Certifying Checksum-Based Logging in the RapidFSCQ Crash-Safe Filesystem

by

Stephanie Wang

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2016

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2016

Certified by . . . .
Dr. Frans Kaashoek
Charles Piper Professor
Thesis Supervisor

Certified by . . . .
Dr. Nickolai Zeldovich
Associate Professor
Thesis Supervisor

Accepted by . . . .
Dr. Christopher Terman
Chairman, Masters of Engineering Thesis Committee

Certifying Checksum-Based Logging in the RapidFSCQ Crash-Safe Filesystem

by

Stephanie Wang

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As more and more software is written every day, so too are bugs. Formal verification is a way of using mathematical methods to prove that a program has no bugs. However, if formal verification is to see widespread use, it must be able to compete with unverified software in performance. Unfortunately, many of the optimizations that we take for granted in unverified software depend on assumptions that are difficult to verify. One such optimization is data checksums in logging systems, used to improve I/O efficiency while still ensuring data integrity after a crash.

This thesis explores a novel method of modeling the probabilistic guarantees of a hash function. This method is then applied to the logging system underlying RapidFSCQ, a certified crash-safe filesystem, to support formally verified checksums. An evaluation of RapidFSCQ shows that it enables end-to-end verification of application and filesystem crash safety, and that RapidFSCQ's optimizations, including checksumming, achieve I/O performance on par with Linux ext4. Thus, this thesis contributes a formal model of hash function behavior with practical application to certified computer systems.

Thesis Supervisor: Dr. Frans Kaashoek
Title: Charles Piper Professor

Thesis Supervisor: Dr. Nickolai Zeldovich
Title: Associate Professor

Acknowledgments

Five years ago, when I first came to MIT as a freshman who didn't even know what a computer program looked like, I never thought a day like this would come. I found a community at MIT like no other, and the first place where I truly felt I belonged.

First to Frans and Nickolai, for introducing me to a new world. They were my first introduction to computer systems back when I had figured out what a program looked like but didn't know what to do with that information. This year, they showed me what a vision looks like and how to make that vision happen. I could not have asked for a better introduction to research. I leave MIT this year knowing that I will have their support wherever I go.

I also owe thanks to Frans and Nickolai, and especially to Haogang Chen, for the many other lines of proof that went into RapidFSCQ. RapidFSCQ is much more than a checksum-based log, and much more than this thesis. Indeed, I would not have this thesis today if it wasn't for Haogang's mastery of the logging system.

I have many good friends to thank for being there along the way and for making me laugh at an inappropriate volume. Thank you to Neha, the one and only bridgetroll in my life; Josh, my personal assistant; Zac and David, my other two thirds; Bennett, my buddy; Jaimit [sic], Nadia, Denis, Nikki, Max, Leigh, Dario, and the others of Burton Third. Thank you also to my G9 pals, James, who overdelivered; and Shoumik, the antisocial barista.

Finally, thank you to Mommy, Daddy, and Jie, to whom I am more grateful than I ever show. They may not always agree on what I should do, but they have always supported me on what I choose to do.

Contents

List of Figures

Chapter 1

Introduction

It's becoming clearer than ever that the current methods for guaranteeing software correctness are not enough. Most industry programmers rely on code reviews, testing, and occasionally static analysis to eliminate bugs. Unfortunately, none of these methods are exhaustive. Other methods of detecting bugs like model checking take a prohibitively long time on complex systems and have no way of producing an executable that is guaranteed to match the model [44]. Nuanced bugs can and do slip through the cracks, sometimes with disastrous results.

Filesystems in particular suffer from subtle bugs that, when combined with a crash at an inopportune time, can lead to critical data loss or data disclosure [27, 13]. This can be catastrophic for applications such as databases that depend on filesystems to persist data across crashes in a consistent manner.

Formal verification can be applied to filesystems to both specify and prove crash consistency, as shown in the FSCQ [9] filesystem. FSCQ is groundbreaking in that it guarantees a precisely defined disk state consistent with application behavior after any possible crash interleaving. However, the write-ahead log that the filesystem is built atop is too simple to achieve good performance, and in fact the FSCQ filesystem performs many times slower than the extensively optimized but unverified Linux ext4. Before verification can see widespread use in practical systems, we must show that it is feasible to achieve performance comparable with that of unverified systems.

ext4 offers several different mounting options for logging optimizations, such as group commit and checksumming. These features provide good performance by maximizing I/O efficiency. Unfortunately, these optimizations also increase the complexity of the logging system, making properties like crash safety difficult to reason about. For example, a bug that only appeared when two mounting options were simultaneously turned on could cause data thought to be deleted to resurface after a crash [27]. If we could formally verify the sophisticated optimizations offered by ext4, we could prevent such bugs *and* achieve high performance. However, the assumptions that these optimizations depend on are hard to formalize and verify, as evidenced by the bugs that appear when we assume incorrectly.

In this thesis, I focus on checksumming, a common logging feature that improves I/O efficiency of transactions while still ensuring data integrity after a crash. The challenge in this project is twofold. First, there is the problem of modeling hash function behavior in a sound way, described in chapter 3. Second, the hashing model must be incorporated into an actual logging system to support checksums, described in chapter 4. These two problems are described in more detail in the following two sections.

1.1 Modeling hash collisions

The core idea of data checksumming depends on the low probability of collision in the hash function we choose. Briefly, whenever writing data to disk, we also write its corresponding checksum. Then, when recovering from a crash, we read out the data stored on disk, compute its checksum, and then check it against the checksum written to disk from before the crash. If the checksums match, then with high probability, the data we read out after the crash is the same as the data we wrote to disk before the crash.

However, there is a very small but nonzero probability that even when the checksums do match, the inputs were different. This is true of any hash function, since by definition, a hash function is a mapping of arbitrarily large inputs to a fixed-size output. In practice, the probability of a hash collision is so low that we can ignore it.

Still, we must account for it in our model of hash function behavior. If we ignored it by stating an axiom that our hash function has no collisions, this would be unsound, meaning that we introduced a statement that implies logical *False*. Since *False* implies any statement, we would no longer be able to guarantee anything of value; all statements would be trivially true. Conversely, if we went so far as to to reason about the degree of collision resistance of a hash function, we'd quickly become trapped into reasoning about probabilities and specific hash function properties that are irrelevant to actual program execution.

The challenge is then how to succinctly capture a hash function's "mostly injective" nature without introducing any false axioms. The key idea is to treat hash collisions as function non-termination in the formal execution semantics. More specifically, in our formal definition of hash function execution, if we encounter a hash collision, then the abstract program enters an infinite loop. Otherwise, we simply return the hash value. In a dependently typed proof language like Coq, we can only prove guarantees for terminating programs [22]. By defining execution semantics in this way, we're able to prove that if a hash function does return, a collision could not have happened. Then, at any point in program execution, if we see two matching hash outputs, we can further reason that they must have come from the same inputs, or else there would've been a collision earlier in the execution that prevented us from reaching the current point.

We describe the precise definition of our hash function model in chapter 3, but the essential idea is to keep *ghost state*, or state used to supplement proofs but that never materializes during execution, to keep track of any hash inputs seen so far. To determine whether there is a hash collision, we look up the hash input in the list of inputs seen so far, using a data structure that we call a *hashset*.

To apply the same idea towards systems that can crash, we must be able to extend this model to hold across some formal definition of a crash. Otherwise, we would encounter the same problem while trying to verify a recovery procedure. During recovery, the checksum that we compare against comes from an input hashed before the crash. Thus, in order to be sure that there are no hash collisions, we must track the hashset across all crashes. To simplify reasoning across program execution execution and potential crashes, we introduce the idea of *hash_ subset*, a relationship that states that one hashset is a subset of another.

