# Continuous Tamper-proof Logging using TPM2.0

Paul England, Limin Jia, James Lorch, and Arunesh Sinha

July 9, 2013

*(Revised June 16, 2014)*

CMU-CyLab-13-008

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

# Continuous Tamper-proof Logging using TPM 2.0

Arunesh Sinha[1], Limin Jia[1], Paul England[2], and Jacob R. Lorch[2]

[1] Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
{aruneshs,liminjia}@cmu.edu
[2] Microsoft Research, Redmond, Washington, USA
{pengland,lorch}@microsoft.com

**Abstract.** Auditing system logs is an important means of ensuring systems' security in situations where run-time security mechanisms are not sufficient to completely prevent potentially malicious activities. A fundamental requirement for reliable auditing is the integrity of the log entries. This paper presents an infrastructure for secure logging that is capable of detecting the tampering of logs by powerful adversaries residing on the device where logs are generated. We rely on novel features of trusted hardware (TPM) to ensure the continuity of the logging infrastructure across power cycles without help from a remote server. Our infrastructure also addresses practical concerns including how to handle high-frequency log updates, how to conserve disk space for storing logs, and how to efficiently verify an arbitrary subset of the log. Importantly, we formally state the tamper-proofness guarantee of our infrastructure and verify that our basic secure logging protocol provides the desired guarantee. To demonstrate that our infrastructure is practical, we implement a prototype and evaluate its performance.

## 1 Introduction

Run-time security mechanisms often are not sufficient to completely prevent malicious activities. Under such circumstances, auditing system logs is an important means of ensuring systems' security. A fundamental requirement for reliable auditing is the integrity of log entries. Adversaries may benefit significantly from tampering with log entries; for instance, malware may erase log entries recording its installation or presence in order to avoid detection and subsequent removal by anti-malware software. Or, an authorized insider may view private customer data in violation of company policy, then remove evidence of his malfeasance from the access log so that audits do not detect it.

There has been much work on developing tamper-proof logging protocols [1–5]. These protocols aim to attest to the integrity of logs as well as detect tampering of logs by the adversary. Some provide tamper-proofness by online commitments of current log state [3]; others store logs in secure memory [4]. Some use the TPM monotonic counter to attest to the integrity of every log entry [2]; others use hash chain based approach [1]. However, these schemes do not meet the stringent requirements for tamper-proof logging in today's computing environment. Next we explain these requirements through a realistic scenario.

Consider a scenario, where the organization, by means of auditing, aims to enforce policies such as, "confidential documents stored on company-owned devices must never be transferred to an external USB storage device." The organization mandates that all employee devices, such as laptops and iPads, run an application that monitors actions relevant to the policy. The logging infrastructure needs to protect audit logs on these devices. Since many of these devices are often offline, the *first requirement* is that the integrity of the audit log is not dependent on continuous connectivity to a central server. Adding a log entry should not require connection to a server. Further, the device could power off, then restart with no connectivity to the network (e.g., the device is turned on during flight). Consequently, a *second requirement* is that the logging infrastructure needs to preserve its continuity across power cycles without contacting a remote server.

It is difficult to segregate security-relevant events from security-irrelevant ones. The logging process is often required to capture a large variety of events from many processes (e.g., OS, browser). Therefore, a *third requirement* is that logging should be fast enough to support high-frequency log updates. Finally, devices have only limited disk space. The *last requirement* is that the logging infrastructure should work with limited disk space for storing logs.

All aforementioned schemes lack at least one of the features required for our application: they lack support for either offline tamper-proofness [3, 5]; or large logs on the order of gigabytes [4]; or continuous logging across power cycles [1, 6]; or high frequency logging [1, 2, 4, 7, 8]. In this paper, we present a logging infrastructure that satisfies all of these requirements. The security guarantee of our logging infrastructure is based on a *forward integrity* adversary model [9], where the adversary can obtain administrative privileges and take complete control of the system. Our infrastructure ensures that the adversary's actions leading up to the action of compromising the machine will be logged and cannot be tampered with, and therefore, can be detected.

Our logging infrastructure is mainly composed of two entities: a logger and a verifier. Initially, the logger and the verifier share a secret key. As the system executes, the logger generates a new key for every new log entry, and uses the key to compute the HMAC of the log entry in order to attest to the log entry's integrity. The key sequence is generated as a *hash chain*; the initial key is known only to the logger and verifier, similarly to the scheme by Schneier et al. [1]. At any given time point, only the key on top of the chain is used; older keys are deleted from memory. When an adversary takes control of the system, it cannot find old keys in memory. The hash-chained key sequence ensures that, without knowledge of the initial key, old keys cannot be derived from the key currently stored in memory. Thus, the adversary cannot produce valid HMACs for earlier log entries.

To allow high-frequency logging, our hash chain is constructed in software, instead of the PCR registers in the TPM. This greatly reduces the time required to append a log entry. Furthermore, we develop mechanisms that allow *truncation* of the log after verification and allow a verifier to efficiently verify any subset of the log. As a result, our infrastructure works with limited disk space.

One of the main novelties of our infrastructure, compared to Schneier et al. [1], lies in leveraging TPM 2.0 features to maintain the continuity and secrecy of the key chain *across a power cycle*. Specifically, we use the ability to seal data to values of a TPM monotonic counter, which TPM1.2 does not allow. At system shutdown, we create a *blob* by sealing the last key before powering off to the value of a TPM monotonic counter. Upon device restarts, the logger can recover the key by unsealing the blob. Our creation of *use once and discard* blobs for logging is a novel use of TPM 2.0's sealing to a monotonic counter feature.

In addition to the design and prototype implementation of our infrastructure, we formally verify the tamper-proofness property of the basic protocol, which we consider as one of our key contributions. We believe this is the first formal proof of security for a logging protocol. The analysis brings out a number of assumptions that the system must satisfy to ensure tamper-proofness.

The rest of the paper is organized as follows. We define the adversary model and review TPM 2.0 features in Section 2. Our logging protocols are presented in Sections 3 and 4. Section 5 details the verification steps of the basic protocol. We describe our prototype implementation and evaluation results in Section 6. Section 7 discusses related work.

## 2 Overview

*Review of TPM2.0* We list features of TPM2.0 that are key to ensuring tamperproofness property of our protocols [10].

**NV memory** TPM 2.0 allows for a larger non-volatile memory than TPM 1.2. Its expected size is more than a megabyte.

**Monotonic NV counter** Any memory slot in NV memory can be tagged as a monotonic counter, which can only be incremented; it starts with a value greater than the maximum of all counters that ever existed in this TPM.

**Enhanced authorizations** TPM 2.0 provides enhanced authorization by defining authorization policies, which can be the conjunctions and disjunctions of basic policies. Basic policies include checking whether an NV memory location stores a specified value and whether a PCR contains a specified value. These authorization policies can be used to implement data sealing.

**Power failure counter** TPM 2.0 has a special 32-bit NV monotonic counter *resetCount* that can be modified by the TPM only. This counter is incremented on a power failure, and thus provides a count of the number of power failures.

*Adversary model* We consider an adversary that controls processes that reside on the same machine as the logging process. We assume that the adversary never controls the hardware, i.e., she cannot snoop on electrical signals, or conduct side-channel attacks by observing physical signals like power consumption. We distinguish between two phases of a system that runs our logging infrastructure. These two phases are separated by the event that the adversary takes control of the machine by gaining root privilege. We assume that in the first phase, the adversary does not have root privileges.

## 3 The Basic Protocol (Protocol A)

In this section, we present our basic protocol (Protocol A), and provide informal arguments for its tamper-proofness.

### 3.1 Protocol Description

Protocol A specifies the behavior of four entities: the logger, the verifier, the TPM, and the OS. We call each entity a role in the protocol. We explain the logger program, as it is the most complex component and uses novel TPM features. We briefly discuss the verifier program, with the detailed program including the OS and TPM presented in Appendix F.

**Logger** The logger uses a sequence of keys ($key(0)$, $key(1)$,...$key(n)$) to produce HMACs of the log data, which arrives sequentially. We annotate each key with the index $i$ of its position in the sequence. The key sequence is a hash chain starting with secret $key(0)$, which is a secret shared between the logger and verifier. The $n^{th}$ key is the hash of the $n{-}1^{th}$ key: $key(n) = hash(key(n-1))$.

The logger has four phases: startup, logging, shutdown, and verification.

*Startup* At machine startup, a sealed key object (sealed blob containing the key) is stored in a designated location *sKeyLoc* on the hard disk. This blob is sealed to the current value of the monotonic TPM counter. Initially, the first sealed key object for $key(0)$ is set up by the administrator. Subsequent sealed key objects are stored by the logger during shutdown.

The logger first acquires locks on its memory locations, the disk location *sKeyLoc* storing the sealed key object, and the disk location *fileLoc* storing the log. These locks prevent any attacker without root privileges from reading from and writing to these locations. They are implemented using mechanisms such as process memory isolation and access control in the file system. On a system restart, these locks are released. Next, the logger unseals the sealed key object to obtain the current key and then increments the TPM counter. At this point, the sealed key object can no longer be unsealed.

*Logging* After startup, the logger, upon receiving new log data, (1) produces an HMAC of the data using the current key $key(k)$, (2) writes the log data and HMAC to disk, (3) generates $key(k+1)$ by computing the hash of the old key $key(k)$, and (4) irretrievably erases the old key from the RAM.

The logger does not use the hash chain feature that TPM offers via PCRs. Instead, it computes the hash in software, which vastly improves the logger's performance, because hashing in memory is much faster than using PCRs.

*Shutdown* Upon receiving a shutdown notification, the logger finishes processing the queue of logs, and then seals the current key to the current monotonic TPM counter value and writes the sealed key object to disk. This phase ensures that when the machine starts up again, there is a sealed key object stored on disk. Protocol A requires the shutdown module of the OS to guarantee that the logger is able to finish its shutdown phase before the machine is powered off.

*Verification* The verification phase is triggered by a verification request from an external verifier; a nonce is sent with such a request. Upon receiving such a request, the logger sends back the log entries (log data and HMACs) stored on disk, and the HMAC of the nonce using the current key.

**Verifier** The verifier initiates the verification phase by sending a nonce along with the verification request to the logger. Upon receiving log entries containing both log data and its HMAC and the HMAC of the nonce using the last key, the verifier checks the HMAC of each log entry and the HMAC of the nonce. The verifier has the initial shared secret and can generate all the keys.

### 3.2 Informal Argument for Tamper-proofness

We explain informally why Protocol A satisfies the tamper-proofness property. Formal analysis of Protocol A is presented in Section 5.

We refer to keys that have smaller indices than the current key used by the logger as old keys. The following two properties hold: (1) an attacker cannot learn the old keys and (2) without the old keys, the attacker cannot tamper with the logs generated prior to the attacker gaining root privilege, i.e., modify entries, remove entries, and truncate the log.

Property (1) holds both before and after the attacker gains root privilege. Before the attacker gains root privilege, the memory and disk locations are properly protected. When the attacker gains root privilege, it has access to all memory and disk locations. However, old keys are not present in the machine's memory as the logger erases these keys upon generation of the next key. The sealed key objects of these old keys cannot be used to extract keys, because these sealed key objects are sealed to past values of the NV monotonic counter of the TPM and there is no way to decrement the counter value. In particular, if the adversary deletes the monotonic counter (by means of his root privilege), then any new monotonic counter will start with the maximum value of all counters that ever existed on the TPM. Finally, the keys form a hash chain, and, therefore, there is no way to generate the old keys directly from the current key. (2) follows directly from (1) and the property of HMACs: without the correct key, an attacker cannot generate valid HMACs that pass the verification. Tamper-proofness follows from (1) and (2).

## 4 Enhanced Protocol (Protocol B)

The basic protocol (Protocol A) has the tamper-proofness property, but is not very practical. Enhanced protocol (Protocol B) uses additional mechanisms to satisfy the following practical requirements. (1) Hard disk space is limited, and, thus, logs need to be periodically truncated. (2) Power failures may not permit the logger's shutdown phase to complete, leading to the loss of the current key. The protocol needs to be able to recover from power failures. (3) For efficient and modular enforcement of several policies, the protocol needs to support verification of an arbitrary subset of the log independently.

### 4.1 New Mechanisms

**Branched key chain** The enhanced protocol evolves keys in a branched manner. Keys are divided into epochs. The initial keys of each epoch form a hash chain staring from $key(0)$. The initial key for epoch $k$ is computed as: $key(k) = hash(key(k-1) \mid {''}epoch{''})$. There is a fixed maximum number ($E$) of keys within an epoch. These keys form another hash chain indexed by the epoch number and a sub-epoch number. The $i^{th}$ sub-epoch key in epoch $k$ ($key(k,i)$) is $hash(key(k,i-1) \mid {''}subepoch{''})$. Here, $key(k,0) = key(k)$.

**Mapping between keys and log entries** To increase the flexibility of the verification and relieve the verifier from the burden of deriving key indices for checking HMACs, we incorporate key index information into the log data. Each log entry now includes the log data, the epoch and sub-epoch indices of the key producing the HMAC, and an HMAC of the log data and the key indices.

### 4.2 Protocol Description

**Logger** The logger in Protocol B cycles through the same phases as in Protocol A. To maintain the branched key chain, the logger starts a new epoch either when the previous epoch is completed or at startup. We first describe the sub-routine that is invoked when a *new epoch* starts. For brevity, we omit the argument of the location of the NV counter from `seal` and `unseal`, as this protocol uses only a fixed monotonic counter. The pseudo code is shown in Figure 1.

*New epoch* In this sub-routine, the sealed object from location *sKeyLoc* is unsealed to obtain the current epoch key. Then, the next epoch key is computed and sealed to the next TPM counter value. Finally, the TPM counter is incremented and the epoch and sub-epoch counters are set appropriately.

A power failure that occurs in the middle of the new epoch routine could create a discrepancy between the TPM monotonic counter value and the value that the sealed blob on disk is sealed to. If the power

```
NewEpoch Sub-routine
$epochkey \leftarrow unseal$(data in $sKeyLoc$)
if $unseal\ fails$ then
    return $fail$;
$nextepochkey \leftarrow hash(epochkey \mid \text{“}epoch\text{”})$
$n \leftarrow$ read TPM counter
$sKeyLoc \leftarrow seal(nextepochkey, n + 1)$
increment TPM counter
$key \leftarrow epochkey$
$epoch \leftarrow n$; $subepoch \leftarrow 0$
```

```
Startup Phase
lock all required memory locations
start newepoch phase
if failure, increment TPM counter
start newepoch phase
if successful, notify the OS
```

```
Logging Phase
while no shutdown notification do
    $data \leftarrow$ get log data
    $logentry \leftarrow (data|epoch|subepoch,$
    $hmac(data|epoch|subepoch, key))$
    increment $subepoch$
    $key \leftarrow hash(key \mid \text{“}subepoch''\text{)}$
    write $logentry$ to disk
    if $subepoch = E$ then
        start newepoch phase
start shutdown phase
```

```
Shutdown Phase
wait for log producer to stop
while message queue is not empty do
    process data as in logging phase
$finaldata \leftarrow hash(key \mid shutdown)$
process $finaldata$ as in logging phase
```

**Fig. 1.** Programs for Protocol B, not including the verifier stage

failure occurs right after the instruction that writes the sealed blob to disk and before the TPM counter is incremented, the TPM counter value will be one step behind the value that the sealed blob is sealed to. The startup phase handles this situation.

*Startup* Similarly to Protocol A, the logger locks critical locations in memory and on disk (and releases them on a restart). Next, it invokes the new epoch sub-routine. Depending on whether the previous power-off is a clean shutdown or a power failure, the sealed blob stored on the hard disk at startup is sealed to either the current value of the monotonic TPM counter or that value incremented by one. If the new epoch sub-routine fails to unseal the key, then TPM counter is incremented and the new epoch sub-routine is called again.

*Logging* The logger computes keys in the branched key chain. It computes a new sub-epoch key for each new log entry until the maximum sub-epoch number is reached. At this point, a new epoch key is computed by invoking the new epoch sub-routine. For each log entry, the logger places the epoch and sub-epoch indices in the log to build an explicit mapping between log entries and keys.

*Shutdown* In the shutdown phase, all remaining log entries are processed. Unlike protocol A, the logger does not create a sealed blob for the current key, as this has been done inside each new epoch sub-routine during logging. Instead, it writes a special log entry $hash(key \mid shutdown)$ to disk indicating the completion of a clean shutdown. The absence of such an entry at machine startup is the evidence of a power failure.

*Verification* The verification phase of the logger is the same as protocol A, except for the deletion of logs after each successful verification and attestation of the *resetCount* value in TPM. The verifier sends a ticket containing encrypted information about how many epochs have been verified. The logger stores this ticket on disk and sends this ticket back to the verifier along with log entries in response to the next verification request.

**Verifier** Differently from protocol A, the verifier starts by asking for the value of *resetCount* to determine if there was a power failure. The verifier additionally generates a ticket attesting to the successful verification up to a check point for the logger. The verifier, after verification till epoch $k$ (the last verified epoch), sends a ticket to the logger stating that the verification till epoch $k$ is successful. The ticket is an encryption: $enc_V(k \mid r)$, where $k$ is the last verified epoch, $r$ is a nonce known only to the verifier and $V$ is the public key of the verifier. The ticket is sent to the verifier in the next verification phase along with log entries from epoch $k + 1$. The verifier uses the information from the ticket sent by the logger to jump to the appropriate epoch key to start the verification.

The verifier's pseudo code is shown in Figure 2. The verifier, upon receiving the log and the ticket, decrypts the ticket to obtain the epoch index. If the ticket is valid, the verifier computes the sub-epoch key and begins verification. In the end, the verifier generates a new ticket and sends it to the logger. It is also

```
Verification Phase
attest to $resetCount$.
$nonce \leftarrow$ recv from verifier
$fhmac \leftarrow hmac(nonce, key)$
$log \leftarrow$ read whole log
$tct \leftarrow$ read ticket
send $(fhm, log, tct)$ to verifier
recv. $(st, tct, del)$ from verifier
if $st = 1$ then
    save $tct$ to disk
    delete epochs till $del$
```

```
Verifier Program
ask for attestation of $resetCount$
$powfail \leftarrow$ current $resetCount \neq$ old $resetCount$
$nonce, r \leftarrow$ generate nonces
send $nonce$ to logger
$(fhm, log, enc_V(n|r')) \leftarrow$ recv from logger
check $r'$ is nonce used in last ticket, halt if not
$epochkey \leftarrow hash^n(sharedsecret|\text{"}epoch''))$
$logavailable \leftarrow$ true; $idx \leftarrow 0$
while $logavailable$ do
    $key \leftarrow epochkey$; $n \leftarrow n + 1$
    for $j$: 0 to $E$-1 do
        if no more log entries then
            $logavailable \leftarrow$ false; break
        verifyhmac$(log(idx)_{data}, log(idx)_{mac}, key)$
        if log entry data is "shutdown" then
            break
        $idx \leftarrow idx + 1$; $key \leftarrow hash(key|\text{"}subepoch'')$
    $epochkey \leftarrow hash(epochkey|\text{"}epoch'')$
verifyhmac$(nonce, fhm, key)$
if all verifyhmac pass and not powfail then
    send $(1, enc_V(n-1|r), n-1)$ to logger
```

**Fig. 2.** Verification stage programs for Protocol B

easy to modify the verifier to verify any subset of the log by making use of the *epoch* and *subepoch* indices contained in each log entry.

