Inserting Intentional Bugs for Model Checking Assurance

Thomas L. Rodeheffer
Microsoft Research, Silicon Valley

Ramakrishna Kotla
Microsoft Research, Silicon Valley

Writing a formal specification for a system or system component forces one to be precise about the system's actions, and this process often flushes out design bugs just by itself. But once the specification is written, how does one know it is correct? Before investing the effort in writing a formal proof, one can use a model checker to explore the specification's state space. Unfortunately, most interesting systems have specifications whose state space is enormous, if not infinite. If the model checker finds no errors in the relatively small, constrained configurations that it can feasibly explore, that is well and good, but if there were an error, would the model checker have found it? One way to get some assurance of this is to introduce some errors on purpose, and see if the model checker finds them.

This paper presents the results of model checking, with inserted errors, a TLA+ specification for a node in Pasture, a messaging library that provides secure offline access to data using a TPM. The model checking results give some assurance that the specification is correct; that is, that it maintains its invariants. This paper also presents a formal proof of correctness, checked by the TLA+ Proof System.

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 [10, 4]. 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+ [8] specification for a node in Pasture, a messaging library that provides secure offline 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 [3]. Appendices contain the full text of the formal specification and formal proof.

2 Overview of Pasture

Pasture [6] is a messaging library that provides secure offline access to data. When online, the *receiver* downloads an encrypted copy of the data from a *sender*. Later, when offline, 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, 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) [1, 2] 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 cryptographically impossible to determine any other sequence that would produce the same result. Pasture uses $\mathrm{PCR_{APP}}$ to hold the log summary.

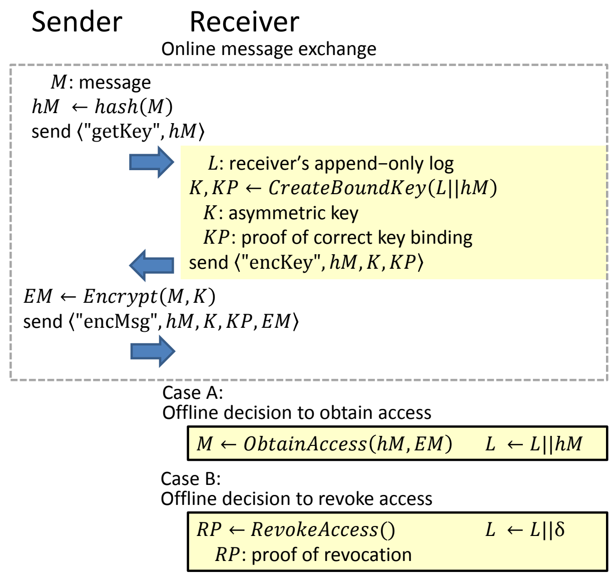

Figure 1: Pasture protocol.

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 $\delta \neq hM$. This extension makes it (cryptographically) 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 [10] presented a general solution to this problem for any deterministic application. Memoir maintains

[1] The 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.

[2] Certain 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.

a cryptographic log summary of application states in a PCR much like Pasture's $\mathrm{PCR_{APP}}$. 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 NV RAM location and then sets an NV RAM flag indicating that the copy is current. The recovery routine checks that the NV RAM value is marked as current and if so plays back measurements from the full log, re-extending the PCR until its content matches the value saved in the NV RAM. 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 NV RAM flag is cleared indicating that the NV RAM copy of the PCR is no longer current.

Memoir exploits Secure Execution Mode (SEM) in the manner developed by Flicker [9]. SEM enables a routine to run in a protected environment, with interrupts, other cores, and DMA disabled, and with a special $\mathrm{PCR_{SEM}}$ 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, 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 $\mathrm{PCR_{APP}}$ from the full log, then enters SEM to verify that $\mathrm{PCR_{APP}}$ matches the value saved in the NV RAM and that the value in NV RAM is current. If so, $\mathrm{PCR_{SEM}}$ is extended by *Happy*, to produce a value *SemHappy* that can be reached in no other way, and the NV RAM flag is cleared to indicate that the NV RAM value can no longer be considered as current. Otherwise, $\mathrm{PCR_{SEM}}$ is extended by *Unhappy*, producing a different value.

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

Likewise, RevokeAccess and Audit quote both

CreateBoundKey(*hM*):

$R_t \leftarrow$ TPM_Read($\mathrm{PCR_{APP}}$)
$R_{t+1} \leftarrow$ SHA1(R_t || *hM*)
[transport session] [BINDKEY] $K \leftarrow$ TPM_CreateWrapKey({
  $\mathrm{PCR_{APP}} = R_{t+1}$ &&
  $\mathrm{PCR_{SEM}}$ = *SemHappy* &&
  $\mathrm{PCR_{SEAL}}$ = *SealReboot* })
$\alpha \leftarrow$
KP $\leftarrow \langle$ "CreateBoundKey", *hM*, R_t, R_{t+1}, $\alpha \rangle$

ObtainAccess(*hM*, *EM*):

append *hM* to full log
TPM_Extend($\mathrm{PCR_{APP}}$, *hM*)
$M \leftarrow$ TPM_Unbind(*EM*)

RevokeAccess():

$R_t \leftarrow$ TPM_Read($\mathrm{PCR_{APP}}$)
append δ to full log
TPM_Extend($\mathrm{PCR_{APP}}$, δ)
R'_{t+1}, S'_{t+1}, A'_{t+1}, $\alpha \leftarrow$
  TPM_Quote($\mathrm{PCR_{APP}}$, $\mathrm{PCR_{SEM}}$, $\mathrm{PCR_{SEAL}}$)
RP $\leftarrow \langle$ "RevokeAccess", δ, R_t, R'_{t+1}, S'_{t+1}, A'_{t+1}, $\alpha \rangle$

Audit(*nonce*):

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

Recover():

FOR EACH entry Δ on full log: TPM_Extend($\mathrm{PCR_{APP}}$, Δ)
[secure execution mode]
IF nv.*current* && nv.*R* = TPM_Read($\mathrm{PCR_{APP}}$)
THEN
  nv.*current* $\leftarrow$ FALSE
  TPM_Extend($\mathrm{PCR_{SEM}}$, *Happy*)
ELSE
  TPM_Extend($\mathrm{PCR_{SEM}}$, *Unhappy*)

Checkpoint():

[transport session] [SEAL]
$R_t \leftarrow$ TPM_Read($\mathrm{PCR_{APP}}$)
$S_t \leftarrow$ TPM_Read($\mathrm{PCR_{SEM}}$)
$A_t \leftarrow$ TPM_Read($\mathrm{PCR_{SEAL}}$)
$C_t \leftarrow$ TPM_ReadCounter(CTR)
$\alpha \leftarrow$ TPM_Extend($\mathrm{PCR_{SEAL}}$, *Seal*)
[secure execution mode]
nv.*R* $\leftarrow R_t$
IF $\mathrm{Valid_{SEAL}}$(α, R_t, S_t, A_t, C_t)
  && S_t = *SemHappy*
  && A_t = *SealReboot*
  && C_t = TPM_ReadCounter(CTR)
THEN
  TPM_IncrementCounter(CTR)
  nv.*current* $\leftarrow$ TRUE
TPM_Extend($\mathrm{PCR_{SEM}}$, *Unhappy*)

Figure 2: Pasture operations.

$\mathrm{PCR_{APP}}$ and $\mathrm{PCR_{SEM}}$ in order to prove that $\mathrm{PCR_{APP}}$ was quoted at a time when $\mathrm{PCR_{SEM}}$ contained *SemHappy*.

Pasture's Checkpoint operation has a difficulty. It needs to verify that $\mathrm{PCR_{SEM}}$ contains *SemHappy* so that it can trust the current contents of $\mathrm{PCR_{APP}}$, 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 $\mathrm{PCR_{APP}}$ and $\mathrm{PCR_{SEM}}$ 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 $\mathrm{PCR_{APP}}$, or else an adversary could take the SEAL, extend $\mathrm{PCR_{APP}}$ 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, $\mathrm{PCR_{SEAL}}$ is used. $\mathrm{PCR_{SEAL}}$ normally contains its initial value *SealReboot*, which is checked in CreateBoundKey and quoted in RevokeAccess and Audit. The SEAL transport session extends $\mathrm{PCR_{SEAL}}$ thus rendering $\mathrm{PCR_{APP}}$ useless until the next reboot.

3 The specification

Appendix A gives a TLA+ [8] 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 $\mathrm{PCR_{APP}}$ from measurements recorded in the full log and then enters SEM to verify that the resulting value in $\mathrm{PCR_{APP}}$ is current. The specification accomplishes the re-extension by allowing any possible sequence of extensions of $\mathrm{PCR_{APP}}$, 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.
- *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 NV RAM. 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

[3]The protection of Pasture's NV RAM depends on the assumption that the value *SemProtect* is present in $\mathrm{PCR_{SEM}}$ only during Pasture's secure execution mode routines, which is the subject of the invariant *InvNvProtection*.

node is in SEM the adversary cannot interpose any actions except to reboot the node.

Next the specification introduces definitions for Pasture's protected NV RAM, 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 *InvNvProtection* asserts that access to Pasture's NV RAM 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 $\mathrm{PCR}_{\mathrm{APP}}$; 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 $\mathrm{PCR}_{\mathrm{SEM}}$. Pasture requires at least one, so that the Pasture SEM routines can extend $\mathrm{PCR}_{\mathrm{SEM}}$ before exiting, which is required to remove access privileges from Pasture's NV RAM. Note that the specification does not count entering a SEM routine as requiring an extension to $\mathrm{PCR}_{\mathrm{SEM}}$, but merely initializes $\mathrm{PCR}_{\mathrm{SEM}}$ with *SemProtected* which represents the result of resetting and then extending with the cryptographic module hash. The TPM semantics of resetting $\mathrm{PCR}_{\mathrm{SEM}}$ ensures that it is cryptographically impossible to reach *SemProtected* in any other way.
- MaxSealPcrLen. The maximum number of extensions of $\mathrm{PCR}_{\mathrm{SEAL}}$. Pasture requires at least one, so that the SEAL transport session can extend $\mathrm{PCR}_{\mathrm{SEAL}}$.
- 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.

MaxAppPcrLen	MaxSemPcrLen	MaxSealPcrLen	MaxTsValues	MaxBootCtr			laptop 4 GB 4 cores	server 128 GB 48 cores
configuration					depth	distinct states	run time	run time
1	1	1	1	1	31	47742	7s	
1	1	1	1	2	32	106556	83s	
1	1	1	2	1	36	966697	110s	
1	1	2	1	1	33	369750	41s	
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.

- MaxBootCtr. The maximum value of the boot counter. Pasture increments this counter once each time through the Checkpoint routine.

We used the TLA+ toolbox [7] 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.

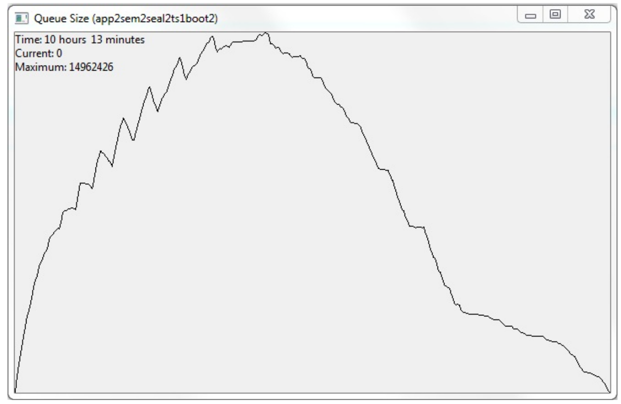

Figure 3: TLC queue size plot for the (2,2,2,1,2) configuration running on the laptop.

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

[4]Assuming 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.

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 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.
- *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 NV RAM.
- *BugRecovNoCheckCur* models what happens if secure execution mode within recovery fails to check that the value saved in the NV RAM is marked as current.
- *BugRecovNoClrCur* models what happens if secure execution mode within recovery fails to clear the *current* flag in the NV RAM.
- *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.

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

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

- *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 NV RAM 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 NV RAM.

- *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 $\mathrm{PCR}_{\mathrm{APP}}$) 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 NV RAM,
3. a reboot,
4. a normal Recovery sequence,
5. an extension of $\mathrm{PCR}_{\mathrm{APP}}$ 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 NV RAM as current, (this is where the bug took effect)
7. a reboot, and finally
8. a normal Recovery sequence, which restored $\mathrm{PCR}_{\mathrm{APP}}$ to the value of the empty log, while establishing $\mathrm{PCR}_{\mathrm{SEM}} = \mathit{SemHappy}$ and $\mathrm{PCR}_{\mathrm{SEAL}} = \mathit{SealReboot}$.

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

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 [11] 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 theses 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 NV RAM 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 $\mathrm{PCR}_{\mathrm{SEAL}} = \mathit{SealReboot}$. 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 session to take a SEAL, then perform some additional extensions on $\mathrm{PCR}_{\mathrm{APP}}$, then perform the transport session a second time to take a second SEAL. Since the checkpoint SEM routine fails to check the value of $\mathrm{PCR}_{\mathrm{SEAL}}$ 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 $\mathrm{PCR}_{\mathrm{SEAL}}$ will no longer contain its original $\mathit{SealReboot}$ 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 [3].

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 $\mathrm{PCR}_{\mathrm{APP}}$, when $\mathrm{PCR}_{\mathrm{SEM}} = \mathit{SemHappy}$ and $\mathrm{PCR}_{\mathrm{SEAL}} = \mathit{SealReboot}$. 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 $\mathrm{PCR}_{\mathrm{SEM}} = \mathit{SemHappy}$, $\mathrm{PCR}_{\mathrm{SEAL}} = \mathit{SealReboot}$, 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 NV RAM, 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 $\mathrm{PCR}_{\mathrm{SEM}}$ and $\mathrm{PCR}_{\mathrm{SEAL}}$ 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 [5].

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 [4]. 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 [5]. So even though a formal specification can be somewhat lengthy, a formal proof of its correctness tends to be much more lengthy.

7 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 confidence that the specification is correct.

References

[1] Trusted Platform Module V1.2 Specification. http://www.trustedcomputinggroup.org.

[2] D. Challener, K. Yoder, R. Catherman, D. Safford, and L. Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, Jan 2008.

[3] K. Chaudhuri, D. Doligez, L. Lamport, and S. Merz. Verifying safety properties with the TLA+ proof system. In *IJCAR*, pages 142–148, 2010.

[4] 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.

[5] 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.

[6] R. Kotla, T. Rodeheffer, I. Roy, P. Steudi, and B. Wester. Secure offline data access using commodity trusted hardware. To appear in *OSDI*, 2012.

[7] L. Lamport. The TLA toolbox. http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html.

[8] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[9] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Eurosys*, 2008.

[10] 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.

[11] T. Rodeheffer. Cyclic commit protocol specifications. Technical Report MSR-TR-2008-125, Microsoft Research, Sep 2008. http://research.microsoft.com/apps/pubs/?id=70631.

