

FastVer2: A Provably Correct Monitor for Concurrent, Key-Value Stores

Arvind Arasu
Microsoft Research

Tahina Ramananandro
Microsoft Research

Aseem Rastogi
Microsoft Research

Nikhil Swamy
Microsoft Research

Aymeric Fromherz
INRIA

Kesha Hietala
University of Maryland

Bryan Parno
CMU

Ravi Ramamurthy
Microsoft Research

Abstract

FastVer [4] is a protocol that uses a variety of memory-checking techniques to monitor the integrity of key-value stores with only a modest runtime cost. Arasu et al. formalize the high-level *design* of FastVer in the F* proof assistant and prove it correct. However, their formalization did not yield a provably correct *implementation*—FastVer is implemented in unverified C++ code.

In this work, we present FastVer2, a low-level, concurrent implementation of FastVer in Steel, an F* DSL based on concurrent separation logic that produces C code, and prove it correct with respect to Arasu et al.’s high-level specification. Our proof is the first end-to-end system proven using Steel, and in doing so we contribute new ghost-state constructions for reasoning about monotonic state. Our proof also uncovered a few bugs in the implementation of FastVer.

We evaluate FastVer2 by comparing it against FastVer. Although our verified monitor is slower in absolute terms than the unverified code, its performance also scales linearly with the number of cores, yielding a throughput of more than 10M op/sec. We identify several opportunities for performance improvement, and expect to address these in the future.

1 Introduction

Consider a system consisting of a key-value store service and a set of clients, whose interactions are represented by a trace of request-response pairs of the form `put k v` (to update the value of key `k` to `v`) or `get k v` (to fetch the current value of `k`, to which the service’s response is `v`). The clients may be skeptical that the service properly stores and retrieves their data, either due to the service operator being malicious or due to bugs in the implementation of the service. We would like to offer clients a guarantee such as sequential consistency, i.e., that every `get` on a key `k` returns the value of the most recent preceding `put k v`.

One way to approach this goal is to formally verify the implementation of the service and prove that it always ensures this property, and then to prove to the client (e.g., using a cryptographically authenticated trusted execution environment (TEE)) that the service is running exactly the verified

code. However, a formal proof of a high-performance, concurrent, key-value store is difficult and generally requires redesigning the service from scratch to enable its proof. Besides, even state of the art formally verified implementations of key-value stores do not yet handle concurrency [12]. Furthermore, formally verifying the code does not exclude the possibility of attacks resulting from tampering with the data storage system directly.

Instead, Arasu et al. [4] propose to attach a cryptographic monitor to an existing key-value store service (with minimal changes to the implementation of the service). Their goal is to ensure that a *monitored execution* of the service is provably sequentially consistent, except for some cryptographic gap, e.g., due to a hash collision. The monitor is able to certify execution logs as being sequentially consistent within some chosen latency window, a parameter of the system. This is a weaker guarantee than our stated goal, but one that is achievable at a much lower cost than verifying the service outright, while also protecting against malicious service operators tampering with the state. The challenge is to design a monitor that can detect violations of sequential consistency with a low runtime cost.

At one end of the spectrum is a monitor that views the service as a black box and simply monitors the log of interactions of all the clients, somehow deciding whether or not the interaction so far is valid. Black box monitoring has obvious benefits in that it requires no changes to the service; however, it is hard to make a black box monitor efficient.

Instead, Arasu et al. propose a protocol called FastVer, which allows the untrusted service to interactively convince the monitor that the operations it has processed are valid, and, if convinced, the monitor can attest to the validity of the interactions so far by, for example, signing valid operations. FastVer is designed to be efficient, enabling the monitor and the service to interact via several concurrent *verifier threads*. Additionally, FastVer offers a suite of techniques that enable trading off latency of verification with throughput of the system, e.g., FastVer can certify a log of operations asynchronously, guaranteeing that all operations up to some *epoch* are correct, with the remainder still to be processed.

Arasu et al. formalize a high-level design of FastVer in F* [22], a proof-oriented programming language, and prove that a monitored execution of a key-value store using FastVer is sequentially consistent up to some epoch (in the sense that there exists an interleaving of all the client operations such that each get returns the value of the most recent preceding put on the same key), otherwise constructing a cryptographic hash collision. Although this provides good confidence in the correctness of the protocol, the proof is based on a functional specification that is optimized for clarity and ease of proof rather than efficient, concurrent execution.

1.1 FastVer2: From Design to Implementation

This paper presents FastVer2, a provably correct, low-level, concurrent implementation of FastVer. FastVer2’s executable code consists of about 4,600 lines of concurrent C code extracted from Steel [10], an F* DSL based on a concurrent separation logic called SteelCore [24]. The development consists of about 22,000 lines of proof-oriented code, in addition to about 3,000 lines of C code for formally verified parsers and serializers produced by the EverParse parser generator [21], and about 950 lines of C code for verified cryptographic primitives from HACLS* [28]. We have submitted all of this code as non-anonymous supplementary materials. Our main contributions are highlighted below.

New ghost-state abstractions for concurrency proofs.

We design and implement a scheme for multiple verifier threads to compute concurrently and for them to periodically share their results for aggregation. To structure our proofs of this concurrency pattern, we rely on SteelCore’s support for ghost state based on partial commutative monoids (PCMs), and introduce a new PCM for refined, monotonic state called the *fractional anchored preorder*, or FRAP.

An incremental, asynchronous API. We design a monitor API that is suitable for incremental use by the service. The service can repeatedly choose a thread on which to run a verifier, feeding it a trace of operations to be verified. Separately and asynchronously, the context can query the monitor, possibly on a separate thread, to determine the epoch up to which verification has completed.

Hardware protections from a TEE. Overall, FastVer2’s design philosophy is to *verify the verifier*. That is, FastVer’s monitor offers a runtime memory verification protocol, while FastVer2 ensures that that protocol is implemented correctly. FastVer2’s C code has no external runtime dependences and is designed to be executable within a TEE, such as an Intel SGX enclave. Our top-level API is also designed for *safe* usage in an untrusted context. As such, based on trust in the hardware, a skeptical client can be sure that the execution of an untrusted service is correctly monitored by high-assurance, formally proven code.

Evaluation. We evaluate FastVer2 by integrating our monitor with the Faster key-value store [7]. We find that the throughput of Faster with our verified monitor is 5x less than the throughput of Faster with the unverified FastVer monitor. Even then, FastVer2 achieves a throughput exceeding 10M ops/sec, which is 2-3 orders of magnitude better any existing formally verified key-value database [12]. As with FastVer, the throughput of FastVer2 scales roughly linearly with the number of CPU cores. We identify the reasons for FastVer2’s performance drop (the main reason being FastVer’s use of hardware-accelerated cryptography, which FastVer2 lacks), and ways to bridge the gap.

2 Goals and High-level Design

In this section we introduce some basic concepts of the FastVer protocol. We convey some of the challenges in achieving a high-performance, concurrent monitor and provide some intuitions for the guarantees offered by the protocol. As we will see, the argument for why FastVer is correct is subtle, and involves reasoning about the interaction of several concurrent sub-protocols. We also summarize our goals for the verified implementation of the FastVer monitor.

2.1 A Client’s View of the System

Consider a set of clients \overline{C} , where each client C_i has a unique identifier i , interacting with a key-value store service. Typically, for scalability, the service is implemented using multiple instances or threads and multiple clients can interact with the service concurrently. Nevertheless, clients expect the service to behave as a single logical key-value store.

From C_i ’s perspective, its interactions with the service S so far can be modeled as a log of executed operations $\mathcal{L}_i = \overline{op}$, tagged with the client’s identifier i and a client-specific sequence number n , where the sequence number of an item op in \mathcal{L}_i is greater than the sequence number of all preceding items. The operations op themselves are either:

- $get_{i,n} k v$, a request to fetch the current value of a key k with the service’s response v ; or,
- $put_{i,n} k v$, a request to update the value of k to v

When the client identifier and sequence numbers are irrelevant, we simply write $get k v$ and $put k v$. Additionally, we write $op_{i,n}$ to mean some operation from client i with sequence number n . Collectively, the state of all the clients \overline{C} is represented by all their logs $\overline{\mathcal{L}}$.

Our overall goal is to guarantee that all the clients logs $\overline{\mathcal{L}}$ can be interleaved in a sequentially consistent manner.

Definition 2.1. Formally, we say that a log of client operations \mathcal{I} is *sequentially consistent* if and only if

- For every $op_{i,n} \in \mathcal{I}$, if it is preceded by an operation $op_{i,m} \in \mathcal{I}$, then $m < n$, i.e., the sequence numbers for each client i are monotonically increasing; and,

- For every *get* $k v$ in \mathcal{I} at position n , there exists a preceding operation *put* $k v$ in \mathcal{I} at position $m < n$ and none of the operations in \mathcal{I} between $m + 1$ and n are of the form *put* $k _$, i.e., each *get* returns the value of the most recent *put* on the same key.

FastVer2 Trust Assumptions (Attacker Model). Clients should not need to place any trust in the implementation of the service S nor in any modifications made to S in support of its integration with the FastVer2 monitor. However, clients must be assured that the implementation of the monitor M correctly enforces the desired safety property, namely, it only accepts sequentially consistent executions of the service. To provide this assurance, we formally prove the correctness of the low-level, concurrent implementation of M in F^* . As such, clients only need to trust the *specification* of the main theorem of our development, the F^* toolchain, which includes the Z3 SMT solver [9], as well as the C compiler we use to compile the code emitted by F^* .

Further, although the implementation of the monitor is verified, to ensure that the runtime execution environment of the monitor is trustworthy, we execute the fully verified code of the monitor within a trusted execution environment (TEE). As such, based on trust in the underlying hardware protection mechanisms, the client can assume that the monitor is indeed executing only the verified code. This also protects against bugs in the untrusted service from compromising the correctness of the monitor, since hardware protections isolate S and M .