## 4.3 Improvements to the Logging Infrastructure

We highlight how the extensions to the protocol address the practical concerns that we summarized at the beginning of this section.

**Rolling logs** Using the ticket, the logger can delete logs up to a verification check point. Instead of sending the entire log starting from the first log entry, the logger only needs to send the ticket for the first $k$ epochs and the log starting from the $k$+1 epoch. To further lower the requirement of disk space for storing the HMACs of logs, it is possible to store a hash of all HMACs in an epoch after the completion of the epoch, instead of storing each HMAC.

**Recovery from power failure** A power failure may prevent the logger from completing the shutdown phase and storing the current key to disk. As a result, the logger in Protocol A has no way of deriving the valid key at the next startup without help from a remote server. The branched key chain used in Protocol B offers a means to recover from such a loss. Dividing the keys into epochs allows the logger to periodically store the sealed blob of the next epoch key to disk without sacrificing performance. Upon rebooting after a power failure, the logger simply increments the TPM counter to retrieve the key from disk.

Portions of the log buffered in memory that are not written to disk due to a power failure are lost. However, a power failure can be detected by the verifier by checking the value of TPM's *resetCount* counter.

**Modular log analysis** With the epoch and sub-epoch indices stored with each log entry, the verifier can request the logger to send portions of the log entries that it wants to verify. One application is enforcing multiple policies on the same system modularly. Each policy analysis can select relevant portions of the log and perform the verification independently; as the verifier can compute the keys based on the *epoch* and *subepoch* information contained in each log entry.

### 4.4 Design Choices and Limitations

**Power attacks** One limitation of our infrastructure is that we cannot distinguish genuine power failures from adversarial system crashes. An attacker can hide malicious activities before the power failure because log entries buffered in memory are lost. Existing logging schemes that use volatile memory for buffering logs [1] or even work in verifiable computation [11] suffer from the same problem. Our choice of using volatile memory for log buffering is driven by the desire to accommodate high-frequency logging. Accesses to non-volatile memory (hard disk or TPM) are slow; thus, it is not feasible to use them to process each log entry. Additional hardware support could mitigate this problem.

**Tradeoffs between performance and security guarantees** Disk write operations are expensive, and, therefore, the bigger the size of the buffered log entry blocks, the more efficient the logger program becomes. However, in case of a power failure, the logger loses the log entries buffered in memory, which may record adversary actions. Consequently, the security guarantee becomes weaker as the block size increases. This problem is mitigated in protocol B by allowing offline recovery from a power failure and detection of the power failure.

Another tradeoff lies in our decision to hash keys in RAM instead of the TPM to accommodate high-frequency log updates. A potential issue is that non-root processes may coerce a root process to write the memory to disk, e.g., by stressing the system memory, and thus leak the keys. Special precaution need to be taken to protect memory regions that store the keys, which we leave for future work.

**Suggested hardware features to defend against power attacks** One way to prevent power attacks is to rely on hardware support to allow for a clean shutdown in spite of a power failure. One possibility is to provide a "fast" memory interface for NV memory of the TPM with assured write on a power failure. The logger uses the NV memory as a buffer instead of the RAM. The logger always maintains an entry composed of a string "power failure" and its HMAC using the current key. This last log entry is never written to disk, except after recovering from a power failure when the TPM NV memory content is flushed to disk. An attacker cannot generate the last entry on its own, so tampering with entries stored in the TPM NV memory can be detected. This scheme requires the logger to compute an additional HMAC for every log entry. However, software HMAC is very fast and is unlikely to be a performance bottleneck.

## 5 Verification

We augment the modeling language and program logic from an existing work [12, 13] and formally prove that Protocol A satisfies the tamper-proofness property. Protocol B uses similar techniques to ensure tamper-proofness, so the verification results of Protocol A can be straightforwardly extended to Protocol B. A detailed formal description of verification is present in the appendix.

**System modeling** We assume the system has a set of principals $\mathcal{P}$ and there is a partial order on the principals: we write $\hat{X} \preceq \hat{Y}$ if $\hat{Y}$ is more privileged than $\hat{X}$, i.e., can access all the resources that $\hat{X}$ can. We write $\widehat{root}$ to denote the root and $\widehat{tpm}$ to denote TPM. $(\mathcal{P}, \preceq)$ is an access control lattice, where the maximal elements are $\widehat{root}$ and $\widehat{tpm}$.

The system is modeled as several components, which we call *threads*, running concurrently. Each thread is owned by a principal. Threads share several common data structures, which include storage (RAM and disk) and read and write locks on storage. The logger, verifier, OS, and TPM are encoded using our modeling language. Other threads (including adversary) in the system are modeled as arbitrary programs interacting with the rest of the system. Their behavior is constrained by our adversary model, which is specified by predicates stating a principal's knowledge based on what it has learned so far. For instance, a principal can compute the HMAC of $d$ using key $k$ if it has both the data and the key. This resembles Dolev-Yao's network adversary.

The behavior of the system is captured by the set of traces generated by all possible interleaving executions of the threads. The security property is specified as a first-order logic formula that holds on every trace of the system. The model and language is discussed in more details in Appendix A.

**Predicates** We define the predicates used in the verification. Action predicates, summarized below, describe the semantics of actions such as read and write, with @ $u$ denoting the time $u$ when the predicate holds.

| | |
|---|---|
| $\mathsf{Read}(i,l,m)@u$ | : thread $i$ reads $m$ from location $l$ |
| $\mathsf{Write}(i,l,m)@u$ | : thread $i$ writes $m$ to location $l$ |
| $\mathsf{Hmac}(i,d,l,m)@u$ | : thread $i$ produces $m = \mathit{hmac}(d,k)$ where key $k$ is stored in location $l$ |
| $\mathsf{VerifyHmac}(i,m,d,k)@u$ | : thread $i$ verifies $m = \mathit{hmac}(d,k)$ |

Other key predicates used in the verification are shown below.

| | |
|---|---|
| $\mathsf{Mem}(l,m)@u$ | : location $l$ has value $m$ |
| $\mathsf{CanRead}(i,l)@u$ | : $i$ can read location $l$ |
| $\mathsf{IsReadLocked}(i,l)@u$ | : thread $i$ holds the read lock of $l$ |
| $\mathsf{HT}(i,\hat{X},e)@u$ | : thread $i$ owned by $\hat{X}$ runs expression $e$ |
| $\mathsf{Has}(i,s)@u$ | : thread $i$ knows $s$ |
| $\mathsf{Owner}(i,\widehat{K})$ | : principal $\widehat{K}$ owns thread $i$ |
| $\mathsf{Contains}(m,m',S)@u$ | : $m'$ can be derived from $m$ using $S$ |
| $\mathsf{MayDerive}(e,e',S)$ | : $e'$ can be derived from $e$ using $S$ |

The $\mathsf{Contains}(m,m',S)$ predicate is true when the term $m'$ can be extracted from $m$ using elements of the set $S$; for instance, $m$ is an encryption of $m'$ using a key $k$ and $S$ contains the key $k$. It is defined with respect to an inductively-defined predicate $\mathsf{MayDerive}$ (details in Appendix C). One example rule is that $\mathsf{MayDerive}(e, \mathit{hash}(e), S)$ is true without any premises: from a term $e$, its hash can always be computed. Predicate $\mathsf{Has}(i,s)$ is true if thread $i$ has the plain text of $s$. It is defined using $\mathsf{Contains}$: $i$ has $s$ if there exists a term $m$ that contains $s$, and thread $i$ receives $m$ or reads $m$ from the storage. These predicates state the assumptions that cryptographic functions are correct and thus capture the adversary's capabilities. We present the details of the logic in Appendix B and detailed definitions of predicates in Appendix C.

**Axioms about actions** Our proof also uses sound axioms specifying the semantics of actions. We show the axioms for generating and verifying HMACs below. Axiom $A_1$ states that on successful verification, it is the case that someone must have produced the HMAC with a key stored in location $l$. Axiom $A_2$ states that if a thread $j$ computes a HMAC using a key $\mathit{key}$ based on location $l$, it must be the case that $j$ can read $l$. Similar to the $\mathsf{Has}$ predicate, these axioms also state assumptions about the correctness of cryptographic functions. Other axioms are listed in Appendix E.

$$\begin{array}{ll} A_1 & \forall i, \mathit{mac}, d, \mathit{key}, u.\ \mathsf{VerifyHmac}(i, \mathit{mac}, d, \mathit{key})\ @\ u \supset \\ & \qquad \exists j, l, u'.(u' < u) \wedge \mathsf{Hmac}(j, d, l, \mathit{mac})\ @\ u' \wedge \mathsf{Mem}(l, \mathit{key})\ @\ u' \\ A_2 & \forall j, \mathit{mac}, d, \mathit{key}, u.\ \mathsf{Hmac}(j, d, l, \mathit{mac})\ @\ u \supset \mathsf{CanRead}(j, l)\ @\ u \end{array}$$

**System assumptions** System assumptions are specified as axioms as well. We define three axioms for this: one specifies the capability of the forward-integrity adversary, one specifies an assumption about the processes running during the logger's startup phase, and one specifies the effect of the access control lattice. We write $u_a$ to denote the time when the adversary gains root privilege.

The following axiom specifies that before time $u_a$, processes owned by $\widehat{root}$ are well-behaved and do not interfere with the logger. Predicate $\mathsf{RW}(i,L)@u$ is true if thread $i$ reads from or writes to any location in the set $L$ at time $u$. The axiom states that processes owned by $\widehat{root}$ do not access any of the storage locations owned by the logger (specified as $\mathsf{LoggerLoc}$), and only threads running with the privilege of the OS can access locations shared between the OS and the logger (specified as $\mathsf{LoggerOSLoc}$).

$$\begin{array}{ll} A_{adv} & = \forall u \leq u_a.\ \mathsf{NoAdv}(u) \\ \mathsf{NoAdv}(u) & = \forall i.\ \mathsf{Owner}(i, \widehat{root})\ @\ u \supset (\forall L.\ \mathsf{LoggerLoc}(L) \supset \neg\mathsf{RW}(i, L)\ @\ u) \\ & \quad \wedge \big(\forall L.\mathsf{LoggerOSLoc}(L) \wedge \mathsf{RW}(i, L)\ @\ u \supset \mathsf{HT}(i, \widehat{root}, \mathtt{OS})\ @\ u\big) \end{array}$$

Axiom $A_{NR}$ states that before the machine is compromised, after any reset, no thread reads and unseals the sealed key before the logger increments the TPM counter. Predicate $\mathsf{Early}(u)$ is true if $u$ is a time point between a reset and the logger incrementing the TPM counter and there are no other resets or counter incrementing operations between them.

$$A_{NR} = \forall u, i, m.\ \mathsf{Early}(u) \wedge (u \leq u_a) \wedge \neg\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u \\ \supset \neg\mathsf{Read}(i, M.disk.sKeyLoc, m) \mathbin{@} u$$

This may seem to be a strong assumption; however, verifying that it holds on a real system is feasible. We discuss this further at the end of this section.

The protection provided by the access-control lattice to guard sensitive operations is captured using axioms similar to the one shown below, which specifies the effects of the lattice on memory read accesses.

$$A_{RDLattice} = \forall i, j, u, l, I, K.\ \mathsf{IsReadLocked}(i, l) \mathbin{@} u \wedge \mathsf{Owner}(i, I) \wedge I \prec K \wedge \\ \mathsf{Owner}(j, K) \supset \mathsf{CanRead}(j, l) \mathbin{@} u$$

If a location $l$ is locked by a thread $i$ owned by principal $I$, then any thread $j$ owned by a principal $K$ higher than $I$ on the access control lattice can read $l$.

**Verification goal** We define an auxiliary predicate $\mathsf{LastLogIdx}(k, u, u_{end})$ to state that before time $u_{end}$, the last log entry the logger writes is indexed by $k$, and written at time $u$. We write $\gamma$ to denote the context containing all the axioms introduced so far. The main result of our verification is a derivation of the following judgment:

$$\gamma \vdash \forall k, k', u_b, u_e, u_l, u_r, u_w, i, j, log, n, fhm.\ \ \mathsf{HT}(i, \hat{V}, \mathtt{VERIFIER}) \text{ on } [u_b, u_e] \wedge \\ (u_b < u_c < u_r < u_v < u_e) \wedge \mathsf{Send}(i, \mathit{VERIFY}) \mathbin{@} u_b \wedge \mathsf{New}(i, nonce) \mathbin{@} u_c \wedge \\ \mathsf{Recv}(i, (log[n], n, fhm)) \mathbin{@} u_r \wedge \mathsf{VerifyHmac}(i, fhm, nonce, key(n+1)) \mathbin{@} u_v \wedge \\ \big((u_r \leq u_a) \supset \mathsf{LastLogIdx}(k, u_l, u_r)\big) \wedge \big((u_r > u_a) \supset \mathsf{LastLogIdx}(k, u_l, u_a)\big) \wedge \\ (1 \leq k' \leq k) \wedge (u_l \geq u_w) \wedge \mathsf{Write}(j, fileloc(k'), v) \mathbin{@} u_w \\ \wedge\ \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_w \supset data(v) = data(log(k'))$$

It says that if the verifier completes successfully then for the log data received by the verifier at time $u_r$, the received data at index $k'$ is the same as the log data $v$ that was written to disk by the logger at index $k'$, conditional on the assumption that $k'$ was written to disk by the logger before time $\min(u_r, u_a)$. In other words, the log entries written before the adversary took control at time $u_a$ will not pass verification if they are tampered with. The formula to the right of the $\vdash$ is the formal definition of the tamper-proofness property. The detailed derivation steps are provided in Appendix E. Here we provide an outline of the derivation.

**Derivation steps** The proof of the tamper-proofness property relies on the following four invariants. Predicate $\mathsf{keyOwnerIn}(u)$ states that at time $u$ only the logger, TPM, and verifier have the key. $\mathsf{keyMemIn}(u)$ states that at time $u$ the only locations that may have the key reside in the memory owned by the logger, or the memory shared between the logger and the TPM, or the disk location that contains the sealed key object. Predicate $\mathsf{oldKeyAdv}(u, u_a)$ states that at time $u$, no thread other than the logger, TPM, or verifier has an old key (key used before time $u_a$). Finally, predicate $\mathsf{oldKeyNotInMem}(u, u_a)$ states that at time $u$, no memory location contains an old key (key used before time $u_a$).

1. $\forall u. u \leq u_a \supset \mathsf{keyOwnerIn}(u)$
2. $\forall u. u \leq u_a \supset \mathsf{keyMemIn}(u)$
3. $\forall u. u > u_a \supset \mathsf{oldKeyAdv}(u, u_a)$
4. $\forall u. u > u_a \supset \mathsf{oldKeyNotInMem}(u, u_a)$

The proofs of these invariants use transfinite induction on time; given the invariants hold before time $u$, we prove that they hold at $u$. In particular, we use the program logic to reason about the protocol roles to show that these invariants are maintained when programs belonging to these roles execute in an adversarial environment.

From (1) and (2), we can prove that the adversary does not have access to any valid keys generated before time $u_a$ at any time prior to $u_a$. (3) and (4) imply that, after $u_a$, the adversary cannot obtain keys that were generated prior to time $u_a$. From the above, we can conclude that at no time does the adversary possess keys used by the logger prior to time $u_a$. Then, it can be shown that the adversary cannot produce valid log entries generated before time $u_a$, which is the desired tamper-proofness property.

**Design decisions based on verification** One important system assumption that the tamper-proofness property depends on is $A_{NR}$: at any time (before the adversary gets root access) between the machine startup and the logger startup, no process should read the sealed blob on disk. The fact that the logger starts soon

| Block size | Total time (ms) | Disk time (ms) |
|---|---|---|
| 512 | 5,135 | 2,513 |
| 256 | 6,675 | 4,056 |
| 128 | 11,074 | 8,320 |
| 64 | 15,882 | 12,997 |
| 32 | 29,148 | 25,505 |
| 16 | 53,306 | 49,168 |

**Table 1.** Time to log 100,000 entries with varying block size.

| Log size | #log entries | Verif. time (s) |
|---|---|---|
| 175MB | 1,211,168 | 27 |
| 390MB | 2,684,760 | 61 |
| 736MB | 5,075,958 | 116 |
| 1.48GB | 10,198,014 | 234 |

**Table 2.** Time (in seconds) to verify logs in a serial manner

after machine startup after a reset makes the number of running threads during that period of time small. The remote attestation feature of the TPM can be used to check that $A_{NR}$ holds by verifying the code that runs on system reset. This assumption leads to important design decisions of the logger. For example, to satisfy this assumption, the logger cannot be implemented as a user-level application. It would be extremely difficult to ensure the tamper-proofness property of such a design, because the logger may not be the first user application to start and other user applications starting before the logger cannot be trusted.

Several axioms (e.g., $A_{RDLattice}$) capture the requirements of access-control lattice. For these axioms to be sound in reality, we need to ensure that the implemented access control mechanisms are correct and cannot be compromised by threads not owned by $\widehat{root}$. For instance, we use process isolation to protect logger-owned memory locations.

## 6 Implementation and Evaluation

**Implementation** We implement a prototype logger and verifier based on Protocol B. Our logger application is a user-level Windows service that uses the ETW logging framework of Windows 7 to receive events from applications and log them. Our implementation relies on the assumption that services in Windows are trusted (see the discussion in Section 5). However, we need not trust any user-level application because services start before these applications. We use keys and HMACs of 256 bits and use SHA256 to produce keys. A 64-bit NV memory location is designated as the monotonic counter that keys are sealed to.

We used a 2.8GHz quad core machine with 6GB of RAM. We use a TPM 2.0 simulator that opens two network ports to receive binary TPM commands and return appropriate responses after processing those commands. The TPM simulator is built from the TPM 2.0 specs and models all TPM 2.0 functionality. We also use a C# TPM library that offers an easy interface to the TPM.

The most significant challenge that we faced in the implementation was that high-level languages that use garbage collectors do not usually provide language support for secure erasure of memory objects, because the memory manager (garbage collector) moves objects around. Though C# offers pinning of memory that can be used to securely erase memory, the use of C# libraries that do not pin memory makes securely erasing keys extremely difficult. However, the tamper-proofness property requires secure erasure of the memory objects that store the keys. Hence, we implement an intermediate layer in C such that the current key always lives in the memory of the C process. This C process uses the TPM library to interact with the TPM. Also, to avoid unexpected behavior due to compiler optimizations we used SecureZeroMemory, which is a guaranteed way of setting memory in Microsoft's version of C.

Our implementation relies on the process memory isolation provided by the operating system to implement locks on volatile memory to prevent an attacker from gaining access to the key during startup phase. We rely on user privilege access control to implement locks on disk.

The prototype system was stable across clean shutdowns and power failures.

**Evaluation** Table 1 shows the logger's log-processing time given different block sizes. As the block size grows, the processing time decreases, and so does the percentage of disk time over the total processing time. This shows that the bigger the block size, the more efficient the logging process. However, the system becomes less secure as block size increases; the attacker has a better chance of hiding its activities in buffered logs that will be discarded after a power failure.

