{Happy})$ 
  ELSE
     $\text{TPM\_Extend}(\text{PCR}_{\text{SEM}}, \text{Unhappy})$ 

```

Checkpoint():

```

transport session
  SEAL
   $R_t \leftarrow \text{TPM\_Read}(\text{PCR}_{\text{APP}})$ 
   $S_t \leftarrow \text{TPM\_Read}(\text{PCR}_{\text{SEM}})$ 
   $C_t \leftarrow \text{TPM\_ReadCounter}(CTR)$ 
   $\alpha \leftarrow \text{TPM\_Extend}(\text{PCR}_{\text{SEM}}, \text{Seal})$ 
secure execution mode
   $\text{nv.R} \leftarrow R_t$ 
  IF  $\text{Valid}_{\text{SEAL}}(\alpha, R_t, S_t, C_t)$ 
  &&  $S_t = \text{SemHappy}$ 
  &&  $C_t = \text{TPM\_ReadCounter}(CTR)$ 
  THEN
     $\text{TPM\_IncrementCounter}(CTR)$ 
     $\text{nv.current} \leftarrow \text{TRUE}$ 
     $\text{TPM\_Extend}(\text{PCR}_{\text{SEM}}, \text{Unhappy})$ 

```

Figure 4: Optimized Pasture operations.

Appendix D gives a TLA+ specification of the state within a Pasture node according to optimized Pasture. This specification was created by a simple edit of the initial specification.

To prove the correctness of optimized Pasture, one could prove a refinement mapping showing that every action in optimized Pasture corresponds to an action in initial Pasture. However, since the change in the specification was slight, it was easier just to edit the initial correctness proof.

Appendix E shows the formal proof of correctness for optimized Pasture. Basically, all that was required in editing the initial proof was to delete all references to PCR_{SEAL} and to rework proof steps in three places regarding the changed NextSealTs action. The hierarchical structure of TLA+ proofs was a great benefit here, because the TLA+ Proof System and toolbox made it easy to find the places that had to be changed. Except for temporal reasoning, which the TLA+ Proof System currently cannot handle, and numerous tedious proofs about properties of sequences, all steps in the reworked proof have been mechanically checked.

8 Conclusion

Model checking with inserted bugs provides reasonable confidence that the specification is correct. Examining the counterexample execution traces can lead to improved understanding of the specification and possible improvements. For example, in the Pasture specification, we discovered that the specification could be made weaker, with the incorporation of two “phantom” entry bugs, and still maintain its invariants. However, weakening the specification in this way would make it much more tedious to prove that the invariants were maintained.

Formal proofs give a greater assurance, but they can be tedious. An enormous amount of detail is required to guide a mechanical proof checker through the verification process. When formal proofs of safety are important, it can sometimes be better to adopt a stronger specification than strictly necessary in order to make maintenance of the safety properties easier to prove.

With the TLA+ proof system, the same specification can be both model checked and augmented with a mechanically checked proof. This gives even more confi-

dence that the specification is correct.

Given an existing formal proof of correctness, the TLA+ hierarchical proof structure made it surprisingly easy to create a proof of correctness for a slightly revised specification. A mechanical proof checker gives assurance that the revised proof is correct even though very little intellectual effort had to be expended to create the revised proof.

References

- [1] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, Jan 2008.
- [2] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, pages 142–148, 2010.
- [3] J. R. Douceur, J. R. Lorch, B. Parno, J. Mickens, and J. M. McCune. Memoir—formal specs and correctness proofs. Technical Report MSR-TR-2011-19, Microsoft Research, Feb 2011.
- [4] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.
- [5] R. Kotla, T. Rodeheffer, I. Roy, P. Steudi, and B. Wester. Secure offline data access using commodity trusted hardware. To appear in *OSDI*, 2012.
- [6] L. Lamport. The TLA toolbox. <http://research.microsoft.com/en-us/people/lamport/tla/toolbox.html>.
- [7] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
- [8] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Eurosys*, 2008.

- [9] B. Parno, J. R. Lorch, J. R. Douceur, J. Mickens, and J. M. McCune. Memoir: Practical state continuity for protected modules. In *IEEE Symposium on Security and Privacy*, 2011.
- [10] T. Rodeheffer. Cyclic commit protocol specifications. Technical Report MSR-TR-2008-125, Microsoft Research, Sep 2008. <http://research.microsoft.com/apps/pubs/?id=70631>.
- [11] T. L. Rodeheffer and R. Kotla. Inserting intentional bugs for model checking assurance. To appear in *International Workshop on the TLA+ Method and Tools*, 2012, held in conjunction with *FM*, 2012.
- [12] Trusted Computing Group, Inc. Trusted Platform Module V1.2 Specification. <http://www.trustedcomputinggroup.org>.
- [13] Trusted Computing Group, Inc. TCG PC client specific TPM interface specification (TIS), 2005. http://www.trustedcomputinggroup.org/files/resource_files/87BCE22B-1D09-3519-ADEBA772F02CBD/TCG_PCClientTPMSpecification_1-20_1-00_FINAL.pdf.

A Specification of Initial Pasture

MODULE *PastureNode*

EXTENDS *Naturals, Sequences, FiniteSets*

Override one of the following definitions to introduce a bug in the specification.

$BugObtainAccessNoCheckHappy \triangleq FALSE$
 $BugObtainAccessNoCheckSeal \triangleq FALSE$
 $BugProveRevokeNoCheckHappy \triangleq FALSE$
 $BugProveRevokeNoCheckSeal \triangleq FALSE$
 $BugRecovNoCheckApp \triangleq FALSE$
 $BugRecovNoCheckCur \triangleq FALSE$
 $BugRecovNoClrCur \triangleq FALSE$
 $BugSealNoExt \triangleq FALSE$
 $BugChkptNoCheckTsHappy \triangleq FALSE$
 $BugChkptNoCheckTsSeal \triangleq FALSE$ not actually a safety bug
 $BugChkptNoCheckTsCtr \triangleq FALSE$
 $BugChkptSaveCurApp \triangleq FALSE$
 $BugChkptNoIncCtr \triangleq FALSE$
 $BugChkptNoSetCur \triangleq FALSE$ liveness bug; not actually a safety bug
 $BugAuditNoCheckHappy \triangleq FALSE$
 $BugAuditNoCheckSeal \triangleq FALSE$ not actually a safety bug

PCR INITIALIZATION VALUES

CONSTANT $PcriAPPBOOT$ reboot initialization of app pcr
 CONSTANT $PcriSEMBOOT$ reboot initialization of sem pcr
 CONSTANT $PcriSEMPROTECT$ secure execution mode entry of sem pcr
 CONSTANT $PcriSEALBOOT$ reboot initialization of seal pcr

$Pcri \triangleq$
 {
 $PcriAPPBOOT$,
 $PcriSEMBOOT$,
 $PcriSEMPROTECT$,
 $PcriSEALBOOT$
 }

Initialization of sem pcr via boot and via secure execution mode entry must be different.

ASSUME $AssSemProtect \triangleq PcriSEMBOOT \neq PcriSEMPROTECT$

PCR EXTENSION VALUES

CONSTANT $PcrxHAPPY$ recover is happy
 CONSTANT $PcrxUNHAPPY$ recover is unhappy or checkpoint is unhappy/finished
 CONSTANT $PcrxSEAL$ seal marker
 CONSTANT $PcrxOBTAIN$ obtain access operation
 CONSTANT $PcrxREVOKE$ revoke access operation

$$Pcrx \triangleq \{ \begin{array}{l} PcrxHAPPY, \\ PcrxUNHAPPY, \\ PcrxSEAL, \\ PcrxOBTAIN, \\ PcrxREVOKE \end{array} \}$$

Extension for obtain access and extension for revoke access must be different.

$$\text{ASSUME } AssObtainNeqRevoke \triangleq PcrxOBTAIN \neq PcrxREVOKE$$

Extension for happy and extension for unhappy must be different.

$$\text{ASSUME } AssSemHappy \triangleq PcrxHAPPY \neq PcrxUNHAPPY$$

PCR VALUES

A pcr value is modeled as an initialization followed by a sequence of extensions.

$$Pcr \triangleq [\begin{array}{l} init : Pcri, \\ extq : Seq(Pcrx) \end{array}]$$

Initial pcr value.

$$PcrInit(i) \triangleq [\begin{array}{l} init \mapsto i, \\ extq \mapsto \langle \rangle \end{array}]$$

Extend a pcr value.

$$\begin{aligned} PcrExtend(p, x) &\triangleq \\ &[\\ &\quad init \mapsto p.init, \\ &\quad extq \mapsto Append(p.extq, x) \\ &] \end{aligned}$$

Number of extensions in a pcr value.

$$\begin{aligned} PcrLen(p) &\triangleq \\ &Len(p.extq) \end{aligned}$$

$Pcr\ s$ is $\leq Pcr\ t$. This means that with zero or more extensions, you can extend s to reach t . This is a partial order relation.

$$\begin{aligned} PcrLeq(s, t) &\triangleq \\ &LET \\ &\quad sinit \triangleq s.init \\ &\quad sextq \triangleq s.extq \\ &\quad sn \triangleq Len(sextq) \\ &\quad \\ &\quad tinit \triangleq t.init \\ &\quad textq \triangleq t.extq \\ &\quad tn \triangleq Len(textq) \\ &\quad \\ &\quad uextq \triangleq SubSeq(textq, 1, sn) \\ &IN \\ &\quad \wedge sinit = tinit \\ &\quad \wedge sn \leq tn \\ &\quad \wedge sextq = uextq \end{aligned}$$

Determine if a pcr value has been extended.

$$\begin{aligned} PcrHasExtension(p) &\triangleq \\ &PcrLen(p) > 0 \end{aligned}$$

Assuming a pcr value has been extended, get the prior pcr value that this one was extended from. We assume the adversary can compute this by watching all pcr computations.

$$\begin{aligned} PcrPrior(p) &\triangleq \\ &CASE\ PcrHasExtension(p) \rightarrow \\ &LET \\ &\quad n \triangleq Len(p.extq) - 1 \\ &IN \\ &[\\ &\quad init \mapsto p.init, \\ &\quad extq \mapsto SubSeq(p.extq, 1, n) \\ &] \end{aligned}$$

Assuming a pcr value has been extended, get the last extension. We assume the adversary can compute this by watching all pcr computations.

$$PcrLastExtension(p) \triangleq$$

$$\text{CASE } PcrHasExtension(p) \rightarrow$$

$$p.extq[Len(p.extq)]$$

WELL KNOWN PCR VALUES

Value of the application pcr attained by rebooting.

$$AppReboot \triangleq PcrInit(PcriAPPBOOT)$$

Value of the secure execution mode pcr attained by rebooting.

$$SemReboot \triangleq PcrInit(PcriSEMBOOT)$$

Value of the secure execution mode pcr attained by entering the protected module in secure execution mode. This value permits access to the Pasture protected *Nv* ram.

$$SemProtect \triangleq PcrInit(PcriSEMPROTECT)$$

Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.

$$SemHappy \triangleq PcrExtend(SemProtect, PcrxHAPPY)$$

Value of the seal pcr attained by rebooting.

$$SealReboot \triangleq PcrInit(PcriSEALBOOT)$$

PC VALUES

anywhere not in secure execution mode

$$PcIDLE \triangleq \text{"idle"}$$

steps in secure execution mode within recover

$$PcRECOV1 \triangleq \text{"recov1"}$$

$$PcRECOV2 \triangleq \text{"recov2"}$$

$$PcRECOV3 \triangleq \text{"recov3"}$$

$$PcRecov \triangleq \{PcRECOV1, PcRECOV2, PcRECOV3\}$$

steps in secure execution mode within checkpoint

$$PcCHKPT1 \triangleq \text{"chkpt1"}$$

$$\begin{aligned}
PcCHKPT2 &\triangleq \text{"chkpt2"} \\
PcCHKPT3 &\triangleq \text{"chkpt3"} \\
PcCHKPT4 &\triangleq \text{"chkpt4"} \\
PcCHKPT5 &\triangleq \text{"chkpt5"} \\
PcChkpt &\triangleq \{PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5\}
\end{aligned}$$

$$Pc \triangleq \{PcIDLE\} \cup PcRecov \cup PcChkpt$$

PROTECTED NV RAM STATE

$$\begin{aligned}
Nv &\triangleq \\
&[\\
&\quad appPcr : Pcr, \quad \text{copy of the application pcr} \\
&\quad current : BOOLEAN \quad \text{copy of application pcr is current} \\
&]
\end{aligned}$$

$$\begin{aligned}
InitNv &\triangleq \\
&[\\
&\quad appPcr \mapsto AppReboot, \\
&\quad current \mapsto TRUE \\
&]
\end{aligned}$$

SEAL OPERATION TRANSPORT SESSION STATE

We model the signed “seal operation” transport session as a record of the input values required in order for the transport session *TPM* signature to be valid.

$$\begin{aligned}
SignedTs &\triangleq \\
&[\\
&\quad semPcr : Pcr, \quad \text{copy of the secure execution mode pcr on entry} \\
&\quad sealPcr : Pcr, \quad \text{copy of the seal pcr on entry} \\
&\quad appPcr : Pcr, \quad \text{copy of the application pcr on entry} \\
&\quad bootCtr : Nat \quad \text{copy of the reboot counter on entry} \\
&]
\end{aligned}$$

The adversary cannot forge a correctly signed seal attestation. We model all incorrectly signed ones as the following single value.

$$NullTs \triangleq \text{CHOOSE } NullTs : NullTs \notin SignedTs$$

$$Ts \triangleq SignedTs \cup \{NullTs\}$$

STATE

VARIABLE <i>nv</i>	Pasture's protected NV RAM region
VARIABLE <i>appPcr</i>	the application pcr
VARIABLE <i>semPcr</i>	the secure execution mode pcr
VARIABLE <i>sealPcr</i>	the seal pcr
VARIABLE <i>bootCtr</i>	the reboot counter
VARIABLE <i>pc</i>	<i>pc</i>
VARIABLE <i>chkptts</i>	<i>ts</i> passed to sem within checkpoint
VARIABLE <i>tsvalues</i>	what <i>ts</i> values are known
VARIABLE <i>obtains</i>	decisions to obtain access
VARIABLE <i>revokes</i>	decisions to prove revoke access

Tuple of all variables.

$$vars \triangleq \langle nv, appPcr, semPcr, sealPcr, bootCtr, pc, chkptts, tsvalues, obtains, revokes \rangle$$

STATE PREDICATES

The node is currently in secure execution mode.

$$InSem \triangleq pc \neq PcIDLE$$

NEXT STATE RELATION

Employ a key binding to obtain access to read a message.

If the last extension to the application pcr was an OBTAIN operation, then in full generality there could have been a key bound to this application pcr value. So record the information that we obtained access to this key binding.

$NextObtainAccess \triangleq$

LET

$pcr1o \triangleq appPcr$ current app pcr

$pcr0 \triangleq PcrPrior(pcr1o)$ prior app pcr

$x \triangleq PcrLastExtension(pcr1o)$ presumed OBTAIN extension

IN

$\wedge \neg InSem$ must not be in secure execution mode

$\wedge PcrHasExtension(pcr1o)$ have an extension

$\wedge x = PcrxOBTAIN$ last extension was OBTAIN

It is a bug to fail to bind the key such that it can be used for decryption only when the secure execution mode pcr is happy.

\wedge IF $BugObtainAccessNoCheckHappy$ THEN TRUE ELSE

$semPcr = SemHappy$

It is a bug to fail to bind the key such that it can be used for decryption only when the seal pcr is in the reboot value.

\wedge IF $BugObtainAccessNoCheckSeal$ THEN TRUE ELSE

$sealPcr = SealReboot$

$\wedge obtains' = obtains \cup \{pcr1o\}$

\wedge UNCHANGED nv

\wedge UNCHANGED $appPcr$

\wedge UNCHANGED $semPcr$

\wedge UNCHANGED $sealPcr$

\wedge UNCHANGED $bootCtr$

\wedge UNCHANGED pc

\wedge UNCHANGED $chkptts$

\wedge UNCHANGED $tvalues$

\wedge UNCHANGED $revokes$

Construct a proof of revocation.

If the last extension to the application pcr was a REVOKE operation, then in full generality there could have been a key bound to the pcr value in which instead the last extension was an OBTAIN. But by extending with a REVOKE we have instead revoked the key binding. So record the information that we could construct a proof of revocation.

A proof of revocation consists of the following exhibits:

- (a) $pcr0$, a purported prior application pcr value
- (b) $pcr1r$, a purported current application pcr value
- (c) x , the REVOKE extension satisfying $pcr1r = PcrExtend(pcr0, x)$
- (d) a quote of the application $pcr = pcr1r$ with the $sem\ pcr = SemHappy$ and the seal $pcr = SealReboot$.

These exhibits suffice to prove revocation of any valid key binding to the application pcr value $PcrExtend(pcr0, OBTAIN)$.

$NextProveRevoke \triangleq$

LET

$pcr1r \triangleq appPcr$ current app pcr

$pcr0 \triangleq PcrPrior(pcr1r)$ prior app pcr
 $x \triangleq PcrLastExtension(pcr1r)$ presumed REVOKE extension
 IN
 $\wedge \neg InSem$ must not be in secure execution mode
 $\wedge PcrHasExtension(pcr1r)$ have an extension
 $\wedge x = PcrxREVOKE$ last extension was a REVOKE
 It is a bug to fail to require the proof of revocation to quote the fact that the secure execution mode pcr is happy.
 \wedge IF *BugProveRevokeNoCheckHappy* THEN TRUE ELSE
 $semPcr = SemHappy$
 It is a bug to fail to require the proof of revocation to quote the fact that the seal pcr is in the reboot value.
 \wedge IF *BugProveRevokeNoCheckSeal* THEN TRUE ELSE
 $sealPcr = SealReboot$
 $\wedge revokes' = revokes \cup \{pcr1r\}$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *sealPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*

Reboot the node.

This can happen at absolutely any time, due to adversarial action. However, if it happens without going through the proper seal and checkpoint actions, liveness may be lost.

Resetting *chkptts* to its initial value erases information and thus reduces the number of distinct states that model checking has to explore. But note that the adversary could always remember whatever value *chkptts* had before and call *sem* checkpoint with that value.

$NextReboot \triangleq$
 $\wedge appPcr' = AppReboot$
 $\wedge semPcr' = SemReboot$
 $\wedge sealPcr' = SealReboot$
 $\wedge pc' = PcIDLE$
 $\wedge chkptts' = NullTs$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Forget one of the seal transport sessions.

This can happen at absolutely any time, and represents a loss of knowledge by the adversary which enables additional execution paths to fall within the model checking constraints.

$$\begin{aligned}
\text{NextForgetSealTs} &\triangleq \\
&\exists ts \in \text{tsvalues} : \\
&\wedge \text{tsvalues}' = \text{tsvalues} \setminus \{ts\} \\
&\wedge \text{UNCHANGED } nv \\
&\wedge \text{UNCHANGED } \text{appPcr} \\
&\wedge \text{UNCHANGED } \text{semPcr} \\
&\wedge \text{UNCHANGED } \text{sealPcr} \\
&\wedge \text{UNCHANGED } \text{bootCtr} \\
&\wedge \text{UNCHANGED } pc \\
&\wedge \text{UNCHANGED } \text{chkptts} \\
&\wedge \text{UNCHANGED } \text{obtains} \\
&\wedge \text{UNCHANGED } \text{revokes}
\end{aligned}$$

Extend application per arbitrarily.

In proper execution, this action is performed as necessary after reboot to re-extend the application per to its last checkpoint value.

In proper execution, this action is performed as desired to decide upon reading or deleting messages.

The adversary can perform this action at any idle time.

$$\begin{aligned}
\text{NextExtendAppPcr} &\triangleq \\
&\wedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
&\wedge \exists x \in \text{Pcrx} : \\
&\wedge \text{appPcr}' = \text{PcrExtend}(\text{appPcr}, x) \\
&\wedge \text{UNCHANGED } nv \\
&\wedge \text{UNCHANGED } \text{semPcr} \\
&\wedge \text{UNCHANGED } \text{sealPcr} \\
&\wedge \text{UNCHANGED } \text{bootCtr} \\
&\wedge \text{UNCHANGED } pc \\
&\wedge \text{UNCHANGED } \text{chkptts} \\
&\wedge \text{UNCHANGED } \text{tsvalues} \\
&\wedge \text{UNCHANGED } \text{obtains} \\
&\wedge \text{UNCHANGED } \text{revokes}
\end{aligned}$$

Extend secure execution mode per arbitrarily, due to adversarial action.

$$\begin{aligned}
\text{NextExtendSemPcr} &\triangleq \\
&\wedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
&\wedge \exists x \in \text{Pcrx} : \\
&\wedge \text{semPcr}' = \text{PcrExtend}(\text{semPcr}, x) \\
&\wedge \text{UNCHANGED } nv
\end{aligned}$$

\wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *sealPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Extend seal pcr arbitrarily, due to adversarial action.

NextExtendSealPcr \triangleq

$\wedge \neg InSem$ must not be in secure execution mode
 $\wedge \exists x \in Pcrx :$
 $\wedge sealPcr' = PcrExtend(sealPcr, x)$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Increment reboot counter arbitrarily, due to adversarial action.

NextIncBootCtr \triangleq

$\wedge \neg InSem$ must not be in secure execution mode
 $\wedge bootCtr' = bootCtr + 1$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *sealPcr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Enter secure execution mode within recovery.

In proper execution, this action is performed during system boot after the application pcr has been re-extended to its last checkpoint value. This re-extension is performed by untrusted code that reads the necessary extension values from a stable log.

The adversary can perform this action at any idle time. But it will not do any good unless the application pcr contains the last checkpoint value and the last checkpoint value is marked as current.

$NextEnterSemRecov \triangleq$

$\wedge \neg InSem$ must not be in secure execution mode
 $\wedge semPcr' = SemProtect$
 $\wedge pc' = PcRECOV1$
 $\wedge UNCHANGED nv$
 $\wedge UNCHANGED appPcr$
 $\wedge UNCHANGED sealPcr$
 $\wedge UNCHANGED bootCtr$
 $\wedge UNCHANGED chkptts$
 $\wedge UNCHANGED tsvalues$
 $\wedge UNCHANGED obtains$
 $\wedge UNCHANGED revokes$

Predicate for correct entry to secure execution mode within recovery.

$EnterSemRecovPredicate \triangleq$

It is a bug for recovery to fail to check that the application pcr has been restored to the value saved in the *nv* ram.
 \wedge IF *BugRecovNoCheckApp* THEN TRUE ELSE
 $nv.appPcr = appPcr$
It is a bug for recovery to fail to check that *nv* ram claims that its saved application pcr is current.
 \wedge IF *BugRecovNoCheckCur* THEN TRUE ELSE
 $nv.current$

Secure execution mode within recovery step 1, when there is correct entry.

$NextSemRecov1WhenCorrect \triangleq$

$\wedge pc = PcRECOV1$
 $\wedge EnterSemRecovPredicate$
 $\wedge pc' = PcRECOV2$
 $\wedge UNCHANGED nv$
 $\wedge UNCHANGED appPcr$
 $\wedge UNCHANGED semPcr$
 $\wedge UNCHANGED sealPcr$
 $\wedge UNCHANGED bootCtr$
 $\wedge UNCHANGED chkptts$
 $\wedge UNCHANGED tsvalues$
 $\wedge UNCHANGED obtains$
 $\wedge UNCHANGED revokes$

Secure execution mode within recovery step 1, when there is incorrect entry.

$$\begin{aligned} \text{NextSemRecov1WhenIncorrect} &\triangleq \\ &\wedge pc = PcRECOV1 \\ &\wedge \neg \text{EnterSemRecovPredicate} \\ &\wedge semPcr' = PcrExtend(semPcr, PcrxUNHAPPY) \\ &\wedge pc' = PcIDLE \\ &\wedge \text{UNCHANGED } nv \\ &\wedge \text{UNCHANGED } appPcr \\ &\wedge \text{UNCHANGED } sealPcr \\ &\wedge \text{UNCHANGED } bootCtr \\ &\wedge \text{UNCHANGED } chkptts \\ &\wedge \text{UNCHANGED } tsvalues \\ &\wedge \text{UNCHANGED } obtains \\ &\wedge \text{UNCHANGED } revokes \end{aligned}$$

Secure execution mode within recovery step 2. Record that the *nv* app pcr might no longer be current.

$$\begin{aligned} \text{NextSemRecov2} &\triangleq \\ &\text{LET} \\ &\quad nvcurent1 \triangleq \\ &\quad \quad \text{It is a bug for recovery to fail to clear the } nv \text{ ram current flag.} \\ &\quad \quad \text{IF } BugRecovNoClrCur \text{ THEN } nv.current \text{ ELSE} \\ &\quad \quad \text{FALSE} \\ &\text{IN} \\ &\wedge pc = PcRECOV2 \\ &\wedge nv' = [nv \text{ EXCEPT } !.current = nvcurent1] \\ &\wedge pc' = PcRECOV3 \\ &\wedge \text{UNCHANGED } appPcr \\ &\wedge \text{UNCHANGED } semPcr \\ &\wedge \text{UNCHANGED } sealPcr \\ &\wedge \text{UNCHANGED } bootCtr \\ &\wedge \text{UNCHANGED } chkptts \\ &\wedge \text{UNCHANGED } tsvalues \\ &\wedge \text{UNCHANGED } obtains \\ &\wedge \text{UNCHANGED } revokes \end{aligned}$$

Secure execution mode within recovery step 3. Declare correct recovery happiness and exit secure execution mode.

$$\begin{aligned} \text{NextSemRecov3} &\triangleq \\ &\wedge pc = PcRECOV3 \\ &\wedge semPcr' = PcrExtend(semPcr, PcrxHAPPY) \\ &\wedge pc' = PcIDLE \\ &\wedge \text{UNCHANGED } nv \\ &\wedge \text{UNCHANGED } appPcr \\ &\wedge \text{UNCHANGED } sealPcr \end{aligned}$$

\wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Perform a “seal operation” and remember the signed transport session.

In proper execution, provided that the secure execution mode pcr shows that recovery was happy, this action is performed as part of checkpoint during system shutdown. Secure execution mode within checkpoint is then invoked with this transport session as data.

This transport session reads the values of the secure execution mode pcr, the application pcr, the seal pcr, and the reboot counter. Then the seal pcr is extended so that no key bindings will be available until the next happy recovery.

The adversary can record all of the signed transport sessions and try to replay an earlier one to convince secure execution mode within checkpoint to save an old application pcr as “current”. Reading the reboot counter here, and incrementing it in secure execution mode within checkpoint, prevents that.

The adversary might try to advance the application pcr so as to read a message or produce a proof of deletion after the “seal operation” and then invoke secure execution mode within checkpoint and then reboot to roll back the application pcr. Extending the seal pcr prevents that.

NextSealTs \triangleq

LET
 ts \triangleq
 [
 semPcr \mapsto *semPcr*, sem pcr on entry
 appPcr \mapsto *appPcr*, app pcr on entry
 sealPcr \mapsto *sealPcr*, seal pcr on entry
 bootCtr \mapsto *bootCtr* reboot ctr on entry
]

sealPcr1 \triangleq

It is a bug for the “seal operation” to fail to extend the seal pcr.

IF *BugSealNoExt* THEN *sealPcr* ELSE
 PcrExtend(*sealPcr*, *Pcr*:*SEAL*)

IN

\wedge \neg *InSem* must not be in secure execution mode
 \wedge *tvalues'* = *tvalues* \cup {*ts*}
 \wedge *sealPcr'* = *sealPcr1*
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Enter secure execution mode within checkpoint.

In proper execution, this action is performed during system shutdown following the seal transport session action.

The adversary can perform this action at any idle time, feeding it any known seal transport session value.

$NextEnterSemChkpt \triangleq$

$\wedge \neg InSem$ must not be in secure execution mode
 $\wedge \exists ts \in tsvalues :$ any known ts value
 $\wedge semPcr' = SemProtect$
 $\wedge pc' = PcCHKPT1$
 $\wedge chkptts' = ts$
 $\wedge UNCHANGED nv$
 $\wedge UNCHANGED appPcr$
 $\wedge UNCHANGED sealPcr$
 $\wedge UNCHANGED bootCtr$
 $\wedge UNCHANGED tsvalues$
 $\wedge UNCHANGED obtains$
 $\wedge UNCHANGED revokes$

Predicate for correct entry to secure execution mode within checkpoint.

$EnterSemChkptPredicate \triangleq$

$\wedge chkptts \in SignedTs$

It is a bug to fail to check that the seal operation recorded that the secure execution mode pcr was happy.

$\wedge IF BugChkptNoCheckTsHappy THEN TRUE ELSE$
 $chkptts.semPcr = SemHappy$

It is a bug to fail to check that the seal operation recorded that the seal pcr was in the reboot value.

$\wedge IF BugChkptNoCheckTsSeal THEN TRUE ELSE$
 $chkptts.sealPcr = SealReboot$

It is a bug to fail to check that the seal operation recorded a reboot counter value that matches the current reboot counter.

$\wedge IF BugChkptNoCheckTsCtr THEN TRUE ELSE$
 $chkptts.bootCtr = bootCtr$

Secure execution mode within checkpoint step 1, when there is correct entry.

$NextSemChkpt1WhenCorrect \triangleq$

$\wedge pc = PcCHKPT1$
 $\wedge EnterSemChkptPredicate$
 $\wedge pc' = PcCHKPT2$
 $\wedge UNCHANGED nv$
 $\wedge UNCHANGED appPcr$
 $\wedge UNCHANGED semPcr$
 $\wedge UNCHANGED sealPcr$
 $\wedge UNCHANGED bootCtr$
 $\wedge UNCHANGED chkptts$

\wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Secure execution mode within checkpoint step 1, when there is incorrect entry.

NextSemChkpt1WhenIncorrect \triangleq
 \wedge *pc* = *PcCHKPT1*
 \wedge \neg *EnterSemChkptPredicate*
 \wedge *semPcr'* = *PcrExtend*(*semPcr*, *PcrxUNHAPPY*)
 \wedge *pc'* = *PcIDLE*
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *sealPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Secure execution mode within checkpoint step 2. Save in *nv appPtr* the app ptr recorded at *ts* entry.

NextSemChkpt2 \triangleq
 LET
 nvappPcr1 \triangleq
 It is a bug for secure execution mode within checkpoint to save in the *nv* ram the current application pcr rather than the seal operation's recorded application pcr.
 IF *BugChkptSaveCurApp* THEN *appPcr* ELSE
 chkptts.appPcr
 IN
 \wedge *pc* = *PcCHKPT2*
 \wedge *nv'* = [*nv* EXCEPT !.*appPcr* = *nvappPcr1*]
 \wedge *pc'* = *PcCHKPT3*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *sealPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Secure execution mode within checkpoint step 3. Prevent a *ts* replay by incrementing the reboot ctr.

$NextSemChkpt3 \triangleq$

LET

$bootCtr1 \triangleq$

It is a bug for secure execution mode within checkpoint to fail to increment the reboot counter.

IF $BugChkptNoIncCtr$ THEN $bootCtr$ ELSE
 $bootCtr + 1$

IN

$\wedge pc = PcCHKPT3$
 $\wedge bootCtr' = bootCtr1$
 $\wedge pc' = PcCHKPT4$
 $\wedge UNCHANGED nv$
 $\wedge UNCHANGED appPcr$
 $\wedge UNCHANGED semPcr$
 $\wedge UNCHANGED sealPcr$
 $\wedge UNCHANGED chkptts$
 $\wedge UNCHANGED tsvalues$
 $\wedge UNCHANGED obtains$
 $\wedge UNCHANGED revokes$

Secure execution mode within checkpoint step 4. Declare that the $nv appPcr$ is current so that after reboot recovery will be able to succeed.

$NextSemChkpt4 \triangleq$

LET

$nvcurrent1 \triangleq$

It is a bug for secure execution mode within checkpoint to fail to set the NV RAM current flag.

Actually, this bug does not result in a safety violation.

IF $BugChkptNoSetCur$ THEN $nv.current$ ELSE
 TRUE

IN

$\wedge pc = PcCHKPT4$
 $\wedge nv' = [nv \text{ EXCEPT } !.current = nvcurrent1]$
 $\wedge pc' = PcCHKPT5$
 $\wedge UNCHANGED appPcr$
 $\wedge UNCHANGED semPcr$
 $\wedge UNCHANGED sealPcr$
 $\wedge UNCHANGED bootCtr$
 $\wedge UNCHANGED chkptts$
 $\wedge UNCHANGED tsvalues$
 $\wedge UNCHANGED obtains$
 $\wedge UNCHANGED revokes$

Secure execution mode within checkpoint step 5. Extend sem pcr with unhappy so protected nv ram will be inaccessible.

$NextSemChkpt5 \triangleq$

$\wedge pc = PcCHKPT5$

$$\begin{aligned}
&\wedge \text{semPcr}' = \text{PcrExtend}(\text{semPcr}, \text{PcrxUNHAPPY}) \\
&\wedge \text{pc}' = \text{PcIDLE} \\
&\wedge \text{UNCHANGED } nv \\
&\wedge \text{UNCHANGED } \text{appPcr} \\
&\wedge \text{UNCHANGED } \text{sealPcr} \\
&\wedge \text{UNCHANGED } \text{bootCtr} \\
&\wedge \text{UNCHANGED } \text{chkptts} \\
&\wedge \text{UNCHANGED } \text{tsvalues} \\
&\wedge \text{UNCHANGED } \text{obtains} \\
&\wedge \text{UNCHANGED } \text{revokes}
\end{aligned}$$

SPECIFICATION

$$\begin{aligned}
\text{Init} &\triangleq \\
&\wedge nv = \text{InitNv} \\
&\wedge \text{appPcr} = \text{AppReboot} \\
&\wedge \text{semPcr} = \text{SemReboot} \\
&\wedge \text{sealPcr} = \text{SealReboot} \\
&\wedge \text{bootCtr} = 0 \\
&\wedge pc = \text{PcIDLE} \\
&\wedge \text{chkptts} = \text{NullTs} \\
&\wedge \text{tsvalues} = \{\text{NullTs}\} \quad \text{anybody can create a NullTs} \\
&\wedge \text{obtains} = \{\} \\
&\wedge \text{revokes} = \{\}
\end{aligned}$$

$$\begin{aligned}
\text{Next} &\triangleq \\
&\vee \text{NextObtainAccess} \\
&\vee \text{NextProveRevoke} \\
&\vee \text{NextReboot} \\
&\vee \text{NextForgetSealTs} \\
&\vee \text{NextExtendAppPcr} \\
&\vee \text{NextExtendSemPcr} \\
&\vee \text{NextExtendSealPcr} \\
&\vee \text{NextIncBootCtr} \\
&\vee \text{NextEnterSemRecov} \\
&\vee \text{NextSemRecov1 WhenCorrect} \\
&\vee \text{NextSemRecov1 WhenIncorrect} \\
&\vee \text{NextSemRecov2} \\
&\vee \text{NextSemRecov3}
\end{aligned}$$

$\vee \text{NextSealTs}$
 $\vee \text{NextEnterSemChkpt}$
 $\vee \text{NextSemChkpt1 WhenCorrect}$
 $\vee \text{NextSemChkpt1 WhenIncorrect}$
 $\vee \text{NextSemChkpt2}$
 $\vee \text{NextSemChkpt3}$
 $\vee \text{NextSemChkpt4}$
 $\vee \text{NextSemChkpt5}$

$\text{Spec} \triangleq \text{Init} \wedge \square[\text{Next}]_{\text{vars}}$

INVARIANTS

Type invariant.

$\text{InvType} \triangleq$
 $\wedge \text{nv}:: \text{nv} \in \text{Nv}$
 $\wedge \text{appPcr}:: \text{appPcr} \in \text{Pcr}$
 $\wedge \text{semPcr}:: \text{semPcr} \in \text{Pcr}$
 $\wedge \text{sealPcr}:: \text{sealPcr} \in \text{Pcr}$
 $\wedge \text{bootCtr}:: \text{bootCtr} \in \text{Nat}$
 $\wedge \text{pc}:: \text{pc} \in \text{Pc}$
 $\wedge \text{chkptts}:: \wedge \text{chkptts} \in \text{Ts}$
 $\wedge \text{pc} \in \text{PcChkpt} \setminus \{\text{PcCHKPT1}\} \Rightarrow \text{chkptts} \in \text{SignedTs}$
 $\wedge \text{tsvalues}:: \text{tsvalues} \in \text{SUBSET Ts}$
 $\wedge \text{obtains}:: \text{obtains} \in \text{SUBSET Pcr}$
 $\wedge \text{revokes}:: \text{revokes} \in \text{SUBSET Pcr}$

Nv protection invariant.

Being in secure execution mode is equivalent to saying that the secure execution mode pcr permits access to protected *Nv* ram.

$\text{InvNvProtection} \triangleq$
 $\text{InSem} \equiv (\text{semPcr} = \text{SemProtect})$

Verifiable revocation invariant. There had better not be any decisions to obtain access for which a proof of revocation was also constructed.

$\text{InvVerifiableRevocation} \triangleq$

$\forall o \in \text{obtains} : \text{last extension was OBTAIN}$
 $\forall r \in \text{revokes} : \text{last extension was REVOKE}$
 $PcrPrior(o) \neq PcrPrior(r) \text{ cannot have both extended from same place}$

Access undeniability.

This invariant is modeled as performing an audit on the present state and seeing that all key bindings that have been used to obtain access appear in the audit report. A key binding o appears in the audit report iff $PcrLeq(o, appPcr)$, which means that there exists a sequence of zero or more extensions from o that reach $appPcr$.

However, it might be impossible to generate a valid audit report in the present node state. That is okay.

$InvAccessUndeniability \triangleq$

It is a bug to fail to require the audit to quote *SemHappy*.

$\wedge \text{ IF } BugAuditNoCheckHappy \text{ THEN TRUE ELSE}$
 $\text{ semPcr} = \text{ SemHappy}$

It is a bug to fail to require the audit to quote *SealReboot*.

$\wedge \text{ IF } BugAuditNoCheckSeal \text{ THEN TRUE ELSE}$
 $\text{ sealPcr} = \text{ SealReboot}$

\Rightarrow

$\forall o \in \text{obtains} : PcrLeq(o, appPcr)$

B Model

MODULE *PastureNodeModel*

VARIABLE *nv* Pasture's protected NV RAM region
 VARIABLE *appPcr* the application pcr
 VARIABLE *semPcr* the secure execution mode pcr
 VARIABLE *sealPcr* the seal pcr
 VARIABLE *bootCtr* the reboot counter
 VARIABLE *pc* *pc*
 VARIABLE *chkptts* ts passed to sem within checkpoint
 VARIABLE *tsvalues* what ts values are known
 VARIABLE *obtains* decisions to obtain access
 VARIABLE *revokes* decisions to prove revoke access

INSTANCE *PastureNode*

WITH

PcriAPPBOOT ← "boot",

SEMBOOT and *SEMPROTECT* must be different.

PcriSEMBOOT ← "boot",

PcriSEMPROTECT ← "protect",

PcriSEALBOOT ← "boot",

HAPPY and *UNHAPPY* must be different.

PcrxHAPPY ← 0,

PcrxUNHAPPY ← 1,

OBTAIN and *REVOKE* must be different.

PcrxOBTAIN ← 0,

PcrxREVOKE ← 1,

PcrxSEAL ← 0

MODEL-CHECKING CONSTRAINT

Override these definitions to adjust the constraint.

MaxAppPcrLen \triangleq 1

MaxSemPcrLen \triangleq 1

MaxSealPcrLen \triangleq 1

MaxTsValues \triangleq 1

MaxBootCtr \triangleq 1

Constrain \triangleq
 $\wedge PcrLen(appPcr) \leq MaxAppPcrLen$
 $\wedge PcrLen(semPcr) \leq MaxSemPcrLen$
 $\wedge PcrLen(sealPcr) \leq MaxSealPcrLen$
 $\wedge Cardinality(tsvalues) \leq MaxTsValues$
