Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3

James Bornholt
Amazon Web Services
& The University of Texas at Austin

Rajeev Joshi
Amazon Web Services

Vytautas Astrauskas
ETH Zurich

Brendan Cully
Amazon Web Services

Bernhard Kragl
Amazon Web Services

Seth Markle
Amazon Web Services

Kyle Sauri
Amazon Web Services

Drew Schleit
Amazon Web Services

Grant Slatton
Amazon Web Services

Serdar Tasiran
Amazon Web Services

Jacob Van Geffen
University of Washington

Andrew Warfield
Amazon Web Services

Abstract

This paper reports our experience applying lightweight formal methods to validate the correctness of ShardStore, a new key-value storage node implementation for the Amazon S3 cloud object storage service. By "lightweight formal methods" we mean a pragmatic approach to verifying the correctness of a production storage node that is under ongoing feature development by a full-time engineering team. We do not aim to achieve full formal verification, but instead emphasize automation, usability, and the ability to continually ensure correctness as both software and its specification evolve over time. Our approach decomposes correctness into independent properties, each checked by the most appropriate tool, and develops executable reference models as specifications to be checked against the implementation. Our work has prevented 16 issues from reaching production, including subtle crash consistency and concurrency problems, and has been extended by non-formal-methods experts to check new features and properties as ShardStore has evolved.

CCS Concepts: • **Software and its engineering → Software verification and validation**.

Keywords: cloud storage, lightweight formal methods

ACM Reference Format:
James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. 2021. Using Lightweight Formal Methods to Validate a Key-Value Storage Node in Amazon S3. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–28, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3477132.3483540

SOSP '21, October 26–28, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483540

1 Introduction

Amazon S3 is a cloud object storage service that offers customers elastic storage with extremely high durability and availability. At the core of S3 are *storage node* servers that persist object data on hard disks. These storage nodes are key-value stores that hold *shards* of object data, replicated by the control plane across multiple nodes for durability. S3 is building a new key-value storage node called *ShardStore* that is being gradually deployed within our current service.

Production storage systems such as ShardStore are notoriously difficult to get right [25]. To achieve high performance, ShardStore combines a soft-updates crash consistency protocol [16], extensive concurrency, append-only IO and garbage collection to support diverse storage media, and other complicating factors. Its current implementation reflects that complexity, comprising over 40,000 lines of Rust code. The implementation is fast moving and frequently changing, and even the specification is not set in stone. It is developed and operated by a team of engineers whose changes are continuously deployed worldwide, and currently stores hundreds of petabytes of customer data as part of a gradual rollout.

This paper reports our experiences applying *lightweight formal methods* [24] such as property-based testing and stateless model checking to validate ShardStore's correctness. We sought *lightweight* methods that were automated and usable by non-formal-methods experts to validate new features on their own; our goal was to mainstream formal methods into our daily engineering practice. We sought *formal* methods because we wanted to validate deep properties of the implementation—functional correctness of API-level calls, crash consistency of on-disk data structures, and concurrent

correctness of API calls and background maintenance tasks—not just general properties like memory safety. In return for being lightweight and easy to apply, we were willing to accept weaker correctness guarantees than full formal verification, which is currently impractical for storage systems at this scale and level of complexity. We were inspired by recent efforts on verified storage systems [8, 18, 49] and hope that our experience provides encouragement for future work.

We settled on an approach with three elements. First, we distill specifications of desired behavior using executable *reference models* that define the expected semantics of the system. A reference model defines the allowed sequential, crash-free behaviors of a component in the system. Reference models are written in the same language as the implementation and embedded in its code base, allowing them to be written and maintained by the engineering team rather than languishing as separate expert-written artifacts. The reference models developed for ShardStore are small executable specifications (1% of the implementation code) that emphasize simplicity; for example, the reference model for a log-structured merge tree [41] implementation is a hash map. We also reuse these reference models as mock implementations for ShardStore unit tests, effectively requiring engineers to update the reference model specifications themselves when developing new code.

Second, we validate that the ShardStore implementation satisfies our desired correctness properties by checking that it refines the reference model. To make the best use of available lightweight formal methods tools, we decompose these properties and apply the most appropriate tool for each component. For functional correctness, we apply property-based testing [9] to check that the implementation and model agree on random sequences of API operations. For crash consistency, we augment the reference models to define the specific recent mutations that soft updates allow to be lost during a crash, and again apply property-based testing to check conformance on histories that include arbitrary crashes. For concurrency, we apply stateless model checking [5, 35] to show that the implementation is linearizable [19] with respect to the reference model. In all three cases the checkers are highly automated and the infrastructure is compact (the combined property definitions and test harnesses comprise 12% of the ShardStore code base). These checks have prevented 16 issues from reaching production, including subtle crash-consistency and concurrency issues that evaded traditional testing methods, and anecdotally have prevented more issues from even reaching code review. These checks are also "pay-as-you-go", in the sense that we can run them for longer to increase the chance of finding issues (like fuzzing), and so they can be run both locally on an engineer's machine during development and at scale before deployments.

Third, we ensure that the results of this validation effort remain relevant as the system evolves by training the engineering team to develop their own reference models and checks. Formal methods experts wrote the initial validation infrastructure for ShardStore, but today 18% of the total reference model and test harness code has been written by the engineering team to check new features and properties, and we expect this percentage to increase as we continue to adopt formal methods across S3. We apply code coverage metrics to monitor the quality of checks over time and ensure that new functionality remains covered.

In summary, this paper makes three main contributions:

- A lightweight approach to specifying and validating a production-scale storage system in the face of frequent changes and new features.
- A decomposition of storage system correctness that allows applying a diverse suite of formal methods tools to best check each property.
- Our experience integrating lightweight formal methods into the practice of a production engineering team and handing over the validation artifacts to be maintained by them rather than formal methods experts.

2 ShardStore

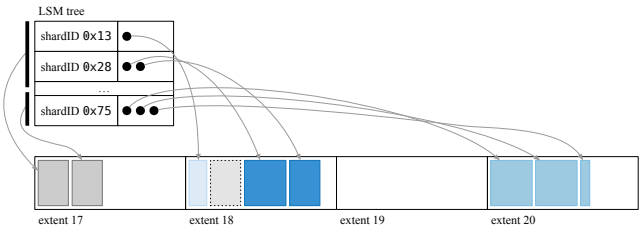
ShardStore is a key-value store currently being deployed within the Amazon S3 cloud object storage service. This section provides background on ShardStore's design and its crash consistency protocol.

Context. The ShardStore key-value store is used by S3 as a *storage node*. Each storage node stores *shards* of customer objects, which are replicated across multiple nodes for durability, and so storage nodes need not replicate their stored data internally. ShardStore is API-compatible with our existing storage node software, and so requests can be served by either ShardStore or our existing key-value stores.

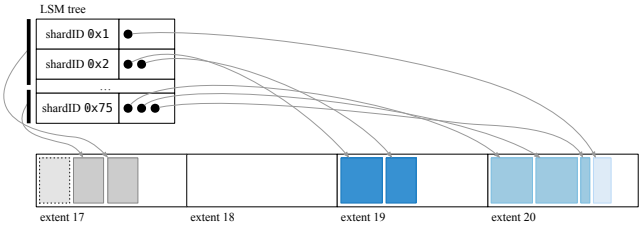
2.1 Design Overview

ShardStore presents a key-value store interface to the rest of the S3 system, where keys are shard identifiers and values are shards of customer object data. Customer requests are mapped to shard identifiers by S3's metadata subsystem [53].

