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 that 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.
A substrategy that we design and implement is a scheme for multiple verifier ware, a skeptical client can be sure that the execution of an untrusted service is correctly monitored by high-assurance, formally proven code. This paper presents FastVer2, a provably correct, low-level, untrusted service is correctly monitored by high-assurance, formally proven code. 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 scalability, the service is implemented using multiple concurrent sub-protocols. We also summarize our goals and ways to bridge the gap.

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 HACL* [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 \( C \), 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 \( L_i = \overline{op} \), tagged with the client’s identifier \( i \) and a client-specific sequence number \( n \), where the sequence number of an item \( op \) in \( L_i \) 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 \( C \) is represented by all their logs \( \overline{L} \).

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

Definition 2.1. Formally, we say that a log of client operations \( I \) is sequentially consistent if and only if:

- For every \( op(i, n) \in I \), if it is preceded by an operation \( op(i, m) \in I \), then \( m < n \), i.e., the sequence numbers for each client \( i \) are monotonically increasing; and,
For every get k v in I at position n, there exists a preceding operation put k v in I at position m < n and none of the operations in 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 Ci’s execution trace, when interleaved with the execution traces of an arbitrary context of other clients Ĉ, is sequentially consistent. We need to assume that the untrusted server S itself is not among the client context Ĉ, since otherwise S can always inject messages to justify otherwise faulty behavior observed by Ci. For example, Ci’s trace put k 5, get k 6 could be justified by an S itself injecting a spurious put k 6 between Ci’s two messages. One way to enforce that S ∈ Ĉ 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 monotor 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 \( \mathcal{V} \). 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 \( \ell L_i \) 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 a non-negative integer epoch and a non-negative integer tick counter, 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 v1 and v2 such that v1 ≠ v2 but hash v1 = hash v2.

let seq_consistent (logs: seq (seq log_entry)) = ...

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
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 V₁ for all epochs up to epoch 2, and epoch 1 for Vₙ. If all threads have committed their add and evict hashes for an epoch in 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 Vᵢ processes an operation, it updates its log to record this fact. As such, Vᵢ 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 Vᵢ needs to update its log, even though it does not exclusively own it. However, in this case Vᵢ 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 Vᵢ’s log and is stable with respect to log append operations, Vᵢ 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 Vᵢ 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 Vᵢ 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 Vᵢ’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 a 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 ppcm a, Steel allows allocating a ghost reference
r.G.ref p, a reference to mutable ghost state holding a value
of type va, and to state assertions of the form 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 λr → G.pts_to r v)

The signature shown above is a specification in Steel, where
the type 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 vprop, and so, p:vprop
and q:→vprop. To emphasize the role of p and q as pre- and
post-conditions, 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 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
G.pts_to r (v₀ `op` v₁) is equivalent to G.pts_to r v₀* G.pts_to r v₁,
as the following two lemmas show.

val share (r:G.ref p): STG unit
(requires G.pts_ro r (v₀ `op` v₁))
(ensures λr → G.pts_to r v₀ + G.pts_to r v₁)

val gather (r:G.ref p): STG unit
(requires G.pts_ro r v₀ + G.pts_to r v₁)
(ensures λr → G.pts_to r (v₀ `op` v₁))

That is, knowledge that r holds a composite value v₀ `op` v₁
can be traded back and forth with separate knowledge about
each component. To enable this, SteelCore requires that ev-
every update to a reference be frame-preserving, i.e., know-
G.pts_to r k, an update should preserve all assertions
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 anchor_rel p is
more interesting, and restricts the preorder as shown below.
It may provide useful intuitions to think of 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 v₁ is not too far ahead of v₀, then v₁ is also
related to v₀ 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, 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 → v → prop)
(∀ v0 v1. v0 `anchors` v₁ ⇒ p v0 v1) ∧
(∀ x z. x `anchors` z ⇒ (∀ y. p x y ∧ p y z ⇒ x `anchors` y))

Next, the carrier type knowledge a of the PCM is shown below,
where Nothing will be the unit of our PCM, and Owns av
represents some non-trivial knowledge.

type knowledge (a:anchor_rel p) =
| Owns : avalue a → knowledge a
| Nothing : knowledge a