 $\wedge bootCtr \leq MaxBootCtr$

C Proof of Initial Pasture

MODULE *PastureNodeProof*

EXTENDS *PastureNode*, *TLAPS*

STATE FUNCTIONS

We talk about the “log” being in various places. Actually, what is in those places is a cryptographic summary of the log, which is of type *Pcr*. However, under the anticollision assumption of *PcrExtend*, the cryptographic summary is effectively in one-to-one correspondance with the actual log. So we talk as if the cryptographic summary were the log, rather than merely a reference to the log.

Check that *ts* is a valid seal attestation in the current node state. To be valid it must be a signed attestation and it must record *SemHappy*, *SealReboot* and the current boot counter.

$$\begin{aligned} \text{CheckTsIsCurrent}(ts) &\triangleq \\ &\wedge ts \in \text{SignedTs} \\ &\wedge ts.\text{semPcr} = \text{SemHappy} \\ &\wedge ts.\text{sealPcr} = \text{SealReboot} \\ &\wedge ts.\text{bootCtr} = \text{bootCtr} \end{aligned}$$

All valid seal attestations in the current node state. Seal attestations can be found among the known values (in *tsvalues*) or in the temporary state variable *chkptts* used during the checkpoint sem routine.

$$\begin{aligned} \text{AllCurrentTs} &\triangleq \\ &\{ts \in \text{tsvalues} \cup \{\text{chkptts}\} : \text{CheckTsIsCurrent}(ts)\} \end{aligned}$$

If there are any valid seal attestations in the current node state, choose one and get its log.

$$\begin{aligned} \text{CurrentTsLog} &\triangleq \\ \text{LET } ts &\triangleq \text{CHOOSE } ts \in \text{AllCurrentTs} : \text{TRUE} \\ \text{IN } &ts.\text{appPcr} \end{aligned}$$

The log is present in the *nv* ram. This is true iff the *nv* ram says it is current.

$$\begin{aligned} \text{LogInNv} &\triangleq \\ &\wedge nv.\text{current} \end{aligned}$$

The log is present in the application pcr. This is true iff the sem pcr contains *SemHappy* and the seal pcr contains *SealReboot*.

$$\begin{aligned} \text{LogInApp} &\triangleq \\ &\wedge \text{semPcr} = \text{SemHappy} \\ &\wedge \text{sealPcr} = \text{SealReboot} \end{aligned}$$

The log is present in some known seal attestation. This is true iff there exists a valid seal attestation in the current node state.

$$\begin{aligned} \text{LogInTs} &\triangleq \\ \text{AllCurrentTs} &\neq \{\} \end{aligned}$$

Assuming the log exists, determine if $Pcr\ p$ is on it.

The log has a domicile in the nv ram, when the nv ram is marked as current. The log has a domicile in the application pcr when the sem pcr contains $SemHappy$ and the seal pcr contains $SealReboot$. The log has a domicile in a seal ts attestation when that attestation quotes $SemHappy$, $SealReboot$, and the current $bootCtr$.

During secure execution mode, the log can also temporarily live in certain places, as it is moved from one domicile to another.

$$\text{IsOnLog}(p) \triangleq$$

Where to find the log at most times.

$$\begin{aligned} \wedge \text{LogInNv} &\Rightarrow PcrLeq(p, nv.appPcr) \\ \wedge \text{LogInApp} &\Rightarrow PcrLeq(p, appPcr) \\ \wedge \text{LogInTs} &\Rightarrow PcrLeq(p, CurrentTsLog) \end{aligned}$$

Special places to find the log during secure execution mode.

$$\begin{aligned} \wedge pc = PcRECOV1 &\Rightarrow \text{TRUE} \\ \wedge pc = PcRECOV2 &\Rightarrow PcrLeq(p, appPcr) \\ \wedge pc = PcRECOV3 &\Rightarrow PcrLeq(p, appPcr) \\ \wedge pc = PcCHKPT1 &\Rightarrow \text{TRUE} \\ \wedge pc = PcCHKPT2 &\Rightarrow PcrLeq(p, chkpts.appPcr) \\ \wedge pc = PcCHKPT3 &\Rightarrow PcrLeq(p, nv.appPcr) \\ \wedge pc = PcCHKPT4 &\Rightarrow PcrLeq(p, nv.appPcr) \\ \wedge pc = PcCHKPT5 &\Rightarrow PcrLeq(p, nv.appPcr) \end{aligned}$$

ADDITIONAL INVARIANTS

When the node is in secure execution mode, the secure execution mode pcr contains $SemProtect$.

$$\begin{aligned} \text{InvInSemProtect} &\triangleq \\ &\wedge \text{InvType} \\ &\wedge \text{goal}:: \\ \text{InSem} &\Rightarrow \text{semPcr} = \text{SemProtect} \end{aligned}$$

When the node is not in secure execution mode, the secure execution mode pcr contains a value from which $SemProtect$ cannot be reached.

$$\begin{aligned} \text{InvUnreachableSemProtect} &\triangleq \\ &\wedge \text{InvType} \\ &\wedge \text{InvInSemProtect} \end{aligned}$$

$$\wedge \text{goal}:: \\ \neg \text{InSem} \Rightarrow \neg \text{PcrLeq}(\text{semPcr}, \text{SemProtect})$$

All known signed seal attestations quote a *bootCtr* that does not exceed the current *bootCtr*.

$$\text{InvSignedTsLeqBoot} \triangleq \\ \wedge \text{InvType} \\ \wedge \text{goal}:: \\ \forall ts \in \text{tsvalues} \cup \{\text{chkptts}\} : \\ ts \in \text{SignedTs} \Rightarrow ts.\text{bootCtr} \leq \text{bootCtr}$$

When the node is not in secure execution mode, the secure execution mode pcr contains either (1) *SemHappy* or (2) a value from which *SemHappy* cannot be reached.

$$\text{InvUnforgeableSemHappy} \triangleq \\ \wedge \text{InvType} \\ \wedge \text{InvInSemProtect} \\ \wedge \text{goal}:: \\ \neg \text{InSem} \Rightarrow \\ \vee \text{semPcr} = \text{SemHappy} \\ \vee \neg \text{PcrLeq}(\text{semPcr}, \text{SemHappy})$$

The seal pcr contains either (1) *SealReboot* or (2) a value from which *SealReboot* cannot be reached.

$$\text{InvUnforgeableSealReboot} \triangleq \\ \wedge \text{InvType} \\ \wedge \text{goal}:: \\ \vee \text{sealPcr} = \text{SealReboot} \\ \vee \neg \text{PcrLeq}(\text{sealPcr}, \text{SealReboot})$$

Every entry in obtains and revokes has a last extension of OBTAIN and REVOKE, respectively.

$$\text{InvProperLastExtension} \triangleq \\ \wedge \text{InvType} \\ \wedge \text{goal}:: \\ \wedge \forall o \in \text{obtains} : \text{PcrHasExtension}(o) \wedge \text{PcrLastExtension}(o) = \text{Pcr}x\text{OBTAIN} \\ \wedge \forall r \in \text{revokes} : \text{PcrHasExtension}(r) \wedge \text{PcrLastExtension}(r) = \text{Pcr}x\text{REVOKE}$$

There is at most one log.

One Log to rule them all,
 One Log to find them,
 One Log to bring them all
 and in the darkness bind them.
 (with apologies to J. R. R. Tolkien)

$InvOneLog \triangleq$

$\wedge InvType$
 $\wedge InvSignedTsLeqBoot$
 $\wedge InvInSemProtect$
 $\wedge InvUnforgeableSemHappy$
 $\wedge InvUnforgeableSealReboot$
 $\wedge InvProperLastExtension$
 $\wedge goal::$

The log can only have at most one domicile at a time.

$\wedge LogInNv \Rightarrow \neg LogInApp \wedge \neg LogInTs$
 $\wedge LogInApp \Rightarrow \neg LogInNv \wedge \neg LogInTs$
 $\wedge LogInTs \Rightarrow \neg LogInNv \wedge \neg LogInApp$

Extra requirements during secure execution mode.

$\wedge pc = PcRECOV1 \Rightarrow TRUE$
 $\wedge pc = PcRECOV2 \Rightarrow LogInNv$
 $\wedge pc = PcRECOV3 \Rightarrow \neg LogInNv \wedge \neg LogInApp \wedge \neg LogInTs$
 $\wedge pc = PcCHKPT1 \Rightarrow TRUE$
 $\wedge pc = PcCHKPT2 \Rightarrow LogInTs \wedge CheckTsIsCurrent(chkptts)$
 $\wedge pc = PcCHKPT3 \Rightarrow LogInTs \wedge CheckTsIsCurrent(chkptts)$
 $\wedge pc = PcCHKPT4 \Rightarrow \neg LogInNv \wedge \neg LogInApp \wedge \neg LogInTs$
 $\wedge pc = PcCHKPT5 \Rightarrow LogInNv$

All seal attestations containing the log must have the same log.

$\wedge \forall ts1, ts2 \in AllCurrentTs : ts1.appPcr = ts2.appPcr$

Every entry in obtains (a decision to obtain access) is recorded on the log (assuming there is one).

$\wedge obtains:: \forall o \in obtains : IsOnLog(o)$

Every entry in revokes (a decision to prove revocation) is recorded on the log (assuming there is one).

$\wedge revokes:: \forall r \in revokes : IsOnLog(r)$

We have verifiable revocation.

$\wedge InvVerifiableRevocation$

NECESSARY FACTS ABOUT NATURALS

The *SMT* prover can prove these easily enough in isolation, but if you ask it to prove them in the middle of other proofs where records and other complicated things are flying around, it usually aborts with a type inference failure.

\leq is a total order

THEOREM *ThmNatLeqIsTotal* $\triangleq \forall i, j \in \text{Nat} : i \leq j \vee j \leq i$ BY SMT
 THEOREM *ThmNatLeqIsReflexive* $\triangleq \forall i \in \text{Nat} : i \leq i$ BY SMT
 THEOREM *ThmNatLeqIsAntisymmetric* $\triangleq \forall i, j \in \text{Nat} : i \leq j \wedge j \leq i \Rightarrow i = j$ BY SMT
 THEOREM *ThmNatLeqIsTransitive* $\triangleq \forall i, j, k \in \text{Nat} : i \leq j \wedge j \leq k \Rightarrow i \leq k$ BY SMT

\leq minimum is 0

THEOREM *ThmNatLeqMinIsZero* $\triangleq \forall i \in \text{Nat} : 0 \leq i$ BY SMT

\leq is the opposite of $>$

THEOREM *ThmNatLeqXorGt* $\triangleq \forall i, j \in \text{Nat} : i \leq j \equiv \neg(i > j)$ BY SMT

THEOREM *ThmNatMore* $\triangleq \forall i, j \in \text{Nat} : i \leq i + j$ BY SMT
 THEOREM *ThmNatLess* $\triangleq \forall i, j \in \text{Nat} : i - j \leq i$ BY SMT
 THEOREM *ThmNatInc* $\triangleq \forall i \in \text{Nat} : i + 1 > i$ BY SMT
 THEOREM *ThmNatDotDot* $\triangleq \forall i, j, k \in \text{Nat} : i \leq j \wedge j \leq k \equiv j \in i .. k$ BY SMT
 THEOREM *ThmNatDecZero* $\triangleq \forall n \in \text{Nat} : n > 0 \Rightarrow n - 1 \in \text{Nat}$ BY SMT
 THEOREM *ThmNatAddEq* $\triangleq \forall i, j, k \in \text{Nat} : i + k = j + k \Rightarrow i = j$ BY SMT
 THEOREM *ThmNatLeqLt* $\triangleq \forall i, j, k \in \text{Nat} : i \leq j \wedge j < k \Rightarrow i < k$ BY SMT

NECESSARY FACTS ABOUT SEQUENCES

I have not been able to figure out how to convince the prover to prove most of these.

Definition of a sequence.

THEOREM *ThmSeqDef* \triangleq
 ASSUME
 NEW CONSTANT S ,
 NEW CONSTANT $q \in \text{Seq}(S)$
 PROVE
 $q = [i \in 1 .. \text{Len}(q) \mapsto q[i]]$
 PROOF
 OMITTED

The empty sequence is a sequence of S , for any S .

THEOREM *ThmSeqEmptyIsSeq* \triangleq

ASSUME

NEW CONSTANT S

PROVE

$\langle \rangle \in Seq(S)$

PROOF

OMITTED

For any sequence q of S , $Len(q) \in Nat$.

THEOREM *ThmSeqLenIsNat* \triangleq

ASSUME

NEW CONSTANT S ,

NEW $q \in Seq(S)$

PROVE

$Len(q) \in Nat$

PROOF

OMITTED

For any non-empty sequence of S , its tail is a sequence of S .

THEOREM *ThmSeqTailIsSeq* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in Seq(S)$,

$q \neq \langle \rangle$

PROVE

$Tail(q) \in Seq(S)$

PROOF

OMITTED

For any sequence of S , appending $x \in S$ yields a sequence of S .

THEOREM *ThmSeqAppendIsSeq* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in Seq(S)$,

NEW CONSTANT $x \in S$

PROVE

$Append(q, x) \in Seq(S)$

PROOF

OMITTED

The result of $Append(q, x)$ is one longer than q .

THEOREM *ThmSeqAppendLen1* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$,

NEW CONSTANT $x \in S$

PROVE

$\text{Len}(\text{Append}(q, x)) = \text{Len}(q) + 1$

PROOF

OMITTED

The result of *Append*(q, x) starts with q .

THEOREM *ThmSeqAppendSubSeq* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$,

NEW CONSTANT $x \in S$

PROVE

$\text{SubSeq}(\text{Append}(q, x), 1, \text{Len}(q)) = q$

PROOF

OMITTED

Appending the last entry onto all but the last of a sequence yields the original sequence.

THEOREM *ThmSeqAppendPriorLast* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$,

$\text{Len}(q) > 0$

PROVE

$\text{Append}(\text{SubSeq}(q, 1, \text{Len}(q) - 1), q[\text{Len}(q)]) = q$

PROOF

OMITTED

The entire initial *SubSeq* of q is q .

THEOREM *ThmSeqEntireInitialSubSeq* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$

PROVE

$q = \text{SubSeq}(q, 1, \text{Len}(q))$

PROOF

OMITTED

Initial *SubSeq* \in sequence.

THEOREM *ThmSeqInitialSubSeqIsSeq* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in \text{Seq}(S)$,
 NEW CONSTANT $n \in \text{Nat}$,
 $n \leq \text{Len}(q)$

PROVE

$\text{SubSeq}(q, 1, n) \in \text{Seq}(S)$

PROOF

OMITTED

Initial *SubSeq* is antisymmetric.

THEOREM *ThmSeqInitialSubSeqIsAntisymmetric* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in \text{Seq}(S)$,
 NEW CONSTANT $r \in \text{Seq}(S)$,
 $\text{Len}(q) \leq \text{Len}(r)$,
 $\text{Len}(r) \leq \text{Len}(q)$,
 $q = \text{SubSeq}(r, 1, \text{Len}(q))$,
 $r = \text{SubSeq}(q, 1, \text{Len}(r))$

PROVE

$q = r$

PROOF

(1) $\text{Len}(q) = \text{Len}(r)$
 (2) USE *ThmSeqLenIsNat*
 (2) QED BY *ThmNatLeqIsAntisymmetric*
 (1) QED BY *ThmSeqEntireInitialSubSeq*

Initial *SubSeq* is transitive.

THEOREM *ThmSeqInitialSubSeqIsTransitive* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in \text{Seq}(S)$,
 NEW CONSTANT $r \in \text{Seq}(S)$,
 NEW CONSTANT $s \in \text{Seq}(S)$,
 $\text{Len}(q) \leq \text{Len}(r)$,
 $\text{Len}(r) \leq \text{Len}(s)$,
 $q = \text{SubSeq}(r, 1, \text{Len}(q))$,
 $r = \text{SubSeq}(s, 1, \text{Len}(r))$

PROVE

$$q = \text{SubSeq}(s, 1, \text{Len}(q))$$

PROOF

OMITTED

Sequence append incompatible.

THEOREM *ThmSeqAppendIncompatible* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$,

NEW CONSTANT $s1 \in S$,

NEW CONSTANT $s2 \in S$,

$s1 \neq s2$

PROVE

$\text{Append}(q, s1) \neq \text{Append}(q, s2)$

PROOF

OMITTED

Sequence append anti-collision.

THEOREM *ThmSeqAppendAnticollision* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT q ,

NEW CONSTANT $q1 \in \text{Seq}(S)$,

NEW CONSTANT $q2 \in \text{Seq}(S)$,

NEW CONSTANT $s1 \in S$,

NEW CONSTANT $s2 \in S$,

$q = \text{Append}(q1, s1)$,

$q = \text{Append}(q2, s2)$

PROVE

$q1 = q2 \wedge s1 = s2$

PROOF

OMITTED

PcrInit \in *Pcr*

THEOREM *ThmPcrInitIsPcr* \triangleq

$\forall i \in Pcri : PcrInit(i) \in Pcr$

PROOF

- $\langle 1 \rangle$ TAKE $i \in Pcri$
- $\langle 1 \rangle$ USE DEF *Pcr*, *PcrInit*
- $\langle 1 \rangle$ DEFINE $p \triangleq PcrInit(i)$
- $\langle 1 \rangle 1.$ $p.init \in Pcri$ OBVIOUS
- $\langle 1 \rangle 2.$ $p.extq \in Seq(Pcrx)$ BY *ThmSeqEmptyIsSeq*
- $\langle 1 \rangle$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$

PcrExtend \in *Pcr*

THEOREM *ThmPcrExtendIsPcr* \triangleq

$\forall p \in Pcr, x \in Pcrx : PcrExtend(p, x) \in Pcr$

PROOF

- $\langle 1 \rangle$ TAKE $p \in Pcr, x \in Pcrx$
- $\langle 1 \rangle$ USE DEF *Pcr*, *PcrExtend*
- $\langle 1 \rangle$ DEFINE $px \triangleq PcrExtend(p, x)$
- $\langle 1 \rangle 1.$ $px.init \in Pcri$ OBVIOUS
- $\langle 1 \rangle 2.$ $px.extq \in Seq(Pcrx)$ BY *ThmSeqAppendIsSeq*
- $\langle 1 \rangle$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$

PcrLen \in *Nat*

THEOREM *ThmPcrLenIsNat* \triangleq

$\forall p \in Pcr : PcrLen(p) \in Nat$

PROOF

- $\langle 1 \rangle$ TAKE $p \in Pcr$
- $\langle 1 \rangle$ USE DEF *PcrLen*
- $\langle 1 \rangle$ USE DEF *Pcr*
- $\langle 1 \rangle$ QED BY *ThmSeqLenIsNat*

$p \leq PcrExtend(p, x)$

THEOREM *ThmPcrExtendLeq* \triangleq

$\forall p \in Pcr, x \in Pcrx : PcrLeq(p, PcrExtend(p, x))$

PROOF

- $\langle 1 \rangle$ TAKE $p \in Pcr, x \in Pcrx$
- $\langle 1 \rangle$ DEFINE $px \triangleq PcrExtend(p, x)$
- $\langle 1 \rangle$ USE DEF *Pcr*
- $\langle 1 \rangle$ USE DEF *PcrExtend*
- $\langle 1 \rangle$ USE DEF *PcrLeq*
- $\langle 1 \rangle 1.$ $p.init = px.init$ OBVIOUS
- $\langle 1 \rangle 2.$ $Len(p.extq) \leq Len(px.extq)$

⟨2⟩ $Len(p.extq) + 1 = Len(px.extq)$ BY *ThmSeqAppendLen1*
 ⟨2⟩ USE *ThmSeqLenIsNat*
 ⟨2⟩ QED BY *ThmNatMore*
 ⟨1⟩3. $p.extq = SubSeq(px.extq, 1, Len(p.extq))$ BY *ThmSeqAppendSubSeq*
 ⟨1⟩ QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3

$p \neq PcrExtend(p, x)$

THEOREM *ThmPcrExtendNeq* \triangleq

$\forall p \in Pcr, x \in Pcrx : p \neq PcrExtend(p, x)$

PROOF

⟨1⟩ TAKE $p \in Pcr, x \in Pcrx$
 ⟨1⟩ DEFINE $px \triangleq PcrExtend(p, x)$
 ⟨1⟩ USE DEF *Pcr*
 ⟨1⟩ USE DEF *PcrExtend*
 ⟨1⟩ $p.extq \neq px.extq$
 ⟨2⟩ $p.extq \in Seq(Pcrx)$ OBVIOUS
 ⟨2⟩ $px.extq \in Seq(Pcrx)$ BY *ThmSeqAppendIsSeq*
 ⟨2⟩ DEFINE $pn \triangleq Len(p.extq)$
 ⟨2⟩ DEFINE $pxn \triangleq Len(px.extq)$
 ⟨2⟩ $pn \neq pxn$
 ⟨3⟩ $pn \in Nat$ BY *ThmSeqLenIsNat*
 ⟨3⟩ $pxn \in Nat$ BY *ThmSeqLenIsNat*
 ⟨3⟩ $pxn = pn + 1$ BY *ThmSeqAppendLen1*
 ⟨3⟩ $pxn > pn$ BY *ThmNatInc*
 ⟨3⟩ $\neg(pxn \leq pn)$ BY *ThmNatLeqXorGt*
 ⟨3⟩ QED BY *ThmNatLeqIsReflexive*
 ⟨2⟩ QED OBVIOUS
 ⟨1⟩ QED OBVIOUS

Pcr equality. This would seem to be trivial but the prover cannot seem to figure it out by itself.

THEOREM *ThmPcrEqual* \triangleq

$\forall p, q \in Pcr :$
 $\wedge p.init = q.init$
 $\wedge p.extq = q.extq$
 $\Rightarrow p = q$

PROOF

⟨1⟩ TAKE $p, q \in Pcr$
 ⟨1⟩ HAVE $p.init = q.init \wedge p.extq = q.extq$
 ⟨1⟩ USE DEF *Pcr*

The following fact seems to be necessary to help the prover.

⟨1⟩ $p = [q \text{ EXCEPT } !.init = p.init, !.extq = p.extq]$ OBVIOUS
 ⟨1⟩ QED OBVIOUS

Anti-collision property.

THEOREM *ThmPcrExtendAnticollision* \triangleq

$\forall p1, p2 \in Pcr, x1, x2 \in Pcrx :$

$PcrExtend(p1, x1) = PcrExtend(p2, x2) \Rightarrow p1 = p2 \wedge x1 = x2$

PROOF

(1) TAKE $p1, p2 \in Pcr, x1, x2 \in Pcrx$

(1) DEFINE $px1 \triangleq PcrExtend(p1, x1)$

(1) DEFINE $px2 \triangleq PcrExtend(p2, x2)$

(1) HAVE $px1 = px2$

(1) USE DEF *Pcr*

(1) USE DEF *PcrExtend*

(1) QED

(2) $p1.init = p2.init$ OBVIOUS

(2) $p1.extq = p2.extq \wedge x1 = x2$

Create definitions for the *extq* fields and then hide them to prevent overwhelming the prover.

(3) DEFINE $p1q \triangleq p1.extq$

(3) DEFINE $p2q \triangleq p2.extq$

(3) HIDE DEF $p1q$

(3) HIDE DEF $p2q$

(3) $Append(p1q, x1) = Append(p2q, x2)$ BY DEF $p1q, p2q$

(3) $p1q \in Seq(Pcrx)$ BY DEF $p1q$

(3) $p2q \in Seq(Pcrx)$ BY DEF $p2q$

(3) $p1q = p2q \wedge x1 = x2$ BY *ThmSeqAppendAnticollision*

(3) QED BY DEF $p1q, p2q$

(2) QED BY *ThmPcrEqual*

If two extensions of the same pcr are both \leq a target pcr, then the extensions must be the same.

THEOREM *ThmPcrExtendLeqAnticollision* \triangleq

$\forall p, t \in Pcr, x1, x2 \in Pcrx :$

$PcrLeq(PcrExtend(p, x1), t) \wedge PcrLeq(PcrExtend(p, x2), t) \Rightarrow x1 = x2$

PROOF

(1) TAKE $p, t \in Pcr, x1, x2 \in Pcrx$

(1) HAVE $PcrLeq(PcrExtend(p, x1), t) \wedge PcrLeq(PcrExtend(p, x2), t)$

(1) USE DEF *Pcr*

(1) USE DEF *PcrExtend*

(1) USE DEF *PcrLeq*

(1) DEFINE $qp \triangleq p.extq$

(1) DEFINE $qt \triangleq t.extq$

(1) DEFINE $qpx1 \triangleq Append(qp, x1)$

(1) DEFINE $qpx2 \triangleq Append(qp, x2)$

(1) $qp \in Seq(Pcrx)$ OBVIOUS

(1) $qt \in Seq(Pcrx)$ OBVIOUS

(1) $Len(qpx1) \leq Len(qt)$ OBVIOUS

(1) $Len(qpx2) \leq Len(qt)$ OBVIOUS

(1) $SubSeq(qt, 1, Len(qpx1)) = qpx1$ OBVIOUS
 (1) $SubSeq(qt, 1, Len(qpx2)) = qpx2$ OBVIOUS
 (1) HIDE DEF qp
 (1) HIDE DEF qt
 (1) $Len(qpx1) = Len(qp) + 1$ BY $ThmSeqAppendLen1$
 (1) $Len(qpx2) = Len(qp) + 1$ BY $ThmSeqAppendLen1$
 (1) $Len(qpx1) = Len(qpx2)$ OBVIOUS
 (1) $qpx1 = qpx2$ OBVIOUS
 (1) QED

The prover really needs help to focus its attention.

(2)1. $x1 \in Pcrx$ OBVIOUS
 (2)2. $x2 \in Pcrx$ OBVIOUS
 (2)3. $qp \in Seq(Pcrx)$ OBVIOUS
 (2)4. $Append(qp, x1) = Append(qp, x2)$ OBVIOUS
 (2) QED BY ONLY (2)1, (2)2, (2)3, (2)4, $ThmSeqAppendAnticollision$

$PcrExtend$ increases the length by 1.

THEOREM $ThmPcrExtendLen1 \triangleq$

$\forall p \in Pcr, x \in Pcrx : PcrLen(PcrExtend(p, x)) = PcrLen(p) + 1$

PROOF

(1) TAKE $p \in Pcr, x \in Pcrx$
 (1) DEFINE $px \triangleq PcrExtend(p, x)$
 (1) $px \in Pcr$ BY $ThmPcrExtendIsPcr$
 (1) USE DEF $PcrLen$
 (1) USE DEF $PcrExtend$
 (1) USE DEF Pcr
 (1) QED BY $ThmSeqAppendLen1$

$PcrLeq$ implies \leq on respective $PcrLen$.

THEOREM $ThmPcrLeqLeq \triangleq$

$\forall p, q \in Pcr : PcrLeq(p, q) \Rightarrow PcrLen(p) \leq PcrLen(q)$

PROOF

(1) TAKE $p, q \in Pcr$
 (1) HAVE $PcrLeq(p, q)$
 (1) USE DEF Pcr
 (1) $Len(p.extq) \leq Len(q.extq)$ BY DEF $PcrLeq$
 (1) $PcrLen(p) = Len(p.extq)$ BY DEF $PcrLen$
 (1) $PcrLen(q) = Len(q.extq)$ BY DEF $PcrLen$
 (1) QED OBVIOUS

PcrLeq is a partial order.

THEOREM *ThmPcrLeqIsReflexive* \triangleq
 $\forall p \in Pcr : PcrLeq(p, p)$

PROOF

- $\langle 1 \rangle$ TAKE $p \in Pcr$
- $\langle 1 \rangle$ USE DEF *PcrLeq*
- $\langle 1 \rangle$ USE DEF *Pcr*
- $\langle 1 \rangle 1$. $p.init = p.init$ OBVIOUS
- $\langle 1 \rangle 2$. $Len(p.extq) \leq Len(p.extq)$
 - $\langle 2 \rangle$ USE *ThmSeqLenIsNat*
 - $\langle 2 \rangle$ USE *ThmNatLeqIsReflexive*
 - $\langle 2 \rangle$ QED OBVIOUS
- $\langle 1 \rangle 3$. $p.extq = SubSeq(p.extq, 1, Len(p.extq))$
 - $\langle 2 \rangle$ USE *ThmSeqEntireInitialSubSeq*
 - $\langle 2 \rangle$ QED OBVIOUS
- $\langle 1 \rangle$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$, $\langle 1 \rangle 3$

THEOREM *ThmPcrLeqIsAntisymmetric* \triangleq
 $\forall p, q \in Pcr : PcrLeq(p, q) \wedge PcrLeq(q, p) \Rightarrow p = q$

PROOF

- $\langle 1 \rangle$ TAKE $p, q \in Pcr$
- $\langle 1 \rangle 2$. HAVE $PcrLeq(p, q) \wedge PcrLeq(q, p)$
- $\langle 1 \rangle 3$. $p.init = q.init$
 - $\langle 2 \rangle$ USE DEF *PcrLeq*
 - $\langle 2 \rangle$ QED BY $\langle 1 \rangle 2$
- $\langle 1 \rangle 4$. $p.extq = q.extq$
 - $\langle 2 \rangle$ USE DEF *PcrLeq*
 - $\langle 2 \rangle$ USE DEF *Pcr*
 - $\langle 2 \rangle$ USE *ThmSeqInitialSubSeqIsAntisymmetric*
 - $\langle 2 \rangle$ QED BY $\langle 1 \rangle 2$
- $\langle 1 \rangle$ QED
 - $\langle 2 \rangle$ USE *ThmPcrEqual*
 - $\langle 2 \rangle$ QED BY $\langle 1 \rangle 3$, $\langle 1 \rangle 4$

THEOREM *ThmPcrLeqIsTransitive* \triangleq
 $\forall p, q, r \in Pcr : PcrLeq(p, q) \wedge PcrLeq(q, r) \Rightarrow PcrLeq(p, r)$

PROOF

- $\langle 1 \rangle$ TAKE $p, q, r \in Pcr$
- $\langle 1 \rangle$ HAVE $PcrLeq(p, q) \wedge PcrLeq(q, r)$
- $\langle 1 \rangle$ USE DEF *PcrLeq*
- $\langle 1 \rangle 1$. $p.init = r.init$ OBVIOUS
- $\langle 1 \rangle 2$. $Len(p.extq) \leq Len(r.extq)$
 - $\langle 2 \rangle$ USE DEF *Pcr*
 - $\langle 2 \rangle$ USE *ThmSeqLenIsNat*

⟨2⟩ QED BY *ThmNatLeqIsTransitive*
 ⟨1⟩3. $p.extq = SubSeq(r.extq, 1, Len(p.extq))$
 ⟨2⟩ USE DEF *Pcr*
 ⟨2⟩ QED BY *ThmSeqInitialSubSeqIsTransitive*
 ⟨1⟩ QED BY ⟨1⟩1, ⟨1⟩2, ⟨1⟩3

An extension of a *Pcr* p cannot reach p .

THEOREM *ThmPcrExtendSelfUnreachable* \triangleq
 $\forall p \in Pcr, x \in Pcrx : \neg PcrLeq(PcrExtend(p, x), p)$

PROOF

⟨1⟩ TAKE $p \in Pcr, x \in Pcrx$
 ⟨1⟩ DEFINE $px \triangleq PcrExtend(p, x)$
 ⟨1⟩ DEFINE $isleq \triangleq PcrLeq(px, p)$
 ⟨1⟩ $px \in Pcr$ BY *ThmPcrExtendIsPcr*

Proof by contradiction.

⟨1⟩1. CASE $\neg isleq$ BY ⟨1⟩1
 ⟨1⟩2. CASE $isleq$
 ⟨2⟩1. $PcrLen(px) \leq PcrLen(p)$
 ⟨3⟩ USE ⟨1⟩2
 ⟨3⟩ USE *ThmPcrLeqLeq*
 ⟨3⟩ QED OBVIOUS
 ⟨2⟩2. $PcrLen(px) > PcrLen(p)$
 ⟨3⟩ $PcrLen(px) = PcrLen(p) + 1$ BY *ThmPcrExtendLen1*
 ⟨3⟩ USE *ThmPcrLenIsNat*
 ⟨3⟩ USE *ThmNatInc*
 ⟨3⟩ QED OBVIOUS
 ⟨2⟩ USE *ThmPcrLenIsNat*
 ⟨2⟩ USE *ThmNatLeqXorGt*
 ⟨2⟩ QED BY ⟨2⟩1, ⟨2⟩2
 ⟨1⟩ QED BY ⟨1⟩2, ⟨1⟩1

If an extension of a *Pcr* can reach a target, the *Pcr* itself can reach the target.

THEOREM *ThmPcrReachableIfExtend* \triangleq
 $\forall p, q \in Pcr, x \in Pcrx :$
 $PcrLeq(PcrExtend(p, x), q) \Rightarrow PcrLeq(p, q)$

PROOF

⟨1⟩ TAKE $p, q \in Pcr, x \in Pcrx$
 ⟨1⟩ DEFINE $px \triangleq PcrExtend(p, x)$
 ⟨1⟩ HAVE $PcrLeq(px, q)$
 ⟨1⟩ $px \in Pcr$ BY *ThmPcrExtendIsPcr*
 ⟨1⟩ $PcrLeq(p, px)$ BY *ThmPcrExtendLeq*
 ⟨1⟩ QED BY *ThmPcrLeqIsTransitive*

If a target Pcr is not reachable from a source Pcr , then it is not reachable from an extension of the source Pcr .

THEOREM *ThmPcrExtendSourceUnreachable* \triangleq

$$\forall p, q \in Pcr, x \in Pcrx : \\ \neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q)$$

PROOF

(1) QED BY *ThmPcrReachableIfExtend*

If p equals q or cannot reach q , then an extension of p cannot reach q .

THEOREM *ThmPcrExtendFromEqOrNotleq* \triangleq

$$\forall p, q \in Pcr, x \in Pcrx : \\ p = q \vee \neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q)$$

PROOF

(1) TAKE $p, q \in Pcr, x \in Pcrx$
 (1) HAVE $p = q \vee \neg PcrLeq(p, q)$
 (1) CASE $p = q$ BY *ThmPcrExtendSelfUnreachable*
 (1) CASE $\neg PcrLeq(p, q)$ BY *ThmPcrReachableIfExtend*
 (1) QED OBVIOUS

Different extensions of a pcr are incompatible.

THEOREM *ThmPcrExtendIncompatible* \triangleq

$$\forall p \in Pcr, x1, x2 \in Pcrx : \\ x1 \neq x2 \Rightarrow \neg PcrLeq(PcrExtend(p, x1), PcrExtend(p, x2))$$

PROOF

(1) TAKE $p \in Pcr, x1, x2 \in Pcrx$
 (1) HAVE $x1 \neq x2$
 (1) DEFINE $p1 \triangleq PcrExtend(p, x1)$
 (1) DEFINE $p2 \triangleq PcrExtend(p, x2)$
 (1)1. CASE $\neg PcrLeq(p1, p2)$ BY (1)1
 (1)2. CASE $PcrLeq(p1, p2)$
 (2) USE (1)2
 (2) USE DEF *PcrLeq*
 (2) USE DEF *PcrExtend*
 (2) USE DEF *Pcr*
 (2) $p1.extq \in Seq(Pcrx)$ BY *ThmSeqAppendIsSeq*
 (2) $p2.extq \in Seq(Pcrx)$ BY *ThmSeqAppendIsSeq*
 (2) $Len(p1.extq) = Len(p2.extq)$
 (3) $Len(p1.extq) = Len(p.extq) + 1$ BY *ThmSeqAppendLen1*
 (3) $Len(p2.extq) = Len(p.extq) + 1$ BY *ThmSeqAppendLen1*
 (3) QED OBVIOUS
 (2) $p1.extq = p2.extq$ BY *ThmSeqEntireInitialSubSeq*
 (2) $p1.extq \neq p2.extq$ BY *ThmSeqAppendIncompatible*

⟨2⟩ QED OBVIOUS
 ⟨1⟩ QED BY ⟨1⟩1, ⟨1⟩2

If a *Pcr* has an extension, applying *PriorPcr* to it yields a *Pcr*.

THEOREM *ThmPcrPriorIsPcr* \triangleq

$\forall p \in Pcr : PcrHasExtension(p) \Rightarrow PcrPrior(p) \in Pcr$

PROOF

⟨1⟩ TAKE $p \in Pcr$
 ⟨1⟩ HAVE *PcrHasExtension*(p)
 ⟨1⟩ USE DEF *PcrHasExtension*
 ⟨1⟩ USE DEF *PcrPrior*
 ⟨1⟩ USE DEF *PcrLen*
 ⟨1⟩ USE DEF *Pcr*
 ⟨1⟩ $PcrPrior(p).extq \in Seq(Pcrx)$
 ⟨2⟩ $Len(p.extq) \in Nat$ BY *ThmSeqLenIsNat*
 ⟨2⟩ $Len(p.extq) - 1 \in Nat$ BY *ThmNatDecZero*
 ⟨2⟩ $Len(p.extq) - 1 \leq Len(p.extq)$ BY *ThmNatLess*
 ⟨2⟩ QED BY *ThmSeqInitialSubSeqIsSeq*
 ⟨1⟩ QED OBVIOUS

Putting the last extension back on the prior pcr yields the original pcr.

THEOREM *ThmPcrExtendPriorLast* \triangleq

$\forall p \in Pcr :$
 $PcrHasExtension(p) \Rightarrow$
 $PcrExtend(PcrPrior(p), PcrLastExtension(p)) = p$

PROOF

⟨1⟩ TAKE $p \in Pcr$
 ⟨1⟩ HAVE *PcrHasExtension*(p)
 ⟨1⟩ USE DEF *PcrHasExtension*
 ⟨1⟩ USE DEF *PcrPrior*
 ⟨1⟩ USE DEF *PcrLastExtension*
 ⟨1⟩ USE DEF *PcrExtend*
 ⟨1⟩ USE DEF *PcrLen*
 ⟨1⟩ USE DEF *Pcr*
 ⟨1⟩ DEFINE $p0 \triangleq PcrPrior(p)$
 ⟨1⟩ DEFINE $x \triangleq PcrLastExtension(p)$
 ⟨1⟩ $PcrExtend(p0, x).init = p.init$ OBVIOUS
 ⟨1⟩ $PcrExtend(p0, x).extq = p.extq$
 ⟨2⟩ DEFINE $qp \triangleq p.extq$
 ⟨2⟩ SUFFICES
 ASSUME
 $Len(qp) > 0,$

$qp \in Seq(Pcrx)$
 PROVE
 $Append(SubSeq(qp, 1, Len(qp) - 1), qp[Len(qp)]) = qp$
 OBVIOUS
 ⟨2⟩ HIDE DEF qp
 ⟨2⟩ QED BY $ThmSeqAppendPriorLast$
 ⟨1⟩ QED BY $ThmPcrEqual$

WELL KNOWN Pcr VALUES

Value of the application pcr attained by rebooting.

THEOREM $ThmAppRebootIsPcr \triangleq AppReboot \in Pcr$

PROOF

⟨1⟩ USE DEF $AppReboot$
 ⟨1⟩ USE DEF $Pcri$
 ⟨1⟩ QED BY $ThmPcrInitIsPcr$

Value of the secure execution mode pcr attained by rebooting.

THEOREM $ThmSemRebootIsPcr \triangleq SemReboot \in Pcr$

PROOF

⟨1⟩ USE DEF $SemReboot$
 ⟨1⟩ USE DEF $Pcri$
 ⟨1⟩ QED BY $ThmPcrInitIsPcr$

Value of the secure execution mode pcr attained by entering the protected module in secure execution mode.

THEOREM $ThmSemProtectIsPcr \triangleq SemProtect \in Pcr$

PROOF

⟨1⟩ USE DEF $SemProtect$
 ⟨1⟩ USE DEF $Pcri$
 ⟨1⟩ QED BY $ThmPcrInitIsPcr$

Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.

THEOREM $ThmSemHappyIsPcr \triangleq SemHappy \in Pcr$

PROOF

⟨1⟩ USE DEF $SemHappy$

- (1) USE *ThmSemProtectIsPcr*
- (1) USE DEF *Pcrx*
- (1) QED BY *ThmPcrExtendIsPcr*

Value of the seal pcr attained by rebooting.

THEOREM *ThmSealRebootIsPcr* \triangleq *SealReboot* \in *Pcr*

PROOF

- (1) USE DEF *SealReboot*
- (1) USE DEF *Pcri*
- (1) QED BY *ThmPcrInitIsPcr*

From *SemReboot* cannot reach *SemProtect*.

THEOREM *ThmSemRebootNotleqSemProtect* \triangleq \neg *PcrLeq*(*SemReboot*, *SemProtect*)

PROOF

- (1) USE DEF *PcrInit*
- (1) USE DEF *SemReboot*
- (1) USE DEF *SemProtect*
- (1) USE *AssSemProtect*
- (1) QED BY DEF *PcrLeq*

PROTECTED NV RAM STATE

THEOREM *ThmInitNvIsNv* \triangleq *InitNv* \in *Nv*

PROOF

- (1) USE DEF *InitNv*
- (1) USE DEF *Nv*
- (1) USE *ThmAppRebootIsPcr*
- (1) QED OBVIOUS

SEAL OPERATION TRANSPORT SESSION STATE

THEOREM $ThmNullTsIsTs \triangleq NullTs \in Ts$

PROOF

- ⟨1⟩ USE DEF Ts
- ⟨1⟩ QED OBVIOUS

THEOREM $ThmNullTsIsntSignedTs \triangleq NullTs \notin SignedTs$

PROOF

- ⟨1⟩ USE DEF $NullTs$
- ⟨1⟩ USE $NoSetContainsEverything$
- ⟨1⟩ QED OBVIOUS

PROOF OF INVARIANT $InvType$

It holds in the initial state.

THEOREM $ThmInitInvType \triangleq$

$Init \Rightarrow InvType$

PROOF

- ⟨1⟩ HAVE $Init$
- ⟨1⟩ USE DEF $Init$
- ⟨1⟩ USE DEF $Pc, PcRecov, PcChkpt$
- ⟨1⟩ USE DEF $PcIDLE$
- ⟨1⟩ USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- ⟨1⟩ USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Just walk through each variable.

- ⟨1⟩ $InvType!nv$ BY $ThmInitNvIsNv$
- ⟨1⟩ $InvType!appPcr$ BY $ThmAppRebootIsPcr$
- ⟨1⟩ $InvType!semPcr$ BY $ThmSemRebootIsPcr$
- ⟨1⟩ $InvType!sealPcr$ BY $ThmSealRebootIsPcr$
- ⟨1⟩ $InvType!bootCtr$ OBVIOUS
- ⟨1⟩ $InvType!pc$ OBVIOUS
- ⟨1⟩ $InvType!chkptts$ BY $ThmNullTsIsTs$
- ⟨1⟩ $InvType!tsvalues$ OBVIOUS
- ⟨1⟩ $InvType!obtains$ OBVIOUS
- ⟨1⟩ $InvType!revokes$ OBVIOUS
- ⟨1⟩ QED BY DEF $InvType$

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM $ThmNextInvType \triangleq$
 $InvType \wedge [Next]_{vars} \Rightarrow InvType'$

PROOF

- (1) HAVE $InvType \wedge [Next]_{vars}$
- (1) USE DEF $InvType$
- (1) USE DEF $Pc, PcRecov, PcChkpt$
- (1) USE DEF $PcIDLE$
- (1) USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- (1) USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Say QED here so that the rest of the proof has an indentation level. This creates a place where I can use the user interface renumbering operation to renumber all of the alternatives below.

(1) QED

Stutter step.

- (2)1. CASE $vars' = vars$
 - (3) USE (2)1
 - (3) USE DEF $vars$
 - (3) QED OBVIOUS

Walk through all *Next* alternatives.

- (2)2. CASE $NextObtainAccess$
 - (3) USE $NextObtainAccess$
 - (3) USE DEF $NextObtainAccess$
 - (3) QED OBVIOUS
- (2)3. CASE $NextProveRevoke$
 - (3) USE $NextProveRevoke$
 - (3) USE DEF $NextProveRevoke$
 - (3) $PcrPrior(appPcr) \in Pcr$ BY $ThmPcrPriorIsPcr$
 - (3) QED BY $ThmPcrExtendIsPcr$ DEF $Pcrx$
- (2)4. CASE $NextReboot$
 - (3) USE $NextReboot$
 - (3) USE DEF $NextReboot$
 - (3) $InvType!appPcr'$ BY $ThmAppRebootIsPcr$
 - (3) $InvType!semPcr'$ BY $ThmSemRebootIsPcr$
 - (3) $InvType!sealPcr'$ BY $ThmSealRebootIsPcr$
 - (3) $InvType!chkptts'$ BY $ThmNullTsIsTs$
 - (3) QED OBVIOUS
- (2)5. CASE $NextForgetSealTs$
 - (3) USE $NextForgetSealTs$
 - (3) USE DEF $NextForgetSealTs$
 - (3) QED OBVIOUS

- (2)6. CASE *NextExtendAppPcr*
 - (3) USE *NextExtendAppPcr*
 - (3) USE DEF *NextExtendAppPcr*
 - (3) $appPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 - (3) QED OBVIOUS
- (2)7. CASE *NextExtendSemPcr*
 - (3) USE *NextExtendSemPcr*
 - (3) USE DEF *NextExtendSemPcr*
 - (3) $InvType!semPcr'$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 - (3) QED OBVIOUS
- (2)8. CASE *NextExtendSealPcr*
 - (3) USE *NextExtendSealPcr*
 - (3) USE DEF *NextExtendSealPcr*
 - (3) $InvType!sealPcr'$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 - (3) QED OBVIOUS
- (2)9. CASE *NextIncBootCtr*
 - (3) USE *NextIncBootCtr*
 - (3) USE DEF *NextIncBootCtr*
 - (3) $bootCtr' \in Nat$
 - (4)1. $bootCtr + 1 \in Nat$ BY *SMT*
 - (4)2. $bootCtr' = bootCtr + 1$ OBVIOUS
 - (4) QED BY (4)1, (4)2
 - (3) QED OBVIOUS
- (2)10. CASE *NextEnterSemRecov*
 - (3) USE *NextEnterSemRecov*
 - (3) USE DEF *NextEnterSemRecov*
 - (3) $InvType!semPcr'$ BY *ThmSemProtectIsPcr*
 - (3) $InvType!pc'$ OBVIOUS
 - (3) $InvType!chkpts'$ OBVIOUS
 - (3) QED OBVIOUS
- (2)11. CASE *NextSemRecov1 WhenCorrect*
 - (3) USE *NextSemRecov1 WhenCorrect*
 - (3) USE DEF *NextSemRecov1 WhenCorrect*
 - (3) QED OBVIOUS
- (2)12. CASE *NextSemRecov1 WhenIncorrect*
 - (3) USE *NextSemRecov1 WhenIncorrect*
 - (3) USE DEF *NextSemRecov1 WhenIncorrect*
 - (3) $semPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 - (3) QED OBVIOUS
- (2)13. CASE *NextSemRecov2*
 - (3) USE *NextSemRecov2*

⟨3⟩ USE DEF *NextSemRecov2*
 ⟨3⟩ $nv' \in Nv$
 ⟨4⟩1. $nv \in Nv$ OBVIOUS
 ⟨4⟩2. $nv'.current \in \text{BOOLEAN}$ BY DEF *Nv*
 ⟨4⟩3. $nv' = [nv \text{ EXCEPT } !.current = nv'.current]$ OBVIOUS
 ⟨4⟩ QED BY ONLY ⟨4⟩1, ⟨4⟩2, ⟨4⟩3 DEF *Nv*
 ⟨3⟩ QED OBVIOUS

⟨2⟩14. CASE *NextSemRecov3*
 ⟨3⟩ USE *NextSemRecov3*
 ⟨3⟩ USE DEF *NextSemRecov3*
 ⟨3⟩ $semPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 ⟨3⟩ $pc' \in Pc$ BY DEF *Pc*, *PcRecov*
 ⟨3⟩ QED OBVIOUS

⟨2⟩15. CASE *NextSealTs*
 ⟨3⟩ USE *NextSealTs*
 ⟨3⟩ USE DEF *NextSealTs*
 ⟨3⟩ $sealPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 ⟨3⟩ $tvalues' \in \text{SUBSET } Ts$
 ⟨4⟩1. $tvalues \in \text{SUBSET } Ts$ OBVIOUS
 ⟨4⟩ DEFINE $ts \triangleq \text{NextSealTs}! : !ts$
 ⟨4⟩2. $ts \in Ts$ BY DEF *Ts*, *SignedTs*
 ⟨4⟩3. $tvalues' = tvalues \cup \{ts\}$ OBVIOUS
 ⟨4⟩ QED BY ⟨4⟩1, ⟨4⟩2, ⟨4⟩3
 ⟨3⟩ QED OBVIOUS

⟨2⟩16. CASE *NextEnterSemChkpt*
 ⟨3⟩ USE *NextEnterSemChkpt*
 ⟨3⟩ USE DEF *NextEnterSemChkpt*
 ⟨3⟩ $semPcr' \in Pcr$ BY *ThmSemProtectIsPcr*
 ⟨3⟩ QED OBVIOUS

⟨2⟩17. CASE *NextSemChkpt1WhenCorrect*
 ⟨3⟩ USE *NextSemChkpt1WhenCorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1WhenCorrect*
 ⟨3⟩ $chkptts' \in \text{SignedTs}$
 ⟨4⟩ USE DEF *EnterSemChkptPredicate*
 ⟨4⟩ USE DEF *Ts*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ QED OBVIOUS

⟨2⟩18. CASE *NextSemChkpt1WhenIncorrect*
 ⟨3⟩ USE *NextSemChkpt1WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1WhenIncorrect*
 ⟨3⟩ $semPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 ⟨3⟩ QED OBVIOUS

(2)19. CASE *NextSemChkpt2*
 (3) USE *NextSemChkpt2*
 (3) USE DEF *NextSemChkpt2*
 (3) $nv' \in Nv$
 (4) DEFINE $nvappPcr1 \triangleq NextSemChkpt2! : !nvappPcr1$
 (4)1. $nvappPcr1 \in Pcr$ BY DEF *SignedTs*
 (4) QED BY (4)1 DEF *Nv*
 (3) QED OBVIOUS

(2)20. CASE *NextSemChkpt3*
 (3) USE *NextSemChkpt3*
 (3) USE DEF *NextSemChkpt3*
 (3) $bootCtr' \in Nat$
 (4)1. $bootCtr \in Nat$ OBVIOUS
 (4)2. $bootCtr + 1 \in Nat$ BY ONLY (4)1, *SMT*
 (4) QED BY (4)1, (4)2
 (3) QED OBVIOUS

(2)21. CASE *NextSemChkpt4*
 (3) USE *NextSemChkpt4*
 (3) USE DEF *NextSemChkpt4*
 (3) $nv' \in Nv$ BY DEF *Nv*
 (3) QED OBVIOUS

(2)22. CASE *NextSemChkpt5*
 (3) USE *NextSemChkpt5*
 (3) USE DEF *NextSemChkpt5*
 (3) $semPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 (3) QED OBVIOUS

(2) QED
 BY (2)1,
 (2)2, (2)3, (2)4, (2)5, (2)6, (2)7, (2)8, (2)9, (2)10,
 (2)11, (2)12, (2)13, (2)14, (2)15, (2)16, (2)17, (2)18,
 (2)19, (2)20, (2)21, (2)22
 DEF *Next*

It is an invariant of the specification.

THEOREM *ThmInvType* \triangleq
Spec $\Rightarrow \square InvType$

PROOF

(1) *Init* $\Rightarrow InvType$ BY *ThmInitInvType*