Our storage overhead for HMACs for approximately 32 million log entries is 1 GB. If we store hashes of HMACs in an epoch, then with an $E$ value (sub-epoch number) of 1000, the storage overhead of 32 billion log entries is 1 GB. Thus, the storage overhead of the HMACs is not a bottleneck. Further, with periodic verification, log entries can be removed frequently.

Table 2 shows the evaluation results of the verifier's performance. Verification is reasonably fast, even with simple sequential verification. We expect a huge speed up if the verification process is parallelized using pre-computed keys.

## 7 Related Work

**Secure logging schemes** Auditing has been studied extensively; for example, in the context of detecting misconfiguration in access control policies [14, 15], and in the context of holding agents accountable for their actions [16]. Security guarantees provided by these systems are based on the assumption that logs are tamper-proof.

Most closely related to our approach is work by Kelsey and Schneier [1, 17]. They also use a hash chain of keys to ensure the integrity and confidentiality of logs. Our main improvement over theirs is that we support continuous logging across machine restarts, which they do not. Our protocol allows truncation of logs after verification. As we only care about integrity of log entries, our scheme is much simpler than theirs, and therefore allows for faster log appending operations. They additionally study variants of untrusted verifier, which we do not consider. It is straightforward to extend our protocol using the ideas introduced by Kelsey and Schneier to lift the assumption that the verifier is trusted. Follow-up work [18, 19] does not tackle the issues we address in this paper, and instead, focuses on making the encrypted log searchable [18] or implementing the scheme [19]. Recent work addresses the issue of log deletion required by law [6] and uses similar scheme as [1, 17], but does not work across system restarts and lacks formal verification.

Monotonic counters have been used to ensure the tamper-proofness of logs [2, 8, 7]. They use the monotonic counter inside the attestation of each log entry, whereas we use a software-based hash chain of keys to generate attestations for log entries and only use the counter on system startup/shutdown to ensure the continuity and secrecy of the keys. More concretely, we seal the current key to the counter value using the TPM. The sealed blob is unrecoverable after the counter increments. Thus, we create *use and discard* blobs, which is a novel use of the monotonic counter. Because we do not use the TPM in normal logging activities, our log appending operation is much faster than that in prior work. In the best case, our scheme appends approximately $20,000$ log entries per second using a Intel 2.8GHz processor with 6GB RAM (Table 1), much faster than prior similar schemes [3, 2, 19, 1, 8, 7]. The best of these schemes can process 1750 entries per second using Intel Core 2 Duo 2.4GHz CPU with 4GB of RAM [3]. A2M, another work on secure logging that precedes TrInc, stores logs in trusted memory [4]. Due to the limited size of trusted memory, this scheme is not practical to be used to protect logs on the order of gigabytes, which our work aims to support.

There has been much work on designing efficient data structures for storing logs along with auxiliary information (such as a hash tree) to provide guarantees of tamper-proofness [20–22, 3, 5]. For instance, the work by Crossby et al. [3] provides a dynamic history tree data structure to store the log and capture the history of log insertions through commitments. These structures require publishing the updated state of the auxiliary data structure quite frequently; e.g., after each log addition. However, in our scenario, external communication may not be feasible given the high bandwidth requirement of logs generated at high frequency. While these schemes are effectively online schemes, our scheme provides the forward integrity guarantee in an offline manner, even if the verifier does not verify before the adversary takes control. Also, our infrastructure is able to append logs at much faster rate due to the simplicity of our approach.

**Other schemes that use trusted hardware** TPMs have been used extensively to design schemes that guarantee some form of trust in computing devices, in spite of malicious software running on the device [23, 24, 11]. We use the TPM to protect a key by producing a sealed object of the key that can only be unsealed when the TPM's monotonic counter has a specific value. Incrementing the counter makes the object unsealable by this TPM in the future.

Due to practical constraints, such as size, power consumption and cost, the TPM is limited in its functionality, e.g., small non-volatile memory that degrades with about 100,000 writes. An ideal hardware solution for tamper-proof logging is trusted secure hardware that stores the whole log itself, guaranteeing not only detection of tampering but also recovery of the tampered logs. Other hardware like iButtons [25] has been used in secure logging [19] that implements the scheme of Kelsey and Schneier [1]. While they provide many of the guarantees that our scheme can, they do not address the issue of auditing across power cycles, and their implementation is slow in appending log entries ($\sim$1 second for an append).

The challenge of distinguishing power failures from malicious power attacks that we face is also encountered in another work using the TPM for secure code execution [11]. A trusted power source or a fail-safe power failure mechanism is needed to allow TPMs to shutdown cleanly in case of a power failure.

**Formal verification of system software** Formally verifying the security guarantees of critical system software has become increasingly important. Several projects have demonstrated the value of formal verification (e.g., [26, 13, 27]) . The high-level goal of our work is the same. A model of an adversary against forward integrity was proposed by Bellare et al. [9], which is the same adversary model in our formal verification. As far as we know, we are the first to formally specify the two-phase adversary model and the forward integrity property in logic. Another semi-formal model proposed by Crossby et al. focuses on specifying integrity as prefix consistency of a log and its extension [3], which essentially requires online commitment of log entries. Therefore, that model is not relevant to our logging scenarios. Ma et al. provide a cryptographic style definition of the security properties of a hash chain [28, 29], much like Bellare et al. [9] However, unlike our analysis, they cannot verify security properties of logging protocols, as they lack the logic/language framework to reason about protocols.

Our verification technique is based on the compositional reasoning principles developed by Garg et al. [12] We additionally allow dynamic forking of new threads and resetting the machine, which are essential in modeling and reasoning about the behavior of protocols across machine resets and power failures.

## 8 Conclusion

Our secure logging protocols use new TPM features to guarantee forward integrity of logs in an offline setting and address practical issues such as limited disk space, high-frequency log updates, and unexpected power failures. As future work, we are interested in investigating how to select log block sizes for optimal balance between the strength of the security guarantees and performance. One promising direction is to include an adaptive block size choice module that takes into consideration the costs of security and performance. Another issue we want to explore is the scheduling priority for the logger process, so that other processes are not able to exhaust machine resources with the aim of preventing logging and causing a system crash.

**Acknowledgment** This work was supported in part by the AFOSR MURI on Science of Cybersecurity and NSF CNS-1018061. Part of this work was done while Arunesh Sinha was an intern at Microsoft Research, Redmond.

## References

1. Schneier, B., Kelsey, J.: Cryptographic support for secure logs on untrusted machines. In: USENIX Security. (1998)
2. Levin, D., Douceur, J.R., Lorch, J.R., Moscibroda, T.: Trinc: Small trusted hardware for large distributed systems. In: NSDI. (2009)
3. Crosby, S.A., Wallach, D.S.: Efficient data structures for tamper evident logging. In: USENIX Security. (2009)
4. Chun, B.G., Maniatis, P., Shenker, S., Kubiatowicz, J.: Attested append-only memory: Making adversaries stick to their word. ACM SIGOPS Operating Systems Review **41**(6) (2007) 189–204
5. Snodgrass, R.T., Yao, S.S., Collberg, C.: Tamper detection in audit logs. In: VLDB. (2004)
6. Von Eye, F., Schmitz, D., Hommel, W.: A framework for secure logging with privacy protection and integrity. In: ICIMP. (2014)
7. Sarmenta, L.F.G., van Dijk, M., O'Donnell, C.W., Rhodes, J., Devadas, S.: Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In: ACM STC. (2006)

8. Van Dijk, M., Rhodes, J., Sarmenta, L.F.G., Devadas, S.: Offline untrusted storage with immediate detection of forking and replay attacks. In: ACM STC. (2007)
9. Bellare, M., Yee, B.: Forward integrity for secure audit logs. Technical report, University of California at San Diego (1997)
10. TrustedComputingGroup: TPM library specification. `http://www.trustedcomputinggroup.org/resources/tpm_library_specification`
11. Parno, B., Lorch, J.R., Douceur, J.R., Mickens, J.W., McCune, J.M.: Memoir: Practical state continuity for protected modules. In: IEEE S&P. (2011)
12. Garg, D., Franklin, J., Kaynar, D.K., Datta, A.: Compositional system security with interface-confined adversaries. In: MFPS. (2010)
13. Datta, A., Franklin, J., Garg, D., Kaynar, D.K.: A logic of secure systems and its application to trusted computing. In: IEEE S&P. (2009)
14. Vaughan, J.A., Jia, L., Mazurak, K., Zdancewic, S.: Evidence-based audit. In: CSF. (2008)
15. Bauer, L., Garriss, S., Reiter, M.K.: Detecting and resolving policy misconfigurations in access-control systems. ACM Transactions on Information and System Security **14**(1) (May 2011)
16. Feigenbaum, J., Jaggard, A.D., Wright, R.N.: Towards a formal model of accountability. In: NSPW. (2011)
17. Kelsey, J., Schneier, B.: Minimizing bandwidth for remote access to cryptographically protected audit logs. In: RAID. (1999)
18. Waters, B.R., Balfanz, D., Durfee, G., Smetters, D.K.: Building an encrypted and searchable audit log. In: NDSS. (2004)
19. Chong, C.N., Peng, Z.: Secure audit logging with tamper-resistant hardware. In: IFIP SEC. (2003)
20. Naor, M., Nissim, K.: Certificate revocation and certificate update. In: USENIX Security. (1998)
21. Goodrich, M.T., Tamassia, R., Schwerin, A.: Implementation of an authenticated dictionary with skip lists and commutative hashing. In: DISCEX. (2001)
22. Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S.G.: A general model for authenticated data structures. Algorithmica **39**(1) (2004) 21–41
23. McCune, J.M., Parno, B.J., Perrig, A., Reiter, M.K., Isozaki, H.: Flicker: An execution infrastructure for TCB minimization. ACM SIGOPS Operating Systems Review **42**(4) (2008) 315–328
24. Parno, B., McCune, J.M., Perrig, A.: Bootstrapping trust in commodity computers. In: IEEE S&P. (2010)
25. MaximIntegrated: What is an iButton device? `http://www.maximintegrated.com/products/ibutton/ibuttons/`
26. Jang, D., Tatlock, Z., Lerner, S.: Establishing browser security guarantees through formal shim verification. In: USENIX Security. (2012)
27. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: SOSP. (2009)
28. Ma, D., Tsudik, G.: Forward-secure sequential aggregate authentication. In: IEEE S&P. (2007)
29. Ma, D., Tsudik, G.: A new approach to secure logging. Trans. Storage **5**(1) (March 2009) 2:1–2:21

## Appendix

We describe the formal verification of Protocol A in detail. First, we present the modeling language in Appendix A, then we explain the program logic in Appendix B. Additional predicates used in the verification are defined in Appendix C. The proof of protocol A's tamper-proofness property is presented in Appendix E. Finally, the encoding of the protocol is shown in Appendix F.

## A Language and Model

**Program syntax.** The syntactic constructs for expressions, denoted $e$, are listed below.

$$\begin{array}{ll} \text{Expressions} & e ::= t \mid \texttt{act}\ a \mid \texttt{let}(e_1, x.e_2) \mid \texttt{if}(b, e_1, e_2) \mid \texttt{call}(f, t) \mid \texttt{fork}\ e, t \mid \texttt{reset} \\ \text{Function dec.} & ::= f(x) \stackrel{\triangle}{=} e \end{array}$$

An expression can be a term $t$, which includes the standard binary and unary operations on constants and variables. Expression `act` $a$ denotes an atomic action, such as `read` or `write`. The language itself is parametrized over these actions, which are instantiated based on the application domain. We include memory read and write actions `read`, `write`; message send and receive actions `recv`, `send`; actions for acquiring read and write locks `readlock`, `writelock`; and a polling action `poll` that checks the value of a memory location and only returns if a specific value is stored in that location. Two pairs of actions that are important for ensuring tamper-proofness property are: `hmac`, `verifyhmac` and `seal`, `unseal`. The `hmac` action takes as an argument, a location $l$ containing the key. To successfully compute an HMAC, $l$ must be readable by the thread invoking `hmac`. The seal and unseal actions model corresponding interfaces of TPM 2.0. For simplicity, we only model sealing (unsealing) data to the value of a particular non-volatile TPM counter.

A sequencing expression is denoted $\texttt{let}(e_1, x.e_2)$, where the return result of the first expression is used in the evaluation of the second. An expression can also be an if statement and a function call. We allow a program to fork new threads ($\texttt{fork}\ e, t$), where $e$ is the code of the new thread and $t$ is the principal that the new thread runs as. The expression `reset` resets (reboots) the machine. The last two constructs are our extensions to Garg et al.'s modeling language [12].

Finally, a program is composed of a list of function declarations, and an expression. The programs for Protocol A's roles can be straightforwardly encoded using this language (presented in Appendix F).

**System Modeling Constructs.** The syntactic constructs used to model our logging infrastructure are defined in Figure 3. $(\mathcal{P}, \preceq)$ is an access control lattice, as mentioned in Section 5.

Our model assumes distributed execution. A configuration of the system state, denoted $\mathcal{C}$, is composed of a shared state $\zeta$ and several threads $T_1, \ldots, T_n$ running in parallel. Some of the threads may be controlled by the adversary. A thread $T$ is a tuple of a unique thread ID $I$, an execution stack $K$, and the expression $e$ currently being evaluated. Note that we use thread as a technical term to denote a run-time entity on the machine. It is not the threads used in operating systems. Since there can be many machines in the system, we assume each machine has a unique ID, denoted $m$. The thread ID $I$ is a tuple composed of the principal $\hat{X}$, with whose privilege the thread runs, a thread identifier unique to each machine ($id$), and the machine ID $M$. The execution stack $K$ is a list of frames, denoted $F$, recording the evaluation context. A frame $x.e$ denotes that this frame is waiting for a return result to be plugged into $e$.

Threads share several data structures, including storage (RAM and disk) $\sigma$, read and write locks on storage $(\rho, \omega)$, and a network message queue $Q$. We use $l$ to denote locations in the storage. To simplify our presentation, the locations are hierarchical: e.g., $M.disk$ refers to all locations on the disk of machine $M$, and $M.disk.sealedloc$ refers to a specific location on $M$'s disk. The read and write locks represent access-control policies on the machine. The read (write) lock map, denoted $\rho$ ($\omega$), maps each storage location to the thread holding the lock of that location. A location that is mapped to the special thread $\_$ is not locked.

Threads across different machines communicate to each other via messages. The message queue $Q$ is a buffer of in flight messages.

We assume there is a *permitted* function deciding whether a thread is permitted to perform an action in a given state. This function is used to control accesses to storage locations based on the read and write locks and the access control lattice. It also decides which messages a thread can see.

|  |  |  |  |
|---|---|---|---|
| Principal | $\hat{X}$ | $\in \mathcal{P}$ | $(\mathcal{P}, \preceq)$ is a lattice |
| Machine Id | $M$ |  |  |
| Thread Id | $I$ | $::=$ | $\hat{X}, id, m$ |
| Frame | $F$ | $::=$ | $x.e$ |
| Stack | $K$ | $::=$ | $[] \mid F :: K$ |
| Thread | $T$ | $::=$ | $I\,; K\,; e$ |
| Storage Locations | $l$ |  |  |
| Storage | $\sigma$ | : | Locations $\rightarrow$ Terms |
| Read Lock | $\rho$ | : | Locations $\rightarrow$ Thread ids $\cup \{\_\}$ |
| Write Lock | $\omega$ | : | Locations $\rightarrow$ Thread ids $\cup \{\_\}$ |
| Message | $M$ | $::=$ | *sender*, *receiver*, *content* |
| MessageQueue | $Q$ | $::=$ | $[] \mid M :: Q$ |
| Shared State | $\zeta$ | $::=$ | $\sigma, \rho, \omega, Q$ |
| Configuration | $\mathcal{C}$ | $::=$ | $\zeta \rhd T_1, \ldots, T_n$ |
| Permissions | *permitted* |  |  |
|  |  | : | (Shared States $\times$ Thread Ids $\times$ actions) $\rightarrow$ Boolean |

**Fig. 3.** Modeling constructs

**Reduction rules.** The run-time behavior of the system is captured by the reduction rules, which specify how the system transitions from one configuration to the next, one step at a time. Selected reduction rules novel to this work are shown in Figure 4; the rest of the rules are same as in Garg et al. [12].

We write $\zeta \rhd T \hookrightarrow \zeta' \rhd T'$ to denote the one-step transition of a single thread, and $\mathcal{C} \xrightarrow{u} \mathcal{C}'$ to denote the top-level reduction steps of the system, where $u$ is the time when the transition happens. A trace $\mathcal{T}$ is a sequence of reduction steps: $\mathcal{C}_1 \xrightarrow{u_1} \mathcal{C}_1 \cdots \xrightarrow{u_n} \mathcal{C}_n$.

In rule Red-Act, a thread executes an atomic action. This rule is restricted by the *permitted* function, which serves to allow/disallow actions based on the current state, e.g., disallow reads on read locked memory using the partial order $\preceq$. The system is parametrized over a set of action reduction rules of the form $\zeta, I, a \rightarrow \zeta', I, t$, which evaluates an action $a$ and return the result $t$ and a new state $\zeta'$. These rules are instantiated based on the application domain.

Rule Red-Conf allows arbitrary interleaving of the reductions of individual threads in the configuration. When a thread forks a new thread, rule Red-Conf-Fork picks a fresh *id* and generates a new thread that executes $e$ with an empty stack. Note that a thread can only fork threads owned by principals that are less privileged than itself. The rule Red-Conf-Reset stops all threads, sets all volatile memory to *null*, clears all locks and starts two threads owned by principals $\widehat{root}$ (the OS) and $\widehat{tpm}$.

## B Logic of Programs

We extend the program logic by Garg et al. [12] to accommodate the new language constructs. Similar to prior work, the program logic aims to derive security properties of a system composed of both trusted components, which runs the prescribed protocol; and adversarial components, which runs arbitrary code, but confined to its capabilities as specified in the adversary model.