An av: avalue a (defined below) is a pair (perm, h) of a per-
mission perm and a h:st p, a type which encodes the entire
preorder/comparable history of values that will be stored at
a ghost reference for this PCM. The h:st p construction ex-
isted 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
cur v is the most recent value in the history. What is more
interesting is the structure of perm = (op, a), itself a pair of an
optional fractional permission, where None represents a
0 permission (useful for snapshots, as we will see). Further,
when oa = Some a, we say that the av has an anchor a. The
anchored 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 & st p)) =
match pv with
| (_ Some a), v → a `arel` cur_v | _ → T
let avalue a = av:(permission v & st p) { anchored a av }

Composability. The composability of v₀, v₁: 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 ((p₀, h₀) (p₁, h₁)): avalue arel =
permission_composable p₀ p₁ ∧ (h₀ ≥ h₁ ∨ h₁ ≥ h₀) ∧
(match p₀, p₁ with
| (None, None), (None, None) → T
| (None, None), (Some _ _) → h₁
| (None, None), (_, Some a) → h₀ ≥ h₁ ⇒ a `arel` cur_h₀
| (Some _, _), (Some _, _) → h₀ == h₁
| (Some _, _), (_, Some a) → h₀ ≥ h₁ ∧ a `arel` cur_h₀
| ... remaining cases are symmetric

Suppose v₀ = (p₀, h₀) and v₁ = (p₁, h₁). To start with, p₀ is com-
posable with p₁ if the sum of their fractions is no more than
1 and, importantly, if at most one of them has an anchor.
Further, h₀ and h₁ must at least be composable in terms of the
regular definition of composability of h:st p, the history-

Based PCM for preorders, i.e., one of them must be an exten-
sion of the other, effectively forbidding "forks" in the history.
The rest of the definition is by case analysis on the permis-
sions. If neither p₀ or p₁ holds any permission then no further
constraints apply. Next, if p₀=none, then we have two
sub-cases: if p₁ holds some non-zero fraction, then h₁ must
be more recent than h₀; otherwise, if p₁ 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 $h_0 \equiv h_1$; 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 $v_0, v_1$; $a$ value $a$, we compose their permissions (summing their fractional permissions and retaining whichever anchor is present) and their histories (tacking 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 $perm\_ok\ a$ (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 $a \equiv a$ ‘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.

```
let value\_of\ av = cur (snd av)
let snapshot (av: avalue s) : avalue s = (None, None, snd av)
let snapshot\_props (a: avalue s { perm\_ok\ a }) : Lemma (a ‘composable’ snapshot a ∧
  a ‘compose’ snapshot a == a ∧
  snapshot a ‘composable’ snapshot a ∧
  snapshot a ‘compose’ snapshot a == snapshot a ∧
  (∀ b: avalue s { has_perm b ∧ b ‘composable’ snapshot a}).
value\_of\ (snapshot a) ≤ value\_of\ b) = ()
```

**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.

```
let split\_anchor (((p, a), v): avalue s) = ((None, None), ((None, a), v)
let split\_anchor\_props (av: avalue s { perm\_ok\ av })
  : Lemma (let kv, ka = split\_anchor\ av in
    kv ‘composable’ ka ∧ kv ‘compose’ ka == av = ()
let elim\_anchor (a: avalue s { has_perm a })
  (b: avalue s { has\_anchor b ∧ composable a b }))
  : Lemma (let (_, Some anc) = b in s anc (value\_of\ a)) = ()
```

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.

```
let log\_grows : preorder log = λl0 l1 → l0 ≤ l1
let is\_last\_committed l0 l1 = l0 ≤ l1 ∧ l0 == \downarrow l1

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.

let log\_frap = frap is_last_committed
let tlm = PCM\_Map\_pointwise\_tid log\_frap
let tlm\_carrier = map tid (knowledge (avalue is_last\_committed))
```