 (1) $InvType \wedge [Next]_{vars} \Rightarrow InvType'$ BY *ThmNextInvType*
 (1) QED

PROOF OF INVARIANT $InvInSemProtect$

It holds in the initial state.

THEOREM $ThmInitInvInSemProtect \triangleq$
 $Init \Rightarrow InvInSemProtect$

PROOF

- $\langle 1 \rangle$ HAVE $Init$
- $\langle 1 \rangle$ $InvType$ BY $ThmInitInvType$
- $\langle 1 \rangle$ USE DEF $Init$
- $\langle 1 \rangle$ USE DEF $InSem$
- $\langle 1 \rangle$ QED BY DEF $InvInSemProtect$

If it holds in the current state, and we perform a $Next$ action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM $ThmNextInvInSemProtect \triangleq$
 $InvInSemProtect \wedge [Next]_{vars} \Rightarrow InvInSemProtect'$

PROOF

- $\langle 1 \rangle$ HAVE $InvInSemProtect \wedge [Next]_{vars}$
- $\langle 1 \rangle$ USE DEF $InvInSemProtect$
- $\langle 1 \rangle$ USE DEF $InSem$

- $\langle 1 \rangle$ $InvType'$ BY $ThmNextInvType$
- $\langle 1 \rangle$ $InvInSemProtect'!$ goal'

- $\langle 2 \rangle$ USE DEF $PcIDLE$
- $\langle 2 \rangle$ USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- $\langle 2 \rangle$ USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Stutter step.

- $\langle 2 \rangle 1.$ CASE $vars' = vars$
- $\langle 3 \rangle$ USE $\langle 2 \rangle 1$
- $\langle 3 \rangle$ USE DEF $vars$
- $\langle 3 \rangle$ QED OBVIOUS

Walk through all $Next$ alternatives.

- $\langle 2 \rangle 2.$ CASE $NextObtainAccess$
- $\langle 3 \rangle$ USE $NextObtainAccess$
- $\langle 3 \rangle$ USE DEF $NextObtainAccess$
- $\langle 3 \rangle$ QED OBVIOUS

- $\langle 2 \rangle 3.$ CASE $NextProveRevoke$
- $\langle 3 \rangle$ USE $NextProveRevoke$

- ⟨3⟩ USE DEF *NextProveRevoke*
- ⟨3⟩ QED OBVIOUS
- (2)4. CASE *NextReboot*
 - ⟨3⟩ USE *NextReboot*
 - ⟨3⟩ USE DEF *NextReboot*
 - ⟨3⟩ USE DEF *PcrLeq*
 - ⟨3⟩ USE DEF *SemProtect*
 - ⟨3⟩ USE DEF *SemReboot*
 - ⟨3⟩ USE DEF *PcrInit*
 - ⟨3⟩ USE DEF *Pcri*
 - ⟨3⟩ USE *AssSemProtect*
 - ⟨3⟩ QED OBVIOUS
- (2)5. CASE *NextForgetSealTs*
 - ⟨3⟩ USE *NextForgetSealTs*
 - ⟨3⟩ USE DEF *NextForgetSealTs*
 - ⟨3⟩ QED OBVIOUS
- (2)6. CASE *NextExtendAppPcr*
 - ⟨3⟩ USE *NextExtendAppPcr*
 - ⟨3⟩ USE DEF *NextExtendAppPcr*
 - ⟨3⟩ QED OBVIOUS
- (2)7. CASE *NextExtendSemPcr*
 - ⟨3⟩ USE *NextExtendSemPcr*
 - ⟨3⟩ USE DEF *NextExtendSemPcr*
 - ⟨3⟩ QED OBVIOUS
- (2)8. CASE *NextExtendSealPcr*
 - ⟨3⟩ USE *NextExtendSealPcr*
 - ⟨3⟩ USE DEF *NextExtendSealPcr*
 - ⟨3⟩ QED OBVIOUS
- (2)9. CASE *NextIncBootCtr*
 - ⟨3⟩ USE *NextIncBootCtr*
 - ⟨3⟩ USE DEF *NextIncBootCtr*
 - ⟨3⟩ QED OBVIOUS
- (2)10. CASE *NextEnterSemRecov*
 - ⟨3⟩ USE *NextEnterSemRecov*
 - ⟨3⟩ USE DEF *NextEnterSemRecov*
 - ⟨3⟩ QED OBVIOUS
- (2)11. CASE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩12. CASE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *PcrExtend*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩13. CASE *NextSemRecov2*
 - ⟨3⟩ USE *NextSemRecov2*
 - ⟨3⟩ USE DEF *NextSemRecov2*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩14. CASE *NextSemRecov3*
 - ⟨3⟩ USE *NextSemRecov3*
 - ⟨3⟩ USE DEF *NextSemRecov3*
 - ⟨3⟩ USE DEF *PcrExtend*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩15. CASE *NextSealTs*
 - ⟨3⟩ USE *NextSealTs*
 - ⟨3⟩ USE DEF *NextSealTs*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩16. CASE *NextEnterSemChkpt*
 - ⟨3⟩ USE *NextEnterSemChkpt*
 - ⟨3⟩ USE DEF *NextEnterSemChkpt*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩17. CASE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩18. CASE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *PcrExtend*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩19. CASE *NextSemChkpt2*
 - ⟨3⟩ USE *NextSemChkpt2*
 - ⟨3⟩ USE DEF *NextSemChkpt2*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩20. CASE *NextSemChkpt3*
 - ⟨3⟩ USE *NextSemChkpt3*
 - ⟨3⟩ USE DEF *NextSemChkpt3*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩21. CASE *NextSemChkpt4*

⟨3⟩ USE *NextSemChkpt4*
 ⟨3⟩ USE DEF *NextSemChkpt4*
 ⟨3⟩ QED OBVIOUS

⟨2⟩22. CASE *NextSemChkpt5*
 ⟨3⟩ USE *NextSemChkpt5*
 ⟨3⟩ USE DEF *NextSemChkpt5*
 ⟨3⟩ USE DEF *PcrExtend*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩8, ⟨2⟩9, ⟨2⟩10,
 ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,
 ⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22
 DEF *Next*
 ⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvInSemProtect* \triangleq
 $Spec \Rightarrow \Box InvInSemProtect$

PROOF

⟨1⟩ $Init \Rightarrow InvInSemProtect$ BY *ThmInitInvInSemProtect*
 ⟨1⟩ $InvInSemProtect \wedge [Next]_{vars} \Rightarrow InvInSemProtect'$
 BY *ThmNextInvInSemProtect*
 ⟨1⟩ USE DEF *Spec*
 ⟨1⟩ QED

PROOF OF INVARIANT *InvUnreachableSemProtect*

It holds in the initial state.

THEOREM *ThmInitInvUnreachableSemProtect* \triangleq
 $Init \Rightarrow InvUnreachableSemProtect$

PROOF

⟨1⟩ HAVE *Init*
 ⟨1⟩ $InvType$ BY *ThmInitInvType*

(1) *InvInSemProtect* BY *ThmInitInvInSemProtect*
 (1) USE DEF *Init*
 (1) USE DEF *InSem*
 (1) \neg *InSemOBVIOUS*
 (1) \neg *PcrLeq*(*semPcr*, *SemProtect*)
 (2) USE DEF *PcrLeq*
 (2) USE DEF *SemProtect*
 (2) USE DEF *SemReboot*
 (2) USE DEF *PcrInit*
 (2) USE DEF *Pcri*
 (2) QED BY *AssSemProtect*
 (1) QED BY DEF *InvUnreachableSemProtect*

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM *ThmNextInvUnreachableSemProtect* \triangleq
InvUnreachableSemProtect \wedge [*Next*]_{vars} \Rightarrow *InvUnreachableSemProtect'*

PROOF

(1) HAVE *InvUnreachableSemProtect* \wedge [*Next*]_{vars}
 (1) USE DEF *InvUnreachableSemProtect*
 (1) USE DEF *InSem*
 (1) USE DEF *InvInSemProtect*

 (1) *InvType'* BY *ThmNextInvType*
 (1) *InvInSemProtect'* BY *ThmNextInvInSemProtect*
 (1) *InvUnreachableSemProtect!* goal'

 (2) USE DEF *PcIDLE*
 (2) USE DEF *PcRECOV1*, *PcRECOV2*, *PcRECOV3*
 (2) USE DEF *PcCHKPT1*, *PcCHKPT2*, *PcCHKPT3*, *PcCHKPT4*, *PcCHKPT5*

Stutter step.

(2)1. CASE *vars'* = *vars*
 (3) USE (2)1
 (3) USE DEF *vars*
 (3) QED OBVIOUS

Walk through all *Next* alternatives.

(2)2. CASE *NextObtainAccess*
 (3) USE *NextObtainAccess*
 (3) USE DEF *NextObtainAccess*
 (3) QED OBVIOUS

 (2)3. CASE *NextProveRevoke*

⟨3⟩ USE *NextProveRevoke*
 ⟨3⟩ USE DEF *NextProveRevoke*
 ⟨3⟩ QED OBVIOUS

(2)4. CASE *NextReboot*
 ⟨3⟩ USE *NextReboot*
 ⟨3⟩ USE DEF *NextReboot*
 ⟨3⟩ USE DEF *PcrLeq*
 ⟨3⟩ USE DEF *SemProtect*
 ⟨3⟩ USE DEF *SemReboot*
 ⟨3⟩ USE DEF *PcrInit*
 ⟨3⟩ USE DEF *Pcri*
 ⟨3⟩ USE *AssSemProtect*
 ⟨3⟩ QED OBVIOUS

(2)5. CASE *NextForgetSealTs*
 ⟨3⟩ USE *NextForgetSealTs*
 ⟨3⟩ USE DEF *NextForgetSealTs*
 ⟨3⟩ QED OBVIOUS

(2)6. CASE *NextExtendAppPcr*
 ⟨3⟩ USE *NextExtendAppPcr*
 ⟨3⟩ USE DEF *NextExtendAppPcr*
 ⟨3⟩ QED OBVIOUS

(2)7. CASE *NextExtendSemPcr*
 ⟨3⟩ USE *NextExtendSemPcr*
 ⟨3⟩ USE DEF *NextExtendSemPcr*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 ⟨4⟩ *SemProtect* \in *PcrBY ThmSemProtectIsPcr*
 ⟨4⟩ QED BY *ThmPcrExtendSourceUnreachable*
 ⟨3⟩ QED OBVIOUS

(2)8. CASE *NextExtendSealPcr*
 ⟨3⟩ USE *NextExtendSealPcr*
 ⟨3⟩ USE DEF *NextExtendSealPcr*
 ⟨3⟩ QED OBVIOUS

(2)9. CASE *NextIncBootCtr*
 ⟨3⟩ USE *NextIncBootCtr*
 ⟨3⟩ USE DEF *NextIncBootCtr*
 ⟨3⟩ QED OBVIOUS

(2)10. CASE *NextEnterSemRecov*
 ⟨3⟩ USE *NextEnterSemRecov*
 ⟨3⟩ USE DEF *NextEnterSemRecov*
 ⟨3⟩ QED OBVIOUS

- ⟨2⟩11. CASE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩12. CASE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *InvType*
 - ⟨3⟩ USE DEF *Pcrx*
 - ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 - ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩13. CASE *NextSemRecov2*
 - ⟨3⟩ USE *NextSemRecov2*
 - ⟨3⟩ USE DEF *NextSemRecov2*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩14. CASE *NextSemRecov3*
 - ⟨3⟩ USE *NextSemRecov3*
 - ⟨3⟩ USE DEF *NextSemRecov3*
 - ⟨3⟩ USE DEF *InvType*
 - ⟨3⟩ USE DEF *Pcrx*
 - ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 - ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩15. CASE *NextSealTs*
 - ⟨3⟩ USE *NextSealTs*
 - ⟨3⟩ USE DEF *NextSealTs*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩16. CASE *NextEnterSemChkpt*
 - ⟨3⟩ USE *NextEnterSemChkpt*
 - ⟨3⟩ USE DEF *NextEnterSemChkpt*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩17. CASE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩18. CASE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *InvType*
 - ⟨3⟩ USE DEF *Pcrx*

⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 ⟨3⟩ QED OBVIOUS

⟨2⟩19. CASE *NextSemChkpt2*
 ⟨3⟩ USE *NextSemChkpt2*
 ⟨3⟩ USE DEF *NextSemChkpt2*
 ⟨3⟩ QED OBVIOUS

⟨2⟩20. CASE *NextSemChkpt3*
 ⟨3⟩ USE *NextSemChkpt3*
 ⟨3⟩ USE DEF *NextSemChkpt3*
 ⟨3⟩ QED OBVIOUS

⟨2⟩21. CASE *NextSemChkpt4*
 ⟨3⟩ USE *NextSemChkpt4*
 ⟨3⟩ USE DEF *NextSemChkpt4*
 ⟨3⟩ QED OBVIOUS

⟨2⟩22. CASE *NextSemChkpt5*
 ⟨3⟩ USE *NextSemChkpt5*
 ⟨3⟩ USE DEF *NextSemChkpt5*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ USE DEF *Pcrx*
 ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩8, ⟨2⟩9, ⟨2⟩10,
 ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,
 ⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22
 DEF *Next*
 ⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvUnreachableSemProtect* \triangleq
 $Spec \Rightarrow \square InvUnreachableSemProtect$

PROOF

⟨1⟩ $Init \Rightarrow InvUnreachableSemProtect$ BY *ThmInitInvUnreachableSemProtect*
 ⟨1⟩ $InvUnreachableSemProtect \wedge [Next]_{vars} \Rightarrow InvUnreachableSemProtect'$
 BY *ThmNextInvUnreachableSemProtect*
 ⟨1⟩ USE DEF *Spec*
 ⟨1⟩ QED

PROOF OF INVARIANT *InvNvProtection*

It is an invariant of the specification.

THEOREM *ThmInvNvProtection* \triangleq

Spec $\Rightarrow \Box$ *InvNvProtection*

PROOF

$\langle 1 \rangle$ *InvInSemProtect* \wedge *InvUnreachableSemProtect* \Rightarrow *InvNvProtection*

$\langle 2 \rangle$ HAVE *InvInSemProtect* \wedge *InvUnreachableSemProtect*

$\langle 2 \rangle$ USE DEF *InvInSemProtect*

$\langle 2 \rangle$ USE DEF *InvUnreachableSemProtect*

$\langle 2 \rangle$ USE DEF *InvNvProtection*

$\langle 2 \rangle$ 1. CASE *InSem* BY $\langle 2 \rangle$ 1

$\langle 2 \rangle$ 2. CASE \neg *InSem*

Proof by contradiction.

$\langle 3 \rangle$ 1. CASE *semPcr* \neq *SemProtect* BY $\langle 3 \rangle$ 1

$\langle 3 \rangle$ 2. CASE *semPcr* = *SemProtect*

$\langle 4 \rangle$ 1. \neg *PcrLeq*(*semPcr*, *SemProtect*) BY $\langle 2 \rangle$ 2

$\langle 4 \rangle$ 2. *PcrLeq*(*semPcr*, *SemProtect*)

$\langle 5 \rangle$ *semPcr* \in *Pcr* BY *InvType* DEF *InvType*

$\langle 5 \rangle$ QED BY $\langle 3 \rangle$ 2, *ThmPcrLeqIsReflexive*

$\langle 4 \rangle$ QED BY $\langle 4 \rangle$ 1, $\langle 4 \rangle$ 2

$\langle 3 \rangle$ QED BY $\langle 3 \rangle$ 2, $\langle 3 \rangle$ 1

$\langle 2 \rangle$ QED BY $\langle 2 \rangle$ 2, $\langle 2 \rangle$ 1

$\langle 1 \rangle$ *Spec* $\Rightarrow \Box$ *InvInSemProtect* BY *ThmInvInSemProtect*

$\langle 1 \rangle$ *Spec* $\Rightarrow \Box$ *InvUnreachableSemProtect* BY *ThmInvUnreachableSemProtect*

$\langle 1 \rangle$ QED

PROOF OF INVARIANT *InvSignedTsLeqBoot*

If it holds in the initial state.

THEOREM $ThmInitInvSignedTsLeqBoot \triangleq$
 $Init \Rightarrow InvSignedTsLeqBoot$

PROOF

- (1) HAVE $Init$
- (1) $InvType$ BY $ThmInitInvType$
- (1) USE DEF $Init$
- (1) USE DEF $InvSignedTsLeqBoot$
- (1) $InvSignedTsLeqBoot!$ goal
 - (2) TAKE $ts \in tsvalues \cup \{chkptts\}$
 - (2) QED BY $ThmNullTsIsntSignedTs$
- (1) QED OBVIOUS

If it holds in the current state, and we perform a $Next$ action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM $ThmNextInvSignedTsLeqBoot \triangleq$
 $InvSignedTsLeqBoot \wedge [Next]_{vars} \Rightarrow InvSignedTsLeqBoot'$

PROOF

- (1) HAVE $InvSignedTsLeqBoot \wedge [Next]_{vars}$
- (1) USE DEF $InvSignedTsLeqBoot$
- (1) $InvType'$ BY $ThmNextInvType$
- (1) $InvSignedTsLeqBoot!$ goal'

Stutter step.

- (2)1. CASE $vars' = vars$
 - (3) USE (2)1
 - (3) USE DEF $vars$
 - (3) QED OBVIOUS

Walk through all $Next$ alternatives.

- (2)2. CASE $NextObtainAccess$
 - (3) USE $NextObtainAccess$
 - (3) USE DEF $NextObtainAccess$
 - (3) QED OBVIOUS
- (2)3. CASE $NextProveRevoke$
 - (3) USE $NextProveRevoke$
 - (3) USE DEF $NextProveRevoke$
 - (3) QED OBVIOUS
- (2)4. CASE $NextReboot$
 - (3) USE $NextReboot$
 - (3) USE DEF $NextReboot$

- ⟨3⟩ QED BY *ThmNullTsIsntSignedTs*
- ⟨2⟩5. CASE *NextForgetSealTs*
 - ⟨3⟩ USE *NextForgetSealTs*
 - ⟨3⟩ USE DEF *NextForgetSealTs*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩6. CASE *NextExtendAppPcr*
 - ⟨3⟩ USE *NextExtendAppPcr*
 - ⟨3⟩ USE DEF *NextExtendAppPcr*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩7. CASE *NextExtendSemPcr*
 - ⟨3⟩ USE *NextExtendSemPcr*
 - ⟨3⟩ USE DEF *NextExtendSemPcr*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩8. CASE *NextExtendSealPcr*
 - ⟨3⟩ USE *NextExtendSealPcr*
 - ⟨3⟩ USE DEF *NextExtendSealPcr*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩9. CASE *NextIncBootCtr*
 - ⟨3⟩ USE *NextIncBootCtr*
 - ⟨3⟩ USE DEF *NextIncBootCtr*
 - ⟨3⟩ USE DEF *InvType*
 - ⟨3⟩ USE DEF *SignedTs*
 - ⟨3⟩ $bootCtr \leq bootCtr + 1$ BY *ThmNatMore*
 - ⟨3⟩ QED BY *ThmNatLeqIsTransitive*
- ⟨2⟩10. CASE *NextEnterSemRecov*
 - ⟨3⟩ USE *NextEnterSemRecov*
 - ⟨3⟩ USE DEF *NextEnterSemRecov*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩11. CASE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩12. CASE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩13. CASE *NextSemRecov2*
 - ⟨3⟩ USE *NextSemRecov2*
 - ⟨3⟩ USE DEF *NextSemRecov2*
 - ⟨3⟩ QED OBVIOUS

- (2)14. CASE *NextSemRecov3*
 - ⟨3⟩ USE *NextSemRecov3*
 - ⟨3⟩ USE DEF *NextSemRecov3*
 - ⟨3⟩ QED OBVIOUS

- (2)15. CASE *NextSealTs*
 - ⟨3⟩ USE *NextSealTs*
 - ⟨3⟩ USE DEF *NextSealTs*
 - ⟨3⟩ DEFINE $ts \triangleq \text{NextSealTs!} : !ts$
 - ⟨3⟩ $ts.\text{bootCtr} \leq \text{bootCtr}$
 - ⟨4⟩ USE DEF *InvType*
 - ⟨4⟩ USE *ThmNatLeqIsReflexive*
 - ⟨4⟩ QED OBVIOUS
 - ⟨3⟩ QED OBVIOUS

- (2)16. CASE *NextEnterSemChkpt*
 - ⟨3⟩ USE *NextEnterSemChkpt*
 - ⟨3⟩ USE DEF *NextEnterSemChkpt*
 - ⟨3⟩ QED OBVIOUS

- (2)17. CASE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS

- (2)18. CASE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ QED OBVIOUS

- (2)19. CASE *NextSemChkpt2*
 - ⟨3⟩ USE *NextSemChkpt2*
 - ⟨3⟩ USE DEF *NextSemChkpt2*
 - ⟨3⟩ QED OBVIOUS

- (2)20. CASE *NextSemChkpt3*
 - ⟨3⟩ USE *NextSemChkpt3*
 - ⟨3⟩ USE DEF *NextSemChkpt3*
 - ⟨3⟩ USE DEF *InvType*
 - ⟨3⟩ USE DEF *SignedTs*
 - ⟨3⟩ $\text{bootCtr} < \text{bootCtr}'$
 - ⟨4⟩ USE *ThmNatLeqIsReflexive*
 - ⟨4⟩ USE *ThmNatMore*
 - ⟨4⟩ QED OBVIOUS
 - ⟨3⟩ USE *ThmNatLeqIsTransitive*
 - ⟨3⟩ QED OBVIOUS

- (2)21. CASE *NextSemChkpt4*

⟨3⟩ USE *NextSemChkpt4*
 ⟨3⟩ USE DEF *NextSemChkpt4*
 ⟨3⟩ QED OBVIOUS

 ⟨2⟩22. CASE *NextSemChkpt5*
 ⟨3⟩ USE *NextSemChkpt5*
 ⟨3⟩ USE DEF *NextSemChkpt5*
 ⟨3⟩ QED OBVIOUS

 ⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩8, ⟨2⟩9, ⟨2⟩10,
 ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,
 ⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22
 DEF *Next*
 ⟨1⟩ QED OBVIOUS

PROOF OF INVARIANT *InvUnforgeableSemHappy*

It holds in the initial state.

THEOREM *ThmInitInvUnforgeableSemHappy* \triangleq
Init \Rightarrow *InvUnforgeableSemHappy*

PROOF

⟨1⟩ HAVE *Init*
 ⟨1⟩ *InvType* BY *ThmInitInvType*
 ⟨1⟩ *InvInSemProtect* BY *ThmInitInvInSemProtect*
 ⟨1⟩ USE DEF *Init*
 ⟨1⟩ USE DEF *InSem*
 ⟨1⟩ \neg *InSem* OBVIOUS
 ⟨1⟩ \neg *PcrLeq*(*semPcr*, *SemHappy*)
 ⟨2⟩ USE DEF *PcrLeq*
 ⟨2⟩ USE DEF *PcrInit*
 ⟨2⟩ USE DEF *PcrExtend*
 ⟨2⟩ USE DEF *SemHappy*
 ⟨2⟩ USE DEF *SemReboot*
 ⟨2⟩ USE DEF *SemProtect*
 ⟨2⟩ USE *AssSemProtect*

- (2) QED OBVIOUS
- (1) QED BY DEF *InvUnforgeableSemHappy*

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM *ThmNextInvUnforgeableSemHappy* \triangleq
InvUnforgeableSemHappy \wedge [*Next*]_{vars} \Rightarrow *InvUnforgeableSemHappy'*

PROOF

- (1) HAVE *InvUnforgeableSemHappy* \wedge [*Next*]_{vars}
- (1) USE DEF *InvUnforgeableSemHappy*
- (1) USE DEF *InSem*
- (1) *InvType'* BY *ThmNextInvType*
- (1) *InvInSemProtect'* BY *ThmNextInvInSemProtect*
- (1) *InvUnforgeableSemHappy!* goal'
- (2) USE DEF *PcIDLE*
- (2) USE DEF *PcRECOV1*, *PcRECOV2*, *PcRECOV3*
- (2) USE DEF *PcCHKPT1*, *PcCHKPT2*, *PcCHKPT3*, *PcCHKPT4*, *PcCHKPT5*

Stutter step.

- (2)1. CASE *vars' = vars*
- (3) USE (2)1
- (3) USE DEF *vars*
- (3) QED OBVIOUS

Walk through all *Next* alternatives.

- (2)2. CASE *NextObtainAccess*
- (3) USE *NextObtainAccess*
- (3) USE DEF *NextObtainAccess*
- (3) QED OBVIOUS
- (2)3. CASE *NextProveRevoke*
- (3) USE *NextProveRevoke*
- (3) USE DEF *NextProveRevoke*
- (3) QED OBVIOUS
- (2)4. CASE *NextReboot*
- (3) USE *NextReboot*
- (3) USE DEF *NextReboot*
- (3) USE DEF *PcrLeq*
- (3) USE DEF *PcrInit*
- (3) USE DEF *PcrExtend*
- (3) USE DEF *SemHappy*
- (3) USE DEF *SemReboot*

⟨3⟩ USE DEF *SemProtect*
 ⟨3⟩ USE *AssSemProtect*
 ⟨3⟩ QED OBVIOUS

⟨2⟩5. CASE *NextForgetSealTs*
 ⟨3⟩ USE *NextForgetSealTs*
 ⟨3⟩ USE DEF *NextForgetSealTs*
 ⟨3⟩ QED OBVIOUS

⟨2⟩6. CASE *NextExtendAppPcr*
 ⟨3⟩ USE *NextExtendAppPcr*
 ⟨3⟩ USE DEF *NextExtendAppPcr*
 ⟨3⟩ QED OBVIOUS

⟨2⟩7. CASE *NextExtendSemPcr*
 ⟨3⟩ USE *NextExtendSemPcr*
 ⟨3⟩ USE DEF *NextExtendSemPcr*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ HAVE $\neg InSem'$
 ⟨3⟩1. CASE $semPcr = SemHappy$
 ⟨4⟩ USE ⟨3⟩1
 ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 ⟨3⟩2. CASE $\neg PcrLeq(semPcr, SemHappy)$
 ⟨4⟩ USE ⟨3⟩2
 ⟨4⟩ $SemHappy \in Pcr$ BY *ThmSemHappyIsPcr*
 ⟨4⟩ $\neg PcrLeq(semPcr', SemHappy)$ BY *ThmPcrExtendSourceUnreachable*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ QED BY ⟨3⟩1, ⟨3⟩2

⟨2⟩8. CASE *NextExtendSealPcr*
 ⟨3⟩ USE *NextExtendSealPcr*
 ⟨3⟩ USE DEF *NextExtendSealPcr*
 ⟨3⟩ QED OBVIOUS

⟨2⟩9. CASE *NextIncBootCtr*
 ⟨3⟩ USE *NextIncBootCtr*
 ⟨3⟩ USE DEF *NextIncBootCtr*
 ⟨3⟩ QED OBVIOUS

⟨2⟩10. CASE *NextEnterSemRecov*
 ⟨3⟩ USE *NextEnterSemRecov*
 ⟨3⟩ USE DEF *NextEnterSemRecov*
 ⟨3⟩ QED OBVIOUS

⟨2⟩11. CASE *NextSemRecov1 WhenCorrect*
 ⟨3⟩ USE *NextSemRecov1 WhenCorrect*
 ⟨3⟩ USE DEF *NextSemRecov1 WhenCorrect*
 ⟨3⟩ QED OBVIOUS

- (2)12. CASE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *InvType*
 - ⟨3⟩ USE DEF *Pcrx*
 - ⟨3⟩ *semPcr = SemProtectBY DEF InvInSemProtect*
 - ⟨3⟩ USE DEF *SemHappy*
 - ⟨3⟩ USE *AssSemHappy*
 - ⟨3⟩ USE *ThmPcrExtendIncompatible*
 - ⟨3⟩ QED OBVIOUS
- (2)13. CASE *NextSemRecov2*
 - ⟨3⟩ USE *NextSemRecov2*
 - ⟨3⟩ USE DEF *NextSemRecov2*
 - ⟨3⟩ QED OBVIOUS
- (2)14. CASE *NextSemRecov3*
 - ⟨3⟩ USE *NextSemRecov3*
 - ⟨3⟩ USE DEF *NextSemRecov3*
 - ⟨3⟩ *semPcr = SemProtectBY DEF InvInSemProtect*
 - ⟨3⟩ USE DEF *SemHappy*
 - ⟨3⟩ QED OBVIOUS
- (2)15. CASE *NextSealTs*
 - ⟨3⟩ USE *NextSealTs*
 - ⟨3⟩ USE DEF *NextSealTs*
 - ⟨3⟩ QED OBVIOUS
- (2)16. CASE *NextEnterSemChkpt*
 - ⟨3⟩ USE *NextEnterSemChkpt*
 - ⟨3⟩ USE DEF *NextEnterSemChkpt*
 - ⟨3⟩ QED OBVIOUS
- (2)17. CASE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS
- (2)18. CASE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *InvType*
 - ⟨3⟩ USE DEF *Pcrx*
 - ⟨3⟩ *semPcr = SemProtectBY DEF InvInSemProtect*
 - ⟨3⟩ USE DEF *SemHappy*
 - ⟨3⟩ USE *AssSemHappy*
 - ⟨3⟩ USE *ThmPcrExtendIncompatible*
 - ⟨3⟩ QED OBVIOUS

⟨2⟩19. CASE *NextSemChkpt2*
 ⟨3⟩ USE *NextSemChkpt2*
 ⟨3⟩ USE DEF *NextSemChkpt2*
 ⟨3⟩ QED OBVIOUS

⟨2⟩20. CASE *NextSemChkpt3*
 ⟨3⟩ USE *NextSemChkpt3*
 ⟨3⟩ USE DEF *NextSemChkpt3*
 ⟨3⟩ QED OBVIOUS

⟨2⟩21. CASE *NextSemChkpt4*
 ⟨3⟩ USE *NextSemChkpt4*
 ⟨3⟩ USE DEF *NextSemChkpt4*
 ⟨3⟩ QED OBVIOUS

⟨2⟩22. CASE *NextSemChkpt5*
 ⟨3⟩ USE *NextSemChkpt5*
 ⟨3⟩ USE DEF *NextSemChkpt5*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ USE DEF *Pcrx*
 ⟨3⟩ *semPcr = SemProtect* BY DEF *InvInSemProtect*
 ⟨3⟩ USE DEF *SemHappy*
 ⟨3⟩ USE *AssSemHappy*
 ⟨3⟩ USE *ThmPcrExtendIncompatible*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩8, ⟨2⟩9, ⟨2⟩10,
 ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,
 ⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22
 DEF *Next*
 ⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvUnforgeableSemHappy* \triangleq
Spec \Rightarrow \square *InvUnforgeableSemHappy*

PROOF

⟨1⟩ *Init* \Rightarrow *InvUnforgeableSemHappy* BY *ThmInitInvUnforgeableSemHappy*
 ⟨1⟩ *InvUnforgeableSemHappy* \wedge $[Next]_{vars} \Rightarrow$ *InvUnforgeableSemHappy'*
 BY *ThmNextInvUnforgeableSemHappy*
 ⟨1⟩ USE DEF *Spec*
 ⟨1⟩ QED

 PROOF OF INVARIANT $InvUnforgeableSealReboot$

It holds in the initial state.

THEOREM $ThmInitInvUnforgeableSealReboot \triangleq$
 $Init \Rightarrow InvUnforgeableSealReboot$

PROOF

- $\langle 1 \rangle$ HAVE $Init$
- $\langle 1 \rangle$ $InvType$ BY $ThmInitInvType$
- $\langle 1 \rangle$ $sealPcr = SealReboot$ BY DEF $Init$
- $\langle 1 \rangle$ QED BY DEF $InvUnforgeableSealReboot$

If it holds in the current state, and we perform a $Next$ action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM $ThmNextInvUnforgeableSealReboot \triangleq$
 $InvUnforgeableSealReboot \wedge [Next]_{vars} \Rightarrow InvUnforgeableSealReboot'$

PROOF

- $\langle 1 \rangle$ HAVE $InvUnforgeableSealReboot \wedge [Next]_{vars}$
- $\langle 1 \rangle$ USE DEF $InvUnforgeableSealReboot$
- $\langle 1 \rangle$ $InvType'$ BY $ThmNextInvType$
- $\langle 1 \rangle$ $InvUnforgeableSealReboot' ! goal'$
- $\langle 2 \rangle$ USE DEF $PcIDLE$
- $\langle 2 \rangle$ USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- $\langle 2 \rangle$ USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Stutter step.

- $\langle 2 \rangle 1.$ CASE $vars' = vars$
- $\langle 3 \rangle$ USE $\langle 2 \rangle 1$
- $\langle 3 \rangle$ USE DEF $vars$
- $\langle 3 \rangle$ QED OBVIOUS

Walk through all $Next$ alternatives.

- $\langle 2 \rangle 2.$ CASE $NextObtainAccess$
- $\langle 3 \rangle$ USE $NextObtainAccess$
- $\langle 3 \rangle$ USE DEF $NextObtainAccess$
- $\langle 3 \rangle$ QED OBVIOUS
- $\langle 2 \rangle 3.$ CASE $NextProveRevoke$
- $\langle 3 \rangle$ USE $NextProveRevoke$
- $\langle 3 \rangle$ USE DEF $NextProveRevoke$
- $\langle 3 \rangle$ QED OBVIOUS

- ⟨2⟩4. CASE *NextReboot*
 - ⟨3⟩ USE *NextReboot*
 - ⟨3⟩ USE DEF *NextReboot*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩5. CASE *NextForgetSealTs*
 - ⟨3⟩ USE *NextForgetSealTs*
 - ⟨3⟩ USE DEF *NextForgetSealTs*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩6. CASE *NextExtendAppPcr*
 - ⟨3⟩ USE *NextExtendAppPcr*
 - ⟨3⟩ USE DEF *NextExtendAppPcr*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩7. CASE *NextExtendSemPcr*
 - ⟨3⟩ USE *NextExtendSemPcr*
 - ⟨3⟩ USE DEF *NextExtendSemPcr*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩8. CASE *NextExtendSealPcr*
 - ⟨3⟩ USE *NextExtendSealPcr*
 - ⟨3⟩ USE DEF *NextExtendSealPcr*
 - ⟨3⟩ $sealPcr \in Pcr$ BY DEF *InvType*
 - ⟨3⟩ $SealReboot \in Pcr$ BY *ThmSealRebootIsPcr*
 - ⟨3⟩ QED BY *ThmPcrExtendFromEqOrNotleq*

- ⟨2⟩9. CASE *NextIncBootCtr*
 - ⟨3⟩ USE *NextIncBootCtr*
 - ⟨3⟩ USE DEF *NextIncBootCtr*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩10. CASE *NextEnterSemRecov*
 - ⟨3⟩ USE *NextEnterSemRecov*
 - ⟨3⟩ USE DEF *NextEnterSemRecov*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩11. CASE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩12. CASE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩13. CASE *NextSemRecov2*
 - ⟨3⟩ USE *NextSemRecov2*

- ⟨3⟩ USE DEF *NextSemRecov2*
- ⟨3⟩ QED OBVIOUS
- (2)14. CASE *NextSemRecov3*
 - ⟨3⟩ USE *NextSemRecov3*
 - ⟨3⟩ USE DEF *NextSemRecov3*
 - ⟨3⟩ QED OBVIOUS
- (2)15. CASE *NextSealTs*
 - ⟨3⟩ USE *NextSealTs*
 - ⟨3⟩ USE DEF *NextSealTs*
 - ⟨3⟩ USE DEF *Pcrx*
 - ⟨3⟩ $sealPcr \in Pcr$ BY DEF *InvType*
 - ⟨3⟩ $SealReboot \in Pcr$ BY *ThmSealRebootIsPcr*
 - ⟨3⟩ QED BY *ThmPcrExtendFromEqOrNotleq*
- (2)16. CASE *NextEnterSemChkpt*
 - ⟨3⟩ USE *NextEnterSemChkpt*
 - ⟨3⟩ USE DEF *NextEnterSemChkpt*
 - ⟨3⟩ QED OBVIOUS
- (2)17. CASE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS
- (2)18. CASE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 - ⟨3⟩ QED OBVIOUS
- (2)19. CASE *NextSemChkpt2*
 - ⟨3⟩ USE *NextSemChkpt2*
 - ⟨3⟩ USE DEF *NextSemChkpt2*
 - ⟨3⟩ QED OBVIOUS
- (2)20. CASE *NextSemChkpt3*
 - ⟨3⟩ USE *NextSemChkpt3*
 - ⟨3⟩ USE DEF *NextSemChkpt3*
 - ⟨3⟩ QED OBVIOUS
- (2)21. CASE *NextSemChkpt4*
 - ⟨3⟩ USE *NextSemChkpt4*
 - ⟨3⟩ USE DEF *NextSemChkpt4*
 - ⟨3⟩ QED OBVIOUS
- (2)22. CASE *NextSemChkpt5*
 - ⟨3⟩ USE *NextSemChkpt5*
 - ⟨3⟩ USE DEF *NextSemChkpt5*

⟨3⟩ QED OBVIOUS

⟨2⟩ QED

BY ⟨2⟩1,

⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩8, ⟨2⟩9, ⟨2⟩10,

⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,

⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22

DEF *Next*

⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvUnforgeableSealReboot* \triangleq

Spec $\Rightarrow \square$ *InvUnforgeableSealReboot*

PROOF

⟨1⟩ *Init* \Rightarrow *InvUnforgeableSealReboot* BY *ThmInitInvUnforgeableSealReboot*

⟨1⟩ *InvUnforgeableSealReboot* \wedge [*Next*]_{vars} \Rightarrow *InvUnforgeableSealReboot'*

BY *ThmNextInvUnforgeableSealReboot*

⟨1⟩ USE DEF *Spec*

⟨1⟩ QED

PROOF OF INVARIANT *InvProperLastExtension*

It holds in the initial state.

THEOREM *ThmInitInvProperLastExtension* \triangleq

Init \Rightarrow *InvProperLastExtension*

PROOF

⟨1⟩ HAVE *Init*

⟨1⟩ *InvType* BY *ThmInitInvType*

⟨1⟩ QED BY DEF *InvProperLastExtension*, *Init*

It it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

THEOREM *ThmNextInvProperLastExtension* \triangleq

InvProperLastExtension \wedge [*Next*]_{vars} \Rightarrow *InvProperLastExtension'*

PROOF

⟨1⟩ HAVE $InvProperLastExtension \wedge [Next]_{vars}$
 ⟨1⟩ USE DEF $InvProperLastExtension$
 ⟨1⟩ $InvType'$ BY $ThmNextInvType$
 ⟨1⟩ $InvProperLastExtension ! goal'$

Wow, this is an easy one.

⟨2⟩ USE DEF $vars$
 ⟨2⟩ USE DEF $NextObtainAccess$
 ⟨2⟩ USE DEF $NextProveRevoke$
 ⟨2⟩ USE DEF $NextReboot$
 ⟨2⟩ USE DEF $NextForgetSealTs$
 ⟨2⟩ USE DEF $NextExtendAppPcr$
 ⟨2⟩ USE DEF $NextExtendSemPcr$
 ⟨2⟩ USE DEF $NextExtendSealPcr$
 ⟨2⟩ USE DEF $NextIncBootCtr$
 ⟨2⟩ USE DEF $NextEnterSemRecov$
 ⟨2⟩ USE DEF $NextSemRecov1 WhenCorrect$
 ⟨2⟩ USE DEF $NextSemRecov1 WhenIncorrect$
 ⟨2⟩ USE DEF $NextSemRecov2$
 ⟨2⟩ USE DEF $NextSemRecov3$
 ⟨2⟩ USE DEF $NextSealTs$
 ⟨2⟩ USE DEF $NextEnterSemChkpt$
 ⟨2⟩ USE DEF $NextSemChkpt1 WhenCorrect$
 ⟨2⟩ USE DEF $NextSemChkpt1 WhenIncorrect$
 ⟨2⟩ USE DEF $NextSemChkpt2$
 ⟨2⟩ USE DEF $NextSemChkpt3$
 ⟨2⟩ USE DEF $NextSemChkpt4$
 ⟨2⟩ USE DEF $NextSemChkpt5$
 ⟨2⟩ QED BY DEF $Next$
 ⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM $ThmInvProperLastExtension \triangleq$
 $Spec \Rightarrow \square InvProperLastExtension$

PROOF

⟨1⟩ $Init \Rightarrow InvProperLastExtension$ BY $ThmInitInvProperLastExtension$
 ⟨1⟩ $InvProperLastExtension \wedge [Next]_{vars} \Rightarrow InvProperLastExtension'$
 BY $ThmNextInvProperLastExtension$
 ⟨1⟩ USE DEF $Spec$
 ⟨1⟩ QED

PROOF OF INVARIANT $InvOneLog$

It holds in the initial state.

THEOREM $ThmInitInvOneLog \triangleq$
 $Init \Rightarrow InvOneLog$

PROOF

- (1) HAVE $Init$
- (1) $InvType$ BY $ThmInitInvType$
- (1) $InvSignedTsLeqBoot$ BY $ThmInitInvSignedTsLeqBoot$
- (1) $InvInSemProtect$ BY $ThmInitInvInSemProtect$
- (1) $InvUnforgeableSemHappy$ BY $ThmInitInvUnforgeableSemHappy$
- (1) $InvUnforgeableSealReboot$ BY $ThmInitInvUnforgeableSealReboot$
- (1) $InvProperLastExtension$ BY $ThmInitInvProperLastExtension$
- (1) USE DEF $Init$
- (1) USE DEF $InitNv$
- (1) USE DEF $PcIDLE$
- (1) USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- (1) USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$
- (1) $LogInNv$ BY DEF $LogInNv$
- (1) $\neg LogInApp$
- (2) $semPcr \neq SemHappy$
- (3) $semPcr.init \neq SemHappy.init$
- (4) $semPcr.init \neq SemProtect.init$
- (5) USE DEF $SemReboot$
- (5) USE DEF $SemProtect$
- (5) USE DEF $PcrInit$
- (5) QED BY $AssSemProtect$
- (4) USE DEF $SemHappy$
- (4) QED BY DEF $PcrExtend$
- (3) QED BY DEF Pcr
- (2) QED BY DEF $LogInApp$
- (1) $\neg LogInTs \wedge AllCurrentTs = \{\}$
- (2) $\neg CheckTsIsCurrent(chkptts)$
- (3) USE DEF $CheckTsIsCurrent$
- (3) QED BY $ThmNullTsIsntSignedTs$
- (2) USE DEF $AllCurrentTs$
- (2) QED BY DEF $LogInTs$
- (1) $InvOneLog!goal!obtains$ BY DEF $IsOnLog$
- (1) $InvOneLog!goal!revokes$ BY DEF $IsOnLog$
- (1) $InvVerifiableRevocation$ BY DEF $InvVerifiableRevocation$
- (1) QED BY DEF $InvOneLog$

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

THEOREM $ThmNextInvOneLog \triangleq$
 $InvOneLog \wedge [Next]_{vars} \Rightarrow InvOneLog'$

PROOF

- (1) HAVE $InvOneLog \wedge [Next]_{vars}$
- (1) $InvType$ BY DEF $InvOneLog$
- (1) $InvSignedTsLeqBoot$ BY DEF $InvOneLog$
- (1) $InvInSemProtect$ BY DEF $InvOneLog$
- (1) $InvUnforgeableSemHappy$ BY DEF $InvOneLog$
- (1) $InvUnforgeableSealReboot$ BY DEF $InvOneLog$
- (1) $InvProperLastExtension$ BY DEF $InvOneLog$
- (1) $InvType'$ BY $ThmNextInvType$
- (1) $InvSignedTsLeqBoot'$ BY $ThmNextInvSignedTsLeqBoot$
- (1) $InvInSemProtect'$ BY $ThmNextInvInSemProtect$
- (1) $InvUnforgeableSemHappy'$ BY $ThmNextInvUnforgeableSemHappy$
- (1) $InvUnforgeableSealReboot'$ BY $ThmNextInvUnforgeableSealReboot$
- (1) $InvProperLastExtension'$ BY $ThmNextInvProperLastExtension$
- (1) $InvOneLog!goal'$
- (2) USE DEF $PcIDLE$
- (2) USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- (2) USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Stutter step.

- (2) 1. CASE $vars' = vars$
 - (3) USE (2)1
 - (3) USE DEF $vars$
 - (3) UNCHANGED $CheckTsIsCurrent(chkpts)$ BY DEF $CheckTsIsCurrent$
 - (3) UNCHANGED $AllCurrentTs$
 - (4) USE DEF $AllCurrentTs$
 - (4) USE DEF $CheckTsIsCurrent$
 - (4) QED BY DEF $LogInTs$
 - (3) UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
 - (3) UNCHANGED $LogInNv$ BY DEF $LogInNv$
 - (3) UNCHANGED $LogInApp$ BY DEF $LogInApp$
 - (3) UNCHANGED $LogInTs$ BY DEF $LogInTs$
 - (3) $InvOneLog!goal!obtains'$ BY DEF $IsOnLog, InvOneLog$
 - (3) $InvOneLog!goal!revokes'$ BY DEF $IsOnLog, InvOneLog$
 - (3) $InvVerifiableRevocation'$ BY DEF $InvVerifiableRevocation, InvOneLog$
 - (3) QED BY DEF $InvOneLog$

NextObtainAccess or *NextProveRevoke*

⟨2⟩2. CASE *NextObtainAccess* \vee *NextProveRevoke*

⟨3⟩ USE ⟨2⟩2

⟨3⟩ USE DEF *NextObtainAccess*

⟨3⟩ USE DEF *NextProveRevoke*

⟨3⟩ UNCHANGED *CheckTsIsCurrent*(*chkptts*) BY DEF *CheckTsIsCurrent*

⟨3⟩ UNCHANGED *AllCurrentTs*

⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ QED BY DEF *LogInTs*

⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*

⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

⟨3⟩ *LogInApp*

⟨4⟩ CASE *NextObtainAccess*

NextObtainAccess is predicated on the fact that the log is in the application pcr.

This depends on

\wedge *BugObtainAccessNoCheckHappy* \triangleq FALSE

\wedge *BugObtainAccessNoCheckSeal* \triangleq FALSE

⟨5⟩ *BugObtainAccessNoCheckHappy* = FALSE BY DEF *BugObtainAccessNoCheckHappy*

⟨5⟩ *BugObtainAccessNoCheckSeal* = FALSE BY DEF *BugObtainAccessNoCheckSeal*

⟨5⟩ QED BY DEF *LogInApp*

⟨4⟩ CASE *NextProveRevoke*

NextProveRevoke is predicated on the fact that the log is in the application pcr.

This depends on

\wedge *BugProveRevokeNoCheckHappy* \triangleq FALSE

\wedge *BugProveRevokeNoCheckSeal* \triangleq FALSE

⟨5⟩ *BugProveRevokeNoCheckHappy* = FALSE BY DEF *BugProveRevokeNoCheckHappy*

⟨5⟩ *BugProveRevokeNoCheckSeal* = FALSE BY DEF *BugProveRevokeNoCheckSeal*

⟨5⟩ QED BY DEF *LogInApp*

⟨4⟩ QED OBVIOUS

⟨3⟩ *InvOneLog!goal!obtains' \wedge InvOneLog!goal!revokes'*

Since the log is in the application pcr, putting a copy of the application pcr into *obtains* or into *revokes* preserves the invariant that everything in *obtains* \cup *revokes* can reach the log.

⟨4⟩ *PcrLeq*(*appPcr*, *appPcr*) BY *ThmPcrLeqIsReflexive* DEF *InvType*

⟨4⟩ QED BY DEF *IsOnLog*, *InvOneLog*

⟨3⟩ *InvVerifiableRevocation'*

Since we only add an element to *obtains* \cup *revokes* when the log is in the application pcr, we know that all elements in *obtains* \cup *revokes* in the new state must be on the log, which we can check as \leq *app* pcr.

So we proceed with proof by contradiction. Assuming that verifiable deletion will be violated in the new state, we pick the $o \in$ *obtains* and $r \in$ *revokes* whose *PcrPrior*'s are the same. But since both o and r must be \leq *app* pcr, this means that their last extension must be the same. This contradicts the assumption that OBTAIN is different from REVOKE.

⟨4⟩ CASE *InvVerifiableRevocation'* OBVIOUS

⟨4⟩ CASE \neg *InvVerifiableRevocation'*

⟨5⟩ PICK $o \in \text{obtains}'$, $r \in \text{revokes}'$: $\text{PcrPrior}(o) = \text{PcrPrior}(r)$
 BY DEF *InvVerifiableRevocation*
 ⟨5⟩ DEFINE $p \triangleq \text{PcrPrior}(o)$
 ⟨5⟩ DEFINE $xo \triangleq \text{PcrLastExtension}(o)$
 ⟨5⟩ DEFINE $xr \triangleq \text{PcrLastExtension}(r)$
 ⟨5⟩ $p \in \text{Pcr}$ BY *ThmPcrPriorIsPcr* DEF *InvType*, *InvProperLastExtension*
 ⟨5⟩ $xo = \text{PcrxOBTAIN}$ BY DEF *InvProperLastExtension*
 ⟨5⟩ $xr = \text{PcrxREVOKE}$ BY DEF *InvProperLastExtension*
 ⟨5⟩ $o = \text{PcrExtend}(p, xo)$ BY *ThmPcrExtendPriorLast* DEF *InvType*, *InvProperLastExtension*
 ⟨5⟩ $r = \text{PcrExtend}(p, xr)$ BY *ThmPcrExtendPriorLast* DEF *InvType*, *InvProperLastExtension*
 ⟨5⟩ $\text{PcrLeq}(o, \text{appPcr}')$ BY DEF *IsOnLog*, *InvOneLog*
 ⟨5⟩ $\text{PcrLeq}(r, \text{appPcr}')$ BY DEF *IsOnLog*, *InvOneLog*
 ⟨5⟩ $xo \in \text{Pcrx}$ BY DEF *Pcrx*
 ⟨5⟩ $xr \in \text{Pcrx}$ BY DEF *Pcrx*
 ⟨5⟩ $xo = xr$
 The prover needs a lot of help to focus its attention.
 ⟨6⟩ HIDE DEF p
 ⟨6⟩ HIDE DEF xo
 ⟨6⟩ HIDE DEF xr
 ⟨6⟩ $\text{appPcr}' \in \text{Pcr}$ BY DEF *InvType*
 ⟨6⟩ QED BY *ThmPcrExtendLeqAnticollision*
 ⟨5⟩ $\text{PcrxOBTAIN} \neq \text{PcrxREVOKE}$ BY *AssObtainNeqRevoke*
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ QED BY DEF *InvOneLog*

NextReboot

⟨2⟩3. CASE *NextReboot*
 ⟨3⟩ USE *NextReboot*
 ⟨3⟩ USE DEF *NextReboot*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, which erases any log that had been in the application *pcr*.

⟨3⟩ $\neg \text{LogInApp}'$
 ⟨4⟩ $\text{semPcr}' \neq \text{SemHappy}$
 ⟨5⟩ $\text{semPcr}'.\text{init} \neq \text{SemHappy}.init$
 ⟨6⟩ $\text{semPcr}'.init \neq \text{SemProtect}.init$
 ⟨7⟩ USE DEF *SemReboot*
 ⟨7⟩ USE DEF *SemProtect*
 ⟨7⟩ USE DEF *PcrInit*
 ⟨7⟩ QED BY *AssSemProtect*
 ⟨6⟩ USE DEF *SemHappy*
 ⟨6⟩ QED BY DEF *PcrExtend*
 ⟨5⟩ QED BY DEF *Pcr*
 ⟨4⟩ QED BY DEF *LogInApp*

Overwrites *chkptts* with an unsigned *ts*, which might erase a log that had been in the seal attestations.