A Specification

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

$$Pcrx \triangleq \{ PcrxHAPPY, PcrxUNHAPPY, PcrxSEAL, PcrxOBTAIN, PcrxREVOKE \}$$

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.

$$Pcr \triangleq [init : Pcri, extq : Seq(Pcrx)]$$

Initial pcr value.

$$PcrInit(i) \triangleq [init \mapsto i, extq \mapsto \langle\rangle]$$

Extend a pcr value.

$PcrExtend(p, x) \triangleq$
  [
    $init \mapsto p.init,$
    $extq \mapsto Append(p.extq, x)$
  ]

Number of extensions in a pcr value.

$PcrLen(p) \triangleq$
  $Len(p.extq)$

$Pcr\ s$ 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.

$PcrLeq(s, t) \triangleq$
  LET
    $sinit \triangleq s.init$
    $sextq \triangleq s.extq$
    $sn \triangleq Len(sextq)$

    $tinit \triangleq t.init$
    $textq \triangleq t.extq$
    $tn \triangleq Len(textq)$

    $uextq \triangleq SubSeq(textq, 1, sn)$
  IN
  $\land\ sinit = tinit$
  $\land\ sn \leq tn$
  $\land\ sextq = uextq$

Determine if a pcr value has been extended.

$PcrHasExtension(p) \triangleq$
  $PcrLen(p) > 0$

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.

$PcrPrior(p) \triangleq$
  CASE $PcrHasExtension(p) \rightarrow$
  LET
    $n \triangleq Len(p.extq) - 1$
  IN
  [
    $init \mapsto p.init,$
    $extq \mapsto SubSeq(p.extq, 1, n)$
  ]

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

$PcrLastExtension(p) \triangleq$
  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

$PcIDLE \triangleq$ "idle"

steps in secure execution mode within recover

$PcRECOV1 \triangleq$ "recov1"
$PcRECOV2 \triangleq$ "recov2"
$PcRECOV3 \triangleq$ "recov3"
$PcRecov \triangleq \{PcRECOV1, PcRECOV2, PcRECOV3\}$

steps in secure execution mode within checkpoint

$PcCHKPT1 \triangleq$ "chkpt1"

$PcCHKPT2 \triangleq$ "chkpt2"
$PcCHKPT3 \triangleq$ "chkpt3"
$PcCHKPT4 \triangleq$ "chkpt4"
$PcCHKPT5 \triangleq$ "chkpt5"
$PcChkpt \triangleq \{PcCHKPT1, PcCHKPT2, PcCHKPT3, PcCHKPT4, PcCHKPT5\}$

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

PROTECTED NV RAM STATE

$Nv \triangleq$
  [
    $appPcr : Pcr,$ copy of the application pcr
    $current$: BOOLEAN copy of application pcr is current
  ]

$InitNv \triangleq$
  [
    $appPcr \mapsto AppReboot,$
    $current \mapsto$ TRUE
  ]

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.

$SignedTs \triangleq$
  [
    $semPcr : Pcr,$ copy of the secure execution mode pcr on entry
    $sealPcr : Pcr,$ copy of the seal pcr on entry
    $appPcr : Pcr,$ copy of the application pcr on entry
    $bootCtr : Nat$ copy of the reboot counter on entry
  ]

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

$NullTs \triangleq$ 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 $sealPcr$ — 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

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

$\quad pcr1o \triangleq appPcr$ current app pcr

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

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

IN

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

$\land PcrHasExtension(pcr1o)$ have an extension

$\land x = PcrxOBTAIN$ 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.

$\land$ 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.

$\land$ IF $BugObtainAccessNoCheckSeal$ THEN TRUE ELSE $sealPcr = SealReboot$

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

$\land$ UNCHANGED nv

$\land$ UNCHANGED $appPcr$

$\land$ UNCHANGED $semPcr$

$\land$ UNCHANGED $sealPcr$

$\land$ UNCHANGED $bootCtr$

$\land$ UNCHANGED pc

$\land$ UNCHANGED $chkptts$

$\land$ UNCHANGED $tsvalues$

$\land$ 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

$\quad pcr1r \triangleq appPcr$ current app pcr

$pcr0 \triangleq PcrPrior(pcr1r)$ prior app pcr
$x \triangleq PcrLastExtension(pcr1r)$ presumed REVOKE extension
IN
$\land \neg InSem$ must not be in secure execution mode
$\land PcrHasExtension(pcr1r)$ have an extension
$\land x = PcrxREVOKE$ 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.

$\land$ 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.

$\land$ IF $BugProveRevokeNoCheckSeal$ THEN TRUE ELSE
$sealPcr = SealReboot$
$\land revokes' = revokes \cup \{pcr1r\}$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $semPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED pc
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ 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 chkppts had before and call sem checkpoint with that value.

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

$NextForgetSealTs \triangleq$

$\exists\, ts \in tsvalues :$

$\wedge\ tsvalues' = tsvalues \setminus \{ts\}$

$\wedge$ UNCHANGED nv

$\wedge$ UNCHANGED $appPcr$

$\wedge$ UNCHANGED $semPcr$

$\wedge$ UNCHANGED $sealPcr$

$\wedge$ 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.

$NextExtendAppPcr \triangleq$

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

$\wedge\ \exists\, x \in Pcrx :$

$\wedge\ appPcr' = PcrExtend(appPcr, x)$

$\wedge$ UNCHANGED nv

$\wedge$ UNCHANGED $semPcr$

$\wedge$ UNCHANGED $sealPcr$

$\wedge$ UNCHANGED $bootCtr$

$\wedge$ UNCHANGED pc

$\wedge$ UNCHANGED $chkptts$

$\wedge$ UNCHANGED $tsvalues$

$\wedge$ UNCHANGED $obtains$

$\wedge$ UNCHANGED $revokes$

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

$NextExtendSemPcr \triangleq$

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

$\wedge\ \exists\, x \in Pcrx :$

$\wedge\ semPcr' = PcrExtend(semPcr, x)$

$\wedge$ UNCHANGED nv

$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED pc
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ UNCHANGED $revokes$

Extend seal pcr arbitrarily, due to adversarial action.

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

Increment reboot counter arbitrarily, due to adversarial action.

$NextIncBootCtr \triangleq$
$\land \neg InSem$ — must not be in secure execution mode
$\land bootCtr' = bootCtr + 1$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $semPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED pc
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ 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.

$NextEnterSemRecov \triangleq$
$\land \neg InSem$ — must not be in secure execution mode
$\land semPcr' = SemProtect$
$\land pc' = PcRECOV1$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ 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.

$\land$ 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.

$\land$ IF $BugRecovNoCheckCur$ THEN TRUE ELSE
$nv.current$

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

$NextSemRecov1WhenCorrect \triangleq$
$\land pc = PcRECOV1$
$\land EnterSemRecovPredicate$
$\land pc' = PcRECOV2$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $semPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ UNCHANGED $revokes$

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

$NextSemRecov1WhenIncorrect \triangleq$
$\land\ pc = PcRECOV1$
$\land\ \neg EnterSemRecovPredicate$
$\land\ semPcr' = PcrExtend(semPcr,\ PcrxUNHAPPY)$
$\land\ pc' = PcIDLE$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ 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
$\land\ pc = PcRECOV2$
$\land\ nv' = [nv$ EXCEPT $!.current = nvcurrent1]$
$\land\ pc' = PcRECOV3$
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $semPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ UNCHANGED $revokes$

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

$NextSemRecov3 \triangleq$
$\land\ pc = PcRECOV3$
$\land\ semPcr' = PcrExtend(semPcr,\ PcrxHAPPY)$
$\land\ pc' = PcIDLE$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $sealPcr$

$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ 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.

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
$PcrExtend(sealPcr,\ PcrxSEAL)$
IN
$\land \neg InSem$ must not be in secure execution mode
$\land tsvalues' = tsvalues \cup \{ts\}$
$\land sealPcr' = sealPcr1$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $semPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED pc
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $obtains$
$\land$ 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.

$NextEnterSemChkpt \triangleq$

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

$\land \exists\, ts \in tsvalues :$ any known ts value

$\land semPcr' = SemProtect$

$\land pc' = PcCHKPT1$

$\land chkptts' = ts$

$\land$ UNCHANGED nv

$\land$ UNCHANGED $appPcr$

$\land$ UNCHANGED $sealPcr$

$\land$ UNCHANGED $bootCtr$

$\land$ UNCHANGED $tsvalues$

$\land$ UNCHANGED $obtains$

$\land$ UNCHANGED $revokes$

Predicate for correct entry to secure execution mode within checkpoint.

$EnterSemChkptPredicate \triangleq$

$\land chkptts \in SignedTs$

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

$\land$ 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.

$\land$ 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.

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

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

$NextSemChkpt1WhenCorrect \triangleq$

$\land pc = PcCHKPT1$

$\land EnterSemChkptPredicate$

$\land pc' = PcCHKPT2$

$\land$ UNCHANGED nv

$\land$ UNCHANGED $appPcr$

$\land$ UNCHANGED $semPcr$

$\land$ UNCHANGED $sealPcr$

$\land$ UNCHANGED $bootCtr$

$\land$ UNCHANGED $chkptts$

$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ UNCHANGED $revokes$

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

$NextSemChkpt1WhenIncorrect \triangleq$
$\land$ $pc = PcCHKPT1$
$\land$ $\neg EnterSemChkptPredicate$
$\land$ $semPcr' = PcrExtend(semPcr, PcrxUNHAPPY)$
$\land$ $pc' = PcIDLE$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ 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
$\land$ $pc = PcCHKPT2$
$\land$ $nv' = [nv$ EXCEPT $!.appPcr = nvappPcr1]$
$\land$ $pc' = PcCHKPT3$
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $semPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ 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

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

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

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

$NextSemChkpt5 \triangleq$

$\land\ pc = PcCHKPT5$

$\land\ semPcr' = PcrExtend(semPcr,\ PcrxUNHAPPY)$
$\land\ pc' = PcIDLE$
$\land$ UNCHANGED nv
$\land$ UNCHANGED $appPcr$
$\land$ UNCHANGED $sealPcr$
$\land$ UNCHANGED $bootCtr$
$\land$ UNCHANGED $chkptts$
$\land$ UNCHANGED $tsvalues$
$\land$ UNCHANGED $obtains$
$\land$ UNCHANGED $revokes$

SPECIFICATION

$Init \triangleq$
$\land\ nv = InitNv$
$\land\ appPcr = AppReboot$
$\land\ semPcr = SemReboot$
$\land\ sealPcr = SealReboot$
$\land\ bootCtr = 0$
$\land\ pc = PcIDLE$
$\land\ chkptts = NullTs$
$\land\ tsvalues = \{NullTs\}$ — anybody can create a $NullTs$
$\land\ obtains = \{\}$
$\land\ revokes = \{\}$

$Next \triangleq$
$\lor\ NextObtainAccess$
$\lor\ NextProveRevoke$
$\lor\ NextReboot$
$\lor\ NextForgetSealTs$
$\lor\ NextExtendAppPcr$
$\lor\ NextExtendSemPcr$
$\lor\ NextExtendSealPcr$
$\lor\ NextIncBootCtr$
$\lor\ NextEnterSemRecov$
$\lor\ NextSemRecov1WhenCorrect$
$\lor\ NextSemRecov1WhenIncorrect$
$\lor\ NextSemRecov2$
$\lor\ NextSemRecov3$

$\lor$ $NextSealTs$
$\lor$ $NextEnterSemChkpt$
$\lor$ $NextSemChkpt1WhenCorrect$
$\lor$ $NextSemChkpt1WhenIncorrect$
$\lor$ $NextSemChkpt2$
$\lor$ $NextSemChkpt3$
$\lor$ $NextSemChkpt4$
$\lor$ $NextSemChkpt5$

$Spec \triangleq Init \land \Box[Next]_{vars}$

INVARIANTS

Type invariant.

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

Nv protection invariant.

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

$InvNvProtection \triangleq$
$InSem \equiv (semPcr = SemProtect)$

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

$InvVerifiableRevocation \triangleq$

$\forall\, o \in \mathit{obtains} :$ last extension was OBTAIN

$\forall\, r \in \mathit{revokes} :$ last extension was REVOKE

$\mathit{PcrPrior}(o) \neq \mathit{PcrPrior}(r)$ 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 $\mathit{PcrLeq}(o,\ \mathit{appPcr})$, which means than 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.

$\mathit{InvAccessUndeniability} \triangleq$

It is a bug to fail to require the audit to quote $\mathit{SemHappy}$.

$\land$ IF $\mathit{BugAuditNoCheckHappy}$ THEN TRUE ELSE
$\mathit{semPcr} = \mathit{SemHappy}$

It is a bug to fail to require the audit to quote $\mathit{SealReboot}$.

$\land$ IF $\mathit{BugAuditNoCheckSeal}$ THEN TRUE ELSE
$\mathit{sealPcr} = \mathit{SealReboot}$

$\Rightarrow$

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

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
$PcriAPPBOOT \leftarrow$ "boot",

$SEMBOOT$ and $SEMPROTECT$ must be different.

$PcriSEMBOOT \leftarrow$ "boot",
$PcriSEMPROTECT \leftarrow$ "protect",

$PcriSEALBOOT \leftarrow$ "boot",

HAPPY and UNHAPPY must be different.

$PcrxHAPPY \leftarrow 0$,
$PcrxUNHAPPY \leftarrow 1$,

OBTAIN and REVOKE must be different.

$PcrxOBTAIN \leftarrow 0$,
$PcrxREVOKE \leftarrow 1$,

$PcrxSEAL \leftarrow 0$

MODEL-CHECKING CONSTRAINT

Override these definitions to adjust the constraint.

$MaxAppPcrLen \triangleq 1$
$MaxSemPcrLen \triangleq 1$
$MaxSealPcrLen \triangleq 1$
$MaxTsValues \triangleq 1$
$MaxBootCtr \triangleq 1$

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

C Proof

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.

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

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.

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

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

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

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

$$
\begin{aligned}
&LogInNv \triangleq \\
&\quad \land\ nv.current
\end{aligned}
$$

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

$$
\begin{aligned}
&LogInApp \triangleq \\
&\quad \land\ semPcr = SemHappy \\
&\quad \land\ sealPcr = SealReboot
\end{aligned}
$$

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

$LogInTs \triangleq$
$\quad AllCurrentTs \neq \{\}$

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 $SemHappy$ 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.

$IsOnLog(p) \triangleq$

Where to find the log at most times.

$\wedge \quad LogInNv \Rightarrow PcrLeq(p,\ nv.appPcr)$
$\wedge \quad LogInApp \Rightarrow PcrLeq(p,\ appPcr)$
$\wedge \quad LogInTs \Rightarrow PcrLeq(p,\ CurrentTsLog)$

Special places to find the log during secure execution mode.