Finally, the main guarantee offered by the system is that a client's C_i 's execution trace, when interleaved with the execution traces of an arbitrary context of other clients \bar{C} , is sequentially consistent. We need to assume that the untrusted server S itself is not among the client context \bar{C} , since otherwise S can always inject messages to justify otherwise faulty behavior observed by C_i . For example, C_i 's trace *put* $k 5$, *get* $k 6$ could be justified by an S itself injecting a spurious *put* $k 6$ between C_i 's two messages. One way to enforce that $S \notin \bar{C}$ is by requiring all client messages to be authenticated using keys issued by some key-issuing authority that all clients trust, though there are several other possible schemes to achieve this. Such protections are beyond the scope of this paper and we instead focus on proving the implementation of the monitor correct.

2.2 The Design of the Monitor

FastVer2 inherits the high-level design of the monitor from FastVer. The monitor M is multi-threaded and consists of several *verifier threads* \bar{V} . The number of verifier threads is an initialization parameter—verifier threads cannot be created dynamically after initialization. Each verifier thread V_i has a unique identifier i and maintains some thread local state. The verifier threads can interact with each other through some verifier shared state A (for aggregate state). Taken together,

the verifier's local and aggregate state are an authenticated abstraction [26] of the entire state of S , using a combination of hardware protection and cryptographic techniques.

Verifier Logs. The service S is modified to interact with the monitor by periodically sending a log of operations L_i to some verifier thread V_i , e.g., by calling an operation *verify-log* $i L_i$ on the monitor. To process L_i , the verifier thread V_i resumes executing in the TEE and processes each operation in L_i sequentially. These logs include client-facing operations *op* that the service S has processed, and it is V_i 's task to certify that *op* was executed correctly by the service; it does so by evaluating *op* with respect to the current authenticated abstraction of the service state. However, in addition to the two client operations *get* and *put*, the verifier offers an API with *seven additional operations*, which the service uses to safely manipulate the verifier's authenticated state and to prove to the verifier that the client's operation was indeed processed correctly. Two of these operations, *NextEpoch* and *VerifyEpoch*, are related to a verifier's notion of logical time, which we discuss in detail shortly. We touch on the remaining operations, which involve various checks and state updates, and if any of them fail, verification aborts and the verifier logs are no longer certified.

To maintain its authenticated data structure, FastVer uses three ingredients:

1. **A thread-local verifier cache** Each verifier thread uses a small amount of hardware-protected memory to store a cache of records currently being processed. The client operations (*get* and *put*) are evaluated with respect to this cache, and the rest of the protocol makes sure that the records needed by the client operations are loaded and evicted as necessary.
2. **A Sparse, Incremental Merkle Tree** The primary mechanism to authenticate the contents of the entire key-value store is a Merkle tree [16]. The tree initially authenticates the initial state of the entire key-value store and the root of the tree is stored in the cache of a designated verifier thread, e.g., V_0 . Two of the seven operations in the FastVer API (*AddM* and *EvictM*) are related to manipulating the interaction between the Merkle tree and cache.
3. **Deferred Memory Checking using Multiset Hashing** Finally, to more efficiently support concurrent access of records in multiple threads, FastVer uses a memory checking technique developed initially by Blum et al. [5], but with several enhancements based on partitioning logs into several epochs and using collision-resistant multiset hashing. Two further operations (*AddB* and *EvictB*) are related to Blum-style memory checking, and an additional hybrid operation (*EvictBM*) supports interactions between the Merkle and Blum schemes.

Each of these techniques on its own presents a formalization challenge, particularly for an efficient, concurrent implementation. The combination of the three, together with the subtle ways in which they interact, is decidedly non-trivial. However, Arasu et al. model FastVer in F^* and prove its *design* correct—we summarize their results next, on which we build a verified *implementation* of the monitor.

2.3 Formalizing the Correctness of FastVer

Arasu et al.’s development has the following main elements (we use the concrete syntax of F^* here, which is modeled after OCaml, and we explain potentially unfamiliar notation as we go; Appendix A provides a brief syntax primer):

Single-thread specification. Arasu et al. provide a functional specification of a single verifier thread as a function `verify_model`, which processes a log of entries provided by the service while transforming its thread-local state `vs:tsm` (for thread-state model).

```
val verify_model (vs:tsm) (log:seq log_entry) : tsm
```

The thread-local state `tsm` contains a number of fields—we show its main elements below, including a fixed `thread_id`; a bit `failed` to record if log verification has failed; and a cache representing the hardware-protected, thread-local verifier cache. The internal state of each verifier thread also includes a logical clock, consisting of a non-negative integer *epoch* `e` and a non-negative integer *tick* counter `c`, where the pair (e, c) increases monotonically, in lexicographic order. Operations processed by a thread are grouped into epochs, and for each epoch the verifier maintains some `epoch_hashes`.

```
type thread_state_t = {
  thread_id:tid; failed:bool; cache:store;
  clock:timestamp; epoch_hashes:epoch_hashes;
  last_verified_epoch:option epoch_id; ... }
```

The `log_entry` type includes the `Get k v` and `Put k v` messages, as well as the seven additional operations provided by FastVer. Of the seven, the main one of interest in this paper is `VerifyEpoch`, a message that requests the verifier to complete verifying all the messages since the last verified epoch. Each time a verifier thread processes a `VerifyEpoch` entry, it advances its last verified epoch, and needs to synchronize with other verifier threads to collectively determine whether or not all operations in that epoch were correct.

Main theorem: Sequentially consistent except for hash collisions. The main theorem offered by Arasu et al. is a correctness property for the execution of multiple verifier threads. We summarize their theorem below:

Given the logs for each of the verifier threads, the state of all the threads is described by `verifier_states`

```
let verifier_states (logs:seq (seq log_entry)) : seq tsm =
  Seq.mapi (λ i → verify_model (init i)) logs
```

The predicate `epoch_ok i logs` (below) asserts that all entries in the logs up to epoch `i` have been verified. This involves running the verifiers on their logs, conceptually in parallel. If none of the verifiers have failed, and if all of them have advanced their `last_verified_epoch` beyond epoch `i`, then we compute an aggregation of the hashes they have computed for that epoch, the `add_set_hash` and the `evict_set_hash`, and if those two hashes are equal, then we can declare that epoch `i` has been verified.

```
let epoch_ok (i:epoch) (logs:seq (seq log_entry)) =
  let vs = verifier_states logs in
  (∀ v ∈ vs. (¬ v.failed) ∧ epoch_is_complete i v) ∧
  let add_set_hash, evict_set_hash =
    aggregate (Seq.map (λ v → v.epoch_hashes.[i]) vs) in
  add_set_hash == evict_set_hash
```

Without further details about the FastVer protocol itself, these checks may seem arbitrary. However, for our purposes, it is sufficient to rely on Arasu et al.’s main theorem, which states that if the logs have been validated up to epoch `i`, then if they are not sequentially consistent we can construct a hash collision, i.e., a pair of values v_1 and v_2 such that $v_1 \neq v_2$ but `hash v1 = hash v2`.

```
let seq_consistent (logs:seq (seq log_entry)) = ... // Definition 2.1
val not_sc_implies_hash_collision (i:epoch)
  (logs:seq (seq log_entry)) {epoch_ok i logs ∧
    ¬ (seq_consistent_up_to i logs)}
  : hash_collision
```

3 Implementing FastVer Correctly

The top-level API of FastVer2 contains three functions: `init` initializes N verifier threads; `verify_log` allows the context (the untrusted service) to pass in a log of operations to be verified on a given thread $i < N$; and `max_certified_epoch` allows the context to query the monitor (on some thread) to determine the maximum epoch up to which the logs have been checked so far.

To state the correctness of this concurrent API, and to maintain its core invariants, we needed to build some specification and proof libraries in Steel. We start our description of FastVer2 by presenting its state and synchronization patterns informally, motivating a new ghost-state construction, the *fractional anchored preorder* PCM.

3.1 Shape of the State

Figure 1 depicts, informally, the state of the system. We have several verifier threads. Each thread’s state is described by a ghost log of operations that it has processed so far; as a thread processes an operation, it updates this log to indicate the progress. These thread-specific ghost logs are the backbone of our invariant: we use them to state the main functional correctness property of each thread, namely that a thread’s current state is related to what is computed by the

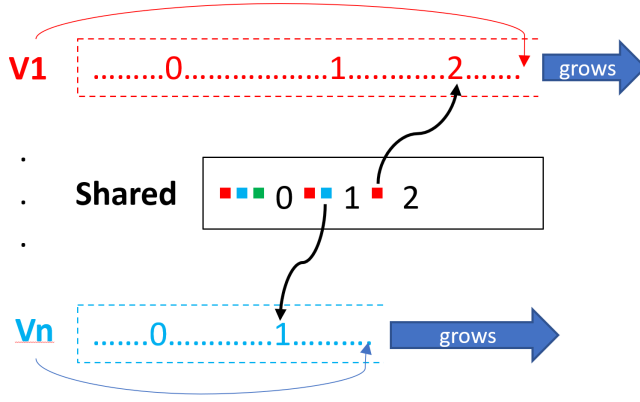


Figure 1. A sketch of FastVer2’s state: Each verifier thread V_i processes and updates its own (ghost) log of operations (each entry is a thread-specific colored dot). At epoch boundaries, each thread copies some of its local state to concrete shared state (the thread-specific colored squares) for aggregation and epoch certification. Here, V_1 (resp. V_n) has completed verification up to epoch 2 (resp. 1) and is processing entries beyond it. Ghost references from the shared state represent the most recent epoch that each thread has completed.

verify_model specification on the ghost log of operations it has processed so far.

In addition to the thread-local state, we have some concrete state shared among all the threads. Each time a thread processes a VerifyEpoch operation to complete an epoch, it copies, or *commits*, some of its local state (the add and evict hashes for that epoch) to the shared state. An invariant (shown in the figure by the black (ghost) pointers) relates the shared state to the thread-local logs. For example, the figure shows that the shared state is up to date with the log of V_1 for all epochs up to epoch 2, and epoch 1 for V_n . If all threads have committed their add and evict hashes for an epoch e to the shared state, another thread can combine and check those hashes and signal whether epoch e has been certified.