ShardStore's key-value store comprises a log-structured merge tree (LSM tree) [41] but with shard data stored outside the tree to reduce write amplification, similar to WiscKey [31]. Figure 1 shows an overview of how this LSM tree stores object data on disk. The LSM tree maps each shard identifier to a list of (pointers to) *chunks*, each of which is stored within an *extent*. Extents are contiguous regions of physical storage on a disk; a typical disk has tens of thousands of extents. ShardStore requires that writes within each extent are sequential, tracked by a *write pointer* defining the next valid write position, and so data on an extent cannot be immediately overwritten. Each extent has a reset operation to return the write pointer to the beginning of the extent and allow overwrites.



(a) Initial state



(b) After reclamation of extent 18 and LSM-tree flush

Figure 1. ShardStore's on-disk layout. Each extent offers append-only writes. A log-structured merge tree (LSM tree) maps shards to their data, stored as chunks on the extents. The LSM tree itself is also stored as chunks on disk (on extent 17). In (a), extent 18 has an unreferenced chunk left by a deleted shard. To make that space available for reuse, *chunk reclamation* evacuates live chunks from extent 18 to elsewhere on disk, resulting in the state in (b).

Rather than centralizing all shard data in a single shared log on disk, ShardStore spreads shard data across extents. This approach gives us flexibility in placing each shard's data on disk to optimize for expected heat and access patterns (e.g., to minimize seek latency). However, the lack of a single log makes crash consistency more complex, as §2.2 describes.

Chunk storage and chunk reclamation. All persistent data is stored in chunks, including the backing storage for the LSM tree itself, as Fig. 1 shows. A *chunk store* abstraction arranges the mapping of chunks onto extents. The chunk store offers PUT(data) → locator and GET(locator) → data interfaces, where locators are opaque chunk identifiers and used as pointers. A single shard comprises one or more chunks depending on its size.

Because extents are append-only, deleting a shard cannot immediately recover the free space occupied by that shard's chunks. For example, in Fig. 1a, extent 18 has a hole left behind by a chunk whose corresponding shard has recently been deleted. To recover and reuse free space, the chunk store has a *reclamation* background task that performs garbage collection. Reclamation selects an extent and scans it to find all chunks it stores. For each chunk, reclamation performs a reverse lookup in the index (the LSM tree); chunks that are still referenced in the index are evacuated to a new extent and their pointers updated in the index as Fig. 1b shows, while unreferenced chunks are simply dropped. Once the entire extent has been scanned, its write pointer is reset and it is available for reuse. Resetting an extent's write pointer makes all data on that extent unreadable even if not yet physically overwritten (ShardStore forbids reads beyond an extent's write pointer), and so the chunk store must enforce a crash-consistent ordering for chunk evacuations, index updates, and extent resets.

As noted above, the LSM tree itself is also stored as chunks written to extents. Maintenance operations like LSM compaction can render these chunks unused. The free space those chunks occupy is reclaimed in the same way as above, except that the reverse lookup is into the LSM tree's metadata structure (stored on disk in a reserved metadata extent) that records locators of chunks currently in use by the tree.

RPC interface. ShardStore runs on storage hosts with multiple HDDs. Each disk is an isolated failure domain and runs an independent key-value store. Clients interact with ShardStore through a shared RPC interface that steers requests to target disks based on shard IDs. The RPC interface provides the usual request-plane calls (put, get, delete) and control-plane operations for migration and repair.

Append-only IO. To support both zoned and conventional disks, ShardStore provides its own implementation of the extent append operation in terms of the `write` system call. It does this by tracking in memory a *soft* write pointer for each extent, internally translating extent appends to `write` system calls accordingly, and persisting the soft write pointer for each extent in a superblock flushed on a regular cadence.

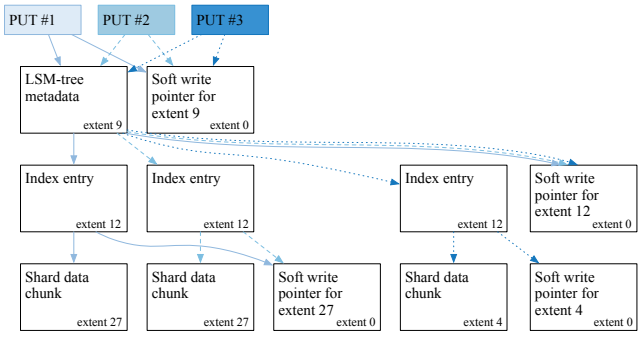
2.2 Crash Consistency

ShardStore uses a crash consistency approach inspired by soft updates [16]. A soft updates implementation orchestrates the order in which writes are sent to disk to ensure that *any* crash state of the disk is consistent. Soft updates avoid the cost of redirecting writes through a write-ahead log and allow flexibility in physical placement of data on disk.

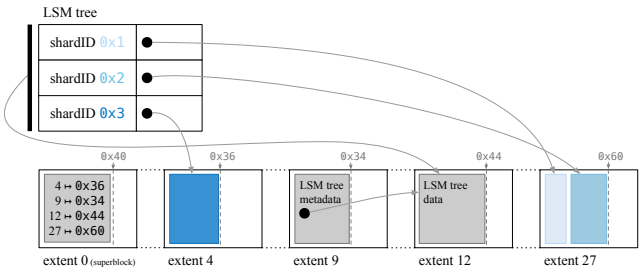
Correctly implementing soft updates requires global reasoning about *all possible orderings* of writebacks to disk. To reduce this complexity, ShardStore's implementation specifies crash-consistent orderings declaratively, using a `Dependency` type to construct dependency graphs at run time that dictate valid write orderings. ShardStore's extent append operation, which is the only way to write to disk, has the type signature:

```
fn append(&self, ..., dep: Dependency) -> Dependency
```

both taking as input and returning a dependency. The contract for the append API is that the append will not be issued to disk until the input dependency has been persisted. ShardStore's IO scheduler ensures that writebacks respect these dependencies. The dependency returned from append can be passed back into subsequent append operations, or first combined with other dependencies (e.g., `dep1.and(dep2)`)



(a) Dependency graph



(b) On-disk layout

Figure 2. Dependency graph for three put operations (a) and the corresponding on-disk data (b) after all three puts persist successfully. Each put is only durable once both the shard data and the index entry that points to it are durable. Writing to each extent also requires updating the soft write pointers stored in the superblock.

to construct more complex dependency graphs. Dependencies also have an `is_persistent` operation that clients can use to poll the persistence of an operation.

Example. Figure 2a shows dependency graphs for three put operations to ShardStore, which after completion result in the on-disk state in Fig. 2b. Each put's graph follows the same pattern involving three writes:

(a) the shard data is chunked and written to an extent.
(b) the index entry for the put is flushed in the LSM tree.
(c) the metadata for the LSM tree is updated to point to the new on-disk index data.

In addition to these writes, every time ShardStore appends to an extent it also updates the corresponding soft write pointer in the superblock (extent 0).

The dependency graphs for these puts are constructed dynamically at run time. The chunk store allocated the shard data chunks for puts #1 and #2 to the same extent on disk, so their writebacks can be coalesced into one IO by the scheduler, and thus their soft write pointer updates are combined into the same superblock update. The shard data chunk for put #3 is allocated to a different extent and so requires a separate soft write pointer update. All three puts arrive close enough together in time to participate in the same LSM-tree flush, which writes a new chunk of LSM tree data (on extent 12) and then updates the LSM tree metadata (on chunk 9) to point to that new chunk.