$\wedge \quad pc = PcRECOV1 \Rightarrow \text{TRUE}$
$\wedge \quad pc = PcRECOV2 \Rightarrow PcrLeq(p,\ appPcr)$
$\wedge \quad pc = PcRECOV3 \Rightarrow PcrLeq(p,\ appPcr)$
$\wedge \quad pc = PcCHKPT1 \Rightarrow \text{TRUE}$
$\wedge \quad pc = PcCHKPT2 \Rightarrow PcrLeq(p,\ chkptts.appPcr)$
$\wedge \quad pc = PcCHKPT3 \Rightarrow PcrLeq(p,\ nv.appPcr)$
$\wedge \quad pc = PcCHKPT4 \Rightarrow PcrLeq(p,\ nv.appPcr)$
$\wedge \quad pc = PcCHKPT5 \Rightarrow PcrLeq(p,\ nv.appPcr)$

ADDITIONAL INVARIANTS

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

$InvInSemProtect \triangleq$
$\quad \wedge\ InvType$
$\quad \wedge\ goal::$
$\quad InSem \Rightarrow semPcr = SemProtect$

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

$InvUnreachableSemProtect \triangleq$
$\quad \wedge\ InvType$
$\quad \wedge\ InvInSemProtect$

$\land$ $goal$::
$\neg InSem \Rightarrow \neg PcrLeq(semPcr, SemProtect)$

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

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

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.

$InvUnforgeableSemHappy \triangleq$
$\land$ $InvType$
$\land$ $InvInSemProtect$
$\land$ $goal$::
$\neg InSem \Rightarrow$
$\lor\ semPcr = SemHappy$
$\lor\ \neg PcrLeq(semPcr, SemHappy)$

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

$InvUnforgeableSealReboot \triangleq$
$\land$ $InvType$
$\land$ $goal$::
$\lor\ sealPcr = SealReboot$
$\lor\ \neg PcrLeq(sealPcr, SealReboot)$

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

$InvProperLastExtension \triangleq$
$\land$ $InvType$
$\land$ $goal$::
$\land\ \forall\, o \in obtains : PcrHasExtension(o) \land PcrLastExtension(o) = PcrxOBTAIN$
$\land\ \forall\, r \in revokes : PcrHasExtension(r) \land PcrLastExtension(r) = PcrxREVOKE$

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)

$InvOneLog \triangleq$

$\land\ InvType$

$\land\ InvSignedTsLeqBoot$

$\land\ InvInSemProtect$

$\land\ InvUnforgeableSemHappy$

$\land\ InvUnforgeableSealReboot$

$\land\ InvProperLastExtension$

$\land\ goal ::$

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

$\land \quad LogInNv \Rightarrow \neg LogInApp \land \neg LogInTs$

$\land \quad LogInApp \Rightarrow \neg LogInNv \land \neg LogInTs$

$\land \quad LogInTs \Rightarrow \neg LogInNv \land \neg LogInApp$

Extra requirements during secure execution mode.

$\land \quad pc = PcRECOV1 \Rightarrow \text{TRUE}$

$\land \quad pc = PcRECOV2 \Rightarrow LogInNv$

$\land \quad pc = PcRECOV3 \Rightarrow \neg LogInNv \land \neg LogInApp \land \neg LogInTs$

$\land \quad pc = PcCHKPT1 \Rightarrow \text{TRUE}$

$\land \quad pc = PcCHKPT2 \Rightarrow LogInTs \land CheckTsIsCurrent(chkptts)$

$\land \quad pc = PcCHKPT3 \Rightarrow LogInTs \land CheckTsIsCurrent(chkptts)$

$\land \quad pc = PcCHKPT4 \Rightarrow \neg LogInNv \land \neg LogInApp \land \neg LogInTs$

$\land \quad pc = PcCHKPT5 \Rightarrow LogInNv$

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

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

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

$\land \quad obtains :: \forall\, o \in obtains : IsOnLog(o)$

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

$\land \quad revokes :: \forall\, r \in revokes : IsOnLog(r)$

We have verifiable revocation.

$\land \quad InvVerifiableRevocation$

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 $ThmNatLeqIsTotal \triangleq \forall i, j \in Nat : i \leq j \lor j \leq i$ BY SMT
THEOREM $ThmNatLeqIsReflexive \triangleq \forall i \in Nat : i \leq i$ BY SMT
THEOREM $ThmNatLeqIsAntisymmetric \triangleq \forall i, j \in Nat : i \leq j \land j \leq i \Rightarrow i = j$ BY SMT
THEOREM $ThmNatLeqIsTransitive \triangleq \forall i, j, k \in Nat : i \leq j \land j \leq k \Rightarrow i \leq k$ BY SMT

$\leq$ minimum is 0

THEOREM $ThmNatLeqMinIsZero \triangleq \forall i \in Nat : 0 \leq i$ BY SMT

$\leq$ is the opposite of $>$

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

THEOREM $ThmNatMore \triangleq \forall i, j \in Nat : i \leq i + j$ BY SMT
THEOREM $ThmNatLess \triangleq \forall i, j \in Nat : i - j \leq i$ BY SMT
THEOREM $ThmNatInc \triangleq \forall i \in Nat : i + 1 > i$ BY SMT
THEOREM $ThmNatDotDot \triangleq \forall i, j, k \in Nat : i \leq j \land j \leq k \equiv j \in i \,..\, k$ BY SMT

THEOREM $ThmNatDecZero \triangleq \forall n \in Nat : n > 0 \Rightarrow n - 1 \in Nat$BY SMT

THEOREM $ThmNatAddEq \triangleq \forall i, j, k \in Nat : i + k = j + k \Rightarrow i = j$BY SMT

THEOREM $ThmNatLeqLt \triangleq \forall i, j, k \in Nat : i \leq j \land j < k \Rightarrow i < k$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 $q \in Seq(S)$
  PROVE
    $q = [i \in 1 \,..\, Len(q) \mapsto q[i]]$
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 $q \in Seq(S)$,
NEW CONSTANT $x \in S$
PROVE
$Len(Append(q, x)) = Len(q) + 1$
PROOF
OMITTED

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

THEOREM $ThmSeqAppendSubSeq \triangleq$
ASSUME
NEW CONSTANT S,
NEW CONSTANT $q \in Seq(S)$,
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 $q \in Seq(S)$,
    NEW CONSTANT $r \in Seq(S)$,
    $Len(q) \leq Len(r)$,
    $Len(r) \leq Len(q)$,
    $q = SubSeq(r, 1, Len(q))$,
    $r = SubSeq(q, 1, Len(r))$
  PROVE
    $q = r$
PROOF
  $\langle 1 \rangle$ $Len(q) = Len(r)$
    $\langle 2 \rangle$ USE $ThmSeqLenIsNat$
    $\langle 2 \rangle$ QED BY $ThmNatLeqIsAntisymmetric$
  $\langle 1 \rangle$ QED BY $ThmSeqEntireInitialSubSeq$

Initial $SubSeq$ is transitive.

THEOREM $ThmSeqInitialSubSeqIsTransitive \triangleq$
  ASSUME
    NEW CONSTANT S,
    NEW CONSTANT $q \in Seq(S)$,
    NEW CONSTANT $r \in Seq(S)$,
    NEW CONSTANT $s \in Seq(S)$,
    $Len(q) \leq Len(r)$,
    $Len(r) \leq Len(s)$,
    $q = SubSeq(r, 1, Len(q))$,
    $r = SubSeq(s, 1, Len(r))$
  PROVE

$q = SubSeq(s, 1, Len(q))$

PROOF

OMITTED

Sequence append incompatible.

THEOREM $ThmSeqAppendIncompatible \triangleq$

ASSUME

NEW CONSTANT S,

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

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

$q1 = q2 \land s1 = s2$

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
$\langle 1 \rangle$ 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 $ThmPcrEqual \triangleq$
$\forall p, q \in Pcr :$
$\land p.init = q.init$
$\land p.extq = q.extq$
$\Rightarrow p = q$
PROOF
⟨1⟩ TAKE $p, q \in Pcr$
⟨1⟩ HAVE $p.init = q.init \land p.extq = q.extq$
⟨1⟩ USE DEF Pcr

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

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

Anti-collision property.

THEOREM $ThmPcrExtendAnticollision \triangleq$
$\forall p1, p2 \in Pcr, x1, x2 \in Pcrx :$
$PcrExtend(p1, x1) = PcrExtend(p2, x2) \Rightarrow p1 = p2 \land x1 = x2$
PROOF
$\langle 1 \rangle$ TAKE $p1, p2 \in Pcr, x1, x2 \in Pcrx$
$\langle 1 \rangle$ DEFINE $px1 \triangleq PcrExtend(p1, x1)$
$\langle 1 \rangle$ DEFINE $px2 \triangleq PcrExtend(p2, x2)$
$\langle 1 \rangle$ HAVE $px1 = px2$
$\langle 1 \rangle$ USE DEF Pcr
$\langle 1 \rangle$ USE DEF $PcrExtend$
$\langle 1 \rangle$ QED
$\langle 2 \rangle$ $p1.init = p2.init$OBVIOUS
$\langle 2 \rangle$ $p1.extq = p2.extq \land x1 = x2$

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

$\langle 3 \rangle$ DEFINE $p1q \triangleq p1.extq$
$\langle 3 \rangle$ DEFINE $p2q \triangleq p2.extq$
$\langle 3 \rangle$ HIDE DEF $p1q$
$\langle 3 \rangle$ HIDE DEF $p2q$
$\langle 3 \rangle$ $Append(p1q, x1) = Append(p2q, x2)$BY DEF $p1q, p2q$
$\langle 3 \rangle$ $p1q \in Seq(Pcrx)$BY DEF $p1q$
$\langle 3 \rangle$ $p2q \in Seq(Pcrx)$BY DEF $p2q$
$\langle 3 \rangle$ $p1q = p2q \land x1 = x2$BY $ThmSeqAppendAnticollision$
$\langle 3 \rangle$ QED BY DEF $p1q, p2q$
$\langle 2 \rangle$ QED BY $ThmPcrEqual$

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

THEOREM $ThmPcrExtendLeqAnticollision \triangleq$
$\forall p, t \in Pcr, x1, x2 \in Pcrx :$
$PcrLeq(PcrExtend(p, x1), t) \land PcrLeq(PcrExtend(p, x2), t) \Rightarrow x1 = x2$
PROOF
$\langle 1 \rangle$ TAKE $p, t \in Pcr, x1, x2 \in Pcrx$
$\langle 1 \rangle$ HAVE $PcrLeq(PcrExtend(p, x1), t) \land PcrLeq(PcrExtend(p, x2), t)$
$\langle 1 \rangle$ USE DEF Pcr
$\langle 1 \rangle$ USE DEF $PcrExtend$
$\langle 1 \rangle$ USE DEF $PcrLeq$
$\langle 1 \rangle$ DEFINE $qp \triangleq p.extq$
$\langle 1 \rangle$ DEFINE $qt \triangleq t.extq$
$\langle 1 \rangle$ DEFINE $qpx1 \triangleq Append(qp, x1)$
$\langle 1 \rangle$ DEFINE $qpx2 \triangleq Append(qp, x2)$
$\langle 1 \rangle$ $qp \in Seq(Pcrx)$OBVIOUS
$\langle 1 \rangle$ $qt \in Seq(Pcrx)$OBVIOUS
$\langle 1 \rangle$ $Len(qpx1) \leq Len(qt)$OBVIOUS
$\langle 1 \rangle$ $Len(qpx2) \leq Len(qt)$OBVIOUS

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

The prover really needs help to focus its attention.

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

$PcrExtend$ increases the length by 1.

THEOREM $ThmPcrExtendLen1 \triangleq$
$\forall p \in Pcr, x \in Pcrx : PcrLen(PcrExtend(p, x)) = PcrLen(p) + 1$
PROOF
$\langle 1 \rangle$ TAKE $p \in Pcr, x \in Pcrx$
$\langle 1 \rangle$ DEFINE $px \triangleq PcrExtend(p, x)$
$\langle 1 \rangle$ $px \in Pcr$BY $ThmPcrExtendIsPcr$
$\langle 1 \rangle$ USE DEF $PcrLen$
$\langle 1 \rangle$ USE DEF $PcrExtend$
$\langle 1 \rangle$ USE DEF Pcr
$\langle 1 \rangle$ 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
$\langle 1 \rangle$ TAKE $p, q \in Pcr$
$\langle 1 \rangle$ HAVE $PcrLeq(p, q)$
$\langle 1 \rangle$ USE DEF Pcr
$\langle 1 \rangle$ $Len(p.extq) \leq Len(q.extq)$BY DEF $PcrLeq$
$\langle 1 \rangle$ $PcrLen(p) = Len(p.extq)$BY DEF $PcrLen$
$\langle 1 \rangle$ $PcrLen(q) = Len(q.extq)$BY DEF $PcrLen$
$\langle 1 \rangle$ 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 $ThmPcrLeqIsAntisymmetric \triangleq$
  $\forall\, p,\, q \in Pcr : PcrLeq(p,\, q) \land PcrLeq(q,\, p) \Rightarrow p = q$
PROOF
  $\langle 1 \rangle$ TAKE $p,\, q \in Pcr$
  $\langle 1 \rangle 2.$ HAVE $PcrLeq(p,\, q) \land PcrLeq(q,\, p)$
  $\langle 1 \rangle 3.$ $p.init = q.init$
    $\langle 2 \rangle$ 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 $ThmPcrLeqIsTransitive \triangleq$
  $\forall\, p,\, q,\, r \in Pcr : PcrLeq(p,\, q) \land PcrLeq(q,\, r) \Rightarrow PcrLeq(p,\, r)$
PROOF
  $\langle 1 \rangle$ TAKE $p,\, q,\, r \in Pcr$
  $\langle 1 \rangle$ HAVE $PcrLeq(p,\, q) \land PcrLeq(q,\, r)$
  $\langle 1 \rangle$ 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 $ThmSeqLenIsNat$

$\langle 2 \rangle$ QED BY $ThmNatLeqIsTransitive$
$\langle 1 \rangle 3.\ p.extq = SubSeq(r.extq,\ 1,\ Len(p.extq))$
$\langle 2 \rangle$ USE DEF Pcr
$\langle 2 \rangle$ QED BY $ThmSeqInitialSubSeqIsTransitive$
$\langle 1 \rangle$ QED BY $\langle 1 \rangle 1,\ \langle 1 \rangle 2,\ \langle 1 \rangle 3$

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

Proof by contradiction.