Finally, we use these low-level execution details to specify and implement programs that perform useful hashing operations, such as comparing two hash values or computing the of a list of values. These basic procedures demonstrate the practicality of the hashing model and are instrumental to supporting checksums in a logging system.

1.2 Verifying checksums in a crash-safe log

A write-ahead log works by writing transactions to the log at commit time, and then later applying the transactions to the data, e.g., the disk. It must account for the possibility that a crash occurs during a write to the log. In this case, upon restart, the log could contain a mixture of committed entries and garbage data. Typically, a recovery procedure is run to distinguish between these and to re-apply any valid committed entries. This requires careful ordering of disk writes and write barriers when appending to the log. For example, in the relatively simple FSCQ design, the log inserts a write barrier between writing log entries to disk and writing the corresponding commit metadata to the log header. This ensures that no matter where the append procedure may crash, the log data corresponding to the commit metadata in the header will already be on disk. In particular, the recovery procedure will see either the old metadata, whose corresponding log entries will still be on disk because the log is append-only, or the new metadata, in which case the extra write barrier ensures that the new transaction will be on disk.

This design is easy to reason about, but performs poorly because of the extra write barrier required per transaction. A more sophisticated design could use data checksums to remove this extra write barrier. At a high level, the logging system would write new log entries and commit metadata, including the new checksum, together to disk before flushing all outstanding writes. In this design, we're still able to guarantee the same crash safety properties because the recovery procedure can compare the checksum of the data on disk against the checksum in the header on disk to determine whether the data is valid.

The primary challenge is to design a write-ahead log that internally uses checksums to support the same guarantees as a log like FSCQ's: that committed entries are never lost and that garbage data is not mistaken for committed entries. Meanwhile, we also want to keep the interface simple. The write-ahead log is a relatively low-level component of a filesystem, since it is the component that reads and writes directly from the disk. Therefore, we want to keep the details of checksum support contained to the write-ahead log, so that other filesystem components can continue to use such a log without modification.

Our target is the logging system that underlies RapidFSCQ, a verifiably crash-safe filesystem that extends FSCQ with optimizations similar to those of ext4. We describe one scheme for modifying the RapidFSCQ log and its recovery procedure to use data checksums, with minimal changes required for higher-level code. With checksum support, we can show that RapidFSCQ reaches I/O efficiency on par with ext4.

1.3 Outline

In chapter 2, I discuss the work done so far in formally verified systems and modeling hash function behavior. In the remaining chapters, I discuss my three main contributions:

- chapter 3: A formal model of hash function behavior that is both logically sound and easy to use when reasoning about program execution and crashes.
- chapter 4: A verified checksum-based logging design, integrated into the RapidFSCQ crash-safe filesystem.

- chapter 5: An evaluation of RapidFSCQ with checksumming, demonstrating I/O efficiency competitive with ext4 and end-to-end crash safety with an application.

Chapter 2

Related Work

The work described in this thesis will ultimately go towards RapidFSCQ, the first file system with a machine-checked proof of crash safety with state-of-the-art logging optimizations, including checksums, and a specification that precisely captures the behavior of both `fsync` and `fdatasync` POSIX system calls. The rest of this chapter relates prior work to our technique for modeling hash collisions with crashes, as well as RapidFSCQ as a whole.

2.1 Modeling hash collisions

The work done so far on modeling hash collisions has not been practical enough to apply to a logging system. The Dolev-Yao model [14] assumes that hash collisions cannot happen by treating hashes symbolically. However, this does not allow us to reason about actual executable code, where a hash value is a concrete sequence of bits that can be written to disk.

Work on formalizing cryptographic protocols has explored how to model collision-resistant hash functions. For example, the RF* verification-oriented programming language [5] maintains, as auxiliary state, a mutable global dictionary from hashes to their inputs. This dictionary includes only the inputs already used in the current program execution, where an error is signaled if a new hash request leads to a collision in this dictionary. We build on this idea of treating hash collisions as program non-termination and extend it to handle crashes by reasoning about subsets of hash histories.

2.2 File-system verification

There has been significant progress on machine-checked proofs of file-system correctness. The two most recent results are FSCQ [9] and Cogent [3]. FSCQ verifies an entire file system, albeit with a synchronous specification that does not allow for deferred writes. Cogent generates highly efficient executable code for a file system, and supports deferred writes, but lacks a specification and proof for the entire file system. The RapidFSCQ prototype builds on top of FSCQ, and as a result, suffers from CPU overheads due to extracting code to Haskell. RapidFSCQ could benefit from using the DSL approach from Cogent to generate more efficient code and reduce the size of the trusted computing base.

Neither FSCQ, Cogent, nor other prior work on file-system verification [25, 17, 6, 26, 4, 38, 20, 18, 7, 15, 16, 26] use RapidFSCQ's technique for formally modeling hash collisions.

Efforts to find bugs in file-system code have been successful in reducing the number of bugs in real-world file systems and other storage systems [42, 43, 41, 24, 29]. However, these approaches cannot guarantee the absence of bugs, especially during recovery.

FSCQ's Crash Hoare Logic (CHL) is particularly important to this work. CHL is a variation of Hoare logic [21], a formal method commonly used to specify and verify programs. Hoare logic allows us to specify a program's correctness in terms of pre- and post-conditions. To prove a program's correctness, we show that if some precondition P holds before entering the program, then a postcondition Q must hold after exiting the program. We can verify the postconditions of larger programs by chaining single operations together and showing that each postcondition fulfills the next operation's precondition. This method of chaining proofs together also lends itself well to proof automation, which is crucial for verifying large systems.

Hoare logic is a powerful system for expressing correct program behavior, but it is not sufficient to prove correctness of a program in the event of a disk crash. This is because Hoare only allows a program's correctness to be defined by its pre- and post-conditions, while a crash can stop program execution at any time.

FSCQ addressed this gap by extending traditional Hoare logic with crash-conditions, in addition to the conventional pre- and post-conditions, to produce CHL. In contrast to postconditions, crash-conditions must be proven to hold true after any single command in a program, since a crash could occur at any of these points. Crash-conditions are the link between a disk crash and the recovery procedure, allowing us to specify and prove the outcome of a sequence of program execution, crash, and recovery.

2.3 Application bugs

It is widely acknowledged that it is easy for application developers to make mistakes in ensuring crash safety for application state [32]. For instance, a change in the ext4 file system *implementation* changed the observable crash behavior of the file system, as far as the application could see. This led to many applications losing data after a crash [12, 19], due to a missing `fsync` call needed to ensure that the contents of a new file are flushed to disk [8]. The ext4 developers, however, maintained that the file system never promised to uphold the earlier behavior, so this was an application bug. Similar issues crop up with different file-system options, which often lead to different crash behavior [32]. In chapter 5, we demonstrate the first end-to-end verified application and file system.

Chapter 3

Modeling hash collisions

Formal verification of checksums requires a logically sound model of hash function behavior that allows us to prove that if we see two matching hashes, their inputs must be equal. In general, we cannot assume this is true, given the theoretically possible but practically unlikely event of a hash collision. On the other hand, if we chose to explicitly account for any hash collisions, we would have to reason about the internals of the hash function we chose. This would force us to take on a large overhead in proof work that is in all likelihood irrelevant to actual program execution.

Our solution is to define our execution semantics to account for the possibility of a hash collision. We describe our definition of hashing semantics, written in the Coq proof assistant [11] and CHL framework [9], in section 3.1. Essentially, whenever a hash collision is encountered, we exit normal program execution and enter an infinite loop.