Some state transitions. When a thread V_i processes an operation, it updates its log to record this fact. As such, V_i requires write permission to its log. However, ghost references from the shared state also reference this log. In most traditional ownership-based program logics, this poses a challenge: updates to the state require exclusive ownership, yet here V_i needs to update its log, even though it does not exclusively own it. However, in this case V_i only *appends* to its log, i.e., the log grows monotonically. As such, as long as the invariant on the shared state only references a *prefix* of V_i ’s log and is stable with respect to log append operations, V_i should be able to safely update its ghost state.

SteelCore, like some other modern separation logics, supports user-defined ghost state based on partial commutative

monoids (PCMs). These ghost state abstractions allow expressing various kinds of spatial and temporal disciplines on multiple components (e.g., threads or modules) to express knowledge about shared resources. One such abstraction is a PCM for monotonic state that supports such knowledge-preserving, shared state updates—others have explored such constructions before [3, 18, 27], including in libraries for Steel [24] that we use here. A PCM for monotonic state also enables taking logical *snapshots* of the system, which is useful to make irrevocable assertions that, say, all operations processed by the system up to some epoch have been certified and will remain so regardless of how the system evolves. This enables the monitor to issue definitive persistent attestations of these facts using, say, digital signatures.

However, just a regular monotonic state PCM is not sufficient here. Consider what happens when a thread V_i reaches an epoch boundary, completing epoch e . At this point, it needs to update the shared state for epoch e and the ghost references pointing back to its log from the shared state to reflect that the shared state is synchronized with its log up to epoch e . However, to do this in an invariant-preserving way, thread V_i has to “know” that the shared state is already synchronized with its state all the way up to its previous epoch $e - 1$. As such, it is not enough for the invariant to only maintain that the shared state is consistent with a prefix of V_i ’s log; we need to prove that the shared state is not “too far behind” the state of each thread.

One possibility is to record in concrete state the last epoch that each thread has committed to the shared state, and when committing a given epoch e , we use a runtime check to confirm that the e is indeed the next epoch. However, such a runtime check is inefficient, and besides, we can prove that it is not necessary—we just need the right ghost-state abstraction. Towards this end, we design and implement a PCM, which we call the *fractional anchored preorder*, or FRAP, and use it to model FastVer2’s ghost state. The next three sections, §3.2, §3.3, and §3.4, describe this construction and require some familiarity with separation logic—readers interested more in our main theorem and less in its proof could skip ahead to §3.5

3.2 The Fractional Anchored Preorder PCM

A PCM in Steel is a typeclass. A `pre_pcm` provides operations to compose elements of `a` that are composable, and a unit for the operation `one`. The type `pcm` `a` associates with `pre_pcm` properties that ensure that the operation is associative and commutative, that `one` is really a unit, and a further property, `refine` which distinguishes a subtype of `a`, which we’ll see in use, shortly.

```
type pre_pcm a = { composable: symrel a; one:a;
                  op: x:a → y:a{composable x y} → a; }
type pcm a = { p:pre_pcm a; comm:commutative p; assoc: assoc p;
              is_unit: is_unit p; refine: a → prop }
```

Given a $p:\text{pcm } a$, Steel allows allocating a ghost reference $r:\text{G.ref } p$, a reference to mutable ghost state holding a value of type $v:a$, and to state assertions of the form $\text{G.pts_to } r \ v$, as in the case of G.alloc below, where ghost references can be initialized with refined values from a PCM p [24].

```
val G.alloc (#a:Type) (#p:pcm a) (v:a { p.refine v }) : STG (G.ref p)
  (requires emp)
  (ensures  $\lambda r \rightarrow \text{G.pts\_to } r \ v$ )
```

The signature shown above is a specification in Steel, where the type $\text{STG } t \ p \ q$ is a *computation type* (similar to Hoare Type Theory [17]) describing a total correctness specification of a *ghost* computation which when called in an initial state validating the separation logic proposition p , returns a value of type $v:t$ in a final state validating $q \ v$. In Steel, the type of separation logic propositions is $v\text{prop}$, and so, $p:v\text{prop}$ and $q:t \rightarrow v\text{prop}$. To emphasize the role of p and q as pre- and postconditions, we often decorate them with the F^* keywords *requires* and *ensures*, respectively. The $\#$ -sign marks an implicit argument in F^* . We often omit implicit binders, adopting a convention that unbound names are implicitly universally bound at the top of the definition. Steel also offers the computation type $\text{ST } a \ p \ q$, a Hoare-style partial correctness specification for a concrete (non-ghost) computation.

The crucial bit with ghost references to PCMs is that $\text{G.pts_to } r \ (v_0 \ \text{op} \ v_1)$ is equivalent to $\text{G.pts_to } r \ v_0 * \text{G.pts_to } r \ v_1$, as the following two lemmas show.

```
val share (r:G.ref p) : STG unit
  (requires G.pts_ro r (v_0 `op` v_1))
  (ensures  $\lambda\_ \rightarrow \text{G.pts\_to } r \ v_0 * \text{G.pts\_to } r \ v_1$ )

val gather (r:G.ref p) : STG unit
  (requires G.pts_ro r v_0 * G.pts_to r v_1)
  (ensures  $\lambda\_ \rightarrow \text{G.pts\_to } r \ (v_0 \ \text{op} \ v_1)$ )
```

That is, knowledge that r holds a composite value $v_0 \ \text{op} \ v_1$ can be traded back and forth with separate knowledge about each component. To enable this, SteelCore requires that every update to a reference be *frame-preserving*, i.e., knowing $\text{G.pts_to } r \ k$, an update should preserve all assertions $\text{G.pts_to } r \ k'$ that are compatible with k .

The FRAP PCM. To define the FRAP PCM, we need some auxiliary notions of preorders and anchors. A preorder v is a reflexive, transitive binary relation on v . The $\text{anchor_rel } p$ is more interesting, and restricts the preorder as shown below. It may provide useful intuitions to think of $\text{anchor_rel } p$, a binary relation on v , as a kind of measure of “distance”. With that, the first conjunct in the refinement of anchors below states that if v_1 is not too far ahead of v_0 , then v_1 is also related to v_0 by the preorder p . The second conjunct says that if z is not too far from x , then all y that are between x and z according to the preorder are also not too far ahead of x . Note, $\text{anchor_rel } p$ is not itself a preorder—it is certainly not transitive, and we don’t even require that it be reflexive.

```
let anchor_rel (#v:Type) (p:preorder v) = anchors:(v  $\rightarrow$  v  $\rightarrow$  prop) {
```

```
( $\forall v_0 \ v_1. v_0 \ \text{anchors} \ v_1 \implies p \ v_0 \ v_1$ )  $\wedge$ 
( $\forall x \ z. x \ \text{anchors} \ z \implies (\forall y. p \ x \ y \ \wedge \ p \ y \ z \implies x \ \text{anchors} \ y)$ )
}
```

Next, the carrier type knowledge a of the PCM is shown below, where Nothing will be the unit of our PCM, and $\text{Owns } a \ v$ represents some non-trivial knowledge.

```
type knowledge (a:anchor_rel p) =
  | Owns : avalue a  $\rightarrow$  knowledge a
  | Nothing : knowledge a
```

An av : $\text{avalue } a$ (defined below) is a pair (perm, h) of a permission perm and a $h:\text{vhist } p$, a type which encodes the entire p -preorder compatible history of values that will be stored at a ghost reference for this PCM. The $\text{vhist } p$ construction existed previously in the Steel libraries and provides a generic way to turn a preorder p into a PCM—see §4.5 of Swamy et al. [24]. We don’t say much more about it here, except that $\text{cur } v$ is the most recent value in the history. What is more interesting is the structure of $\text{perm} = (\text{op}, \text{oa})$, itself a pair of an optional fractional permission, where None represents a 0 permission (useful for snapshots, as we will see). Further, when $\text{oa} = \text{Some } a$, we say that the av has an anchor a . The anchored $a \ av$ refinement states that if av has an anchor, then its current value is “not too far ahead” of the anchor.

```
let permission v = option perm & option v
let anchored (arel:anchor_rel p) (pv:(permission v & vhist p)) =
  match pv with | (_, Some a), v  $\rightarrow$  a `arel` cur v | _  $\rightarrow$   $\top$ 
let avalue a = av:(permission v & vhist p) { anchored a av }
```

Composability. The composability of v_0, v_1 : $\text{avalue } a$ is the key bit of the whole PCM, since it defines when one entity’s knowledge is compatible with another’s.

```
let avalue_composable ((p_0, h_0) (p_1, h_1): avalue arel) =
  permission_composable p_0 p_1  $\wedge$  (h_0  $\geq$  h_1  $\vee$  h_1  $\geq$  h_0)  $\wedge$ 
  (match p_0, p_1 with
  | (None, None), (None, None)  $\rightarrow$   $\top$ 
  | (None, None), (Some _, _)  $\rightarrow$  h_1  $\geq$  h_0
  | (None, None), (_, Some a)  $\rightarrow$  h_0  $\geq$  h_1  $\implies$  a `arel` cur h_0
  | (Some _, _), (Some _, _)  $\rightarrow$  h_0 == h_1
  | (Some _, _), (_, Some a)  $\rightarrow$  h_0  $\geq$  h_1  $\wedge$  a `arel` cur h_0
  | ... remaining cases are symmetric
```

Suppose $v_0 = (p_0, h_0)$ and $v_1 = (p_1, h_1)$. To start with, p_0 is composable with p_1 if the sum of their fractions is no more than 1 and, importantly, if *at most one* of them has an anchor. Further, h_0 and h_1 must at least be composable in terms of the regular definition of composability of $\text{vhist } p$, the history-based PCM for preorders, i.e., one of them must be an extension of the other, effectively forbidding “forks” in the history. The rest of the definition is by case analysis on the permissions. If neither p_0 or p_1 holds any permission then no further constraints apply. Next, if $p_0 = \text{None, None}$ then we have two sub-cases: if p_1 holds some non-zero fraction, then h_1 must be more recent than h_0 ; otherwise, if p_1 holds an anchor a ,

and if h_0 is more recent than h_1 , then its current value is still anchored by v , i.e., holding an anchor, even with a zero fraction, still prevents the state from evolving too far from the anchor. Finally, if both p_0 and p_1 hold some permission, then we have two sub-cases: if they both hold non-zero fractions, then $h_0 == h_1$; otherwise, exactly one of them can hold an anchor, and the other must hold a non-zero fraction (due to the composability of p_0 and p_1). If p_0 holds the anchor (call it a), then the history of h_1 must be more recent than h_0 but still be anchored by a . Remaining cases are symmetric, since composability must be a symmetric relation for a PCM. Note the interplay between fractions, preorders, and anchors—we don't think there is a way to derive the FRAP from smaller, orthogonal PCMs.