Why be crash consistent? Amazon S3 is designed for eleven nines of data durability, and replicates object data across multiple storage nodes, so single-node crash consistency issues do not cause data loss. We instead see crash consistency as reducing the cost and operational impact of storage node failures. Recovering from a crash that loses an entire storage node's data creates large amounts of repair network traffic and IO load across the storage node fleet. Crash consistency also ensures that the storage node recovers to a safe state after a crash, and so does not exhibit unexpected behavior that may require manual operator intervention.

3 Validating a Storage System

A production storage system combines several difficult-to-implement complexities [25]: intricate on-disk data structures, concurrent accesses and mutations to them, and the need to maintain consistency across crashes. The scale of a realistic implementation mirrors this complexity: ShardStore is over 40,000 lines of code and changes frequently.

Facing these challenges early in ShardStore's design process, we took inspiration from recent successes in storage system verification [8, 18, 49] and resolved to apply formal methods to increase our confidence. We chose formal methods because they would allow us to validate deep properties of ShardStore's implementation that are difficult to test with off-the-shelf tools at S3's scale and complexity—functional correctness of API-level calls, crash consistency of on-disk data structures, and correctness of concurrent executions including API calls and maintenance tasks like garbage collection. Given the complexity of the system and the rapid rate of change, we needed our results to outlast the involvement of formal methods experts, and so we sought a *lightweight* approach that could be automated and developed by the engineering team itself.

This section gives an overview of our approach to validating ShardStore, including the properties we focused on and how we distilled *reference model* specifications to check against the implementation. §4–7 detail how we check that the implementation conforms to these specifications. §8 summarizes our experiences with this approach, which has prevented subtle issues from reaching production.

3.1 Correctness Properties

Correctness for a production storage system is multifaceted: it should be durable in the absence of crashes, consistent in the presence of crashes and concurrency, highly available during normal operation, and meet our performance goals. Our validation effort considers availability and performance properties out of scope, as S3 has successfully

established other methods for validating these properties, including integration tests, load testing in pre-production environments, and staggered deployments with monitoring in production [29]. We chose to focus on durability and consistency properties, which are hard to establish by traditional testing alone and more difficult to recover from if violated.

In our approach, we ask developers to write an executable reference model (§3.2) to specify the expected state of the system after each API call. Our durability property then is that the model and implementation remain in equivalent states after each API call. Since the system is a key-value store, we define equivalence as having the same key-value mapping. This invariant establishes durability because it ensures that the implementation only loses or changes data when the specification allows (e.g., by a delete operation). However, this property is too strong in the face of two types of non-determinism: crashes may cause data loss that the reference model does not allow, and concurrency allows operations to overlap, meaning concurrent operations may be in flight when we check equivalence after each API call.

To address these limitations, we found it useful to decompose the durability property into three parts and reason about each separately:

1. For sequential crash-free executions, we check for equivalence directly (§4).
2. For sequential crashing executions, we extend the reference model to define which data can be lost after a crash, and check a corresponding weaker equivalence that establishes both durability and consistency (§5).
3. For concurrent crash-free executions, we write separate reference models and check linearizability (§6).

(We do not currently check properties of concurrent crashing executions because we have not found an effective automated approach.) This decomposition aids the specification effort by allowing us to separate concerns—the initial reference models are simple and easy to audit, and then we extend them separately to handle more complex properties. Decomposition also helps us scale the validation effort by using the most appropriate tools for each type of execution, including property-based testing [9] and stateless model checking [5, 35].

Additional properties. While these properties cover the core functionality of ShardStore, we also identified cases where checking localized properties (about undefined behavior, bounds checking, etc) with specialized tools would improve our confidence. §7 covers these properties and how we check them in a lightweight fashion.

3.2 Reference Models

For each ShardStore component we developed a *reference model*—an executable specification in Rust that provides the same interface as the component but using a simpler implementation. For instance, for the index component that maps shard identifiers to chunk locators, we define a reference model that uses a simple hash table to store the mapping, rather than the persistent LSM-tree described in §2.1.

Since the reference models provide the same interface as the implementation, ideally they should give identical results, so that equivalence would be a simple equality check. This is true on the happy path, but we found it very difficult to enforce strict equality for failures. The reference models can fail in limited ways (e.g., reads of keys that were never written should fail), but we choose to omit other implementation failures (IO errors, resource exhaustion, etc.) from the models. This choice simplifies the models at the expense of making checking slightly more complex and precluding us from reasoning about most availability or performance properties. §4.4 discusses failure testing in more detail.

Implementing the reference model as executable code is a design choice. We could instead have used a language intended for modeling (Alloy [23], Promela [22], P [12], etc.), which would have given us better expressiveness and built-in checking tools. However, we found that by writing reference models in the same language as the implementation, we make them easier for engineers to keep up to date. We also minimize the cognitive burden of learning a new language and mapping concepts between model and implementation.

Mocking. Writing reference models in the implementation language means we can also use them as mocks during unit testing. For example, unit tests at ShardStore's API layer use the index reference model (a hash map) as a mock of the index component, rather than instantiating the real LSM tree. This reuse helps keep the reference models up-to-date over time, as writing unit tests for new features often requires updating the mock. We have already seen several ShardStore developers extend the models this way in the course of their development work, as §8.2 details.

Model verification. The reduced complexity of the reference model makes it possible in principle to verify desirable properties of the model itself to increase confidence in its sufficiency (similar to *declarative specifications* in Hyperkernel [37]). For example, we could prove that the LSM-tree reference model removes a key-value mapping if and only if it receives a delete operation for that key. We have experimented with writing these proofs using the Prusti [2] verifier for Rust. Our early experience with these proofs has been limited but positive. We have been collaborating with the Prusti maintainers to open-source benchmarks based on our code and to implement the additional features we need for these proofs. We are also excited by rapid progress in auto-active [27] Rust verifiers [3, 14, 15, 40, 48] that promise to make this verification effort easier and more scalable.

4 Conformance Checking

Once we have a reference model specifying the expected behavior of the system, we need to check that the implementation conforms to the reference model according to the correctness properties above. This section details how we implement this check using property-based testing, and how we ensure good coverage of the possible behaviors of the implementation.

4.1 Property-Based Testing

To check that the implementation satisfies the reference model specification, we use *property-based testing* [9], which generates inputs to a test case and checks that a user-provided property holds when that test is run. Property-based testing can be thought of as an extension of fuzzing with user-provided correctness properties and structured inputs, which allow it to check richer behaviors than fuzzing alone. When a property-based test fails, the generated inputs allow the failure to be replayed (provided the test is deterministic).

We use property-based testing to check that implementation code *refines* the reference model: any observable behavior of the implementation must be allowed by the model. We frame this check as a property-based test that takes as input a *sequence of operations* drawn from an alphabet we define. For each operation in the sequence, the test case applies the operation to both reference model and implementation, compares the output of each for equivalence, and then checks invariants that relate the two systems. To ensure determinism and testing performance, the implementation under test uses an in-memory user-space disk, but all components above the disk layer use their actual implementation code.