Since our proofs reason only about program executions that terminate, we can thus directly infer from our execution semantics that there are no hash collisions. In section 3.2, we specify and prove these basic correctness guarantees.

In section 3.3, we describe how to build on these basic specifications to specify similar guarantees about hash collisions for larger programs. We also describe how to keep specifications lightweight, so that we can automate proofs for programs that don't need to make explicit guarantees about specific hash values.

Finally, in section 3.4, we provide abstractions, as well as matching implementations, that allow us to reason about the hashes of *lists* of values. This abstraction will serve as a useful primitive for building a certified checksum-based log, which must hash across a series of disk blocks.

3.1 Execution semantics

The basis of the CHL framework is a set of opcodes that describe primitive operations on the disk, such as reading and writing a single block. The execution semantics define what each of these opcodes means with respect to the disk state. We can define conditions for execution in this way. For example, to execute a *Read(a)* operation, the disk must have some value v at address a and the return value of the operation is then specified to be v.

To support our model of hash collisions, we define a new opcode, *Hash(k)*, conditional on the value k. Specifically, if k doesn't collide with any other input we've seen so far, *Hash(k)* returns the hash of k, according to some hash function h. Otherwise, the program enters an infinite loop and fails to terminate. We keep some extra state in a data structure called a *hashset* to track the hash inputs that we've seen throughout program execution.

Note that this definition of execution semantics is only an abstraction that is leveraged for proof work; the actual program extracted from the Coq implementation would continue executing whether or not there were any hash collisions. Similarly, the hashset state is only an abstraction used to facilitate proofs. Lookups of past hash inputs never actually happen during program execution and therefore add no memory or CPU overhead.

This also means that there is a chance that actual program execution could diverge from the execution defined by our formal semantics, i.e., the actual program would continue executing while the abstract program would loop forever on a hash collision. In this case, we could no longer guarantee the same specifications about the actual program execution after that point. Fortunately, the probability of this event is negligible as long as we use a collision-resistant hash function.

The hashset data structure is a map from hash values, or outputs of a hash function h, to keys, or inputs to h. Key-value pairs are only ever added to the hashset, never deleted. Keys are also never overwritten, unless by the same value at that key already in the hashset.

We keep a single hashset across an entire program execution. The semantics of a *Hash(k)* operation are defined according to this hashset, shown in Figure 3-1. We check if a *Hash(k)* operation is safe or not by looking up $h(k)$ in the hashset. If $h(k)$ is not in the hashset, then we've never hashed k or any other colliding key before. If $h(k)$ already points to k in the hashset, then we are hashing a value we've seen before. In either case, there is no hash collision, so it is safe to return the hash of k. We always update the hashset with the entry $(h(k), k)$ when returning from the *Hash* operation, and the disk remains unchanged. The crash semantics are relatively simple: the hashset remains unchanged after recovering from a crash.

$$\frac{hs[h(k)] = k \;\vee\; hs[h(k)] = \texttt{None}}{\big(m,\, hs,\, Hash(k);\, \texttt{return}\big) \rightarrow \big(m,\, hs[h(k) := k],\, \texttt{return}\; h(k)\big)}$$

Figure 3-1: Small-step semantics for the *Hash* opcode. m is the disk state, hs is the *hashset* state, and `return` is a continuation function that binds the return value of *Hash(k)* to the rest of the program.

3.2 Specification

The operational semantics described in section 3.1 are enough to specify and verify a simple Hoare specification for a *Hash(k)* operation, shown in Figure 3-2. The precondition is `True`, signifying that the *Hash* operation can be called at any time on any key. The postcondition has three parts: (1) the value returned is $h(k)$; (2) the key k is safe with respect to the initial hashset, i.e., k does not collide with any

SPEC	*Hash(k)*
PRE:hs_{PRE}	*True*
POST:hs_{POST}	$ret = h(k) \bigwedge$
	$\big(hs_{PRE}[h(k)] = \texttt{None} \ \vee \ hs_{PRE}[h(k)] = k\big) \bigwedge$
	$hs_{POST} = hs_{PRE}[h(k) := k]$
CRASH:hs_{CRASH}	$hs_{CRASH} = hs_{PRE}$

Figure 3-2: Specification for the *Hash* operation. The *hs* variables refer to the hashsets at different points in the program execution: before, after, or during a crash.

previous input; and (3) the final hashset is the initial one, updated with the pair $(h(k), k)$. Note that the condition of hash collision safety is in the postcondition, not the precondition. This is because the Hoare specification only applies to terminating programs, so our proof can assume that hashing k does not cause a collision and therefore it must have been safe according to the initial hashset. We can prove the post-condition using the small-step semantics defined in Figure 3-1. The crash-condition simply states that the hashset at the time of the crash is equal to the initial hashset, which is trivial to prove since a crash in a single-operation program can only occur before the operation returns.

```
def compare(k1, k2):
    h1 = Hash(k1)
    h2 = Hash(k2)
    if (h1 == h2):
        return True
    else:
        return False
```

(a) Pseudocode.

SPEC	compare(*k1*, *k2*)
PRE	*True*
POST	$ret = \texttt{True} \iff \mathrm{k1} = \mathrm{k2}$
CRASH	*True*

(b) Specification.

Figure 3-3: A simple program that compares two hashes.

Once we have the specification for the *Hash* operation, we can chain it together with other operations to produce larger programs. To demonstrate the injectivity of the *Hash* operation on single values, we present pseudocode for a simple program in Figure 3-3a. The program takes in two input values, calls *Hash* on each, and returns

`True` or `False` depending on whether the hash values are equal. Using the *Hash* specification, we can prove that the boolean returned by the program also represents whether the *input* values were equal. This is captured in the post-condition of the program specification, shown in Figure 3-3b.

We outline a sketch of the proof as follows. Let k_1 and k_2 be the two input values. After executing *Hash*(k_1), the initial hashset is updated with the pair $(h(k_1), k_1)$, according to postcondition (3) of *Hash*. We'll call this intermediate hashset hs. After executing *Hash*(k_2), we know that k_2 does not cause any hash collisions with respect to hs, according to postcondition (2) of *Hash*.

There are two cases to consider: either the hashes are equal and we return `True`, or the hashes are not equal and we return `False`. In the first case, we know that $hs[h(k_1)] = k_1$ by definition of hs. Since k_2 is safe with respect to hs, there are two possibilities, $h(k_2)$ is not in hs or $hs[h(k_2)] = k_2$. Since $h(k_1) = h(k_2)$ and $h(k_1)$ is in hs, it must be the latter. Then, by substitution, $k_1 = k_2$. In the second case, we can simply use the fact that h is a function to show that $h(k_1) \neq h(k_2)$ implies that $k_1 \neq k_2$. In both cases, we have shown that the hashes are equal if and only if the input values are equal.

3.3 Hash subsets

Although it is necessary to the injectivity proof to state the exact mutations to the hashset in the *Hash* operation's postcondition, for the typical program, we only need to remember facts about certain entries in the hashset. To allow us to state these facts concisely without having to reason about the entire history of hashset updates, we use the idea of *subsets*. One hashset is the subset of another if and only if the latter has every entry in the former.

This definition has a few advantages. First, by stating for all programs that the hashset in the post- and crash-conditions must be a superset of the initial hashset, we can effectively abstract away the specific updates to the hashset. This is especially useful for stating crash-conditions, which must describe the state at any point during the program execution. For example, in a program that uses five different `Hash` operations, there are six possible hashsets if the program crashes, one for the initial hashset and one each after every `Hash` operation. It is undesirable to have to list out each of these possible hashsets in the crash-condition. Instead, we can simply state that the hashset at the time of the crash is some superset of the original, and selectively state propositions about the keys that we want to remember.