```

<3> LogInTs' ⇒ LogInTs
  <4> HAVE LogInTs'
  <4> AllCurrentTs' ≠ {} BY DEF LogInTs
  <4> USE DEF AllCurrentTs
  <4> USE DEF CheckTsIsCurrent
  <4> ¬CheckTsIsCurrent(chkptts)' BY ThmNullTsIsntSignedTs
  <4> ∃ ts ∈ tsvalues' : CheckTsIsCurrent(ts) OBVIOUS
  <4> QED BY DEF LogInTs

```

Any remaining current seal attestations existed previously, so they must contain the same log.

```

<3> ∀ ts1, ts2 ∈ AllCurrentTs' : ts1.appPcr = ts2.appPcr
  <4> TAKE ts1, ts2 ∈ AllCurrentTs'
  <4> USE DEF AllCurrentTs
  <4> USE DEF CheckTsIsCurrent
  <4> ¬CheckTsIsCurrent(chkptts)' BY ThmNullTsIsntSignedTs
  <4> ts1 ∈ AllCurrentTs OBVIOUS
  <4> ts2 ∈ AllCurrentTs OBVIOUS
  <4> QED BY DEF InvOneLog

```

If there are any remaining current seal attestations, the log in them has to be the same as before.

```

<3> LogInTs' ⇒ UNCHANGED CurrentTsLog
  <4> HAVE LogInTs'
  <4> ∃ ts ∈ AllCurrentTs' : ts ∈ AllCurrentTs
    <5> USE DEF LogInTs
    <5> USE DEF AllCurrentTs
    <5> USE DEF CheckTsIsCurrent
    <5> ¬CheckTsIsCurrent(chkptts)' BY ThmNullTsIsntSignedTs
    <5> ∀ ts ∈ AllCurrentTs' : ts ∈ AllCurrentTs OBVIOUS
    <5> QED OBVIOUS
  <4> ∀ ts1 ∈ AllCurrentTs' :
    ∀ ts0 ∈ AllCurrentTs :
      ts1.appPcr = ts0.appPcr
    BY DEF InvOneLog
  <4> QED BY DEF CurrentTsLog
<3> InvOneLog!goal!obtains'   BY DEF IsOnLog, InvOneLog
<3> InvOneLog!goal!revokes'   BY DEF IsOnLog, InvOneLog
<3> InvVerifiableRevocation' BY DEF InvVerifiableRevocation, InvOneLog
<3> QED BY DEF InvOneLog

```

NextForgetSealTs