Fig. 3 shows an example of this property-based test for the index component. The `IndexOp` enumeration (lines 1–7) defines an alphabet of allowed operations. It covers both the component's API operations (lines 2–4: `Get`, `Put`, etc.) and background operations such as reclamation and clean reboots (lines 5–6). These background operations are no-ops in the reference model (they do not change the key-value mapping of the index) but including them validates that their implementations do not corrupt the index. Each test run chooses a set of arbitrary sequences from the operation alphabet with arbitrarily chosen arguments (e.g., keys and values), and invokes `proptest_index` (line 10) with each sequence. For each operation in a sequence, the test applies it both to the implementation (line 17) and the reference (line 18) and compare the results as §3.2 describes. Finally (line 24), after each operation we perform additional invariant checks comparing the implementation and the reference (e.g., checking that both store the same key-value mapping).

```
1  enum IndexOp<Key, Value> {
2      Get(Key),
3      Put(Key, Value),
4      ...
5      Reclaim,
6      Reboot,
7  }
8
9  #[proptest]
10 fn proptest_index(ops: Vec<IndexOp<u32, u32>>) {
11     let mut reference = ReferenceIndex::new();
12     let mut implementation = PersistentLSMTIndex::new();
13     for op in ops {
14         match op {
15             Put(key, value) => {
16                 compare_results!(
17                     implementation.put(key, value),
18                     reference.put(key, value),
19                 );
20             }
21             Get(key) => { ... }
22             ...
23         }
24         check_invariants(&reference, &implementation);
25     }
26 }
```

Figure 3. Property-based test harness for the index reference model.

4.2 Coverage

Property-based testing selects random sequences of operations to test, and so can miss bugs. We reduce this risk by increasing the scale of testing—we routinely run tens of millions of random test sequences before every ShardStore deployment—but these tests can only ever check system states that the test harness is able to reach. The key challenge in using property-based testing for validation is ensuring that it can reach interesting states of the system. We do this by introducing domain knowledge into argument selection via biasing, and by employing code coverage mechanisms to monitor test effectiveness.

Argument bias. We apply biases in the property-based test harnesses when selecting arguments for operations in a test's alphabet. For example, a naive implementation of Fig. 3 would generate random keys for `Get` and `Put` operations, which would rarely coincide, and so would almost never test the successful `Get` path. We instead bias the argument to `Get` to prefer to choose from keys that were `Put` earlier. We also bias argument selection towards corner cases, such as read/write sizes close to the disk page size, which in our experience are frequent causes of bugs. These biases are always probabilistic: they only increase the chance of selecting desirable cases, but other cases remain possible. For example, we still want to test `Get`s that are unsuccessful, so we do not *require* the key to be one we have previously `Put`, only increase the probability that it is.

Argument biasing is important for increasing coverage, but it also has limitations. While elaborate argument selection strategies can often be easily justified ("our production customer traffic follows this distribution so we should mirror it here"), we have found little benefit to them. For example, we experimented with replicating production object size and

`Get`/`Put` ratio distributions with no effect. More generally, biasing introduces the risk of baking our assumptions into our tests, when our goal in adopting formal methods is to invalidate exactly such assumptions by checking behaviors we did not consider. Our methodology has settled on trusting default randomness wherever possible, and only introducing bias where we have quantitative evidence that it is beneficial (as in the page-size example above).

Coverage metrics. As the code evolves over time, new functionality may be added that is not reachable by the existing property-based test harness. At the interface level, a component may gain new API methods or arguments that need to be incorporated into the operation alphabet. At the implementation level, a component may gain new functionality (e.g., a cache) that affects the set of states reachable by the existing test harness. Both types of changes risk eroding the coverage of property-based tests and increase the chance that testing misses bugs. To mitigate these risks, our test harnesses generate code coverage metrics for the implementation code to help us identify blind spots that are not sufficiently checked, including newly added functionality that the reference model may not know about, and we tune our argument selection strategies to remedy them.

4.3 Minimization

Our property-based tests generate random sequences of operations as input, and when they find a failing test case, the sequence of operations can be used to reproduce the failure as a unit test. Most property-based testing tools automatically minimize the failing input to simplify the debugging experience [20, 32, 44]. Given a failing sequence of operations, the testing tool repeatedly applies reduction heuristics to simplify the sequence until the reduced version no longer fails. These reduction heuristics are generally simple transformations such as "remove an operation from the sequence" or "shrink an integer argument towards zero". They usually do not make any guarantees about finding a minimum failing input, but are effective in practice, and we have not found the need to develop custom heuristics. For example, when discovering bug #9 in Fig. 5, the first random sequence that failed the test had 61 operations, including 9 crashes and 14 writes totalling 226 KiB of data; the final automatically minimized sequence had 6 operations, including 1 crash and 2 writes totalling 2 B of data.

Although the minimization process is automated, we apply two design techniques to improve its effectiveness. First, we design ShardStore components to be as deterministic as possible, and where non-determinism is required we ensure it can be controlled during property-based testing. Non-determinism interferes with minimization because the reduction process stops as soon as a test execution does not fail. Determinism as a system design principle is not new, but non-determinism sneaks into modern code in surprising ways; for example, the default hash algorithm for `HashMap` types in Rust is randomized and so the iteration order of a map's contents varies across runs. Second, we design our alphabet of operations with minimization heuristics in mind. For example, the property-based testing tool we use [30] minimizes enumeration types by preferring earlier variants in the definition (i.e., it will prefer `Get` in the `IndexOp` enumeration in Fig. 3), and so we arrange operation alphabets in increasing order of complexity.

4.4 Failure Injection

In addition to checking correctness in ideal conditions, we also use property-based testing to check that ShardStore correctly handles failures in its environment. Hard disks fail at a non-trivial rate [43], and at scale these failures must be handled automatically without operator intervention. We consider three distinct classes of environmental failures:

1. Fail-stop crashes (e.g., power outages, kernel panics)
2. Transient or permanent disk IO failures (e.g., HDD failures, timeouts)
3. Resource exhaustion (e.g., out of memory or disk space)

Fail-stop crashes involve the ShardStore software itself crashing and recovering; §5 describes how we test crash consistency in this failure mode.

Disk IO failures. To test transient or permanent IO failures while ShardStore continues running, we extend our property-based tests to inject IO failures into their in-memory disk. We extend the operation alphabet with new failure operations (e.g., `FailDiskOnce(ExtentId)` causes the next IO to the chosen extent to fail) and allow the property-based test to generate those operations like any other.

Failure injection requires us to relax the conformance check in Fig. 3 slightly, as implementation operations may not be atomic with respect to injected failures. For example, if a single implementation operation performed three IOs, and one of those IOs suffered an injected failure, the other two IOs may have succeeded and so their effects will be visible. This would cause divergence between the reference model and the implementation—the reference model does not keep enough state to tell which effects should be visible after an IO failure—and so the strict `compare_results` check in Fig. 3 is too strong. One solution would be to track *all* possible states of the reference model after a failure, but this blows up quickly in the face of multiple failures. We could instead try to determine the semantic effect of individual IOs and so track which data a failure is expected to affect, but this would be fragile and make the reference model expensive to maintain.

Instead, the harness tracks a simple "has failed" flag that relaxes equivalence checks between reference and implementation after a failure is injected. The relaxed check allows operations to disagree with the reference model, but only in some ways; for example, a `Get` operation with an injected IO error is allowed to fail by returning no data, but is never allowed to

return the *wrong* data—we expect ShardStore components to detect and fail operations that involve corruption even in the presence of IO errors (e.g., by validating checksums). To ensure these relaxations do not cause our testing to miss bugs unrelated to failures, we separately run our property-based tests with and without failure injection enabled.

Resource exhaustion. We do not currently apply property-based testing for resource exhaustion failures such as running out of disk space. We have found these scenarios difficult to test automatically because we lack a correctness oracle: resource exhaustion tests need to distinguish real bugs (e.g., a space leak) from failures that are expected because the test allocates more space than the environment has available. However, all storage systems introduce some amount of space overhead and amplification (due to metadata, checksums, block alignment, allocation, etc.). To distinguish real failures requires accounting for these overheads, but they are intrinsic to the implementation and difficult to compute abstractly. We think adopting ideas from the resource analysis literature [21] would be promising future work to address this problem.