Composition. Once composability is settled, defining the composition of knowledge a is fairly obvious—Nothing is the unit, and to compose $v_0, v_1: \text{avalue } a$, we compose their permissions (summing their fractional permissions and retaining whichever anchor is present) and their histories (taking the most recent history).

Proving that all these definitions produce a PCM is almost entirely automated by F^* and its Z3-assisted [9] typechecker—the hard part was settling on the definitions. What is more interesting are the lemmas that one can now prove about mutually compatible forms of knowledge.

Some properties of snapshots. A snapshot of av retains its value while dropping all its permissions. For a value a , provided $\text{perm_ok } a$ (meaning that its fractional permission does not exceed 1, a basic well-formedness property), the lemma below proves that 1. one can always take a snapshot of a value a , since $a == a \text{ `compose` snapshot } a$, the share operation on a ghost reference can always be applied; 2. that snapshots are duplicable; and 3. that any knowledge b composable with snapshot a , provided b holds some non-zero permission (and hence is not a snapshot itself), must have a current value related to the snapshot by the preorder, i.e., snapshots remain valid in the face of preorder-preserving updates to the state. The lemma proof is automatic.

```

700 let value_of av = cur (snd av)
701 let snapshot (av: avalue s) : avalue s = (None, None), snd av
702 let snapshot_props (a: avalue s { perm_ok a }): Lemma (
703   a `composable` snapshot a  $\wedge$ 
704   a `compose` snapshot a == a  $\wedge$ 
705   snapshot a `composable` snapshot a  $\wedge$ 
706   snapshot a `compose` snapshot a == snapshot a  $\wedge$ 
707   ( $\forall$  (b: avalue s { has_perm b  $\wedge$  b `composable` snapshot a)).
708     value_of (snapshot a)  $\leq$  value_of b)) = ()
709

```

Some properties of anchors. While snapshots enable describing knowledge about *some* history of the system, anchors allow speaking about *recent* histories. The function `split_anchor av` splits the permissions associated with av into a

fractional part and an anchored part. Lemma `split_anchor_props` states that one can always split knowledge of av into these two parts without losing any information. The most interesting part is `elim_anchor`: it states that if any (non-snapshot) knowledge a composable with anchored knowledge b , then a 's value is “not too far ahead” of b 's anchor. The proofs are again automatic.

```

716 let split_anchor (((p, a), v): avalue s) = ((p, None), v), ((None, a), v)
717 let split_anchor_props (av: avalue s { perm_ok av })
718   : Lemma (let kv, ka = split_anchor av in
719     kv `composable` ka  $\wedge$  kv `compose` ka == av) = ()
720 let elim_anchor (a: avalue s { has_perm a })
721   (b: avalue s { has_anchor b  $\wedge$  composable a b })
722   : Lemma (let (_, Some anc), _ = b in s anc (value_of a)) = ()
723

```

Appendix B describes the usage of a FRAP for a simple scenario independent of FastVer2, involving two threads sharing a monotonic counter.

3.3 A FRAP for FastVer2

With our generic FRAP construction in place, we turn to its instantiation in FastVer2. The specific anchor relation we use is shown below, where $\downarrow l$ is the greatest prefix of l that ends with a `VerifyEpoch` entry, or the empty log if there is no such entry, i.e., it precisely captures the state of a verifier thread that has been committed to the shared state. Note that `is_last_committed` is not reflexive in general: only empty logs or logs that end with a `VerifyEpoch` can be anchors, a useful property, as we will see shortly.

```

734 let log_grows : preorder log =  $\lambda l_0 l_1 \rightarrow l_0 \leq l_1$ 
735 let is_last_committed  $l_0 l_1 = l_0 \leq l_1 \wedge l_0 == \downarrow l_1$ 
736

```

F^* can automatically prove that `is_last_committed` has type `anchor_rel log_grows` which makes it easy to construct `log_frap`, a FRAP instance for this relation. This gives us a FRAP for a single log—to get a FRAP for all the logs, we use a Steel library to compose PCM's pointwise to obtain a `tlm` (for thread-log map), a FRAP PCM for a map from `tid` (thread ids) to knowledge (`avalue is_last_committed`), the carrier of `log_frap`.

```

737 let log_frap = frap is_last_committed
738 let tlm = PCMMMap.pointwise tid log_frap
739 let tlm_carrier = map tid (knowledge (avalue is_last_committed))
740

```

Finally, to model FastVer2's ghost state, we introduce `mlogs = G.ref tlm`, the type of a ghost reference that holds a thread-log map. This allows us to form assertions of the form `G.pts_to r m`, where $m : \text{tlm_carrier}$. However, it is more convenient to work with several derived abstract predicates shown below.

Abstract predicates for thread-log maps. The predicate `tids_pts_to r frac m anchor` asserts `frac` knowledge on the state of all the threads, where `sel m tid = Some log` states that `tid` has processed exactly `log`, since `frac` is non-zero. Further, if anchor is true, then we can conclude that `log` is also exactly the

committed state of tid . $\text{tid_pts_to } r \text{ tid } f \text{ l } a$ is just the singleton variant of tids_pts_to . We use tid_pts_to to express a thread’s knowledge of its own state, i.e., in terms of Figure 1, tid_pts_to expresses the knowledge held in the thread-specific ghost references associated with V_1, \dots, V_n .

```

776 let lmap = map tid (option log)
777 let rel_dom (l:lmap) (k:tlm_carrier) =
778    $\forall \text{tid. Some? (sel l tid)} \iff \text{Owns? (sel k tid)}$ 
779 let tids_pts_to (r:mlogs) (frac:perm) (m:lmap) (anchor:bool) =
780    $\exists k. G.\text{pts\_to } r \text{ k} * \text{pure (rel\_dom m k} \wedge \text{perm\_ok (Some frac)} \wedge$ 
781      $(\forall \text{tid. Owns? (sel k tid)} \implies$ 
782        $\text{perm\_of k tid} == \text{Some frac} \wedge$ 
783        $(\text{anchor} \iff \text{has\_anchor k tid}) \wedge$ 
784        $(\text{get\_cur (sel k tid)} == \text{Some?.v (sel m tid)}))$ 
785 let tid_pts_to r tid f l a = tids_pts_to r f (singleton tid l) a

```

We can also define predicates that hold only anchored or snapshot knowledge, without holding any fraction of the current knowledge of the thread state: global_anchor expresses anchored knowledge about all the threads (i.e., a recent history), while global_snapshot expresses knowledge of some history of the threads. We elide their definitions, which are similar in structure to tids_pts_to .

```

794 val global_anchor (l:mlogs) (m:lmap) : vprop
795 val global_snapshot (l:mlogs) (m:lmap) : vprop

```

In terms of Figure 1, global_anchor expresses the knowledge held in the black references that point back to the thread logs from the shared state. We will see a use of global_snapshot in §3.5.

Valid triples and ghost-state transitions. Given the lemmas (like snapshot_props and $\text{split_anchor_props}$) we’ve proven on FRAPs and also lemmas available for PCM maps, we can derive the following (ghost) operations—these are lemmas that prove the validity of certain Hoare triples, or from the perspective of a logic like Iris [15] one may think of ghost operations as view shifts.

Our first pair of lemmas shows that non-anchored fractional knowledge can be split and recombined, like standard fractional permissions [6], but applied to our abstract predicate. The converse lemma that sums fractions is also provable, but we don’t show it here.

```

813 val share_tids_pts_to (x:mlogs) (m:lmap) : STG unit
814   (requires tids_pts_to x f m false)
815   (ensures  $\lambda\_ \rightarrow \text{tids\_pts\_to } x \text{ (half\_perm } f) \text{ m false} *$ 
816      $\text{tids\_pts\_to } x \text{ (half\_perm } f) \text{ m false}$ )

```

Next, take_tid shows that one can extract knowledge about a single thread from collective knowledge about multiple threads—a fact that follows from structure of PCM maps. The converse lemma is also provable, though we do not show it.

```

822 val take_tid (x:mlogs) (m:lmap) (t:tid {Some? (sel m t)}) : STG unit
823   (requires tids_pts_to x f m false)
824   (ensures  $\lambda\_ \rightarrow \text{tid\_pts\_to } x \text{ t f (Some?.v (sel m t)) false} *$ 

```

```

826    $\text{tids\_pts\_to } x \text{ f (upd m t None) false}$ )

```

Next, following from $\text{split_anchor_props}$ and elim_anchor , take_anchor shows that one can combine knowledge of the current state of a thread t ’s log l with an anchor, taking ownership of the anchor from the global_anchor and conclude the anchor $a = \text{Some?.v (sel m t)}$ is the committed prefix of l .

```

831 val take_anchor x m (t:{Some? (sel m t)}) f l : STG unit
832   (requires  $\text{global\_anchor } x \text{ m} * \text{tid\_pts\_to } x \text{ t f l false}$ )
833   (ensures  $\lambda\_ \rightarrow \text{global\_anchor } x \text{ (upd m t None) } *$ 
834      $\text{tid\_pts\_to } x \text{ t f l true} * \text{pure } (\Downarrow = \text{Some?.v (sel m t)})$ )