```

<2>4. CASE NextForgetSealTs
  <3> USE NextForgetSealTs
  <3> USE DEF NextForgetSealTs
  <3> UNCHANGED LogInNv       BY DEF LogInNv
  <3> UNCHANGED LogInApp      BY DEF LogInApp

```

⟨3⟩ UNCHANGED $CheckTsIsCurrent(chkptts)$ BY DEF $CheckTsIsCurrent$

Forgets a seal attestation, which might erase a log that had been in the seal attestations.

⟨3⟩ $LogInTs' \Rightarrow LogInTs$
 ⟨4⟩ HAVE $LogInTs'$
 ⟨4⟩ $AllCurrentTs' \neq \{\}$ BY DEF $LogInTs$
 ⟨4⟩ USE DEF $AllCurrentTs$
 ⟨4⟩ USE DEF $CheckTsIsCurrent$
 ⟨4⟩ QED BY DEF $LogInTs$

Any remaining current seal attestations existed previously, so they must contain the same log.

⟨3⟩ $\forall ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$
 ⟨4⟩ TAKE $ts1, ts2 \in AllCurrentTs'$
 ⟨4⟩ USE DEF $AllCurrentTs$
 ⟨4⟩ USE DEF $CheckTsIsCurrent$
 ⟨4⟩ $ts1 \in AllCurrentTs$ OBVIOUS
 ⟨4⟩ $ts2 \in AllCurrentTs$ OBVIOUS
 ⟨4⟩ QED BY DEF $InvOneLog$

If $chkptts$ contains a current seal attestation, then the log is in the seal attestations.

⟨3⟩ $CheckTsIsCurrent(chkptts)' \Rightarrow LogInTs'$
 ⟨4⟩ USE DEF $LogInTs$
 ⟨4⟩ USE DEF $AllCurrentTs$
 ⟨4⟩ USE DEF $CheckTsIsCurrent$
 ⟨4⟩ QED OBVIOUS

If there are any remaining current seal attestations, the log in them has to be the same as before.

⟨3⟩ $LogInTs' \Rightarrow$ UNCHANGED $CurrentTsLog$
 ⟨4⟩ HAVE $LogInTs'$
 ⟨4⟩ $\exists ts \in AllCurrentTs' : ts \in AllCurrentTs$
 ⟨5⟩ USE DEF $LogInTs$
 ⟨5⟩ USE DEF $AllCurrentTs$
 ⟨5⟩ USE DEF $CheckTsIsCurrent$
 ⟨5⟩ $\forall ts \in AllCurrentTs' : ts \in AllCurrentTs$ OBVIOUS
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ $\forall ts1 \in AllCurrentTs' :$
 $\forall ts0 \in AllCurrentTs :$
 $ts1.appPcr = ts0.appPcr$
 BY DEF $InvOneLog$
 ⟨4⟩ QED BY DEF $CurrentTsLog$
 ⟨3⟩ $InvOneLog!goal!obtains'$ BY DEF $IsOnLog, InvOneLog$
 ⟨3⟩ $InvOneLog!goal!revokes'$ BY DEF $IsOnLog, InvOneLog$
 ⟨3⟩ $InvVerifiableRevocation'$ BY DEF $InvVerifiableRevocation, InvOneLog$
 ⟨3⟩ QED BY DEF $InvOneLog$

NextExtendAppPcr

⟨2⟩5. CASE *NextExtendAppPcr*
 ⟨3⟩ USE *NextExtendAppPcr*
 ⟨3⟩ USE DEF *NextExtendAppPcr*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!obtains' ∧ InvOneLog!goal!revokes'*
 ⟨4⟩ CASE \neg *LogInApp*
 ⟨5⟩ USE DEF *LogInApp*
 ⟨5⟩ USE DEF *IsOnLog, InvOneLog*
NextExtendAppPcr is predicated on not being in sem, so none of the sem clauses apply.
 ⟨5⟩ USE DEF *InSem*
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ CASE *LogInApp*
 ⟨5⟩ USE DEF *LogInApp*
 If the log is in the application pcr, extending the application pcr preserves the fact that all entries in *obtains* \cup *revokes* can reach it.
 ⟨5⟩ $\forall p \in \text{obtains} \cup \text{revokes} : \text{LogInApp} \Rightarrow \text{PcrLeq}(p, \text{appPcr}')$
 ⟨6⟩ UNCHANGED (*obtains* \cup *revokes*) OBVIOUS
 ⟨6⟩ TAKE $p \in \text{obtains} \cup \text{revokes}$
 ⟨6⟩ HAVE *LogInApp*
 ⟨6⟩ *PcrLeq*(p, appPcr) BY DEF *IsOnLog, InvOneLog*
 ⟨6⟩ *PcrLeq*($\text{appPcr}, \text{appPcr}'$) BY *ThmPcrExtendLeq* DEF *InvType*
 ⟨6⟩ QED BY *ThmPcrLeqIsTransitive* DEF *InvType*
 ⟨5⟩ QED BY DEF *IsOnLog, InvOneLog*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ QED BY DEF *InvOneLog*

NextExtendSemPcr

⟨2⟩6. CASE *NextExtendSemPcr*
 ⟨3⟩ USE *NextExtendSemPcr*
 ⟨3⟩ USE DEF *NextExtendSemPcr*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, which erases any log that had been in the application pcr.

⟨3⟩ $\neg \text{LogInApp}'$
 ⟨4⟩ $\text{semPcr}' \neq \text{SemHappy}$
 ⟨5⟩ $\text{semPcr} = \text{SemHappy} \vee \neg \text{PcrLeq}(\text{semPcr}, \text{SemHappy})$
 BY DEF *InvUnforgeableSemHappy*
 ⟨5⟩ $\text{semPcr} \in \text{Pcr}$ BY DEF *InvType*
 ⟨5⟩ $\text{SemHappy} \in \text{Pcr}$ BY *ThmSemHappyIsPcr*
 ⟨5⟩ $\neg \text{PcrLeq}(\text{semPcr}', \text{SemHappy})$ BY *ThmPcrExtendFromEqOrNotleq*
 ⟨5⟩ QED BY *ThmPcrLeqIsReflexive*
 ⟨4⟩ QED BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextExtendSealPcr

⟨2⟩7. CASE *NextExtendSealPcr*
 ⟨3⟩ USE *NextExtendSealPcr*
 ⟨3⟩ USE DEF *NextExtendSealPcr*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SealReboot* if we had it, which erases any log that had been in the application pcr.

⟨3⟩ $\neg \text{LogInApp}'$
 ⟨4⟩ $\text{sealPcr}' \neq \text{SealReboot}$
 ⟨5⟩ $\text{sealPcr} = \text{SealReboot} \vee \neg \text{PcrLeq}(\text{sealPcr}, \text{SealReboot})$
 BY DEF *InvUnforgeableSealReboot*
 ⟨5⟩ $\text{sealPcr} \in \text{Pcr}$ BY DEF *InvType*
 ⟨5⟩ $\text{SealReboot} \in \text{Pcr}$ BY *ThmSealRebootIsPcr*
 ⟨5⟩ $\neg \text{PcrLeq}(\text{sealPcr}', \text{SealReboot})$ BY *ThmPcrExtendFromEqOrNotleq*
 ⟨5⟩ QED BY *ThmPcrLeqIsReflexive*
 ⟨4⟩ QED BY DEF *LogInApp*

⟨3⟩ UNCHANGED $LogInTs$
 ⟨4⟩ USE DEF $AllCurrentTs$
 ⟨4⟩ USE DEF $CheckTsIsCurrent$
 ⟨4⟩ QED BY DEF $LogInTs$
 ⟨3⟩ $InvOneLog!goal!obtains'$ BY DEF $IsOnLog, InvOneLog$
 ⟨3⟩ $InvOneLog!goal!revokes'$ BY DEF $IsOnLog, InvOneLog$
 ⟨3⟩ $InvVerifiableRevocation'$ BY DEF $InvVerifiableRevocation, InvOneLog$
 ⟨3⟩ QED BY DEF $InvOneLog$

NextIncBootCtr

⟨2⟩8. CASE $NextIncBootCtr$
 ⟨3⟩ USE $NextIncBootCtr$
 ⟨3⟩ USE DEF $NextIncBootCtr$
 ⟨3⟩ UNCHANGED $LogInNv$ BY DEF $LogInNv$
 ⟨3⟩ UNCHANGED $LogInApp$ BY DEF $LogInApp$

Since no signed ts seal can have a $bootCtr$ greater than the current $bootCtr$, incrementing $bootCtr$ erases any log that had been in a seal attestation.

⟨3⟩ $\neg LogInTs'$
 ⟨4⟩ USE DEF $AllCurrentTs$
 ⟨4⟩ USE DEF $CheckTsIsCurrent$
 ⟨4⟩ $\forall ts \in tvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootCtr'$
 ⟨5⟩ TAKE $ts \in tvalues' \cup \{chkptts'\}$
 ⟨5⟩ $ts \in tvalues \cup \{chkptts\}$ OBVIOUS
 ⟨5⟩ HAVE $ts \in SignedTs$
 ⟨5⟩ $ts.bootCtr \leq bootCtr$ BY DEF $InvSignedTsLeqBoot$
 ⟨5⟩ $ts.bootCtr \in Nat$ BY DEF $SignedTs$
 ⟨5⟩ $bootCtr \in Nat$ BY DEF $InvType$
 ⟨5⟩ $bootCtr' \in Nat$ BY DEF $InvType$
 ⟨5⟩ $bootCtr < bootCtr'$ BY $ThmNatInc$
 ⟨5⟩ $ts.bootCtr < bootCtr'$ BY $ThmNatLeqLt$
 ⟨5⟩ QED BY $ThmNatLeqXorGt, ThmNatLeqIsReflexive$
 ⟨4⟩ QED BY DEF $LogInTs$

Erases all current ts seal attestations.

⟨3⟩ $AllCurrentTs' = \{\} \wedge \neg CheckTsIsCurrent(chkptts)'$
 ⟨4⟩ USE DEF $AllCurrentTs$
 ⟨4⟩ USE DEF $CheckTsIsCurrent$
 ⟨4⟩ QED BY DEF $LogInTs$
 ⟨3⟩ USE DEF $InSem$
 ⟨3⟩ $InvOneLog!goal!obtains'$ BY DEF $IsOnLog, InvOneLog$
 ⟨3⟩ $InvOneLog!goal!revokes'$ BY DEF $IsOnLog, InvOneLog$
 ⟨3⟩ $InvVerifiableRevocation'$ BY DEF $InvVerifiableRevocation, InvOneLog$
 ⟨3⟩ QED BY DEF $InvOneLog$

NextEnterSemRecov

(2)9. CASE *NextEnterSemRecov*
 (3) USE *NextEnterSemRecov*
 (3) USE DEF *NextEnterSemRecov*
 (3) USE DEF *InSem*
 (3) UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 (3) UNCHANGED *AllCurrentTs*
 (4) USE DEF *AllCurrentTs*
 (4) USE DEF *CheckTsIsCurrent*
 (4) QED BY DEF *LogInTs*
 (3) UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 (3) UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, which erases any log that had been in the application pcr.

(3) $\neg \text{LogInApp}'$
 (4) *semPcr' \neq SemHappy*
 (5) USE DEF *SemHappy*
 (5) USE DEF *SemProtect*
 (5) USE DEF *Pcri*
 (5) USE DEF *Pcrx*
 (5) USE *ThmPcrInitIsPcr*
 (5) USE *ThmPcrExtendIsPcr*
 (5) *PcrLeq(SemProtect, SemHappy)* BY *ThmPcrExtendLeq*
 (5) $\neg \text{PcrLeq}(SemHappy, SemProtect)$ BY *ThmPcrExtendSelfUnreachable*
 (5) QED BY *ThmPcrLeqIsAntisymmetric*
 (4) QED BY DEF *LogInApp*
 (3) UNCHANGED *LogInTs*
 (4) USE DEF *AllCurrentTs*
 (4) USE DEF *CheckTsIsCurrent*
 (4) QED BY DEF *LogInTs*
 (3) *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 (3) *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 (3) *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 (3) QED BY DEF *InvOneLog*

NextSemRecov1WhenCorrect

(2)10. CASE *NextSemRecov1WhenCorrect*
 (3) USE *NextSemRecov1WhenCorrect*
 (3) USE DEF *NextSemRecov1WhenCorrect*
 (3) UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 (3) UNCHANGED *AllCurrentTs*
 (4) USE DEF *AllCurrentTs*
 (4) USE DEF *CheckTsIsCurrent*
 (4) QED BY DEF *LogInTs*
 (3) UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

EnterSemRecovPredicate guarantees that the log is in the *nv* ram.

This depends on *BugRecovNoCheckCur* \triangleq FALSE

⟨3⟩ *LogInNv*
 ⟨4⟩ USE DEF *EnterSemRecovPredicate*
 ⟨4⟩ *BugRecovNoCheckCur* = FALSE BY DEF *BugRecovNoCheckCur*
 ⟨4⟩ QED BY DEF *LogInNv*

EnterSemRecovPredicate guarantees that the application pcr equals the log saved in the *nv* ram.

This depends on *BugRecovNoCheckApp* \triangleq FALSE

⟨3⟩ *appPcr* = *nv.appPcr*
 ⟨4⟩ USE DEF *EnterSemRecovPredicate*
 ⟨4⟩ *BugRecovNoCheckApp* = FALSE BY DEF *BugRecovNoCheckApp*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemRecov1WhenIncorrect

⟨2⟩ 11. CASE *NextSemRecov1WhenIncorrect*
 ⟨3⟩ USE *NextSemRecov1WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemRecov1WhenIncorrect*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending sem pcr with Unhappy results in something other than *SemHappy*, which indicates that the log is not in the application pcr.

⟨3⟩ \neg *LogInApp'*
 ⟨4⟩ *semPcr' \neq SemHappy*
 ⟨5⟩ USE DEF *SemHappy*
 ⟨5⟩ USE DEF *SemProtect*
 ⟨5⟩ USE DEF *PcrI*
 ⟨5⟩ USE DEF *Pcrx*
 ⟨5⟩ USE *ThmPcrInitIsPcr*
 ⟨5⟩ USE *ThmPcrExtendIsPcr*
 ⟨5⟩ *semPcr* = *SemProtect* BY DEF *InvInSemProtect, InSem*
 ⟨5⟩ USE *AssSemHappy*
 ⟨5⟩ QED BY *ThmPcrExtendAnticollision*

⟨4⟩ QED BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemRecov2

⟨2⟩ 12. CASE *NextSemRecov2*
 ⟨3⟩ USE *NextSemRecov2*
 ⟨3⟩ USE DEF *NextSemRecov2*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

Clearing *nv* current erases the log from the *nv* ram.

This depends on *BugRecovNoClrCur* \triangleq FALSE

⟨3⟩ \neg *LogInNv'*
 ⟨4⟩ \neg *nv'.current*
 ⟨5⟩ *BugRecovNoClrCur* = FALSE BY DEF *BugRecovNoClrCur*
 ⟨5⟩ QED BY DEF *InvType, Nv*
 ⟨4⟩ QED BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemRecov3

⟨2⟩ 13. CASE *NextSemRecov3*
 ⟨3⟩ USE *NextSemRecov3*
 ⟨3⟩ USE DEF *NextSemRecov3*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending sem pcr to *SemHappy* puts the log in the application pcr, provided that the seal pcr contains *SealReboot*.

But in the current state the log has no domicile. So the fact that its domicile might be the application pcr in the next state does not require a proof that it is not living anywhere else, since we get that for free.

⟨3⟩ *LogInApp'* ∈ BOOLEAN BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!*goal! *obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!*goal! *revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSealTs

⟨2⟩ 14. CASE *NextSealTs*
 ⟨3⟩ USE *NextSealTs*
 ⟨3⟩ USE DEF *NextSealTs*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SealReboot* if we had it, which erases any log that had been in the application pcr.

This depends on *BugSealNoExt* \triangleq FALSE

⟨3⟩ \neg *LogInApp'*
 ⟨4⟩ *sealPcr' ≠ SealReboot*
 ⟨5⟩ *sealPcr = SealReboot* \vee \neg *PcrLeq(sealPcr, SealReboot)*
 BY DEF *InvUnforgeableSealReboot*
 ⟨5⟩ *sealPcr* ∈ *Pcr* BY DEF *InvType*
 ⟨5⟩ *SealReboot* ∈ *Pcr* BY *ThmSealRebootIsPcr*
 ⟨5⟩ \neg *PcrLeq(sealPcr', SealReboot)*
 ⟨6⟩ *sealPcr' = PcrExtend(sealPcr, PcrxSEAL)*
 ⟨7⟩ *BugSealNoExt = FALSE* BY DEF *BugSealNoExt*
 ⟨7⟩ QED OBVIOUS
 ⟨6⟩ USE DEF *Pcrx*
 ⟨6⟩ QED BY *ThmPcrExtendFromEqOrNotleq*
 ⟨5⟩ QED BY *ThmPcrLeqIsReflexive*
 ⟨4⟩ QED BY DEF *LogInApp*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*

If the log was not in the application pcr, the resulting seal attestation will not be valid, so there is no change in *AllCurrentTs* or *LogSummaryInTs*.

⟨3⟩ CASE \neg *LogInApp*
 ⟨4⟩ UNCHANGED *AllCurrentTs*
 ⟨5⟩ USE DEF *AllCurrentTs*
 ⟨5⟩ USE DEF *CheckTsIsCurrent*

Use PICK to make *ts* a CONSTANT so that \neg *CheckTsIsCurrent(ts)'* means the *ts* picked now evaluated with *CheckTsIsCurrent* in the next state.

⟨5⟩ PICK $ts \in SignedTs : ts = NextSealTs! : !ts$ BY DEF *InvType*, *SignedTs*
 ⟨5⟩ $\forall ts1 \in tsvalues' : CheckTsIsCurrent(ts1)' \Rightarrow ts1 \in tsvalues$
 ⟨6⟩ $tsvalues' = tsvalues \cup \{ts\}$ OBVIOUS
 ⟨6⟩ $\neg CheckTsIsCurrent(ts)'$ BY DEF *LogInApp*
 ⟨6⟩ QED OBVIOUS
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*
 ⟨4⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨4⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog*, *InvOneLog*
 ⟨4⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog*, *InvOneLog*
 ⟨4⟩ QED BY DEF *InvOneLog*

If the log was in the application pcr, the resulting seal attestation will be valid. But then the old *AllCurrentTs* had to be empty, since the log could not have been in the seal attestations.

⟨3⟩ CASE *LogInApp*

Use PICK to make ts a CONSTANT so that $CheckTsIsCurrent(ts)'$ means the ts picked now evaluated with $CheckTsIsCurrent$ in the next state.

⟨4⟩ PICK $ts \in SignedTs : ts = NextSealTs! : !ts$ BY DEF *InvType*, *SignedTs*
 ⟨4⟩ $CheckTsIsCurrent(ts)'$
 ⟨5⟩ $CheckTsIsCurrent(ts)$ BY DEF *CheckTsIsCurrent*, *LogInApp*
 ⟨5⟩ QED BY DEF *CheckTsIsCurrent*
 ⟨4⟩ $AllCurrentTs' = \{ts\}$
 ⟨5⟩ $\forall ts1 \in tsvalues \cup \{chkptts'\} : \neg CheckTsIsCurrent(ts1)'$
 ⟨6⟩ $AllCurrentTs = \{\}$
 ⟨7⟩ $\neg LogInTs$ BY DEF *InvOneLog*
 ⟨7⟩ QED BY DEF *LogInTs*
 ⟨6⟩ $\neg CheckTsIsCurrent(chkptts)'$
 ⟨7⟩ $\neg CheckTsIsCurrent(chkptts)$ BY DEF *AllCurrentTs*
 ⟨7⟩ QED BY DEF *CheckTsIsCurrent*
 ⟨6⟩ $\forall ts1 \in tsvalues : \neg CheckTsIsCurrent(ts1)'$
 ⟨7⟩ $\forall ts1 \in tsvalues : \neg CheckTsIsCurrent(ts1)$ BY DEF *AllCurrentTs*
 ⟨7⟩ QED BY DEF *CheckTsIsCurrent*
 ⟨6⟩ QED BY DEF *CheckTsIsCurrent*
 ⟨5⟩ $tsvalues' = tsvalues \cup \{ts\}$ OBVIOUS
 ⟨5⟩ $ts \in AllCurrentTs'$ BY DEF *AllCurrentTs*
 ⟨5⟩ QED BY DEF *AllCurrentTs*
 ⟨4⟩ $\forall ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$ OBVIOUS
 ⟨4⟩ $LogInTs'$ BY DEF *LogInTs*
 ⟨4⟩ $CurrentTsLog' = ts.appPcr$ BY DEF *CurrentTsLog*
 ⟨4⟩ $InvOneLog!goal!obtains'$ BY DEF *IsOnLog*, *InvOneLog*
 ⟨4⟩ $InvOneLog!goal!revokes'$ BY DEF *IsOnLog*, *InvOneLog*
 ⟨4⟩ QED BY DEF *InvOneLog*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ 15. CASE *NextEnterSemChkpt*
 ⟨3⟩ USE *NextEnterSemChkpt*
 ⟨3⟩ USE DEF *NextEnterSemChkpt*
 ⟨3⟩ USE DEF *InSem*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, so erases any log that might have been in the application pcr.

⟨3⟩ $\neg \text{LogInApp}'$
 ⟨4⟩ $\text{semPcr}' \neq \text{SemHappy}$
 ⟨5⟩ USE DEF *SemHappy*
 ⟨5⟩ USE DEF *SemProtect*
 ⟨5⟩ USE DEF *Pcri*
 ⟨5⟩ USE DEF *Pcrr*
 ⟨5⟩ USE *ThmPcrInitIsPcr*
 ⟨5⟩ USE *ThmPcrExtendIsPcr*
 ⟨5⟩ *PcrLeq*(*SemProtect*, *SemHappy*)BY *ThmPcrExtendLeq*
 ⟨5⟩ $\neg \text{PcrLeq}$ (*SemHappy*, *SemProtect*)BY *ThmPcrExtendSelfUnreachable*
 ⟨5⟩ QED BY *ThmPcrLeqIsAntisymmetric*
 ⟨4⟩ QED BY DEF *LogInApp*

Overwrites *chkptts* with a value from *tvalues*, so if *chkptts* had been the only seal log, we just erased it.

⟨3⟩ $\text{LogInTs}' \Rightarrow \text{LogInTs}$
 ⟨4⟩ HAVE *LogInTs'*
 ⟨4⟩ $\text{AllCurrentTs}' \neq \{\}$ BY DEF *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ $\text{chkptts}' \in \text{tvalues}'$ OBVIOUS
 ⟨4⟩ $\exists ts \in \text{tvalues}' : \text{CheckTsIsCurrent}(ts)$ OBVIOUS
 ⟨4⟩ QED BY DEF *LogInTs*

Any remaining current seal attestations existed previously, so they must contain the same log.

⟨3⟩ $\forall ts1, ts2 \in \text{AllCurrentTs}' : ts1.\text{appPcr} = ts2.\text{appPcr}$
 ⟨4⟩ TAKE $ts1, ts2 \in \text{AllCurrentTs}'$
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ $\text{chkptts}' \in \text{tvalues}$ OBVIOUS
 ⟨4⟩ $ts1 \in \text{AllCurrentTs}$ OBVIOUS
 ⟨4⟩ $ts2 \in \text{AllCurrentTs}$ OBVIOUS
 ⟨4⟩ QED BY DEF *InvOneLog*

If there are any remaining current seal attestations, the log in them has to be the same as before.

⟨3⟩ $\text{LogInTs}' \Rightarrow \text{UNCHANGED CurrentTsLog}$
 ⟨4⟩ HAVE *LogInTs'*
 ⟨4⟩ $\exists ts \in \text{AllCurrentTs}' : ts \in \text{AllCurrentTs}$
 ⟨5⟩ USE DEF *LogInTs*
 ⟨5⟩ USE DEF *AllCurrentTs*
 ⟨5⟩ USE DEF *CheckTsIsCurrent*
 ⟨5⟩ $\forall ts \in \text{AllCurrentTs}' : ts \in \text{AllCurrentTs}$ OBVIOUS

⟨5⟩ QED OBVIOUS
 ⟨4⟩ $\forall ts1 \in AllCurrentTs'$:
 $\forall ts0 \in AllCurrentTs$:
 $ts1.appPcr = ts0.appPcr$
 BY DEF *InvOneLog*
 ⟨4⟩ QED BY DEF *CurrentTsLog*
 ⟨3⟩ *InvOneLog!*goal!obtains' BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvOneLog!*goal!revokes' BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation!* BY DEF *InvVerifiableRevocation*, *InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt1WhenCorrect

⟨2⟩ 16. CASE *NextSemChkpt1WhenCorrect*
 ⟨3⟩ USE *NextSemChkpt1WhenCorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1WhenCorrect*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

EnterSemChkptPredicate guarantees that the log is in the seal attestations (in particular, in *chkptts*).

This depends on

$\wedge BugChkptNoCheckTsHappy \triangleq FALSE$
 $\wedge BugChkptNoCheckTsSeal \triangleq FALSE$
 $\wedge BugChkptNoCheckTsCtr \triangleq FALSE$

⟨3⟩ $LogInTs \wedge CheckTsIsCurrent(chkptts) \wedge CurrentTsLog = chkptts.appPcr$
 ⟨4⟩ USE DEF *EnterSemChkptPredicate*
 ⟨4⟩ $BugChkptNoCheckTsHappy = FALSE$ BY DEF *BugChkptNoCheckTsHappy*
 ⟨4⟩ $BugChkptNoCheckTsSeal = FALSE$ BY DEF *BugChkptNoCheckTsSeal*
 ⟨4⟩ $BugChkptNoCheckTsCtr = FALSE$ BY DEF *BugChkptNoCheckTsCtr*
 ⟨4⟩ *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨4⟩ $AllCurrentTs \neq \{\}$ BY DEF *AllCurrentTs*
 ⟨4⟩ $CurrentTsLog = chkptts.appPcr$
 ⟨5⟩ $\forall ts \in AllCurrentTs : ts.appPcr = chkptts.appPcr$
 BY DEF *AllCurrentTs*, *InvOneLog*
 ⟨5⟩ QED BY DEF *CurrentTsLog*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!*goal!obtains' BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvOneLog!*goal!revokes' BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation!* BY DEF *InvVerifiableRevocation*, *InvOneLog*

⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt1 WhenIncorrect

⟨2⟩ 17. CASE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending *sem pcr* with *Unhappy* results in something other than *SemHappy*, which indicates that the log is not in the application *pcr*.

⟨3⟩ \neg *LogInApp'*
 ⟨4⟩ $semPcr' \neq SemHappy$
 ⟨5⟩ USE DEF *SemHappy*
 ⟨5⟩ USE DEF *SemProtect*
 ⟨5⟩ USE DEF *Pcri*
 ⟨5⟩ USE DEF *Pcrr*
 ⟨5⟩ USE *ThmPcrInitIsPcr*
 ⟨5⟩ USE *ThmPcrExtendIsPcr*
 ⟨5⟩ $semPcr = SemProtect$ BY DEF *InvInSemProtect, InSem*
 ⟨5⟩ USE *AssSemHappy*
 ⟨5⟩ QED BY *ThmPcrExtendAnticollision*
 ⟨4⟩ QED BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt2

⟨2⟩ 18. CASE *NextSemChkpt2*
 ⟨3⟩ USE *NextSemChkpt2*
 ⟨3⟩ USE DEF *NextSemChkpt2*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv*
 ⟨4⟩ USE DEF *InvType*
 ⟨4⟩ USE DEF *Nv*
 ⟨4⟩ QED BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

Storing the log from *chkptts* to the *nv* ram.

This depends on *BugChkptSaveCurApp* \triangleq FALSE

⟨3⟩ *nv'.appPcr* = *chkptts.appPcr*
 ⟨4⟩ *BugChkptSaveCurApp* = FALSE BY DEF *BugChkptSaveCurApp*
 ⟨4⟩ QED BY DEF *InvType*, *Nv*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation*, *InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt3

⟨2⟩ 19. CASE *NextSemChkpt3*
 ⟨3⟩ USE *NextSemChkpt3*
 ⟨3⟩ USE DEF *NextSemChkpt3*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*

Since no signed *ts* seal can have a *bootCtr* greater than the current *bootCtr*, incrementing *bootCtr* erases any log that might have been in a seal attestation.

This depends on *BugChkptNoIncCtr* \triangleq FALSE

⟨3⟩ \neg *LogInTs'*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ $\forall ts \in tsvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootCtr'$
 ⟨5⟩ TAKE $ts \in tsvalues' \cup \{chkptts'\}$
 ⟨5⟩ $ts \in tsvalues \cup \{chkptts\}$ OBVIOUS
 ⟨5⟩ HAVE $ts \in SignedTs$
 ⟨5⟩ $ts.bootCtr \leq bootCtr$ BY DEF *InvSignedTsLeqBoot*
 ⟨5⟩ $ts.bootCtr \in Nat$ BY DEF *SignedTs*
 ⟨5⟩ $bootCtr \in Nat$ BY DEF *InvType*
 ⟨5⟩ $bootCtr' \in Nat$ BY DEF *InvType*
 ⟨5⟩ $bootCtr < bootCtr'$
 ⟨6⟩ *BugChkptNoIncCtr* = FALSE BY DEF *BugChkptNoIncCtr*
 ⟨6⟩ QED BY *ThmNatInc*
 ⟨5⟩ $ts.bootCtr < bootCtr'$ BY *ThmNatLeqLt*
 ⟨5⟩ QED BY *ThmNatLeqXorGt*, *ThmNatLeqIsReflexive*

⟨4⟩ QED BY DEF *LogInTs*

Erases all current *ts* seal attestations.

⟨3⟩ $AllCurrentTs' = \{\} \wedge \neg CheckTsIsCurrent(chkptts)'$

⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ QED BY DEF *LogInTs*

⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*

⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*

⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*

⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt4

⟨2⟩20. CASE *NextSemChkpt4*

⟨3⟩ USE *NextSemChkpt4*

⟨3⟩ USE DEF *NextSemChkpt4*

⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*

⟨3⟩ UNCHANGED *AllCurrentTs*

⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ QED BY DEF *LogInTs*

⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

Setting *nv* current indicates that the log is in the *nv* ram.

This depends on *BugChkptNoSetCur* \triangleq FALSE

⟨3⟩ *LogInNv'*

⟨4⟩ *nv'.current*

⟨5⟩ *BugChkptNoSetCur* = FALSE BY DEF *BugChkptNoSetCur*

⟨5⟩ QED BY DEF *InvType, Nv*

⟨4⟩ QED BY DEF *LogInNv*

⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*

⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

Since *nv.current* was changed, the prover needs to see the type of *nv* to know that the *appPcr* field did not change.

⟨3⟩ UNCHANGED *nv.appPcr* BY DEF *InvType, Nv*

⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*

⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*

⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*

⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt5

⟨2⟩21. CASE *NextSemChkpt5*

⟨3⟩ USE *NextSemChkpt5*

⟨3⟩ USE DEF *NextSemChkpt5*

⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*

⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending *sem pcr* with *Unhappy* results in something other than *SemHappy*, which indicates that the log is not in the application *pcr*.

⟨3⟩ $\neg \text{LogInApp}'$
 ⟨4⟩ $\text{semPcr}' \neq \text{SemHappy}$
 ⟨5⟩ USE DEF *SemHappy*
 ⟨5⟩ USE DEF *SemProtect*
 ⟨5⟩ USE DEF *Pcri*
 ⟨5⟩ USE DEF *Pcrx*
 ⟨5⟩ USE *ThmPcrInitIsPcr*
 ⟨5⟩ USE *ThmPcrExtendIsPcr*
 ⟨5⟩ $\text{semPcr} = \text{SemProtect}$ BY DEF *InvInSemProtect*, *InSem*
 ⟨5⟩ USE *AssSemHappy*
 ⟨5⟩ QED BY *ThmPcrExtendAnticollision*
 ⟨4⟩ QED BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation*, *InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩8, ⟨2⟩9,
 ⟨2⟩10, ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17,
 ⟨2⟩18, ⟨2⟩19, ⟨2⟩20, ⟨2⟩21
 DEF *Next*
 ⟨1⟩ QED BY DEF *InvOneLog*

It is an invariant of the specification.

THEOREM *ThmInvOneLog* \triangleq
Spec $\Rightarrow \Box \text{InvOneLog}$

PROOF

⟨1⟩ *Init* $\Rightarrow \text{InvOneLog}$ BY *ThmInitInvOneLog*
 ⟨1⟩ $\text{InvOneLog} \wedge [\text{Next}]_{\text{vars}} \Rightarrow \text{InvOneLog}'$
 BY *ThmNextInvOneLog*
 ⟨1⟩ QED BY *RuleINV1* DEF *Spec*

PROOF OF INVARIANT *InvAccessUndeniability*

It is an invariant of the specification.

THEOREM *ThmInvAccessUndeniability* \triangleq
 $Spec \Rightarrow \Box InvAccessUndeniability$

PROOF

- (1) $InvOneLog \Rightarrow InvAccessUndeniability$
 - (2) HAVE *InvOneLog*
 - (2) USE DEF *InvOneLog*
 - (2) USE DEF *InvAccessUndeniability*
 - (2) *BugAuditNoCheckHappy* = FALSEBY DEF *BugAuditNoCheckHappy*
 - (2) *BugAuditNoCheckSeal* = FALSEBY DEF *BugAuditNoCheckSeal*
 - (2) CASE $\neg LogInApp$ BY DEF *LogInApp* unable to audit
 - (2) CASE *LogInApp* BY DEF *IsOnLog* audit
 - (2) QED OBVIOUS

(1) $Spec \Rightarrow \Box InvOneLog$ BY *ThmInvOneLog*

(1) QED

PROOF OF INVARIANT *InvVerifiableRevocation*

It is an invariant of the specification.

THEOREM *ThmInvVerifiableRevocation* \triangleq
 $Spec \Rightarrow \Box InvVerifiableRevocation$

PROOF

- (1) $InvOneLog \Rightarrow InvVerifiableRevocation$
 - (2) HAVE *InvOneLog*
 - (2) USE DEF *InvOneLog*
 - (2) QED OBVIOUS

(1) $Spec \Rightarrow \Box InvOneLog$ BY *ThmInvOneLog*

⟨1⟩ QED



D Specification of Optimized Pasture

MODULE *OptPastureNode*

EXTENDS *Naturals, Sequences, FiniteSets*

Override one of the following definitions to introduce a bug in the specification.

BugObtainAccessNoCheckHappy \triangleq FALSE
BugProveRevokeNoCheckHappy \triangleq FALSE
BugRecovNoCheckApp \triangleq FALSE
BugRecovNoCheckCur \triangleq FALSE
BugRecovNoClrCur \triangleq FALSE
BugSealNoExt \triangleq FALSE
BugChkptNoCheckTsHappy \triangleq FALSE
BugChkptNoCheckTsCtr \triangleq FALSE
BugChkptSaveCurApp \triangleq FALSE
BugChkptNoIncCtr \triangleq FALSE
BugChkptNoSetCur \triangleq FALSE liveness bug; not actually a safety bug
BugAuditNoCheckHappy \triangleq FALSE

PCR INITIALIZATION VALUES

CONSTANT *PcriAPPBOOT* reboot initialization of app pcr
 CONSTANT *PcriSEMBOOT* reboot initialization of sem pcr
 CONSTANT *PcriSEMPROTECT* secure execution mode entry of sem pcr

Pcri \triangleq
 {
 PcriAPPBOOT,
 PcriSEMBOOT,
 PcriSEMPROTECT
 }

Initialization of sem pcr via boot and via secure execution mode entry must be different.

ASSUME *AssSemProtect* \triangleq *PcriSEMBOOT* \neq *PcriSEMPROTECT*

PCR EXTENSION VALUES

CONSTANT *PcrxHAPPY* recover is happy

CONSTANT *PcrxUNHAPPY* recover is unhappy or checkpoint is unhappy/finished

CONSTANT *PcrxSEAL* seal marker

CONSTANT *PcrxOBTAIN* obtain access operation

CONSTANT *PcrxREVOKE* revoke access operation

$$Pcrx \triangleq \{ \begin{array}{l} PcrxHAPPY, \\ PcrxUNHAPPY, \\ PcrxSEAL, \\ PcrxOBTAIN, \\ PcrxREVOKE \end{array} \}$$

Extension for obtain access and extension for revoke access must be different.

ASSUME *AssObtainNeqRevoke* $\triangleq PcrxOBTAIN \neq PcrxREVOKE$

Extension for happy and extension for unhappy must be different.

ASSUME *AssSemHappy* $\triangleq PcrxHAPPY \neq PcrxUNHAPPY$

PCR VALUES

A pcr value is modeled as an initialization followed by a sequence of extensions.

$$Pcr \triangleq [\begin{array}{l} init : Pcr, \\ extq : Seq(Pcrx) \end{array}]$$

Initial pcr value.

$$PcrInit(i) \triangleq [\begin{array}{l} init \mapsto i, \\ extq \mapsto \langle \rangle \end{array}]$$

Extend a pcr value.

$$PcrExtend(p, x) \triangleq$$

$$[$$

$$\begin{array}{l} \textit{init} \mapsto p.\textit{init}, \\ \textit{extq} \mapsto \textit{Append}(p.\textit{extq}, x) \end{array}$$

$$]$$

Number of extensions in a pcr value.

$$\textit{PcrLen}(p) \triangleq$$

$$\textit{Len}(p.\textit{extq})$$

$\textit{Pcr } s \text{ is } \leq \textit{Pcr } t$. This means that with zero or more extensions, you can extend s to reach t . This is a partial order relation.

$$\textit{PcrLeq}(s, t) \triangleq$$

LET

$$\begin{array}{l} \textit{sinit} \triangleq s.\textit{init} \\ \textit{sextq} \triangleq s.\textit{extq} \\ \textit{sn} \triangleq \textit{Len}(\textit{sextq}) \end{array}$$

$$\begin{array}{l} \textit{tinit} \triangleq t.\textit{init} \\ \textit{textq} \triangleq t.\textit{extq} \\ \textit{tn} \triangleq \textit{Len}(\textit{textq}) \end{array}$$

$$\textit{uextq} \triangleq \textit{SubSeq}(\textit{textq}, 1, \textit{sn})$$

IN

$$\begin{array}{l} \wedge \textit{sinit} = \textit{tinit} \\ \wedge \textit{sn} \leq \textit{tn} \\ \wedge \textit{sextq} = \textit{uextq} \end{array}$$

Determine if a pcr value has been extended.

$$\textit{PcrHasExtension}(p) \triangleq$$

$$\textit{PcrLen}(p) > 0$$

Assuming a pcr value has been extended, get the prior pcr value that this one was extended from. We assume the adversary can compute this by watching all pcr computations.

$$\textit{PcrPrior}(p) \triangleq$$

CASE $\textit{PcrHasExtension}(p) \rightarrow$

LET

$$n \triangleq \textit{Len}(p.\textit{extq}) - 1$$

IN

$$[$$

$$\begin{array}{l} \textit{init} \mapsto p.\textit{init}, \\ \textit{extq} \mapsto \textit{SubSeq}(p.\textit{extq}, 1, n) \end{array}$$

$$]$$

Assuming a pcr value has been extended, get the last extension. We assume the adversary can compute this by watching all pcr computations.

$$PcrLastExtension(p) \triangleq$$

$$\text{CASE } PcrHasExtension(p) \rightarrow$$

$$p.extq[Len(p.extq)]$$

WELL KNOWN PCR VALUES

Value of the application pcr attained by rebooting.

$$AppReboot \triangleq PcrInit(PcriAPPBOOT)$$

Value of the secure execution mode pcr attained by rebooting.

$$SemReboot \triangleq PcrInit(PcriSEMBOOT)$$

Value of the secure execution mode pcr attained by entering the protected module in secure execution mode. This value permits access to the Pasture protected *Nv* ram.

$$SemProtect \triangleq PcrInit(PcriSEMPROTECT)$$

Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.

$$SemHappy \triangleq PcrExtend(SemProtect, PcrxHAPPY)$$

PC VALUES

anywhere not in secure execution mode

$$PcIDLE \triangleq \text{"idle"}$$

steps in secure execution mode within recover

$$PcRECOV1 \triangleq \text{"recov1"}$$

$$PcRECOV2 \triangleq \text{"recov2"}$$

$$PcRECOV3 \triangleq \text{"recov3"}$$

$$PcRecov \triangleq \{PcRECOV1, PcRECOV2, PcRECOV3\}$$

steps in secure execution mode within checkpoint

$$PcCHKPT1 \triangleq \text{"chkpt1"}$$

$$PcCHKPT2 \triangleq \text{"chkpt2"}$$

$$PcCHKPT3 \triangleq \text{"chkpt3"}$$

$$PcCHKPT4 \triangleq \text{"chkpt4"}$$

$$PcCHKPT5 \triangleq \text{"chkpt5"}$$

$$PcChkpt \triangleq \{PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5\}$$

$$Pc \triangleq \{PcIDLE\} \cup PcRecov \cup PcChkpt$$

PROTECTED NV RAM STATE

$$Nv \triangleq [\begin{array}{l} appPcr : Pcr, \quad \text{copy of the application pcr} \\ current : BOOLEAN \quad \text{copy of application pcr is current} \end{array}]$$

$$InitNv \triangleq [\begin{array}{l} appPcr \mapsto AppReboot, \\ current \mapsto TRUE \end{array}]$$

SEAL OPERATION TRANSPORT SESSION STATE

We model the signed “seal operation” transport session as a record of the input values required in order for the transport session *TPM* signature to be valid.

$$SignedTs \triangleq [\begin{array}{l} semPcr : Pcr, \quad \text{copy of the secure execution mode pcr on entry} \\ appPcr : Pcr, \quad \text{copy of the application pcr on entry} \\ bootCtr : Nat \quad \text{copy of the reboot counter on entry} \end{array}]$$

The adversary cannot forge a correctly signed seal attestation. We model all incorrectly signed ones as the following single value.

$$NullTs \triangleq \text{CHOOSE } NullTs : NullTs \notin SignedTs$$

$$Ts \triangleq SignedTs \cup \{NullTs\}$$

STATE

VARIABLE *nv* Pasture's protected NV RAM region
 VARIABLE *appPcr* the application pcr
 VARIABLE *semPcr* the secure execution mode pcr
 VARIABLE *bootCtr* the reboot counter
 VARIABLE *pc* *pc*
 VARIABLE *chkptts* *ts* passed to sem within checkpoint
 VARIABLE *tsvalues* what *ts* values are known
 VARIABLE *obtains* decisions to obtain access
 VARIABLE *revokes* decisions to prove revoke access

Tuple of all variables.

$vars \triangleq \langle nv, appPcr, semPcr, bootCtr, pc, chkptts, tsvalues, obtains, revokes \rangle$

STATE PREDICATES

The node is currently in secure execution mode.

$InSem \triangleq pc \neq PcIDLE$

NEXT STATE RELATION

Employ a key binding to obtain access to read a message.

If the last extension to the application pcr was an OBTAIN operation, then in full generality there could have been a key bound to this application pcr value. So record the information that we obtained access to this key binding.

$NextObtainAccess \triangleq$

LET

$pcr1o \triangleq appPcr$ current app pcr

$pcr0 \triangleq PcrPrior(pcr1o)$ prior app pcr

$x \triangleq PcrLastExtension(pcr1o)$ presumed OBTAIN extension

IN

$\wedge \neg InSem$ must not be in secure execution mode

$\wedge PcrHasExtension(pcr1o)$ have an extension
 $\wedge x = PcrxOBTAIN$ last extension was OBTAIN

It is a bug to fail to bind the key such that it can be used for decryption only when the secure execution mode pcr is happy.

\wedge IF *BugObtainAccessNoCheckHappy* THEN TRUE ELSE
 $semPcr = SemHappy$
 $\wedge obtains' = obtains \cup \{pcr1o\}$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tvalues*
 \wedge UNCHANGED *revokes*

Construct a proof of revocation.

If the last extension to the application *pcr* was a REVOKE operation, then in full generality there could have been a key bound to the *pcr* value in which instead the last extension was an *OBTAIN*. But by extending with a REVOKE we have instead revoked the key binding. So record the information that we could construct a proof of revocation.

A proof of revocation consists of the following exhibits:

- (a) *pcr0*, a purported prior application *pcr* value
- (b) *pcr1r*, a purported current application *pcr* value
- (c) *x*, the REVOKE extension satisfying $pcr1r = PcrExtend(pcr0, x)$
- (d) a quote of the application $pcr = pcr1r$ with the sem $pcr = SemHappy$ and the seal $pcr = SealReboot$.

These exhibits suffice to prove revocation of any valid key binding to the application *pcr* value $PcrExtend(pcr0, OBTAIN)$.

$NextProveRevoke \triangleq$

LET
 $pcr1r \triangleq appPcr$ current app pcr
 $pcr0 \triangleq PcrPrior(pcr1r)$ prior app pcr
 $x \triangleq PcrLastExtension(pcr1r)$ presumed REVOKE extension

IN

$\wedge \neg InSem$ must not be in secure execution mode
 $\wedge PcrHasExtension(pcr1r)$ have an extension
 $\wedge x = PcrxREVOKE$ last extension was a REVOKE

It is a bug to fail to require the proof of revocation to quote the fact that the secure execution mode pcr is happy.

\wedge IF *BugProveRevokeNoCheckHappy* THEN TRUE ELSE
 $semPcr = SemHappy$
 $\wedge revokes' = revokes \cup \{pcr1r\}$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*

\wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tvalues*
 \wedge UNCHANGED *obtains*

Reboot the node.

This can happen at absolutely any time, due to adversarial action. However, if it happens without going through the proper seal and checkpoint actions, liveness may be lost.

Resetting *chkptts* to its initial value erases information and thus reduces the number of distinct states that model checking has to explore. But note that the adversary could always remember whatever value *chkptts* had before and call *sem* checkpoint with that value.

$NextReboot \triangleq$
 \wedge *appPcr'* = *AppReboot*
 \wedge *semPcr'* = *SemReboot*
 \wedge *pc'* = *PcIDLE*
 \wedge *chkptts'* = *NullTs*
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *tvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Forget one of the seal transport sessions.

This can happen at absolutely any time, and represents a loss of knowledge by the adversary which enables additional execution paths to fall within the model checking constraints.

$NextForgetSealTs \triangleq$
 $\exists ts \in tvalues :$
 \wedge *tvalues'* = *tvalues* \setminus {*ts*}
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Extend application pcr arbitrarily.

In proper execution, this action is performed as necessary after reboot to re-extend the application pcr to its last checkpoint value.

In proper execution, this action is performed as desired to decide upon reading or deleting messages.

The adversary can perform this action at any idle time.

$$\begin{aligned}
 \text{NextExtendAppPcr} &\triangleq \\
 &\wedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
 &\wedge \exists x \in \text{Pcrx} : \\
 &\wedge \text{appPcr}' = \text{PcrExtend}(\text{appPcr}, x) \\
 &\wedge \text{UNCHANGED } nv \\
 &\wedge \text{UNCHANGED } \text{semPcr} \\
 &\wedge \text{UNCHANGED } \text{bootCtr} \\
 &\wedge \text{UNCHANGED } pc \\
 &\wedge \text{UNCHANGED } \text{chkptts} \\
 &\wedge \text{UNCHANGED } \text{tsvalues} \\
 &\wedge \text{UNCHANGED } \text{obtains} \\
 &\wedge \text{UNCHANGED } \text{revokes}
 \end{aligned}$$

Extend secure execution mode pcr arbitrarily, due to adversarial action.

$$\begin{aligned}
 \text{NextExtendSemPcr} &\triangleq \\
 &\wedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
 &\wedge \exists x \in \text{Pcrx} : \\
 &\wedge \text{semPcr}' = \text{PcrExtend}(\text{semPcr}, x) \\
 &\wedge \text{UNCHANGED } nv \\
 &\wedge \text{UNCHANGED } \text{appPcr} \\
 &\wedge \text{UNCHANGED } \text{bootCtr} \\
 &\wedge \text{UNCHANGED } pc \\
 &\wedge \text{UNCHANGED } \text{chkptts} \\
 &\wedge \text{UNCHANGED } \text{tsvalues} \\
 &\wedge \text{UNCHANGED } \text{obtains} \\
 &\wedge \text{UNCHANGED } \text{revokes}
 \end{aligned}$$

Increment reboot counter arbitrarily, due to adversarial action.

$$\begin{aligned}
 \text{NextIncBootCtr} &\triangleq \\
 &\wedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\
 &\wedge \text{bootCtr}' = \text{bootCtr} + 1 \\
 &\wedge \text{UNCHANGED } nv \\
 &\wedge \text{UNCHANGED } \text{appPcr} \\
 &\wedge \text{UNCHANGED } \text{semPcr} \\
 &\wedge \text{UNCHANGED } pc \\
 &\wedge \text{UNCHANGED } \text{chkptts} \\
 &\wedge \text{UNCHANGED } \text{tsvalues}
 \end{aligned}$$

\wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Enter secure execution mode within recovery.

In proper execution, this action is performed during system boot after the application *pcr* has been re-extended to its last checkpoint value. This re-extension is performed by untrusted code that reads the necessary extension values from a stable log.

The adversary can perform this action at any idle time. But it will not do any good unless the application *pcr* contains the last checkpoint value and the last checkpoint value is marked as current.

NextEnterSemRecov \triangleq
 $\wedge \neg InSem$ must not be in secure execution mode
 $\wedge semPcr' = SemProtect$
 $\wedge pc' = PcRECOV1$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Predicate for correct entry to secure execution mode within recovery.

EnterSemRecovPredicate \triangleq
 It is a bug for recovery to fail to check that the application *pcr* has been restored to the value saved in the *nv* ram.
 \wedge IF *BugRecovNoCheckApp* THEN TRUE ELSE
 $nv.appPcr = appPcr$
 It is a bug for recovery to fail to check that *nv* ram claims that its saved application *pcr* is current.
 \wedge IF *BugRecovNoCheckCur* THEN TRUE ELSE
 $nv.current$

Secure execution mode within recovery step 1, when there is correct entry.

NextSemRecov1WhenCorrect \triangleq
 $\wedge pc = PcRECOV1$
 $\wedge EnterSemRecovPredicate$
 $\wedge pc' = PcRECOV2$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*

\wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Secure execution mode within recovery step 1, when there is incorrect entry.

NextSemRecov1WhenIncorrect \triangleq
 $\wedge pc = PcRECOV1$
 $\wedge \neg EnterSemRecovPredicate$
 $\wedge semPcr' = PcrExtend(semPcr, PcrxUNHAPPY)$
 $\wedge pc' = PcIDLE$
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Secure execution mode within recovery step 2. Record that the *nv* app pcr might no longer be current.

NextSemRecov2 \triangleq
 LET
 nvcurrent1 \triangleq
 It is a bug for recovery to fail to clear the *nv* ram current flag.
 IF *BugRecovNoClrCur* THEN *nv.current* ELSE
 FALSE
 IN
 $\wedge pc = PcRECOV2$
 $\wedge nv' = [nv \text{ EXCEPT } !.current = nvcurrent1]$
 $\wedge pc' = PcRECOV3$
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *semPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tsvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Secure execution mode within recovery step 3. Declare correct recovery happiness and exit secure execution mode.

NextSemRecov3 \triangleq
 $\wedge pc = PcRECOV3$
 $\wedge semPcr' = PcrExtend(semPcr, PcrxHAPPY)$
 $\wedge pc' = PcIDLE$
 \wedge UNCHANGED *nv*

\wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *tvalues*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Perform a “seal operation” and remember the signed transport session.

In proper execution, provided that the secure execution mode pcr shows that recovery was happy, this action is performed as part of checkpoint during system shutdown. Secure execution mode within checkpoint is then invoked with this transport session as data.

This transport session reads the values of the secure execution mode pcr, the application pcr, and the reboot counter. Then the secure execution mode pcr is extended so that no key bindings will be available until the next happy recovery.

The adversary can record all of the signed transport sessions and try to replay an earlier one to convince secure execution mode within checkpoint to save an old application pcr as “current”. Reading the reboot counter here, and incrementing it in secure execution mode within checkpoint, prevents that.

NextSealTs \triangleq

LET
 ts \triangleq
 [
 semPcr \mapsto *semPcr*, sem pcr on entry
 appPcr \mapsto *appPcr*, app pcr on entry
 bootCtr \mapsto *bootCtr* reboot ctr on entry
]

semPcr1 \triangleq

It is a bug for the “seal operation” to fail to extend the sem pcr.

IF *BugSealNoExt* THEN *semPcr* ELSE
 PcrExtend(*semPcr*, *PcrxSEAL*)

IN

\wedge \neg *InSem* must not be in secure execution mode
 \wedge *tvalues'* = *tvalues* \cup {*ts*}
 \wedge *semPcr'* = *semPcr1*
 \wedge UNCHANGED *nv*
 \wedge UNCHANGED *appPcr*
 \wedge UNCHANGED *bootCtr*
 \wedge UNCHANGED *pc*
 \wedge UNCHANGED *chkptts*
 \wedge UNCHANGED *obtains*
 \wedge UNCHANGED *revokes*

Enter secure execution mode within checkpoint.

In proper execution, this action is performed during system shutdown following the seal transport session action.

The adversary can perform this action at any idle time, feeding it any known seal transport session value.

$$\begin{aligned} \text{NextEnterSemChkpt} &\triangleq \\ &\wedge \neg \text{InSem} \quad \text{must not be in secure execution mode} \\ &\wedge \exists ts \in \text{tsvalues} : \quad \text{any known } ts \text{ value} \\ &\wedge \text{semPcr}' = \text{SemProtect} \\ &\wedge \text{pc}' = \text{PcCHKPT1} \\ &\wedge \text{chkptts}' = ts \\ &\wedge \text{UNCHANGED } nv \\ &\wedge \text{UNCHANGED } \text{appPcr} \\ &\wedge \text{UNCHANGED } \text{bootCtr} \\ &\wedge \text{UNCHANGED } \text{tsvalues} \\ &\wedge \text{UNCHANGED } \text{obtains} \\ &\wedge \text{UNCHANGED } \text{revokes} \end{aligned}$$

Predicate for correct entry to secure execution mode within checkpoint.

$$\begin{aligned} \text{EnterSemChkptPredicate} &\triangleq \\ &\wedge \text{chkptts} \in \text{SignedTs} \\ &\text{It is a bug to fail to check that the seal operation recorded that the secure execution mode pcr was happy.} \\ &\wedge \text{IF } \text{BugChkptNoCheckTsHappy} \text{ THEN TRUE ELSE} \\ &\quad \text{chkptts.semPcr} = \text{SemHappy} \\ &\text{It is a bug to fail to check that the seal operation recorded a reboot counter value that matches the current reboot counter.} \\ &\wedge \text{IF } \text{BugChkptNoCheckTsCtr} \text{ THEN TRUE ELSE} \\ &\quad \text{chkptts.bootCtr} = \text{bootCtr} \end{aligned}$$

Secure execution mode within checkpoint step 1, when there is correct entry.

$$\begin{aligned} \text{NextSemChkpt1WhenCorrect} &\triangleq \\ &\wedge \text{pc} = \text{PcCHKPT1} \\ &\wedge \text{EnterSemChkptPredicate} \\ &\wedge \text{pc}' = \text{PcCHKPT2} \\ &\wedge \text{UNCHANGED } nv \\ &\wedge \text{UNCHANGED } \text{appPcr} \\ &\wedge \text{UNCHANGED } \text{semPcr} \\ &\wedge \text{UNCHANGED } \text{bootCtr} \\ &\wedge \text{UNCHANGED } \text{chkptts} \\ &\wedge \text{UNCHANGED } \text{tsvalues} \\ &\wedge \text{UNCHANGED } \text{obtains} \\ &\wedge \text{UNCHANGED } \text{revokes} \end{aligned}$$

Secure execution mode within checkpoint step 1, when there is incorrect entry.

$$\begin{aligned}
\text{NextSemChkpt1WhenIncorrect} &\triangleq \\
&\wedge pc = PcCHKPT1 \\
&\wedge \neg \text{EnterSemChkptPredicate} \\
&\wedge semPcr' = PcrExtend(semPcr, PcrxUNHAPPY) \\
&\wedge pc' = PcIDLE \\
&\wedge \text{UNCHANGED } nv \\
&\wedge \text{UNCHANGED } appPcr \\
&\wedge \text{UNCHANGED } bootCtr \\
&\wedge \text{UNCHANGED } chkptts \\
&\wedge \text{UNCHANGED } tsvalues \\
&\wedge \text{UNCHANGED } obtains \\
&\wedge \text{UNCHANGED } revokes
\end{aligned}$$

Secure execution mode within checkpoint step 2. Save in nv $appPtr$ the app ptr recorded at ts entry.

$$\begin{aligned}
\text{NextSemChkpt2} &\triangleq \\
&\text{LET} \\
&\quad nvappPcr1 \triangleq \\
&\quad \quad \text{It is a bug for secure execution mode within checkpoint to save in the } nv \text{ ram the current application pcr rather than the seal operation's} \\
&\quad \quad \text{recorded application pcr.} \\
&\quad \text{IF } BugChkptSaveCurApp \text{ THEN } appPcr \text{ ELSE} \\
&\quad \quad chkptts.appPcr \\
&\text{IN} \\
&\wedge pc = PcCHKPT2 \\
&\wedge nv' = [nv \text{ EXCEPT } !.appPcr = nvappPcr1] \\
&\wedge pc' = PcCHKPT3 \\
&\wedge \text{UNCHANGED } appPcr \\
&\wedge \text{UNCHANGED } semPcr \\
&\wedge \text{UNCHANGED } bootCtr \\
&\wedge \text{UNCHANGED } chkptts \\
&\wedge \text{UNCHANGED } tsvalues \\
&\wedge \text{UNCHANGED } obtains \\
&\wedge \text{UNCHANGED } revokes
\end{aligned}$$

Secure execution mode within checkpoint step 3. Prevent a ts replay by incrementing the reboot ctr.

$$\begin{aligned}
\text{NextSemChkpt3} &\triangleq \\
&\text{LET} \\
&\quad bootCtr1 \triangleq \\
&\quad \quad \text{It is a bug for secure execution mode within checkpoint to fail to increment the reboot counter.} \\
&\quad \text{IF } BugChkptNoIncCtr \text{ THEN } bootCtr \text{ ELSE} \\
&\quad \quad bootCtr + 1 \\
&\text{IN} \\
&\wedge pc = PcCHKPT3 \\
&\wedge bootCtr' = bootCtr1
\end{aligned}$$

$\wedge pc' = PcCHKPT4$
 $\wedge UNCHANGED\ nv$
 $\wedge UNCHANGED\ appPcr$
 $\wedge UNCHANGED\ semPcr$
 $\wedge UNCHANGED\ chkptts$
 $\wedge UNCHANGED\ tsvalues$
 $\wedge UNCHANGED\ obtains$
 $\wedge UNCHANGED\ revokes$

Secure execution mode within checkpoint step 4. Declare that the *nv appPcr* is current so that after reboot recovery will be able to succeed.

NextSemChkpt4 \triangleq

LET

nvcurent1 \triangleq

It is a bug for secure execution mode within checkpoint to fail to set the *NV* RAM current flag.

Actually, this bug does not result in a safety violation.

IF *BugChkptNoSetCur* THEN *nv.current* ELSE

TRUE

IN

$\wedge pc = PcCHKPT4$
 $\wedge nv' = [nv\ EXCEPT\ !.current = nvcurent1]$
 $\wedge pc' = PcCHKPT5$
 $\wedge UNCHANGED\ appPcr$
 $\wedge UNCHANGED\ semPcr$
 $\wedge UNCHANGED\ bootCtr$
 $\wedge UNCHANGED\ chkptts$
 $\wedge UNCHANGED\ tsvalues$
 $\wedge UNCHANGED\ obtains$
 $\wedge UNCHANGED\ revokes$

Secure execution mode within checkpoint step 5. Extend sem pcr with unhappy so protected *nv* ram will be inaccessible.

NextSemChkpt5 \triangleq

$\wedge pc = PcCHKPT5$
 $\wedge semPcr' = PcrExtend(semPcr, PcrxUNHAPPY)$
 $\wedge pc' = PcIDLE$
 $\wedge UNCHANGED\ nv$
 $\wedge UNCHANGED\ appPcr$
 $\wedge UNCHANGED\ bootCtr$
 $\wedge UNCHANGED\ chkptts$
 $\wedge UNCHANGED\ tsvalues$
 $\wedge UNCHANGED\ obtains$
 $\wedge UNCHANGED\ revokes$

SPECIFICATION

$$\begin{aligned}
Init &\triangleq \\
&\wedge nv = InitNv \\
&\wedge appPcr = AppReboot \\
&\wedge semPcr = SemReboot \\
&\wedge bootCtr = 0 \\
&\wedge pc = PcIDLE \\
&\wedge chkptts = NullTs \\
&\wedge tsvalues = \{NullTs\} \quad \text{anybody can create a } NullTs \\
&\wedge obtains = \{\} \\
&\wedge revokes = \{\}
\end{aligned}$$

$$\begin{aligned}
Next &\triangleq \\
&\vee NextObtainAccess \\
&\vee NextProveRevoke \\
&\vee NextReboot \\
&\vee NextForgetSealTs \\
&\vee NextExtendAppPcr \\
&\vee NextExtendSemPcr \\
&\vee NextIncBootCtr \\
&\vee NextEnterSemRecov \\
&\vee NextSemRecov1 WhenCorrect \\
&\vee NextSemRecov1 WhenIncorrect \\
&\vee NextSemRecov2 \\
&\vee NextSemRecov3 \\
&\vee NextSealTs \\
&\vee NextEnterSemChkpt \\
&\vee NextSemChkpt1 WhenCorrect \\
&\vee NextSemChkpt1 WhenIncorrect \\
&\vee NextSemChkpt2 \\
&\vee NextSemChkpt3 \\
&\vee NextSemChkpt4 \\
&\vee NextSemChkpt5
\end{aligned}$$

$$Spec \triangleq Init \wedge \square [Next]_{vars}$$

INVARIANTS

Type invariant.

$$\begin{aligned}
 \text{InvType} &\triangleq \\
 &\wedge \text{nv}:: \text{nv} \in \text{Nv} \\
 &\wedge \text{appPcr}:: \text{appPcr} \in \text{Pcr} \\
 &\wedge \text{semPcr}:: \text{semPcr} \in \text{Pcr} \\
 &\wedge \text{bootCtr}:: \text{bootCtr} \in \text{Nat} \\
 &\wedge \text{pc}:: \text{pc} \in \text{Pc} \\
 &\wedge \text{chkptts}:: \wedge \text{chkptts} \in \text{Ts} \\
 &\quad \wedge \text{pc} \in \text{PcChkpt} \setminus \{\text{PcCHKPT1}\} \Rightarrow \text{chkptts} \in \text{SignedTs} \\
 &\wedge \text{tsvalues}:: \text{tsvalues} \in \text{SUBSET Ts} \\
 &\wedge \text{obtains}:: \text{obtains} \in \text{SUBSET Pcr} \\
 &\wedge \text{revokes}:: \text{revokes} \in \text{SUBSET Pcr}
 \end{aligned}$$

Nv protection invariant.

Being in secure execution mode is equivalent to saying that the secure execution mode pcr permits access to protected Nv ram.

$$\begin{aligned}
 \text{InvNvProtection} &\triangleq \\
 \text{InSem} &\equiv (\text{semPcr} = \text{SemProtect})
 \end{aligned}$$

Verifiable revocation invariant. There had better not be any decisions to obtain access for which a proof of revocation was also constructed.

$$\begin{aligned}
 \text{InvVerifiableRevocation} &\triangleq \\
 \forall o \in \text{obtains} : &\quad \text{last extension was OBTAIN} \\
 \forall r \in \text{revokes} : &\quad \text{last extension was REVOKE} \\
 \text{PcrPrior}(o) \neq \text{PcrPrior}(r) &\quad \text{cannot have both extended from same place}
 \end{aligned}$$

Access undeniability.

This invariant is modeled as performing an audit on the present state and seeing that all key bindings that have been used to obtain access appear in the audit report. A key binding o appears in the audit report iff $\text{PcrLeq}(o, \text{appPcr})$, which means that there exists a sequence of zero or more extensions from o that reach appPcr .

However, it might be impossible to generate a valid audit report in the present node state. That is okay.

$$\begin{aligned}
 \text{InvAccessUndeniability} &\triangleq \\
 &\text{It is a bug to fail to require the audit to quote SemHappy.} \\
 &\wedge \quad \text{IF BugAuditNoCheckHappy THEN TRUE ELSE} \\
 &\quad \text{semPcr} = \text{SemHappy} \\
 &\Rightarrow \\
 &\forall o \in \text{obtains} : \text{PcrLeq}(o, \text{appPcr})
 \end{aligned}$$

E Proof of Optimized Pasture

MODULE *OptPastureNodeProof*

EXTENDS *OptPastureNode*, *TLAPS*

STATE FUNCTIONS

We talk about the “log” being in various places. Actually, what is in those places is a cryptographic summary of the log, which is of type *Pcr*. However, under the anticollision assumption of *PcrExtend*, the cryptographic summary is effectively in one-to-one correspondence with the actual log. So we talk as if the cryptographic summary were the log, rather than merely a reference to the log.

Check that *ts* is a valid seal attestation in the current node state. To be valid it must be a signed attestation and it must record *SemHappy* and the current boot counter.

$$\begin{aligned} \text{CheckTsIsCurrent}(ts) &\triangleq \\ &\wedge ts \in \text{SignedTs} \\ &\wedge ts.\text{semPcr} = \text{SemHappy} \\ &\wedge ts.\text{bootCtr} = \text{bootCtr} \end{aligned}$$

All valid seal attestations in the current node state. Seal attestations can be found among the known values (in *tsvalues*) or in the temporary state variable *chkptts* used during the checkpoint sem routine.

$$\begin{aligned} \text{AllCurrentTs} &\triangleq \\ &\{ts \in \text{tsvalues} \cup \{\text{chkptts}\} : \text{CheckTsIsCurrent}(ts)\} \end{aligned}$$

If there are any valid seal attestations in the current node state, choose one and get its log.

$$\begin{aligned} \text{CurrentTsLog} &\triangleq \\ &\text{LET } ts \triangleq \text{CHOOSE } ts \in \text{AllCurrentTs} : \text{TRUE} \\ &\text{IN } ts.\text{appPcr} \end{aligned}$$

The log is present in the *nv* ram. This is true iff the *nv* ram says it is current.

$$\begin{aligned} \text{LogInNv} &\triangleq \\ &\wedge nv.\text{current} \end{aligned}$$

The log is present in the application pcr. This is true iff the sem pcr contains *SemHappy*.

$$\begin{aligned} \text{LogInApp} &\triangleq \\ &\wedge \text{semPcr} = \text{SemHappy} \end{aligned}$$

The log is present in some known seal attestation. This is true iff there exists a valid seal attestation in the current node state.

$$\text{LogInTs} \triangleq$$

$AllCurrentTs \neq \{\}$

Assuming the log exists, determine if *Pcr p* is on it.

The log has a domicile in the *nv* ram, when the *nv* ram is marked as current. The log has a domicile in the application *pcr* when the *sem pcr* contains *SemHappy*. The log has a domicile in a *ts* attestation when that attestation quotes *SemHappy* and the current *bootCtr*.

During secure execution mode, the log can also temporarily live in certain places, as it is moved from one domicile to another.

$IsOnLog(p) \triangleq$

Where to find the log at most times.

- $\wedge \quad LogInNv \quad \Rightarrow PcrLeq(p, nv.appPcr)$
- $\wedge \quad LogInApp \quad \Rightarrow PcrLeq(p, appPcr)$
- $\wedge \quad LogInTs \quad \Rightarrow PcrLeq(p, CurrentTsLog)$

Special places to find the log during secure execution mode.

- $\wedge \quad pc = PcRECOV1 \quad \Rightarrow TRUE$
- $\wedge \quad pc = PcRECOV2 \quad \Rightarrow PcrLeq(p, appPcr)$
- $\wedge \quad pc = PcRECOV3 \quad \Rightarrow PcrLeq(p, appPcr)$
- $\wedge \quad pc = PcCHKPT1 \quad \Rightarrow TRUE$
- $\wedge \quad pc = PcCHKPT2 \quad \Rightarrow PcrLeq(p, chkpts.appPcr)$
- $\wedge \quad pc = PcCHKPT3 \quad \Rightarrow PcrLeq(p, nv.appPcr)$
- $\wedge \quad pc = PcCHKPT4 \quad \Rightarrow PcrLeq(p, nv.appPcr)$
- $\wedge \quad pc = PcCHKPT5 \quad \Rightarrow PcrLeq(p, nv.appPcr)$

ADDITIONAL INVARIANTS

When the node is in secure execution mode, the secure execution mode *pcr* contains *SemProtect*.

$InvInSemProtect \triangleq$

- $\wedge \quad InvType$
- $\wedge \quad goal::$
- $InSem \Rightarrow semPcr = SemProtect$

When the node is not in secure execution mode, the secure execution mode *pcr* contains a value from which *SemProtect* cannot be reached.

$InvUnreachableSemProtect \triangleq$

- $\wedge \quad InvType$
- $\wedge \quad InvInSemProtect$
- $\wedge \quad goal::$
- $\neg InSem \Rightarrow \neg PcrLeq(semPcr, SemProtect)$

All known signed seal attestations quote a *bootCtr* that does not exceed the current *bootCtr*.

$$\begin{aligned}
\text{InvSignedTsLeqBoot} &\triangleq \\
&\wedge \text{InvType} \\
&\wedge \text{goal}:: \\
&\forall ts \in \text{tsvalues} \cup \{\text{chkptts}\} : \\
&ts \in \text{SignedTs} \Rightarrow ts.\text{bootCtr} \leq \text{bootCtr}
\end{aligned}$$

When the node is not in secure execution mode, the secure execution mode pcr contains either (1) *SemHappy* or (2) a value from which *SemHappy* cannot be reached.

$$\begin{aligned}
\text{InvUnforgeableSemHappy} &\triangleq \\
&\wedge \text{InvType} \\
&\wedge \text{InvInSemProtect} \\
&\wedge \text{goal}:: \\
&\neg \text{InSem} \Rightarrow \\
&\quad \vee \text{semPcr} = \text{SemHappy} \\
&\quad \vee \neg \text{PcrLeq}(\text{semPcr}, \text{SemHappy})
\end{aligned}$$

Every entry in obtains and revokes has a last extension of OBTAIN and REVOKE, respectively.

$$\begin{aligned}
\text{InvProperLastExtension} &\triangleq \\
&\wedge \text{InvType} \\
&\wedge \text{goal}:: \\
&\wedge \forall o \in \text{obtains} : \text{PcrHasExtension}(o) \wedge \text{PcrLastExtension}(o) = \text{PcrxOBTAIN} \\
&\wedge \forall r \in \text{revokes} : \text{PcrHasExtension}(r) \wedge \text{PcrLastExtension}(r) = \text{PcrxREVOKE}
\end{aligned}$$

There is at most one log.

One Log to rule them all,
 One Log to find them,
 One Log to bring them all
 and in the darkness bind them.
 (with apologies to *J. R. R. Tolkien*)

$$\begin{aligned}
\text{InvOneLog} &\triangleq \\
&\wedge \text{InvType} \\
&\wedge \text{InvSignedTsLeqBoot} \\
&\wedge \text{InvInSemProtect} \\
&\wedge \text{InvUnforgeableSemHappy} \\
&\wedge \text{InvProperLastExtension} \\
&\wedge \text{goal}::
\end{aligned}$$

The log can only have at most one domicile at a time.

- $\wedge \text{LogInNv} \Rightarrow \neg \text{LogInApp} \wedge \neg \text{LogInTs}$
- $\wedge \text{LogInApp} \Rightarrow \neg \text{LogInNv} \wedge \neg \text{LogInTs}$
- $\wedge \text{LogInTs} \Rightarrow \neg \text{LogInNv} \wedge \neg \text{LogInApp}$

Extra requirements during secure execution mode.

- $\wedge pc = \text{PcRECOV1} \Rightarrow \text{TRUE}$
- $\wedge pc = \text{PcRECOV2} \Rightarrow \text{LogInNv}$
- $\wedge pc = \text{PcRECOV3} \Rightarrow \neg \text{LogInNv} \wedge \neg \text{LogInApp} \wedge \neg \text{LogInTs}$
- $\wedge pc = \text{PcCHKPT1} \Rightarrow \text{TRUE}$
- $\wedge pc = \text{PcCHKPT2} \Rightarrow \text{LogInTs} \wedge \text{CheckTsIsCurrent}(\text{chkptts})$
- $\wedge pc = \text{PcCHKPT3} \Rightarrow \text{LogInTs} \wedge \text{CheckTsIsCurrent}(\text{chkptts})$
- $\wedge pc = \text{PcCHKPT4} \Rightarrow \neg \text{LogInNv} \wedge \neg \text{LogInApp} \wedge \neg \text{LogInTs}$
- $\wedge pc = \text{PcCHKPT5} \Rightarrow \text{LogInNv}$

All seal attestations containing the log must have the same log.

- $\wedge \forall ts1, ts2 \in \text{AllCurrentTs} : ts1.\text{appPcr} = ts2.\text{appPcr}$

Every entry in obtains (a decision to obtain access) is recorded on the log (assuming there is one).

- $\wedge \text{obtains} :: \forall o \in \text{obtains} : \text{IsOnLog}(o)$

Every entry in revokes (a decision to prove revocation) is recorded on the log (assuming there is one).

- $\wedge \text{revokes} :: \forall r \in \text{revokes} : \text{IsOnLog}(r)$

We have verifiable revocation.

- $\wedge \text{InvVerifiableRevocation}$

NECESSARY FACTS ABOUT NATURALS

The *SMT* prover can prove these easily enough in isolation, but if you ask it to prove them in the middle of other proofs where records and other complicated things are flying around, it usually aborts with a type inference failure.

\leq is a total order

- THEOREM $\text{ThmNatLeqIsTotal} \triangleq \forall i, j \in \text{Nat} : i \leq j \vee j \leq i$ BY *SMT*
- THEOREM $\text{ThmNatLeqIsReflexive} \triangleq \forall i \in \text{Nat} : i \leq i$ BY *SMT*
- THEOREM $\text{ThmNatLeqIsAntisymmetric} \triangleq \forall i, j \in \text{Nat} : i \leq j \wedge j \leq i \Rightarrow i = j$ BY *SMT*
- THEOREM $\text{ThmNatLeqIsTransitive} \triangleq \forall i, j, k \in \text{Nat} : i \leq j \wedge j \leq k \Rightarrow i \leq k$ BY *SMT*

\leq minimum is 0

- THEOREM $\text{ThmNatLeqMinIsZero} \triangleq \forall i \in \text{Nat} : 0 \leq i$ BY *SMT*

\leq is the opposite of $>$

- THEOREM $\text{ThmNatLeqXorGt} \triangleq \forall i, j \in \text{Nat} : i \leq j \equiv \neg(i > j)$ BY *SMT*

- THEOREM $\text{ThmNatMore} \triangleq \forall i, j \in \text{Nat} : i \leq i + j$ BY *SMT*

THEOREM *ThmNatLess* $\triangleq \forall i, j \in \text{Nat} : i - j \leq i$ BY SMT

THEOREM *ThmNatInc* $\triangleq \forall i \in \text{Nat} : i + 1 > i$ BY SMT

THEOREM *ThmNatDotDot* $\triangleq \forall i, j, k \in \text{Nat} : i \leq j \wedge j \leq k \equiv j \in i .. k$ BY SMT

THEOREM *ThmNatDecZero* $\triangleq \forall n \in \text{Nat} : n > 0 \Rightarrow n - 1 \in \text{Nat}$ BY SMT

THEOREM *ThmNatAddEq* $\triangleq \forall i, j, k \in \text{Nat} : i + k = j + k \Rightarrow i = j$ BY SMT

THEOREM *ThmNatLeqLt* $\triangleq \forall i, j, k \in \text{Nat} : i \leq j \wedge j < k \Rightarrow i < k$ BY SMT

NECESSARY FACTS ABOUT SEQUENCES

I have not been able to figure out how to convince the prover to prove most of these.

Definition of a sequence.

THEOREM *ThmSeqDef* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$

PROVE

$q = [i \in 1 .. \text{Len}(q) \mapsto q[i]]$

PROOF

OMITTED

The empty sequence is a sequence of S , for any S .

THEOREM *ThmSeqEmptyIsSeq* \triangleq

ASSUME

NEW CONSTANT S

PROVE

$\langle \rangle \in \text{Seq}(S)$

PROOF

OMITTED

For any sequence q of S , $\text{Len}(q) \in \text{Nat}$.

THEOREM *ThmSeqLenIsNat* \triangleq

ASSUME

NEW CONSTANT S ,

NEW $q \in \text{Seq}(S)$

PROVE

$$\text{Len}(q) \in \text{Nat}$$

PROOF

OMITTED

For any non-empty sequence of S , its tail is a sequence of S .

THEOREM *ThmSeqTailIsSeq* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$,

$q \neq \langle \rangle$

PROVE

$$\text{Tail}(q) \in \text{Seq}(S)$$

PROOF

OMITTED

For any sequence of S , appending $x \in S$ yields a sequence of S .

THEOREM *ThmSeqAppendIsSeq* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$,

NEW CONSTANT $x \in S$

PROVE

$$\text{Append}(q, x) \in \text{Seq}(S)$$

PROOF

OMITTED

The result of $\text{Append}(q, x)$ is one longer than q .

THEOREM *ThmSeqAppendLen1* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT $q \in \text{Seq}(S)$,

NEW CONSTANT $x \in S$

PROVE

$$\text{Len}(\text{Append}(q, x)) = \text{Len}(q) + 1$$

PROOF

OMITTED

The result of $\text{Append}(q, x)$ starts with q .

THEOREM *ThmSeqAppendSubSeq* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in Seq(S)$,
 NEW CONSTANT $x \in S$

PROVE

$$SubSeq(Append(q, x), 1, Len(q)) = q$$

PROOF

OMITTED

Appending the last entry onto all but the last of a sequence yields the original sequence.

THEOREM *ThmSeqAppendPriorLast* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in Seq(S)$,
 $Len(q) > 0$

PROVE

$$Append(SubSeq(q, 1, Len(q) - 1), q[Len(q)]) = q$$

PROOF

OMITTED

The entire initial *SubSeq* of q is q .

THEOREM *ThmSeqEntireInitialSubSeq* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in Seq(S)$

PROVE

$$q = SubSeq(q, 1, Len(q))$$

PROOF

OMITTED

Initial *SubSeq* \in sequence.

THEOREM *ThmSeqInitialSubSeqIsSeq* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in Seq(S)$,
 NEW CONSTANT $n \in Nat$,
 $n \leq Len(q)$

PROVE

$$SubSeq(q, 1, n) \in Seq(S)$$

PROOF

OMITTED

Initial *SubSeq* is antisymmetric.

THEOREM *ThmSeqInitialSubSeqIsAntisymmetric* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in \text{Seq}(S)$,
 NEW CONSTANT $r \in \text{Seq}(S)$,
 $\text{Len}(q) \leq \text{Len}(r)$,
 $\text{Len}(r) \leq \text{Len}(q)$,
 $q = \text{SubSeq}(r, 1, \text{Len}(q))$,
 $r = \text{SubSeq}(q, 1, \text{Len}(r))$

PROVE

$q = r$

PROOF

(1) $\text{Len}(q) = \text{Len}(r)$
 (2) USE *ThmSeqLenIsNat*
 (2) QED BY *ThmNatLeqIsAntisymmetric*
 (1) QED BY *ThmSeqEntireInitialSubSeq*

Initial *SubSeq* is transitive.

THEOREM *ThmSeqInitialSubSeqIsTransitive* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in \text{Seq}(S)$,
 NEW CONSTANT $r \in \text{Seq}(S)$,
 NEW CONSTANT $s \in \text{Seq}(S)$,
 $\text{Len}(q) \leq \text{Len}(r)$,
 $\text{Len}(r) \leq \text{Len}(s)$,
 $q = \text{SubSeq}(r, 1, \text{Len}(q))$,
 $r = \text{SubSeq}(s, 1, \text{Len}(r))$

PROVE

$q = \text{SubSeq}(s, 1, \text{Len}(q))$

PROOF

OMITTED

Sequence append incompatible.

THEOREM *ThmSeqAppendIncompatible* \triangleq

ASSUME

NEW CONSTANT S ,
 NEW CONSTANT $q \in \text{Seq}(S)$,
 NEW CONSTANT $s1 \in S$,
 NEW CONSTANT $s2 \in S$,
 $s1 \neq s2$

PROVE

$Append(q, s1) \neq Append(q, s2)$

PROOF

OMITTED

Sequence append anti-collision.

THEOREM *ThmSeqAppendAnticollision* \triangleq

ASSUME

NEW CONSTANT S ,

NEW CONSTANT q ,

NEW CONSTANT $q1 \in Seq(S)$,

NEW CONSTANT $q2 \in Seq(S)$,

NEW CONSTANT $s1 \in S$,

NEW CONSTANT $s2 \in S$,

$q = Append(q1, s1)$,

$q = Append(q2, s2)$

PROVE

$q1 = q2 \wedge s1 = s2$

PROOF

OMITTED

PCR OPERATOR THEOREMS

PcrInit \in *Pcr*

THEOREM *ThmPcrInitIsPcr* \triangleq

$\forall i \in Pcri : PcrInit(i) \in Pcr$

PROOF

$\langle 1 \rangle$ TAKE $i \in Pcri$

$\langle 1 \rangle$ USE DEF *Pcr*, *PcrInit*

$\langle 1 \rangle$ DEFINE $p \triangleq PcrInit(i)$

$\langle 1 \rangle 1. p.init \in Pcri$ OBVIOUS

$\langle 1 \rangle 2. p.extq \in Seq(Pcrx)$ BY *ThmSeqEmptyIsSeq*

$\langle 1 \rangle$ QED BY $\langle 1 \rangle 1$, $\langle 1 \rangle 2$

PcrExtend \in *Pcr*

THEOREM $ThmPcrExtendIsPcr \triangleq$

$\forall p \in Pcr, x \in Pcrx : PcrExtend(p, x) \in Pcr$

PROOF

- (1) TAKE $p \in Pcr, x \in Pcrx$
- (1) USE DEF $Pcr, PcrExtend$
- (1) DEFINE $px \triangleq PcrExtend(p, x)$
- (1)1. $px.init \in Pcri$ OBVIOUS
- (1)2. $px.extq \in Seq(Pcrx)$ BY $ThmSeqAppendIsSeq$
- (1) QED BY (1)1, (1)2

$PcrLen \in Nat$

THEOREM $ThmPcrLenIsNat \triangleq$

$\forall p \in Pcr : PcrLen(p) \in Nat$

PROOF

- (1) TAKE $p \in Pcr$
- (1) USE DEF $PcrLen$
- (1) USE DEF Pcr
- (1) QED BY $ThmSeqLenIsNat$

$p \leq PcrExtend(p, x)$

THEOREM $ThmPcrExtendLeq \triangleq$

$\forall p \in Pcr, x \in Pcrx : PcrLeq(p, PcrExtend(p, x))$

PROOF

- (1) TAKE $p \in Pcr, x \in Pcrx$
- (1) DEFINE $px \triangleq PcrExtend(p, x)$
- (1) USE DEF Pcr
- (1) USE DEF $PcrExtend$
- (1) USE DEF $PcrLeq$
- (1)1. $p.init = px.init$ OBVIOUS
- (1)2. $Len(p.extq) \leq Len(px.extq)$
 - (2) $Len(p.extq) + 1 = Len(px.extq)$ BY $ThmSeqAppendLen1$
 - (2) USE $ThmSeqLenIsNat$
 - (2) QED BY $ThmNatMore$
- (1)3. $p.extq = SubSeq(px.extq, 1, Len(p.extq))$ BY $ThmSeqAppendSubSeq$
- (1) QED BY (1)1, (1)2, (1)3

$p \neq PcrExtend(p, x)$

THEOREM $ThmPcrExtendNeq \triangleq$

$\forall p \in Pcr, x \in Pcrx : p \neq PcrExtend(p, x)$

PROOF

- (1) TAKE $p \in Pcr, x \in Pcrx$
- (1) DEFINE $px \triangleq PcrExtend(p, x)$
- (1) USE DEF Pcr