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

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
$\langle 1 \rangle$ TAKE $p,\ q \in Pcr,\ x \in Pcrx$
$\langle 1 \rangle$ DEFINE $px \triangleq PcrExtend(p,\ x)$
$\langle 1 \rangle$ HAVE $PcrLeq(px,\ q)$
$\langle 1 \rangle$ $px \in Pcr$ BY $ThmPcrExtendIsPcr$
$\langle 1 \rangle$ $PcrLeq(p,\ px)$ BY $ThmPcrExtendLeq$
$\langle 1 \rangle$ 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
  $\langle 1 \rangle$ QED BY $ThmPcrReachableIfExtend$

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

THEOREM $ThmPcrExtendFromEqOrNotleq \triangleq$
  $\forall p, q \in Pcr, x \in Pcrx :$
    $p = q \lor \neg PcrLeq(p, q) \Rightarrow \neg PcrLeq(PcrExtend(p, x), q)$
PROOF
  $\langle 1 \rangle$ TAKE $p, q \in Pcr, x \in Pcrx$
  $\langle 1 \rangle$ HAVE $p = q \lor \neg PcrLeq(p, q)$
  $\langle 1 \rangle$ CASE $p = q$BY $ThmPcrExtendSelfUnreachable$
  $\langle 1 \rangle$ CASE $\neg PcrLeq(p, q)$BY $ThmPcrReachableIfExtend$
  $\langle 1 \rangle$ 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
  $\langle 1 \rangle$ TAKE $p \in Pcr, x1, x2 \in Pcrx$
  $\langle 1 \rangle$ HAVE $x1 \neq x2$
  $\langle 1 \rangle$ DEFINE $p1 \triangleq PcrExtend(p, x1)$
  $\langle 1 \rangle$ DEFINE $p2 \triangleq PcrExtend(p, x2)$
  $\langle 1 \rangle$1. CASE $\neg PcrLeq(p1, p2)$BY $\langle 1 \rangle 1$
  $\langle 1 \rangle$2. CASE $PcrLeq(p1, p2)$
    $\langle 2 \rangle$ USE $\langle 1 \rangle 2$
    $\langle 2 \rangle$ USE DEF $PcrLeq$
    $\langle 2 \rangle$ USE DEF $PcrExtend$
    $\langle 2 \rangle$ USE DEF Pcr
    $\langle 2 \rangle$ $p1.extq \in Seq(Pcrx)$BY $ThmSeqAppendIsSeq$
    $\langle 2 \rangle$ $p2.extq \in Seq(Pcrx)$BY $ThmSeqAppendIsSeq$
    $\langle 2 \rangle$ $Len(p1.extq) = Len(p2.extq)$
      $\langle 3 \rangle$ $Len(p1.extq) = Len(p.extq) + 1$BY $ThmSeqAppendLen1$
      $\langle 3 \rangle$ $Len(p2.extq) = Len(p.extq) + 1$BY $ThmSeqAppendLen1$
      $\langle 3 \rangle$ QED OBVIOUS
    $\langle 2 \rangle$ $p1.extq = p2.extq$BY $ThmSeqEntireInitialSubSeq$
    $\langle 2 \rangle$ $p1.extq \neq p2.extq$BY $ThmSeqAppendIncompatible$

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

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
$\langle 1 \rangle$ TAKE $p \in Pcr$
$\langle 1 \rangle$ HAVE $PcrHasExtension(p)$
$\langle 1 \rangle$ USE DEF $PcrHasExtension$
$\langle 1 \rangle$ USE DEF $PcrPrior$
$\langle 1 \rangle$ USE DEF $PcrLen$
$\langle 1 \rangle$ USE DEF Pcr
$\langle 1 \rangle$ $PcrPrior(p).extq \in Seq(Pcrx)$
$\langle 2 \rangle$ $Len(p.extq) \in Nat$BY $ThmSeqLenIsNat$
$\langle 2 \rangle$ $Len(p.extq) - 1 \in Nat$BY $ThmNatDecZero$
$\langle 2 \rangle$ $Len(p.extq) - 1 \leq Len(p.extq)$BY $ThmNatLess$
$\langle 2 \rangle$ QED BY $ThmSeqInitialSubSeqIsSeq$
$\langle 1 \rangle$ 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
$\langle 1 \rangle$ TAKE $p \in Pcr$
$\langle 1 \rangle$ HAVE $PcrHasExtension(p)$
$\langle 1 \rangle$ USE DEF $PcrHasExtension$
$\langle 1 \rangle$ USE DEF $PcrPrior$
$\langle 1 \rangle$ USE DEF $PcrLastExtension$
$\langle 1 \rangle$ USE DEF $PcrExtend$
$\langle 1 \rangle$ USE DEF $PcrLen$
$\langle 1 \rangle$ USE DEF Pcr
$\langle 1 \rangle$ DEFINE $p0 \triangleq PcrPrior(p)$
$\langle 1 \rangle$ DEFINE $x \triangleq PcrLastExtension(p)$
$\langle 1 \rangle$ $PcrExtend(p0, x).init = p.init$OBVIOUS
$\langle 1 \rangle$ $PcrExtend(p0, x).extq = p.extq$
$\langle 2 \rangle$ DEFINE $qp \triangleq p.extq$
$\langle 2 \rangle$ SUFFICES
ASSUME
$Len(qp) > 0,$

$qp \in Seq(Pcrx)$

PROVE

$Append(SubSeq(qp, 1, Len(qp) - 1), qp[Len(qp)]) = qp$

OBVIOUS

$\langle 2 \rangle$ HIDE DEF qp

$\langle 2 \rangle$ QED BY $ThmSeqAppendPriorLast$

$\langle 1 \rangle$ QED BY $ThmPcrEqual$

WELL KNOWN PCR VALUES

Value of the application pcr attained by rebooting.

THEOREM $ThmAppRebootIsPcr \triangleq AppReboot \in Pcr$

PROOF

$\langle 1 \rangle$ USE DEF $AppReboot$

$\langle 1 \rangle$ USE DEF $Pcri$

$\langle 1 \rangle$ QED BY $ThmPcrInitIsPcr$

Value of the secure execution mode pcr attained by rebooting.

THEOREM $ThmSemRebootIsPcr \triangleq SemReboot \in Pcr$

PROOF

$\langle 1 \rangle$ USE DEF $SemReboot$

$\langle 1 \rangle$ USE DEF $Pcri$

$\langle 1 \rangle$ 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

$\langle 1 \rangle$ USE DEF $SemProtect$

$\langle 1 \rangle$ USE DEF $Pcri$

$\langle 1 \rangle$ 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

$\langle 1 \rangle$ 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 $ThmNextInvType \triangleq$

$InvType \land [Next]_{vars} \Rightarrow InvType'$

PROOF

$\langle 1 \rangle$ HAVE $InvType \land [Next]_{vars}$

$\langle 1 \rangle$ USE DEF $InvType$

$\langle 1 \rangle$ USE DEF Pc, $PcRecov$, $PcChkpt$

$\langle 1 \rangle$ USE DEF $PcIDLE$

$\langle 1 \rangle$ USE DEF $PcRECOV1$, $PcRECOV2$, $PcRECOV3$

$\langle 1 \rangle$ 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.

$\langle 1 \rangle$ QED

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 $NextProveRevoke$

$\langle 3 \rangle$ $PcrPrior(appPcr) \in Pcr$BY $ThmPcrPriorIsPcr$

$\langle 3 \rangle$ QED BY $ThmPcrExtendIsPcr$DEF $Pcrx$

$\langle 2 \rangle 4.$ CASE $NextReboot$

$\langle 3 \rangle$ USE $NextReboot$

$\langle 3 \rangle$ USE DEF $NextReboot$

$\langle 3 \rangle$ $InvType!appPcr'$ BY $ThmAppRebootIsPcr$

$\langle 3 \rangle$ $InvType!semPcr'$ BY $ThmSemRebootIsPcr$

$\langle 3 \rangle$ $InvType!sealPcr'$BY $ThmSealRebootIsPcr$

$\langle 3 \rangle$ $InvType!chkptts'$BY $ThmNullTsIsTs$

$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 5.$ CASE $NextForgetSealTs$

$\langle 3 \rangle$ USE $NextForgetSealTs$

$\langle 3 \rangle$ USE DEF $NextForgetSealTs$

$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$6. CASE $NextExtendAppPcr$
  $\langle 3 \rangle$ USE $NextExtendAppPcr$
  $\langle 3 \rangle$ USE DEF $NextExtendAppPcr$
  $\langle 3 \rangle$ $appPcr' \in Pcr$BY $ThmPcrExtendIsPcr$DEF $Pcrx$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$7. CASE $NextExtendSemPcr$
  $\langle 3 \rangle$ USE $NextExtendSemPcr$
  $\langle 3 \rangle$ USE DEF $NextExtendSemPcr$
  $\langle 3 \rangle$ $InvType\,!\,semPcr'$BY $ThmPcrExtendIsPcr$DEF $Pcrx$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$8. CASE $NextExtendSealPcr$
  $\langle 3 \rangle$ USE $NextExtendSealPcr$
  $\langle 3 \rangle$ USE DEF $NextExtendSealPcr$
  $\langle 3 \rangle$ $InvType\,!\,sealPcr'$BY $ThmPcrExtendIsPcr$DEF $Pcrx$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$9. CASE $NextIncBootCtr$
  $\langle 3 \rangle$ USE $NextIncBootCtr$
  $\langle 3 \rangle$ USE DEF $NextIncBootCtr$
  $\langle 3 \rangle$ $bootCtr' \in Nat$
    $\langle 4 \rangle$1. $bootCtr + 1 \in Nat$BY SMT
    $\langle 4 \rangle$2. $bootCtr' = bootCtr + 1$OBVIOUS
    $\langle 4 \rangle$ QED BY $\langle 4 \rangle 1$, $\langle 4 \rangle 2$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$10. CASE $NextEnterSemRecov$
  $\langle 3 \rangle$ USE $NextEnterSemRecov$
  $\langle 3 \rangle$ USE DEF $NextEnterSemRecov$
  $\langle 3 \rangle$ $InvType\,!\,semPcr'$ BY $ThmSemProtectIsPcr$
  $\langle 3 \rangle$ $InvType\,!\,pc'$ OBVIOUS
  $\langle 3 \rangle$ $InvType\,!\,chkptts'$OBVIOUS
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$11. CASE $NextSemRecov1WhenCorrect$
  $\langle 3 \rangle$ USE $NextSemRecov1WhenCorrect$
  $\langle 3 \rangle$ USE DEF $NextSemRecov1WhenCorrect$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$12. CASE $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ USE $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ USE DEF $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ $semPcr' \in Pcr$BY $ThmPcrExtendIsPcr$DEF $Pcrx$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$13. CASE $NextSemRecov2$
  $\langle 3 \rangle$ USE $NextSemRecov2$

$\langle 3 \rangle$ USE DEF $NextSemRecov2$
$\langle 3 \rangle$ $nv' \in Nv$
$\langle 4 \rangle 1.$ $nv \in Nv$OBVIOUS
$\langle 4 \rangle 2.$ $nv'.current \in$ BOOLEAN BY DEF Nv
$\langle 4 \rangle 3.$ $nv' = [nv$ EXCEPT $!.current = nv'.current]$OBVIOUS
$\langle 4 \rangle$ QED BY ONLY $\langle 4 \rangle 1$, $\langle 4 \rangle 2$, $\langle 4 \rangle 3$ DEF Nv
$\langle 3 \rangle$ QED OBVIOUS

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

$\langle 2 \rangle 15.$ CASE $NextSealTs$
$\langle 3 \rangle$ USE $NextSealTs$
$\langle 3 \rangle$ USE DEF $NextSealTs$
$\langle 3 \rangle$ $sealPcr' \in Pcr$BY $ThmPcrExtendIsPcr$DEF $Pcrx$
$\langle 3 \rangle$ $tsvalues' \in$ SUBSET Ts
$\langle 4 \rangle 1.$ $tsvalues \in$ SUBSET TsOBVIOUS
$\langle 4 \rangle$ DEFINE $ts \triangleq NextSealTs!:!ts$
$\langle 4 \rangle 2.$ $ts \in Ts$BY DEF Ts, $SignedTs$
$\langle 4 \rangle 3.$ $tsvalues' = tsvalues \cup \{ts\}$OBVIOUS
$\langle 4 \rangle$ QED BY $\langle 4 \rangle 1$, $\langle 4 \rangle 2$, $\langle 4 \rangle 3$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 16.$ CASE $NextEnterSemChkpt$
$\langle 3 \rangle$ USE $NextEnterSemChkpt$
$\langle 3 \rangle$ USE DEF $NextEnterSemChkpt$
$\langle 3 \rangle$ $semPcr' \in Pcr$BY $ThmSemProtectIsPcr$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 17.$ CASE $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ USE $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ USE DEF $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ $chkptts' \in SignedTs$
$\langle 4 \rangle$ USE DEF $EnterSemChkptPredicate$
$\langle 4 \rangle$ USE DEF Ts
$\langle 4 \rangle$ QED OBVIOUS
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 18.$ CASE $NextSemChkpt1WhenIncorrect$
$\langle 3 \rangle$ USE $NextSemChkpt1WhenIncorrect$
$\langle 3 \rangle$ USE DEF $NextSemChkpt1WhenIncorrect$
$\langle 3 \rangle$ $semPcr' \in Pcr$BY $ThmPcrExtendIsPcr$DEF $Pcrx$
$\langle 3 \rangle$ 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 $ThmInvType \triangleq$
$Spec \Rightarrow \Box InvType$
PROOF
⟨1⟩ $Init \Rightarrow InvType$BY $ThmInitInvType$
⟨1⟩ $InvType \land [Next]_{vars} \Rightarrow InvType'$BY $ThmNextInvType$
⟨1⟩ QED

PROOF OF INVARIANT *InvInSemProtect*

It holds in the initial state.