5 Checking Crash Consistency

Crash consistency can be a source of bugs in storage systems [42]. ShardStore uses a crash consistency protocol based on soft updates (§2.2), which introduces implementation complexity. For ShardStore, reasoning about crash consistency was a primary motivation for introducing formal methods during development, and so it was a focus of our efforts.

We validate crash consistency by using the user-space dependencies discussed in §2.2. Each mutating ShardStore operation returns a `Dependency` object that can be polled to determine whether it has been persisted. As a specification, we define two crash consistency properties in terms of these dependencies:

1. *persistence*: if a dependency says an operation has persisted before a crash, it should be readable after a crash (unless superseded by a later persisted operation)
2. *forward progress*: after a non-crashing shutdown, every operation's dependency should indicate it is persistent

Forward progress rules out dependencies being so strong that they never complete. We do not enforce any finer-grained properties about dependency strength, and so a dependency returned by the implementation could be stronger than strictly necessary to ensure consistency, but the persistence property ensures it is not too weak.

To check that the implementation satisfies these crash consistency properties, we extend the property-based testing approach in §4 to generate and check crash states. We augment the operation alphabet to include a `DirtyReboot(RebootType)` operation. The `RebootType` parameter controls which data in volatile memory is persisted to disk by the crash; for example, it can choose whether to flush the in-memory section of the LSM-tree, whether to flush the buffer cache, etc.

After reboot and recovery, the test iterates through the dependencies returned by each mutating operation on the implementation and checks the two properties above. For the `ReferenceIndex` example in Fig. 3, the persistence check is:

```
for (key, dependency) in dependency_map {
    assert!(
        !dependency.is_persistent()
          || reference.get(key) == implementation.get(key)
    );
}
```

Here, the check first polls each operation's `Dependency` object to determine whether it was persisted before the crash; if so, it then checks that the corresponding data can be read and has the expected value. If the check fails, there is a crash consistency bug, such as the `Dependency` graph missing a required ordering (edge) or write operation (vertex) to ensure consistency. To validate forward progress, the model checks the stronger property

```
assert!(dependency.is_persistent());
```

for each `dependency` on a clean reboot.

Block-level crash states. This approach generates crash states at a coarse granularity: the `RebootType` parameter makes a single choice for each component (e.g., the LSM tree) about whether its *entire* state is flushed to disk. Coarse flushes can miss bugs compared to existing techniques that exhaustively enumerate all crash states at the block level [42] or use symbolic evaluation to check all crash states within an SMT solver [4].

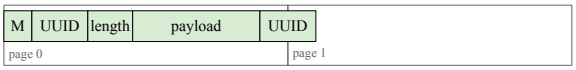
To increase the chance of finding bugs, the operation alphabet for the crash-consistency tests includes flush operations for each component that can be interleaved with other operations. For example, a history like `Put(0, 3)`, `IndexFlush`, `Put(1, 7)`, `DirtyReboot(None)` would flush the index entry for key `0` to the on-disk LSM tree, even though the later `DirtyReboot` does not specify any components to flush. The resulting crash state would therefore reflect an index that is only partially flushed.

We have also implemented a variant of `DirtyReboot` that *does* enumerate crash states at the block level, similar to BOB [42] and CrashMonkey [33]. However, this exhaustive approach has not found additional bugs and is dramatically slower to test, so we do not use it by default.

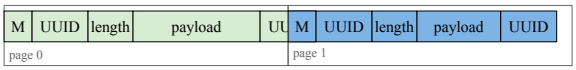
Example. Issue #10 in Fig. 5 was a crash consistency bug involving a planned change to chunk reclamation that our validation uncovered. Because our methodology is lightweight and automated, the developer was able to run property-based tests locally and discover this issue before even submitting for code review.

The issue arose from how chunk data is serialized. Chunk data is framed on disk with a two-byte magic header ("M" below) and a random UUID, repeated on both ends to allow

validating the chunk's length. The issue involved a chunk whose serialized form spilled the UUID onto a second page on an extent:



At this point, a crash occurs and loses the data on page 1 but not the data on page 0, which had been flushed before the crash. This data loss corrupts the chunk, but so far there is no consistency issue because we never considered the chunk persistent (as the Dependency graph for the chunk includes both page 0 and page 1). After reboot and recovery, a second chunk is written to the same extent, starting from page 1 (the current write pointer):



This second chunk is then flushed to disk, and so is considered persistent. Next, the scenario runs reclamation on this extent, which scans all the chunks on the extent. Normally this reclamation would see the first chunk is corrupted because its leading and trailing UUIDs do not match, and so it would skip to the second chunk, which it can decode successfully. However, this logic fails if the trailing bytes of the first chunk's UUID (the bytes that spilled onto page 1 and were lost in the crash) are the same as the magic bytes. In that case, the reclamation will successfully decode that first chunk, and then skip over the second chunk because reclamation does not expect overlapping chunks. Once the reclamation finishes, the second chunk becomes inaccessible, as resetting the extent prevents access to data stored on it. This outcome is a consistency violation: the Dependency for the second chunk was considered persistent once the chunk was flushed to disk, but after a subsequent reclamation, the chunk was lost.

This was a subtle issue that involved a particular choice of random UUID to collide with the magic bytes, a chunk that was the right size to just barely spill onto a second page, and a crash that lost only the second page. Nonetheless, our conformance checks automatically discovered and minimized this test case.

6 Checking Concurrent Executions

Our validation approach thus far deals only with sequential correctness, as the conformance checking approach in §4 tests only deterministic single-threaded executions. In practice, a production storage system like ShardStore is highly concurrent, with each disk servicing several concurrent requests and background maintenance tasks (e.g., LSM tree compaction, buffer cache flushing, etc). Rust's type system guarantees data-race freedom within the safe fragment of the language [52], but cannot make guarantees about higher-level race conditions (e.g., atomicity violations), which are difficult to test and debug as they introduce non-determinism into the execution.

To extend our approach to check concurrent properties, we hand-wrote harnesses for key properties and validated them using *stateless model checking* [17], which explores concurrent interleavings of a program. We use this approach both to check concurrent executions of the storage system and to validate some ShardStore-specific concurrency primitives. Stateless model checkers can both validate concurrency properties (e.g., consistency) and test for deadlocks (by finding interleavings that end with all threads blocked).

Properties and tools. In principle, we would like to check that concurrent executions of ShardStore are linearizable with respect to the sequential reference models. We check such fine-grained concurrency properties using the Loom stateless model checker for Rust [28], which implements the CDSChecker [39] algorithm for sound model checking in the release/acquire memory model, and uses bounded partial-order reduction [10] to reduce state explosion.

However, sound stateless model checking tools like Loom are not scalable enough to check linearizability of end-to-end ShardStore tests—even a relatively small test involves tens of thousands of atomic steps whose interleavings must be explored, and the largest tests involve over a million steps. For these tests, we developed and open-sourced the stateless model checker Shuttle [47], which implements randomized algorithms such as probabilistic concurrency testing [5]. The two tools offer a soundness–scalability trade-off. We use Loom to soundly check all interleavings of small, correctness-critical code such as custom concurrency primitives, and Shuttle to randomly check interleavings of larger test harnesses to which Loom does not scale such as end-to-end stress tests of the ShardStore stack.

Fig. 4 shows an example of a Loom test harness for ShardStore's LSM-tree-based index (a corresponding Shuttle test harness looks similar). The test mocks out the persistent chunk storage that backs the LSM tree as a conceit to scalability. The harness first initializes the index's state with a fixed set of keys and values. It then spawns three concurrent threads: chunk reclamation for a single arbitrarily chosen extent (line 11), LSM tree compaction for the index (line 14), and a thread that overwrites some keys in the index and then reads them back (line 17). The test checks read-after-write consistency for a fixed read/write history: interleavings of concurrent background threads between the line 20 write and line 21 read should not affect the read-after-write property.