```

Conversely, a thread can cede ownership of an anchor back to the global_anchor , provided its current state l is fully committed.

```

840 val put_anchor x m t f (l:{l=l}) : STG unit
841   (requires  $\text{tid\_pts\_to } x \text{ t f l true} * \text{global\_anchor } x \text{ m}$ )
842   (ensures  $\lambda\_ \rightarrow \text{tid\_pts\_to } x \text{ t f l false} *$ 
843      $\text{global\_anchor } x \text{ (upd m t (Some l))}$ )

```

Next, update_log shows that if a thread owns full non-anchored permission to a log, it can extend the log so long as it does not include any more VerifyEpoch entries—this corresponds to the state transitions that an individual thread makes as it processes log entries and updates its thread-local state, without needing to synchronize with the shared state.

```

851 val update_log x t l0 (l1:log {l0 ≤ l1 ∧  $\Downarrow l_0 == \Downarrow l_1$ }) : STG unit
852   (requires  $\text{tid\_pts\_to } x \text{ t full } l_0 \text{ false}$ )
853   (ensures  $\lambda\_ \rightarrow \text{tid\_pts\_to } x \text{ t full } l_1 \text{ false}$ )

```

On the other hand, if a thread needs to update its log with a VerifyEpoch entry, it must hold anchored knowledge of its state and advance both its state and the anchor simultaneously. In other words, when processing a VerifyEpoch , a verifier thread V_i must first take knowledge of its anchor from the shared state using take_anchor , then transition using $\text{update_anchored_log}$, and finally return knowledge of the updated anchor to the shared state using put_anchor .

```

863 val update_anchored_log x t l0 (l1:{l0 ≤ l1 ∧  $\Downarrow l_1 == l_1$ }) : STG unit
864   (requires  $\text{tid\_pts\_to } x \text{ t full } l_0 \text{ true}$ )
865   (ensures  $\lambda\_ \rightarrow \text{tid\_pts\_to } x \text{ t full } l_1 \text{ true}$ )

```

3.4 FastVer2 Main Data Structures and Invariants

All the state of the FastVer2 monitor is held in a top_level_state structure, shown below.

```

870 type top_level_state = { aeh:aggregate_epoch_hashes;
871   all_threads:larray (thread_state_and_lock aeh.mlogs) n_threads }

```

The first field, $\text{aeh: aggregate_epoch_hashes}$, references mutable shared state containing the hashes computed for an epoch by each thread—the shared state at the center of Figure 1. The second field, all_threads is an array storing the thread-local state $\text{thread_state_and_lock aeh.mlogs}$ for each thread—its type is indexed by the shared state and the contents of the two related by an invariant, as we’ll soon see.

The top-level invariant, `core_inv t`, merely asserts some permission over the `t.all_threads` array and states that the thread with id `i:tid` is at position `i` in the array (`ind_ok`); the most interesting parts of the invariant are held in locks stored in other fields of the top-level state:

```
let core_inv t = ∃p v. A.pts_to t.all_threads p v * pure (ind_ok v)
```

The thread-local state is represented by the structure shown below, pairing the actual thread state `ts` with a lock, an instance of a *cancellable lock*, a wrapper around Steel’s verified implementation of a CAS-based spin lock, enabling the lock to be released while canceling the invariant that it protects (useful in case a verifier thread enters an unrecoverable error), while acquiring the lock only conditionally provides the invariant. Before calling any operation on a thread, this lock must be acquired, protecting against re-entrancy, and released before returning. Interestingly, the prior unverified implementation of FastVer neglected to protect against re-entrancy in this manner, leading to a potential source of attacks from a malicious service.

```
type thread_state_and_lock mlogs = {
  i: tid; ts: thread_state_t{thread_id ts == i};
  lock: Lock.cancellable_lock (∃ tsm. thread_inv ts mlogs tsm) }
```

The invariant held by the lock, `thread_inv`, states that the concrete state of the thread reachable from `ts` is a refinement of `ts.thread_state_model`, the functional specification of a verifier thread’s state from §2.2; the verifier thread has not failed yet; and, it holds half permission to a non-anchored knowledge of the shared state `map mlogs` to assert that it has processed all the entries as required by the functional specification `tsm.processed_entries`. As we will see, the other half permission is passed to the context and used to specify the top-level incremental API.

```
let thread_inv ts mlogs tsm =
  state_refinement ts tsm * pure (¬ tsm.failed) *
  tid_pts_to mlogs tsm.thread_id half tsm.processed_entries false
```

The `aeh:aggregate_epoch_hashes` field stores the shared epoch hashes and related metadata from each thread, all protected by a cancellable lock that protects the main invariant on the aggregate state.

```
type aggregate_epoch_hashes = {
  hashes : epoch_tid_hashes; bitmaps : epoch_tid_bitmaps;
  max : R.ref (option epoch_id); mlogs : mlogs;
  lock: Lock.cancellable_lock (agg_inv hashes bitmaps max mlogs) }
```

The invariant `agg_inv` states that the hashes and bitmaps are maps (implemented using new libraries for hash tables that we programmed in Steel) that point to logical witnesses `hv` and `bv` (the latter used to record which threads have completed a given epoch); that `max` points to some `max_v`; and, importantly, that all these values are related to the functional correctness specification by `hashes_bitmaps_max_ok` of

running all the verifiers on the logs `mlogs_v`, the logs corresponding to the last synchronized state of all the verifiers, expressed using the FRAP-based abstract predicate, `global_anchor`.

```
let agg_inv hashes bitmaps max mlogs = ∃hv bv max_v mlogs_v.
  EpochMap.pts_to hashes hv *
  EpochMap.pts_to full bitmaps bv *
  pts_to max full max_v *
  global_anchor mlogs (map_of_seq mlogs_v) *
  pure (hashes_bitmaps_max_ok hv bv max_v mlogs_v)
```

To update the aggregate state at the end of an epoch, a verifier thread must acquire `aeh.lock`, take ownership of its anchor, then update both the concrete and ghost state (the latter using `update_anchored_log`, as described in §3.3), put the updated anchor back, and release the lock. While this lock introduces some contention, we have not found this to be a performance bottleneck, since epoch boundaries are relatively sparse. Should this become an issue, with some more accounting, we believe verifier threads may propagate hashes into the aggregate state in lock-free manner.

3.5 Incremental API

We offer a precise, incremental API to the FastVer2 monitor intended for use by a *verified context* that also runs within the TEE. Such a verified context may provide services on top of the monitor, e.g., to issue signatures of attesting to the validity of client operations.

3.5.1 Initializing the monitor. Initializing the monitor involves calling `init ()`, which initializes `n_threads` (a compile-time constant) verifier threads.

```
let ilogs t = tids_pts_to t.aeh.mlogs half (const (Some empty)) false
val init (:_unit) : ST (ref top_level_state)
  (requires emp)
  (ensures λr → ∃ t. pts_to r full t * core_inv t * ilogs t)
```

Since its precondition is just `emp`, `init ()` can be called at any time, including multiple times. It returns a fresh handle to an instance of the monitor, `r:ref top_level_state`, a concrete reference to `top_level_state`, an abstract type, such that:

1. The caller has full permission to `r`, which points to some value `ts`, a logical witness to the value stored in the heap at `r`.
2. It (separately) provides an abstract predicate `core_inv ts`, encapsulating the main invariant of the monitor. The `core_inv ts` is duplicable, meaning that from `core_inv ts` it is possible to derive `core_inv ts * core_inv ts`.
3. And, finally, `ilogs` gives the caller half, non-anchored permission to the logs of all threads, each initialized to the empty sequence.

3.5.2 Verifying a log of operations. The signature of `verify_log` is shown below. It is invoked to request a given thread `i` to verify some input log of entries.

```

991 let log_of_tid t tid e = tid_pts_to t.aeh.mlogs i half e
992 val verify_log (r:R.ref top_level_state) (i:tid)
993   (input:larray U8.t len { len ≠ 0ul })
994   : ST (option verify_result)
995   (requires pts_to r p t * core_inv t * A.pts_to input lp lb *
996    log_of_tid t i ents)
997   (ensures λres →
998    pts_to r p t * core_inv t * A.pts_to input lp lb *
999    (match res with
1000     | Some (Verify_success read wrote) → ∃ents'.
1001      log_of_tid t i (ents @ ents') *
1002      pure (let s = verify_model (init_model tid) ents in
1003        let s' = verify_model s ents' in
1004          read == len ∧
1005          parse_log_up_to lb (U32.v read) == Some ents')
1006     | _ → ∃e'. log_of_tid t tid e'))

```

Precondition. To call `verify_log`, the context prepares a log of binary-formatted entries in an input array of bytes, whose length is `len`. They then call `verify_log r i input`, passing in the top-level state `handle r` and requesting that the input be processed on thread `i`. The precondition requires passing in some permission `p` to the top-level state `r` and `core_inv t`, i.e., read permission is enough, enabling the context to simultaneously call `verify_log r j` on some other thread, since `core_inv t` is also duplicable. Finally, the precondition includes `tid_pts_to` stating that thread `i` must have processed exactly `ents` so far, which the context can obtain by using `take_tid` on the `ilogs` predicate.

Postcondition. `verify_log` always returns back to the caller permission to `r`, the `core_inv`, and the *unchanged* input array. If it returns successfully with `Some (Verify_success read wrote)`, then we provide a full functional correctness specification of how the log was processed; otherwise, we leave failed runs underspecified. In the success case, we prove that the log processed by thread `i` is extended to `ents @ ents'`. The pure predicate relates these to the FastVer functional specification of a single thread, i.e., `verify_model`, which was outlined in §2.2. In addition, we prove that we read the entire input array, and that parsing that input array produces `Some ents'`, relating the binary format of the logs to the specification using `EverParse` [21].