```

<1> USE DEF PcrExtend
<1>  $p.extq \neq px.extq$ 
  <2>  $p.extq \in Seq(Pcrx)$  OBVIOUS
  <2>  $px.extq \in Seq(Pcrx)$  BY ThmSeqAppendIsSeq
  <2> DEFINE  $pn \triangleq Len(p.extq)$ 
  <2> DEFINE  $pxn \triangleq Len(px.extq)$ 
  <2>  $pn \neq pxn$ 
    <3>  $pn \in Nat$  BY ThmSeqLenIsNat
    <3>  $pxn \in Nat$  BY ThmSeqLenIsNat
    <3>  $pxn = pn + 1$  BY ThmSeqAppendLen1
    <3>  $pxn > pn$  BY ThmNatInc
    <3>  $\neg(pxn \leq pn)$  BY ThmNatLeqXorGt
    <3> QED BY ThmNatLeqIsReflexive
  <2> QED OBVIOUS
<1> QED OBVIOUS

```

Pcr equality. This would seem to be trivial but the prover cannot seem to figure it out by itself.

THEOREM *ThmPcrEqual* \triangleq

```

 $\forall p, q \in Pcr :$ 
   $\wedge p.init = q.init$ 
   $\wedge p.extq = q.extq$ 
   $\Rightarrow p = q$ 

```

PROOF

```

<1> TAKE  $p, q \in Pcr$ 
<1> HAVE  $p.init = q.init \wedge p.extq = q.extq$ 
<1> USE DEF Pcr

```

The following fact seems to be necessary to help the prover.

```

<1>  $p = [q \text{ EXCEPT } !.init = p.init, !.extq = p.extq]$  OBVIOUS
<1> QED OBVIOUS

```

Anti-collision property.

THEOREM *ThmPcrExtendAnticollision* \triangleq

```

 $\forall p1, p2 \in Pcr, x1, x2 \in Pcrx :$ 
   $PcrExtend(p1, x1) = PcrExtend(p2, x2) \Rightarrow p1 = p2 \wedge x1 = x2$ 

```

PROOF

```

<1> TAKE  $p1, p2 \in Pcr, x1, x2 \in Pcrx$ 
<1> DEFINE  $px1 \triangleq PcrExtend(p1, x1)$ 
<1> DEFINE  $px2 \triangleq PcrExtend(p2, x2)$ 
<1> HAVE  $px1 = px2$ 
<1> USE DEF Pcr
<1> USE DEF PcrExtend
<1> QED
  <2>  $p1.init = p2.init$  OBVIOUS

```

⟨2⟩ $p1.extq = p2.extq \wedge x1 = x2$

Create definitions for the *extq* fields and then hide them to prevent overwhelming the prover.

⟨3⟩ DEFINE $p1q \triangleq p1.extq$
 ⟨3⟩ DEFINE $p2q \triangleq p2.extq$
 ⟨3⟩ HIDE DEF $p1q$
 ⟨3⟩ HIDE DEF $p2q$
 ⟨3⟩ $Append(p1q, x1) = Append(p2q, x2)$ BY DEF $p1q, p2q$
 ⟨3⟩ $p1q \in Seq(Pcrx)$ BY DEF $p1q$
 ⟨3⟩ $p2q \in Seq(Pcrx)$ BY DEF $p2q$
 ⟨3⟩ $p1q = p2q \wedge x1 = x2$ BY *ThmSeqAppendAnticollision*
 ⟨3⟩ QED BY DEF $p1q, p2q$
 ⟨2⟩ QED BY *ThmPcrEqual*

If two extensions of the same pcr are both \leq a target pcr, then the extensions must be the same.

THEOREM *ThmPcrExtendLeqAnticollision* \triangleq

$\forall p, t \in Pcr, x1, x2 \in Pcrx :$

$PcrLeq(PcrExtend(p, x1), t) \wedge PcrLeq(PcrExtend(p, x2), t) \Rightarrow x1 = x2$

PROOF

⟨1⟩ TAKE $p, t \in Pcr, x1, x2 \in Pcrx$
 ⟨1⟩ HAVE $PcrLeq(PcrExtend(p, x1), t) \wedge PcrLeq(PcrExtend(p, x2), t)$
 ⟨1⟩ USE DEF *Pcr*
 ⟨1⟩ USE DEF *PcrExtend*
 ⟨1⟩ USE DEF *PcrLeq*
 ⟨1⟩ DEFINE $qp \triangleq p.extq$
 ⟨1⟩ DEFINE $qt \triangleq t.extq$
 ⟨1⟩ DEFINE $qpx1 \triangleq Append(qp, x1)$
 ⟨1⟩ DEFINE $qpx2 \triangleq Append(qp, x2)$
 ⟨1⟩ $qp \in Seq(Pcrx)$ OBVIOUS
 ⟨1⟩ $qt \in Seq(Pcrx)$ OBVIOUS
 ⟨1⟩ $Len(qpx1) \leq Len(qt)$ OBVIOUS
 ⟨1⟩ $Len(qpx2) \leq Len(qt)$ OBVIOUS
 ⟨1⟩ $SubSeq(qt, 1, Len(qpx1)) = qpx1$ OBVIOUS
 ⟨1⟩ $SubSeq(qt, 1, Len(qpx2)) = qpx2$ OBVIOUS
 ⟨1⟩ HIDE DEF qp
 ⟨1⟩ HIDE DEF qt
 ⟨1⟩ $Len(qpx1) = Len(qp) + 1$ BY *ThmSeqAppendLen1*
 ⟨1⟩ $Len(qpx2) = Len(qp) + 1$ BY *ThmSeqAppendLen1*
 ⟨1⟩ $Len(qpx1) = Len(qpx2)$ OBVIOUS
 ⟨1⟩ $qpx1 = qpx2$ OBVIOUS
 ⟨1⟩ QED

The prover really needs help to focus its attention.

⟨2⟩1. $x1 \in Pcrx$ OBVIOUS
 ⟨2⟩2. $x2 \in Pcrx$ OBVIOUS
 ⟨2⟩3. $qp \in Seq(Pcrx)$ OBVIOUS

⟨2⟩4. $Append(qp, x1) = Append(qp, x2)$ OBVIOUS
 ⟨2⟩ QED BY ONLY ⟨2⟩1, ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, $ThmSeqAppendAnticollision$

PcrExtend increases the length by 1.

THEOREM $ThmPcrExtendLen1 \triangleq$

$\forall p \in Pcr, x \in Pcrx : PcrLen(PcrExtend(p, x)) = PcrLen(p) + 1$

PROOF

⟨1⟩ TAKE $p \in Pcr, x \in Pcrx$
 ⟨1⟩ DEFINE $px \triangleq PcrExtend(p, x)$
 ⟨1⟩ $px \in Pcr$ BY $ThmPcrExtendIsPcr$
 ⟨1⟩ USE DEF $PcrLen$
 ⟨1⟩ USE DEF $PcrExtend$
 ⟨1⟩ USE DEF Pcr
 ⟨1⟩ QED BY $ThmSeqAppendLen1$

PcrLeq implies \leq on respective *PcrLen*.

THEOREM $ThmPcrLeqLeq \triangleq$

$\forall p, q \in Pcr : PcrLeq(p, q) \Rightarrow PcrLen(p) \leq PcrLen(q)$

PROOF

⟨1⟩ TAKE $p, q \in Pcr$
 ⟨1⟩ HAVE $PcrLeq(p, q)$
 ⟨1⟩ USE DEF Pcr
 ⟨1⟩ $Len(p.extq) \leq Len(q.extq)$ BY DEF $PcrLeq$
 ⟨1⟩ $PcrLen(p) = Len(p.extq)$ BY DEF $PcrLen$
 ⟨1⟩ $PcrLen(q) = Len(q.extq)$ BY DEF $PcrLen$
 ⟨1⟩ QED OBVIOUS

PcrLeq is a partial order.

THEOREM $ThmPcrLeqIsReflexive \triangleq$

$\forall p \in Pcr : PcrLeq(p, p)$

PROOF

⟨1⟩ TAKE $p \in Pcr$
 ⟨1⟩ USE DEF $PcrLeq$
 ⟨1⟩ USE DEF Pcr
 ⟨1⟩ 1. $p.init = p.init$ OBVIOUS
 ⟨1⟩ 2. $Len(p.extq) \leq Len(p.extq)$
 ⟨2⟩ USE $ThmSeqLenIsNat$
 ⟨2⟩ USE $ThmNatLeqIsReflexive$
 ⟨2⟩ QED OBVIOUS

- (1)3. $p.\text{extq} = \text{SubSeq}(p.\text{extq}, 1, \text{Len}(p.\text{extq}))$
- (2) USE *ThmSeqEntireInitialSubSeq*
- (2) QED OBVIOUS
- (1) QED BY (1)1, (1)2, (1)3

THEOREM *ThmPcrLeqIsAntisymmetric* \triangleq
 $\forall p, q \in \text{Pcr} : \text{PcrLeq}(p, q) \wedge \text{PcrLeq}(q, p) \Rightarrow p = q$

PROOF

- (1) TAKE $p, q \in \text{Pcr}$
- (1)2. HAVE $\text{PcrLeq}(p, q) \wedge \text{PcrLeq}(q, p)$
- (1)3. $p.\text{init} = q.\text{init}$
- (2) USE DEF *PcrLeq*
- (2) QED BY (1)2
- (1)4. $p.\text{extq} = q.\text{extq}$
- (2) USE DEF *PcrLeq*
- (2) USE DEF *Pcr*
- (2) USE *ThmSeqInitialSubSeqIsAntisymmetric*
- (2) QED BY (1)2
- (1) QED
- (2) USE *ThmPcrEqual*
- (2) QED BY (1)3, (1)4

THEOREM *ThmPcrLeqIsTransitive* \triangleq
 $\forall p, q, r \in \text{Pcr} : \text{PcrLeq}(p, q) \wedge \text{PcrLeq}(q, r) \Rightarrow \text{PcrLeq}(p, r)$

PROOF

- (1) TAKE $p, q, r \in \text{Pcr}$
- (1) HAVE $\text{PcrLeq}(p, q) \wedge \text{PcrLeq}(q, r)$
- (1) USE DEF *PcrLeq*
- (1)1. $p.\text{init} = r.\text{init}$ OBVIOUS
- (1)2. $\text{Len}(p.\text{extq}) \leq \text{Len}(r.\text{extq})$
- (2) USE DEF *Pcr*
- (2) USE *ThmSeqLenIsNat*
- (2) QED BY *ThmNatLeqIsTransitive*
- (1)3. $p.\text{extq} = \text{SubSeq}(r.\text{extq}, 1, \text{Len}(p.\text{extq}))$
- (2) USE DEF *Pcr*
- (2) QED BY *ThmSeqInitialSubSeqIsTransitive*
- (1) QED BY (1)1, (1)2, (1)3

An extension of a *Pcr* p cannot reach p .

THEOREM *ThmPcrExtendSelfUnreachable* \triangleq
 $\forall p \in \text{Pcr}, x \in \text{Pcr}x : \neg \text{PcrLeq}(\text{PcrExtend}(p, x), p)$

PROOF

- (1) TAKE $p \in \text{Pcr}, x \in \text{Pcr}x$
- (1) DEFINE $px \triangleq \text{PcrExtend}(p, x)$

⟨1⟩ DEFINE $isleq \triangleq PcrLeq(px, p)$
 ⟨1⟩ $px \in Pcr$ BY $ThmPcrExtendIsPcr$

Proof by contradiction.

⟨1⟩1. CASE $\neg isleq$ BY ⟨1⟩1
 ⟨1⟩2. CASE $isleq$
 ⟨2⟩1. $PcrLen(px) \leq PcrLen(p)$
 ⟨3⟩ USE ⟨1⟩2
 ⟨3⟩ USE $ThmPcrLeqLeq$
 ⟨3⟩ QED OBVIOUS
 ⟨2⟩2. $PcrLen(px) > PcrLen(p)$
 ⟨3⟩ $PcrLen(px) = PcrLen(p) + 1$ BY $ThmPcrExtendLen1$
 ⟨3⟩ USE $ThmPcrLenIsNat$
 ⟨3⟩ USE $ThmNatInc$
 ⟨3⟩ QED OBVIOUS
 ⟨2⟩ USE $ThmPcrLenIsNat$
 ⟨2⟩ USE $ThmNatLeqXorGt$
 ⟨2⟩ QED BY ⟨2⟩1, ⟨2⟩2
 ⟨1⟩ QED BY ⟨1⟩2, ⟨1⟩1

If an extension of a Pcr can reach a target, the Pcr itself can reach the target.

THEOREM $ThmPcrReachableIfExtend \triangleq$
 $\forall p, q \in Pcr, x \in Pcrx :$
 $PcrLeq(PcrExtend(p, x), q) \Rightarrow PcrLeq(p, q)$

PROOF

⟨1⟩ TAKE $p, q \in Pcr, x \in Pcrx$
 ⟨1⟩ DEFINE $px \triangleq PcrExtend(p, x)$
 ⟨1⟩ HAVE $PcrLeq(px, q)$
 ⟨1⟩ $px \in Pcr$ BY $ThmPcrExtendIsPcr$
 ⟨1⟩ $PcrLeq(p, px)$ BY $ThmPcrExtendLeq$
 ⟨1⟩ QED BY $ThmPcrLeqIsTransitive$

If a target Pcr is not reachable from a source Pcr , then it is not reachable from an extension of the source Pcr .

THEOREM $ThmPcrExtendSourceUnreachable \triangleq$
 $\forall p, q \in Pcr, x \in Pcrx :$
 $\neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q)$

PROOF

⟨1⟩ QED BY $ThmPcrReachableIfExtend$

If p equals q or cannot reach q , then an extension of p cannot reach q .

THEOREM *ThmPcrExtendFromEqOrNotleq* \triangleq

$\forall p, q \in Pcr, x \in Pcrx :$

$p = q \vee \neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q)$

PROOF

(1) TAKE $p, q \in Pcr, x \in Pcrx$

(1) HAVE $p = q \vee \neg PcrLeq(p, q)$

(1) CASE $p = q$ BY *ThmPcrExtendSelfUnreachable*

(1) CASE $\neg PcrLeq(p, q)$ BY *ThmPcrReachableIfExtend*

(1) QED OBVIOUS

Different extensions of a pcr are incompatible.

THEOREM *ThmPcrExtendIncompatible* \triangleq

$\forall p \in Pcr, x1, x2 \in Pcrx :$

$x1 \neq x2 \Rightarrow \neg PcrLeq(PcrExtend(p, x1), PcrExtend(p, x2))$

PROOF

(1) TAKE $p \in Pcr, x1, x2 \in Pcrx$

(1) HAVE $x1 \neq x2$

(1) DEFINE $p1 \triangleq PcrExtend(p, x1)$

(1) DEFINE $p2 \triangleq PcrExtend(p, x2)$

(1)1. CASE $\neg PcrLeq(p1, p2)$ BY (1)1

(1)2. CASE $PcrLeq(p1, p2)$

(2) USE (1)2

(2) USE DEF *PcrLeq*

(2) USE DEF *PcrExtend*

(2) USE DEF *Pcr*

(2) $p1.extq \in Seq(Pcrx)$ BY *ThmSeqAppendIsSeq*

(2) $p2.extq \in Seq(Pcrx)$ BY *ThmSeqAppendIsSeq*

(2) $Len(p1.extq) = Len(p2.extq)$

(3) $Len(p1.extq) = Len(p.extq) + 1$ BY *ThmSeqAppendLen1*

(3) $Len(p2.extq) = Len(p.extq) + 1$ BY *ThmSeqAppendLen1*

(3) QED OBVIOUS

(2) $p1.extq = p2.extq$ BY *ThmSeqEntireInitialSubSeq*

(2) $p1.extq \neq p2.extq$ BY *ThmSeqAppendIncompatible*

(2) QED OBVIOUS

(1) QED BY (1)1, (1)2

If a *Pcr* has an extension, applying *PriorPcr* to it yields a *Pcr*.

THEOREM *ThmPcrPriorIsPcr* \triangleq

$\forall p \in Pcr : PcrHasExtension(p) \Rightarrow PcrPrior(p) \in Pcr$

PROOF

(1) TAKE $p \in Pcr$

(1) HAVE *PcrHasExtension*(p)

⟨1⟩ USE DEF *PcrHasExtension*
 ⟨1⟩ USE DEF *PcrPrior*
 ⟨1⟩ USE DEF *PcrLen*
 ⟨1⟩ USE DEF *Pcr*
 ⟨1⟩ *PcrPrior*(*p*).*extq* ∈ *Seq*(*Pcrx*)
 ⟨2⟩ *Len*(*p.extq*) ∈ *Nat* BY *ThmSeqLenIsNat*
 ⟨2⟩ *Len*(*p.extq*) − 1 ∈ *Nat* BY *ThmNatDecZero*
 ⟨2⟩ *Len*(*p.extq*) − 1 ≤ *Len*(*p.extq*) BY *ThmNatLess*
 ⟨2⟩ QED BY *ThmSeqInitialSubSeqIsSeq*
 ⟨1⟩ QED OBVIOUS

Putting the last extension back on the prior pcr yields the original pcr.

THEOREM *ThmPcrExtendPriorLast* \triangleq

$\forall p \in Pcr :$
PcrHasExtension(*p*) \Rightarrow
PcrExtend(*PcrPrior*(*p*), *PcrLastExtension*(*p*)) = *p*

PROOF

⟨1⟩ TAKE *p* ∈ *Pcr*
 ⟨1⟩ HAVE *PcrHasExtension*(*p*)
 ⟨1⟩ USE DEF *PcrHasExtension*
 ⟨1⟩ USE DEF *PcrPrior*
 ⟨1⟩ USE DEF *PcrLastExtension*
 ⟨1⟩ USE DEF *PcrExtend*
 ⟨1⟩ USE DEF *PcrLen*
 ⟨1⟩ USE DEF *Pcr*
 ⟨1⟩ DEFINE *p0* \triangleq *PcrPrior*(*p*)
 ⟨1⟩ DEFINE *x* \triangleq *PcrLastExtension*(*p*)
 ⟨1⟩ *PcrExtend*(*p0*, *x*).*init* = *p.init* OBVIOUS
 ⟨1⟩ *PcrExtend*(*p0*, *x*).*extq* = *p.extq*
 ⟨2⟩ DEFINE *qp* \triangleq *p.extq*
 ⟨2⟩ SUFFICES
 ASSUME
 Len(*qp*) > 0,
 qp ∈ *Seq*(*Pcrx*)
 PROVE
 Append(*SubSeq*(*qp*, 1, *Len*(*qp*) − 1), *qp*[*Len*(*qp*)]) = *qp*
 OBVIOUS
 ⟨2⟩ HIDE DEF *qp*
 ⟨2⟩ QED BY *ThmSeqAppendPriorLast*
 ⟨1⟩ QED BY *ThmPcrEqual*

WELL KNOWN *PCR* VALUES

Value of the application pcr attained by rebooting.

THEOREM $ThmAppRebootIsPcr \triangleq AppReboot \in Pcr$

PROOF

(1) USE DEF *AppReboot*(1) USE DEF *Pcri*(1) QED BY *ThmPcrInitIsPcr*

Value of the secure execution mode pcr attained by rebooting.

THEOREM $ThmSemRebootIsPcr \triangleq SemReboot \in Pcr$

PROOF

(1) USE DEF *SemReboot*(1) USE DEF *Pcri*(1) QED BY *ThmPcrInitIsPcr*

Value of the secure execution mode pcr attained by entering the protected module in secure execution mode.

THEOREM $ThmSemProtectIsPcr \triangleq SemProtect \in Pcr$

PROOF

(1) USE DEF *SemProtect*(1) USE DEF *Pcri*(1) QED BY *ThmPcrInitIsPcr*

Value of the secure execution mode pcr that indicates that Pasture is happy. Recovery has been properly performed and bound keys may be used. Checkpoint has not yet been invoked.

THEOREM $ThmSemHappyIsPcr \triangleq SemHappy \in Pcr$

PROOF

(1) USE DEF *SemHappy*(1) USE *ThmSemProtectIsPcr*(1) USE DEF *Pcrx*(1) QED BY *ThmPcrExtendIsPcr*From *SemReboot* cannot reach *SemProtect*.THEOREM $ThmSemRebootNotleqSemProtect \triangleq \neg PcrLeq(SemReboot, SemProtect)$

PROOF

(1) USE DEF *PcrInit*(1) USE DEF *SemReboot*

⟨1⟩ USE DEF *SemProtect*
 ⟨1⟩ USE *AssSemProtect*
 ⟨1⟩ QED BY DEF *PcrLeq*

 PROTECTED NV RAM STATE

THEOREM $ThmInitNvIsNv \triangleq InitNv \in Nv$

PROOF

⟨1⟩ USE DEF *InitNv*
 ⟨1⟩ USE DEF *Nv*
 ⟨1⟩ USE *ThmAppRebootIsPcr*
 ⟨1⟩ QED OBVIOUS

 SEAL OPERATION TRANSPORT SESSION STATE

THEOREM $ThmNullTsIsTs \triangleq NullTs \in Ts$

PROOF

⟨1⟩ USE DEF *Ts*
 ⟨1⟩ QED OBVIOUS

THEOREM $ThmNullTsIsntSignedTs \triangleq NullTs \notin SignedTs$

PROOF

⟨1⟩ USE DEF *NullTs*
 ⟨1⟩ USE *NoSetContainsEverything*
 ⟨1⟩ QED OBVIOUS

PROOF OF INVARIANT $InvType$

It holds in the initial state.

THEOREM $ThmInitInvType \triangleq$
 $Init \Rightarrow InvType$

PROOF

- (1) HAVE $Init$
- (1) USE DEF $Init$
- (1) USE DEF $Pc, PcRecov, PcChkpt$
- (1) USE DEF $PcIDLE$
- (1) USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- (1) USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Just walk through each variable.

- (1) $InvType!nv$ BY $ThmInitNvIsNv$
- (1) $InvType!appPcr$ BY $ThmAppRebootIsPcr$
- (1) $InvType!semPcr$ BY $ThmSemRebootIsPcr$
- (1) $InvType!bootCtr$ OBVIOUS
- (1) $InvType!pc$ OBVIOUS
- (1) $InvType!chkptts$ BY $ThmNullTsIsTs$
- (1) $InvType!tsvalues$ OBVIOUS
- (1) $InvType!obtains$ OBVIOUS
- (1) $InvType!revokes$ OBVIOUS
- (1) QED BY DEF $InvType$

If it holds in the current state, and we perform a $Next$ action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM $ThmNextInvType \triangleq$
 $InvType \wedge [Next]_{vars} \Rightarrow InvType'$

PROOF

- (1) HAVE $InvType \wedge [Next]_{vars}$
- (1) USE DEF $InvType$
- (1) USE DEF $Pc, PcRecov, PcChkpt$
- (1) USE DEF $PcIDLE$
- (1) USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
- (1) USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Say QED here so that the rest of the proof has an indentation level. This creates a place where I can use the user interface renumbering operation to renumber all of the alternatives below.

(1) QED

Stutter step.

- (2)1. CASE $vars' = vars$
- (3) USE (2)1

- ⟨3⟩ USE DEF *vars*
- ⟨3⟩ QED OBVIOUS

Walk through all *Next* alternatives.

- ⟨2⟩2. CASE *NextObtainAccess*
 - ⟨3⟩ USE *NextObtainAccess*
 - ⟨3⟩ USE DEF *NextObtainAccess*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩3. CASE *NextProveRevoke*
 - ⟨3⟩ USE *NextProveRevoke*
 - ⟨3⟩ USE DEF *NextProveRevoke*
 - ⟨3⟩ $PcrPrior(appPcr) \in Pcr$ BY *ThmPcrPriorIsPcr*
 - ⟨3⟩ QED BY *ThmPcrExtendIsPcr* DEF *Pcrx*
- ⟨2⟩4. CASE *NextReboot*
 - ⟨3⟩ USE *NextReboot*
 - ⟨3⟩ USE DEF *NextReboot*
 - ⟨3⟩ $InvType!appPcr' \text{ BY } ThmAppRebootIsPcr$
 - ⟨3⟩ $InvType!semPcr' \text{ BY } ThmSemRebootIsPcr$
 - ⟨3⟩ $InvType!chkptts' \text{ BY } ThmNullTsIsTs$
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩5. CASE *NextForgetSealTs*
 - ⟨3⟩ USE *NextForgetSealTs*
 - ⟨3⟩ USE DEF *NextForgetSealTs*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩6. CASE *NextExtendAppPcr*
 - ⟨3⟩ USE *NextExtendAppPcr*
 - ⟨3⟩ USE DEF *NextExtendAppPcr*
 - ⟨3⟩ $appPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩7. CASE *NextExtendSemPcr*
 - ⟨3⟩ USE *NextExtendSemPcr*
 - ⟨3⟩ USE DEF *NextExtendSemPcr*
 - ⟨3⟩ $InvType!semPcr' \text{ BY } ThmPcrExtendIsPcr$ DEF *Pcrx*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩9. CASE *NextIncBootCtr*
 - ⟨3⟩ USE *NextIncBootCtr*
 - ⟨3⟩ USE DEF *NextIncBootCtr*
 - ⟨3⟩ $bootCtr' \in Nat$
 - ⟨4⟩1. $bootCtr + 1 \in Nat$ BY *SMT*
 - ⟨4⟩2. $bootCtr' = bootCtr + 1$ OBVIOUS
 - ⟨4⟩ QED BY ⟨4⟩1, ⟨4⟩2
 - ⟨3⟩ QED OBVIOUS

- ⟨2⟩10. CASE *NextEnterSemRecov*
 ⟨3⟩ USE *NextEnterSemRecov*
 ⟨3⟩ USE DEF *NextEnterSemRecov*
 ⟨3⟩ *InvType!semPcr'* BY *ThmSemProtectIsPcr*
 ⟨3⟩ *InvType!pc'* OBVIOUS
 ⟨3⟩ *InvType!chkptts'* OBVIOUS
 ⟨3⟩ QED OBVIOUS
- ⟨2⟩11. CASE *NextSemRecov1WhenCorrect*
 ⟨3⟩ USE *NextSemRecov1WhenCorrect*
 ⟨3⟩ USE DEF *NextSemRecov1WhenCorrect*
 ⟨3⟩ QED OBVIOUS
- ⟨2⟩12. CASE *NextSemRecov1WhenIncorrect*
 ⟨3⟩ USE *NextSemRecov1WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemRecov1WhenIncorrect*
 ⟨3⟩ *semPcr' ∈ Pcr* BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 ⟨3⟩ QED OBVIOUS
- ⟨2⟩13. CASE *NextSemRecov2*
 ⟨3⟩ USE *NextSemRecov2*
 ⟨3⟩ USE DEF *NextSemRecov2*
 ⟨3⟩ *nv' ∈ Nv*
 ⟨4⟩1. *nv ∈ Nv* OBVIOUS
 ⟨4⟩2. *nv'.current ∈ BOOLEAN* BY DEF *Nv*
 ⟨4⟩3. *nv' = [nv EXCEPT !.current = nv'.current]* OBVIOUS
 ⟨4⟩ QED BY ONLY ⟨4⟩1, ⟨4⟩2, ⟨4⟩3 DEF *Nv*
 ⟨3⟩ QED OBVIOUS
- ⟨2⟩14. CASE *NextSemRecov3*
 ⟨3⟩ USE *NextSemRecov3*
 ⟨3⟩ USE DEF *NextSemRecov3*
 ⟨3⟩ *semPcr' ∈ Pcr* BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 ⟨3⟩ *pc' ∈ Pc* BY DEF *Pc, PcRecov*
 ⟨3⟩ QED OBVIOUS
- ⟨2⟩15. CASE *NextSealTs*
 ⟨3⟩ USE *NextSealTs*
 ⟨3⟩ USE DEF *NextSealTs*
 ⟨3⟩ *semPcr' ∈ Pcr* BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 ⟨3⟩ *tvalues' ∈ SUBSET Ts*
 ⟨4⟩1. *tvalues ∈ SUBSET Ts* OBVIOUS
 ⟨4⟩ DEFINE *ts* $\hat{=}$ *NextSealTs! : !ts*
 ⟨4⟩2. *ts ∈ Ts* BY DEF *Ts, SignedTs*
 ⟨4⟩3. *tvalues' = tvalues ∪ {ts}* OBVIOUS
 ⟨4⟩ QED BY ⟨4⟩1, ⟨4⟩2, ⟨4⟩3
 ⟨3⟩ QED OBVIOUS

- (2)16. CASE *NextEnterSemChkpt*
 ⟨3⟩ USE *NextEnterSemChkpt*
 ⟨3⟩ USE DEF *NextEnterSemChkpt*
 ⟨3⟩ $semPcr' \in Pcr$ BY *ThmSemProtectIsPcr*
 ⟨3⟩ QED OBVIOUS
- (2)17. CASE *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ $chkptts' \in SignedTs$
 ⟨4⟩ USE DEF *EnterSemChkptPredicate*
 ⟨4⟩ USE DEF *Ts*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ QED OBVIOUS
- (2)18. CASE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ $semPcr' \in Pcr$ BY *ThmPcrExtendIsPcr* DEF *Pcrx*
 ⟨3⟩ QED OBVIOUS
- (2)19. CASE *NextSemChkpt2*
 ⟨3⟩ USE *NextSemChkpt2*
 ⟨3⟩ USE DEF *NextSemChkpt2*
 ⟨3⟩ $nv' \in Nv$
 ⟨4⟩ DEFINE $nvappPcr1 \triangleq NextSemChkpt2! : !nvappPcr1$
 ⟨4⟩1. $nvappPcr1 \in Pcr$ BY DEF *SignedTs*
 ⟨4⟩ QED BY ⟨4⟩1 DEF *Nv*
 ⟨3⟩ QED OBVIOUS
- (2)20. CASE *NextSemChkpt3*
 ⟨3⟩ USE *NextSemChkpt3*
 ⟨3⟩ USE DEF *NextSemChkpt3*
 ⟨3⟩ $bootCtr' \in Nat$
 ⟨4⟩1. $bootCtr \in Nat$ OBVIOUS
 ⟨4⟩2. $bootCtr + 1 \in Nat$ BY ONLY ⟨4⟩1, *SMT*
 ⟨4⟩ QED BY ⟨4⟩1, ⟨4⟩2
 ⟨3⟩ QED OBVIOUS
- (2)21. CASE *NextSemChkpt4*
 ⟨3⟩ USE *NextSemChkpt4*
 ⟨3⟩ USE DEF *NextSemChkpt4*
 ⟨3⟩ $nv' \in Nv$ BY DEF *Nv*
 ⟨3⟩ QED OBVIOUS
- (2)22. CASE *NextSemChkpt5*
 ⟨3⟩ USE *NextSemChkpt5*
 ⟨3⟩ USE DEF *NextSemChkpt5*

$\langle 3 \rangle \text{ semPcr}' \in \text{Pcr} \text{ BY } \text{ThmPcrExtendIsPcr} \text{ DEF } \text{Pcrx}$
 $\langle 3 \rangle \text{ QED OBVIOUS}$

$\langle 2 \rangle \text{ QED}$
 BY $\langle 2 \rangle 1,$
 $\langle 2 \rangle 2, \langle 2 \rangle 3, \langle 2 \rangle 4, \langle 2 \rangle 5, \langle 2 \rangle 6, \langle 2 \rangle 7, \langle 2 \rangle 9, \langle 2 \rangle 10,$
 $\langle 2 \rangle 11, \langle 2 \rangle 12, \langle 2 \rangle 13, \langle 2 \rangle 14, \langle 2 \rangle 15, \langle 2 \rangle 16, \langle 2 \rangle 17, \langle 2 \rangle 18,$
 $\langle 2 \rangle 19, \langle 2 \rangle 20, \langle 2 \rangle 21, \langle 2 \rangle 22$
 DEF *Next*

It is an invariant of the specification.

THEOREM $\text{ThmInvType} \triangleq$
 $\text{Spec} \Rightarrow \Box \text{InvType}$

PROOF

$\langle 1 \rangle \text{ Init} \Rightarrow \text{InvType} \text{ BY } \text{ThmInitInvType}$
 $\langle 1 \rangle \text{ InvType} \wedge [\text{Next}]_{\text{vars}} \Rightarrow \text{InvType}' \text{ BY } \text{ThmNextInvType}$
 $\langle 1 \rangle \text{ QED}$

PROOF OF INVARIANT *InvInSemProtect*

It holds in the initial state.

THEOREM $\text{ThmInitInvInSemProtect} \triangleq$
 $\text{Init} \Rightarrow \text{InvInSemProtect}$

PROOF

$\langle 1 \rangle \text{ HAVE } \text{Init}$
 $\langle 1 \rangle \text{ InvType} \text{ BY } \text{ThmInitInvType}$
 $\langle 1 \rangle \text{ USE DEF } \text{Init}$
 $\langle 1 \rangle \text{ USE DEF } \text{InSem}$
 $\langle 1 \rangle \text{ QED BY DEF } \text{InvInSemProtect}$

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

Note that none of the *Bug** definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM $\text{ThmNextInvInSemProtect} \triangleq$
 $\text{InvInSemProtect} \wedge [\text{Next}]_{\text{vars}} \Rightarrow \text{InvInSemProtect}'$

PROOF

⟨1⟩ HAVE $InvInSemProtect \wedge [Next]_{vars}$
 ⟨1⟩ USE DEF $InvInSemProtect$
 ⟨1⟩ USE DEF $InSem$

⟨1⟩ $InvType'$ BY $ThmNextInvType$
 ⟨1⟩ $InvInSemProtect ! goal'$

(2) USE DEF $PcIDLE$
 (2) USE DEF $PcRECOV1, PcRECOV2, PcRECOV3$
 (2) USE DEF $PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5$

Stutter step.

(2)1. CASE $vars' = vars$
 ⟨3⟩ USE ⟨2⟩1
 ⟨3⟩ USE DEF $vars$
 ⟨3⟩ QED OBVIOUS

Walk through all $Next$ alternatives.

(2)2. CASE $NextObtainAccess$
 ⟨3⟩ USE $NextObtainAccess$
 ⟨3⟩ USE DEF $NextObtainAccess$
 ⟨3⟩ QED OBVIOUS

(2)3. CASE $NextProveRevoke$
 ⟨3⟩ USE $NextProveRevoke$
 ⟨3⟩ USE DEF $NextProveRevoke$
 ⟨3⟩ QED OBVIOUS

(2)4. CASE $NextReboot$
 ⟨3⟩ USE $NextReboot$
 ⟨3⟩ USE DEF $NextReboot$
 ⟨3⟩ USE DEF $PcrLeq$
 ⟨3⟩ USE DEF $SemProtect$
 ⟨3⟩ USE DEF $SemReboot$
 ⟨3⟩ USE DEF $PcrInit$
 ⟨3⟩ USE DEF $Pcri$
 ⟨3⟩ USE $AssSemProtect$
 ⟨3⟩ QED OBVIOUS

(2)5. CASE $NextForgetSealTs$
 ⟨3⟩ USE $NextForgetSealTs$
 ⟨3⟩ USE DEF $NextForgetSealTs$
 ⟨3⟩ QED OBVIOUS

(2)6. CASE $NextExtendAppPcr$
 ⟨3⟩ USE $NextExtendAppPcr$
 ⟨3⟩ USE DEF $NextExtendAppPcr$

- ⟨3⟩ QED OBVIOUS
- ⟨2⟩7. CASE *NextExtendSemPcr*
 - ⟨3⟩ USE *NextExtendSemPcr*
 - ⟨3⟩ USE DEF *NextExtendSemPcr*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩9. CASE *NextIncBootCtr*
 - ⟨3⟩ USE *NextIncBootCtr*
 - ⟨3⟩ USE DEF *NextIncBootCtr*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩10. CASE *NextEnterSemRecov*
 - ⟨3⟩ USE *NextEnterSemRecov*
 - ⟨3⟩ USE DEF *NextEnterSemRecov*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩11. CASE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenCorrect*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩12. CASE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *NextSemRecov1 WhenIncorrect*
 - ⟨3⟩ USE DEF *PcrExtend*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩13. CASE *NextSemRecov2*
 - ⟨3⟩ USE *NextSemRecov2*
 - ⟨3⟩ USE DEF *NextSemRecov2*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩14. CASE *NextSemRecov3*
 - ⟨3⟩ USE *NextSemRecov3*
 - ⟨3⟩ USE DEF *NextSemRecov3*
 - ⟨3⟩ USE DEF *PcrExtend*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩15. CASE *NextSealTs*
 - ⟨3⟩ USE *NextSealTs*
 - ⟨3⟩ USE DEF *NextSealTs*
 - ⟨3⟩ QED OBVIOUS
- ⟨2⟩16. CASE *NextEnterSemChkpt*
 - ⟨3⟩ USE *NextEnterSemChkpt*
 - ⟨3⟩ USE DEF *NextEnterSemChkpt*
 - ⟨3⟩ QED OBVIOUS

(2)17. CASE *NextSemChkpt1 WhenCorrect*
 (3) USE *NextSemChkpt1 WhenCorrect*
 (3) USE DEF *NextSemChkpt1 WhenCorrect*
 (3) QED OBVIOUS

(2)18. CASE *NextSemChkpt1 WhenIncorrect*
 (3) USE *NextSemChkpt1 WhenIncorrect*
 (3) USE DEF *NextSemChkpt1 WhenIncorrect*
 (3) USE DEF *PcrExtend*
 (3) QED OBVIOUS

(2)19. CASE *NextSemChkpt2*
 (3) USE *NextSemChkpt2*
 (3) USE DEF *NextSemChkpt2*
 (3) QED OBVIOUS

(2)20. CASE *NextSemChkpt3*
 (3) USE *NextSemChkpt3*
 (3) USE DEF *NextSemChkpt3*
 (3) QED OBVIOUS

(2)21. CASE *NextSemChkpt4*
 (3) USE *NextSemChkpt4*
 (3) USE DEF *NextSemChkpt4*
 (3) QED OBVIOUS

(2)22. CASE *NextSemChkpt5*
 (3) USE *NextSemChkpt5*
 (3) USE DEF *NextSemChkpt5*
 (3) USE DEF *PcrExtend*
 (3) QED OBVIOUS

(2) QED
 BY (2)1,
 (2)2, (2)3, (2)4, (2)5, (2)6, (2)7, (2)9, (2)10,
 (2)11, (2)12, (2)13, (2)14, (2)15, (2)16, (2)17, (2)18,
 (2)19, (2)20, (2)21, (2)22
 DEF *Next*
 (1) QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvInSemProtect* \triangleq
Spec \Rightarrow \square *InvInSemProtect*

PROOF

(1) *Init* \Rightarrow *InvInSemProtect* BY *ThmInitInvInSemProtect*
 (1) *InvInSemProtect* \wedge $[Next]_{vars} \Rightarrow$ *InvInSemProtect'*

BY *ThmNextInvInSemProtect*

(1) USE DEF *Spec*

(1) QED

PROOF OF INVARIANT *InvUnreachableSemProtect*

It holds in the initial state.

THEOREM *ThmInitInvUnreachableSemProtect* \triangleq
Init \Rightarrow *InvUnreachableSemProtect*

PROOF

(1) HAVE *Init*
(1) *InvType* BY *ThmInitInvType*
(1) *InvInSemProtect* BY *ThmInitInvInSemProtect*
(1) USE DEF *Init*
(1) USE DEF *InSem*
(1) \neg *InSem* OBVIOUS
(1) \neg *PcrLeq*(*semPcr*, *SemProtect*)
 (2) USE DEF *PcrLeq*
 (2) USE DEF *SemProtect*
 (2) USE DEF *SemReboot*
 (2) USE DEF *PcrInit*
 (2) USE DEF *Pcri*
 (2) QED BY *AssSemProtect*
(1) QED BY DEF *InvUnreachableSemProtect*

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM *ThmNextInvUnreachableSemProtect* \triangleq
InvUnreachableSemProtect \wedge [*Next*]_{vars} \Rightarrow *InvUnreachableSemProtect'*

PROOF

(1) HAVE *InvUnreachableSemProtect* \wedge [*Next*]_{vars}
(1) USE DEF *InvUnreachableSemProtect*
(1) USE DEF *InSem*
(1) USE DEF *InvInSemProtect*

⟨1⟩ *InvType*' BY *ThmNextInvType*
 ⟨1⟩ *InvInSemProtect*' BY *ThmNextInvInSemProtect*
 ⟨1⟩ *InvUnreachableSemProtect* ! *goal*'

⟨2⟩ USE DEF *PcIDLE*
 ⟨2⟩ USE DEF *PcRECOV1*, *PcRECOV2*, *PcRECOV3*
 ⟨2⟩ USE DEF *PcCHKPT1*, *PcCHKPT2*, *PcCHKPT3*, *PcCHKPT4*, *PcCHKPT5*

Stutter step.

⟨2⟩1. CASE *vars' = vars*
 ⟨3⟩ USE ⟨2⟩1
 ⟨3⟩ USE DEF *vars*
 ⟨3⟩ QED OBVIOUS

Walk through all *Next* alternatives.

⟨2⟩2. CASE *NextObtainAccess*
 ⟨3⟩ USE *NextObtainAccess*
 ⟨3⟩ USE DEF *NextObtainAccess*
 ⟨3⟩ QED OBVIOUS

⟨2⟩3. CASE *NextProveRevoke*
 ⟨3⟩ USE *NextProveRevoke*
 ⟨3⟩ USE DEF *NextProveRevoke*
 ⟨3⟩ QED OBVIOUS

⟨2⟩4. CASE *NextReboot*
 ⟨3⟩ USE *NextReboot*
 ⟨3⟩ USE DEF *NextReboot*
 ⟨3⟩ USE DEF *PcrLeq*
 ⟨3⟩ USE DEF *SemProtect*
 ⟨3⟩ USE DEF *SemReboot*
 ⟨3⟩ USE DEF *PcrInit*
 ⟨3⟩ USE DEF *Pcri*
 ⟨3⟩ USE *AssSemProtect*
 ⟨3⟩ QED OBVIOUS

⟨2⟩5. CASE *NextForgetSealTs*
 ⟨3⟩ USE *NextForgetSealTs*
 ⟨3⟩ USE DEF *NextForgetSealTs*
 ⟨3⟩ QED OBVIOUS

⟨2⟩6. CASE *NextExtendAppPcr*
 ⟨3⟩ USE *NextExtendAppPcr*
 ⟨3⟩ USE DEF *NextExtendAppPcr*
 ⟨3⟩ QED OBVIOUS

⟨2⟩7. CASE *NextExtendSemPcr*
 ⟨3⟩ USE *NextExtendSemPcr*

⟨3⟩ USE DEF *NextExtendSemPcr*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 ⟨4⟩ *SemProtect* $\in Pcr$ BY *ThmSemProtectIsPcr*
 ⟨4⟩ QED BY *ThmPcrExtendSourceUnreachable*
 ⟨3⟩ QED OBVIOUS

⟨2⟩9. CASE *NextIncBootCtr*
 ⟨3⟩ USE *NextIncBootCtr*
 ⟨3⟩ USE DEF *NextIncBootCtr*
 ⟨3⟩ QED OBVIOUS

⟨2⟩10. CASE *NextEnterSemRecov*
 ⟨3⟩ USE *NextEnterSemRecov*
 ⟨3⟩ USE DEF *NextEnterSemRecov*
 ⟨3⟩ QED OBVIOUS

⟨2⟩11. CASE *NextSemRecov1 WhenCorrect*
 ⟨3⟩ USE *NextSemRecov1 WhenCorrect*
 ⟨3⟩ USE DEF *NextSemRecov1 WhenCorrect*
 ⟨3⟩ QED OBVIOUS

⟨2⟩12. CASE *NextSemRecov1 WhenIncorrect*
 ⟨3⟩ USE *NextSemRecov1 WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemRecov1 WhenIncorrect*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ USE DEF *Pcrx*
 ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 ⟨3⟩ QED OBVIOUS

⟨2⟩13. CASE *NextSemRecov2*
 ⟨3⟩ USE *NextSemRecov2*
 ⟨3⟩ USE DEF *NextSemRecov2*
 ⟨3⟩ QED OBVIOUS

⟨2⟩14. CASE *NextSemRecov3*
 ⟨3⟩ USE *NextSemRecov3*
 ⟨3⟩ USE DEF *NextSemRecov3*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ USE DEF *Pcrx*
 ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 ⟨3⟩ QED OBVIOUS

⟨2⟩15. CASE *NextSealTs*
 ⟨3⟩ USE *NextSealTs*
 ⟨3⟩ USE DEF *NextSealTs*