THEOREM *ThmInitInvInSemProtect* $\triangleq$
  $Init \Rightarrow InvInSemProtect$
PROOF
  ⟨1⟩ HAVE *Init*
  ⟨1⟩ *InvType*BY *ThmInitInvType*
  ⟨1⟩ USE DEF *Init*
  ⟨1⟩ USE DEF *InSem*
  ⟨1⟩ 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* $\triangleq$
  $InvInSemProtect \land [Next]_{vars} \Rightarrow InvInSemProtect'$
PROOF
  ⟨1⟩ HAVE $InvInSemProtect \land [Next]_{vars}$
  ⟨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⟩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 *NextSemRecov*1*WhenCorrect*
⟨3⟩ USE *NextSemRecov*1*WhenCorrect*
⟨3⟩ USE DEF *NextSemRecov*1*WhenCorrect*
⟨3⟩ QED OBVIOUS

$\langle 2\rangle 12.$ CASE $NextSemRecov1WhenIncorrect$
  $\langle 3\rangle$ USE $NextSemRecov1WhenIncorrect$
  $\langle 3\rangle$ USE DEF $NextSemRecov1WhenIncorrect$
  $\langle 3\rangle$ USE DEF $PcrExtend$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 13.$ CASE $NextSemRecov2$
  $\langle 3\rangle$ USE $NextSemRecov2$
  $\langle 3\rangle$ USE DEF $NextSemRecov2$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 14.$ CASE $NextSemRecov3$
  $\langle 3\rangle$ USE $NextSemRecov3$
  $\langle 3\rangle$ USE DEF $NextSemRecov3$
  $\langle 3\rangle$ USE DEF $PcrExtend$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 15.$ CASE $NextSealTs$
  $\langle 3\rangle$ USE $NextSealTs$
  $\langle 3\rangle$ USE DEF $NextSealTs$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 16.$ CASE $NextEnterSemChkpt$
  $\langle 3\rangle$ USE $NextEnterSemChkpt$
  $\langle 3\rangle$ USE DEF $NextEnterSemChkpt$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 17.$ CASE $NextSemChkpt1WhenCorrect$
  $\langle 3\rangle$ USE $NextSemChkpt1WhenCorrect$
  $\langle 3\rangle$ USE DEF $NextSemChkpt1WhenCorrect$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 18.$ CASE $NextSemChkpt1WhenIncorrect$
  $\langle 3\rangle$ USE $NextSemChkpt1WhenIncorrect$
  $\langle 3\rangle$ USE DEF $NextSemChkpt1WhenIncorrect$
  $\langle 3\rangle$ USE DEF $PcrExtend$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 19.$ CASE $NextSemChkpt2$
  $\langle 3\rangle$ USE $NextSemChkpt2$
  $\langle 3\rangle$ USE DEF $NextSemChkpt2$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 20.$ CASE $NextSemChkpt3$
  $\langle 3\rangle$ USE $NextSemChkpt3$
  $\langle 3\rangle$ USE DEF $NextSemChkpt3$
  $\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle 21.$ 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⟩ $InvInSemProtect \land [Next]_{vars} \Rightarrow InvInSemProtect'$
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⟩ ¬*InSem* OBVIOUS
⟨1⟩ ¬*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* $\triangleq$
  $InvUnreachableSemProtect \land [Next]_{vars} \Rightarrow InvUnreachableSemProtect'$
PROOF
⟨1⟩ HAVE $InvUnreachableSemProtect \land [Next]_{vars}$
⟨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⟩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

$\langle 2\rangle$11. CASE $NextSemRecov1WhenCorrect$
$\langle 3\rangle$ USE $NextSemRecov1WhenCorrect$
$\langle 3\rangle$ USE DEF $NextSemRecov1WhenCorrect$
$\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle$12. CASE $NextSemRecov1WhenIncorrect$
$\langle 3\rangle$ USE $NextSemRecov1WhenIncorrect$
$\langle 3\rangle$ USE DEF $NextSemRecov1WhenIncorrect$
$\langle 3\rangle$ USE DEF $InvType$
$\langle 3\rangle$ USE DEF $Pcrx$
$\langle 3\rangle$ $\neg PcrLeq(semPcr', SemProtect)$
$\langle 4\rangle$ QED BY $ThmPcrExtendSelfUnreachable$
$\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle$13. CASE $NextSemRecov2$
$\langle 3\rangle$ USE $NextSemRecov2$
$\langle 3\rangle$ USE DEF $NextSemRecov2$
$\langle 3\rangle$ QED OBVIOUS

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

$\langle 2\rangle$15. CASE $NextSealTs$
$\langle 3\rangle$ USE $NextSealTs$
$\langle 3\rangle$ USE DEF $NextSealTs$
$\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle$16. CASE $NextEnterSemChkpt$
$\langle 3\rangle$ USE $NextEnterSemChkpt$
$\langle 3\rangle$ USE DEF $NextEnterSemChkpt$
$\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle$17. CASE $NextSemChkpt1WhenCorrect$
$\langle 3\rangle$ USE $NextSemChkpt1WhenCorrect$
$\langle 3\rangle$ USE DEF $NextSemChkpt1WhenCorrect$
$\langle 3\rangle$ QED OBVIOUS

$\langle 2\rangle$18. CASE $NextSemChkpt1WhenIncorrect$
$\langle 3\rangle$ USE $NextSemChkpt1WhenIncorrect$
$\langle 3\rangle$ USE DEF $NextSemChkpt1WhenIncorrect$
$\langle 3\rangle$ USE DEF $InvType$
$\langle 3\rangle$ 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 $ThmInvUnreachableSemProtect \triangleq$
$Spec \Rightarrow \Box InvUnreachableSemProtect$
PROOF
⟨1⟩ $Init \Rightarrow InvUnreachableSemProtect$BY $ThmInitInvUnreachableSemProtect$
⟨1⟩ $InvUnreachableSemProtect \land [Next]_{vars} \Rightarrow InvUnreachableSemProtect'$
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

$\langle 1 \rangle$ $InvInSemProtect \land InvUnreachableSemProtect \Rightarrow InvNvProtection$
$\langle 2 \rangle$ HAVE $InvInSemProtect \land InvUnreachableSemProtect$
$\langle 2 \rangle$ 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*

It holds in the initial state.

THEOREM $ThmInitInvSignedTsLeqBoot \triangleq$
  $Init \Rightarrow InvSignedTsLeqBoot$
PROOF
  $\langle 1 \rangle$ HAVE $Init$
  $\langle 1 \rangle$ $InvType$BY $ThmInitInvType$
  $\langle 1 \rangle$ USE DEF $Init$
  $\langle 1 \rangle$ USE DEF $InvSignedTsLeqBoot$
  $\langle 1 \rangle$ $InvSignedTsLeqBoot\,!\,goal$
    $\langle 2 \rangle$ TAKE $ts \in tsvalues \cup \{chkptts\}$
    $\langle 2 \rangle$ QED BY $ThmNullTsIsntSignedTs$
  $\langle 1 \rangle$ 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 $ThmNextInvSignedTsLeqBoot \triangleq$
  $InvSignedTsLeqBoot \land [Next]_{vars} \Rightarrow InvSignedTsLeqBoot'$
PROOF
  $\langle 1 \rangle$ HAVE $InvSignedTsLeqBoot \land [Next]_{vars}$
  $\langle 1 \rangle$ USE DEF $InvSignedTsLeqBoot$

  $\langle 1 \rangle$ $InvType'$BY $ThmNextInvType$
  $\langle 1 \rangle$ $InvSignedTsLeqBoot\,!\,goal'$

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

  $\langle 2 \rangle 4.$ CASE $NextReboot$
    $\langle 3 \rangle$ USE $NextReboot$
    $\langle 3 \rangle$ USE DEF $NextReboot$

$\langle 3 \rangle$ QED BY *ThmNullTsIsntSignedTs*

$\langle 2 \rangle$5. CASE *NextForgetSealTs*
$\langle 3 \rangle$ USE *NextForgetSealTs*
$\langle 3 \rangle$ USE DEF *NextForgetSealTs*
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$6. CASE *NextExtendAppPcr*
$\langle 3 \rangle$ USE *NextExtendAppPcr*
$\langle 3 \rangle$ USE DEF *NextExtendAppPcr*
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$7. CASE *NextExtendSemPcr*
$\langle 3 \rangle$ USE *NextExtendSemPcr*
$\langle 3 \rangle$ USE DEF *NextExtendSemPcr*
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$8. CASE *NextExtendSealPcr*
$\langle 3 \rangle$ USE *NextExtendSealPcr*
$\langle 3 \rangle$ USE DEF *NextExtendSealPcr*
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$9. CASE *NextIncBootCtr*
$\langle 3 \rangle$ USE *NextIncBootCtr*
$\langle 3 \rangle$ USE DEF *NextIncBootCtr*
$\langle 3 \rangle$ USE DEF *InvType*
$\langle 3 \rangle$ USE DEF *SignedTs*
$\langle 3 \rangle$ $bootCtr \leq bootCtr + 1$BY *ThmNatMore*
$\langle 3 \rangle$ QED BY *ThmNatLeqIsTransitive*

$\langle 2 \rangle$10. CASE *NextEnterSemRecov*
$\langle 3 \rangle$ USE *NextEnterSemRecov*
$\langle 3 \rangle$ USE DEF *NextEnterSemRecov*
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$11. CASE *NextSemRecov*1 *WhenCorrect*
$\langle 3 \rangle$ USE *NextSemRecov*1 *WhenCorrect*
$\langle 3 \rangle$ USE DEF *NextSemRecov*1 *WhenCorrect*
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$12. CASE *NextSemRecov*1 *WhenIncorrect*
$\langle 3 \rangle$ USE *NextSemRecov*1 *WhenIncorrect*
$\langle 3 \rangle$ USE DEF *NextSemRecov*1 *WhenIncorrect*
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$13. CASE *NextSemRecov*2
$\langle 3 \rangle$ USE *NextSemRecov*2
$\langle 3 \rangle$ USE DEF *NextSemRecov*2
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$14. CASE $NextSemRecov3$
  $\langle 3 \rangle$ USE $NextSemRecov3$
  $\langle 3 \rangle$ USE DEF $NextSemRecov3$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$15. CASE $NextSealTs$
  $\langle 3 \rangle$ USE $NextSealTs$
  $\langle 3 \rangle$ USE DEF $NextSealTs$
  $\langle 3 \rangle$ DEFINE $ts \triangleq NextSealTs ! : ! ts$
  $\langle 3 \rangle$ $ts.bootCtr \leq bootCtr$
    $\langle 4 \rangle$ USE DEF $InvType$
    $\langle 4 \rangle$ USE $ThmNatLeqIsReflexive$
    $\langle 4 \rangle$ QED OBVIOUS
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$16. CASE $NextEnterSemChkpt$
  $\langle 3 \rangle$ USE $NextEnterSemChkpt$
  $\langle 3 \rangle$ USE DEF $NextEnterSemChkpt$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$17. CASE $NextSemChkpt1WhenCorrect$
  $\langle 3 \rangle$ USE $NextSemChkpt1WhenCorrect$
  $\langle 3 \rangle$ USE DEF $NextSemChkpt1WhenCorrect$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$18. CASE $NextSemChkpt1WhenIncorrect$
  $\langle 3 \rangle$ USE $NextSemChkpt1WhenIncorrect$
  $\langle 3 \rangle$ USE DEF $NextSemChkpt1WhenIncorrect$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$19. CASE $NextSemChkpt2$
  $\langle 3 \rangle$ USE $NextSemChkpt2$
  $\langle 3 \rangle$ USE DEF $NextSemChkpt2$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$20. CASE $NextSemChkpt3$
  $\langle 3 \rangle$ USE $NextSemChkpt3$
  $\langle 3 \rangle$ USE DEF $NextSemChkpt3$
  $\langle 3 \rangle$ USE DEF $InvType$
  $\langle 3 \rangle$ USE DEF $SignedTs$
  $\langle 3 \rangle$ $bootCtr \leq bootCtr'$
    $\langle 4 \rangle$ USE $ThmNatLeqIsReflexive$
    $\langle 4 \rangle$ USE $ThmNatMore$
    $\langle 4 \rangle$ QED OBVIOUS
  $\langle 3 \rangle$ USE $ThmNatLeqIsTransitive$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$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 $ThmNextInvUnforgeableSemHappy \triangleq$
$InvUnforgeableSemHappy \land [Next]_{vars} \Rightarrow InvUnforgeableSemHappy'$
PROOF
⟨1⟩ HAVE $InvUnforgeableSemHappy \land [Next]_{vars}$
⟨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 $SemReboot$

$\langle 3 \rangle$ USE DEF $SemProtect$
$\langle 3 \rangle$ USE $AssSemProtect$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 5.$ CASE $NextForgetSealTs$
$\langle 3 \rangle$ USE $NextForgetSealTs$
$\langle 3 \rangle$ USE DEF $NextForgetSealTs$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 6.$ CASE $NextExtendAppPcr$
$\langle 3 \rangle$ USE $NextExtendAppPcr$
$\langle 3 \rangle$ USE DEF $NextExtendAppPcr$
$\langle 3 \rangle$ QED OBVIOUS

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

$\langle 2 \rangle 8.$ CASE $NextExtendSealPcr$
$\langle 3 \rangle$ USE $NextExtendSealPcr$
$\langle 3 \rangle$ USE DEF $NextExtendSealPcr$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 9.$ CASE $NextIncBootCtr$
$\langle 3 \rangle$ USE $NextIncBootCtr$
$\langle 3 \rangle$ USE DEF $NextIncBootCtr$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 10.$ CASE $NextEnterSemRecov$
$\langle 3 \rangle$ USE $NextEnterSemRecov$
$\langle 3 \rangle$ USE DEF $NextEnterSemRecov$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle 11.$ CASE $NextSemRecov1WhenCorrect$
$\langle 3 \rangle$ USE $NextSemRecov1WhenCorrect$
$\langle 3 \rangle$ USE DEF $NextSemRecov1WhenCorrect$
$\langle 3 \rangle$ QED OBVIOUS

⟨2⟩12. CASE *NextSemRecov1WhenIncorrect*
⟨3⟩ USE *NextSemRecov1WhenIncorrect*
⟨3⟩ USE DEF *NextSemRecov1WhenIncorrect*
⟨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⟩ QED OBVIOUS

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

⟨2⟩17. CASE *NextSemChkpt1WhenCorrect*
⟨3⟩ USE *NextSemChkpt1WhenCorrect*
⟨3⟩ USE DEF *NextSemChkpt1WhenCorrect*
⟨3⟩ QED OBVIOUS

⟨2⟩18. CASE *NextSemChkpt1WhenIncorrect*
⟨3⟩ USE *NextSemChkpt1WhenIncorrect*
⟨3⟩ USE DEF *NextSemChkpt1WhenIncorrect*
⟨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⟩ $semPcr = SemProtect$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 $ThmInvUnforgeableSemHappy \triangleq$
$Spec \Rightarrow \Box InvUnforgeableSemHappy$
PROOF
⟨1⟩ $Init \Rightarrow InvUnforgeableSemHappy$BY $ThmInitInvUnforgeableSemHappy$
⟨1⟩ $InvUnforgeableSemHappy \land [Next]_{vars} \Rightarrow InvUnforgeableSemHappy'$
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 $ThmNextInvUnforgeableSealReboot \triangleq$
  $InvUnforgeableSealReboot \land [Next]_{vars} \Rightarrow InvUnforgeableSealReboot'$
PROOF
  $\langle 1 \rangle$ HAVE $InvUnforgeableSealReboot \land [Next]_{vars}$
  $\langle 1 \rangle$ 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

$\langle 2 \rangle$4. CASE $NextReboot$
  $\langle 3 \rangle$ USE $NextReboot$
  $\langle 3 \rangle$ USE DEF $NextReboot$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$5. CASE $NextForgetSealTs$
  $\langle 3 \rangle$ USE $NextForgetSealTs$
  $\langle 3 \rangle$ USE DEF $NextForgetSealTs$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$6. CASE $NextExtendAppPcr$
  $\langle 3 \rangle$ USE $NextExtendAppPcr$
  $\langle 3 \rangle$ USE DEF $NextExtendAppPcr$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$7. CASE $NextExtendSemPcr$
  $\langle 3 \rangle$ USE $NextExtendSemPcr$
  $\langle 3 \rangle$ USE DEF $NextExtendSemPcr$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$8. CASE $NextExtendSealPcr$
  $\langle 3 \rangle$ USE $NextExtendSealPcr$
  $\langle 3 \rangle$ USE DEF $NextExtendSealPcr$
  $\langle 3 \rangle$ $sealPcr \in Pcr$BY DEF $InvType$
  $\langle 3 \rangle$ $SealReboot \in Pcr$BY $ThmSealRebootIsPcr$
  $\langle 3 \rangle$ QED BY $ThmPcrExtendFromEqOrNotleq$