One subtlety is that even though the hashset does not actually change during a crash, this weaker crash specification allows the program to add any keys not already in the hashset in between program crash and program recovery. Although this does not affect any keys that we explicity state propositions about at the time of the crash, since keys are never deleted or overwritten, one may think that this implicitly allows for false hash collisions in the future. However, as long as the execution semantics do not actually add any more keys to the hashset during a crash, we can safely assume that the program execution will not encounter a false hash collision.

Second, specifications stated in this way are easy to automate, given the transitivity of the hashset-subset property. For a program that calls some number of functions in sequence, each of which guarantees the subset property in its postcondition, we simply chain together the subset relationships until we can prove that the final hashset is indeed a supserset of the original. For example, if hs_0 is the initial hashset, hs_1 is the hashset after running program p_1, and hs_2 is the hashset after running program p_2, we can prove that hs_0 is a subset of hs_2 using $hs_0 \subseteq hs_1 \subseteq hs_2$.

Finally, propositions about specific keys in a hashset can easily be carried across the subset relationship. This is because any key that we've already seen with respect to a hashset will still be safe to hash with respect to any superset. This is critical for maintaining information about the checksum across crashes. In particular, we must preserve the list of inputs that we hashed to compute the on-disk checksum so that we can compare it to the on-disk log data after a crash.

3.4 Modeling checksums

Although the specifications we've defined so far are enough to prove injectivity of single values, we'd also like to prove injectivity for *lists* of values, since a checksum should be able to represent multiple log entries. To do this, we first formalize a standard method for computing a checksum that chains together a list of inputs into a single hash value. The value zero, again the size of one disk block, serves as the default initial key. The checksum of a list of i inputs is defined as:

$$checksum_i = \begin{cases} h(checksum_{i-1} \,||\, k_i), & \text{if } i > 0 \\ h(0), & \text{else} \end{cases}$$

Using this definition, we can define an inductive relationship in Coq called *hash_ list*, which relates a hash value, a list of hash inputs, and a hashset, shown in Figure 3-4. The inputs could be arbitrarily sized, but for our use case, we assume that they are each the size of one disk block.

```
Inductive hash_list : list -> hash -> hashset -> Proposition :=
  | HL_nil : forall checksum hs,
      hs[h(0)] = 0 ->
      hash_list [] h(0) hs
  | HL_cons : forall l checksum x hs,
      hash_list l hl hs ->
      hs[h(checksum || x)] = checksum || x ->
      hash_list (l ++ [x]) h(checksum || x) hs.
```

Figure 3-4: Inductive definition of the *hash_ list* relationship in Coq.

The *hash_ list* relationship closely follows the definition of $checksum_i$ given above, but also requires that all checksums computed are safe with respect to the given hashset. Specifically, if we look up any checksum in hs, we will get the corresponding input to the hash function. Similar to how the hashset was used in section 3.1 to prove injectivity for single values, the hashset is also necessary here to prove injectivity for multiple values.

We can use this relationship to prove injectivity of *hash_list* on lists of inputs, assuming the same hashset. In other words, if the checksums and hashsets are equal, then the lists of inputs must also be equal. First, we will prove a couple helper lemmas. The first states that for any *hash_list* relationship, $h(0)$ must point to 0 in the hashset. The second states that no other list besides the empty list has the default hash value as its checksum. This second lemma is important because it allows us to prove that we will not mistakenly lose entries, thinking that the length of the log is zero when it isn't.

Lemma 3.4.1 *Let hash_list hold for* (l, c, hs)*. Then,* $hs[h(0)] = 0$*.*

Proof By induction on the *hash_list* relation. ∎

Lemma 3.4.2 *Let* l *be a non-empty list, and assume that hash_list holds for* (l, c, hs)*. Then,* $c \neq h(0)$*.*

Proof By contradiction. Assume that $c = h(0)$. Since l is non-empty, in order for *hash_list* to hold for (l, c, hs), there must exist some previous checksum c' and element x in l such that $hs[h(0)] = c'||x$. But, by Lemma 3.4.1, $hs[h(0)] = 0$. Since 0 and x are both the size of one disk block, there cannot exist a c' such that $c'||x = 0$.[1] ∎

Now, we can define a lemma stating injectivity of the *hash_list* relationship, assuming the same hashset:

Lemma 3.4.3 *Let* l_1 *and* l_2 *be two lists of hash inputs. For some checksum* c *and hashset* hs*, assume that the hash_list relation holds for* (l_1, c, hs) *and* (l_2, c, hs)*. Then,* $l_1 = l_2$*.*

[1] A more standard way of computing the checksum is to prepend a single 1-bit to $c'||x$, which allows us to prove the same fact that there exists no c' such that $1||c'||x = 0$ without having to depend on word sizes. Still, it is most likely safe to depend on word sizes for correctness, since the practical probability of a hash function producing a zero checksum c' is very low.

Proof By induction on the *hash_ list* relation.

The base case is when l_1 is an empty list and $c = h(0)$. Then, by Lemma 3.4.2, the only possible value for l_2 is the empty list, so $l_1 = l_2$.

Next, we want to show that for any k_1 and k_2, if *hash_ list* holds for $(l_1 ++ [k_1], c, hs)$ and $(l_2 ++ [k_2], c, hs)$, then $l_1 ++ [k_1] = l_2 ++ [k_2]$.

Note that there must have been two previous checksums c_1 and c_2 that correspond to l_1 and l_2, respectively, such that $c = h(c_1||k_1)$ and $c = h(c_2||k_2)$. By definition of *hash_ list*, $hs[c] = c_1||k_1$ and $hs[c] = c_2||k_2$. Since c_1 and c_2 have the same number of bits, we can conclude that $c_1 = c_2$ and $k_1 = k_2$.

Since the checksums c_1 and c_2 are equal, then by induction, $l_1 = l_2$. ∎

```
def checksum(c, keys):
    for i in range(len(keys)):
        c = Hash(c || keys[i])
    return c
```

(a) Pseudocode.

SPEC	*checksum(c, l′)*
PRE:hs_{PRE}	*hash_ list* (l, c, hs_{PRE})
POST:hs_{POST}	*hash_ list* $(l ++ l', ret, hs_{POST})$
LOOP:hs_{LOOP}, i, c	*hash_ list* $(l ++ l'[:i], c, hs_{LOOP})$
CRASH:hs_{CRASH}	*True*

(b) Specification.

Figure 3-5: The *checksum* program. The loop invariant refers to the current hashset, loop variable i, and loop return variable c.

Finally, we demonstrate a program that computes the checksum of a list of inputs using the *Hash* operation, shown in Figure 3-5. The program takes in a starting checksum and list of inputs and repeatedly hashes the concatenated checksum and next input. The specification guarantees that if *hash_ list* holds for the given starting checksum and some list l, then *hash_ list* will hold for the final checksum returned by the function and the list l appended with the given list of inputs.

The *checksum* correctness proof requires some discussion of loop invariants in the context of hashing semantics. The for-loop body is generally parametrized on a loop variable, e.g., the index i into the list of hash inputs. For-loops are executed by repeatedly executing the body until the loop variable reaches some condition, e.g., the length of the list. Since hashing semantics record a single hashset across all of program execution, every loop program must additionally parametrize each iteration of the loop with a hashset. This allows us to prove the loop invariant that the current hashset is a superset of the initial hashset, which is necessary to prove the hashset subset postcondition.