**Property specifications.** Security properties are specified as first-order logic formulas (shown below). Most constructs are standard. We write $p$ @ $u$ to mean that predicate $p$ is true at time $u$. Similarly, $\varphi$ @ $u$ means that formula $\varphi$ holds at time $u$. We write ($\varphi$ on $I$) to denote that $\varphi$ is true during the time interval $I$. Using these explicit time points, we can concisely specify event-orderings on the trace, which is essential for the formal specification of the tamper-proofness property.

$$\begin{array}{rl} \text{Formulas } \varphi, \psi ::= & p \text{ @ } u \mid b \mid t = t' \mid u_1 \leq u_2 \mid \top \mid \bot \\ \mid & \varphi \text{ @ } u \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \neg\varphi \mid \forall x.\varphi \mid \exists x.\varphi \end{array}$$

$\boxed{\zeta \rhd T \hookrightarrow \zeta' \rhd T'}$

$$\frac{\mathit{permitted}(\zeta, I, a) \qquad \zeta, I, a \to \zeta', I, t}{\zeta \rhd I\,;x.e :: K\,;\mathtt{act}\ a \mapsto \zeta' \rhd I\,;K\,;e[t/x]}\ \textsc{Red-Act}$$

$\boxed{\mathcal{C} \longrightarrow \mathcal{C}'}$

$$\frac{\zeta \rhd T_i \hookrightarrow \zeta' \rhd T_i'}{\zeta \rhd T_1 \mid \ldots \mid T_i \mid \ldots \mid T_n \longrightarrow \zeta' \rhd T_1 \mid \ldots \mid T_i' \mid \ldots \mid T_n}\ \textsc{Red-Conf}$$

$$\frac{m = \mathit{machine}(I) \qquad \mathit{id}\text{ fresh} \qquad \hat{X} \preceq \mathit{owner}(I)}{\begin{array}{c}\zeta \rhd T_1 \mid \ldots \mid I\,;(x.e) :: K\,;\mathtt{fork}\ e', \hat{X} \mid \ldots \mid T_n \longrightarrow \\ \zeta \rhd T_1 \mid \ldots \mid I\,;K\,;e\{J/x\} \mid \ldots \mid T_n \mid (\hat{X}, \mathit{id}, m)\,;.\,;e'\end{array}}\ \textsc{Red-Conf-Fork}$$

$$\frac{\begin{array}{c} I \in \omega(m.\mathit{ram.sh}) \qquad I'.\mathit{id}'\text{ fresh} \\ I''.\mathit{id}''\text{ fresh} \qquad \sigma' = \sigma[(\mathit{locs}(m) - \mathit{locs}(m, \{\mathit{disk}, \mathit{tpm.nv}\}) \mapsto \mathit{null}](\rho', \omega') = (\rho, \omega)[\mathit{locs}(m) \mapsto \_] \\ I' = (\widehat{\mathit{root}}, \mathit{id}', m) \text{ and } I'' = (\widehat{\mathit{tpm}}, \mathit{id}'', m) \qquad m = \mathit{machine}(I)\end{array}}{\zeta \rhd T_1 \mid \ldots \mid I\,;K\,;\mathtt{reset} \mid \ldots \mid T_n \longrightarrow \qquad \zeta' \rhd (T_1 \mid \ldots \mid T_n)\backslash \mathit{Threads}(m) \mid I'\,;.\,;e_{OS} \mid I''\,;.\,;e_{\mathit{tpm}}}\ \textsc{Red-Conf-Reset}$$

**Fig. 4.** Selected reduction rules

| | |
|---|---|
| $\Gamma \vdash [e]\langle u_b, u_e, i, x\rangle\varphi$ | Given a trace that contains the complete evaluation of $e$ between time $U_b$ and $U_e$ by thread $I$ and returns $t$, $\varphi[U_b, U_e, I, t/u_b, u_e, i, x]$ holds on that trace. This trace is allowed to contain other executing threads, including malicious ones. |
| $\Gamma \vdash \{e\}\langle u_b, u_e, i\rangle\varphi$ | Given a trace that contains an incomplete evaluation of $e$ between time $U_b$ and $U_e$ by thread $I$, $\varphi[U_b, U_e, I/u_b, u_e, i]$ holds on the trace. Similarly, this trace may contain attacker threads. |
| $\Gamma \vdash \varphi$ | $\varphi$ holds on any trace that satisfies $\Gamma$ |

**Fig. 5.** Summary of judgments

**Reasoning principals.** The program logic consists of three main judgments, summarized in Figure 5. For clarity of presentation, we omit several contexts from the judgments as they mostly remain unchanged. Context $\Gamma$ contains formulas that are assumed true. The first judgment asserts partial correctness properties of an expression; the second asserts invariant properties of an expression; and the last judgment is the standard first-order logic judgment.

To give a flavor of the rules in our program logic, we show one rule for deriving the effects of completing a `fork` action and one rule deriving properties of systems containing a trusted component.

$$\frac{L \preceq \mathsf{owner}(i) \qquad J\text{ fresh}}{\Gamma \vdash [\mathtt{fork}\ e, L]\langle u_b, u_e, i, x\rangle(x = J) \wedge \mathsf{Start}(J, e)@u_e \wedge \mathsf{Owner}(J, L)\ @\ u_e}\ \mathrm{PK}$$

$$\frac{\Gamma \vdash \{e\}\langle u_b, u_e, i\rangle\varphi(u_b, u_e, i) \qquad \Gamma \vdash \mathsf{Start}((\hat{X}, \mathit{id}, M), e)\ @\ u}{\Gamma \vdash \forall u'.\ (u' > u) \wedge \neg\mathsf{Reset}(M)\text{ on }(u, u'] \supset \varphi(u, u', (\hat{X}, \mathit{id}, M))}\ \textsc{Honth}$$

Rule PK states that `fork` $e, L$ is only allowed if the new thread runs as a principal that has fewer privileges than its parent thread; that the return value is the newly generated thread ID $J$; and that the new thread $J$ starts to execute $e$ when fork finishes. Rule Honth is the key rule transitioning from reasoning about the program of trusted components to deriving a property of the trace. This is similar to the rule of the same name in [12]. Essentially, if a thread $I$ starts $e$ at time $u$, and we know that $e$ has an invariant $\varphi$ (at any time point before $e$ finishes, $\varphi$ is true), then at any later time point $u'$ and the machine has not been reset in between, then the invariant guaranteed by $e$ holds between these two time points. For instance, if the thread invariant states that this thread never reads location $M.\mathit{disk.sealedloc}$, then that invariant holds from the start of the thread till the next reset.

## C Additional predicates

Predicate $\mathsf{MayDerive}(e_1, e_2, S)$ means that $e_2$ can be derived from $e_1$ given the set of terms $S$. We show the new rules below and other rules are the same as those in [13]. The first rule states that from a term $e$, its hash can always be computed. The last two rules state that data sealed within a blob can be extracted if either the TPM counter's value is the same as the value that blob is sealed to, or the TPM's key is known.

$$\frac{}{\mathsf{Mayderive}(e, hash(e), S)} \qquad \frac{tpm.counter = n}{\mathsf{Mayderive}(SEAL_{tpm.key}(t,n), t, S)} \qquad \frac{tpm.key \in S}{\mathsf{Mayderive}(SEAL_{tpm.key}(t,n), t, S)}$$

The predicate $\mathsf{Has}(i, s)@u$ is defined as there exists $m$ such that (1) $m$ contains $s$ given $S$, (2) $i$ has $S$, and (3) the thread $i$ has either received $m$, or read $m$ from the storage, or computed $m$ using the hmac action.

$$\begin{aligned}
\mathsf{Has}(i,s) \mathbin{@} u = \quad & \exists u', u''.\ (u'' \le u' \le u) \wedge \\
& \exists m. \Big( \big(\exists l, j, m'.\ \mathsf{Recv}(i,j,m) \mathbin{@} u'' \vee \mathsf{Read}(i,l,m) \mathbin{@} u'' \vee \mathsf{Hmac}(i,m',l,m) \mathbin{@} u''\big) \wedge \\
& \qquad \big(\exists S.\ \mathsf{HasSet}(i,S) \mathbin{@} u' \wedge \mathsf{Contains}(m,s,S) \mathbin{@} u'\big)\Big) \wedge \\
& \neg\mathsf{Reset}() \text{ on } [u'', u] \\
\mathsf{HasSet}(i,S) \mathbin{@} u = \quad & \forall s.(s \in S) \supset \mathsf{Has}(i,s) \mathbin{@} u
\end{aligned}$$

The predicate $\mathsf{Has}$ satisfies the following properties. These are derivable from the definition of $\mathsf{Has}$.

$$\begin{aligned}
& \forall i, s, u, u'.\ \mathsf{Has}(i,s) \mathbin{@} u \wedge (u' > u) \wedge \neg\mathsf{Reset}() \text{ on } [u,u'] \supset \mathsf{Has}(i,s) \mathbin{@} u' \qquad (H1) \\
& \forall i, s, u, u'.\ \mathsf{Has}(i,s) \mathbin{@} u \wedge (u' < u) \wedge \neg\mathsf{Has}(i,s) \text{ on } [u', u) \supset \\
& \qquad \exists m.\big( (\exists l, j, m'.\ \mathsf{Recv}(i,j,m) \mathbin{@} u \vee \mathsf{Read}(i,l,m) \mathbin{@} u \vee \mathsf{Hmac}(i,m',l,m) \mathbin{@} u) \\
& \qquad\qquad \wedge (\exists S.\ \mathsf{HasSet}(i,S) \mathbin{@} u \wedge \mathsf{Contains}(m,s,S) \mathbin{@} u)\big) \qquad (H2)
\end{aligned}$$

$H1$ states that a thread retains information $s$, unless there is a reset. $H2$ states that the time $u$ when $\mathsf{Has}$ becomes true for thread $i$ is the time when thread $i$ obtains $m$ that enables the derivation of $s$.

We expand the definition of $\mathsf{NoAdv}(u)$ from Section 5. Locations $\mathsf{LoggerLoc}(L)$ are instantiated with the locations $M.ram.keyloc, M.tpm.v.channel, M.ram.logger\_tpm, M.ram.logger\_sh$ and $\mathsf{LoggerOSLoc}(L)$ with $M.ram.logger\_os$.

$$\begin{aligned}
& \forall i, v_1, \ldots, v_{12}, u.\ \mathsf{Owner}(\mathsf{i}, \widehat{\mathsf{root}}) \mathbin{@} u \supset \neg\mathsf{RW}(i, M.ram.keyloc, v_1, v_2) \mathbin{@} u \wedge \neg\mathsf{RW}(i, m.tpm.v.channel, v_3, v_4) \mathbin{@} u \wedge \\
& \qquad \neg\mathsf{RW}(i, m.tpm.v.channel, v_5, v_6) \mathbin{@} u \wedge \neg\mathsf{RW}(i, M.ram.logger\_tpm, v_7, v_8) \mathbin{@} u \wedge \\
& \qquad \neg\mathsf{RW}(i, M.ram.logger\_sh, v_9, v_{10}) \mathbin{@} u \wedge \\
& \qquad \big(\mathsf{RW}(i, M.ram.logger\_os, v_{11}, v_{12}) \mathbin{@} u \supset \mathsf{HT}(i, \widehat{root}, \mathtt{OS})\big)
\end{aligned}$$

Finally, we define predicates used in the main theorem and the key invariant properties (Section 5).

$$\mathsf{RW}(i,l,v_1,v_2) \mathbin{@} u = \ \mathsf{Read}(i,l,v_1) \mathbin{@} u \vee \mathsf{Write}(i,l,v_2)$$

$$\mathsf{inMem}(i,m) \mathbin{@} u = \exists l.\ \mathsf{Mem}(l,m) \mathbin{@} u \wedge \mathsf{CanRead}(i,l) \mathbin{@} u$$

$$\begin{aligned}
\mathsf{LastLogIdx}(k, u, u_{end}) = \ & (k \ge 0) \wedge (u \le u_{end}) \wedge \\
& \forall u', i, j, v, v', z.\ \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u \wedge (u \le u' \le u_{end}) \wedge (k \ge 1) \wedge (z > k) \wedge \mathsf{Write}(i, \mathit{fileLoc}(k), v) \mathbin{@} u \wedge \\
& \qquad \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u' \supset \neg\mathsf{Write}(j, \mathit{fileLoc}(z), v') \mathbin{@} u'
\end{aligned}$$

$$\begin{aligned}
\mathsf{keyOwnerIn}(u) = \forall u', m, n, i, S.\ & (u' \ge u) \wedge \mathsf{Has}(i,m) \mathbin{@} u \wedge \mathsf{Contains}(m, key(n), S) \mathbin{@} u' \wedge \neg(tpm.key \in S) \\
& \supset \mathsf{HT}(i, , \hat{L}, \mathtt{LOGGER}) \mathbin{@} u \vee \mathsf{Owner}(i, \widehat{tpm}) \mathbin{@} u \vee \mathsf{Owner}(i, \hat{V}) \mathbin{@} u
\end{aligned}$$

$$\begin{aligned}
\mathsf{keyMemIn}(u) = \forall u', n, l, m, S.\ \ & (u' \ge u) \wedge \mathsf{Mem}(l,m) \mathbin{@} u \wedge \mathsf{Contains}(m, key(n), S) \mathbin{@} u' \wedge \neg(tpm.key \in S) \\
& \supset l \in \{M.disk.sealedkey, M.ram.keyloc, M.tpm.v.channel\}
\end{aligned}$$

$$\begin{aligned}\mathsf{oldKeyAdv}(u,u_a) = \forall i,m,v,S,k_a,u_l.\ &(v \geq u \geq u_a) \wedge \mathsf{Has}(i,m) @ u \wedge \mathit{tpm.key} \notin S \wedge \\ &\mathsf{Contains}(m, \mathit{key}(k_a), S) @ v \wedge \neg \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) @ u \wedge \\ &\neg \mathsf{Owner}(i, \widehat{tpm}) @ u \wedge \mathsf{LastLogIdx}(k, u_l, u_a) \wedge (k_a \geq 1) \\ &\supset k_a \geq k+1\end{aligned}$$

$$\begin{aligned}\mathsf{oldKeyNotInMem}(u,u_a) = \forall l,u_l,k,k',S,m.\ &(u \geq u_a) \wedge (k \geq 1) \wedge \mathsf{LastLogIdx}(k,u_l,u_a) \wedge \\ &\wedge (1 \leq k' \leq k) \wedge (l \in M) \wedge \mathsf{Mem}(l,m) @ u \wedge \mathit{tpm.key} \notin S \\ &\supset \neg \mathsf{Contains}(m, \mathit{key}(k'), S) @ u\end{aligned}$$

## D Axioms

We assume the following axioms. which have been proven sound based on the semantic model.

| | |
|---|---|
| $Ax1$ | $\forall l,m,u,v.\ \mathsf{Mem}(l,m) @ u \wedge (v < u) \wedge \neg \mathsf{Mem}(l,m) @ v \supset$ $\exists u',i.\ (v < u' \leq u) \wedge \mathsf{Write}(i,l,m) @ u'$ |
| $Ax2$ | $\forall i,l,m,u'.\ \mathsf{Write}(i,l,m) @ u' \supset \exists u''.(u'' < u') \wedge \mathsf{Has}(i,m) @ u'' \wedge \neg \mathsf{Reset}()$ on $[u'',u']$ |
| $Ax3$ | $\forall i,j,m,u'.\ \mathsf{Recv}(i,j,m) @ u' \supset \exists u''.(u'' < u') \wedge \mathsf{Send}(j,i,m) @ u''$ |
| $Ax4$ | $\forall i,j,m,u'.\ \mathsf{Send}(i,j,m) @ u' \supset \exists u''.(u'' < u') \wedge \mathsf{Has}(i,m) @ u'' \wedge \neg \mathsf{Reset}()$ on $(u'',u']$ |
| $Ax5$ | $\forall i,l,m,u'.\ \mathsf{Read}(i,l,m) @ u' \supset \exists u''.(u'' < u') \wedge \mathsf{Mem}(l,m)$ on $[u'',u'] \wedge \neg \mathsf{Reset}()$ on $(u'',u']$ |
| $Ax6$ | $\forall i,i',u,u',l,v.\ \mathsf{Poll}(i,l,v) @ u \wedge \mathsf{CanWrite}(\{i,i'\},l) @ u \wedge \neg \mathsf{Mem}(l,v) @ u' \wedge (u' < u) \wedge$ $\neg \mathsf{Reset}()$ on $(u',u] \wedge \neg \mathsf{Write}(i,l,v)$ on $(u',u] \supset \mathsf{Write}(i',l,v) @ u$ |
| $Ax7rd$ | $\forall i,j,u,u',l,I,J.\ \mathsf{IsReadLocked}(i,l) @ u \wedge \mathsf{Owner}(i,I) \wedge I \prec J \wedge \mathsf{Owner}(j,J) \supset \mathsf{CanRead}(j,l) @ u$ |
| $Ax7wr$ | $\forall i,j,u,u',l,I,J.\ \mathsf{IsWriteLocked}(i,l) @ u \wedge \mathsf{Owner}(i,I) \wedge I \prec J \wedge \mathsf{Owner}(j,J) \supset \mathsf{CanWrite}(j,l) @ u$ |
| $Ax8$ | $\forall i,l,m,u.\ \mathsf{Read}(i,l,m) @ u \supset \mathsf{Mem}(l,m) @ u$ |
| $Ax9$ | $\forall i,\mathit{mac},d,\mathit{key},u.\ \mathsf{VerifyHmac}(i,\mathit{mac},d,\mathit{key}) @ u \supset \exists j,l,u'.(u' < u) \wedge \mathsf{Hmac}(j,d,l,\mathit{key}) @ u'$ |
| $Ax10$ | $\forall i,\mathit{mac},d,\mathit{key},u.\ \mathsf{Hmac}(i,d,l,\mathit{key}) @ u \supset \mathsf{CanRead}(i,l) @ u \wedge \mathsf{Mem}(l,\mathit{key}) @ u$ |
| $Ax11$ | $\forall i,\hat{K},u'',u.\ \mathsf{Owner}(i,\widehat{K}) @ u'' \wedge \neg \mathsf{Reset}()$ on $(u'',u] \supset \mathsf{Owner}(i,\widehat{K}) @ u$ |
| $Ax12$ | $\forall i,\hat{K},u'',u.\ \mathsf{HT}(i,\hat{P},\widehat{K}) @ u'' \wedge \neg \mathsf{Reset}()$ on $(u'',u] \supset \mathsf{HT}(i,\hat{P},\widehat{K}) @ u$ |
| $Ax13$ | $\forall i,\hat{K},u'',u.\ \neg \mathsf{Owner}(i,\widehat{K}) @ u'' \wedge \neg \mathsf{Reset}()$ on $(u'',u] \supset \neg \mathsf{Owner}(i,\widehat{K}) @ u$ |
| $Ax14$ | $\forall i,\hat{K},u'',u.\ \neg \mathsf{HT}(i,\hat{P},\widehat{K}) @ u'' \wedge \neg \mathsf{Reset}()$ on $(u'',u] \supset \neg \mathsf{HT}(i,\hat{P},\widehat{K}) @ u$ |

## E Proofs

We present the proofs of the following four properties.

1. $\forall u. u \leq u_a \supset \mathsf{keyOwnerIn}(u)$
2. $\forall u. u \leq u_a \supset \mathsf{keyMemIn}(u)$
3. $\forall u. u > u_a \supset \mathsf{oldKeyAdv}(u,u_a)$
4. $\forall u. u > u_a \supset \mathsf{oldKeyNotInMem}(u,u_a)$

### E.1 Proofs of 1 and 2

The proof proceeds by transfinite induction on $u$. First, we show that $\mathsf{keyOwnerIn}(-\infty) \wedge \mathsf{keyMemIn}(-\infty)$. Clearly, $\forall i,t.\ \neg \mathsf{Has}(i,t) @ -\infty$ since Has relies on actions in the past and no action has happened at $-\infty$. Thus, $\mathsf{keyOwnerIn}(-\infty)$ trivially holds. Also, we assume that $\mathsf{keyMemIn}(-\infty)$ holds, which is justified by the fact that the system administrator hands over the machine with the key at location $\mathit{M.disk.sealedkey}$.

For the induction hypothesis, assume that $\mathsf{keyOwnerIn}(u) \wedge \mathsf{keyMemIn}(u)$ for all $u < u_0 \leq u_a$ for some $u_0$. We need to show $\mathsf{keyOwnerIn}(u_0) \wedge \mathsf{keyMemIn}(u_0)$.