3.5.3 Main theorem: Max certified epoch. The third and final operation in our top-level API is `max_certified_epoch`, which allows the context to request the monitor to aggregate the results from all verifier threads and report up to which epoch verification has been completed. As a precondition, the context only needs to provide some permission to `r`: `ref top_level_state` and the same permission is returned to the caller in the postcondition. The rest of the postcondition reflects the main partial correctness result of FastVer2, in case `max_certified_epoch r` returns `Read_max_some max`.

```

val max_certified_epoch (r:ref top_level_state)
  : ST max_certified_epoch_result
  (requires pts_to r p t)
  (ensures λres → pts_to r p t *
  match res with
  | Read_max_some max →
    ∃logs. global_snapshot t (map_of_seq logs) *
    pure (seq_consistent_except_if_hash_collision logs max)
  | _ → emp)

```

The predicate `global_snapshot t (map_of_seq logs)` says that all the threads have *at least* collectively processed the entries in `logs`, and since it is a FRAP snapshot, we can prove that `logs` is a valid history of the entire system. The rest of the postcondition relates this history to the correctness theorem of FastVer. That is, there exists an interleaving of all the get and put operations in the log, up to epoch `max` that is sequentially consistent, except if there is a hash collision—this is our main theorem.

3.6 A Verified Wrapper for TEEs

We want our API to protect from some misuse by an untrusted context: we should not return or receive pointers to our internal state; we should defend against the context passing bogus pointers for the input log; and we need to remove as many of the preconditions to our operations as possible, since an unverified context may not respect them. To defend against such misuse, we write and verify a small wrapper for our API that runs within the TEE and presents the same operations purged of these attack surfaces.

Top-level state. The API shall not pass a `ref top_level_state` back and forth across the boundaries of the TEE. Indeed, if it does, then this could allow an untrusted caller to pass in a bogus pointer, leading to crashes or, worse, memory corruption and wrong results.

To defend against this, we store the top-level state as a global variable. Allocating such a state will create a permission to access it later, but by default, if we carelessly perform that allocation as an F^* top-level variable, we lose that permission and we have no way to recover it. To avoid that situation, we store the permission in an *invariant* [25, §4.3 sqq.]:

This is enough, since once the top-level state is allocated, it is read-only. While the invariant can be temporarily opened only to perform at most one atomic observable operation, our setting is actually weaker: we open the invariant only to duplicate the permission on the top-level state, which is possible by halving the permission on the reference and duplicating `core_inv`. This is proof-only, ghost code, so it is not even observable. Thus, our invariant is transparent to the user and has no observable impact on concurrent accesses.

External input pointer. The service calls `verify_log` with a pointer to its input log. This exposes two problems. First,

nothing prevents the service from passing garbage; however, the TEE has a primitive to check the validity of the pointer provided by the service. So, our verified code needs to make sure to call this primitive before accessing the pointer. Second, nothing guarantees that the service will not write to the input buffer while we are accessing it; thus we cannot assume that two successive reads to a given byte in the input buffer will always return the same result. So we need to make sure to read each input byte at most once. While EverParse allows generating input data validators formally guaranteed against double fetches [23], these validators do not cover the subset of EverParse that we are using for our log types. So, we need to allocate a temporary buffer and copy the contents of the input buffer there, and let the verifier operate from there. To solve both issues, we introduce an abstract type for externally-provided input pointers and a unique operation to copy its contents. That function will call the TEE pointer checker before copying and the abstract type forces us to use this operation before reading from such a pointer.

Thread permission. With the internal state properly hidden as a global variable, and the input buffer properly copied to a temporary buffer, the last precondition left for `verify_log` is that the service have access permission on the thread on which they want to call the verifier. However, this precondition (`log_of_tid`) is needed only to state some property of ghost state of the verifier to relate the thread logs before and after the call.

Thus, we amend the incremental API described in 3.5 with a *ghost* boolean switch, true if we want to enable such incremental proof on `verify_log`. This boolean switch is *ghost*, so only specifications and proofs can branch on it, not the actual code. If set to true, then the caller needs to have one half of the fractional permission on the thread log, and the internal thread lock keeps the other half, as described in 3.4; but it set to false, no such caller permission is needed, and the internal thread lock keeps the full permission. Thus, this ghost switch propagates up to the definition of the `top_level_state` and `thread_state_and_lock` types.

Then, verified applications can still use the incremental API with this ghost switch set to true, whereas for the hardened API for untrusted clients, we set this ghost switch to false. Anyway, the value of this ghost switch only impacts the correctness statement of `verify_log`, so the final theorem on `max_certified_epoch` still holds in both cases.

3.7 Some Statistics

We briefly cover some statistics about our development, aiming to characterize the effort involved in various aspects of the proof (Table 1). Our total development consists of about 22,000 lines of F^{*} code, including comments (which we use maintain as the code evolves). Not included are libraries that specify and prove the correctness of FastVer, together with various utilities developed in that context, as well as new

Description	F [*] LOC	C LOC
Ghost state constructions	1,719	-
Core implementation	8,426	4,622
Lowest-level functional spec	1,863	-
Simulation proof	8,242	-
Parsers	2,092	3,053
HACL Blake2	-	950

Table 1. Breaking down the FastVer2 development in six categories, with lines of F^{*} and extracted C code

data structures and libraries developed for Steel, including various kinds of hash tables and arrays. The functional specification of our core implementation is about 1,863 lines and the main semantic proof, in about 8,200 lines, relates this functional specification to the specification of the FastVer design and produces our main correctness theorem. Overall, the proof to executable code ratio is comparable to other developments, e.g., Hawblitzel et al. [13] report a proof-to-code ratio of about 5:1. We are hopeful that with improvements to our tools and libraries, the proof of the core implementation could be much more compact.

Our executable code also includes about 3,000 lines of C code for parsers auto-generated by EverParse from a data format description and 950 lines of pre-existing C code distributed by HACL*. Both these pieces of code were verified using Low* [20], another F^{*} DSL that produces C code. A caveat: Low* specifications are not formally relatable to Steel specifications, so we wrote small admitted wrappers for our Steel code to call into these Low* verified components. We are in the process of migrating our use of EverParse libraries to a Steel-based parser generator instead, which should allow us to remove some of these admitted wrappers.

4 Monitoring Faster with FastVer2

We have integrated Faster with FastVer2 monitoring. Our integration closely resembles what Arasu et al. [4] did for FastVer, except for the formally verified monitor, so we elide a detailed description. For example, like FastVer, our integration does not require any code changes to Faster, but relies on customization hooks provided by Faster to perform asynchronous monitoring; likewise, FastVer2 has a 1:1 correspondence between verifier and Faster threads and the same OS thread multiplexes performing computations of both.

Our goal here is to quantify the overheads of FastVer2 monitoring compared to the unverified monitoring of FastVer. Our experimental setup is identical to what was used for FastVer; it uses the same YCSB [?] key-value benchmark and a machine with identical specification. Our evaluation skips authentication using counters discussed in Section 2.1, to better focus on core the key-value functionality monitoring.

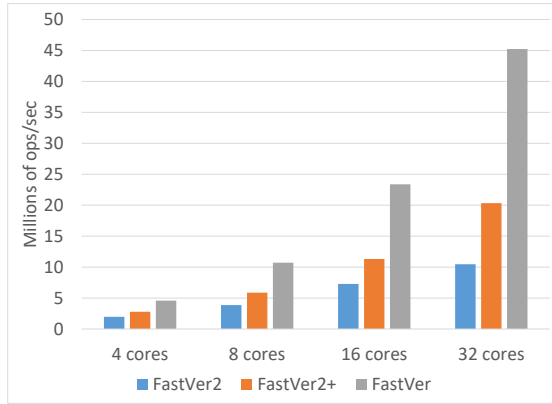


Figure 2. Throughput of FastVer2 vs. that of FastVer for a database of 16M records for the YCSB A benchmark. Also shown (as FastVer2+) is the throughput of FastVer2 with the overhead of Blake2 hashing during set-hash incremental computations removed.

Figure 2 presents a performance comparison between FastVer2 and FastVer, showing the overall maximum throughput (not constraining the latency) for the YCSB A benchmark (with an equal distribution of *get* and *put* operations) for a database of size 16M records with 8-byte keys and values. FastVer2 sees a performance drop of around a factor of 5 compared to FastVer. Despite this drop, FastVer2 still achieves a throughput exceeding 10M ops/sec, which is 2-3 orders of magnitude better any current formally verified key-value database [12]. Beyond the drop in throughput, the other performance characteristics of FastVer2 remain unchanged compared to FastVer. In particular, the throughput scales roughly linearly with the number of CPU cores.

We performed various micro-experiments to identify the main reasons for the lower throughput. The main contributor is the different implementations of the multi-set hash function: FastVer uses an AES-based construction leveraging the Intel *aesni* hardware instructions [8]; FastVer2 relies on a slower Blake2-based construction. On our setup, the AES-based implementation achieves around 200M incremental hash computations, while the Blake2-based one achieves around 4M computations, a 50x slowdown. When we replace the Blake2 based multi-set hash function with an efficient (but insecure) “dummy” hash function, the performance of FastVer2 increases by about 50%, reducing the throughput gap with FastVer to around 3x as shown in Figure 2.

Several other small inefficiencies cause the remaining performance gap. FastVer2 uses a byte-level xor’ing of 32 byte values during incremental hash computation, while FastVer uses word-level xor’ing (around 7% overhead). FastVer2 also

currently involves (avoidable) memory allocation and deallocation on the hot-path (around 5% overhead). Some of the performance overheads arise from fixing legitimate bugs that our formalization found, e.g., FastVer2 uses a lock, incurring an overhead, to prevent re-entrancy for a verified thread, while FastVer, incorrectly, does not prevent it.

None of the above mentioned reasons for slowdown are fundamental. We plan to switch to an AES-based hash construction by using components from EverCrypt’s [19] AES-GCM implementation in assembly, which also uses *aesni*. The other inefficiencies (temporary allocations, optimized xor’ing, etc.) can also be eliminated with some code and proofs rearrangements. Nevertheless, we are encouraged that without any specific attention to optimization, FastVer2’s performance scales linearly with the number of cores and is within an order of magnitude of FastVer.

5 Related Work

We cover three strands of related work: verified implementations of authenticated data structures; program logics for reasoning about monotonicity; and hardening verified code to run in an unsafe context.