Proving the *checksum* program's postcondition requires a more specific loop invariant on the hashset, defined formally in Figure 3-5b. In this case, the loop invariant is that the current checksum computed must match the list of inputs hashed so far, according to the *hash_list* relation. The precondition of *checksum* directly implies the loop invariant before entering the for-loop, while the loop invariant directly implies the postcondition. To prove that the loop invariant holds from one iteration of the loop to the next, we simply use the postcondition of the *Hash* operation to prove that *hash_list* holds for the next checksum and input.

With these building block programs proven, we are ready to design a fully developed logging system that guarantees data integrity using checksums, discussed in chapter 4.

Chapter 4

Logging with checksums

The FSCQ log's `commit` procedure follows a design similar to ext4's `data=journal` mode in order to ensure data integrity in the log. This can be seen in the FSCQ [9] pseudocode, adapted in Figure 4-1 to highlight the code that can be optimized using data checksums. Specifically, before applying the commit, the FSCQ log writes the log entries to disk, flushes all outstanding writes to disk, writes the commit metadata to the log header, and flushes again. By separating the log entries and commit metadata writes with a disk flush barrier, FSCQ ensures that no matter where the log update procedure may crash, if the recovery procedure tries to recover the commit, then the correct log data is already on disk.

```
def log_commit():
    logHeader = log_flush()
    if logHeader is None: return False
    disk_sync()
    disk_write(COMMITBLOCK, logHeader)
    disk_sync()
    log_apply()
    disk_sync()
    logHeader.len = 0
    disk_write(COMMITBLOCK, logHeader)
    disk_sync()
    inMemoryLog = {}
    return True
```

Figure 4-1: The FSCQ `commit` procedure. The highlighted `disk_sync` can be removed using checksumming.

Checksumming improves I/O efficiency by removing the write barrier between writing the log entries and the log header. In this chapter, we apply the formalizations developed in chapter 3 towards supporting log checksums in an existing verified logging system, the RapidFSCQ log. RapidFSCQ is a crash-safe filesystem that provides a precise specification for the POSIX `fsync` and `fdatasync` calls. The core of the filesystem is the RapidFSCQ log, a highly optimized write-ahead log.

In section 4.1, we give an overview of the RapidFSCQ interface and logging design. The RapidFSCQ log includes sophisticated optimizations such as group commit and deferred application of log entries, which further reduce the number of write barriers per transaction. Due to the complexity of the RapidFSCQ logging optimizations, it is necessary to keep the system as modular as possible. At a high level, the RapidFSCQ log is separated into several logical layers built on top of each other, each of which is responsible for certain optimizations.

Our goal is to incorporate checksums into the RapidFSCQ logging design with minimal modifications to other layers. In section 4.2, we describe the implementation modifications required. These include the modifications to the on-disk layout of the log, as well as the procedures to recover and append to the log.

Finally, we describe the specification changes required in each layer in section 4.3. The majority of these changes go towards accounting for the addition of a new invalid log state, in which the data in the log does not match the checksum in the header.

4.1 RapidFSCQ Overview

RapidFSCQ is an I/O-efficient filesystem with a precise specification for crash safety. All of RapidFSCQ, including its specifications, implementation, and proofs, are written in the Coq proof assistant. This has two benefits. First, it allows us to use the same environment to write all parts of RapidFSCQ. Second, it ensures that our proofs are checked by the Coq proof assistant, giving us strong confidence in their correctness. RapidFSCQ runs on Linux using the FUSE interface. Following FSCQ's approach, RapidFSCQ's implementation is extracted from Coq into Haskell code, and compiled into a user-space FUSE server. Figure 4-2 shows the lines of code for different components of RapidFSCQ, including the modifications required for checksumming. The proofs are complete, and guarantee that RapidFSCQ's implementation meets its specification. The total development effort took 4 people one year. In the rest of this section, we describe the RapidFSCQ specification and optimizations, not including checksumming.

Component	**Lines of code**
FSCQ and CHL infrastructure	19,220
Hashing semantics	460
General data structures	3,615
Buffer cache	1,267
Write-ahead log	10,190
Inodes and files	2,907
Directories	5,470
RapidFSCQ's top-level API	1,750
Total	44,879

Figure 4-2: Combined lines of code and proof for RapidFSCQ components

4.1.1 Specification

The POSIX standard is notoriously vague on what crash-safety guarantees file-system operations provide. A particular concern is the guarantees provided by `fsync` and `fdatasync`, which give applications fine-grained control over what data the file system flushes to persistent storage. Unfortunately, file systems provide imprecise promises on exactly what data is flushed, and, in fact, for Linux ext4 file system it depends on the options that an administrator specifies when mounting the file system [32]. Because of this lack of precision, applications such as databases and mail servers, which try hard to make sequences of file creates, writes, and renames crash-safe by inserting `fsync`s and `fdatasync`s in the sequence, may still lose data when the file system it is running on crashes at an inopportune time [45, 8].

```
tmpfile = "crashsafe.tmp"

def crash_safe_update(filename, data_blocks):
  f = open(tmpfile, "w")
  for block in data_blocks:
    f.write(block)
  f.close()

  fdatasync(tmpfile)
  rename(tmpfile, filename)
  fsync(dirname(filename))

def crash_safe_recover():
  unlink(tmpfile)
```

Figure 4-3: Pseudocode for an application library that updates the contents of a file in a crash-safe manner.

For example, Figure 4-3 shows how a prototypical application uses `fsync` and `fdatasync`, in combination with other file-system API calls, to update a file in a crash-safe manner. This pattern shows up in many real applications, such as a mail server, a text editor, a database, etc. Of course, prior research has shown that file systems provide different crash semantics [32, 8], so our example may be not crash-safe on some file systems, and crash-safe on others. Nonetheless, we will explain why a developer might expect it to be crash-safe; this code also happens to be crash-safe on a file system that satisfies the RapidFSCQ specification. Our example code assumes that the application never runs this function concurrently.

`crash_safe_update(f, data)` ensures that, after a crash, file `f` will have either its old contents or the new `data`; it will not have a mixture of old and new data, or partial new data, or any other intermediate state. To ensure this property, `crash_safe_update` first writes the new data into a temporary file. A file system might bypass the journal when writing the new data to the file's data blocks; this is a common optimization implemented by Linux ext4 among others, which we call *log bypass*. Many file systems also implement writeback caching, so the new data may not have been written to the file's data block yet.

Once `crash_safe_update` has finished writing data to the temporary file, it invokes `fdatasync` to force the file system to flush any buffered changes to the temporary file's data blocks from the writeback cache out to disk (and to issue a disk write barrier).

After `fdatasync` returns, `crash_safe_update` replaces the original file with the new temporary file using `rename`. Since the file system's `rename` is atomic with respect to crashes, and the temporary file's contents are already on disk, if the system crashes at this point, the application will observe either the original contents (of the old file) or the new contents (of the new file). Finally, `crash_safe_update` uses `fsync` to flush its change to the directory, so that upon return, an application can be sure that the new data will survive a crash.

If the system crashes while executing `crash_safe_update`, it must first execute the file system's recovery code (which may replay transactions that have been committed but not applied), followed by its own recovery code. In our example, the application-specific recovery code `crash_safe_recover` simply deletes the temporary file if one exists. This is sufficient for our example, since if the temporary file exists, we must have crashed in the middle of `crash_safe_update`, and thus the original file still has its old contents.