**Proving $\mathsf{keyMemIn}(u_0)$ holds:** We do so by contradiction. First, assume $\gamma \vdash \neg \mathsf{keyMemIn}(u_0)$. This can be expanded to be written as (with $L$ standing for $\{\mathit{M.disk.sealedkey}, \mathit{M.ram.keyloc}, \mathit{M.tpm.v.channel}\}$)

$$\psi = \exists u',n,l,m,S.(u' \geq u_0) \wedge \mathsf{Mem}(l,m) @ u_0 \wedge \mathsf{Contains}(m, \mathit{key}(n), S) @ u' \wedge \neg(\mathit{tpm.key} \in S) \wedge l \notin L$$

Choose fresh values of the existential (captured in the same formula used on the left as $\psi$) to get

$$\gamma, \psi \vdash (u_0' \geq u_0) \wedge \mathsf{Mem}(l, m) \text{ @ } u_0 \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \text{ @ } u_0' \wedge \neg(tpm.key \in S_0) \wedge l_0 \notin L \quad (A1)$$

Using Axion $Ax1$ and $Ax2$ it is easy to prove that

$$\begin{aligned}&\forall l, m, u, v.\ \mathsf{Mem}(l, m) \text{ @ } u \wedge (v < u) \wedge \neg\mathsf{Mem}(l, m) \text{ @ } v \supset \\ &\quad \exists u'', u', i.(u'' < u' \leq u) \wedge \mathsf{Write}(i, l, m) \text{ @ } u' \wedge \mathsf{Has}(i, m) \text{ @ } u'' \wedge \neg\mathsf{Reset}() \text{ on } (u'', u']\end{aligned}$$

In particular, if we choose $l = l_0, m = m_0, u = u_0, v = -\infty$, we get

$$\begin{aligned}&\gamma, \psi \vdash \mathsf{Mem}(l_0, m_0) \text{ @ } u_0 \wedge \neg\mathsf{Mem}(l_0, m_0) \text{ @ } -\infty \supset \\ &\quad \exists u'', u', i.(u'' < u' \leq u_0) \wedge \mathsf{Write}(i, l_0, m_0) \text{ @ } u' \wedge \mathsf{Has}(i, m_0) \text{ @ } u'' \wedge \neg\mathsf{Reset}() \text{ on } (u'', u'] \qquad (R1)\end{aligned}$$

The contrapositive within $\mathsf{keyMemIn}(-\infty)$ yields

$$\mathsf{keyMemIn}(-\infty) = \forall n, l, m, S, u'.\ l \notin L \supset \neg\big(\mathsf{Mem}(l, m) \text{ @ } -\infty \wedge \mathsf{Contains}(m, key(n), S) \text{ @ } u' \wedge \neg(tpm.key \in S)\big)$$

In particular, if we choose $l = l_0, m = m_0, n = n_0, S = S_0, u' = u_0'$, we get

$$\gamma, \psi \vdash l_0 \notin L \supset \ \neg\big(\mathsf{Mem}(l_0, m_0) \text{ @ } -\infty \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \text{ @ } u_0' \wedge \neg(tpm.key \in S_0)\big)$$

Moving clauses to other side of implication we get

$$\gamma, \psi \vdash l_0 \notin L \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \text{ @ } u_0' \wedge \neg(tpm.key \in S_0) \supset \ \neg\mathsf{Mem}(l_0, m_0) \text{ @ } -\infty$$

Now, using $(R1)$ we get

$$\begin{aligned}&\gamma, \psi \vdash \mathsf{Mem}(l_0, m_0) \text{ @ } u_0 \wedge l_0 \notin L \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \text{ @ } u_0' \wedge \neg(tpm.key \in S_0) \supset \\ &\quad \exists u'', u', i.(u'' < u' \leq u_0) \wedge \mathsf{Write}(i, l_0, m_0) \wedge \mathsf{Has}(i, m_0) \text{ @ } u'' \wedge \neg\mathsf{Reset}() \text{ on } (u'', u'] \qquad (R3)\end{aligned}$$

Combining $R3$ with $A1$ and using modus ponens we get

$$\begin{aligned}&\gamma, \psi \vdash \exists u', u'', i.\ (u'' < u' \leq u_0) \wedge \mathsf{Has}(i, m_0) \text{ @ } u'' \wedge \neg\mathsf{Reset}() \text{ on } (u'', u'] \wedge \mathsf{Write}(i, l_0, m_0) \text{ @ } u' \\ &\quad \wedge (u_0' \geq u_0) \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \text{ @ } u_0' \wedge l_0 \notin L \wedge \neg(tpm.key \in S_0)\end{aligned}$$

Again using fresh values for the existential we get

$$\begin{aligned}&\gamma, \psi \vdash (u_{-1}'' < u_{-1}' \leq u_0 \leq u_0') \wedge \mathsf{Has}(i_0, m_0) \text{ @ } u_{-1}'' \wedge \neg\mathsf{Reset}() \text{ on } (u_{-1}'', u_{-1}'] \wedge \\ &\quad \mathsf{Write}(i_0, l_0, m_0) \text{ @ } u_{-1}' \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \text{ @ } u_0' \wedge l_0 \notin L \wedge \neg(tpm.key \in S_0)\end{aligned}$$

Since, $u_{-1}'' < u_0$ we have from induction hypothesis that $\mathsf{keyOwnerIn}(u_{-1}'')$ holds. From the definition of $\mathsf{keyOwnerIn}(u_{-1}'')$ and the just proved

$$\mathsf{Has}(i_0, m_0) \text{ @ } u_{-1}'' \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \text{ @ } u_0' \wedge \neg(tpm.key \in S_0)$$

we can conclude that

$$\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_{-1}'' \vee \mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u_{-1}''$$

Thus,

$$\begin{aligned}&\gamma, \psi \vdash (u_{-1}' \leq u_0 \leq u_0') \wedge \big(\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_{-1}'' \vee \mathsf{Owner}(i_0, \widehat{tpm}) \text{ @ } u_{-1}''\big) \wedge \\ &\quad \neg\mathsf{Reset}() \text{ on } (u_{-1}'', u_{-1}'] \wedge \mathsf{Write}(i_0, l_0, m_0) \text{ @ } u_{-1}' \wedge \mathsf{Contains}(m_0, key(n_0), \phi) \text{ @ } u_0' \\ &\quad \wedge l_0 \notin L \wedge \neg(tpm.key \in S_0)\end{aligned}$$

Using $Ax11$ and $Ax12$ with $\neg\mathsf{Reset}()$ on $(u_{-1}'', u_{-1}']$, we get $\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER})$ @ $u_{-1}' \vee \mathsf{Owner}(i_0, \widehat{tpm})$ @ $u_{-1}'$. Thus, we get

$$\begin{aligned}&\gamma, \psi \vdash (u_{-1}' \leq u_0 \leq u_0') \wedge \big(\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_{-1}' \vee \mathsf{Owner}(i_0, \widehat{tpm}) \text{ @ } u_{-1}'\big) \wedge \\ &\quad \mathsf{Write}(i_0, l_0, m_0) \text{ @ } u_{-1}' \wedge \mathsf{Contains}(m_0, key(n_0), \phi) \text{ @ } u_0' \wedge l_0 \notin L \wedge \neg(tpm.key \in S_0) \qquad (R2)\end{aligned}$$

It is possible to prove by analyzing the programs that

$$\begin{aligned}&\gamma, \psi \vdash \forall n, l, m, u, i, v, S.\ (u \leq u_0) \wedge (u \leq v) \wedge \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \ @\ u \wedge \\ &\quad \mathsf{Write}(i, l, m) \ @\ u \wedge \mathsf{Contains}(m, key(n), S) \ @\ v \wedge \neg(tpm.key \in S) \supset l \in L \\ &\gamma, \psi \vdash \forall n, l, m, u, i, v, S.\ (u \leq u_0) \wedge (u \leq v) \wedge \mathsf{Owner}(i, \widehat{tpm}) \ @\ u \wedge \mathsf{Write}(i, l, m) \ @\ u \wedge \\ &\quad\quad \mathsf{Contains}(m, key(n), S) \ @\ v \wedge \neg(tpm.key \in S) \supset l = m.tpm.v.channel\end{aligned}$$

The above proofs rely on the induction hypothesis about keyOwnerIn and keyMemIn to claim that the log entries themselves do not contain any key. Instantiate the universal to specific values and combine the above two results as follows

$$\begin{aligned}&\gamma, \psi \vdash (u'_{-1} \leq u_0 \leq u'_0) \wedge \big(\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \ @\ u'_{-1} \vee \mathsf{Owner}(i_0, \widehat{tpm}) \ @\ u'_{-1}\big) \wedge \\ &\quad \mathsf{Write}(i_0, l_0, m_0) \ @\ u'_{-1} \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \ @\ u'_0 \wedge \neg(tpm.key \in S_0) \supset l_0 \in L\end{aligned}$$

Thus, using $(R2)$ and the above result, we can prove the following, which is a contradiction.

$$\gamma, \psi \vdash l_0 \in L \wedge l_0 \notin L$$

**Proving keyOwnerIn($u_0$) holds:** We do so by contradiction.
Assume $\gamma \vdash \neg\mathsf{keyOwnerIn}(u_0)$. In an expanded form this is same as

$$\begin{aligned}&\exists u', m, n, i, S.\ (u' \geq u_0) \wedge \mathsf{Has}(i, m) \ @\ u_0 \wedge \mathsf{Contains}(m, key(n), S) \ @\ u' \wedge \\ &\quad \neg(tpm.key \in S) \wedge \neg\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \ @\ u_0 \wedge \neg\mathsf{Owner}(i, \widehat{tpm}) \ @\ u_0 \wedge \neg\mathsf{Owner}(i, \hat{V}) \ @\ u_0\end{aligned}$$

Let the above formula be $\psi$. Again fix values of existential so that

$$\begin{aligned}&\gamma, \psi \vdash (u'_0 \geq u_0) \wedge \mathsf{Has}(i_0, m_0) \ @\ u_0 \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \ @\ u'_0 \wedge \\ &\quad \neg(tpm.key \in S_0) \wedge \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \ @\ u_0 \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \ @\ u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \ @\ u_0 \quad (A2)\end{aligned}$$

Now, we know that $\mathsf{Has}(i_0, m_0) \supset \neg\mathsf{Reset}() \ @\ u$. Also, using $Ax13$ and $Ax14$ and picking $u_p$ to be the max of the witnesses for the existential $u'$ for the tpm thread and the logger thread, with the verifier never reseting, we get $\forall u.\ (u_p < u < u_0) \supset \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \ @\ u \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \ @\ u \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \ @\ u$. Using facts from $A2$ we get

$$\begin{aligned}&\forall u.\ (u_p < u < u_0) \supset \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \ @\ u \wedge \\ &\quad \neg\mathsf{Owner}(i_0, \widehat{tpm}) \ @\ u \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \ @\ u \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \ @\ u'_0 \wedge \neg(tpm.key \in S_0) \qquad (B1)\end{aligned}$$

Taking contrapositive within $\mathsf{keyOwnerIn}(u)$ we get

$$\begin{aligned}\mathsf{keyOwnerIn}(u) = &\ \forall u', m, n, i, S.\ \neg\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \ @\ u \wedge \neg\mathsf{Owner}(i, \widehat{tpm}) \ @\ u \wedge \neg\mathsf{Owner}(i, \hat{V}) \ @\ u \supset \\ &\quad \neg\Big((u' \geq u) \wedge \mathsf{Has}(i, m) \ @\ u \wedge \mathsf{Contains}(m, key(n), S) \ @\ u' \wedge \neg(tpm.key \in S)\Big)\end{aligned}$$

Instantiating, with $u < u_0$, and noting that $u < u_0 \leq u'_0$ we get

$$\begin{aligned}&\gamma \vdash \forall u.\ (u < u_0) \wedge \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \ @\ u \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \ @\ u \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \ @\ u \supset \\ &\quad \neg\Big(\mathsf{Has}(i_0, m_0) \ @\ u \wedge \mathsf{Contains}(m_0, key(n_0), S_0) \ @\ u'_0 \wedge \neg(tpm.key \in S_0)\Big)\end{aligned}$$

or rearranging

$$\begin{aligned}&\gamma \vdash \forall u.\ \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \ @\ u \wedge \neg(tpm.key \in S_0) \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \ @\ u \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \ @\ u \wedge \\ &\quad \mathsf{Contains}(m_0, key(n_0), S_0) \ @\ u'_0 \supset \neg(u < u_0) \vee \neg\mathsf{Has}(i_0, m_0) \ @\ u\end{aligned}$$

Using this with $(B1)$ we get

$$\gamma \vdash \forall u.\ (u_p < u < u_0) \supset \neg(u < u_0) \vee \neg\mathsf{Has}(i_0, m_0) \ @\ u$$

which is same as $\forall u.\ (u_p < u < u_0) \supset \neg\mathsf{Has}(i_0, m_0) \mathbin{@} u$ and which is same as $\neg\mathsf{Has}(i_0, m_0)$ on $[u_p, u_0)$. Thus, we have

$$\gamma, \psi \vdash \neg\mathsf{Has}(i_0, m_0) \text{ on } [u_p, u_0)$$

We know from $A2$ that

$$\gamma, \psi \vdash \mathsf{Has}(i_0, m_0) \mathbin{@} u_0$$

Now, invoking property $H2$ about $\mathsf{Has}$ we get

$$\begin{aligned}\gamma, \psi \vdash \exists m.\big(\ &(\exists l, j, m'.\ \mathsf{Recv}(i_0, j, m) \mathbin{@} u_0 \vee \mathsf{Read}(i_0, l, m) \mathbin{@} u_0 \vee \mathsf{Hmac}(i_0, m', l, m) \mathbin{@} u_0)\\ &\wedge (\exists S.\ \mathsf{HasSet}(i_0, S) \mathbin{@} u_0 \wedge \mathsf{Contains}(m, m_0, S) \mathbin{@} u_0)\big)\end{aligned}$$

Instantiating the existential with fresh values we get (with the formula captures in $\psi$)

$$\begin{aligned}\gamma, \psi \vdash\ &(\mathsf{Recv}(i_0, j_0, m_{-1}) \mathbin{@} u_0 \vee \mathsf{Read}(i_0, l_0, m_{-1}) \mathbin{@} u_0 \vee \mathsf{Hmac}(i_0, m_0', l_0, m_{-1}) \mathbin{@} u_0)\\ &\wedge \mathsf{HasSet}(i_0, S_{-1}) \mathbin{@} u_0 \wedge \mathsf{Contains}(m_{-1}, m_0, S_{-1}) \mathbin{@} u_0 \wedge \neg(\mathit{tpm.key} \in S_0)\end{aligned}$$

From $A2$ we know that

$$\begin{aligned}\gamma, \psi \vdash\ &(u_0' \geq u_0) \wedge \mathsf{Contains}(m_0, \mathit{key}(n_0), S_0) \mathbin{@} u_0' \wedge \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_0 \wedge\\ &\neg\mathsf{Owner}(i_0, \widehat{tpm}) \mathbin{@} u_0 \wedge \neg(\mathit{tpm.key} \in S_0) \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \mathbin{@} u_0\end{aligned}$$

First, note that $\mathsf{Contains}(m_{-1}, m_0, S_{-1}) \mathbin{@} u_0 \wedge (u_0' \geq u_0) \wedge \mathsf{Contains}(m_0, \mathit{key}(n_0), S_0) \mathbin{@} u_0'$ implies that $\mathsf{Contains}(m_{-1}, \mathit{key}(n_0), S_0 \cup S_{-1}) \mathbin{@} u_0' \wedge (u_0' \geq u_0)$. Also, we know that $\neg\mathsf{Owner}(i_0, \widehat{tpm}) \wedge \mathsf{HasSet}(i_0, S_{-1}) \supset \neg(\mathit{tpm.key} \in S_{-1})$ Thus, we can infer that

$$\begin{aligned}\gamma, \psi \vdash\ &(u_0' \geq u_0) \wedge \mathsf{Contains}(m_{-1}, \mathit{key}(n_0), S_0 \cup S_{-1}) \mathbin{@} u_0' \wedge \neg(\mathit{tpm.key} \in S_0 \cup S_{-1}) \wedge\\ &\neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \mathbin{@} u_0 \wedge\\ &(\mathsf{Recv}(i_0, j_0, m_{-1}) \mathbin{@} u_0 \vee \mathsf{Read}(i_0, l_0, m_{-1}) \mathbin{@} u_0 \vee \mathsf{Hmac}(i_0, m_0', l_0, m_{-1}) \mathbin{@} u_0)\end{aligned}$$

We analyze the three disjuncts separately reaching contradictions in each case. We skip the $\mathsf{Hmac}$ case, since it is similar to $\mathsf{Read}$. Thus, we analyze two sub cases below.

**SubCase 1:** $\mathsf{Recv}(i_0, j_0, m_{-1}) \mathbin{@} u_0$
This case yields

$$\begin{aligned}\gamma, \psi \vdash\ &(u_0' \geq u_0) \wedge \mathsf{Contains}(m_{-1}, \mathit{key}(n_0), S_0 \cup S_{-1}) \mathbin{@} u_0' \wedge \neg(\mathit{tpm.key} \in S_0 \cup S_{-1}) \wedge\\ &\neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \mathbin{@} u_0 \wedge \mathsf{Recv}(i_0, j_0, m_{-1}) \mathbin{@} u_0\end{aligned}$$

Using $Ax3$ and $Ax4$ we can conclude that

$$\begin{aligned}\gamma, \psi \vdash\ &(u_0' \geq u_0 > u_{-1} > u_{-2}) \wedge \mathsf{Contains}(m_{-1}, \mathit{key}(n_0), S_0 \cup S_{-1}) \mathbin{@} u_0' \wedge \neg(\mathit{tpm.key} \in S_0 \cup S_{-1}) \wedge\\ &\neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \mathbin{@} u_0 \wedge \mathsf{Send}(j_0, i_0, m_{-1}) \mathbin{@} u_{-1}\\ &\wedge \mathsf{Has}(j_0, m_{-1}) \mathbin{@} u_{-2} \wedge \neg\mathsf{Reset}() \text{ on } (u_{-2}, u_{-1}]\end{aligned}$$

Now, we can invoke $\mathsf{keyOwnerIn}(u_{-2})$ as $u_{-2} < u_0 \leq u_0'$ with suitable instantiation to get

$$\begin{aligned}\gamma, \psi \vdash\ &(u_0' \geq u_0 > u_{-1} > u_{-2}) \wedge \big(\mathsf{HT}(j_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_{-2} \vee \mathsf{Owner}(j_0, \widehat{tpm}) \mathbin{@} u_{-2} \vee \mathsf{Owner}(j_0, \hat{V}) \mathbin{@} u_{-2}\big) \wedge\\ &\neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \mathbin{@} u_0 \wedge \mathsf{Send}(j_0, i_0, m_{-1}) \mathbin{@} u_{-1} \wedge\\ &\mathsf{Contains}(m_{-1}, \mathit{key}(n_0), S_0 \cup S_{-1}) \mathbin{@} u_0' \wedge \neg\mathsf{Reset}() \text{ on } (u_{-2}, u_{-1}] \qquad (B2)\end{aligned}$$

It is possible to show by analyzing the programs that

$$\begin{aligned}\gamma, \psi \vdash\ &\forall n, m, u, i, j, v, S.\ (u_{-2} \leq u < u_0) \wedge (u \leq v) \wedge \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_{-2} \wedge\\ &\neg\mathsf{Reset}() \text{ on } (u_{-2}, u_{-1}] \wedge \mathsf{Contains}(m, \mathit{key}(n), S) \mathbin{@} v \supset \neg\mathsf{Send}(i, j, m) \mathbin{@} u\\ \gamma, \psi \vdash\ &\forall n, m, u, i, j, v, S.\ (u_{-2} \leq u < u_0) \wedge (u \leq v) \wedge \mathsf{Owner}(i, \widehat{tpm}) \mathbin{@} u_{-2} \wedge\\ &\neg\mathsf{Reset}() \text{ on } (u_{-2}, u_{-1}] \wedge \mathsf{Contains}(m, \mathit{key}(n), S) \mathbin{@} v \supset \neg\mathsf{Send}(i, j, m) \mathbin{@} u\\ \gamma, \psi \vdash\ &\forall n, m, u, i, j, v, S.\ (u_{-2} \leq u < u_0) \wedge (u \leq v) \wedge \mathsf{Owner}(i, \hat{V}) \mathbin{@} u_{-2} \wedge\\ &\neg\mathsf{Reset}() \text{ on } (u_{-2}, u_{-1}] \wedge \mathsf{Contains}(m, \mathit{key}(n), S) \mathbin{@} v \supset \neg\mathsf{Send}(i, j, m) \mathbin{@} u\end{aligned}$$

Instantiating, and combining we get

$$\gamma, \psi \vdash (u_{-2} \leq u_{-1} < u_0 \leq u'_0) \wedge \big(\mathsf{HT}(j_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_{-2} \vee \mathsf{Owner}(j_0, \widehat{tpm}) \text{ @ } u_{-2} \vee \mathsf{Owner}(j_0, \hat{V}) \text{ @ } u_{-2}\big) \wedge \\ \neg\mathsf{Reset}() \text{ on } (u_{-2}, u_{-1}] \wedge \mathsf{Contains}(m_{-1}, key(n_0), S_0 \cup S_{-1}) \text{ @ } u'_0 \supset \neg\mathsf{Send}(j_0, i_0, m_{-1}) \text{ @ } u_{-1}$$

Thus, using the above result with $(B2)$ we get

$$\gamma, \psi \vdash \neg\mathsf{Send}(j_0, i_0, m_{-1}) \text{ @ } u_{-1}$$

but, we also know from $B2$ that $\gamma, \psi \vdash \mathsf{Send}(j_0, i_0, m_{-1}) \text{ @ } u_{-1}$, which is a contradiction.