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 $m$ : 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 $fr \frac{m}$ 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
we can derive the following (ghost) operations—these are a variant of taken
converse lemma is also provable, though we do not show it.
threads—a fact that follows from structure of PCM maps. The
take anchor shows that one can combine knowledge of the current state of a thread’s log
owning the anchor from the global anchor and conclude the anchor a = Some? v (sel m t) is the committed prefix of l.
val take_anchor x m (t: [Some? (sel m t)]) f l : STG unit
(ensures global_anchor x m = tid_pts_to x t f l false)
(ensures \lambda a \rightarrow (global_anchor x (upd m t None) * tid_pts_to x t f l true) + pure (\exists l: Some? v (sel m t))
Conversely, a thread can cede ownership of an anchor back to the global anchor, provided its current state is fully committed.
val put_anchor x m t f (l: [\exists l=\ll]) : STG unit
(ensures tid_pts_to x t f l false * global_anchor x m)
(ensures \lambda a \rightarrow tid_pts_to x t f l false * global_anchor x (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 it thread-local state, without needing to synchronize with the shared state.
val update_log x t l 0 (l: [\forall l \leq l_1 \land \forall l_0=\ll_1]) : STG unit
(ensures tid_pts_to x t 0 l_0 false)
(ensures \lambda a \rightarrow tid_pts_to x t 0 l_1 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 Vt must first take knowledge of its anchor from the shared state using take_anchor, then transition using update_anchored_log, and finally return knowledge of the updated anchor to the shared state using put_anchor.
val update_anchored_log x t l 0 (l: [\forall l \leq l_1 \land \forall l_0=\ll_1]) : STG unit
(ensures tid_pts_to x t 0 l_0 true)
(ensures \lambda a \rightarrow tid_pts_to x t 0 l_1 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.

```haskell
type top_level_state = { aeh:aggregate_epoch_hashes;
all_threads:array (thread_state_and_lock aeh.mlogs) n_threads }```

The first field, 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 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.

```haskell```
```c
```
The top-level invariant, \( \text{core\_inv} \), merely asserts some permission over the \( \text{t\_all\_threads} \) array and states that the thread with id \( \text{id} \) is at position \( \text{i} \) in the array (\( \text{ind\_ok} \)); the most interesting parts of the invariant are held in locks stored in other fields of the top-level state:

\[
\text{let core\_inv t} = \exists p. \text{v. A.pts\_to t\_all\_threads p v} \lor \text{pure (ind\_ok v)}
\]

The thread-local state is represented by the structure shown below, pairing the actual thread state \( ts \) with a lock, an instance of a \( \text{cancellable\_lock} \), 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.

\[
\text{type thread\_state\_and\_lock mlogs} = \\
\{ \\
i: \text{tid}; ts:\text{thread\_state} t\{\text{thread\_id} ts \leftrightarrow i\}; \\
\text{lock: Lock.cancellable\_lock (3 tsm. thread\_inv ts mlogs tsm)} \}
\]

The invariant held by the lock, \( \text{thread\_inv} \), states that the concrete state of the thread reachable from \( ts \) is a refinement of \( ts: \text{thread\_state\_model} \), 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 \( tsm. \text{processed\_entries} \). As we will see, the other half permission is passed to the context and used to specify the top-level incremental API.

\[
\text{let thread\_inv ts mlogs tsm} = \\
\text{state\_refinement ts tsm} \lor \text{pure (tsm.failed)} \lor \\
\text{tid\_pts\_to mlogs tsm. thread\_id half tsm. processed\_entries false}
\]

The \( \text{aeh\_aggregate\_epoch\_hashes} \) 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.

\[
\text{type aggregate\_epoch\_hashes} = \\
\{ \\
\text{hashes : epoch\_tid\_hashes; bitmaps : epoch\_tid\_bitmaps; } \\
\text{max : R.ref (option epoch\_id); mlogs : mlogs; lock: Lock.cancellable\_lock (agg\_inv hashes bitmaps map mlogs)} \}
\]

The invariant \( \text{agg\_inv} \) 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 \( \text{max\_v} \); and, importantly, that all these values are related to the functional correctness specification by \( \text{hashes\_bitmaps\_max\_ok} \) 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, \( \text{global\_anchor} \).

\[
\text{let agg\_inv hashes bitmaps max mlogs} = \exists hv bv \text{max\_v mlogs\_v. } \\
\text{EpochMap.pts\_to hv} \lor \\
\text{EpochMap.pts\_to full bitmaps bv} \lor \\
\text{pts\_to max full max\_v} \lor \\
\text{global\_anchor mlogs (map of seq mlogs\_v)} \lor \\
\text{pure (hashes\_bitmaps\_max\_ok hv bv max\_v mlogs\_v)}
\]

To update the aggregate state at the end of an epoch, a verifier thread must acquire \( \text{aeh\_lock} \), take ownership of its anchor, then update both the concrete and ghost state (the latter using \( \text{update\_anchored\_log} \), 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 \( \text{verified context} \) 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 \( \text{init}() \), which initializes \( n\_threads \) (a compile-time constant) verifier threads.