RapidFSCQ provides a precise specification for the `fsync` and `fdatasync` system calls, allowing an application such as `crash_safe_update` to prove its own correctness. Briefly, the specification guarantees that in the event of a crash, the disk after recovery will reflect a complete and in-order prefix of all metadata updates up until the crash.

This specification provides a clear contract between applications and a file system. Ensuring metadata ordering helps developers reason about the possible states of the directory structure after a crash: If some operation survives a crash, then all preceding operations must have also survived. For instance, in the `crash_safe_update` function from Figure 4-3, the developer knows that all directory changes have been flushed to disk once `fsync(dirname(filename))` returns. Notably, this includes any possible pending changes to parent directories as well: for instance, if the application had just created the parent directory prior to calling `crash_safe_update`.

4.1.2 Implementation

The RapidFSCQ specification strikes a reasonable balance between ease of use for application programmers and allowing file systems to implement optimizations that provide high I/O performance. RapidFSCQ supports three optimizations besides log checksumming, which is described in the remainder of this chapter:

1. Group commit: Transactions are batched when appended to the on-disk log, reducing the number of disk flushes needed to persist each transaction.

2. Deferred writes: The log waits until it is full to apply all the transactions in the log to the disk at once, rather than applying each transaction individually with disk flushes in between.

3. Log bypass: Writes can bypass the log and go directly to disk without being committed as part of a transaction. Bypassed writes are handled by the RapidFSCQ log abstraction, but are not written to the on-disk log.

The key idea behind verifying a system as complex as the RapidFSCQ log is one familiar from building the unverified equivalent: modularity. The RapidFSCQ log is divided into four logical layers: LogAPI, GroupCommit, Applier, and DiskLog, shown in Figure 4-4. Here, we briefly describe the functionality of each layer and focus on DiskLog, the layer for which we'll provide checksum support.

LogAPI, the uppermost layer, exposes an interface with a single active transaction, and allows higher-level code (i.e., the file system) to read and write disk blocks. Writing blocks builds up an in-memory transaction, which is passed to the GroupCommit layer once the higher-level code invokes `commit`. LogAPI exposes the size of the transaction in its specification, and guarantees that transactions below a certain size will be able to commit. This is important for proving that some system calls, such as `unlink`, never fail as a result of running out of log space.

GroupCommit accepts committed transactions from LogAPI and implements group commit by buffering them in memory. GroupCommit also exposes a `flush` function, which flushes the in-memory transactions to the on-disk log. This allows the file

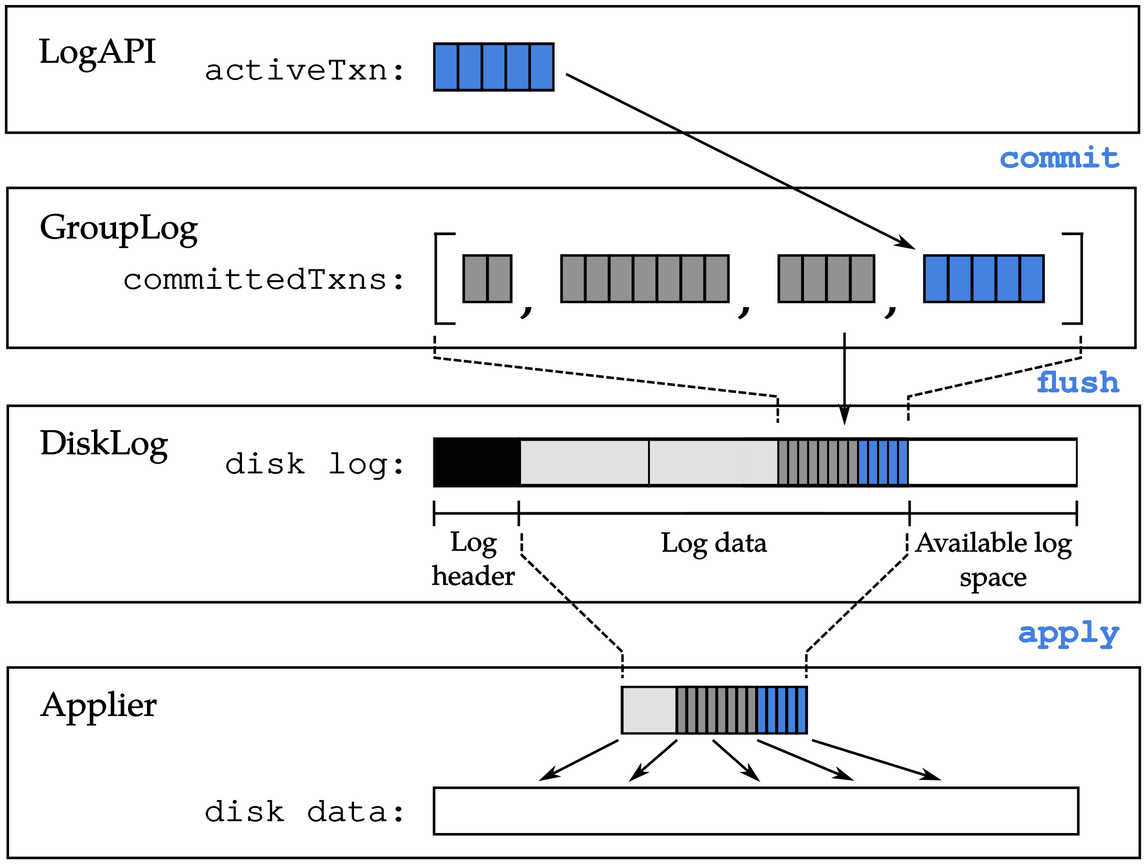

Figure 4-4: The RapidFSCQ log.

system to implement the `fsync` system call by flushing all metadata changes from GroupCommit to disk. GroupCommit's specification also allows it to flush transactions to disk on its own at any time. GroupCommit can choose to merge all buffered transactions into a single transaction, or it can flush them one at a time to disk if the single transaction is too large.

Applier manages the data part of the disk (i.e., everything but the log) by applying the log entries to the disk and truncating when the log fills up. While there is still room on the disk for more transactions to be written, Applier buffers unapplied writes in memory. By deferring the application of log entries, Applier is able to absorb repeated writes to the same address in multiple on-disk transactions.

DiskLog implements the on-disk log, without checksumming. It provides only two functions: `append` and `truncate`, pseudocode shown in Figure 4-5. `append` durably appends a given transaction to the on-disk log, using a design similar to FSCQ's `commit` procedure in Figure 4-1. `truncate` returns the log to length zero, allowing Applier to remove applied entries. DiskLog also exposes the size of the on-disk log to guarantee to Applier that transactions of a certain size will fit and thus be able to commit.

Recovery is relatively simple in RapidFSCQ. LogAPI exposes a read-only recovery procedure that rebuilds all layers' in-memory state, shown in Figure 4-5. Since DiskLog's `append` function uses a disk-write barrier to order the log entries before the log header, it is safe for the recovery procedure to read out all on-disk log entries without checking for data corruption. Thus, the procedure simply recovers all layers' in-memory state by reading out the log entries on disk.

The on-disk log abstracted by the DiskLog layer consists of three regions: the log header, descriptor, and data, shown in Figure 4-5. The log header stores the length of the list, which we use to determine how many valid entries there are. Each entry in the log consists of a disk block address and a value to update the corresponding block. The disk block addresses are stored in the descriptor region, with many packed into a single disk block, while the values are stored in the data region in the same order.