```

⟨3⟩ USE DEF InvType
*****
OPT: REWORKED PROOF HERE TO DEAL WITH EXTENSION OF SEM PCR
*****
⟨3⟩ CASE BugSealNoExt OBVIOUS
⟨3⟩ CASE  $\neg$ BugSealNoExt
  ⟨4⟩  $\neg$ PcrLeq(semPcr', SemProtect)
    ⟨5⟩ SemProtect  $\in$  Pcr BY ThmSemProtectIsPcr
    ⟨5⟩ QED BY ThmPcrExtendSourceUnreachable DEF Pcrx
  ⟨4⟩ QED OBVIOUS
⟨3⟩ QED OBVIOUS

(2)16. CASE NextEnterSemChkpt
  ⟨3⟩ USE NextEnterSemChkpt
  ⟨3⟩ USE DEF NextEnterSemChkpt
  ⟨3⟩ QED OBVIOUS

(2)17. CASE NextSemChkpt1 WhenCorrect
  ⟨3⟩ USE NextSemChkpt1 WhenCorrect
  ⟨3⟩ USE DEF NextSemChkpt1 WhenCorrect
  ⟨3⟩ QED OBVIOUS

(2)18. CASE NextSemChkpt1 WhenIncorrect
  ⟨3⟩ USE NextSemChkpt1 WhenIncorrect
  ⟨3⟩ USE DEF NextSemChkpt1 WhenIncorrect
  ⟨3⟩ USE DEF InvType
  ⟨3⟩ USE DEF Pcrx
  ⟨3⟩  $\neg$ PcrLeq(semPcr', SemProtect)
  ⟨4⟩ QED BY ThmPcrExtendSelfUnreachable
  ⟨3⟩ QED OBVIOUS

(2)19. CASE NextSemChkpt2
  ⟨3⟩ USE NextSemChkpt2
  ⟨3⟩ USE DEF NextSemChkpt2
  ⟨3⟩ QED OBVIOUS

(2)20. CASE NextSemChkpt3
  ⟨3⟩ USE NextSemChkpt3
  ⟨3⟩ USE DEF NextSemChkpt3
  ⟨3⟩ QED OBVIOUS

(2)21. CASE NextSemChkpt4
  ⟨3⟩ USE NextSemChkpt4
  ⟨3⟩ USE DEF NextSemChkpt4
  ⟨3⟩ QED OBVIOUS

(2)22. CASE NextSemChkpt5
  ⟨3⟩ USE NextSemChkpt5

```

⟨3⟩ USE DEF *NextSemChkpt5*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ USE DEF *Pcrx*
 ⟨3⟩ $\neg PcrLeq(semPcr', SemProtect)$
 ⟨4⟩ QED BY *ThmPcrExtendSelfUnreachable*
 ⟨3⟩ QED OBVIOUS

 ⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩9, ⟨2⟩10,
 ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,
 ⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22
 DEF *Next*
 ⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvUnreachableSemProtect* \triangleq
 $Spec \Rightarrow \square InvUnreachableSemProtect$

PROOF

⟨1⟩ $Init \Rightarrow InvUnreachableSemProtect$ BY *ThmInitInvUnreachableSemProtect*
 ⟨1⟩ $InvUnreachableSemProtect \wedge [Next]_{vars} \Rightarrow InvUnreachableSemProtect'$
 BY *ThmNextInvUnreachableSemProtect*
 ⟨1⟩ USE DEF *Spec*
 ⟨1⟩ QED

PROOF OF INVARIANT *InvNvProtection*

It is an invariant of the specification.

THEOREM *ThmInvNvProtection* \triangleq
 $Spec \Rightarrow \square InvNvProtection$

PROOF

⟨1⟩ $InvInSemProtect \wedge InvUnreachableSemProtect \Rightarrow InvNvProtection$
 ⟨2⟩ HAVE $InvInSemProtect \wedge InvUnreachableSemProtect$

⟨2⟩ USE DEF *InvInSemProtect*
 ⟨2⟩ USE DEF *InvUnreachableSemProtect*
 ⟨2⟩ USE DEF *InvNvProtection*
 ⟨2⟩1. CASE *InSem* BY ⟨2⟩1
 ⟨2⟩2. CASE \neg *InSem*
 Proof by contradiction.
 ⟨3⟩1. CASE *semPcr* \neq *SemProtect* BY ⟨3⟩1
 ⟨3⟩2. CASE *semPcr* = *SemProtect*
 ⟨4⟩1. \neg *PcrLeq*(*semPcr*, *SemProtect*) BY ⟨2⟩2
 ⟨4⟩2. *PcrLeq*(*semPcr*, *SemProtect*)
 ⟨5⟩ *semPcr* \in *Pcr* BY *InvType* DEF *InvType*
 ⟨5⟩ QED BY ⟨3⟩2, *ThmPcrLeqIsReflexive*
 ⟨4⟩ QED BY ⟨4⟩1, ⟨4⟩2
 ⟨3⟩ QED BY ⟨3⟩2, ⟨3⟩1
 ⟨2⟩ QED BY ⟨2⟩2, ⟨2⟩1
 ⟨1⟩ *Spec* \Rightarrow \square *InvInSemProtect* BY *ThmInvInSemProtect*
 ⟨1⟩ *Spec* \Rightarrow \square *InvUnreachableSemProtect* BY *ThmInvUnreachableSemProtect*
 ⟨1⟩ QED

PROOF OF INVARIANT *InvSignedTsLeqBoot*

It holds in the initial state.

THEOREM *ThmInitInvSignedTsLeqBoot* \triangleq
Init \Rightarrow *InvSignedTsLeqBoot*

PROOF

⟨1⟩ HAVE *Init*
 ⟨1⟩ *InvType* BY *ThmInitInvType*
 ⟨1⟩ USE DEF *Init*
 ⟨1⟩ USE DEF *InvSignedTsLeqBoot*
 ⟨1⟩ *InvSignedTsLeqBoot* ! *goal*
 ⟨2⟩ TAKE *ts* \in *tsvalues* \cup {*chkptts*}

- ⟨2⟩ QED BY *ThmNullTsIsntSignedTs*
- ⟨1⟩ QED OBVIOUS

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM *ThmNextInvSignedTsLeqBoot* \triangleq
InvSignedTsLeqBoot \wedge [*Next*]_{vars} \Rightarrow *InvSignedTsLeqBoot'*

PROOF

- ⟨1⟩ HAVE *InvSignedTsLeqBoot* \wedge [*Next*]_{vars}
- ⟨1⟩ USE DEF *InvSignedTsLeqBoot*
- ⟨1⟩ *InvType'* BY *ThmNextInvType*
- ⟨1⟩ *InvSignedTsLeqBoot' !goal'*

Stutter step.

- ⟨2⟩1. CASE *vars' = vars*
- ⟨3⟩ USE ⟨2⟩1
- ⟨3⟩ USE DEF *vars*
- ⟨3⟩ QED OBVIOUS

Walk through all *Next* alternatives.

- ⟨2⟩2. CASE *NextObtainAccess*
- ⟨3⟩ USE *NextObtainAccess*
- ⟨3⟩ USE DEF *NextObtainAccess*
- ⟨3⟩ QED OBVIOUS
- ⟨2⟩3. CASE *NextProveRevoke*
- ⟨3⟩ USE *NextProveRevoke*
- ⟨3⟩ USE DEF *NextProveRevoke*
- ⟨3⟩ QED OBVIOUS
- ⟨2⟩4. CASE *NextReboot*
- ⟨3⟩ USE *NextReboot*
- ⟨3⟩ USE DEF *NextReboot*
- ⟨3⟩ QED BY *ThmNullTsIsntSignedTs*
- ⟨2⟩5. CASE *NextForgetSealTs*
- ⟨3⟩ USE *NextForgetSealTs*
- ⟨3⟩ USE DEF *NextForgetSealTs*
- ⟨3⟩ QED OBVIOUS
- ⟨2⟩6. CASE *NextExtendAppPcr*
- ⟨3⟩ USE *NextExtendAppPcr*
- ⟨3⟩ USE DEF *NextExtendAppPcr*
- ⟨3⟩ QED OBVIOUS

- (2)7. CASE *NextExtendSemPcr*
 - (3) USE *NextExtendSemPcr*
 - (3) USE DEF *NextExtendSemPcr*
 - (3) QED OBVIOUS

- (2)9. CASE *NextIncBootCtr*
 - (3) USE *NextIncBootCtr*
 - (3) USE DEF *NextIncBootCtr*
 - (3) USE DEF *InvType*
 - (3) USE DEF *SignedTs*
 - (3) $bootCtr \leq bootCtr + 1BY$ *ThmNatMore*
 - (3) QED BY *ThmNatLeqIsTransitive*

- (2)10. CASE *NextEnterSemRecov*
 - (3) USE *NextEnterSemRecov*
 - (3) USE DEF *NextEnterSemRecov*
 - (3) QED OBVIOUS

- (2)11. CASE *NextSemRecov1 WhenCorrect*
 - (3) USE *NextSemRecov1 WhenCorrect*
 - (3) USE DEF *NextSemRecov1 WhenCorrect*
 - (3) QED OBVIOUS

- (2)12. CASE *NextSemRecov1 WhenIncorrect*
 - (3) USE *NextSemRecov1 WhenIncorrect*
 - (3) USE DEF *NextSemRecov1 WhenIncorrect*
 - (3) QED OBVIOUS

- (2)13. CASE *NextSemRecov2*
 - (3) USE *NextSemRecov2*
 - (3) USE DEF *NextSemRecov2*
 - (3) QED OBVIOUS

- (2)14. CASE *NextSemRecov3*
 - (3) USE *NextSemRecov3*
 - (3) USE DEF *NextSemRecov3*
 - (3) QED OBVIOUS

- (2)15. CASE *NextSealTs*
 - (3) USE *NextSealTs*
 - (3) USE DEF *NextSealTs*
 - (3) DEFINE $ts \triangleq NextSealTs! : !ts$
 - (3) $ts.bootCtr \leq bootCtr$
 - (4) USE DEF *InvType*
 - (4) USE *ThmNatLeqIsReflexive*
 - (4) QED OBVIOUS
 - (3) QED OBVIOUS

⟨2⟩ 16. CASE *NextEnterSemChkpt*
 ⟨3⟩ USE *NextEnterSemChkpt*
 ⟨3⟩ USE DEF *NextEnterSemChkpt*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ 17. CASE *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ 18. CASE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ 19. CASE *NextSemChkpt2*
 ⟨3⟩ USE *NextSemChkpt2*
 ⟨3⟩ USE DEF *NextSemChkpt2*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ 20. CASE *NextSemChkpt3*
 ⟨3⟩ USE *NextSemChkpt3*
 ⟨3⟩ USE DEF *NextSemChkpt3*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ USE DEF *SignedTs*
 ⟨3⟩ $bootCtr \leq bootCtr'$
 ⟨4⟩ USE *ThmNatLeqIsReflexive*
 ⟨4⟩ USE *ThmNatMore*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ USE *ThmNatLeqIsTransitive*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ 21. CASE *NextSemChkpt4*
 ⟨3⟩ USE *NextSemChkpt4*
 ⟨3⟩ USE DEF *NextSemChkpt4*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ 22. CASE *NextSemChkpt5*
 ⟨3⟩ USE *NextSemChkpt5*
 ⟨3⟩ USE DEF *NextSemChkpt5*
 ⟨3⟩ QED OBVIOUS

⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩9, ⟨2⟩10,
 ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,
 ⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22
 DEF *Next*

⟨1⟩ QED OBVIOUS

PROOF OF INVARIANT *InvUnforgeableSemHappy*

It holds in the initial state.

THEOREM *ThmInitInvUnforgeableSemHappy* \triangleq
Init \Rightarrow *InvUnforgeableSemHappy*

PROOF

⟨1⟩ HAVE *Init*
 ⟨1⟩ *InvType* BY *ThmInitInvType*
 ⟨1⟩ *InvInSemProtect* BY *ThmInitInvInSemProtect*
 ⟨1⟩ USE DEF *Init*
 ⟨1⟩ USE DEF *InSem*
 ⟨1⟩ \neg *InSem* OBVIOUS
 ⟨1⟩ \neg *PcrLeq*(*semPcr*, *SemHappy*)
 ⟨2⟩ USE DEF *PcrLeq*
 ⟨2⟩ USE DEF *PcrInit*
 ⟨2⟩ USE DEF *PcrExtend*
 ⟨2⟩ USE DEF *SemHappy*
 ⟨2⟩ USE DEF *SemReboot*
 ⟨2⟩ USE DEF *SemProtect*
 ⟨2⟩ USE *AssSemProtect*
 ⟨2⟩ QED OBVIOUS
 ⟨1⟩ QED BY DEF *InvUnforgeableSemHappy*

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

Note that none of the Bug* definitions are needed anywhere in this proof, so this proof goes through no matter what intentional bugs are introduced.

THEOREM *ThmNextInvUnforgeableSemHappy* \triangleq
InvUnforgeableSemHappy \wedge [*Next*]_{vars} \Rightarrow *InvUnforgeableSemHappy*'

PROOF

⟨1⟩ HAVE *InvUnforgeableSemHappy* \wedge [*Next*]_{vars}
 ⟨1⟩ USE DEF *InvUnforgeableSemHappy*
 ⟨1⟩ USE DEF *InSem*

- ⟨1⟩ *InvType*'BY *ThmNextInvType*
- ⟨1⟩ *InvInSemProtect*'BY *ThmNextInvInSemProtect*
- ⟨1⟩ *InvUnforgeableSemHappy*!goal'

- ⟨2⟩ USE DEF *PcIDLE*
- ⟨2⟩ USE DEF *PcRECOV1*, *PcRECOV2*, *PcRECOV3*
- ⟨2⟩ USE DEF *PcCHKPT1*, *PcCHKPT2*, *PcCHKPT3*, *PcCHKPT4*, *PcCHKPT5*

Stutter step.

- ⟨2⟩1. CASE *vars'* = *vars*
- ⟨3⟩ USE ⟨2⟩1
- ⟨3⟩ USE DEF *vars*
- ⟨3⟩ QED OBVIOUS

Walk through all *Next* alternatives.

- ⟨2⟩2. CASE *NextObtainAccess*
- ⟨3⟩ USE *NextObtainAccess*
- ⟨3⟩ USE DEF *NextObtainAccess*
- ⟨3⟩ QED OBVIOUS

- ⟨2⟩3. CASE *NextProveRevoke*
- ⟨3⟩ USE *NextProveRevoke*
- ⟨3⟩ USE DEF *NextProveRevoke*
- ⟨3⟩ QED OBVIOUS

- ⟨2⟩4. CASE *NextReboot*
- ⟨3⟩ USE *NextReboot*
- ⟨3⟩ USE DEF *NextReboot*
- ⟨3⟩ USE DEF *PcrLeq*
- ⟨3⟩ USE DEF *PcrInit*
- ⟨3⟩ USE DEF *PcrExtend*
- ⟨3⟩ USE DEF *SemHappy*
- ⟨3⟩ USE DEF *SemReboot*
- ⟨3⟩ USE DEF *SemProtect*
- ⟨3⟩ USE *AssSemProtect*
- ⟨3⟩ QED OBVIOUS

- ⟨2⟩5. CASE *NextForgetSealTs*
- ⟨3⟩ USE *NextForgetSealTs*
- ⟨3⟩ USE DEF *NextForgetSealTs*
- ⟨3⟩ QED OBVIOUS

- ⟨2⟩6. CASE *NextExtendAppPcr*
- ⟨3⟩ USE *NextExtendAppPcr*
- ⟨3⟩ USE DEF *NextExtendAppPcr*
- ⟨3⟩ QED OBVIOUS

- ⟨2⟩7. CASE *NextExtendSemPcr*

```

⟨3⟩ USE NextExtendSemPcr
⟨3⟩ USE DEF NextExtendSemPcr
⟨3⟩ USE DEF InvType
⟨3⟩ HAVE  $\neg InSem'$ 
⟨3⟩ 1. CASE semPcr = SemHappy
  ⟨4⟩ USE ⟨3⟩1
  ⟨4⟩ QED BY ThmPcrExtendSelfUnreachable
⟨3⟩ 2. CASE  $\neg PcrLeq(semPcr, SemHappy)$ 
  ⟨4⟩ USE ⟨3⟩2
  ⟨4⟩ SemHappy  $\in Pcr$  BY ThmSemHappyIsPcr
  ⟨4⟩  $\neg PcrLeq(semPcr', SemHappy)$  BY ThmPcrExtendSourceUnreachable
  ⟨4⟩ QED OBVIOUS
⟨3⟩ QED BY ⟨3⟩1, ⟨3⟩2

⟨2⟩ 9. CASE NextIncBootCtr
  ⟨3⟩ USE NextIncBootCtr
  ⟨3⟩ USE DEF NextIncBootCtr
  ⟨3⟩ QED OBVIOUS

⟨2⟩ 10. CASE NextEnterSemRecov
  ⟨3⟩ USE NextEnterSemRecov
  ⟨3⟩ USE DEF NextEnterSemRecov
  ⟨3⟩ QED OBVIOUS

⟨2⟩ 11. CASE NextSemRecov1 WhenCorrect
  ⟨3⟩ USE NextSemRecov1 WhenCorrect
  ⟨3⟩ USE DEF NextSemRecov1 WhenCorrect
  ⟨3⟩ QED OBVIOUS

⟨2⟩ 12. CASE NextSemRecov1 WhenIncorrect
  ⟨3⟩ USE NextSemRecov1 WhenIncorrect
  ⟨3⟩ USE DEF NextSemRecov1 WhenIncorrect
  ⟨3⟩ USE DEF InvType
  ⟨3⟩ USE DEF Pcrx
  ⟨3⟩ semPcr = SemProtect BY DEF InvInSemProtect
  ⟨3⟩ USE DEF SemHappy
  ⟨3⟩ USE AssSemHappy
  ⟨3⟩ USE ThmPcrExtendIncompatible
  ⟨3⟩ QED OBVIOUS

⟨2⟩ 13. CASE NextSemRecov2
  ⟨3⟩ USE NextSemRecov2
  ⟨3⟩ USE DEF NextSemRecov2
  ⟨3⟩ QED OBVIOUS

⟨2⟩ 14. CASE NextSemRecov3
  ⟨3⟩ USE NextSemRecov3
  ⟨3⟩ USE DEF NextSemRecov3

```

```

⟨3⟩ semPcr = SemProtect BY DEF InvInSemProtect
⟨3⟩ USE DEF SemHappy
⟨3⟩ QED OBVIOUS

⟨2⟩ 15. CASE NextSealTs
⟨3⟩ USE NextSealTs
⟨3⟩ USE DEF NextSealTs
⟨3⟩ USE DEF InvType

```

```

*****
OPT: REWORKED PROOF HERE TO DEAL WITH EXTENSION OF SEM PCR
*****

```

```

⟨3⟩ CASE BugSealNoExt OBVIOUS
⟨3⟩ CASE  $\neg$ BugSealNoExt
  ⟨4⟩ HAVE  $\neg$ InSem'
  ⟨4⟩ 1. CASE semPcr = SemHappy
    ⟨5⟩ USE ⟨4⟩ 1
    ⟨5⟩ QED BY ThmPcrExtendSelfUnreachable DEF Pcrx
  ⟨4⟩ 2. CASE  $\neg$ PcrLeq(semPcr, SemHappy)
    ⟨5⟩ USE ⟨4⟩ 2
    ⟨5⟩ SemHappy  $\in$  Pcr BY ThmSemHappyIsPcr
    ⟨5⟩  $\neg$ PcrLeq(semPcr', SemHappy) BY ThmPcrExtendSourceUnreachable DEF Pcrx
    ⟨5⟩ QED OBVIOUS
  ⟨4⟩ QED BY ⟨4⟩ 1, ⟨4⟩ 2
⟨3⟩ QED OBVIOUS

⟨2⟩ 16. CASE NextEnterSemChkpt
⟨3⟩ USE NextEnterSemChkpt
⟨3⟩ USE DEF NextEnterSemChkpt
⟨3⟩ QED OBVIOUS

⟨2⟩ 17. CASE NextSemChkpt1 WhenCorrect
⟨3⟩ USE NextSemChkpt1 WhenCorrect
⟨3⟩ USE DEF NextSemChkpt1 WhenCorrect
⟨3⟩ QED OBVIOUS

⟨2⟩ 18. CASE NextSemChkpt1 WhenIncorrect
⟨3⟩ USE NextSemChkpt1 WhenIncorrect
⟨3⟩ USE DEF NextSemChkpt1 WhenIncorrect
⟨3⟩ USE DEF InvType
⟨3⟩ USE DEF Pcrx
⟨3⟩ semPcr = SemProtect BY DEF InvInSemProtect
⟨3⟩ USE DEF SemHappy
⟨3⟩ USE AssSemHappy
⟨3⟩ USE ThmPcrExtendIncompatible
⟨3⟩ QED OBVIOUS