Authenticated data structures. While there have been many schemes and libraries proposed for designing and implementing authenticated data structures, only a few have actually built formally verified implementations, e.g., EverCrypt [19] offers a formally verified incremental Merkle tree. However, as far as we are aware, FastVer2 is the first fully verified implementation of a concurrency-capable authenticated data structure, supporting both sparse, incremental Merkle trees and delayed memory verification.

Program logics for monotonicity. Ahman et al. [3] design a Hoare logic for reasoning about programs whose state is required to evolve monotonically according to some preorder. However, their approach is not based on separation logic and does not support concurrency. Building on Ahman et al.’s approach, Swamy et al. [24] develop SteelCore, a PCM-based separation logic which supports reasoning about monotonic state. In particular, Swamy et al. show how to derive a PCM from any preorder, and prove that frame-preserving updates for that PCM are also preorder respecting. Their construction is based on ghost state that stores histories, the type $\text{vhist } p$ that we use as a building block of the FRAP. Timany and Birkedal [27] also study reasoning about monotonicity in separation logic and present another construction to turn any preorder into a PCM. Rather than relying on histories, their construction is based on constructing a generic semi-lattice from a preorder. Earlier, and going back to Jensen and Birkedal’s [14] fictional separation logic, many others have developed PCM-based constructions for specific instances of monotonic state. As far as we are aware, no one else has proposed *anchored preorders*, which allows

naturally modeling scenarios where state evolves monotonically, but one thread’s knowledge of the state cannot run too far ahead of another’s.

Hardening APIs for use in an untrusted context. Our safe API wraps our incremental API with runtime checks to ensure that adversarial code calling it cannot break its safety. We achieve this by relying on specific hardware protections provided by TEE APIs to check untrusted pointers. Agten et al. [2] study the problem of executing verified code in an untrusted context in depth and develop a generic class of runtime checks that can be used to protect such verified APIs, while also relying on some notion of module-private memory at runtime (which we realize using the TEE). We were concerned with protecting the FastVer2 API specifically, rather than developing a generic system for protecting a class of APIs, nevertheless, the similarity in approaches is striking. Also related are typing disciplines for security protocols that offer robust safety or Un-typed APIs for safe use in an attacker’s context [1, 11].

References

- [1] Martin Abadi. 1997. Secrecy by Typing in Security Protocols. In *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings (Lecture Notes in Computer Science, Vol. 1281)*, Martin Abadi and Takayasu Ito (Eds.). Springer, 611–638. <https://doi.org/10.1007/BFb0014571>
- [2] Pieter Agten, Bart Jacobs, and Frank Piessens. 2015. Sound Modular Verification of C Code Executing in an Unverified Context. *SIGPLAN Not.* 50, 1 (jan 2015), 581–594. <https://doi.org/10.1145/2775051.2676972>
- [3] Danel Ahman, Cédric Fournet, Cătălin Hrițcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 2018. Recalling a Witness: Foundations and Applications of Monotonic State. *PACMPL* 2, POPL (jan 2018), 65:1–65:30. <https://arxiv.org/abs/1707.02466>
- [4] Arvind Arasu, Badrish Chandramouli, Johannes Gehrke, Esha Ghosh, Donald Kossmann, Jonathan Protzenko, Ravi Ramamurthy, Tahina Ramananandro, Aseem Rastogi, Srinath Setty, Nikhil Swamy, Alexander van Renen, and Min Xu. 2021. FastVer: Making Data Integrity a Commodity. In *Proceedings of the 2021 International Conference on Management of Data (Virtual Event, China) (SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 89–101. <https://doi.org/10.1145/3448016.3457312>
- [5] M. Blum, W. Evans, P. Gemmell, S. Kannan, and M. Naor. 1991. Checking the correctness of memories. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*. 90–99. <https://doi.org/10.1109/SFCS.1991.185352>
- [6] John Boyland. 2003. Checking Interference with Fractional Permissions. In *Static Analysis*, Radhia Cousot (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 55–72.
- [7] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: An Embedded Concurrent Key-Value Store for State Management. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1930–1933. <https://doi.org/10.14778/3229863.3236227>
- [8] Intel Corp. 2012. Intel Advanced Encryption Standard Instructions (AES-NI). <https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-encryption-standard-instructions-aes-ni.html>.
- [9] Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS (Lecture Notes in Computer Science, Vol. 4963)*. Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24
- [10] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 2021. Steel: Proof-Oriented Programming in a Dependently Typed Concurrent Separation Logic. In *Proceedings of the International Conference on Functional Programming (ICFP)*.
- [11] Andrew D. Gordon and Alan Jeffrey. 2003. Authenticity by Typing for Security Protocols. *J. Comput. Secur.* 11, 4 (2003), 451–520. <https://doi.org/10.3233/jcs-2003-11402>
- [12] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. 2020. Storage Systems Are Distributed Systems (so Verify Them That Way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 6, 17 pages.
- [13] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. 2014. Ironclad Apps: End-to-End Security via Automated Full-System Verification. In *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014*, Jason Flinn and Hank Levy (Eds.). USENIX Association, 165–181. <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/hawblitzel>
- [14] Jonas Braband Jensen and Lars Birkedal. 2012. Fictional Separation Logic. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7211)*, Helmut Seidl (Ed.). Springer, 377–396. https://doi.org/10.1007/978-3-642-28869-2_19
- [15] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.* 28 (2018), e20. <https://doi.org/10.1017/S0956796818000151>
- [16] Ralph Merkle. 1979. Method of providing digital signatures. US Patent US4309569A.
- [17] Aleksandar Nanevski, J. Gregory Morrisett, and Lars Birkedal. 2008. Hoare type theory, polymorphism and separation. *J. Funct. Program.* 18, 5-6 (2008), 865–911. <http://ynot.cs.harvard.edu/papers/jfpsep07.pdf>
- [18] Alexandre Pilkiewicz and François Pottier. 2011. The essence of monotonic state. In *Proceedings of TLDI 2011: 2011 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Austin, TX, USA, January 25, 2011*, Stephanie Weirich and Derek Dreyer (Eds.). ACM, 73–86. <https://doi.org/10.1145/1929553.1929565>
- [19] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. 2020. EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider. In *2020 IEEE Symposium on Security and Privacy (SP)*. 983–1002. <https://doi.org/10.1109/SP40000.2020.00114>
- [20] Jonathan Protzenko, Jean-Karim Zinzindhoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hrițcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. 2017. Verified Low-Level Programming Embedded in F*. *PACMPL* 1, ICFP (Sept. 2017), 17:1–17:29. <https://doi.org/10.1145/3110261>
- [21] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 2019. EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats. In *Proceedings of the 28th USENIX Conference on Security Symposium (Santa Clara, CA, USA) (USENIX Security 2019)*.

- 1431 USENIX Association, USA, 1465–1482.
- 1432 [22] Nikhil Swamy, Cătălin Hrițcu, Chantal Keller, Aseem Rastogi, An- 1487
1433 toine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric 1488
1434 Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzin- 1489
1435 dohoué, and Santiago Zanella-Béguelin. 2016. Dependent Types and 1490
1436 Multi-Monadic Effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposi- 1491*
1437 *um on Principles of Programming Languages (POPL)*. ACM, 256–270. 1492
1438 <https://www.fstar-lang.org/papers/mumom/> 1493
- 1439 [23] Nikhil Swamy, Tahina Ramananandro, Aseem Rastogi, Irina Spiri- 1494
1440 donova, Haobin Ni, Dmitry Malloy, Juan Vazquez, Michael Tang, 1495
1441 Omar Cardona, and Arti Gupta. 2022. Hardening Attack Surfaces 1496
1442 with Formally Proven Binary Format Parsers. In *Proceedings of the 1497*
1443 *43rd ACM SIGPLAN International Conference on Programming Lan- 1498*
1444 *guage Design and Implementation (San Diego, CA, USA) (PLDI 2022)*. 1499
1445 Association for Computing Machinery, New York, NY, USA, 31–45. 1500
1446 <https://doi.org/10.1145/3519939.3523708> 1501
- 1447 [24] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, 1502
1448 Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible 1503
1449 Concurrent Separation Logic for Effectful Dependently Typed Pro- 1504
1450 grams. *Proc. ACM Program. Lang.* 4, ICFP, Article 121 (Aug. 2020), 1505
1451 30 pages. <https://doi.org/10.1145/3409003> 1506
- 1452 [25] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, 1507
1453 Danel Ahman, and Guido Martínez. 2020. SteelCore: An Extensible 1508
1454 Concurrent Separation Logic for Effectful Dependently Typed Pro- 1509
1455 grams. *Proc. ACM Program. Lang.* 4, ICFP, Article 121 (Aug. 2020), 1510
1456 30 pages. <https://doi.org/10.1145/3409003> 1511
- 1457 [26] Roberto Tamassia. 2003. Authenticated Data Structures. In *Algorithms 1512*
1458 *- ESA 2003*, Giuseppe Di Battista and Uri Zwick (Eds.). Springer Berlin 1513
1459 Heidelberg, Berlin, Heidelberg, 2–5. 1514
- 1460 [27] Amin Timany and Lars Birkedal. 2021. Reasoning about Monotonicity 1515
1461 in Separation Logic. In *Proceedings of the 10th ACM SIGPLAN Interna- 1516*
1462 *tional Conference on Certified Programs and Proofs (Virtual, Denmark) 1517*
1463 *(CPP 2021)*. Association for Computing Machinery, New York, NY, 1518
1464 USA, 91–104. <https://doi.org/10.1145/3437992.3439931> 1519
- 1465 [28] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan 1520
1466 Protzenko, and Benjamin Beurdouche. 2017. HACl*: A Verified Mod- 1521
1467 ern Cryptographic Library. In *ACM Conference on Computer and Com- 1522*
1468 *munications Security*. ACM, 1789–1806. <http://eprint.iacr.org/2017/536> 1523

1464 A Background on F*

1465 F* is a programming language and proof assistant based on 1524
1466 a dependent type theory (like Coq, Agda, or Lean). F* also 1525
1467 offers an effect system, extensible with user-defined effects, 1526
1468 and makes use of SMT solving to automate some proofs. 1527