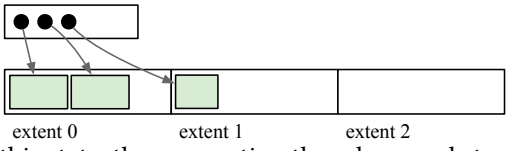
Example. Issue #14 in Fig. 5 was a race condition in the LSM tree implementation that was caught by the Loom test harness in Fig. 4 before even reaching code review. The LSM tree implementation uses an in-memory metadata object to track the set of chunks that are currently being used to store LSM tree data on disk. This metadata is mutated by two concurrent background tasks: (1) LSM tree compaction flushes the

```
loom::model(|| {
    let chunk_store = MockChunkStore::new();
    let index = PersistentLSMTIndex::new(chunk_store);

    // Set up some initial state in the index
    for (key, value) in &[...] {
        index.put(key, value);
    }

    // Spawn concurrent operations
    let t1 = thread::spawn(|| {
        chunk_store.reclaim();
    });
    let t2 = thread::spawn(|| {
        index.compact();
    });
    let t3 = thread::spawn(|| {
        // Overwrite keys and check the new value sticks
        for (key, value) in &[...] {
            index.put(key, value);
            let new_value = index.get(key);
            assert_eq!(new_value, value);
        }
    });

    t1.join(); t2.join(); t3.join();
})
```

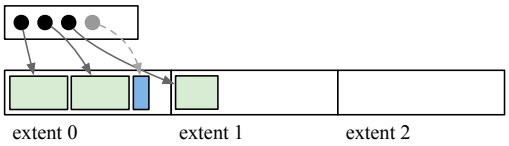
Figure 4. Stateless model checking harness for the index. The test validates that the index exhibits read-after-write consistency for a set of keys under concurrent background maintenance tasks (chunk reclamation and LSM tree compaction). The Loom stateless model checker [28] explores all concurrent interleavings of the closure passed to `model`. (Some `Arc` code to please the borrow checker is omitted.)

in-memory section of the index to disk, creating new chunks to be added to the metadata and removing chunks whose contents have been compacted; and (2) chunk reclamation scans extents used by the LSM tree to recover free space, and updates the metadata to point to relocated chunks.

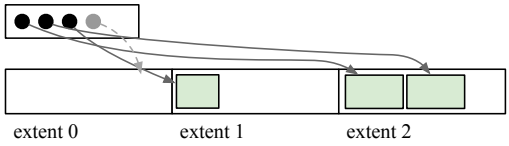
The issue involved a concurrent compaction and reclamation on an index where the metadata pointed to three chunks currently storing the LSM tree:



From this state, the compaction thread ran and stored the in-memory section of the LSM tree into a new chunk, which was small enough to write into extent 0:



The compaction thread's next step is to update the in-memory metadata to include a pointer to the new chunk (the dashed pointer above). However, before it could do this, it was preempted by the reclamation thread, which chose to perform a reclamation on extent 0. The reclamation thread scanned the three chunks on that extent. The first two were referenced by the metadata, and so were evacuated to extent 2. However, the newly written chunk was not yet referenced by the metadata (as the compaction thread was preempted before it could update the metadata), and so the reclamation dropped the new chunk and reset the extent:



Now when the compaction thread resumes and updates the metadata, it will introduce a dangling pointer to a chunk that was dropped by reclamation, and so the index entries the LSM tree wanted to store on that chunk are lost. The fix was to make compaction lock the extents it writes new chunks into until it can update the metadata to point to them.

7 Other Properties

The validation approach described so far covers the properties in §3.1. However, while developing ShardStore, we identified two more localized classes of issues that our existing approach would not detect, and adopted specific reasoning techniques to check for them.

Undefined behavior. ShardStore is written in Rust, which ensures type and memory safety for the safe subset of the language. However, as a low-level system, ShardStore requires a small amount of *unsafe* Rust code, mostly for interacting with block devices. Unsafe Rust is exempt from some of Rust's safety guarantees and introduces the risk of undefined behavior [1]. To rule out this class of bugs, we run ShardStore's test suite under the Miri interpreter [50], a dynamic analysis tool built into the Rust compiler that can detect certain classes of undefined behavior. We had to extend Miri to support basic concurrency primitives (threads and locks), and have upstreamed that work to the Rust compiler.

Serialization. ShardStore has a number of serializers and deserializers that marshal data between in-memory and on-disk formats. Both bit rot and transient failures can corrupt on-disk data, and so we treat data read from disk as untrusted. We want to ensure that deserialization code running on such untrusted data is robust to corruption. We worked with the developers of Crux [51], a bounded symbolic evaluation engine, to prove panic-freedom of our deserialization code. We proved that for any sequence of on-disk bytes (up to a size bound), our deserializers cannot panic (crash the thread) due to out of bounds accesses or other logic errors. We also fuzz the same deserializer code on larger inputs to work around the small size bound required to make symbolic evaluation tractable. We also supplied benchmarks based on these problems to the Crux developers to aid in further development.

8 Experience

This section reports on our experience validating ShardStore, including the effort to perform validation and the level of adoption by the engineering team.

8.1 Bugs Prevented

Figure 5 presents a catalog of issues detected by our validation effort, broken down by the top-level property violated. Two of these issues were presented in more detail in §5 and §6. Each of these issues was prevented from reaching production, and detected using the mechanisms described in §4–6 (property-based testing for functional correctness and crash consistency issues; stateless model checking for concurrency issues). Most issues related to data integrity—they could cause data to be lost or stored incorrectly—while one concurrency issue (#12) was an availability issue caused by a deadlock. Each issue was detected automatically by our tools, although diagnosis remains a developer-intensive manual effort aided by automated test-case minimization for property-based testing.

The issues in Fig. 5 span most of ShardStore's components and suggest that our approach has been effective at preventing bugs in practice. One aspect these results do not measure is effectiveness at detecting bugs early in development, as they include only issues that reached (and were blocked by) our continuous integration pipeline. Anecdotally, developers have told us of more bugs our checks detected during local development that never made it to to code review.

8.2 Validation Effort

Figure 6 shows the size of the ShardStore code base. Excluding the reference models and validation test harnesses described in this paper, unit and integration tests comprise 31% of the code base. Figure 6 also tallies the size of the reference models and validation test harnesses for each of the top-level properties in §3.1. From the code base's perspective, the validation tests are just unit tests like any other: they are run using Rust's built-in testing infrastructure, and they are distinguished from other tests only by naming conventions and module hierarchy. These artifacts combined are only 13% of the total code base and 20% of the size of the implementation code, an overhead that compares favorably to formal verification approaches that report 3–10× overhead [8, 18, 49].

Development of the reference models and test harnesses was led initially by two formal methods experts working full-time for nine months and a third expert who joined for three months. However, since that initial effort, most work on them has been taken over by the ShardStore engineering team, none of whom have any prior formal methods experience. As a crude measure of this adoption, 18% of the lines of code in the test harnesses were last edited by a non-formal-methods expert according to git blame. Three engineers have each written more than 100 lines of code extending these tests, including crash consistency conformance checks for new functionality. Four engineers have written new stateless model checking harnesses for concurrent code they have added, and the need for such harnesses is now a standard question during code review of new concurrent functionality.