\[
\text{let ilogs t = tid\_pts\_to t.aeh.mlogs half (const (Some empty)) false } \\
\text{val init (_, unit) : ST (ref top\_level\_state)}
\]

\[
\text{(requires emp)}
\]

\[
\text{(ensures \lambda r \rightarrow \exists t. pts\_to r full t \leftrightarrow core\_inv t \lor ilogs t)}
\]

Since its precondition is just \( \text{emp} \), \( \text{init}() \) can be called at any time, including multiple times. It returns a fresh handle to an instance of the monitor, \( \text{rref top\_level\_state} \), a concrete reference to \( \text{top\_level\_state} \), 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 \( \text{core\_inv} \), encapsulating the main invariant of the monitor. The \( \text{core\_inv} \) is duplicable, meaning that from \( \text{core\_inv} \) it is possible to derive \( \text{core\_inv} ts \leftrightarrow \text{core\_inv} ts \).
3. And, finally, \( \text{ilogs} \) 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 \( \text{verify\_log} \) is shown below. It is invoked to request a given thread \( i \) to verify some input log of entries.
let log_of_tid tid tid e = tid_pts_to tid @ mlogs i half e
val verify_log (r: ref top_level_state) (tid) =
  (input:array U8.1 len (len #0ul))
: ST (option verify_result)
  (requires pts_to r tid为核心 inv t * A pts_to input lpb b *
  log_of_tid tid i ents)
  (ensures res is ->
    pts_to r tid * core inv t * A pts_to input lpb b *
    (match res with
      | Some (Verify success read wrote) -> ∃ents
      log_of tid tid i ents @ ents ')
    pure (let s = verify_model (inital_model tid) ents in
      let s' = verify_model s ents ' in
      read == len ∧
        parse_log_up_to_lpb (U32 v read) == Some ents')
    | _ -> Some log_of_tid tid 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 t. 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 si-
multaneously 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 aggre-
gate 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 tid)
  (ensures res is ->
    match res with
      | Read_max_some max ->
        Some 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 un-
trusted 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 permis-
sion to access it later, but by default, if we carelessly perform
that allocation as an F* top-level variable, we lose that per-
mission 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 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.

<table>
<thead>
<tr>
<th>Description</th>
<th>F* LOC</th>
<th>C LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ghost state constructions</td>
<td>1,719</td>
<td>-</td>
</tr>
<tr>
<td>Core implementation</td>
<td>8,426</td>
<td>4,622</td>
</tr>
<tr>
<td>Lowest-level functional spec</td>
<td>1,863</td>
<td>-</td>
</tr>
<tr>
<td>Simulation proof</td>
<td>8,242</td>
<td>-</td>
</tr>
<tr>
<td>Parsers</td>
<td>2,092</td>
<td>3,053</td>
</tr>
<tr>
<td>HACL Blake2</td>
<td></td>
<td>950</td>
</tr>
</tbody>
</table>

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

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 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 than 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 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


is for an implicit argument. The syntax $\lambda(b_1) \ldots (b_n) \to t$ introduces a lambda abstraction, whereas $b_1 \to \ldots \to b_n \to c$ is the shape of a curried function type. Refinement types are written $b(t)$, e.g., $\text{int}([\leq 0])$ is the type of non-negative integers (i.e., non-negative natural numbers). As usual, a bound variable is in scope to the right of its binding; we omit the type in a binding when it can be inferred; and for non-dependent function types, we omit the variable name. The $c$ to the right of an arrow is a computation type. An example of a computation type is a $\text{Tot} \to \text{bool}$ function, the type of total computations returning a boolean.

The Steel DSL also has its own family of computation types, similar to Hoare Type Theory [17], e.g., $\text{ST} t p q$ is the type of a concurrent computation returning a value $v$, with (separation logic) precondition $p$ and postcondition $q \lor v$. By default, function arrows have a $\text{Tot}$ co-domain, so, rather than decorating the right-hand-side of every arrow with a $\text{Tot}$, the type of, say, the pure append function on vectors can be written as:

$$\lambda a. \text{Type} \to \#\text{nat} \to \#\text{nat} \to \text{vec} a m \to \text{vec} a n \to \text{vec} (a + n)$$

with the two explicit arguments and the return type depending on the three implicit arguments marked with ‘$\#$‘. We often omit implicit binders and treat all unbound names as implicitly bound at the top, e.g., $\text{vec} a m \to \text{vec} a n \to \text{vec} (a + n)$.