Next, we analyze the second disjunct

**SubCase 2:** $\mathsf{Read}(i_0, l_0, m_{-1}) \text{ @ } u_0$
This case yields

$$\gamma, \psi \vdash (u'_0 \geq u_0) \wedge \mathsf{Contains}(m_{-1}, key(n_0), S_0 \cup S_{-1}) \text{ @ } u'_0 \wedge \mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_0 \wedge \\ \neg\neg\mathsf{Owner}(i_0, \widehat{tpm}) \text{ @ } u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \text{ @ } u_0 \wedge \\ (tpm.key \notin S_0 \cup S_{-1}) \wedge \mathsf{Read}(i_0, l_0, m_{-1}) \text{ @ } u_0$$

Now, using $Ax5$ we can conclude that

$$\gamma, \psi \vdash (u'_0 \geq u_0 > u'') \wedge \mathsf{Contains}(m_{-1}, key(n_0), S_0 \cup S_{-1}) \text{ @ } u'_0 \wedge \mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_0 \wedge \\ \neg\mathsf{Owner}(i_0, \widehat{tpm}) \text{ @ } u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \text{ @ } u_0 \wedge (tpm.key \notin S_0 \cup S_{-1}) \wedge \\ \mathsf{Read}(i_0, l_0, m_{-1}) \text{ @ } u_0 \wedge \mathsf{Mem}(l_0, m_{-1}) \text{ on } [u'', u_0] \wedge \neg\mathsf{Reset}() \text{ on } [u'', u_0]$$

Now using induction hypothesis for predicate $\mathsf{keyMemIn}(u'')$, given $u'' < u_0$, we get

$$\gamma, \psi \vdash (u'_0 \geq u_0 > u'') \wedge \mathsf{Contains}(m_{-1}, key(n_0), S_0 \cup S_{-1}) \text{ @ } u'_0 \wedge \mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_0 \wedge \\ \neg\mathsf{Owner}(i_0, \widehat{tpm}) \text{ @ } u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \text{ @ } u_0 \wedge (tpm.key \notin S_0 \cup S_{-1}) \wedge \\ \mathsf{Read}(i_0, l_0, m_{-1}) \text{ @ } u_0 \wedge \mathsf{Mem}(l_0, m_{-1}) \text{ on } [u'', u_0] \wedge (l_0 \in L) \wedge \neg\mathsf{Reset}() \text{ on } [u'', u_0] \qquad (C1)$$

For now, assume a formula $\xi$ ($\xi$ will be proved later) given by

$$\gamma \vdash \forall u, u'', i, n, m, S, v, l.\ \neg\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \wedge \neg\mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u \wedge \neg\mathsf{Owner}(i, \hat{V}) \text{ @ } u \\ \wedge \mathsf{Mem}(l, m) \text{ on } [u'', u] \wedge (v \geq u) \wedge (u'' < u \leq u_0) \wedge l \in L \wedge \neg\mathsf{Reset}() \text{ on } [u'', u] \wedge \\ (tpm.key \notin S) \supset \neg\mathsf{Read}(i, l, m) \text{ @ } u \vee \neg\mathsf{Contains}(m, key(n), S) \text{ @ } v$$

$\xi$ basically states that any thread that is not the logger, tpm or verifier does not read $l \in L$ or that memory location $l$ does not contain any key. $\xi$ can be rearranged as

$$\gamma \vdash \forall u, u'', i, n, m, S, v, l.\ \neg\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \wedge \neg\mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u \wedge \neg\mathsf{Owner}(i, \hat{V}) \text{ @ } u \wedge \\ \mathsf{Mem}(l, m) \text{ on } [u'', u] \wedge (v \geq u) \wedge (u'' < u \leq u_0) \wedge \neg\mathsf{Reset}() \text{ on } [u'', u] \wedge (tpm.key \notin S) \wedge \\ \mathsf{Read}(i, l, m) \text{ @ } u \wedge \mathsf{Contains}(m, key(n), S) \text{ @ } v \supset l \notin L$$

Then, instantiating with proper values (e.g., $u = u_0$) we get

$$\gamma \vdash \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_0 \wedge \neg\mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u_0 \wedge \neg\mathsf{Owner}(i, \hat{V}) \text{ @ } u_0 \wedge (u'_0 \geq u_0) \wedge (u'' < u_0) \wedge \\ \mathsf{Mem}(l_0, m_{-1}) \text{ on } [u'', u] \wedge \neg\mathsf{Reset}() \text{ on } [u'', u_0] \wedge (tpm.key \notin S_0 \cup S_{-1}) \wedge \\ \mathsf{Read}(i_0, l_0, m_{-1}) \text{ @ } u_0 \wedge \mathsf{Contains}(m_{-1}, key(n_0), S_0 \cup S_{-1}) \text{ @ } u'_0 \supset l_0 \notin L$$

Thus, using with $(C1)$ we obtain the contradiction

$$\gamma, \psi \vdash l_0 \in L \wedge l_0 \notin L$$

Thus, we now focus on proving $\xi$

$$\gamma \vdash \forall u, u'', i, n, m, S, v, l.\ \neg\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \wedge \neg\mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u \wedge \neg\mathsf{Owner}(i, \hat{V}) \text{ @ } u \\ \wedge \mathsf{Mem}(l, m) \text{ on } [u'', u] \wedge (v \geq u) \wedge (u'' < u \leq u_0) \wedge l \in L \wedge \neg\mathsf{Reset}() \text{ on } [u'', u] \wedge \\ (tpm.key \notin S) \supset \neg\mathsf{Read}(i, l, m) \text{ @ } u \vee \neg\mathsf{Contains}(m, key(n), S) \text{ @ } v$$

Here we ask the reader to recall about our splitting the startup to shutdown time into intervals: one from startup to when the logger incremented the counter, and the other interval from when the logger incremented the counter to shutdown, as stated in the main body. We do so using the following predicate

$$\mathsf{Early}(u) = \forall u'.\forall j.\ \neg(\mathsf{LastReset}(u')\ @\ u) \vee \neg\mathsf{HT}(j, \hat{L}, \texttt{LOGGER})\ @\ u \vee \neg\mathsf{Counterup}(j) \text{ on } (u', u]$$

where $\mathsf{LastReset}(u')\ @\ u = (u' < u) \wedge \mathsf{Reset}()\ @\ u' \wedge \neg\mathsf{Reset}() \text{ on } (u', u]$. Thus, we also have

$$\neg\mathsf{Early}(u) = \exists u'.\exists j.\ \mathsf{LastReset}(u')\ @\ u \wedge \mathsf{HT}(j, \hat{L}, \texttt{LOGGER}) \wedge \neg(\neg\mathsf{Counterup}(j) \text{ on } (u', u])$$

We can split formula $\xi$ into two parts $E$ and $\neg E$ given respectively by

$$\begin{aligned}\gamma \vdash \forall u, u'', i, n, m, S, v, l.\ &\mathsf{Early}(u) \wedge \neg\mathsf{HT}(i, \hat{L}, \texttt{LOGGER})\ @\ u \wedge \neg\mathsf{Owner}(i, \widehat{tpm})\ @\ u \wedge \neg\mathsf{Owner}(i, \hat{V})\ @\ u \\ &\wedge \mathsf{Mem}(l, m) \text{ on } [u'', u] \wedge (v \geq u) \wedge (u'' < u \leq u_0) \wedge l \in L \wedge \neg\mathsf{Reset}() \text{ on } [u'', u] \wedge \\ &(tpm.key \notin S) \supset \neg\mathsf{Read}(i, l, m)\ @\ u \vee \neg\mathsf{Contains}(m, key(n), S)\ @\ v\end{aligned}$$

$$\begin{aligned}\gamma \vdash \forall u, u'', i, n, m, S, v, l.\ &\neg\mathsf{Early}(u) \wedge \neg\mathsf{HT}(i, \hat{L}, \texttt{LOGGER})\ @\ u \wedge \neg\mathsf{Owner}(i, \widehat{tpm})\ @\ u \wedge \neg\mathsf{Owner}(i, \hat{V})\ @\ u \\ &\wedge \mathsf{Mem}(l, m) \text{ on } [u'', u] \wedge (v \geq u) \wedge (u'' < u \leq u_0) \wedge l \in L \wedge \neg\mathsf{Reset}() \text{ on } [u'', u] \wedge \\ &(tpm.key \notin S) \supset \neg\mathsf{Read}(i, l, m)\ @\ u \vee \neg\mathsf{Contains}(m, key(n), S)\ @\ v\end{aligned}$$

We do the proof in three parts: first for the location $M.disk.sealedkey$ and then for $M.ram.keyloc$ and $M.tpm.v.channel$.

**Subsubcase 1**: $l = m.disk.sealedkey$
Remember, we had assumed that

$$A_{NR} = \forall u, i, m.\ \mathsf{Early}(u) \wedge (u \leq u_a) \wedge \neg\mathsf{HT}(i, \hat{L}, \texttt{LOGGER})\ @\ u \supset \neg\mathsf{Read}(i, M.disk.sealedkey, m)\ @\ u$$

which takes care of the $E$ part of the formula.

For the $\neg E$ part we again split time into two parts: one part till the time the logger writes the sealed object to disk, and other part from the time the logger writes the sealed object to actual shutdown.

We first show that $\neg\mathsf{Contains}(m, key(n), S)\ @\ v$ till the time the logger writes the sealed object again at shutdown. Assume the antecedent holds. Look at the following parts of the antecedent of the required to prove formula (with universals instantiated): $\neg\mathsf{Early}(u) \wedge \mathsf{Mem}(M.disk.sealedkey, m) \text{ on } [u'', u] \wedge (v \geq u) \wedge (tpm.key \notin S)$. First, note that by definition of of $\neg\mathsf{Early}(u)$ choosing fresh values $u', j$ we get

$$\mathsf{LastReset}(u')\ @\ u \wedge \mathsf{HT}(j, \hat{L}, \texttt{LOGGER})\ @\ u \wedge \exists u_c.(u' < u_c < u) \wedge \mathsf{Counterup}(j)\ @\ u_c$$

The partial correctness for the `unseal_data` function $[\texttt{unseal\_data}]\langle y, u_b, u_e, j, x\rangle$ implies $\exists m, d, u_u.\ (u_b < u_u < u_e) \wedge \mathsf{Unseal}(k, m, d, tpm.key)\ @\ u_u \wedge \mathsf{Owner}(k, \widehat{tpm})\ @\ u_u$. Thus, we have using the order of execution of programs that (note $u_c$ is the time when counter was incremented)

$$\begin{aligned}&\exists c, m', u_u, u_d, u_w, u_s.\ (u' < u_s < u_w < u_r < u_d < u_u < u_c) \wedge \mathsf{Counterval}(c)\ @\ u_u \wedge \mathsf{Unseal}(k, m', d)\ @\ u_u \\ &\wedge \mathsf{Owner}(k, \widehat{tpm})\ @\ u_u \wedge \mathsf{Mem}(M.disk.sealedkey, m')\ @\ u_d \wedge \mathsf{ReadLock}(j, M.disk.sealedkey)\ @\ u_r \wedge \\ &\mathsf{WriteLock}(j, M.disk.sealedkey)\ @\ u_w \wedge \mathsf{Start}(j)\ @\ u_s\end{aligned}$$

Pick fresh variables for existential, and we use the same names as the quantifier variables. We use the following short form notation for sake of readability

$$\mathsf{Unseal}_{c,\widehat{tpm}}(k, m', d)\ @\ u_u = \mathsf{Counterval}(c)\ @\ u_u \wedge \mathsf{Unseal}(k, m', d)\ @\ u_u \wedge \mathsf{Owner}(k, \widehat{tpm})\ @\ u_u$$

The following is immediate (for given $u_u, u_c$) from the monotonicity of the counter:

$$\forall j, u, c, c*.\ (u_u < u_c < u) \wedge \mathsf{Counterup}(j)\ @\ u_c \wedge \mathsf{Counterval}(c)\ @\ u_u \supset \mathsf{Counterval}(c*)\ @\ u \wedge (c* > c)$$

Also, we have

$$\forall j, u.\ \mathsf{Start}(j)\ @\ u_s \wedge (u_s < u) \wedge \neg\mathsf{Reset}() \text{ on } (u_s, u] \wedge \mathsf{HT}(j, \hat{L}, \texttt{LOGGER})\ @\ u \supset \mathsf{HT}(j, \hat{L}, \texttt{LOGGER}) \text{ on } [u_s, u]$$

Next, note that under assumption about root threads $\mathsf{NoAdv}$ and $Ax7wr$ we have ($\psi$ denotes all the existential that have been picked)

$$\gamma, \psi \vdash \forall v'.\ (u' < u_s < u_w < u_r < u) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \\ \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ on } [u_s, u] \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \mathrel{@} u_w \wedge \mathsf{WriteLock}(j, M.disk.sealedkey) \mathrel{@} u_w \\ \wedge \neg\mathsf{Write}(j, M.disk.sealedkey, v') \text{ on } (u_w, u] \supset \forall i.\neg\mathsf{Write}(i, M.disk.sealedkey, v') \text{ on } (u_w, u]$$

Thus, by preservation of memory value, given no writes, (for all values $m'$)

$$\gamma, \psi \vdash \forall v', m'.\ (u' < u_s < u_w < u_r < u_d < u) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{WriteLock}(j, M.disk.sealedkey) \mathrel{@} u_w \\ \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \mathrel{@} u_r \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ on } [u_s, u] \wedge \mathsf{Mem}(M.disk.sealedkey, m') \mathrel{@} u_d \wedge \\ \neg\mathsf{Write}(j, M.disk.sealedkey, v') \text{ on } (u_w, u] \supset \mathsf{Mem}(M.disk.sealedkey, m') \text{ on } [u_d, u]$$

But, we know from assumed true antecedent of $\neg E$ that $\mathsf{Mem}(M.disk.sealedkey, m)$ on $[u'', u]$, which implies $m' = m$. Also, by definition of $\mathsf{Contains}$ for all $S$

$$\mathsf{Mem}(M.disk.sealedkey, m) \mathrel{@} u \wedge \mathsf{Unseal}_{c,\widehat{tpm}}(k, m, d) \mathrel{@} u_u \wedge \mathsf{Counterval}(c*) \mathrel{@} u \wedge (c* > c) \wedge (tpm.key \notin S) \\ (u_u < u) \wedge (v \geq u) \supset \neg\mathsf{Contains}(m, key(n), S) \mathrel{@} v$$

Thus, we have (using relaxation of the consequent with $\mathsf{Read}$)

$$\gamma, \psi \vdash \forall i, v', m.\ (u' < u_s < u_w < u_r < u_d < u \leq v) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{WriteLock}(j, M.disk.sealedkey) \mathrel{@} u_w \\ \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \mathrel{@} u_r \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ on } [u_s, u] \wedge \mathsf{Mem}(M.disk.sealedkey, m) \mathrel{@} u_d \wedge (tpm.key \notin S) \\ \wedge \mathsf{Unseal}_{c,\widehat{tpm}}(k, m, d) \mathrel{@} u_u \wedge \neg\mathsf{Write}(j, M.disk.sealedkey, v') \text{ on } (u_w, u] \supset \\ \neg\mathsf{Contains}(m, key(n), S) \mathrel{@} v \vee \neg\mathsf{Read}(i, (M.disk.sealedkey, m) \mathrel{@} u \qquad (BeforeFinalSeal)$$

This part takes care of the time till logger writes the sealed key at shutdown.