The interface exposed by each layer of the logging system consists of the methods allowed and the layer's possible states. The specification for each method is stated in terms of the layer's possible states. DiskLog exposes three possible states to Applier: *Synced*, *Truncating*, and *Extending*. These states also take a list of log entries as an argument, representing the current entries on disk, but we will only include this argument when necessary for clarity. Each layer builds its own state definitions out of the states of the layer below. For example, Applier has a state called *Applying* that is defined as either DiskLog's *Synced* or *Truncating* states; either Applier is applying the log updates to disk, or it has finished and is cleaning up the applied entries.

Synced is the stable state, in which there are no outstanding writes to disk, and the number of entries in the on-disk log match the length in the header. *Truncating* is the state when we've just written the length of the log to be zero, but haven't yet flushed the disk. *Extending* is the state when we've just written the new length of

```
# Applier's collapsed map of unapplied updates.
allFlushedTxns = {}

# Called by Applier layer after applying log to disk.
def log_truncate(txn):
    logHeader = disk_read(COMMITBLOCK)
    logHeader.len = 0
    disk_write(COMMITBLOCK, logHeader)
    disk_barrier_wait()

# Called by Applier layer, which must guarantee that
# there's enough space.
def log_append(txn):
    logHeader = disk_read(COMMITBLOCK)
    i = logHeader.len
    for (a, v) in txn.iteritems():
        disk_write(LOGSTART + i, (a, v))
        allFlushedTxns[a] = v
        i = i + 1
    disk_barrier_wait()
    logHeader.len = logHeader.len + len(txn)
    disk_write(COMMITBLOCK, logHeader)
    disk_barrier_wait()

# Recovers Applier's in-memory state.
def log_recover():
    logHeader = disk_read(COMMIT_BLOCK)
    for i in range(0, logHeader.len):
        (a, v) = disk_read(LOGSTART + i)
        allFlushedTxns[a] = v
```

Figure 4-5: DiskLog pseudocode, without checksumming.

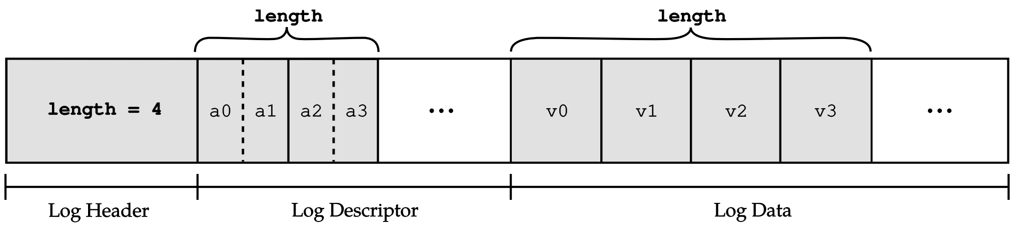

Figure 4-6: DiskLog layout, without checksumming.

the log, but haven't yet flushed the disk. Note that in the *Extending* state, the new log data has already been flushed completely to disk, so if the new header makes it to disk, it will be consistent with the log data.

The most interesting case of state transition is during the `append` call, whose CHL specification is shown in Figure 4-7. In this case, we could crash before we write the new length of the header, in which case we may lose the new data that we wrote to disk, but we'll crash in a stable state. Otherwise, we may crash during the write of the new length to the header, the *Extending* state. Then, we may restart after the crash with either the old length or the new length in the header.

SPEC	`append`(txn)
PRE	*Synced* l
POST	*Synced* $(l \mathbin{++} txn)$
CRASH	*Synced* $l \vee$ *Extending* l txn

Figure 4-7: CHL specification for DiskLog `append`. l is the list of log entries currently on disk, while txn is the transaction we want to append.

4.2 Checksummed log implementation

To support checksumming in the RapidFSCQ log, we make two modifications to the on-disk log layout, highlighted in Figure 4-8. First, we add a `checksum` field to the header. Second, we add a `previous_length` field, which represents the length of the log immediately before the most recent `append` call. `previous_length` is necessary for recovery, so that we do not lose the entire log of transactions when only the last transaction appended is corrupted by the crash. With these additional fields, the header can still fit in a single disk block, so we assume that writes to multiple fields of the header are atomic.

Next, we describe the modifications required for each of the two existing DiskLog methods, `truncate` and `append`, as well as a new `recover` method implemented at the DiskLog level. The pseudocode for each is shown in Figure 4-9.

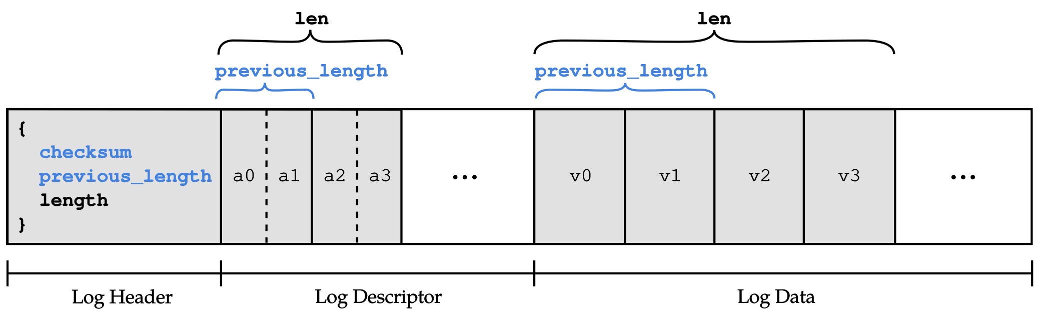

Figure 4-8: DiskLog layout, with checksumming.

`truncate` without checksumming already worked by writing the log header to return the log length to zero. Checksumming requires only an additional write to the log header to update the checksum to $h(0)$, the default initial hash value.

`append` is more complex. First, we remove the write barrier between writing the log entries to disk and the log header. Next, we use the *checksum* method defined in section 3.4 to hash the current checksum stored in the header with all of the block values for the new transaction.[1] We write this new checksum to the log header. Finally, we shift out the current length to `previous_length` and replace it with the new length, including the appended transaction.

The RapidFSCQ recovery procedure requires the most modification. Previously, the log could never crash to an inconsistent state and therefore DiskLog did not even include a `recover` operation, as the disk was read-only during recovery. With checksumming, the log may be corrupted and no longer match the on-disk checksum after restarting from a crash. We introduce a DiskLog `recover` method that modifies the disk to return the log to a consistent state (i.e., *Synced*). Upper logging layers can call DiskLog's `recover` function and then continue as before to recover any in-memory state.

First, `recover` compares checksums to determine if the log is corrupt or not. The procedure reads the `length` from the header and reads that many log entries from disk. It computes their checksum using the *checksum* method and compares it against

[1]For simplicity, we actually store two separate checksums, one each for the descriptor and data regions.

```
allFlushedTxns = {}

def log_truncate(txn):
    logHeader = disk_read(COMMITBLOCK)
    logHeader.checksum = hash(0)
    logHeader.previous_len = logHeader.len
    logHeader.len = 0
    disk_write(COMMITBLOCK, logHeader)
    disk_barrier_wait()

def log_append(txn):
    logHeader = disk_read(COMMITBLOCK)
    i = logHeader.len
    for (a, v) in txn.iteritems():
        disk_write(LOGSTART + i, (a, v))
        allFlushedTxns[a] = v
        logHeader.checksum =
          hash(logHeader.checksum || a || v)
        i = i + 1
    logHeader.previous_len = logHeader.len
    logHeader.len = logHeader.len + len(txn)
    disk_write(COMMITBLOCK, logHeader)
    disk_barrier_wait()

def log_recover():
    logHeader = disk_read(COMMIT_BLOCK)
    checksum = hash(0)
    for i in range(0, logHeader.len):
        (a, v) = disk_read(LOGSTART + i)
        checksum = hash(checksum || a || v)
    if checksum != logHeader.checksum:
        checksum = hash(0)
        for i in range(0, logHeader.previous_len):
            (a, v) = disk_read(LOGSTART + i)
            checksum = hash(checksum || a || v)
        logHeader.checksum = checksum
        logHeader.len = logHeader.previous_len
        disk_write(COMMITBLOCK, logHeader)
        disk_barrier_wait()
    for i in range(0, logHeader.len):
        (a, v) = disk_read(LOGSTART + i)
        allFlushedTxns[a] = v
```

Figure 4-9: DiskLog pseudocode, with checksumming.

the `checksum` value stored in the on-disk header. If the two checksums are different, `recover` must recover to a previously valid state.