$\langle 2 \rangle$9. CASE $NextIncBootCtr$
  $\langle 3 \rangle$ USE $NextIncBootCtr$
  $\langle 3 \rangle$ USE DEF $NextIncBootCtr$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$10. CASE $NextEnterSemRecov$
  $\langle 3 \rangle$ USE $NextEnterSemRecov$
  $\langle 3 \rangle$ USE DEF $NextEnterSemRecov$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$11. CASE $NextSemRecov1WhenCorrect$
  $\langle 3 \rangle$ USE $NextSemRecov1WhenCorrect$
  $\langle 3 \rangle$ USE DEF $NextSemRecov1WhenCorrect$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$12. CASE $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ USE $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ USE DEF $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$13. CASE $NextSemRecov2$
  $\langle 3 \rangle$ USE $NextSemRecov2$

$\langle 3 \rangle$ USE DEF $NextSemRecov2$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$14. CASE $NextSemRecov3$
$\langle 3 \rangle$ USE $NextSemRecov3$
$\langle 3 \rangle$ USE DEF $NextSemRecov3$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$15. CASE $NextSealTs$
$\langle 3 \rangle$ USE $NextSealTs$
$\langle 3 \rangle$ USE DEF $NextSealTs$
$\langle 3 \rangle$ USE DEF $Pcrx$
$\langle 3 \rangle$ $sealPcr \in Pcr$BY DEF $InvType$
$\langle 3 \rangle$ $SealReboot \in Pcr$BY $ThmSealRebootIsPcr$
$\langle 3 \rangle$ QED BY $ThmPcrExtendFromEqOrNotleq$

$\langle 2 \rangle$16. CASE $NextEnterSemChkpt$
$\langle 3 \rangle$ USE $NextEnterSemChkpt$
$\langle 3 \rangle$ USE DEF $NextEnterSemChkpt$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$17. CASE $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ USE $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ USE DEF $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$18. CASE $NextSemChkpt1WhenIncorrect$
$\langle 3 \rangle$ USE $NextSemChkpt1WhenIncorrect$
$\langle 3 \rangle$ USE DEF $NextSemChkpt1WhenIncorrect$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$19. CASE $NextSemChkpt2$
$\langle 3 \rangle$ USE $NextSemChkpt2$
$\langle 3 \rangle$ USE DEF $NextSemChkpt2$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$20. CASE $NextSemChkpt3$
$\langle 3 \rangle$ USE $NextSemChkpt3$
$\langle 3 \rangle$ USE DEF $NextSemChkpt3$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$21. CASE $NextSemChkpt4$
$\langle 3 \rangle$ USE $NextSemChkpt4$
$\langle 3 \rangle$ USE DEF $NextSemChkpt4$
$\langle 3 \rangle$ QED OBVIOUS

$\langle 2 \rangle$22. CASE $NextSemChkpt5$
$\langle 3 \rangle$ USE $NextSemChkpt5$
$\langle 3 \rangle$ USE DEF $NextSemChkpt5$

$\langle 3 \rangle$ QED OBVIOUS

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

It is an invariant of the specification.

THEOREM $ThmInvUnforgeableSealReboot \triangleq$
$Spec \Rightarrow \Box InvUnforgeableSealReboot$
PROOF
$\langle 1 \rangle$ $Init \Rightarrow InvUnforgeableSealReboot$BY $ThmInitInvUnforgeableSealReboot$
$\langle 1 \rangle$ $InvUnforgeableSealReboot \land [Next]_{vars} \Rightarrow InvUnforgeableSealReboot'$
BY $ThmNextInvUnforgeableSealReboot$
$\langle 1 \rangle$ USE DEF $Spec$
$\langle 1 \rangle$ QED

PROOF OF INVARIANT $InvProperLastExtension$

It holds in the initial state.

THEOREM $ThmInitInvProperLastExtension \triangleq$
$Init \Rightarrow InvProperLastExtension$
PROOF
$\langle 1 \rangle$ HAVE $Init$
$\langle 1 \rangle$ $InvType$BY $ThmInitInvType$
$\langle 1 \rangle$ 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 $ThmNextInvProperLastExtension \triangleq$
$InvProperLastExtension \land [Next]_{vars} \Rightarrow InvProperLastExtension'$

PROOF

$\langle 1 \rangle$ HAVE $InvProperLastExtension \wedge [Next]_{vars}$

$\langle 1 \rangle$ USE DEF $InvProperLastExtension$

$\langle 1 \rangle$ $InvType'$BY $ThmNextInvType$

$\langle 1 \rangle$ $InvProperLastExtension\,!\,goal'$

Wow, this is an easy one.

$\langle 2 \rangle$ USE DEF $vars$

$\langle 2 \rangle$ USE DEF $NextObtainAccess$

$\langle 2 \rangle$ USE DEF $NextProveRevoke$

$\langle 2 \rangle$ USE DEF $NextReboot$

$\langle 2 \rangle$ USE DEF $NextForgetSealTs$

$\langle 2 \rangle$ USE DEF $NextExtendAppPcr$

$\langle 2 \rangle$ USE DEF $NextExtendSemPcr$

$\langle 2 \rangle$ USE DEF $NextExtendSealPcr$

$\langle 2 \rangle$ USE DEF $NextIncBootCtr$

$\langle 2 \rangle$ USE DEF $NextEnterSemRecov$

$\langle 2 \rangle$ USE DEF $NextSemRecov1WhenCorrect$

$\langle 2 \rangle$ USE DEF $NextSemRecov1WhenIncorrect$

$\langle 2 \rangle$ USE DEF $NextSemRecov2$

$\langle 2 \rangle$ USE DEF $NextSemRecov3$

$\langle 2 \rangle$ USE DEF $NextSealTs$

$\langle 2 \rangle$ USE DEF $NextEnterSemChkpt$

$\langle 2 \rangle$ USE DEF $NextSemChkpt1WhenCorrect$

$\langle 2 \rangle$ USE DEF $NextSemChkpt1WhenIncorrect$

$\langle 2 \rangle$ USE DEF $NextSemChkpt2$

$\langle 2 \rangle$ USE DEF $NextSemChkpt3$

$\langle 2 \rangle$ USE DEF $NextSemChkpt4$

$\langle 2 \rangle$ USE DEF $NextSemChkpt5$

$\langle 2 \rangle$ QED BY DEF $Next$

$\langle 1 \rangle$ QED OBVIOUS

It is an invariant of the specification.

THEOREM $ThmInvProperLastExtension \triangleq$

$Spec \Rightarrow \Box InvProperLastExtension$

PROOF

$\langle 1 \rangle$ $Init \Rightarrow InvProperLastExtension$BY $ThmInitInvProperLastExtension$

$\langle 1 \rangle$ $InvProperLastExtension \wedge [Next]_{vars} \Rightarrow InvProperLastExtension'$

BY $ThmNextInvProperLastExtension$

$\langle 1 \rangle$ USE DEF $Spec$

$\langle 1 \rangle$ QED

PROOF OF INVARIANT *InvOneLog*

It holds in the initial state.

THEOREM $ThmInitInvOneLog \triangleq$
  $Init \Rightarrow InvOneLog$
PROOF
  $\langle 1\rangle$ HAVE $Init$
  $\langle 1\rangle$ $InvType$ BY $ThmInitInvType$
  $\langle 1\rangle$ $InvSignedTsLeqBoot$ BY $ThmInitInvSignedTsLeqBoot$
  $\langle 1\rangle$ $InvInSemProtect$ BY $ThmInitInvInSemProtect$
  $\langle 1\rangle$ $InvUnforgeableSemHappy$ BY $ThmInitInvUnforgeableSemHappy$
  $\langle 1\rangle$ $InvUnforgeableSealReboot$ BY $ThmInitInvUnforgeableSealReboot$
  $\langle 1\rangle$ $InvProperLastExtension$ BY $ThmInitInvProperLastExtension$
  $\langle 1\rangle$ USE DEF $Init$
  $\langle 1\rangle$ USE DEF $InitNv$
  $\langle 1\rangle$ USE DEF $PcIDLE$
  $\langle 1\rangle$ USE DEF $PcRECOV1$, $PcRECOV2$, $PcRECOV3$
  $\langle 1\rangle$ USE DEF $PcCHKPT1$, $PcCHKPT2$, $PcCHKPT3$, $PcCHKPT4$, $PcCHKPT5$
  $\langle 1\rangle$ $LogInNv$BY DEF $LogInNv$
  $\langle 1\rangle$ $\neg LogInApp$
    $\langle 2\rangle$ $semPcr \neq SemHappy$
      $\langle 3\rangle$ $semPcr.init \neq SemHappy.init$
        $\langle 4\rangle$ $semPcr.init \neq SemProtect.init$
          $\langle 5\rangle$ USE DEF $SemReboot$
          $\langle 5\rangle$ USE DEF $SemProtect$
          $\langle 5\rangle$ USE DEF $PcrInit$
          $\langle 5\rangle$ QED BY $AssSemProtect$
        $\langle 4\rangle$ USE DEF $SemHappy$
        $\langle 4\rangle$ QED BY DEF $PcrExtend$
      $\langle 3\rangle$ QED BY DEF Pcr
    $\langle 2\rangle$ QED BY DEF $LogInApp$
  $\langle 1\rangle$ $\neg LogInTs \land AllCurrentTs = \{\}$
    $\langle 2\rangle$ $\neg CheckTsIsCurrent(chkptts)$
      $\langle 3\rangle$ USE DEF $CheckTsIsCurrent$
      $\langle 3\rangle$ QED BY $ThmNullTsIsntSignedTs$
    $\langle 2\rangle$ USE DEF $AllCurrentTs$
    $\langle 2\rangle$ QED BY DEF $LogInTs$
  $\langle 1\rangle$ $InvOneLog!goal!obtains$ BY DEF $IsOnLog$
  $\langle 1\rangle$ $InvOneLog!goal!revokes$ BY DEF $IsOnLog$
  $\langle 1\rangle$ $InvVerifiableRevocation$ BY DEF $InvVerifiableRevocation$
  $\langle 1\rangle$ 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 $ThmNextInvOneLog \triangleq$
  $InvOneLog \land [Next]_{vars} \Rightarrow InvOneLog'$
PROOF
  $\langle 1 \rangle$ HAVE $InvOneLog \land [Next]_{vars}$

  $\langle 1 \rangle$ $InvType$ BY DEF $InvOneLog$
  $\langle 1 \rangle$ $InvSignedTsLeqBoot$ BY DEF $InvOneLog$
  $\langle 1 \rangle$ $InvInSemProtect$ BY DEF $InvOneLog$
  $\langle 1 \rangle$ $InvUnforgeableSemHappy$ BY DEF $InvOneLog$
  $\langle 1 \rangle$ $InvUnforgeableSealReboot$ BY DEF $InvOneLog$
  $\langle 1 \rangle$ $InvProperLastExtension$ BY DEF $InvOneLog$

  $\langle 1 \rangle$ $InvType'$ BY $ThmNextInvType$
  $\langle 1 \rangle$ $InvSignedTsLeqBoot'$ BY $ThmNextInvSignedTsLeqBoot$
  $\langle 1 \rangle$ $InvInSemProtect'$ BY $ThmNextInvInSemProtect$
  $\langle 1 \rangle$ $InvUnforgeableSemHappy'$ BY $ThmNextInvUnforgeableSemHappy$
  $\langle 1 \rangle$ $InvUnforgeableSealReboot'$ BY $ThmNextInvUnforgeableSealReboot$
  $\langle 1 \rangle$ $InvProperLastExtension'$ BY $ThmNextInvProperLastExtension$

  $\langle 1 \rangle$ $InvOneLog!goal'$

    $\langle 2 \rangle$ 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 $vars$
      $\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$
      $\langle 3 \rangle$ UNCHANGED $AllCurrentTs$
        $\langle 4 \rangle$ USE DEF $AllCurrentTs$
        $\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
        $\langle 4 \rangle$ QED BY DEF $LogInTs$
      $\langle 3 \rangle$ UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
      $\langle 3 \rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$
      $\langle 3 \rangle$ UNCHANGED $LogInApp$ BY DEF $LogInApp$
      $\langle 3 \rangle$ UNCHANGED $LogInTs$ 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 $InvOneLog$

NextObtainAccess or *NextProveRevoke*

⟨2⟩2. CASE $NextObtainAccess \lor NextProveRevoke$

⟨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
$\land$ *BugObtainAccessNoCheckHappy* $\triangleq$ FALSE
$\land$ *BugObtainAccessNoCheckSeal* $\triangleq$ FALSE

⟨5⟩ $BugObtainAccessNoCheckHappy =$ FALSEBY DEF *BugObtainAccessNoCheckHappy*

⟨5⟩ $BugObtainAccessNoCheckSeal =$ FALSEBY 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
$\land$ *BugProveRevokeNoCheckHappy* $\triangleq$ FALSE
$\land$ *BugProveRevokeNoCheckSeal* $\triangleq$ FALSE

⟨5⟩ $BugProveRevokeNoCheckHappy =$ FALSEBY DEF *BugProveRevokeNoCheckHappy*

⟨5⟩ $BugProveRevokeNoCheckSeal =$ FALSEBY DEF *BugProveRevokeNoCheckSeal*

⟨5⟩ QED BY DEF *LogInApp*

⟨4⟩ QED OBVIOUS

⟨3⟩ $InvOneLog\,!\,goal\,!\,obtains' \land InvOneLog\,!\,goal\,!\,revokes'$

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 $\neg InvVerifiableRevocation'$

$\langle 5 \rangle$ PICK $o \in obtains', r \in revokes' : PcrPrior(o) = PcrPrior(r)$
BY DEF $InvVerifiableRevocation$
$\langle 5 \rangle$ DEFINE $p \triangleq PcrPrior(o)$
$\langle 5 \rangle$ DEFINE $xo \triangleq PcrLastExtension(o)$
$\langle 5 \rangle$ DEFINE $xr \triangleq PcrLastExtension(r)$
$\langle 5 \rangle$ $p \in Pcr$ BY $ThmPcrPriorIsPcr$ DEF $InvType, InvProperLastExtension$
$\langle 5 \rangle$ $xo = PcrxOBTAIN$ BY DEF $InvProperLastExtension$
$\langle 5 \rangle$ $xr = PcrxREVOKE$ BY DEF $InvProperLastExtension$
$\langle 5 \rangle$ $o = PcrExtend(p, xo)$ BY $ThmPcrExtendPriorLast$ DEF $InvType, InvProperLastExtension$
$\langle 5 \rangle$ $r = PcrExtend(p, xr)$ BY $ThmPcrExtendPriorLast$ DEF $InvType, InvProperLastExtension$
$\langle 5 \rangle$ $PcrLeq(o, appPcr')$ BY DEF $IsOnLog, InvOneLog$
$\langle 5 \rangle$ $PcrLeq(r, appPcr')$ BY DEF $IsOnLog, InvOneLog$
$\langle 5 \rangle$ $xo \in Pcrx$ BY DEF $Pcrx$
$\langle 5 \rangle$ $xr \in Pcrx$ BY DEF $Pcrx$
$\langle 5 \rangle$ $xo = xr$

The prover needs a lot of help to focus its attention.