To reason about the shutdown phase (after the logger writes the sealed key) we proceed below. We know from the logger and shutdown expression that

$$\forall u_z.\ (u_z \leq U) \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \mathrel{@} u_z \wedge \mathsf{Write}(j, M.disk.sealedkey, v) \mathrel{@} u_z \wedge \\ \neg\mathsf{Reset}() \text{ on } [u_z, u] \supset \mathsf{Mem}(M.ram.sh, -1) \text{ on } [u_z, u]$$

Thus, we have

$$\gamma, \psi \vdash \forall u_z, v'.\ (u' < u_s < u_w < u_r < u_z \leq u) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{WriteLock}(j, M.disk.sealedkey) \mathrel{@} u_w \\ \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \mathrel{@} u_r \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ on } [u_s, u] \wedge \mathsf{Write}(j, M.disk.sealedkey, v') \mathrel{@} u_z \\ \supset \mathsf{Mem}(M.ram.sh, -1) \text{ on } [u_z, u]$$

Next, note that under assumption $\mathsf{NoAdv}$ about root threads we have

$$\mathsf{Mem}(M.ram.sh, -1) \mathrel{@} u \wedge \mathsf{Owner}(i, \widehat{root}) \mathrel{@} u \supset \neg\mathsf{Read}(i, M.disk.sealedkey, m) \mathrel{@} u\ .$$

Thus, using $Ax7rd$ we get for all $i$, $(u_r < u_z \leq u) \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \mathrel{@} u_r \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER})$ on $[u_s, u] \wedge \mathsf{Mem}(M.ram.sh, -1) \mathrel{@} u \supset \neg\mathsf{Read}(i, M.disk.sealedkey, v) \mathrel{@} u$. Thus, we have

$$\gamma, \psi \vdash \forall i, u_z, v', m.\ (u' < u_s < u_w < u_r < u_z \leq u) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{WriteLock}(j, m.disk.sealedkey) \mathrel{@} u_w \\ \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \mathrel{@} u_r \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ on } [u_s, u] \wedge \mathsf{Write}(j, M.disk.sealedkey, v') \mathrel{@} u_z \\ \supset \neg\mathsf{Read}(i, M.disk.sealedkey, m) \mathrel{@} u$$

Thus, moving the $\forall u_z$ as an existential into the precedent we get $\exists u_z.\ (u_w < u_z \leq u) \wedge \mathsf{Write}(j, M.disk.sealedkey, v') \mathrel{@} u_z$ which is same as $\neg(\neg\mathsf{Write}(j, M.disk.sealedkey, v) \text{ on } (u_w, u])$. Then, the strengthening antecedent and relaxing the consequent we get

$$\gamma, \psi \vdash \forall i, v', m.(u' < u_s < u_w < u_r < u_d < u \leq v) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{WriteLock}(j, M.disk.sealedkey) \mathrel{@} u_w \\ \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \mathrel{@} u_r \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ on } [u_s, u] \wedge \mathsf{Mem}(M.disk.sealedkey, m) \mathrel{@} u_d \wedge (tpm.key \notin S) \wedge \\ (\neg(\neg\mathsf{Write}(j, M.disk.sealedkey, v') \text{ on } (u_w, u]) \wedge (v \geq u) \supset \\ \neg\mathsf{Contains}(m, key(n), S) \mathrel{@} v \vee \neg\mathsf{Read}(i, M.disk.sealedkey, m) \mathrel{@} u$$

Combining this with the formula ($BeforeFinalSeal$) we get (note same consequent)

$$\gamma, \psi \vdash \forall i, v', m.(u' < u_s < u_w < u_r < u_d < u \leq v) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{WriteLock}(j, M.disk.sealedkey) \text{ @ } u_w \\ \wedge \mathsf{ReadLock}(j, M.disk.sealedkey) \text{ @ } u_r \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ on } [u_s, u] \wedge \mathsf{Mem}(M.disk.sealedkey, m) \text{ @ } u_d \\ \wedge (tpm.key \notin S) \wedge (v \geq u) \supset \neg\mathsf{Contains}(m, key(n), S) \text{ @ } v \vee \neg\mathsf{Read}(i, M.disk.sealedkey, m) \text{ @ } u$$

Note that the consequent does not depend on any of the existential witnesses. Thus, we have

$$\gamma \vdash \forall u, u'', i, S, v, m.\ \neg\mathsf{Early}(u) \wedge \mathsf{Mem}(M.disk.sealedkey, m) \text{ on } [u'', u] \wedge (v \geq u) \\ \wedge (tpm.key \notin S) \supset \neg\mathsf{Contains}(m, key(n), S) \text{ @ } v \vee \neg\mathsf{Read}(i, (M.disk.sealedkey, m) \text{ @ } u$$

The antecedent can be strengthened to obtain the desired result $\neg E$.

Next, we show the proof for that for $l = M.ram.keyloc$. As the proof for $M.tpm.v.channel$ is similar we skip that part. The proof is similar to the last part where we split the required to prove formula into two $E$ and $\neg E$, one with Early and one with $\neg$Early.

**Subsubcase 2**: $l = M.ram.keyloc$

We start with proving $E$. Assume part of antecedent of $E$ after instantiating all universals: $\mathsf{Early}(u) \wedge (v \geq u) \wedge tpm.key \notin S$. Then, $\mathsf{Early}(u)$ holds. Choose $u'$ such that $\mathsf{LastReset}(u')$ @ $u$ (there always exists such a $u'$). Thus, this implies that $\mathsf{HT}(j, \hat{L}, \mathtt{LOGGER})$ @ $u \supset \neg\mathsf{Counterup}(j)$ on $(u', u]$. First, the following hold due to the operational semantics of `reset`, $\mathsf{Reset}()$ @ $u' \supset \mathsf{Mem}(M.ram.keyloc, null)$ @ $u'$. Again, using $Ax1$ we get

$$\forall m, u''.\ \mathsf{Mem}(M.ram.keyloc, m) \text{ @ } u'' \wedge (u'' < u) \wedge \neg\mathsf{Reset}() \text{ on } (u', u] \\ \supset (m = null) \vee \exists k, u'''.\ (u' < u''' \leq u'') \wedge \mathsf{Write}(k, M.ram.keyloc, m) \text{ @ } u'''$$

We know that since $\mathsf{Early}(u)$ holds we have

$$\neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \supset \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \wedge \neg\mathsf{Counterup}(j) \text{ on } (u', u]$$

Also, we know from analysis of logger program that

$$\forall j, v'.\ \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \wedge \neg\mathsf{Counterup}(j) \text{ on } (u', u] \supset \neg\mathsf{Write}(j, M.ram.keyloc, v') \text{ on } (u', u]$$

Thus, by using modus ponens on the last two formulas we get

$$\forall j, v'.\ \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \supset \neg\mathsf{Write}(j, M.ram.keyloc, v') \text{ on } (u', u]$$

Also, from the tpm program we know that

$$\forall i, v'. \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u \supset \neg\mathsf{Write}(i, M.ram.keyloc, v') \text{ on } (u', u]$$

Also, from the verifier program we know that

$$\forall i, v'. \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{Owner}(i, \hat{V}) \text{ @ } u \supset \neg\mathsf{Write}(i, M.ram.keyloc, v') \text{ on } (u', u]$$

Thus, combining last three formulas we have

$$\forall i, v'. \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \big(\mathsf{Owner}(i, \hat{V}) \text{ @ } u \vee \mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u \vee \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u\big) \\ \supset \neg\mathsf{Write}(i, M.ram.keyloc, v') \text{ on } (u', u]$$

which can be rearranged as

$$\forall i, v', u_w. \neg\mathsf{Reset}() \text{ on } (u', u] \wedge \mathsf{Write}(i, M.ram.keyloc, v') \text{ @ } u_w \supset \\ \neg\mathsf{Owner}(i, \hat{V}) \text{ @ } u \wedge \neg\mathsf{Owner}(i, \widehat{tpm}) \text{ @ } u \wedge \neg\mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \qquad (W1)$$

As shown above, we also have

$$(m = null) \vee \exists k, u'''.\ (u' < u''' \leq u'') \wedge \mathsf{Write}(k, M.ram.keyloc, m) \text{ @ } u'''$$

Clearly

$$m = null \wedge (v \geq u) \supset \neg\mathsf{Contains}(m, key(n), S) \ @ \ v$$

In the other scenario ($m \neq null$), using the above result $W1$ and thread $k$ writing to memory location $M.ram.keyloc$ we get $\neg\mathsf{Owner}(k, \hat{V}) \ @ \ u''' \wedge \neg\mathsf{Owner}(k, \widehat{tpm}) \ @ \ u''' \wedge \neg\mathsf{HT}(k, \hat{L}, \mathtt{LOGGER}) \ @ \ u''' \wedge \mathsf{Write}(k, M.ram.keyloc, m) \ @ \ u'''$.

We can now use $Ax2$ and $H1$ to claim $\mathsf{Has}(k, m) \ @ \ u'''$ from $\mathsf{Write}(k, M.ram.keyloc, m) \ @ \ u'''$. As $u''' < u \leq u_0$ we can invoke $\mathsf{keyOwnerIn}(u''')$

$$\begin{aligned}\mathsf{keyOwnerIn}(u''') = \forall u', m, n, i, S.\ &(u' \geq u''') \wedge \mathsf{Has}(i, m) \ @ \ u''' \wedge \mathsf{Contains}(m, key(n), S) \ @ \ u' \\ &\wedge \neg(tpm.key \in S) \supset \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \ @ \ u''' \vee \mathsf{Owner}(i, \widehat{tpm}) \ @ \ u''' \vee \mathsf{Owner}(i, \hat{V}) \ @ \ u'''\end{aligned}$$

to claim that (using $v \geq u > u'''$)

$$\begin{aligned}&\neg\mathsf{HT}(k, \hat{L}, \mathtt{LOGGER}) \ @ \ u''' \wedge \neg\mathsf{Owner}(k, \widehat{tpm}) \ @ \ u''' \wedge \neg\mathsf{Owner}(k, \hat{V}) \ @ \ u'' \wedge (v \geq u''') \\ &\quad \wedge \mathsf{Has}(k, m) \ @ \ u''' \wedge \neg(tpm.key \in S) \supset \neg\mathsf{Contains}(m, key(n), S) \ @ \ v\end{aligned}$$

Thus, we infer $\neg\mathsf{Contains}(m, key(n), S) \ @ \ v$ which can be weakened to $\neg\mathsf{Contains}(m, key(n), S) \ @ \ v \vee \neg\mathsf{Read}(i, M.ram.keyloc, m) \ @ \ u$ for any $i$, and is independent of all existentially chosen fresh variables. Thus, we infer

$$\begin{aligned}\gamma \vdash \forall u, v, S, i, n, m.\ &\mathsf{Early}(u) \wedge (v \geq u) \wedge tpm.key \notin S \supset \\ &\neg\mathsf{Contains}(m, key(n), S) \ @ \ v \vee \neg\mathsf{Read}(i, M.ram.keyloc, m) \ @ \ u\end{aligned}$$

which can be strengthened to obtain $E$.

We skip the proof of $\neg E$ here, but, state that the proof relies on the fact that the logger has write lock on $M.ram.keyloc$ and only the logger ever reads that location.

## E.2 Proving property 3 and 4

Next, we prove property 3 and 4, i.e., $\forall u. u > u_a \supset \mathsf{oldKeyAdv}(u, u_a) \wedge \mathsf{oldKeyNotInMem}(u, u_a)$.

We do transfinite induction on $u$. We first prove the base case where $u = u_a$. $\mathsf{oldKeyAdv}(u_a, u_a)$ holds as the antecedent is false because of $\mathsf{keyOwnerIn}(u)$. We prove $\mathsf{oldKeyNotInMem}(u_a, u_a)$ next.

By analyzing the program of logger we have

$$\begin{aligned}&\forall i, u_l, k, k', k_e, S, m.\ (k \geq 1) \wedge \mathsf{LastLogIdx}(k, u_l, u_a) \wedge \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \ @ \ u_a \wedge \\ &\quad \mathsf{inMem}(i, key(k_e)) \ @ \ u_a \wedge (1 \leq k' \leq k) \wedge \mathsf{inMem}(i, m) \ @ \ u_a \supset k_e \geq k + 1 \wedge \neg\mathsf{Contains}(m, key(k'), S) \ @ \ u_a\end{aligned}$$

From program analysis of tpm we have

$$\begin{aligned}&\forall i, u_l, k, k', k_e, S, m.\ (k \geq 1) \wedge \mathsf{LastLogIdx}(k, u_l, u_a) \wedge \mathsf{Owner}(i, \widehat{tpm}) \ @ \ u_a \\ &\quad \mathsf{inMem}(i, key(k_e)) \ @ \ u_a \wedge (1 \leq k' \leq k) \wedge \mathsf{inMem}(i, m) \ @ \ u_a \supset k_e \geq k + 1 \wedge \neg\mathsf{Contains}(m, key(k'), S) \ @ \ u_a\end{aligned}$$

Combined with the already proven $\mathsf{keyOwnerIn}(u_a)$ we get that

$$\begin{aligned}\forall i, u_l, k, k', S, m.\ &(k \geq 1) \wedge \mathsf{LastLogIdx}(k, u_l, u_a) \wedge (1 \leq k' \leq k) \wedge \neg\mathsf{Owner}(i, \hat{V}) \\ &\wedge \mathsf{inMem}(i, m) \ @ \ u_a \wedge tpm.key \notin S \supset \neg\mathsf{Contains}(m, key(k'), S) \ @ \ u_a\end{aligned}$$

The above can be simplified by noting that $\forall l. \exists i.\ \mathsf{CanRead}(i, l)$ and verifier cannot read memory in machine $M$ to get

$$\begin{aligned}\forall l, u_l, k, k', S, m.\ &(k \geq 1) \wedge \mathsf{LastLogIdx}(k, u_l, u_a) \wedge (1 \leq k' \leq k) \\ &\wedge (l \in M) \wedge \mathsf{Mem}(l, m) \ @ \ u_a \supset \neg\mathsf{Contains}(m, key(k'), S) \ @ \ u_a\end{aligned}$$

which proves $\mathsf{oldKeyNotInMem}(u_a, u_a)$.

For the induction hypothesis, assume that $\mathsf{oldKeyNotInMem}(u, u_a) \wedge \mathsf{oldkeyAdv}(u, u_a)$ for all $u_a \leq u < u_0$ for some $u_0$. We need to show $\mathsf{oldKeyNotInMem}(u_0, u_a) \wedge \mathsf{oldkeyAdv}(u_0, u_a)$.

**$\mathsf{oldKeyNotInMem}(u_0, u_a)$ holds:** We do so by contradiction.
Assume the formula $\mathsf{oldKeyNotInMem}$ does not hold for $u_0$, then we obtain a contradiction. The negation of $\mathsf{oldKeyNotInMem}(u_0, u_a)$ is

$$\exists l_0, u_l, k, k', S_0, m.\ (u_0 \geq u_a) \wedge (k \geq 1) \wedge P(k, u_l, u_a) \wedge (1 \leq k' \leq k) \wedge \mathsf{Mem}(l_0, m) \mathbin{@} u_0 \wedge \\ \mathit{tpm.key} \notin S \wedge \mathsf{Contains}(m, \mathit{key}(k'), S_0) \mathbin{@} u_0$$

Choose fresh values for existential and let the above formula be $\psi$. From the induction hypothesis we know $\mathsf{oldKeyNotInMem}(u_a, u_a)$ is true, thus,

$$\mathsf{Mem}(l_0, m) \mathbin{@} u_0 \wedge \mathsf{Mem}(l_0, m') \mathbin{@} u_a \supset m \neq m'$$

Thus,

$$\exists j_0, u_w.\ (u_a < u_w < u_0) \wedge \mathsf{Write}(j_0, l_0, m) \mathbin{@} u_w$$

Thus, from $Ax2$ and $H1$ we get

$$\exists j, u_w.\ (u_a < u_w < u_0) \wedge \mathsf{Write}(j_0, l_0, m) \mathbin{@} u_w \wedge \mathsf{Has}(j_0, m) \mathbin{@} u_w$$

Now, from $\mathsf{oldkeyAdv}(u_w, u_a)$ combined with $\psi$ and $\mathsf{Has}(j_0, m) \mathbin{@} u_w$ it is possible to say that

$$\mathsf{HT}(j_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_w \vee \mathsf{Owner}(j_0, \widehat{tpm}) \mathbin{@} u_w \vee \mathsf{Owner}(j_0, \hat{V}) \mathbin{@} u_w \qquad (j_0 threads)$$

It is possible to prove by analyzing programs that

$$\gamma, \psi \vdash \forall k', k, l, m, u, u_l, i, v, S.\ (u_a < u \leq u_0) \wedge (u \leq v) \wedge \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u \wedge P(k, u_l, u_a) \wedge \\ (1 \leq k' \leq k) \wedge \mathsf{Write}(i, l, m) \mathbin{@} u \wedge \mathsf{Contains}(m, \mathit{key}(k'), S) \mathbin{@} v \wedge \neg(\mathit{tpm.key} \in S) \supset k' \geq k + 1$$
$$\gamma, \psi \vdash \forall k', k, l, m, u, i, v, S.\ (u_a < u \leq u_0) \wedge (u \leq v) \wedge \mathsf{Owner}(i, \widehat{tpm}) \mathbin{@} u \wedge \\ P(k, u_l, u_a) \wedge (1 \leq k' \leq k) \wedge \mathsf{Write}(i, l, m) \mathbin{@} u \wedge \mathsf{Contains}(m, \mathit{key}(k'), S) \mathbin{@} v \wedge \neg(\mathit{tpm.key} \in S) \supset k' \geq k + 1$$
$$\gamma, \psi \vdash \forall k', k, l, m, u, i, v, S.\ (u_a < u \leq u_0) \wedge (u \leq v) \wedge \mathsf{Owner}(i, \hat{V}) \mathbin{@} u \wedge P(k, u_l, u_a) \wedge \\ (1 \leq k' \leq k) \wedge \mathsf{Write}(i, l, m) \mathbin{@} u \wedge \mathsf{Contains}(m, \mathit{key}(k'), S) \mathbin{@} v \wedge \neg(\mathit{tpm.key} \in S) \supset k' \geq k + 1$$

The above proof relies on the induction hypothesis $\mathsf{oldKeyAdv}$ and $\mathsf{oldKeyNotInMem}$ to ensure that the log entries do not contain keys with index lower than $k+1$. Then, instantiating $i$ with $j_0$, $l$ with $l_0$, $m$ with $m$, $v$ with $u_0$ and $S$ with $S_0$, and combined with $\psi$ and $(j_0 threads)$ we conclude that $k' \geq k + 1$, but, we also have from $\psi$ that $k' \leq k$ which is contradiction.

**Proving $\mathsf{oldKeyAdv}(u_0, u_a)$ holds:** We do so by contradiction.
Assume the formula $\mathsf{oldKeyAdv}(u_0, u_a)$ does not hold, then we show a contradiction.

$$\exists i_0, m, v, S_0, k_a, u_0, u_l.\ (v \geq u_0 \geq u_a) \wedge \mathsf{Has}(i_0, m) \mathbin{@} u_0 \wedge \mathsf{Contains}(m, \mathit{key}(k_a), S_0) \mathbin{@} v \wedge \neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_0 \\ \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \mathbin{@} u_0 \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \mathbin{@} u_0 \wedge P(k, u_l) \wedge \mathit{tpm.key} \notin S_0 \wedge (k_a \geq 1) \wedge (k_a < k + 1)$$

Denote above by $\psi$. Pick the existential values (denote them by the same variable names). From $\mathsf{Has}$ we infer that in past at time $u_r$, $i_0$ read (or hmaced) a message $m'$ from a location $l$ that contained $m$ and $\neg\mathsf{Reset}() \in [u_r, u_0]$. The receive is ruled out, as by induction hypothesis non-logger , non-verifier and non-tpm threads do not have $\mathit{key}(k_a)$ as $k_a < k + 1$, and the logger, verifier and tpm do not send out the key. We reason about the read case below, deriving a contradiction. (The hmac case is similar so we skip it).