1469 F*'s toolchain includes support for several other embed- 1528
1470 ded DSLs, notably Steel [10, 24], for reasoning about con- 1529
1471 current, imperative programs in a concurrent separation 1530
1472 logic. Steel programs can be extracted from F* to C, and a 1531
1473 metatheorem establishes that extracted C programs simulate 1532
1474 the F* program. However, the tool implementing this extrac- 1533
1475 tion pipeline (named KaRaMeL), although modeled after this 1534
1476 metatheory, is not formally verified and is part of our trusted 1535
1477 computing base, which also includes the F* typechecker and 1536
1478 the Z3 SMT solver. 1537

1480 *Syntax: Binders, lambda, arrows, computation types.*

1481 F* syntax is roughly modeled on OCaml (`val`, `let`, `match` etc.) 1538
1482 with differences to account for the additional typing features. 1539
1483 Binding occurrences `b` of variables take the form `x:t`, declar- 1540
1484 ing a variable `x` at type `t`; or `#x:t` indicating that the binding 1541

1486 is for an implicit argument. The syntax $\lambda(b_1) \dots (b_n) \rightarrow t$ 1487
1488 introduces a lambda abstraction, whereas $b_1 \rightarrow \dots \rightarrow b_n \rightarrow c$ is 1489
1490 the shape of a curried function type. Refinement types are 1491
1492 written `b{t}`, e.g., `x:int{x ≥ 0}` is the type of non-negative integers 1493
1494 (i.e., `nat`). As usual, a bound variable is in scope to the 1495
1496 right of its binding; we omit the type in a binding when it 1496
1497 can be inferred; and for non-dependent function types, we 1497
1498 omit the variable name. The `c` to the right of an arrow is 1498
1499 a *computation type*. An example of a computation type is 1500
1501 `Tot bool`, the type of total computations returning a boolean. 1502
1503 The Steel DSL also has its own family of computation types, 1503
1504 similar to Hoare Type Theory [17], e.g., `ST t p q` is the type 1504
1505 of a concurrent computation returning a `v:t`, with (separa- 1505
1506 tion logic) precondition `p` and postcondition `q v`. By default, 1506
1507 function arrows have `Tot` co-domains, so, rather than deco- 1507
1508 rating the right-hand side of every arrow with a `Tot`, the type 1508
1509 of, say, the pure append function on vectors can be written 1509
1510 `#a:Type → #m:nat → #n:nat → vec a m → vec a n → vec a (m + n)`, 1510
1511 with the two explicit arguments and the return type depend- 1511
1512 ing on the three implicit arguments marked with `#`. We often 1512
1513 omit implicit binders and treat all unbound names as implic- 1513
1514 itly bound at the top, e.g., `vec a m → vec a n → vec a (m + n)` 1514
1515

1515 B A FRAP for Counters

1516 Backing up for the moment from the specifics for FastVer2, 1517
1518 consider the following simpler scenario. Say we have two 1518
1519 threads sharing a mutable increment-only counter `i`. Thread 1519
1520 `P` (the producer) atomically increments the counter, while 1520
1521 thread `C` (the consumer) reads it. If `P` owns the assertion 1521
1522 $i \mapsto n$, it should be able to update the state to $i \mapsto m$, only if 1522
1523 $m \geq n$. Meanwhile, if `C` owns the read-only assertion $i \mapsto' n$, 1523
1524 it should be able to conclude that reading `i` returns a value 1524
1525 $m \geq n$ —this should be a fairly familiar scenario, explored 1525
1526 by several papers related to separation logic and monotonic 1526
1527 state, e.g., by Pilkiewicz and Pottier [18]. 1527

1528 The twist here is to find a way to ensure that `C`'s knowl- 1528
1529 edge of the value of `i` is never too far behind `P`'s knowledge 1529
1530 of its current value. For example, when distributing permis- 1530
1531 sions to the counter to `P` and `C`, we may decide that `P` must 1531
1532 synchronize with `C` every time it increments the counter to 1532
1533 the next even number. In particular, we say that the assertion 1533
1534 $i \overset{\square}{\mapsto} n$ anchors the counter at `n`, for some even number `n`, 1534
1535 ensuring that `P` cannot advance the counter beyond `n + 1`; 1535
1536 and when `C` reads the counter obtaining `m` it can conclude 1536
1537 that $m \leq n + 1$. 1537

1538 To model this scenario in Steel using a FRAP, one simply 1538
1539 defines the preorder and anchors relation as shown below: 1539

```
1540 let p : preorder nat = λx y → x ≤ y
1541 let at_most_one_away : anchor_rel p =
1542   λx y → x%2=0 ∧ x ≤ y ∧ y ≤ x + 1
```

1543 Then, `frap at_most_one_away` is an instance of a FRAP. 1543

1544 Now, to work with ghost references that store values of 1544
1545 `frap at_most_one_away`, one can define some derived notions, 1545

1541 where $r \xrightarrow{f} v$ represents fractional non-anchored knowledge
 1542 of the contents of r ; $r \overset{\square}{\mapsto} v$ anchors r to v ; $r \overset{\square f}{\mapsto} v$ anchors r to v
 1543 while also holding fraction f ; and finally, $r \overset{\bullet}{\mapsto} v$ is a snapshot
 1544 of the contents of r .
 1545 `let vhist p v = h:vhist p { cur h == v }`
 1546 `r \xrightarrow{f} v = \exists (h:vhist p v). G.pts_to r (Owns ((Some f, None), h))`
 1547 `r $\overset{\square}{\mapsto}$ v = \exists (h:vhist p v {v%2=0}). G.pts_to r (Owns ((None, Some v), h))`
 1548 `r $\overset{\square f}{\mapsto}$ v = \exists (h:vhist p v {v%2=0}). G.pts_to r (Owns ((Some f, Some v), h))`
 1549 `r $\overset{\bullet}{\mapsto}$ v = \exists (h:vhist p v). G.pts_to r (Owns ((None, None), v))`

The lemmas we've proven about the FRAP PCM show that the following Hoare triples are provable in Steel, starting with `split_anchor` which allows splitting our knowledge of the anchor into separate permissions to hand to P and C .

1557 `val split_anchor (r:G.ref (frap at_most_one_away)) : STG unit`
 1558 `(requires $r \overset{\square f}{\mapsto} v$) (ensures $\lambda_ \rightarrow r \xrightarrow{f} v * r \overset{\square}{\mapsto} v$)`

Next, `snap` allows taking snapshots and proving that snapshots are always valid histories.

1562 `val snap (r:G.ref (frap at_most_one_away)) : STG unit`
 1563 `(requires $r \xrightarrow{f} v$) (ensures $\lambda_ \rightarrow r \xrightarrow{f} v * r \overset{\bullet}{\mapsto} v$)`
 1564 `val snap_hist (r:G.ref (frap at_most_one_away)) : STG (_:unit { v' ≤ v })`
 1565 `(requires $r \xrightarrow{f} v * r \overset{\bullet}{\mapsto} v'$) (ensures $\lambda_ \rightarrow r \xrightarrow{f} v * r \overset{\bullet}{\mapsto} v'$)`

Since only even values can be anchors, P can increment the counter to an odd value can be done without synchronizing with C —this corresponds in FastVer2 to a thread advancing its state without traversing an epoch boundary and without synchronizing with the shared state.

1573 `val increment_odd (r:G.ref (frap at_most_one_away)) : STG unit`
 1574 `(requires $r \xrightarrow{f} v * \text{pure } (v\%2=0)$)`
 1575 `(ensures $r \xrightarrow{f} (v + 1)$)`

But, to increment the counter to the next even value requires gathering both knowledge of the current value of r as well as knowledge of its anchor, incrementing r and advancing the anchor to the next even value, which corresponds in FastVer2 to a verifier thread completing an epoch and synchronizing with the shared state.

1584 `val increment_even (r:G.ref (frap at_most_one_away)) : STG unit`
 1585 `(requires $r \xrightarrow{f} (v + 1) * r \overset{\square}{\mapsto} v$)`
 1586 `(ensures $r \xrightarrow{f} (v + 2) * r \overset{\square}{\mapsto} (v + 2)$)`

1589 C Hardening the API

1591 `noeq type state_t = {`
 1592 `state_ref: ref top_level_state;`
 1593 `ghost_state: Ghost.erased top_level_state;`
 1594 `_inv: inv ($\exists p$. pts_to state_ref p ghost_state * core_inv ghost_state));`

1596 }
 1597
 1598 `let state = begin`
 1599 `let state_ref = init () in`
 1600 `assert_ (\exists state . pts_to state_ref 1.0state * core_inv state);`
 1601 `let ghost_state = elim_ \exists () in`
 1602 `let _inv = new_invariant`
 1603 `($\exists p$. pts_to state_ref p ghost_state * core_inv ghost_state)) in`
 1604 `return`
 1605 `({ state_ref = state_ref; ghost_state = ghost_state; _inv = _inv; })`
 1606 `end <: STT state_t emp ($\lambda_ \rightarrow$ emp)`

In practice, the initializer needs to be called upon start. Instead, we could also allow the user to initialize the top-level state later as they wished. To this end, we would allocate a `ref` (`ref top_level_state`) initially set to `NULL`. The invariant as designed makes the `ref` read-only, and even if we constrained the permission held in the invariant to be equal to 1 if the state is uninitialized, this would not be enough, since we need to check that the reference is null before updating it with a fresh top-level state, both of which cannot be done together atomically. So, we need to create a lock to protect that reference. A lock alone could be enough for safety, but it would actually prevent all concurrency. Thus, we need to keep the invariant, and say that the lock and the invariant each hold some (potentially different) permission on the reference, and that the sum of the two permissions equals 1 as long as the reference points to `NULL`. Then, when the user explicitly initializes the top-level state, we acquire the lock, allocate a top-level state, then we open the invariant, and we atomically set the `ref` to the fresh top-level state, and finally we can close the invariant and release the lock. Then, once the top-level state is initialized, the reference becomes read-only and the invariant is enough as before, the lock need no longer be acquired.