8.3 Limitations

As discussed in §4, our validation checks can miss bugs—their reporting success does not mean the code is correct, only that they could not find a bug. We are not currently aware of any bugs that have reached production after being missed due to this limitation, but of course we cannot rule it out. We have seen one example of a bug that our validation should have caught but did not, and was instead caught during manual code review. That issue involved an earlier code change that had added a new cache to a ShardStore component. Our existing property-based tests had trouble reaching the cache-miss code path in this change because the cache size was configured to be very large in all tests. The new bug was in a change to that cache-miss path, and so was not reached by the property-based tests; after reducing the cache size, the tests automatically found the issue. This missed bug was one motivation for our work on coverage metrics §4.2.

A non-trivial amount of ShardStore's code base is parsing of S3's messaging protocol, request routing, and business logic, which we are still working on validating. We also do not have a complete reference model (§3.2) for some operations ShardStore exposes to S3's control plane. As future work, we plan to model the parts of these control plane interactions that are necessary to establish durability properties.

8.4 Lessons Learned

Our lightweight formal methods approach was motivated by two goals: to detect bugs as early as possible in development, and to ensure that our validation work remains relevant as the code changes over time. We learned useful lessons in both directions in the course of doing this work.

Early detection. We began formal methods work on ShardStore early in development, when the design was still being iterated on and code was being delivered incrementally. To fit this development process, we focused on delivering models and tests one component at a time. As soon as we had a reference model for one ShardStore component (chunk storage and reclamation), we integrated that into the code base and began training engineers, while in parallel extending the models to cover more components. This approach both helped detect bugs early and increased interest in the results of our work—having seen the issues detected even in this one component, engineers encouraged us to make the tests release blockers for continuous delivery, while also independently starting to develop models and properties for the components they owned.

ID	Component	Description
Functional Correctness		
#1	Chunk store	Off-by-one error in reclamation for chunks of size close to `PAGE_SIZE`
#2	Buffer cache	Cache was not correctly drained after resetting an extent
#3	Index	Metadata was not flushed correctly during shutdown if an extent was reset
#4	API	Shards could be lost if a disk was removed from service and then later returned
#5	Chunk store	Reclamation could forget chunks after a transient read IO error
Crash Consistency		
#6	Superblock	Superblock `Dependency` for extent ownership was incorrect after a reboot
#7	Superblock	Mismatch between soft and hard write pointers in a crash after an extent reset
#8	Buffer cache	Writes did not include a dependency on the soft write pointer update
#9	Chunk store	Reference model was not updated correctly after a crash during reclamation
#10	Chunk store	Reclamation could forget chunks after a crash and UUID collision
Concurrency		
#11	Chunk store	Chunk locators could become invalid after a race between write and flush
#12	Superblock	Buffer pool exhaustion could cause threads waiting for a superblock update to deadlock
#13	API	Race between control plane operations for listing and removal of shards
#14	Index	Race between reclamation and LSM compaction could lose recent index entries
#15	Chunk store	Reference model could re-use chunk locators, which other code assumed were unique
#16	API	Race between control plane bulk operations for creating and removing shards

Figure 5. ShardStore issues prevented from reaching production by our validation effort.

Component	Lines
ShardStore	
Implementation	44,048
Unit tests & integration tests	19,540
Specification	
Reference models (§3.2)	450
Validation	
Functional correctness checks (§3)	4,860
Crash consistency checks (§5)	2,661
Concurrency checks (§6)	901
Total	72,460

Figure 6. Lines of code for ShardStore implementation and validation artifacts. All lines are Rust code.

By focusing on modeling components whose APIs had become stable, we avoided too much churn in the specification and code for the models, which would have made them much more expensive to maintain. Had we started looking at formal methods later, when design and code were more stable, we would have been tempted to write a single reference model for just the public interface to ShardStore. This approach would lower the overhead of writing specifications, but we think it would have been the wrong decision and would have caught fewer bugs. We found it much easier to exercise corner case scenarios (especially fault scenarios) by writing tests that directly exercise internal component APIs, and engineers have found it easier to debug and fix failures in their own changes by not having to trace them back through the entire implementation stack.

Continuous validation. Tightly integrating reference models into the code base has been crucial for ensuring that our validation remains effective as the code changes over time. Developers must update the models whenever code changes in order to avoid breaking the build; having the models in Rust makes the overhead of these updates low enough to be practical. From the formal methods perspective, this integration is a way to convince engineers to write specifications themselves. In practice, after the first iteration of the reference models was written by formal methods experts, they have been updated almost exclusively by the engineering team.

We focused heavily on lowering the marginal cost of *future* validation: we would not have considered this work successful if future code changes by engineers required kicking off new formal methods engagements. Early in our work, we wrote reference models using modeling languages we were familiar with (Alloy [23], SPIN [22], and Yggdrasil-style Python [49]) and imagined developing tooling to check the Rust code against them. It was only when we discussed long-term maintenance implications with the team that we realized writing the models themselves in Rust was a much better choice, and even later when we realized the reference models could serve double duty as mocks for unit testing.

Future directions. While we're pleased with the effectiveness of our validation work for ShardStore, we're not done. Our work so far has focused on modeling and checking key correctness properties (such as crash consistency) of ShardStore viewed as a single storage node. But Amazon S3 is a complex distributed system with hundreds of microservices, several of which interact with ShardStore storage nodes to

replicate customer data and perform control plane operations. AWS has been using the P language for asynchronous programs [11–13] to validate the correctness of new S3 features such as strong consistency [36], and using TLA+ to validate the designs of a number of systems [38]. We are exploring ways of combining P with Rust to extend these results to include reasoning about ShardStore's role in the S3 system. We also continue to explore ways to strengthen the guarantees provided by our validation work, including the recent rapid advances in Rust verification tools [45].

9 Related Work

Storage verification. There has been significant recent interest in formally verified storage systems. Yggdrasil [49] is a "push-button" verified file system implementation that formalizes a refinement to define allowed states after a crash. FSCQ [8] is a formally verified filesystem that instead adopts a crash-aware Hoare logic to define the allowed crash behaviors of each POSIX system call. VeriBetrKV [18] is a more recent effort to verify a key-value store with a focus on automated proofs that delivers stronger guarantees than we do at the cost of more overhead ("tedium" in their terms): 7 lines of proof for every line of implementation. Our approach does not have the soundness guarantees of these examples—we can miss bugs these systems would have caught—but our lightweight tools dramatically increase the ability for our engineering teams to adopt and integrate formal methods into their development practice.

We considered dealing with concurrency essential for our approach to be successful. Recent verification efforts for concurrent storage systems have focused on simple replicated disks [6] and journaling file systems [7]. ShardStore is substantially more complex than these designs, and so we focused on lightweight stateless model checking and on providing a good debugging experience for concurrent test failures using the Shuttle model checker [47], again at the cost of potentially missing bugs.

Storage testing. Several projects have explored using model checking or fuzzing techniques to find bugs in storage system, with a particular focus on crash consistency. EXPLODE [54] uses model checking to validate a file system against a user-written test harness. We take a similar approach but with richer specifications and less reliance on user-written harnesses—with property-based testing we ask the user to write only an operation alphabet, and the testing harness chooses inputs to run. Ferrite [4] includes an SMT-based model checker for detecting file system crash consistency bugs, but reasons only about an abstract model of the file system rather than implementation code. CrashMonkey [33] checks metadata crash consistency of an unmodified file system by running small workloads and interposing at the block layer to trace writes. We take a coarser-grained approach to crash consistency (§5) that scales our testing to larger scenarios and covers data consistency.