$\langle 6 \rangle$ HIDE DEF p
$\langle 6 \rangle$ HIDE DEF xo
$\langle 6 \rangle$ HIDE DEF xr
$\langle 6 \rangle$ $appPcr' \in Pcr$ BY DEF $InvType$
$\langle 6 \rangle$ QED BY $ThmPcrExtendLeqAnticollision$
$\langle 5 \rangle$ $PcrxOBTAIN \neq PcrxREVOKE$ BY $AssObtainNeqRevoke$
$\langle 5 \rangle$ QED OBVIOUS
$\langle 4 \rangle$ QED OBVIOUS
$\langle 3 \rangle$ QED BY DEF $InvOneLog$

NextReboot

$\langle 2 \rangle$3. CASE $NextReboot$
$\langle 3 \rangle$ USE $NextReboot$
$\langle 3 \rangle$ USE DEF $NextReboot$
$\langle 3 \rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$

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

$\langle 3 \rangle$ $\neg LogInApp'$
$\langle 4 \rangle$ $semPcr' \neq SemHappy$
$\langle 5 \rangle$ $semPcr'.init \neq SemHappy.init$
$\langle 6 \rangle$ $semPcr'.init \neq SemProtect.init$
$\langle 7 \rangle$ USE DEF $SemReboot$
$\langle 7 \rangle$ USE DEF $SemProtect$
$\langle 7 \rangle$ USE DEF $PcrInit$
$\langle 7 \rangle$ QED BY $AssSemProtect$
$\langle 6 \rangle$ USE DEF $SemHappy$
$\langle 6 \rangle$ QED BY DEF $PcrExtend$
$\langle 5 \rangle$ QED BY DEF Pcr
$\langle 4 \rangle$ 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 $LogInApp$

$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$

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

$\langle 3 \rangle$ $LogInTs' \Rightarrow LogInTs$

$\langle 4 \rangle$ HAVE $LogInTs'$

$\langle 4 \rangle$ $AllCurrentTs' \neq \{\}$BY DEF $LogInTs$

$\langle 4 \rangle$ USE DEF $AllCurrentTs$

$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$

$\langle 4 \rangle$ QED BY DEF $LogInTs$

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

$\langle 3 \rangle$ $\forall\, ts1,\ ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$

$\langle 4 \rangle$ TAKE $ts1,\ ts2 \in AllCurrentTs'$

$\langle 4 \rangle$ USE DEF $AllCurrentTs$

$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$

$\langle 4 \rangle$ $ts1 \in AllCurrentTs$OBVIOUS

$\langle 4 \rangle$ $ts2 \in AllCurrentTs$OBVIOUS

$\langle 4 \rangle$ QED BY DEF $InvOneLog$

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

$\langle 3 \rangle$ $CheckTsIsCurrent(chkptts)' \Rightarrow LogInTs'$

$\langle 4 \rangle$ USE DEF $LogInTs$

$\langle 4 \rangle$ USE DEF $AllCurrentTs$

$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$

$\langle 4 \rangle$ QED OBVIOUS

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

$\langle 3 \rangle$ $LogInTs' \Rightarrow$ UNCHANGED $CurrentTsLog$

$\langle 4 \rangle$ HAVE $LogInTs'$

$\langle 4 \rangle$ $\exists\, ts \in AllCurrentTs' : ts \in AllCurrentTs$

$\langle 5 \rangle$ USE DEF $LogInTs$

$\langle 5 \rangle$ USE DEF $AllCurrentTs$

$\langle 5 \rangle$ USE DEF $CheckTsIsCurrent$

$\langle 5 \rangle$ $\forall\, ts \in AllCurrentTs' : ts \in AllCurrentTs$OBVIOUS

$\langle 5 \rangle$ QED OBVIOUS

$\langle 4 \rangle$ $\forall\, ts1 \in AllCurrentTs' :$

$\forall\, ts0 \in AllCurrentTs :$

$ts1.appPcr = ts0.appPcr$

BY DEF $InvOneLog$

$\langle 4 \rangle$ QED BY DEF $CurrentTsLog$

$\langle 3 \rangle$ $InvOneLog!goal!obtains'$ 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 $InvOneLog$

NextExtendAppPcr

$\langle 2 \rangle$5. CASE $NextExtendAppPcr$
$\langle 3 \rangle$ USE $NextExtendAppPcr$
$\langle 3 \rangle$ USE DEF $NextExtendAppPcr$
$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$
$\langle 3 \rangle$ UNCHANGED $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
$\langle 4 \rangle$ QED BY DEF $LogInTs$
$\langle 3 \rangle$ UNCHANGED $CurrentTsLog$BY DEF $CurrentTsLog$
$\langle 3 \rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$
$\langle 3 \rangle$ UNCHANGED $LogInApp$ BY DEF $LogInApp$
$\langle 3 \rangle$ UNCHANGED $LogInTs$
$\langle 4 \rangle$ USE DEF $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
$\langle 4 \rangle$ QED BY DEF $LogInTs$
$\langle 3 \rangle$ $InvVerifiableRevocation'$ BY DEF $InvVerifiableRevocation$, $InvOneLog$
$\langle 3 \rangle$ $InvOneLog\,!\,goal\,!\,obtains' \land InvOneLog\,!\,goal\,!\,revokes'$
$\langle 4 \rangle$ CASE $\neg LogInApp$
$\langle 5 \rangle$ USE DEF $LogInApp$
$\langle 5 \rangle$ USE DEF $IsOnLog$, $InvOneLog$

$NextExtendAppPcr$ is predicated on not being in sem, so none of the sem clauses apply.

$\langle 5 \rangle$ USE DEF $InSem$
$\langle 5 \rangle$ QED OBVIOUS
$\langle 4 \rangle$ CASE $LogInApp$
$\langle 5 \rangle$ 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.

$\langle 5 \rangle$ $\forall\, p \in obtains \cup revokes : LogInApp \Rightarrow PcrLeq(p,\, appPcr')$
$\langle 6 \rangle$ UNCHANGED $(obtains \cup revokes)$OBVIOUS
$\langle 6 \rangle$ TAKE $p \in obtains \cup revokes$
$\langle 6 \rangle$ HAVE $LogInApp$
$\langle 6 \rangle$ $PcrLeq(p,\, appPcr)$BY DEF $IsOnLog$, $InvOneLog$
$\langle 6 \rangle$ $PcrLeq(appPcr,\, appPcr')$BY $ThmPcrExtendLeq$DEF $InvType$
$\langle 6 \rangle$ QED BY $ThmPcrLeqIsTransitive$DEF $InvType$
$\langle 5 \rangle$ QED BY DEF $IsOnLog$, $InvOneLog$
$\langle 4 \rangle$ QED OBVIOUS
$\langle 3 \rangle$ QED BY DEF $InvOneLog$

$NextExtendSemPcr$

$\langle 2 \rangle$6. CASE $NextExtendSemPcr$
$\langle 3 \rangle$ USE $NextExtendSemPcr$
$\langle 3 \rangle$ USE DEF $NextExtendSemPcr$
$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$
$\langle 3 \rangle$ UNCHANGED $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $AllCurrentTs$

$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
$\langle 4 \rangle$ QED BY DEF $LogInTs$
$\langle 3 \rangle$ UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
$\langle 3 \rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$

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

$\langle 3 \rangle$ $\neg LogInApp'$
$\langle 4 \rangle$ $semPcr' \neq SemHappy$
$\langle 5 \rangle$ $semPcr = SemHappy \lor \neg PcrLeq(semPcr, SemHappy)$
BY DEF $InvUnforgeableSemHappy$
$\langle 5 \rangle$ $semPcr \in Pcr$ BY DEF $InvType$
$\langle 5 \rangle$ $SemHappy \in Pcr$ BY $ThmSemHappyIsPcr$
$\langle 5 \rangle$ $\neg PcrLeq(semPcr', SemHappy)$ BY $ThmPcrExtendFromEqOrNotleq$
$\langle 5 \rangle$ QED BY $ThmPcrLeqIsReflexive$
$\langle 4 \rangle$ QED BY DEF $LogInApp$
$\langle 3 \rangle$ UNCHANGED $LogInTs$
$\langle 4 \rangle$ 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 $InvOneLog$

NextExtendSealPcr

$\langle 2 \rangle 7.$ CASE $NextExtendSealPcr$
$\langle 3 \rangle$ USE $NextExtendSealPcr$
$\langle 3 \rangle$ USE DEF $NextExtendSealPcr$
$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$ BY DEF $CheckTsIsCurrent$
$\langle 3 \rangle$ UNCHANGED $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
$\langle 4 \rangle$ QED BY DEF $LogInTs$
$\langle 3 \rangle$ UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
$\langle 3 \rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$

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

$\langle 3 \rangle$ $\neg LogInApp'$
$\langle 4 \rangle$ $sealPcr' \neq SealReboot$
$\langle 5 \rangle$ $sealPcr = SealReboot \lor \neg PcrLeq(sealPcr, SealReboot)$
BY DEF $InvUnforgeableSealReboot$
$\langle 5 \rangle$ $sealPcr \in Pcr$ BY DEF $InvType$
$\langle 5 \rangle$ $SealReboot \in Pcr$ BY $ThmSealRebootIsPcr$
$\langle 5 \rangle$ $\neg PcrLeq(sealPcr', SealReboot)$ BY $ThmPcrExtendFromEqOrNotleq$
$\langle 5 \rangle$ QED BY $ThmPcrLeqIsReflexive$
$\langle 4 \rangle$ QED BY DEF $LogInApp$

$\langle 3 \rangle$ UNCHANGED $LogInTs$
$\langle 4 \rangle$ 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 $InvOneLog$

NextIncBootCtr

$\langle 2 \rangle 8.$ CASE $NextIncBootCtr$
$\langle 3 \rangle$ USE $NextIncBootCtr$
$\langle 3 \rangle$ USE DEF $NextIncBootCtr$
$\langle 3 \rangle$ 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 had been in a seal attestation.

$\langle 3 \rangle$ $\neg LogInTs'$
$\langle 4 \rangle$ USE DEF $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
$\langle 4 \rangle$ $\forall\, ts \in tsvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootCtr'$
$\langle 5 \rangle$ TAKE $ts \in tsvalues' \cup \{chkptts'\}$
$\langle 5 \rangle$ $ts \in tsvalues \cup \{chkptts\}$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 $InvType$
$\langle 5 \rangle$ $bootCtr < bootCtr'$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 $LogInTs$
$\langle 3 \rangle$ USE DEF $InSem$
$\langle 3 \rangle$ $InvOneLog!goal!obtains'$ 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 $InvOneLog$

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⟩ $\neg LogInApp'$
    ⟨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⟩ ¬*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*

$\langle 3 \rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$
$\langle 3 \rangle$ UNCHANGED $LogInApp$ BY DEF $LogInApp$
$\langle 3 \rangle$ UNCHANGED $LogInTs$ BY DEF $LogInTs$

EnterSemRecovPredicate guarantees that the log is in the nv ram.

This depends on $BugRecovNoCheckCur \triangleq$ FALSE

$\langle 3 \rangle$ $LogInNv$
  $\langle 4 \rangle$ USE DEF $EnterSemRecovPredicate$
  $\langle 4 \rangle$ $BugRecovNoCheckCur =$ FALSEBY DEF $BugRecovNoCheckCur$
  $\langle 4 \rangle$ QED BY DEF $LogInNv$

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

This depends on $BugRecovNoCheckApp \triangleq$ FALSE

$\langle 3 \rangle$ $appPcr = nv.appPcr$
  $\langle 4 \rangle$ USE DEF $EnterSemRecovPredicate$
  $\langle 4 \rangle$ $BugRecovNoCheckApp =$ FALSEBY DEF $BugRecovNoCheckApp$
  $\langle 4 \rangle$ QED OBVIOUS
$\langle 3 \rangle$ $InvOneLog!goal!obtains'$ 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 $InvOneLog$

NextSemRecov1WhenIncorrect

$\langle 2 \rangle$11. CASE $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ USE $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ USE DEF $NextSemRecov1WhenIncorrect$
  $\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$
  $\langle 3 \rangle$ UNCHANGED $AllCurrentTs$
    $\langle 4 \rangle$ USE DEF $AllCurrentTs$
    $\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
    $\langle 4 \rangle$ QED BY DEF $LogInTs$
  $\langle 3 \rangle$ UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
  $\langle 3 \rangle$ 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.

  $\langle 3 \rangle$ $\neg LogInApp'$
    $\langle 4 \rangle$ $semPcr' \neq SemHappy$
      $\langle 5 \rangle$ USE DEF $SemHappy$
      $\langle 5 \rangle$ USE DEF $SemProtect$
      $\langle 5 \rangle$ USE DEF $Pcri$
      $\langle 5 \rangle$ USE DEF $Pcrx$
      $\langle 5 \rangle$ USE $ThmPcrInitIsPcr$
      $\langle 5 \rangle$ USE $ThmPcrExtendIsPcr$
      $\langle 5 \rangle$ $semPcr = SemProtect$BY DEF $InvInSemProtect, InSem$
      $\langle 5 \rangle$ USE $AssSemHappy$
      $\langle 5 \rangle$ QED BY $ThmPcrExtendAnticollision$

$\langle 4 \rangle$ QED BY DEF *LogInApp*
$\langle 3 \rangle$ UNCHANGED *LogInTs*
$\langle 4 \rangle$ 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*

*NextSemRecov*2

$\langle 2 \rangle$12. CASE *NextSemRecov*2
$\langle 3 \rangle$ USE *NextSemRecov*2
$\langle 3 \rangle$ USE DEF *NextSemRecov*2
$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF *CheckTsIsCurrent*
$\langle 3 \rangle$ UNCHANGED *AllCurrentTs*
$\langle 4 \rangle$ USE DEF *AllCurrentTs*
$\langle 4 \rangle$ USE DEF *CheckTsIsCurrent*
$\langle 4 \rangle$ QED BY DEF *LogInTs*
$\langle 3 \rangle$ UNCHANGED *CurrentTsLog*BY DEF *CurrentTsLog*

Clearing nv current erases the log from the nv ram.