B A FRAP for Counters

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

The twist here is to find a way to ensure that $C$’s knowledge of the value of $i$ is never too far behind $P$’s knowledge of its current value. For example, when distributing permissions to the counter to $P$ and $C$, we may decide that $P$ must synchronize with $C$ every time it increments the counter to the next even number. In particular, we say that the assertion $i \leftrightarrow n$ anchors the counter at $n$, for some even number $n$, ensuring that $P$ cannot advance the counter beyond $n + 1$; and when $C$ reads the counter obtaining $m$ it can conclude that $m \leq n + 1$.

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

\[
\text{let } p : \text{preorder } \text{nat} = \lambda x y : x \leq y \\
\text{let } \text{at_most_one_away} : \text{anchor } \text{rel} p = \lambda x y : x \% 2 = 0 \land x \leq y \land y \leq x + 1
\]

Then, \text{frap at_most_one_away} is an instance of a FRAP.

Now, to work with ghost references that store values of \text{frap at_most_one_away}, one can define some derived notions.
where \( r \mapsto v \) represents fractional non-anchored knowledge of the contents of \( r \); \( r \leftrightharpoons v \) anchors \( r \) to \( v \); \( r \mapsto v \) anchors \( r \) to \( v \) while also holding fraction \( f \); and finally, \( r \mapsto v \) is a snapshot of the contents of \( r \).

let hist p v = h:hist p { cur h == v }  
\( r \mapsto v = \exists (h:hist p v) \). G.pts_to r (Owns ((Some f, None), h))  
\( r \leftrightharpoons v = \exists (h:hist p v[v\%2=0]). G.pts_to r (Owns ((None, Some v), h)) \)  
\( r \mapsto v = \exists (h:hist p v[v\%2=0]). G.pts_to r (Owns ((Some f, Some v), h)) \)  
\( r \mapsto \bullet v = \exists (h:hist p v). G.pts_to r (Owns ((None, None), v)) \)

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 \( C \).

val split_anchor (r:G.ref (frap at most one away)) : STG unit  
(requires \( r \mapsto v \)) (ensures \( \lambda \_ \rightarrow r \mapsto v \) \( \leftrightarrow r \mapsto v \))

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

val snap (r:G.ref (frap at most one away)) : STG unit  
(requires \( r \mapsto v \)) (ensures \( \lambda \_ \rightarrow r \mapsto v \) \( \leftrightarrow r \mapsto v \))

val snap_hist (r:G.ref (frap at most one away)) : STG (_unit \{ \ v' \le v \})  
(requires \( r \mapsto v \bullet r \mapsto v \)) (ensures \( \lambda \_ \rightarrow r \mapsto v \bullet r \mapsto v \))

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.

val increment_odd (r:G.ref (frap at most one away)) : STG unit  
(requires \( r \mapsto v \)) (ensures \( \lambda \_ \rightarrow r \mapsto pure (v/2=0) \))

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.

val increment_even (r:G.ref (frap at most one away)) : STG unit  
(requires \( r \mapsto v \)) (ensures \( \lambda \_ \rightarrow r \mapsto (v+1) \))

\( \begin{align*} & \text{let state } = \text{ begin} \\
& \text{let state_ref } = \text{ init } () \quad \text{in} \\
& \text{assert } (_3 \text{ state } . \text{ pts_to state_ref } 1.0 \text{ state } \cdot \text{ core_inv state } ) \quad \text{in} \\
& \text{let ghost_state } = \text{ elim}_3 () \quad \text{in} \\
& \text{let inv } = \text{ new_invariant} \\
& \quad (\exists p \text{ pts_to state_ref } p \text{ ghost_state } \cdot \text{ core_inv ghost_state } ) \quad \text{in} \\
& \text{return} \\
& \quad (\{ \text{ state_ref } = \text{ state_ref } ; \text{ ghost_state } = \text{ ghost_state } ; \_ \text{ inv } = \_ \text{ inv } ; \} ) \quad \text{end} <:\text{ STT state_t emp } (\lambda \_ \rightarrow \text{ emp}) \end{align*} \)

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.

C Hardening the API

noeq type state_t = {  
state_ref: ref top_level_state;  
ghost_state: Ghost.ghost state;  
_inv: inv (\exists p \text{ pts_to state_ref } p \text{ ghost_state } \cdot \text{ core_inv ghost_state })
}