Storage specifications. The POSIX standard for file systems is imprecise prose, and so difficult to build a validation approach around. SibylFS [46] is an effort to build an executable formal model in Lem [34] of the POSIX file system interface and test conformance of file system implementations. Kang and Jackson [26] develop a formal model of a flash file system in Alloy [23] and check its correctness against an abstract POSIX specification. In contrast to these approaches, our reference model specifications are written in the same language as the implementation, making them easier to update by an engineering team as requirements change over time.

10 Conclusion

ShardStore is a new storage backend for Amazon S3 for which we decided early in the design process to involve formal methods. Our experience with lightweight formal methods has been positive, with a number of issues prevented from reaching production and substantial adoption by the ShardStore engineering team. We are excited to further improve our results by applying stronger verification techniques and expanding the scope of our effort to validate more of S3.

Acknowledgments

We thank the anonymous reviewers, Murat Demirbas, Emina Torlak, Xi Wang, and our shepherd Manos Kapritsos for their feedback on this paper. We thank Stuart Pernsteiner for his work applying the Crux symbolic execution engine to ShardStore. Finally, we thank the entire S3 ShardStore team at AWS for their collaboration and ongoing support of this effort.

References

[1] Vytautas Astrauskas, Christoph Matheja, Federico Poli, Peter Müller, and Alexander J. Summers. 2020. How do programmers use unsafe Rust? *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 136:1–136:27.

[2] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30.

[3] Marek Baranowski, Shaobo He, and Zvonimir Rakamaric. 2018. Verifying Rust Programs with SMACK. In *Proceedings of the 16th International Symposium on Automated Technology for Verification and Analysis (ATVA)*. Los Angeles, CA, USA, 528–535.

[4] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. 2016. Specifying and checking file system crash-consistency models. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 83–98.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. 2010. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th International*

Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). Pittsburgh, PA, USA, 167–178.

[6] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. 2019. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)*. Huntsville, ON, Canada, 243–258.

[7] Tej Chajed, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. 2021. GoJournal: a verified, concurrent, crash-safe journaling system. In *Proceedings of the 15th Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual, 423–439.

[8] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2015. Using Crash Hoare Logic for Certifying the FSCQ File System. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, USA, 18–37.

[9] Koen Claessen and John Hughes. 2000. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Montreal, Canada, 268–279.

[10] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. 2013. Bounded partial-order reduction. In *Proceedings of the 28th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Indianapolis, IN, USA, 833–848.

[11] Ankush Desai. 2021. P. https://github.com/p-org/p

[12] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. 2013. P: safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Seattle, WA, USA, 321–332.

[13] Ankush Desai, Amar Phanishayee, Shaz Qadeer, and Sanjit A. Seshia. 2018. Compositional programming and testing of dynamic distributed systems. *Proc. ACM Program. Lang.* 2, OOPSLA (2018), 159:1–159:30.

[14] Craig Disselkoen, Sunjay Cauligi, Dean Tullsen, and Deian Stefan. 2020. Finding and Eliminating Timing Side-Channels in Crypto Code with Pitchfork. In *TECHCON*.

[15] Facebook. 2021. MIRAI. https://github.com/facebookexperimental/MIRAI

[16] Gregory R. Ganger and Yale N. Patt. 1994. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI)*. Monterey, CA, USA, 49–60.

[17] Patrice Godefroid. 1997. Model Checking for Programming Languages using Verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. Paris, France, 174–186.

[18] 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 Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event, 99–115.

[19] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (1990), 463–492.

[20] Ralf Hildebrandt and Andreas Zeller. 2000. Simplifying failure-inducing input. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*. Portland, OR, USA, 135–145.

[21] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV)*. Berkeley, CA, USA, 781–786.

[22] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Trans. Softw. Eng.* 23, 5 (May 1997), 279–295.

[23] Daniel Jackson. 2009. *Software Abstractions: logic, language, and analysis* (2nd ed.). MIT Press.

[24] Daniel Jackson and Jeannette Wing. 1996. Lightweight Formal Methods. *Computer* 29, 4 (1996).

[25] Rajeev Joshi and Gerard J. Holzmann. 2007. A Mini Challenge: Build a Verifiable Filesystem. *Formal Aspects of Computing* 19, 2 (June 2007), 269–272.

[26] Eunsuk Kang and Daniel Jackson. 2008. Formal Modeling and Analysis of a Flash Filesystem in Alloy. In *Proceedings of the 1st International Conference of Abstract State Machines, B, and Z (ABZ)*. London, UK.

[27] K. Rustan M. Leino and Michał Moskal. 2010. Usable Auto-Active Verification. In *Workshop on Usable Verification*. Redmond, WA, USA.

[28] Carl Lerche. 2020. Loom. https://github.com/tokio-rs/loom

[29] Clare Ligouri. 2020. Automating safe, hands-off deployments. Amazon Builders' Library. https://aws.amazon.com/builders-library/automating-safe-hands-off-deployments/

[30] Jason Lingle. 2020. Proptest. https://github.com/AltSysrq/proptest

[31] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, CA, USA, 133–148.

[32] David Maciver and Alastair F. Donaldson. 2020. Test-Case Reduction via Test-Case Generation: Insights from the Hypothesis Reducer. In *Proceedings of the 34th European Conference on Object-Oriented Programming (ECOOP)*. Berlin, Germany, 13:1–13:27.

[33] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th Symposium on Operating Systems Design and Implementation (OSDI)*. Carlsbad, CA, USA, 33–50.

[34] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. 2014. Lem: reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. Gothenburg, Sweden, 175–188.

[35] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. 2008. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, CA, USA, 267–280.

[36] Vishwas Narendra, Serdar Tasiran, and Ankush Desai. 2021. Amazon S3 Strong Consistency. https://www.youtube.com/watch?v=B0yXz6EeCaA

[37] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. 2017. Hyperkernel: Push-Button Verification of an OS Kernel. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*. Shanghai, China, 252–269.

[38] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. 2015. How Amazon Web Services Uses Formal Methods. *Commun. ACM* 58, 4 (March 2015), 66–73.

[39] Brian Norris and Brian Demsky. 2013. CDSchecker: checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 28th ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. Indianapolis, IN, USA, 131–150.

[40] Project Oak. 2021. Rust Verification Tools. https://github.com/project-oak/rust-verification-tools/

[41] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (June 1996), 351–385.

[42] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and*

Implementation (OSDI). Broomfield, CO, USA, 433–448.

[43] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Failure Trends in a Large Disk Drive Population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*. San Jose, CA, USA, 17–28.

[44] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, 335–346.

[45] Alastair Reid and Shaked Flur. 2021. Rust Verification Tools: Retrospective. https://project-oak.github.io/rust-verification-tools/2021/09/01/retrospective.html

[46] Tom Ridge, David Sheets, Thomas Tuerk, Andrea Giugliano, Anil Madhavapeddy, and Peter Sewell. 2015. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*. Monterey, CA, USA, 38–53.

[47] Amazon Web Services. 2020. Shuttle. https://github.com/awslabs/shuttle

[48] Amazon Web Services. 2021. Rust Model Checker (RMC). https://github.com/model-checking/rmc

[49] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*. Savannah, GA, USA, 1–16.

[50] Miri team. 2021. Miri. https://github.com/rust-lang/miri

[51] Aaron Tomb. 2020. Crux: Introducing our new open-source tool for software verification. https://galois.com/blog/2020/10/crux-introducing-our-new-open-source-tool-for-software-verification/

[52] Aaron Turon. 2015. Fearless Concurrency with Rust. https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html

[53] Werner Vogels. 2021. Diving Deep on S3 Consistency. https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html

[54] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. 2006. eXplode: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, WA, USA, 131–146.