Since the message read $m'$ contained $m$ we know there exists a $S'$ such that $\mathsf{Contains}(m', m, S')$. Also, by the security of the tpm key we have $\mathit{tpm.key} \notin S'$. As argued in previous part we have $\mathsf{Contains}(m', \mathit{key}(k_a), S' \cup S_0) \mathbin{@} v$. The $\neg\mathsf{Reset}() \in [u_r, u_0]$ with the assumed assertion that $i_0$ is not the logger, tpm or verifier at time $u_0$ implies

$$\neg\mathsf{HT}(i_0, \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_r \wedge \neg\mathsf{Owner}(i_0, \widehat{tpm}) \mathbin{@} u_r \wedge \neg\mathsf{Owner}(i_0, \hat{V}) \mathbin{@} u_r$$

Consider two cases $u_r \leq u_a$ and $u_r > u_a$. We first derive a contradiction from the first case. From $\mathsf{keyOwnerIn}(u_r)$ we know that

$$\forall v', S, i'.\ (u_r \leq u_a) \wedge \mathsf{Has}(i', m') \mathbin{@} u_r \wedge (\mathit{tpm.key} \notin S) \wedge \neg\mathsf{HT}(i', \hat{L}, \mathtt{LOGGER}) \mathbin{@} u_r \wedge \neg\mathsf{Owner}(i', \widehat{tpm}) \mathbin{@} u_r \\ \wedge \neg\mathsf{Owner}(i', \hat{V}) \mathbin{@} u_r \wedge (v \geq u_r) \supset \neg\mathsf{Contains}(m', \mathit{key}(k_a), S) \mathbin{@} v'$$

Instantiate $v'$ with $v$ and $S$ with $S' \cup S_0$ and $i'$ with $i_0$. Thus, the precedent is true (using $\psi$) and we get $\neg\mathsf{Contains}(m', key(k_a), S' \cup S_0)$ @ $v$, but, we already concluded that $\mathsf{Contains}(m', key(k_a), S' \cup S_0)$ @ $v$. Thus, we have a contradiction.

Next, consider the case when read time $u_r > u_a$. Let the location read be $l$. Thus, $\mathsf{Has}(i_0, m)$ @ $u_r$. Then, as before there is a set $S'$ such hat

$$tpm.key \notin S' \cup S_0 \wedge \mathsf{Mem}(l, m) \text{ @ } u_r \wedge \mathsf{Contains}(m, key(k_a), S' \cup S_0) \text{ @ } u_0$$

However, this violates the induction hypothesis $\mathsf{oldKeyNotInMem}(u_r, u_a)$ for $u_r < u_0$. Thus, $u_r$ must be $u_0$. Thus, $\mathsf{Mem}(l, m)$ @ $u_0$ $\wedge$ $\mathsf{Contains}(m, key(k_a), S_0)$ @ $u_0$. From the induction hypothesis $\mathsf{oldKeyAdv}(u)$ we know that $\mathsf{Contains}(m, key(k_a), S_0)$ @ $u_0 \supset \neg\mathsf{Mem}(l, m)$ @ $u$ where $u < u_0$. Thus, we get

$$(u < u_0) \wedge \mathsf{Mem}(l, m) \text{ @ } u_0 \wedge \neg\mathsf{Mem}(l, m) \text{ @ } u$$

Using $Ax1$ we get

$$\exists j_0, u'.\ (u < u' \leq u_0) \wedge \mathsf{Write}(j_0, l, m) \text{ @ } u'$$

Using $Ax2$ we get (with the $j_0$ above)

$$\exists u'', u'.\ (u < u'' < u' \leq u_0) \wedge \mathsf{Has}(j_0, m) \text{ @ } u'' \wedge \mathsf{Write}(j_0, l, m) \text{ @ } u' \wedge \neg\mathsf{Reset}() \text{ on } [u'', u']$$

Now, as in the last part, from $\mathsf{oldKeyAdv}(u'', u_a)$ combined with $\psi$ and $\mathsf{Has}(j_0, m)$ @ $u''$ it is possible to say that

$$\mathsf{HT}(j_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u'' \vee \mathsf{Owner}(j_0, \widehat{tpm}) \text{ @ } u'' \vee \mathsf{Owner}(j_0, \hat{V}) \text{ @ } u''$$

Using the above assertion that $\wedge \neg\mathsf{Reset}()$ on $[u'', u']$ we get

$$\mathsf{HT}(j_0, \hat{L}, \mathtt{LOGGER}) \text{ @ } u' \vee \mathsf{Owner}(j_0, \widehat{tpm}) \text{ @ } u' \vee \mathsf{Owner}(j_0, \hat{V}) \text{ @ } u'$$

Now, just as in the last part (by analyzing programs) we can show that $k_a \geq k + 1$, but, we already assert that $k_a < k + 1$, which is a contradiction.

## E.3 Main Result

Finally, we prove the main result of the paper

$$\begin{aligned}
\gamma \vdash & \forall k, k', u_b, u_e, u_l, u_r, u_w, i, j, log, n, fhm.\ \mathsf{HT}(i, \hat{V}, \mathtt{VERIFIER}) \text{ on } [u_b, u_e] \wedge \\
& (u_b < u_c < u_r < u_v < u_e) \wedge \mathsf{Send}(i, \mathit{VERIFY})@u_b \wedge \mathsf{New}(i, nonce)@u_c \wedge \\
& \mathsf{Recv}(i, (log[n], n, fhm)) \text{ @ } u_r \wedge \mathsf{VerifyHmac}(i, fhm, nonce, key(n+1))@u_v \wedge \\
& \big((u_r \leq u_a) \supset \mathsf{LastLogIdx}(k, u_l, u_r)\big) \wedge \big((u_r > u_a) \supset \mathsf{LastLogIdx}(k, u_l, u_a)\big) \wedge \\
& (1 \leq k' \leq k) \wedge (u_l \geq u_w) \wedge \mathsf{Write}(j, fileloc(k'), v) \text{ @ } u_w \\
& \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_w \supset data(v) = data(log(k'))
\end{aligned}$$

where

$$\begin{aligned}
\mathsf{LastLogIdx}(k, u, u_{end}) = \ & (k \geq 0) \wedge (u \leq u_{end}) \wedge \\
& \forall u', i, j.\ \mathsf{HT}(i, \hat{L}, \mathtt{LOGGER}) \text{ @ } u \wedge (u \leq u' \leq u_{end}) \wedge (k \geq 1) \wedge \mathsf{Write}(i, fileLoc(k), v) \text{ @ } u \wedge \\
& \qquad \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u' \supset \neg\mathsf{Write}(j, fileLoc(k+1), v) \text{ @ } u'
\end{aligned}$$

We assume that the required to prove formula does not hold and derive a contradiction. The negation is

$$\begin{aligned}
\gamma \vdash & \exists k, k', u_b, u_e, u_l, u_r, u_w, i, j, log, n, fhm.\ \mathsf{HT}(i, \hat{V}, \mathtt{VERIFIER}) \text{ on } [u_b, u_e] \wedge \\
& (u_b < u_c < u_r < u_v < u_e) \wedge \mathsf{Send}(i, \mathit{VERIFY})@u_b \wedge \mathsf{New}(i, nonce)@u_c \wedge \\
& \mathsf{Recv}(i, (log[n], n, fhm)) \text{ @ } u_r \wedge \mathsf{VerifyHmac}(i, fhm, nonce, key(n+1))@u_v \wedge \\
& \big((u_r \leq u_a) \supset \mathsf{LastLogIdx}(k, u_l, u_r)\big) \wedge \big((u_r > u_a) \supset \mathsf{LastLogIdx}(k, u_l, u_a)\big) \wedge \\
& (1 \leq k' \leq k) \wedge (u_l \geq u_w) \wedge \mathsf{Write}(j, fileloc(k'), v) \text{ @ } u_w \\
& \wedge \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER}) \text{ @ } u_w \wedge (data(v) \neq data(log(k')))
\end{aligned}$$

Denote the above as $\psi$ after picking fresh values for the existential with same variable names, except $j_0$ for the variable $j$. By the definition of $\mathsf{LastLogIdx}$ and $\psi$ we get that $u_l \leq \min(u_r, u_a)$.

Also, by analyzing the program of the verifier we know that

$$\begin{aligned}&\mathsf{HT}(i, \hat{V}, \mathtt{VERIFIER}) \text{ on } [u_b, u_e] \wedge \mathsf{VerifyHmac}(i, \mathit{fhm}, \mathit{nonce}, \mathit{key}(n+1))@u_v \wedge @\ u \supset \\ &\quad \exists u.\ (u_r < u < u_v) \wedge \mathsf{VerifyHmac}(i, \mathit{data}(k')_{mac}, \mathit{data}(k')_{data}, \mathit{key}(k'))\ @\ u\end{aligned}$$

Using the hmac axioms $Ax9$ and $Ax10$ for $\mathsf{VerifyHmac}(i, \mathit{data}(k')_{mac}, \mathit{data}(k')_{data}, \mathit{key}(k'))$ and the fact that data was received at time $u_r$ we get

$$\begin{aligned}&\mathsf{VerifyHmac}(i, fhm, nonce, key(n+1))\ @\ u_v \supset \exists j, l, u'.(u' < u_r) \wedge \mathsf{Hmac}(j, \mathit{data}(k')_{data}, l, \mathit{data}(k')_{mac})\ @\ u' \\ &\quad \wedge \mathsf{inMem}(j, \mathit{key}(k'))\ @\ u'\end{aligned}$$

Let the above witness for $j$ be $j'$. First, we show that $u' \leq u_a$. Assume the contrary $u' > u_a$, which also implies that $u_r > u_a$. Hence, $\mathsf{LastLogIdx}(k, u_l, u_a)$ holds, and $\mathsf{inMem}(j, \mathit{key}(k'))\ @\ u'$ implies $\exists l.\ \mathsf{Mem}(l, \mathit{key}(k'))\ @\ u'$. We show using $\mathsf{oldKeyNotInMem}(u', u_a)$ a contradiction.

From $\mathsf{oldKeyNotInMem}(u', u_a)$ we can claim that $k' > k$. To get this, re-arrange $\mathsf{oldKeyNotInMem}$ to move the $\neg\mathsf{Contains}$ to antecedent as $\neg\mathsf{Contains}$ and the $k' \leq k$ to consequent as $k' > k$.

$$\begin{aligned}\mathsf{oldKeyNotInMem}(u, u_a) = {}& \forall l, u_l, k, k', S, m.\ (u \geq u_a) \wedge (k \geq 1) \wedge \mathsf{LastLogIdx}(k, u_l, u_a) \wedge \\ & \wedge (1 \leq k') \wedge \mathsf{Mem}(l, m)\ @\ u \wedge \mathit{tpm.key} \notin S \wedge \mathsf{Contains}(m, \mathit{key}(k'), S)\ @\ u \\ & \supset (k' > k)\end{aligned}$$

We already showed that $\mathsf{LastLogIdx}(k, u_l, u_a)$ and $\mathsf{Mem}(l, \mathit{key}(k'))\ @\ u'$, and take $S = \phi$. Hence, we can infer that $k' > k$. However, we asserted above that $k' \leq k$ which is a contradiction. Thus, $u' \leq u_a$.

It is possible to show that

$$\forall j.\ \mathsf{Hmac}(j, \mathit{data}(k')_{data}, l, \mathit{data}(k')_{mac})\ @\ u' \supset \neg\mathsf{Owner}(j, \widehat{tpm})\ @\ u' \wedge \neg\mathsf{Owner}(j, \hat{V})\ @\ u'$$

by analyzing the tpm and verifier program. It is also possible to show using $\mathsf{keyOwnerIn}(u')$ that

$$\mathsf{inMem}(j', \mathit{key}(k'))\ @\ u' \wedge (u' \leq u_a) \supset \mathsf{HT}(j', \hat{L}, \mathtt{LOGGER})\ @\ u' \vee \mathsf{Owner}(j', \widehat{tpm})\ @\ u' \vee \mathsf{Owner}(j', \hat{V})\ @\ u'$$

Thus, we obtain $\mathsf{HT}(j', \hat{L}, \mathtt{LOGGER})\ @\ u'$. Also, from the logger program it can be shown that

$$\begin{aligned}&\forall j, u'.\ \mathsf{HT}(j, \hat{L}, \mathtt{LOGGER})\ @\ u' \wedge \mathsf{Hmac}(j, \mathit{data}(k')_{data}, l, \mathit{data}(k')_{mac})\ @\ u' \\ &\quad \wedge P(k, u_l, u_a) \wedge (1 \leq k' \leq k) \wedge (u' \leq u_a) \\ &\qquad \supset \exists u'_w.\ \mathsf{Write}(j, \mathit{fileloc}(k'), \mathit{data}(k'))\ @\ u'_w \wedge (u' < u'_w \leq u_l) \wedge \neg\mathsf{Reset}() \text{ on } [u', u'_w] \wedge \\ &\qquad \forall j'', u, v'.\ (u \leq u_a) \wedge (u \neq u'_w) \wedge \mathsf{HT}(j'', \hat{L}, \mathtt{LOGGER})\ @\ u \supset \neg\mathsf{Write}(j'', \mathit{fileloc}(k'), v')\ @\ u\end{aligned}$$

Instantiate $j = j'$ and $u' = u'$ in the above formula.

The antecedent holds by our assumptions and hence does the conclusion. We already have $\mathsf{HT}(j_0, \hat{L}, \mathtt{LOGGER})\ @\ u_w$ and $u_w \leq u_a$ and $\mathsf{Write}(j_0, \mathit{fileloc}(k'), v)\ @\ u_w$, which means that a possible value for $u'_w$ is $u_w$, Now, the forall in the consequent implies that $u'_w = u_w$ since for no other value of $u$ is there a write to $\mathit{fileloc}(k')$. Also, from $\neg\mathsf{Reset}()$ on $[u', u_w]$ and $\mathsf{HT}(j', \hat{L}, \mathtt{LOGGER})\ @\ u'$ we get that $\mathsf{HT}(j', \hat{L}, \mathtt{LOGGER})\ @\ u_w$, but, we already have $\mathsf{HT}(j_0, \hat{L}, \mathtt{LOGGER})\ @\ u_w$ (from $\psi$). Thus, $j' = j_0$.

Again from $\mathsf{Write}(j', \mathit{fileloc}(k'), \mathit{data}(k'))\ @\ u'_w$ we get $\mathsf{Write}(j_0, \mathit{fileloc}(k'), \mathit{data}(k'))\ @\ u_w$, which combined with $\mathsf{Write}(j_0, \mathit{fileloc}(k'), v)\ @\ u_w$ means $v = \mathit{data}(k')$. However, we also have $v \neq \mathit{data}(k')$ (from $\psi$). Thus, we have a contradiction.

## F Protocol Encoding

**Logger program**

```
LOGGER =
call logger_start();
call logger_body();
logger_start() $\triangleq$
   call setup_loggerchannel($\_self$);
   writelock $M.ram.keyloc$;
   readlock $M.ram.keyloc$;
   writelock $M.disk.sealedkey$;
   readlock $M.disk.sealedkey$;
//handles shutdown attack
   $s$ = read $M.disk.sealedkey$;
   $key$ = call unseal_data($s$);
   counterup $M.tpm.nv.counter.1$; //counter 1
   write $M.ram.keyloc, key$;
   write $M.ram.logger\_os, -1$;
logger_body() $\triangleq$
   $shstatus$ = read $M.ram.sh$; //shutdown status
   $key$ = read $M.ram.keyloc$;
   $data$ = recv;
   if ($data == LOGGING\_DONE$
         $\wedge\ shstatus == -1$)
     then call logger_shutdown();
   else
     if ($data == VERIFY$)
       then call verify_logger();
     else
       $hm$ = hmac $data, key$;
       $key'$ = hash $key$;
       write $M.ram.keyloc, key'$;
       $y$ = read $M.disk.currentpos$; //current pointer
       write $M.disk.y, (data, hm)$;
       write $M.disk.currentpos, y + 1$;
verify_logger() $\triangleq$
   $curr$ = read $M.disk.currentpos$;
   $start$ = read $M.disk.startoffile$;
   $log$ = read $M.disk.startoffile$; //reads whole file
   $nonce$ = recv;
   $hm$ = hmac $nonce, key$;
   send($log, curr - start, hm$); //file, size, hm
logger_shutdown() $\triangleq$
   $s$ = read $M.tpm.nv.counter.1$;
   $key$ = read $M.ram.keyloc$;
   $sk$ = call seal_data($key, 1, s + 1$);
   write $M.disk.sealedkey, sk$;
   write $M.ram.logger\_sh, -1$;
```

```
unseal_data($s$) $\triangleq$
   write $M.tpm.v.channel$, $UNSEAL\_COMMAND$;
     //notify TPM of command
   write $M.ram.logger\_tpm, -2$;
     //wait for response of TPM
   poll $M.ram.logger\_tpm, -1$;
     //put data on channel
   write $M.tpm.v.channel, s$;
     //notify TPM of data
   write $M.ram.logger\_tpm, 1$;
     //wait for response of TPM
   poll $M.tpm.logger\_tpm, 2$;
     //read response of TPM
   $x$ = read $M.tpm.v.channel$;
   $x$; //return x
seal_data($data, counterindex, counter$) $\triangleq$
   write $M.tpm.v.channel$, $SEAL\_COMMAND$;
   write $M.ram.logger\_tpm, -2$;
     //wait for response of TPM
   poll $M.ram.logger\_tpm, -1$;
     //put data on channel
  write $M.tpm.v.channel$,
          ($data, counterindex, counter$);
     //notify TPM of data
   write $M.ram.logger\_tpm, 1$;
     //wait for response of TPM
   poll $M.tpm.logger\_tpm, 2$;
     //read response of TPM
   $x$ = read $M.tpm.v.channel$;
   $x$;
```

**OS program**

```
OS =
call init();
$id$ = fork $LOGGER, \hat{L}$;
writelock $M.ram.logger\_os, id$;
write $M.ram.logger\_os, -2$;
poll $M.ram.logger\_os, -1$;
writelock $M.ram.sh\_os$;
write $M.ram.sh\_os, -2$;
$id'$ = fork SHUTDOWN, $\widehat{root}$; //enable shutdown
poll $M.ram.sh\_os, -1$;
fork PRODUCER;
$e'$;

SHUTDOWN =
writelock $M.ram.sh$;
//only the thread that locks $ram.sh$ can reset
write $M.ram.sh\_os, -1$;
write $M.ram.sh, -1$;
//write -1 non-deterministically at any time
poll $M.ram.logger\_sh, -1$; //wait for logger
reset;
```

**Verifier program**

```
VERIFIER =
send (VERIFY, 0);
key = hash SS;
nonce = new;
send nonce;
(alldata, size, finalhm) = recv;
call verifier_loop(0, key, size, nonce, finalhm, alldata);

verifier_loop(n, key, size, nonce, finalhm, alldata) ≜
    if (n == size)
    then verifyhmac finalhm, nonce, key;
    else
      (log, hm) = alldata[n];
      verifyhmac hm, log, key;
      key' = hash key;
      call verifier_loop(n + 1, key', size, nonce, finalhm)
```

**TPM**

```
TPM =
    call tpm_body();

tpm_body() ≜
      //private key of TPM
    prvkey = read M.tpm.nv.locked.prvkey;
      //wait for command
    poll M.ram.logger_tpm, −2;
    command = read M.tpm.v.channel;
    write M.ram.logger_tpm, −1;
    poll M.ram.logger_tpm, 1;
    if (command == SEAL_COMMAND) then
      (data, counterindex, count) = read M.tpm.v.channel;
      s = seal data, counterindex, count, prvkey;
      write M.tpm.v.channel, s;
    else if (command == UNSEAL_COMMAND)
      s = read M.tpm.v.channel;
      data = unseal s, prvkey;
      write M.tpm.v.channel, data;
     //notify caller that done
    write M.ram.logger_tpm, 2;
    call tpm_body();

setup_loggerchannel(id) ≜
//take various locks on communication channels
    writelock M.tpm.v.channel, id;
    readlock M.tpm.v.channel, id;
    writelock M.ram.logger_tpm, id;
    readlock M.ram.logger_tpm, id;
    writelock M.ram.logger_sh, id;
```