⟨2⟩ 19. CASE NextSemChkpt2
⟨3⟩ USE NextSemChkpt2

```

⟨3⟩ USE DEF *NextSemChkpt2*
 ⟨3⟩ QED OBVIOUS

 ⟨2⟩20. CASE *NextSemChkpt3*
 ⟨3⟩ USE *NextSemChkpt3*
 ⟨3⟩ USE DEF *NextSemChkpt3*
 ⟨3⟩ QED OBVIOUS

 ⟨2⟩21. CASE *NextSemChkpt4*
 ⟨3⟩ USE *NextSemChkpt4*
 ⟨3⟩ USE DEF *NextSemChkpt4*
 ⟨3⟩ QED OBVIOUS

 ⟨2⟩22. CASE *NextSemChkpt5*
 ⟨3⟩ USE *NextSemChkpt5*
 ⟨3⟩ USE DEF *NextSemChkpt5*
 ⟨3⟩ USE DEF *InvType*
 ⟨3⟩ USE DEF *Pcrx*
 ⟨3⟩ *semPcr = SemProtectBY DEF InvInSemProtect*
 ⟨3⟩ USE DEF *SemHappy*
 ⟨3⟩ USE *AssSemHappy*
 ⟨3⟩ USE *ThmPcrExtendIncompatible*
 ⟨3⟩ QED OBVIOUS

 ⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩7, ⟨2⟩9, ⟨2⟩10,
 ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17, ⟨2⟩18,
 ⟨2⟩19, ⟨2⟩20, ⟨2⟩21, ⟨2⟩22
 DEF *Next*
 ⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvUnforgeableSemHappy* \triangleq
Spec \Rightarrow \square *InvUnforgeableSemHappy*

PROOF

⟨1⟩ *Init* \Rightarrow *InvUnforgeableSemHappy* BY *ThmInitInvUnforgeableSemHappy*
 ⟨1⟩ *InvUnforgeableSemHappy* \wedge $[Next]_{vars} \Rightarrow$ *InvUnforgeableSemHappy'*
 BY *ThmNextInvUnforgeableSemHappy*
 ⟨1⟩ USE DEF *Spec*
 ⟨1⟩ QED

PROOF OF INVARIANT $InvProperLastExtension$

It holds in the initial state.

THEOREM $ThmInitInvProperLastExtension \triangleq$
 $Init \Rightarrow InvProperLastExtension$

PROOF

- (1) HAVE $Init$
- (1) $InvType$ BY $ThmInitInvType$
- (1) QED BY DEF $InvProperLastExtension, Init$

If it holds in the current state, and we perform a $Next$ action, then it will hold in the next state.

THEOREM $ThmNextInvProperLastExtension \triangleq$
 $InvProperLastExtension \wedge [Next]_{vars} \Rightarrow InvProperLastExtension'$

PROOF

- (1) HAVE $InvProperLastExtension \wedge [Next]_{vars}$
- (1) USE DEF $InvProperLastExtension$
- (1) $InvType'$ BY $ThmNextInvType$
- (1) $InvProperLastExtension' ! goal'$

Wow, this is an easy one.

- (2) USE DEF $vars$
- (2) USE DEF $NextObtainAccess$
- (2) USE DEF $NextProveRevoke$
- (2) USE DEF $NextReboot$
- (2) USE DEF $NextForgetSealTs$
- (2) USE DEF $NextExtendAppPcr$
- (2) USE DEF $NextExtendSemPcr$
- (2) USE DEF $NextIncBootCtr$
- (2) USE DEF $NextEnterSemRecov$
- (2) USE DEF $NextSemRecov1 WhenCorrect$
- (2) USE DEF $NextSemRecov1 WhenIncorrect$
- (2) USE DEF $NextSemRecov2$
- (2) USE DEF $NextSemRecov3$
- (2) USE DEF $NextSealTs$
- (2) USE DEF $NextEnterSemChkpt$
- (2) USE DEF $NextSemChkpt1 WhenCorrect$
- (2) USE DEF $NextSemChkpt1 WhenIncorrect$
- (2) USE DEF $NextSemChkpt2$
- (2) USE DEF $NextSemChkpt3$
- (2) USE DEF $NextSemChkpt4$
- (2) USE DEF $NextSemChkpt5$

- ⟨2⟩ QED BY DEF *Next*
- ⟨1⟩ QED OBVIOUS

It is an invariant of the specification.

THEOREM *ThmInvProperLastExtension* \triangleq
 $Spec \Rightarrow \Box InvProperLastExtension$

PROOF

- ⟨1⟩ *Init* \Rightarrow *InvProperLastExtension* BY *ThmInitInvProperLastExtension*
- ⟨1⟩ *InvProperLastExtension* $\wedge [Next]_{vars} \Rightarrow$ *InvProperLastExtension'*
 BY *ThmNextInvProperLastExtension*
- ⟨1⟩ USE DEF *Spec*
- ⟨1⟩ QED

PROOF OF INVARIANT *InvOneLog*

It holds in the initial state.

THEOREM *ThmInitInvOneLog* \triangleq
 $Init \Rightarrow InvOneLog$

PROOF

- ⟨1⟩ HAVE *Init*
- ⟨1⟩ *InvType* BY *ThmInitInvType*
- ⟨1⟩ *InvSignedTsLeqBoot* BY *ThmInitInvSignedTsLeqBoot*
- ⟨1⟩ *InvInSemProtect* BY *ThmInitInvInSemProtect*
- ⟨1⟩ *InvUnforgeableSemHappy* BY *ThmInitInvUnforgeableSemHappy*
- ⟨1⟩ *InvProperLastExtension* BY *ThmInitInvProperLastExtension*
- ⟨1⟩ USE DEF *Init*
- ⟨1⟩ USE DEF *InitNv*
- ⟨1⟩ USE DEF *PcIDLE*
- ⟨1⟩ USE DEF *PcRECOV1, PcRECOV2, PcRECOV3*
- ⟨1⟩ USE DEF *PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5*
- ⟨1⟩ *LogInNv* BY DEF *LogInNv*
- ⟨1⟩ $\neg LogInApp$
- ⟨2⟩ *semPcr* \neq *SemHappy*
- ⟨3⟩ *semPcr.init* \neq *SemHappy.init*

⟨4⟩ $semPcr.init \neq SemProtect.init$
 ⟨5⟩ USE DEF *SemReboot*
 ⟨5⟩ USE DEF *SemProtect*
 ⟨5⟩ USE DEF *PcrInit*
 ⟨5⟩ QED BY *AssSemProtect*
 ⟨4⟩ USE DEF *SemHappy*
 ⟨4⟩ QED BY DEF *PcrExtend*
 ⟨3⟩ QED BY DEF *Pcr*
 ⟨2⟩ QED BY DEF *LogInApp*
 (1) $\neg LogInTs \wedge AllCurrentTs = \{\}$
 ⟨2⟩ $\neg CheckTsIsCurrent(chkptts)$
 ⟨3⟩ USE DEF *CheckTsIsCurrent*
 ⟨3⟩ QED BY *ThmNullTsIsntSignedTs*
 ⟨2⟩ USE DEF *AllCurrentTs*
 ⟨2⟩ QED BY DEF *LogInTs*
 (1) *InvOneLog!goal!obtains* BY DEF *IsOnLog*
 (1) *InvOneLog!goal!revokes* BY DEF *IsOnLog*
 (1) *InvVerifiableRevocation* BY DEF *InvVerifiableRevocation*
 (1) QED BY DEF *InvOneLog*

If it holds in the current state, and we perform a *Next* action, then it will hold in the next state.

THEOREM *ThmNextInvOneLog* \triangleq
InvOneLog $\wedge [Next]_{vars} \Rightarrow InvOneLog'$

PROOF

(1) HAVE *InvOneLog* $\wedge [Next]_{vars}$
 (1) *InvType* BY DEF *InvOneLog*
 (1) *InvSignedTsLeqBoot* BY DEF *InvOneLog*
 (1) *InvInSemProtect* BY DEF *InvOneLog*
 (1) *InvUnforgeableSemHappy* BY DEF *InvOneLog*
 (1) *InvProperLastExtension* BY DEF *InvOneLog*
 (1) *InvType'* BY *ThmNextInvType*
 (1) *InvSignedTsLeqBoot'* BY *ThmNextInvSignedTsLeqBoot*
 (1) *InvInSemProtect'* BY *ThmNextInvInSemProtect*
 (1) *InvUnforgeableSemHappy'* BY *ThmNextInvUnforgeableSemHappy*
 (1) *InvProperLastExtension'* BY *ThmNextInvProperLastExtension*
 (1) *InvOneLog!goal'*
 ⟨2⟩ USE DEF *PcIDLE*
 ⟨2⟩ USE DEF *PcRECOV1*, *PcRECOV2*, *PcRECOV3*
 ⟨2⟩ USE DEF *PcCHKPT1*, *PcCHKPT2*, *PcCHKPT3*, *PcCHKPT4*, *PcCHKPT5*

Stutter step.

(2) 1. CASE $vars' = vars$
 (3) USE (2)1
 (3) USE DEF $vars$
 (3) UNCHANGED $CheckTsIsCurrent(chkpts)$ BY DEF $CheckTsIsCurrent$
 (3) UNCHANGED $AllCurrentTs$
 (4) USE DEF $AllCurrentTs$
 (4) USE DEF $CheckTsIsCurrent$
 (4) QED BY DEF $LogInTs$
 (3) UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
 (3) UNCHANGED $LogInNv$ BY DEF $LogInNv$
 (3) UNCHANGED $LogInApp$ BY DEF $LogInApp$
 (3) UNCHANGED $LogInTs$ BY DEF $LogInTs$
 (3) $InvOneLog!goal!obtains'$ BY DEF $IsOnLog, InvOneLog$
 (3) $InvOneLog!goal!revokes'$ BY DEF $IsOnLog, InvOneLog$
 (3) $InvVerifiableRevocation'$ BY DEF $InvVerifiableRevocation, InvOneLog$
 (3) QED BY DEF $InvOneLog$

NextObtainAccess or *NextProveRevoke*

(2) 2. CASE $NextObtainAccess \vee NextProveRevoke$
 (3) USE (2)2
 (3) USE DEF $NextObtainAccess$
 (3) USE DEF $NextProveRevoke$

 (3) UNCHANGED $CheckTsIsCurrent(chkpts)$ BY DEF $CheckTsIsCurrent$
 (3) UNCHANGED $AllCurrentTs$
 (4) USE DEF $AllCurrentTs$
 (4) USE DEF $CheckTsIsCurrent$
 (4) QED BY DEF $LogInTs$
 (3) UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
 (3) UNCHANGED $LogInNv$ BY DEF $LogInNv$
 (3) UNCHANGED $LogInApp$ BY DEF $LogInApp$
 (3) UNCHANGED $LogInTs$ BY DEF $LogInTs$
 (3) $LogInApp$
 (4) CASE $NextObtainAccess$
 $NextObtainAccess$ is predicated on the fact that the log is in the application pcr.
 This depends on
 $\wedge BugObtainAccessNoCheckHappy \triangleq FALSE$
 (5) $BugObtainAccessNoCheckHappy = FALSE$ BY DEF $BugObtainAccessNoCheckHappy$
 (5) QED BY DEF $LogInApp$
 (4) CASE $NextProveRevoke$
 $NextProveRevoke$ is predicated on the fact that the log is in the application pcr.
 This depends on
 $\wedge BugProveRevokeNoCheckHappy \triangleq FALSE$

⟨5⟩ *BugProveRevokeNoCheckHappy* = FALSEBY DEF *BugProveRevokeNoCheckHappy*
 ⟨5⟩ QED BY DEF *LogInApp*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ *InvOneLog!goal!obtains' ∧ InvOneLog!goal!revokes'*
 Since the log is in the application pcr, putting a copy of the application pcr into obtains or into revokes preserves the invariant that everything in *obtains ∪ revokes* can reach the log.
 ⟨4⟩ *PcrLeq(appPcr, appPcr)*BY *ThmPcrLeqIsReflexive*DEF *InvType*
 ⟨4⟩ QED BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'*
 Since we only add an element to *obtains ∪ revokes* when the log is in the application pcr, we know that all elements in *obtains ∪ revokes* in the new state must be on the log, which we can check as \leq *app pcr*.
 So we proceed with proof by contradiction. Assuming that verifiable deletion will be violated in the new state, we pick the $o \in$ *obtains* and $r \in$ *revokes* whose *PcrPrior*'s are the same. But since both o and r must be \leq *app pcr*, this means that their last extension must be the same. This contradicts the assumption that OBTAIN is different from REVOKE.
 ⟨4⟩ CASE *InvVerifiableRevocation'* OBVIOUS
 ⟨4⟩ CASE \neg *InvVerifiableRevocation'*
 ⟨5⟩ PICK $o \in$ *obtains'*, $r \in$ *revokes'* : *PcrPrior*(o) = *PcrPrior*(r)
 BY DEF *InvVerifiableRevocation*
 ⟨5⟩ DEFINE $p \triangleq$ *PcrPrior*(o)
 ⟨5⟩ DEFINE $xo \triangleq$ *PcrLastExtension*(o)
 ⟨5⟩ DEFINE $xr \triangleq$ *PcrLastExtension*(r)
 ⟨5⟩ $p \in$ *Pcr*BY *ThmPcrPriorIsPcr*DEF *InvType, InvProperLastExtension*
 ⟨5⟩ $xo =$ *Pcrx*OBTAINBY DEF *InvProperLastExtension*
 ⟨5⟩ $xr =$ *Pcrx*REVOKEBY DEF *InvProperLastExtension*
 ⟨5⟩ $o =$ *PcrExtend*(p, xo)BY *ThmPcrExtendPriorLast*DEF *InvType, InvProperLastExtension*
 ⟨5⟩ $r =$ *PcrExtend*(p, xr)BY *ThmPcrExtendPriorLast*DEF *InvType, InvProperLastExtension*
 ⟨5⟩ *PcrLeq*($o, appPcr'$)BY DEF *IsOnLog, InvOneLog*
 ⟨5⟩ *PcrLeq*($r, appPcr'$)BY DEF *IsOnLog, InvOneLog*
 ⟨5⟩ $xo \in$ *Pcrx*BY DEF *Pcrx*
 ⟨5⟩ $xr \in$ *Pcrx*BY DEF *Pcrx*
 ⟨5⟩ $xo = xr$
 The prover needs a lot of help to focus its attention.
 ⟨6⟩ HIDE DEF p
 ⟨6⟩ HIDE DEF xo
 ⟨6⟩ HIDE DEF xr
 ⟨6⟩ $appPcr' \in$ *Pcr*BY DEF *InvType*
 ⟨6⟩ QED BY *ThmPcrExtendLeqAnticollision*
 ⟨5⟩ *Pcrx*OBTAIN \neq *Pcrx*REVOKEBY *AssObtainNeqRevoke*
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ QED BY DEF *InvOneLog*

(2)3. CASE *NextReboot*

⟨3⟩ USE *NextReboot*

⟨3⟩ USE DEF *NextReboot*

⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, which erases any log that had been in the application pcr.

⟨3⟩ \neg *LogInApp'*

⟨4⟩ $semPcr' \neq SemHappy$

⟨5⟩ $semPcr'.init \neq SemHappy.init$

⟨6⟩ $semPcr'.init \neq SemProtect.init$

⟨7⟩ USE DEF *SemReboot*

⟨7⟩ USE DEF *SemProtect*

⟨7⟩ USE DEF *PcrInit*

⟨7⟩ QED BY *AssSemProtect*

⟨6⟩ USE DEF *SemHappy*

⟨6⟩ QED BY DEF *PcrExtend*

⟨5⟩ QED BY DEF *Pcr*

⟨4⟩ QED BY DEF *LogInApp*

Overwrites *chkptts* with an unsigned *ts*, which might erase a log that had been in the seal attestations.

⟨3⟩ $LogInTs' \Rightarrow LogInTs$

⟨4⟩ HAVE *LogInTs'*

⟨4⟩ $AllCurrentTs' \neq \{\}$ BY DEF *LogInTs*

⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ $\neg CheckTsIsCurrent(chkptts)'$ BY *ThmNullTsIsntSignedTs*

⟨4⟩ $\exists ts \in tsvalues' : CheckTsIsCurrent(ts)$ OBVIOUS

⟨4⟩ QED BY DEF *LogInTs*

Any remaining current seal attestations existed previously, so they must contain the same log.

⟨3⟩ $\forall ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$

⟨4⟩ TAKE $ts1, ts2 \in AllCurrentTs'$

⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ $\neg CheckTsIsCurrent(chkptts)'$ BY *ThmNullTsIsntSignedTs*

⟨4⟩ $ts1 \in AllCurrentTs$ OBVIOUS

⟨4⟩ $ts2 \in AllCurrentTs$ OBVIOUS

⟨4⟩ QED BY DEF *InvOneLog*

If there are any remaining current seal attestations, the log in them has to be the same as before.

⟨3⟩ $LogInTs' \Rightarrow$ UNCHANGED *CurrentTsLog*

⟨4⟩ HAVE *LogInTs'*

⟨4⟩ $\exists ts \in AllCurrentTs' : ts \in AllCurrentTs$

⟨5⟩ USE DEF *LogInTs*

⟨5⟩ USE DEF *AllCurrentTs*

⟨5⟩ USE DEF *CheckTsIsCurrent*

⟨5⟩ $\neg CheckTsIsCurrent(chkptts)'$ BY *ThmNullTsIsntSignedTs*

⟨5⟩ $\forall ts \in AllCurrentTs' : ts \in AllCurrentTs$ OBVIOUS

⟨5⟩ QED OBVIOUS
 ⟨4⟩ $\forall ts1 \in AllCurrentTs'$:
 $\forall ts0 \in AllCurrentTs$:
 $ts1.appPcr = ts0.appPcr$
 BY DEF *InvOneLog*
 ⟨4⟩ QED BY DEF *CurrentTsLog*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextForgetSealTs

⟨2⟩4. CASE *NextForgetSealTs*
 ⟨3⟩ USE *NextForgetSealTs*
 ⟨3⟩ USE DEF *NextForgetSealTs*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*

Forgets a seal attestation, which might erase a log that had been in the seal attestations.

⟨3⟩ $LogInTs' \Rightarrow LogInTs$
 ⟨4⟩ HAVE *LogInTs'*
 ⟨4⟩ $AllCurrentTs' \neq \{\}$ BY DEF *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*

Any remaining current seal attestations existed previously, so they must contain the same log.

⟨3⟩ $\forall ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$
 ⟨4⟩ TAKE $ts1, ts2 \in AllCurrentTs'$
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ $ts1 \in AllCurrentTs$ OBVIOUS
 ⟨4⟩ $ts2 \in AllCurrentTs$ OBVIOUS
 ⟨4⟩ QED BY DEF *InvOneLog*

If *chkptts* contains a current seal attestation, then the log is in the seal attestations.

⟨3⟩ $CheckTsIsCurrent(chkptts)' \Rightarrow LogInTs'$
 ⟨4⟩ USE DEF *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED OBVIOUS

If there are any remaining current seal attestations, the log in them has to be the same as before.

⟨3⟩ $LogInTs' \Rightarrow$ UNCHANGED *CurrentTsLog*
 ⟨4⟩ HAVE *LogInTs'*
 ⟨4⟩ $\exists ts \in AllCurrentTs' : ts \in AllCurrentTs$

⟨5⟩ USE DEF *LogInTs*
 ⟨5⟩ USE DEF *AllCurrentTs*
 ⟨5⟩ USE DEF *CheckTsIsCurrent*
 ⟨5⟩ $\forall ts \in AllCurrentTs' : ts \in AllCurrentTs$ OBVIOUS
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ $\forall ts1 \in AllCurrentTs' :$
 $\forall ts0 \in AllCurrentTs :$
 $ts1.appPcr = ts0.appPcr$
 BY DEF *InvOneLog*
 ⟨4⟩ QED BY DEF *CurrentTsLog*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextExtendAppPcr

⟨2⟩5. CASE *NextExtendAppPcr*
 ⟨3⟩ USE *NextExtendAppPcr*
 ⟨3⟩ USE DEF *NextExtendAppPcr*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkpts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!obtains' \wedge InvOneLog!goal!revokes'*
 ⟨4⟩ CASE $\neg LogInApp$
 ⟨5⟩ USE DEF *LogInApp*
 ⟨5⟩ USE DEF *IsOnLog, InvOneLog*

NextExtendAppPcr is predicated on not being in sem, so none of the sem clauses apply.

⟨5⟩ USE DEF *InSem*
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ CASE *LogInApp*
 ⟨5⟩ USE DEF *LogInApp*

If the log is in the application pcr, extending the application pcr preserves the fact that all entries in *obtains* \cup *revokes* can reach it.

⟨5⟩ $\forall p \in \text{obtains} \cup \text{revokes} : \text{LogInApp} \Rightarrow \text{PcrLeq}(p, \text{appPcr}')$
 ⟨6⟩ UNCHANGED (*obtains* \cup *revokes*)OBVIOUS
 ⟨6⟩ TAKE $p \in \text{obtains} \cup \text{revokes}$
 ⟨6⟩ HAVE *LogInApp*
 ⟨6⟩ $\text{PcrLeq}(p, \text{appPcr})$ BY DEF *IsOnLog*, *InvOneLog*
 ⟨6⟩ $\text{PcrLeq}(\text{appPcr}, \text{appPcr}')$ BY *ThmPcrExtendLeq*DEF *InvType*
 ⟨6⟩ QED BY *ThmPcrLeqIsTransitive*DEF *InvType*
 ⟨5⟩ QED BY DEF *IsOnLog*, *InvOneLog*
 ⟨4⟩ QED OBVIOUS
 ⟨3⟩ QED BY DEF *InvOneLog*

NextExtendSemPcr

⟨2⟩6. CASE *NextExtendSemPcr*
 ⟨3⟩ USE *NextExtendSemPcr*
 ⟨3⟩ USE DEF *NextExtendSemPcr*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent*(*chkptts*)BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, which erases any log that had been in the application pcr.

⟨3⟩ $\neg \text{LogInApp}'$
 ⟨4⟩ $\text{semPcr}' \neq \text{SemHappy}$
 ⟨5⟩ $\text{semPcr} = \text{SemHappy} \vee \neg \text{PcrLeq}(\text{semPcr}, \text{SemHappy})$
 BY DEF *InvUnforgeableSemHappy*
 ⟨5⟩ $\text{semPcr} \in \text{Pcr}$ BY DEF *InvType*
 ⟨5⟩ $\text{SemHappy} \in \text{Pcr}$ BY *ThmSemHappyIsPcr*
 ⟨5⟩ $\neg \text{PcrLeq}(\text{semPcr}', \text{SemHappy})$ BY *ThmPcrExtendFromEqOrNotleq*
 ⟨5⟩ QED BY *ThmPcrLeqIsReflexive*
 ⟨4⟩ QED BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ $\text{InvOneLog}! \text{goal}! \text{obtains}'$ BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ $\text{InvOneLog}! \text{goal}! \text{revokes}'$ BY DEF *IsOnLog*, *InvOneLog*
 ⟨3⟩ $\text{InvVerifiableRevocation}'$ BY DEF *InvVerifiableRevocation*, *InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextIncBootCtr

(2)8. CASE *NextIncBootCtr*

- ⟨3⟩ USE *NextIncBootCtr*
- ⟨3⟩ USE DEF *NextIncBootCtr*
- ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
- ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*

Since no signed *ts* seal can have a *bootCtr* greater than the current *bootCtr*, incrementing *bootCtr* erases any log that had been in a seal attestation.

- ⟨3⟩ $\neg \text{LogInTs}'$
- ⟨4⟩ USE DEF *AllCurrentTs*
- ⟨4⟩ USE DEF *CheckTsIsCurrent*
- ⟨4⟩ $\forall ts \in \text{tsvalues}' \cup \{\text{chkptts}'\} : ts \in \text{SignedTs} \Rightarrow ts.\text{bootCtr} \neq \text{bootCtr}'$
- ⟨5⟩ TAKE $ts \in \text{tsvalues}' \cup \{\text{chkptts}'\}$
- ⟨5⟩ $ts \in \text{tsvalues} \cup \{\text{chkptts}\}$ OBVIOUS
- ⟨5⟩ HAVE $ts \in \text{SignedTs}$
- ⟨5⟩ $ts.\text{bootCtr} \leq \text{bootCtr}$ BY DEF *InvSignedTsLeqBoot*
- ⟨5⟩ $ts.\text{bootCtr} \in \text{Nat}$ BY DEF *SignedTs*
- ⟨5⟩ $\text{bootCtr} \in \text{Nat}$ BY DEF *InvType*
- ⟨5⟩ $\text{bootCtr}' \in \text{Nat}$ BY DEF *InvType*
- ⟨5⟩ $\text{bootCtr} < \text{bootCtr}'$ BY *ThmNatInc*
- ⟨5⟩ $ts.\text{bootCtr} < \text{bootCtr}'$ BY *ThmNatLeqLt*
- ⟨5⟩ QED BY *ThmNatLeqXorGt*, *ThmNatLeqIsReflexive*
- ⟨4⟩ QED BY DEF *LogInTs*

Erases all current *ts* seal attestations.

- ⟨3⟩ $\text{AllCurrentTs}' = \{\} \wedge \neg \text{CheckTsIsCurrent}(\text{chkptts})'$
- ⟨4⟩ USE DEF *AllCurrentTs*
- ⟨4⟩ USE DEF *CheckTsIsCurrent*
- ⟨4⟩ QED BY DEF *LogInTs*
- ⟨3⟩ USE DEF *InSem*
- ⟨3⟩ *InvOneLog!*goal!*obtains'* BY DEF *IsOnLog*, *InvOneLog*
- ⟨3⟩ *InvOneLog!*goal!*revokes'* BY DEF *IsOnLog*, *InvOneLog*
- ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation*, *InvOneLog*
- ⟨3⟩ QED BY DEF *InvOneLog*

NextEnterSemRecov

(2)9. CASE *NextEnterSemRecov*

- ⟨3⟩ USE *NextEnterSemRecov*
- ⟨3⟩ USE DEF *NextEnterSemRecov*
- ⟨3⟩ USE DEF *InSem*
- ⟨3⟩ UNCHANGED *CheckTsIsCurrent*(*chkptts*) BY DEF *CheckTsIsCurrent*
- ⟨3⟩ UNCHANGED *AllCurrentTs*
- ⟨4⟩ USE DEF *AllCurrentTs*
- ⟨4⟩ USE DEF *CheckTsIsCurrent*
- ⟨4⟩ QED BY DEF *LogInTs*
- ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, which erases any log that had been in the application pcr.

⟨3⟩ $\neg \text{LogInApp}'$

⟨4⟩ $\text{semPcr}' \neq \text{SemHappy}$

⟨5⟩ USE DEF *SemHappy*

⟨5⟩ USE DEF *SemProtect*

⟨5⟩ USE DEF *Pcri*

⟨5⟩ USE DEF *Pcrr*

⟨5⟩ USE *ThmPcrInitIsPcr*

⟨5⟩ USE *ThmPcrExtendIsPcr*

⟨5⟩ *PcrLeq*(*SemProtect*, *SemHappy*)BY *ThmPcrExtendLeq*

⟨5⟩ $\neg \text{PcrLeq}$ (*SemHappy*, *SemProtect*)BY *ThmPcrExtendSelfUnreachable*

⟨5⟩ QED BY *ThmPcrLeqIsAntisymmetric*

⟨4⟩ QED BY DEF *LogInApp*

⟨3⟩ UNCHANGED *LogInTs*

⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ QED BY DEF *LogInTs*

⟨3⟩ *InvOneLog!*goal!*obtains'* BY DEF *IsOnLog*, *InvOneLog*

⟨3⟩ *InvOneLog!*goal!*revokes'* BY DEF *IsOnLog*, *InvOneLog*

⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation*, *InvOneLog*

⟨3⟩ QED BY DEF *InvOneLog*

NextSemRecov1WhenCorrect

⟨2⟩ 10. CASE *NextSemRecov1WhenCorrect*

⟨3⟩ USE *NextSemRecov1WhenCorrect*

⟨3⟩ USE DEF *NextSemRecov1WhenCorrect*

⟨3⟩ UNCHANGED *CheckTsIsCurrent*(*chkptts*)BY DEF *CheckTsIsCurrent*

⟨3⟩ UNCHANGED *AllCurrentTs*

⟨4⟩ USE DEF *AllCurrentTs*

⟨4⟩ USE DEF *CheckTsIsCurrent*

⟨4⟩ QED BY DEF *LogInTs*

⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*

⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

EnterSemRecovPredicate guarantees that the log is in the *nv* ram.

This depends on *BugRecovNoCheckCur* \triangleq FALSE

⟨3⟩ *LogInNv*

⟨4⟩ USE DEF *EnterSemRecovPredicate*

⟨4⟩ *BugRecovNoCheckCur* = FALSEBY DEF *BugRecovNoCheckCur*

⟨4⟩ QED BY DEF *LogInNv*

EnterSemRecovPredicate guarantees that the application *pcr* equals the log saved in the *nv* ram.

This depends on *BugRecovNoCheckApp* \triangleq FALSE

- ⟨3⟩ *appPcr* = *nv.appPcr*
- ⟨4⟩ USE DEF *EnterSemRecovPredicate*
- ⟨4⟩ *BugRecovNoCheckApp* = FALSE BY DEF *BugRecovNoCheckApp*
- ⟨4⟩ QED OBVIOUS
- ⟨3⟩ *InvOneLog!**goal!**obtains'* BY DEF *IsOnLog*, *InvOneLog*
- ⟨3⟩ *InvOneLog!**goal!**revokes'* BY DEF *IsOnLog*, *InvOneLog*
- ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation*, *InvOneLog*
- ⟨3⟩ QED BY DEF *InvOneLog*

NextSemRecov1WhenIncorrect

- ⟨2⟩ 11. CASE *NextSemRecov1WhenIncorrect*
- ⟨3⟩ USE *NextSemRecov1WhenIncorrect*
- ⟨3⟩ USE DEF *NextSemRecov1WhenIncorrect*
- ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
- ⟨3⟩ UNCHANGED *AllCurrentTs*
- ⟨4⟩ USE DEF *AllCurrentTs*
- ⟨4⟩ USE DEF *CheckTsIsCurrent*
- ⟨4⟩ QED BY DEF *LogInTs*
- ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
- ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending *sem pcr* with *Unhappy* results in something other than *SemHappy*, which indicates that the log is not in the application *pcr*.

- ⟨3⟩ \neg *LogInApp'*
- ⟨4⟩ *semPcr' ≠ SemHappy*
- ⟨5⟩ USE DEF *SemHappy*
- ⟨5⟩ USE DEF *SemProtect*
- ⟨5⟩ USE DEF *Pcri*
- ⟨5⟩ USE DEF *Pcrx*
- ⟨5⟩ USE *ThmPcrInitIsPcr*
- ⟨5⟩ USE *ThmPcrExtendIsPcr*
- ⟨5⟩ *semPcr* = *SemProtect* BY DEF *InvInSemProtect*, *InSem*
- ⟨5⟩ USE *AssSemHappy*
- ⟨5⟩ QED BY *ThmPcrExtendAnticollision*
- ⟨4⟩ QED BY DEF *LogInApp*
- ⟨3⟩ UNCHANGED *LogInTs*
- ⟨4⟩ USE DEF *AllCurrentTs*
- ⟨4⟩ USE DEF *CheckTsIsCurrent*
- ⟨4⟩ QED BY DEF *LogInTs*
- ⟨3⟩ *InvOneLog!**goal!**obtains'* BY DEF *IsOnLog*, *InvOneLog*
- ⟨3⟩ *InvOneLog!**goal!**revokes'* BY DEF *IsOnLog*, *InvOneLog*
- ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation*, *InvOneLog*
- ⟨3⟩ QED BY DEF *InvOneLog*

NextSemRecov2

- ⟨2⟩ 12. CASE *NextSemRecov2*
- ⟨3⟩ USE *NextSemRecov2*
- ⟨3⟩ USE DEF *NextSemRecov2*
- ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
- ⟨3⟩ UNCHANGED *AllCurrentTs*
- ⟨4⟩ USE DEF *AllCurrentTs*
- ⟨4⟩ USE DEF *CheckTsIsCurrent*
- ⟨4⟩ QED BY DEF *LogInTs*
- ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

Clearing *nv* current erases the log from the *nv* ram.

This depends on *BugRecovNoClrCur* \triangleq FALSE

- ⟨3⟩ \neg *LogInNv'*
- ⟨4⟩ \neg *nv'.current*
- ⟨5⟩ *BugRecovNoClrCur* = FALSE BY DEF *BugRecovNoClrCur*
- ⟨5⟩ QED BY DEF *InvType, Nv*
- ⟨4⟩ QED BY DEF *LogInNv*
- ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
- ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*
- ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
- ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
- ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
- ⟨3⟩ QED BY DEF *InvOneLog*

NextSemRecov3

- ⟨2⟩ 13. CASE *NextSemRecov3*
- ⟨3⟩ USE *NextSemRecov3*
- ⟨3⟩ USE DEF *NextSemRecov3*
- ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
- ⟨3⟩ UNCHANGED *AllCurrentTs*
- ⟨4⟩ USE DEF *AllCurrentTs*
- ⟨4⟩ USE DEF *CheckTsIsCurrent*
- ⟨4⟩ QED BY DEF *LogInTs*
- ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
- ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending sem per to *SemHappy* puts the log in the application per, provided that the seal per contains *SealReboot*.

But in the current state the log has no domicile. So the fact that its domicile might be the application per in the next state does not require a proof that it is not living anywhere else, since we get that for free.

- ⟨3⟩ *LogInApp' ∈ BOOLEAN* BY DEF *LogInApp*
- ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*
- ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
- ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
- ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*

⟨3⟩ QED BY DEF *InvOneLog*

NextSealTs

⟨2⟩ 14. CASE *NextSealTs*

⟨3⟩ USE *NextSealTs*

⟨3⟩ USE DEF *NextSealTs*

⟨3⟩ UNCHANGED *CheckTsIsCurrent*(*chkptts*) BY DEF *CheckTsIsCurrent*

⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Cancels *SemHappy* if we had it, which erases any log that had been in the application pcr.

This depends on *BugSealNoExt* $\stackrel{\Delta}{=} \text{FALSE}$

⟨3⟩ $\neg \text{LogInApp}'$

OPT: REWORKED PROOF HERE TO DEAL WITH EXTENSION OF SEM PCR

⟨4⟩ $\text{semPcr}' \neq \text{SemHappy}$

⟨5⟩ $\text{semPcr} = \text{SemHappy} \vee \neg \text{PcrLeq}(\text{semPcr}, \text{SemHappy})$

BY DEF *InvUnforgeableSemHappy*

⟨5⟩ $\text{semPcr} \in \text{Pcr}$ BY DEF *InvType*

⟨5⟩ $\text{SemHappy} \in \text{Pcr}$ BY *ThmSemHappyIsPcr*

⟨5⟩ $\neg \text{PcrLeq}(\text{semPcr}', \text{SemHappy})$

⟨6⟩ $\text{semPcr}' = \text{PcrExtend}(\text{semPcr}, \text{PcrxSEAL})$

⟨7⟩ $\text{BugSealNoExt} = \text{FALSE}$ BY DEF *BugSealNoExt*

⟨7⟩ QED OBVIOUS

⟨6⟩ QED BY *ThmPcrExtendFromEqOrNotleq* DEF *Pcrx*

⟨5⟩ QED BY *ThmPcrLeqIsReflexive*

⟨4⟩ QED BY DEF *LogInApp*

⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation*, *InvOneLog*

If the log was not in the application pcr, the resulting seal attestation will not be valid, so there is no change in *AllCurrentTs* or *LogSummaryInTs*.

⟨3⟩ CASE $\neg \text{LogInApp}$

⟨4⟩ UNCHANGED *AllCurrentTs*

⟨5⟩ USE DEF *AllCurrentTs*

⟨5⟩ USE DEF *CheckTsIsCurrent*

Use PICK to make *ts* a CONSTANT so that $\neg \text{CheckTsIsCurrent}(ts)'$ means the *ts* picked now evaluated with *CheckTsIsCurrent* in the next state.

⟨5⟩ PICK $ts \in \text{SignedTs} : ts = \text{NextSealTs}! : !ts$ BY DEF *InvType*, *SignedTs*

⟨5⟩ $\forall ts1 \in \text{tsvalues}' : \text{CheckTsIsCurrent}(ts1)' \Rightarrow ts1 \in \text{tsvalues}$

⟨6⟩ $\text{tsvalues}' = \text{tsvalues} \cup \{ts\}$ OBVIOUS

⟨6⟩ $\neg \text{CheckTsIsCurrent}(ts)'$ BY DEF *LogInApp*

⟨6⟩ QED OBVIOUS

⟨5⟩ QED OBVIOUS

⟨4⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

⟨4⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

⟨4⟩ $\text{InvOneLog}! \text{goal}! \text{obtains}'$ BY DEF *IsOnLog*, *InvOneLog*

- ⟨4⟩ $InvOneLog!goal!revokes'$ BY DEF $IsOnLog, InvOneLog$
 ⟨4⟩ QED BY DEF $InvOneLog$

If the log was in the application pcr, the resulting seal attestation will be valid. But then the old $AllCurrentTs$ had to be empty, since the log could not have been in the seal attestations.

⟨3⟩ CASE $LogInApp$

Use PICK to make ts a CONSTANT so that $CheckTsIsCurrent(ts)'$ means the ts picked now evaluated with $CheckTsIsCurrent$ in the next state.

- ⟨4⟩ PICK $ts \in SignedTs : ts = NextSealTs! : !ts$ BY DEF $InvType, SignedTs$
 ⟨4⟩ $CheckTsIsCurrent(ts)'$
 ⟨5⟩ $CheckTsIsCurrent(ts)$ BY DEF $CheckTsIsCurrent, LogInApp$
 ⟨5⟩ QED BY DEF $CheckTsIsCurrent$
 ⟨4⟩ $AllCurrentTs' = \{ts\}$
 ⟨5⟩ $\forall ts1 \in tvalues \cup \{chkptts'\} : \neg CheckTsIsCurrent(ts1)'$
 ⟨6⟩ $AllCurrentTs = \{\}$
 ⟨7⟩ $\neg LogInTs$ BY DEF $InvOneLog$
 ⟨7⟩ QED BY DEF $LogInTs$
 ⟨6⟩ $\neg CheckTsIsCurrent(chkptts)'$
 ⟨7⟩ $\neg CheckTsIsCurrent(chkptts)$ BY DEF $AllCurrentTs$
 ⟨7⟩ QED BY DEF $CheckTsIsCurrent$
 ⟨6⟩ $\forall ts1 \in tvalues : \neg CheckTsIsCurrent(ts1)'$
 ⟨7⟩ $\forall ts1 \in tvalues : \neg CheckTsIsCurrent(ts1)$ BY DEF $AllCurrentTs$
 ⟨7⟩ QED BY DEF $CheckTsIsCurrent$
 ⟨6⟩ QED BY DEF $CheckTsIsCurrent$
 ⟨5⟩ $tvalues' = tvalues \cup \{ts\}$ OBVIOUS
 ⟨5⟩ $ts \in AllCurrentTs'$ BY DEF $AllCurrentTs$
 ⟨5⟩ QED BY DEF $AllCurrentTs$
 ⟨4⟩ $\forall ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$ OBVIOUS
 ⟨4⟩ $LogInTs'$ BY DEF $LogInTs$
 ⟨4⟩ $CurrentTsLog' = ts.appPcr$ BY DEF $CurrentTsLog$
 ⟨4⟩ $InvOneLog!goal!obtains'$ BY DEF $IsOnLog, InvOneLog$
 ⟨4⟩ $InvOneLog!goal!revokes'$ BY DEF $IsOnLog, InvOneLog$
 ⟨4⟩ QED BY DEF $InvOneLog$
 ⟨3⟩ QED OBVIOUS

NextEnterSemChkpt

⟨2⟩ 15. CASE $NextEnterSemChkpt$

- ⟨3⟩ USE $NextEnterSemChkpt$
 ⟨3⟩ USE DEF $NextEnterSemChkpt$
 ⟨3⟩ USE DEF $InSem$
 ⟨3⟩ UNCHANGED $LogInNv$ BY DEF $LogInNv$

Cancels $SemHappy$ if we had it, so erases any log that might have been in the application pcr.

- ⟨3⟩ $\neg LogInApp'$
 ⟨4⟩ $semPcr' \neq SemHappy$
 ⟨5⟩ USE DEF $SemHappy$

⟨5⟩ USE DEF *SemProtect*
 ⟨5⟩ USE DEF *Pcri*
 ⟨5⟩ USE DEF *Pcrx*
 ⟨5⟩ USE *ThmPcrInitIsPcr*
 ⟨5⟩ USE *ThmPcrExtendIsPcr*
 ⟨5⟩ *PcrLeq(SemProtect, SemHappy)*BY *ThmPcrExtendLeq*
 ⟨5⟩ \neg *PcrLeq(SemHappy, SemProtect)*BY *ThmPcrExtendSelfUnreachable*
 ⟨5⟩ QED BY *ThmPcrLeqIsAntisymmetric*
 ⟨4⟩ QED BY DEF *LogInApp*

Overwrites *chkptts* with a value from *tsvalues*, so if *chkptts* had been the only seal log, we just erased it.

⟨3⟩ *LogInTs' ⇒ LogInTs*
 ⟨4⟩ HAVE *LogInTs'*
 ⟨4⟩ *AllCurrentTs' ≠ {}*BY DEF *LogInTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ *chkptts' ∈ tsvalues'*OBVIOUS
 ⟨4⟩ $\exists ts \in tsvalues' : CheckTsIsCurrent(ts)$ OBVIOUS
 ⟨4⟩ QED BY DEF *LogInTs*

Any remaining current seal attestations existed previously, so they must contain the same log.

⟨3⟩ $\forall ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$
 ⟨4⟩ TAKE *ts1, ts2 ∈ AllCurrentTs'*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ *chkptts' ∈ tsvalues*OBVIOUS
 ⟨4⟩ *ts1 ∈ AllCurrentTs*OBVIOUS
 ⟨4⟩ *ts2 ∈ AllCurrentTs*OBVIOUS
 ⟨4⟩ QED BY DEF *InvOneLog*

If there are any remaining current seal attestations, the log in them has to be the same as before.

⟨3⟩ *LogInTs' ⇒ UNCHANGED CurrentTsLog*
 ⟨4⟩ HAVE *LogInTs'*
 ⟨4⟩ $\exists ts \in AllCurrentTs' : ts \in AllCurrentTs$
 ⟨5⟩ USE DEF *LogInTs*
 ⟨5⟩ USE DEF *AllCurrentTs*
 ⟨5⟩ USE DEF *CheckTsIsCurrent*
 ⟨5⟩ $\forall ts \in AllCurrentTs' : ts \in AllCurrentTs$ OBVIOUS
 ⟨5⟩ QED OBVIOUS
 ⟨4⟩ $\forall ts1 \in AllCurrentTs' :$
 $\forall ts0 \in AllCurrentTs :$
 $ts1.appPcr = ts0.appPcr$
 BY DEF *InvOneLog*
 ⟨4⟩ QED BY DEF *CurrentTsLog*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt1 WhenCorrect

⟨2⟩16. CASE *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ USE *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1 WhenCorrect*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*
 ⟨3⟩ UNCHANGED *LogInApp* BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*

EnterSemChkptPredicate guarantees that the log is in the seal attestations (in particular, in *chkptts*).

This depends on

$\wedge \text{BugChkptNoCheckTsHappy} \triangleq \text{FALSE}$
 $\wedge \text{BugChkptNoCheckTsCtr} = \text{FALSE}$

⟨3⟩ $\text{LogInTs} \wedge \text{CheckTsIsCurrent}(\text{chkptts}) \wedge \text{CurrentTsLog} = \text{chkptts.appPcr}$
 ⟨4⟩ USE DEF *EnterSemChkptPredicate*
 ⟨4⟩ $\text{BugChkptNoCheckTsHappy} = \text{FALSE}$ BY DEF *BugChkptNoCheckTsHappy*
 ⟨4⟩ $\text{BugChkptNoCheckTsCtr} = \text{FALSE}$ BY DEF *BugChkptNoCheckTsCtr*
 ⟨4⟩ *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨4⟩ $\text{AllCurrentTs} \neq \{\}$ BY DEF *AllCurrentTs*
 ⟨4⟩ $\text{CurrentTsLog} = \text{chkptts.appPcr}$
 ⟨5⟩ $\forall ts \in \text{AllCurrentTs} : ts.appPcr = \text{chkptts.appPcr}$
 BY DEF *AllCurrentTs, InvOneLog*
 ⟨5⟩ QED BY DEF *CurrentTsLog*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ $\text{InvOneLog}!\text{goal}!\text{obtains}'$ BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ $\text{InvOneLog}!\text{goal}!\text{revokes}'$ BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ $\text{InvVerifiableRevocation}'$ BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

NextSemChkpt1 WhenIncorrect

⟨2⟩17. CASE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ USE DEF *NextSemChkpt1 WhenIncorrect*
 ⟨3⟩ UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 ⟨3⟩ UNCHANGED *AllCurrentTs*
 ⟨4⟩ USE DEF *AllCurrentTs*
 ⟨4⟩ USE DEF *CheckTsIsCurrent*
 ⟨4⟩ QED BY DEF *LogInTs*
 ⟨3⟩ UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 ⟨3⟩ UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending *sem pcr* with *Unhappy* results in something other than *SemHappy*, which indicates that the log is not in the application *pcr*.

```

<3> ¬LogInApp'
  <4> semPcr' ≠ SemHappy
    <5> USE DEF SemHappy
    <5> USE DEF SemProtect
    <5> USE DEF Pcri
    <5> USE DEF Pcrx
    <5> USE ThmPcrInitIsPcr
    <5> USE ThmPcrExtendIsPcr
    <5> semPcr = SemProtect BY DEF InvInSemProtect, InSem
    <5> USE AssSemHappy
    <5> QED BY ThmPcrExtendAnticollision
  <4> QED BY DEF LogInApp
<3> UNCHANGED LogInTs
  <4> USE DEF AllCurrentTs
  <4> USE DEF CheckTsIsCurrent
  <4> QED BY DEF LogInTs
<3> InvOneLog!goal!obtains'      BY DEF IsOnLog, InvOneLog
<3> InvOneLog!goal!revokes'     BY DEF IsOnLog, InvOneLog
<3> InvVerifiableRevocation'   BY DEF InvVerifiableRevocation, InvOneLog
<3> QED BY DEF InvOneLog

```

NextSemChkpt2

```

<2> 18. CASE NextSemChkpt2
  <3> USE NextSemChkpt2
  <3> USE DEF NextSemChkpt2
  <3> UNCHANGED CheckTsIsCurrent(chkptts) BY DEF CheckTsIsCurrent
  <3> UNCHANGED AllCurrentTs
    <4> USE DEF AllCurrentTs
    <4> USE DEF CheckTsIsCurrent
    <4> QED BY DEF LogInTs
  <3> UNCHANGED CurrentTsLog BY DEF CurrentTsLog
  <3> UNCHANGED LogInNv
    <4> USE DEF InvType
    <4> USE DEF Nv
    <4> QED BY DEF LogInNv
  <3> UNCHANGED LogInApp      BY DEF LogInApp
  <3> UNCHANGED LogInTs      BY DEF LogInTs

```

Storing the log from *chkptts* to the *nv* ram.

This depends on *BugChkptSaveCurApp* \triangleq FALSE

```

<3> nv'.appPcr = chkptts.appPcr
  <4> BugChkptSaveCurApp = FALSE BY DEF BugChkptSaveCurApp
  <4> QED BY DEF InvType, Nv
<3> InvOneLog!goal!obtains'      BY DEF IsOnLog, InvOneLog

```

$\langle 3 \rangle$ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 $\langle 3 \rangle$ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 $\langle 3 \rangle$ QED BY DEF *InvOneLog*

NextSemChkpt3

$\langle 2 \rangle$ 19. CASE *NextSemChkpt3*

$\langle 3 \rangle$ USE *NextSemChkpt3*
 $\langle 3 \rangle$ USE DEF *NextSemChkpt3*
 $\langle 3 \rangle$ UNCHANGED *LogInNv* BY DEF *LogInNv*
 $\langle 3 \rangle$ UNCHANGED *LogInApp* BY DEF *LogInApp*

Since no signed *ts* seal can have a *bootCtr* greater than the current *bootCtr*, incrementing *bootCtr* erases any log that might have been in a seal attestation.

This depends on *BugChkptNoIncCtr* \triangleq FALSE

$\langle 3 \rangle$ \neg *LogInTs'*
 $\langle 4 \rangle$ USE DEF *AllCurrentTs*
 $\langle 4 \rangle$ USE DEF *CheckTsIsCurrent*
 $\langle 4 \rangle$ $\forall ts \in tvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootCtr'$
 $\langle 5 \rangle$ TAKE $ts \in tvalues' \cup \{chkptts'\}$
 $\langle 5 \rangle$ $ts \in tvalues \cup \{chkptts\}$ OBVIOUS
 $\langle 5 \rangle$ HAVE $ts \in SignedTs$
 $\langle 5 \rangle$ $ts.bootCtr \leq bootCtr$ BY DEF *InvSignedTsLeqBoot*
 $\langle 5 \rangle$ $ts.bootCtr \in Nat$ BY DEF *SignedTs*
 $\langle 5 \rangle$ $bootCtr \in Nat$ BY DEF *InvType*
 $\langle 5 \rangle$ $bootCtr' \in Nat$ BY DEF *InvType*
 $\langle 5 \rangle$ $bootCtr < bootCtr'$
 $\langle 6 \rangle$ *BugChkptNoIncCtr* = FALSE BY DEF *BugChkptNoIncCtr*
 $\langle 6 \rangle$ QED BY *ThmNatInc*
 $\langle 5 \rangle$ $ts.bootCtr < bootCtr'$ BY *ThmNatLeqLt*
 $\langle 5 \rangle$ QED BY *ThmNatLeqXorGt, ThmNatLeqIsReflexive*
 $\langle 4 \rangle$ QED BY DEF *LogInTs*

Erases all current *ts* seal attestations.

$\langle 3 \rangle$ $AllCurrentTs' = \{\} \wedge \neg CheckTsIsCurrent(chkptts')$
 $\langle 4 \rangle$ USE DEF *AllCurrentTs*
 $\langle 4 \rangle$ USE DEF *CheckTsIsCurrent*
 $\langle 4 \rangle$ QED BY DEF *LogInTs*
 $\langle 3 \rangle$ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 $\langle 3 \rangle$ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 $\langle 3 \rangle$ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 $\langle 3 \rangle$ QED BY DEF *InvOneLog*

NextSemChkpt4

(2)20. CASE *NextSemChkpt4*
 (3) USE *NextSemChkpt4*
 (3) USE DEF *NextSemChkpt4*
 (3) UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 (3) UNCHANGED *AllCurrentTs*
 (4) USE DEF *AllCurrentTs*
 (4) USE DEF *CheckTsIsCurrent*
 (4) QED BY DEF *LogInTs*
 (3) UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*

Setting *nv* current indicates that the log is in the *nv* ram.

This depends on *BugChkptNoSetCur* \triangleq FALSE

(3) *LogInNv'*
 (4) *nv'.current*
 (5) *BugChkptNoSetCur* = FALSE BY DEF *BugChkptNoSetCur*
 (5) QED BY DEF *InvType, Nv*
 (4) QED BY DEF *LogInNv*
 (3) UNCHANGED *LogInApp* BY DEF *LogInApp*
 (3) UNCHANGED *LogInTs* BY DEF *LogInTs*

Since *nv.current* was changed, the prover needs to see the type of *nv* to know that the *appPcr* field did not change.

(3) UNCHANGED *nv.appPcr* BY DEF *InvType, Nv*
 (3) *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 (3) *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 (3) *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 (3) QED BY DEF *InvOneLog*

NextSemChkpt5

(2)21. CASE *NextSemChkpt5*
 (3) USE *NextSemChkpt5*
 (3) USE DEF *NextSemChkpt5*
 (3) UNCHANGED *CheckTsIsCurrent(chkptts)* BY DEF *CheckTsIsCurrent*
 (3) UNCHANGED *AllCurrentTs*
 (4) USE DEF *AllCurrentTs*
 (4) USE DEF *CheckTsIsCurrent*
 (4) QED BY DEF *LogInTs*
 (3) UNCHANGED *CurrentTsLog* BY DEF *CurrentTsLog*
 (3) UNCHANGED *LogInNv* BY DEF *LogInNv*

Extending sem pcr with Unhappy results in something other than *SemHappy*, which indicates that the log is not in the application pcr.

(3) \neg *LogInApp'*
 (4) *semPcr' \neq SemHappy*
 (5) USE DEF *SemHappy*
 (5) USE DEF *SemProtect*
 (5) USE DEF *Pcri*
 (5) USE DEF *Pcrx*

⟨5⟩ USE *ThmPcrInitIsPcr*
 ⟨5⟩ USE *ThmPcrExtendIsPcr*
 ⟨5⟩ *semPcr = SemProtect* BY DEF *InvInSemProtect, InSem*
 ⟨5⟩ USE *AssSemHappy*
 ⟨5⟩ QED BY *ThmPcrExtendAnticollision*
 ⟨4⟩ QED BY DEF *LogInApp*
 ⟨3⟩ UNCHANGED *LogInTs* BY DEF *LogInTs*
 ⟨3⟩ *InvOneLog!goal!obtains'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvOneLog!goal!revokes'* BY DEF *IsOnLog, InvOneLog*
 ⟨3⟩ *InvVerifiableRevocation'* BY DEF *InvVerifiableRevocation, InvOneLog*
 ⟨3⟩ QED BY DEF *InvOneLog*

 ⟨2⟩ QED
 BY ⟨2⟩1,
 ⟨2⟩2, ⟨2⟩3, ⟨2⟩4, ⟨2⟩5, ⟨2⟩6, ⟨2⟩8, ⟨2⟩9,
 ⟨2⟩10, ⟨2⟩11, ⟨2⟩12, ⟨2⟩13, ⟨2⟩14, ⟨2⟩15, ⟨2⟩16, ⟨2⟩17,
 ⟨2⟩18, ⟨2⟩19, ⟨2⟩20, ⟨2⟩21
 DEF *Next*
 ⟨1⟩ QED BY DEF *InvOneLog*

It is an invariant of the specification.

THEOREM *ThmInvOneLog* \triangleq

Spec \Rightarrow \square *InvOneLog*

PROOF

⟨1⟩ *Init* \Rightarrow *InvOneLog* BY *ThmInitInvOneLog*

⟨1⟩ *InvOneLog* \wedge [*Next*]_{vars} \Rightarrow *InvOneLog'*

BY *ThmNextInvOneLog*

⟨1⟩ QED BY *RuleINV1* DEF *Spec*

It is an invariant of the specification.

THEOREM *ThmInvAccessUndeniability* \triangleq
 $Spec \Rightarrow \Box InvAccessUndeniability$

PROOF

- (1) $InvOneLog \Rightarrow InvAccessUndeniability$
- (2) HAVE *InvOneLog*
- (2) USE DEF *InvOneLog*
- (2) USE DEF *InvAccessUndeniability*
- (2) $BugAuditNoCheckHappy = FALSEBY DEF BugAuditNoCheckHappy$
- (2) CASE $\neg LogInApp$ BY DEF *LogInApp* unable to audit
- (2) CASE *LogInApp* BY DEF *IsOnLog* audit
- (2) QED OBVIOUS
- (1) $Spec \Rightarrow \Box InvOneLog$ BY *ThmInvOneLog*
- (1) QED

PROOF OF INVARIANT *InvVerifiableRevocation*

It is an invariant of the specification.

THEOREM *ThmInvVerifiableRevocation* \triangleq
 $Spec \Rightarrow \Box InvVerifiableRevocation$

PROOF

- (1) $InvOneLog \Rightarrow InvVerifiableRevocation$
- (2) HAVE *InvOneLog*
- (2) USE DEF *InvOneLog*
- (2) QED OBVIOUS
- (1) $Spec \Rightarrow \Box InvOneLog$ BY *ThmInvOneLog*
- (1) QED