This depends on $BugRecovNoClrCur \triangleq$ FALSE

$\langle 3 \rangle$ $\neg LogInNv'$
$\langle 4 \rangle$ $\neg nv'.current$
$\langle 5 \rangle$ $BugRecovNoClrCur =$ FALSEBY DEF *BugRecovNoClrCur*
$\langle 5 \rangle$ QED BY DEF *InvType*, *Nv*
$\langle 4 \rangle$ QED BY DEF *LogInNv*
$\langle 3 \rangle$ UNCHANGED *LogInApp* BY DEF *LogInApp*
$\langle 3 \rangle$ UNCHANGED *LogInTs* 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*

*NextSemRecov*3

$\langle 2 \rangle$13. CASE *NextSemRecov*3
$\langle 3 \rangle$ USE *NextSemRecov*3
$\langle 3 \rangle$ USE DEF *NextSemRecov*3
$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF *CheckTsIsCurrent*
$\langle 3 \rangle$ UNCHANGED *AllCurrentTs*
$\langle 4 \rangle$ USE DEF *AllCurrentTs*
$\langle 4 \rangle$ USE DEF *CheckTsIsCurrent*

$\langle 4 \rangle$ QED BY DEF $LogInTs$
$\langle 3 \rangle$ UNCHANGED $CurrentTsLog$ BY DEF $CurrentTsLog$
$\langle 3 \rangle$ 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.

$\langle 3 \rangle$ $LogInApp' \in$ BOOLEAN BY DEF $LogInApp$
$\langle 3 \rangle$ UNCHANGED $LogInTs$ 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 $InvOneLog$

$NextSealTs$

$\langle 2 \rangle$14. CASE $NextSealTs$
$\langle 3 \rangle$ USE $NextSealTs$
$\langle 3 \rangle$ USE DEF $NextSealTs$
$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$
$\langle 3 \rangle$ 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

$\langle 3 \rangle$ $\neg LogInApp'$
$\langle 4 \rangle$ $sealPcr' \neq SealReboot$
$\langle 5 \rangle$ $sealPcr = SealReboot \lor \neg PcrLeq(sealPcr, SealReboot)$
BY DEF $InvUnforgeableSealReboot$
$\langle 5 \rangle$ $sealPcr \in Pcr$BY DEF $InvType$
$\langle 5 \rangle$ $SealReboot \in Pcr$BY $ThmSealRebootIsPcr$
$\langle 5 \rangle$ $\neg PcrLeq(sealPcr', SealReboot)$
$\langle 6 \rangle$ $sealPcr' = PcrExtend(sealPcr, PcrxSEAL)$
$\langle 7 \rangle$ $BugSealNoExt =$ FALSEBY DEF $BugSealNoExt$
$\langle 7 \rangle$ QED OBVIOUS
$\langle 6 \rangle$ USE DEF $Pcrx$
$\langle 6 \rangle$ QED BY $ThmPcrExtendFromEqOrNotleq$
$\langle 5 \rangle$ QED BY $ThmPcrLeqIsReflexive$
$\langle 4 \rangle$ QED BY DEF $LogInApp$
$\langle 3 \rangle$ $InvVerifiableRevocation'$ 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 $LogSummaryInTs$.

$\langle 3 \rangle$ CASE $\neg LogInApp$
$\langle 4 \rangle$ UNCHANGED $AllCurrentTs$
$\langle 5 \rangle$ USE DEF $AllCurrentTs$
$\langle 5 \rangle$ 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.

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

$\langle 3 \rangle$ 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.

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

$NextEnterSemChkpt$

$\langle 2\rangle$15. CASE $NextEnterSemChkpt$

$\langle 3\rangle$ USE $NextEnterSemChkpt$

$\langle 3\rangle$ USE DEF $NextEnterSemChkpt$

$\langle 3\rangle$ USE DEF $InSem$

$\langle 3\rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$

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

$\langle 3\rangle$ $\neg LogInApp'$

$\langle 4\rangle$ $semPcr' \neq SemHappy$

$\langle 5\rangle$ USE DEF $SemHappy$

$\langle 5\rangle$ USE DEF $SemProtect$

$\langle 5\rangle$ USE DEF $Pcri$

$\langle 5\rangle$ USE DEF $Pcrx$

$\langle 5\rangle$ USE $ThmPcrInitIsPcr$

$\langle 5\rangle$ USE $ThmPcrExtendIsPcr$

$\langle 5\rangle$ $PcrLeq(SemProtect, SemHappy)$BY $ThmPcrExtendLeq$

$\langle 5\rangle$ $\neg PcrLeq(SemHappy, SemProtect)$BY $ThmPcrExtendSelfUnreachable$

$\langle 5\rangle$ QED BY $ThmPcrLeqIsAntisymmetric$

$\langle 4\rangle$ QED BY DEF $LogInApp$

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

$\langle 3\rangle$ $LogInTs' \Rightarrow LogInTs$

$\langle 4\rangle$ HAVE $LogInTs'$

$\langle 4\rangle$ $AllCurrentTs' \neq \{\}$BY DEF $LogInTs$

$\langle 4\rangle$ USE DEF $AllCurrentTs$

$\langle 4\rangle$ USE DEF $CheckTsIsCurrent$

$\langle 4\rangle$ $chkptts' \in tsvalues'$OBVIOUS

$\langle 4\rangle$ $\exists\, ts \in tsvalues' : CheckTsIsCurrent(ts)$OBVIOUS

$\langle 4\rangle$ QED BY DEF $LogInTs$

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

$\langle 3\rangle$ $\forall\, ts1, ts2 \in AllCurrentTs' : ts1.appPcr = ts2.appPcr$

$\langle 4\rangle$ TAKE $ts1, ts2 \in AllCurrentTs'$

$\langle 4\rangle$ USE DEF $AllCurrentTs$

$\langle 4\rangle$ USE DEF $CheckTsIsCurrent$

$\langle 4\rangle$ $chkptts' \in tsvalues$OBVIOUS

$\langle 4\rangle$ $ts1 \in AllCurrentTs$OBVIOUS

$\langle 4\rangle$ $ts2 \in AllCurrentTs$OBVIOUS

$\langle 4\rangle$ QED BY DEF $InvOneLog$

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

$\langle 3\rangle$ $LogInTs' \Rightarrow$ UNCHANGED $CurrentTsLog$

$\langle 4\rangle$ HAVE $LogInTs'$

$\langle 4\rangle$ $\exists\, ts \in AllCurrentTs' : ts \in AllCurrentTs$

$\langle 5\rangle$ USE DEF $LogInTs$

$\langle 5\rangle$ USE DEF $AllCurrentTs$

$\langle 5\rangle$ USE DEF $CheckTsIsCurrent$

$\langle 5\rangle$ $\forall\, ts \in AllCurrentTs' : ts \in AllCurrentTs$OBVIOUS

$\langle 5 \rangle$ QED OBVIOUS
$\langle 4 \rangle$ $\forall\, ts1 \in AllCurrentTs'$:
$\forall\, ts0 \in AllCurrentTs$:
$ts1.appPcr = ts0.appPcr$
BY DEF $InvOneLog$
$\langle 4 \rangle$ QED BY DEF $CurrentTsLog$
$\langle 3 \rangle$ $InvOneLog!goal!obtains'$ 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 $InvOneLog$

NextSemChkpt1WhenCorrect

$\langle 2 \rangle$16. CASE $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ USE $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ USE DEF $NextSemChkpt1WhenCorrect$
$\langle 3 \rangle$ UNCHANGED $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$
$\langle 3 \rangle$ UNCHANGED $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $AllCurrentTs$
$\langle 4 \rangle$ USE DEF $CheckTsIsCurrent$
$\langle 4 \rangle$ QED BY DEF $LogInTs$
$\langle 3 \rangle$ UNCHANGED $CurrentTsLog$BY DEF $CurrentTsLog$
$\langle 3 \rangle$ UNCHANGED $LogInNv$ BY DEF $LogInNv$
$\langle 3 \rangle$ UNCHANGED $LogInApp$ BY DEF $LogInApp$
$\langle 3 \rangle$ UNCHANGED $LogInTs$ BY DEF $LogInTs$

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

This depends on
$\land\; BugChkptNoCheckTsHappy \triangleq$ FALSE
$\land\; BugChkptNoCheckTsSeal \triangleq$ FALSE
$\land\; BugChkptNoCheckTsCtr \triangleq$ FALSE

$\langle 3 \rangle$ $LogInTs \land CheckTsIsCurrent(chkptts) \land CurrentTsLog = chkptts.appPcr$
$\langle 4 \rangle$ USE DEF $EnterSemChkptPredicate$
$\langle 4 \rangle$ $BugChkptNoCheckTsHappy =$ FALSEBY DEF $BugChkptNoCheckTsHappy$
$\langle 4 \rangle$ $BugChkptNoCheckTsSeal =$ FALSEBY DEF $BugChkptNoCheckTsSeal$
$\langle 4 \rangle$ $BugChkptNoCheckTsCtr =$ FALSEBY DEF $BugChkptNoCheckTsCtr$
$\langle 4 \rangle$ $CheckTsIsCurrent(chkptts)$BY DEF $CheckTsIsCurrent$
$\langle 4 \rangle$ $AllCurrentTs \neq \{\}$BY DEF $AllCurrentTs$
$\langle 4 \rangle$ $CurrentTsLog = chkptts.appPcr$
$\langle 5 \rangle$ $\forall\, ts \in AllCurrentTs : ts.appPcr = chkptts.appPcr$
BY DEF $AllCurrentTs$, $InvOneLog$
$\langle 5 \rangle$ QED BY DEF $CurrentTsLog$
$\langle 4 \rangle$ 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 $InvVerifiableRevocation$, $InvOneLog$

⟨3⟩ QED BY DEF $InvOneLog$

$NextSemChkpt1WhenIncorrect$

⟨2⟩17. CASE $NextSemChkpt1WhenIncorrect$
⟨3⟩ USE $NextSemChkpt1WhenIncorrect$
⟨3⟩ USE DEF $NextSemChkpt1WhenIncorrect$
⟨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 $InvOneLog$

$NextSemChkpt2$

⟨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 $CheckTsIsCurrent$

$\langle 4 \rangle$ QED BY DEF $LogInTs$
$\langle 3 \rangle$ UNCHANGED $CurrentTsLog$BY DEF $CurrentTsLog$
$\langle 3 \rangle$ UNCHANGED $LogInNv$
$\langle 4 \rangle$ USE DEF $InvType$
$\langle 4 \rangle$ USE DEF Nv
$\langle 4 \rangle$ QED BY DEF $LogInNv$
$\langle 3 \rangle$ UNCHANGED $LogInApp$ BY DEF $LogInApp$
$\langle 3 \rangle$ UNCHANGED $LogInTs$ BY DEF $LogInTs$

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

This depends on $BugChkptSaveCurApp \triangleq$ FALSE

$\langle 3 \rangle$ $nv'.appPcr = chkptts.appPcr$
$\langle 4 \rangle$ $BugChkptSaveCurApp =$ FALSEBY DEF $BugChkptSaveCurApp$
$\langle 4 \rangle$ QED BY DEF $InvType, Nv$
$\langle 3 \rangle$ $InvOneLog!goal!obtains'$ 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 $InvOneLog$

$NextSemChkpt3$

$\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 $CheckTsIsCurrent$
$\langle 4 \rangle$ $\forall ts \in tsvalues' \cup \{chkptts'\} : ts \in SignedTs \Rightarrow ts.bootCtr \neq bootCtr'$
$\langle 5 \rangle$ TAKE $ts \in tsvalues' \cup \{chkptts'\}$
$\langle 5 \rangle$ $ts \in tsvalues \cup \{chkptts\}$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 $InvType$
$\langle 5 \rangle$ $bootCtr < bootCtr'$
$\langle 6 \rangle$ $BugChkptNoIncCtr =$ FALSEBY DEF $BugChkptNoIncCtr$
$\langle 6 \rangle$ QED BY $ThmNatInc$
$\langle 5 \rangle$ $ts.bootCtr < bootCtr'$BY $ThmNatLeqLt$
$\langle 5 \rangle$ QED BY $ThmNatLeqXorGt, ThmNatLeqIsReflexive$

⟨4⟩ QED BY DEF $LogInTs$

Erases all current ts seal attestations.

⟨3⟩ $AllCurrentTs' = \{\} \land \neg CheckTsIsCurrent(chkptts)'$
⟨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⟩ $BugChkptNoSetCur =$ FALSEBY 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⟩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* $\triangleq$
$Spec \Rightarrow \Box InvOneLog$
PROOF
⟨1⟩ $Init \Rightarrow InvOneLog$ BY *ThmInitInvOneLog*
⟨1⟩ $InvOneLog \wedge [Next]_{vars} \Rightarrow InvOneLog'$
BY *ThmNextInvOneLog*
⟨1⟩ QED BY *RuleINV*1 DEF *Spec*

PROOF OF INVARIANT $InvAccessUndeniability$

It is an invariant of the specification.

THEOREM $ThmInvAccessUndeniability \triangleq$
  $Spec \Rightarrow \Box InvAccessUndeniability$
PROOF
  $\langle 1 \rangle$ $InvOneLog \Rightarrow InvAccessUndeniability$
    $\langle 2 \rangle$ HAVE $InvOneLog$
    $\langle 2 \rangle$ USE DEF $InvOneLog$
    $\langle 2 \rangle$ USE DEF $InvAccessUndeniability$
    $\langle 2 \rangle$ $BugAuditNoCheckHappy =$ FALSEBY DEF $BugAuditNoCheckHappy$
    $\langle 2 \rangle$ $BugAuditNoCheckSeal =$ FALSEBY DEF $BugAuditNoCheckSeal$
    $\langle 2 \rangle$ CASE $\neg LogInApp$ BY DEF $LogInApp$ unable to audit
    $\langle 2 \rangle$ CASE $LogInApp$ BY DEF $IsOnLog$ audit
    $\langle 2 \rangle$ QED OBVIOUS

  $\langle 1 \rangle$ $Spec \Rightarrow \Box InvOneLog$BY $ThmInvOneLog$
  $\langle 1 \rangle$ QED

PROOF OF INVARIANT $InvVerifiableRevocation$

It is an invariant of the specification.

THEOREM $ThmInvVerifiableRevocation \triangleq$
  $Spec \Rightarrow \Box InvVerifiableRevocation$
PROOF
  $\langle 1 \rangle$ $InvOneLog \Rightarrow InvVerifiableRevocation$
    $\langle 2 \rangle$ HAVE $InvOneLog$
    $\langle 2 \rangle$ USE DEF $InvOneLog$
    $\langle 2 \rangle$ QED OBVIOUS

  $\langle 1 \rangle$ $Spec \Rightarrow \Box InvOneLog$BY $ThmInvOneLog$

$\langle 1 \rangle$ QED