`recover` uses the `previous_length` value stored on disk to recover to a previous log state. It reads out the first `previous_length` log entries on disk and computes their checksum. It writes this checksum value and `previous_length` as the new value for `length` in the header. Finally, it uses a write barrier to bring the log back into a *Synced* state. At this point, upper layers may continue the recovery process as if without checksums.

4.3 Checksummed log specification

With the addition of checksum support in the low-level DiskLog, we'd like to promise the same guarantees and interface for the top-level LogAPI. This involves modifying each logging layer's specification to account for checksumming and reproving the new specifications. In this section, we show that we can keep most of the modifications contained to the DiskLog layer.

There are two primary modifications necessary to the DiskLog specification. First, we add propositions to each of the possible DiskLog states guaranteeing that the checksum in the header matches the log entries on disk. Most of these propositions are *hash_list* relations between the checksum in the header, the list of entries on disk, and a given hashset. The hashset is passed into the DiskLog state representation function as a part of program specification. For example, *Synced* previously required that there was some list l of log entries that was completely synced on disk, whose length matched the value of `length` in the log header. With the addition of checksums, it now also requires that *hash_list* holds for (l, c, hs), where c is the value of the `checksum` field in the log header and hs is the current hashset.

Second, we define two new DiskLog states. Most of the possible states that a checksummed DiskLog could be in are already covered by the states introduced in section 4.1. However, since recovery in the non-checksummed DiskLog was a read-only operation, there are checksummed DiskLog states possible during recovery that

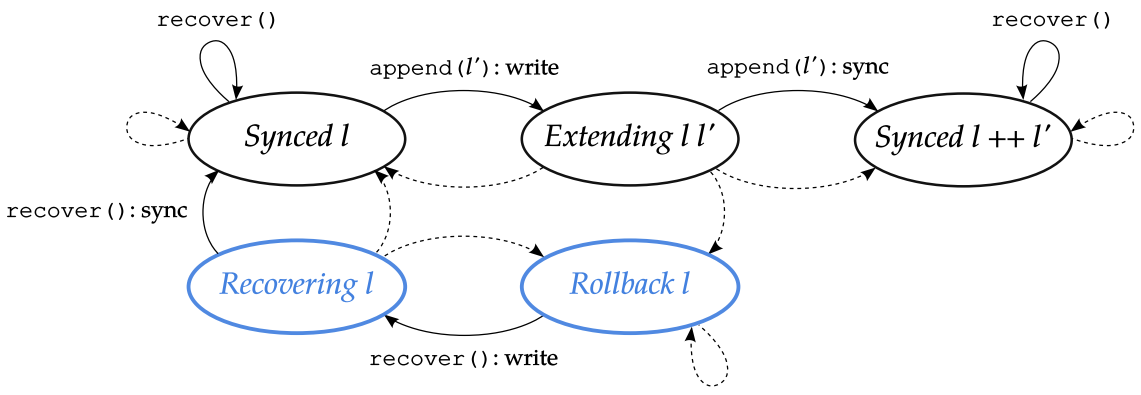

Figure 4-10: State diagram for DiskLog **append** and **recover** procedures. The dotted edges represent a crash and restart. Note that all states crash to either *Synced* or *Rollback*, the precondition for **recover**. Also, any path made up of **recover** and crash edges can only end in a single *Synced* state, demonstrating idempotency.

have no equivalent non-checksummed states. These can appear when recovering from a crash during a **append** operation, as demonstrated by the state machine in Figure 4-10.

The first of these states is *Rollback*, which like *Synced*, represents a disk with no outstanding writes. Unlike *Synced*, however, *Rollback* states that the on-disk log entries do *not* match the checksum in the header. DiskLog can restart the disk in this state if a crash occurs during an **append** operation after writing the log data and header, but before syncing the disk. In this case, the new log header may make it to disk after the crash without the new log data. *Synced* and *Rollback* are both parametrized by a list of log entries, which both promise is on disk. However, whereas the log header matches this list in the *Synced* state, it does not in *Rollback*. The *Rollback* list is the most recent log state when the header did match the log data, which the DiskLog **recover** method will eventually recover to.

It is surprisingly tricky to correctly define the fact that the on-disk checksum does not match the on-disk log entries in the *Rollback* state. The most obvious way is to directly state that the on-disk checksum is not equal to the checksum of the on-disk log data, but this turns out to be impossible to prove according to our hashing execution semantics. Even though we have enough information to say what the checksum of the on-disk log data *would* be, that information may not yet exist in the current hashset, since there is no guarantee that the data that the disk restarts with is data that we've previously passed through the *Hash* operation.

This particular definition doesn't work because of our inability to guess before recovery has started whether the data on disk would collide with some previously hashed value; we cannot know that there was no collision until *after* the recovery procedure and all relevant *Hash* operations have returned. Thus, the only way we can define the checksum mismatch is by reasoning directly about the input values themselves. In other words, we must explicitly state that the on-disk blocks are different from the transaction we should have appended to the log, i.e., the original argument to the `append` call made right before the crash. Only then can we guarantee that the checksum we compute from the log data will not match the checksum on disk.

The second additional state is *Recovering*, which is exactly the same as *Rollback*, except that the log header block has an outstanding write. This is the write done by `recover` to return the log to a consistent state.

Outside of these changes, the checksummed DiskLog specification differs from that of RapidFSCQ DiskLog only after a crash, since that is when data corruption may occur. Fortunately, this means that the proofs for non-recovery operations require little to no modification after the addition of checksums. The extra *hash_list* propositions can be easily proven using the postcondition of the *checksum* method.

SPEC `recover()`
PRE:hs_{PRE} $\textit{Synced}\ l\ hs_{PRE} \vee \textit{Rollback}\ l\ hs_{PRE}$
POST:hs_{POST} $\textit{Synced}\ l\ hs_{POST}$
CRASH:hs_{CRASH} $\textit{Synced}\ l\ hs_{CRASH} \vee$
$\quad\textit{Rollback}\ l\ hs_{CRASH} \vee$
$\quad\textit{Recovering}\ l\ hs_{CRASH}$

Figure 4-11: CHL specification for DiskLog `recover`, with checksumming.

The `recover` operation, on the other hand, is entirely new. Its specification is given in Figure 4-11. The precondition says that recovery can be run only on a log l that's in a consistent state, or else a log that, based on the mismatching on-disk values, we know should be rolled back to a previous log *l*. The postcondition guarantees that whichever state we start in, the log will be in a consistent state, with matching checksum, and entries *l* on disk. The crash-condition says that we either crash at the beginning of the program, or else we crash while in the *Recovering* state. This represents the period when the recovery procedure tries to restore a *Rollback* state to the *Synced* state.

The recovery specification guarantees a final, precise log state. For this to be useful, we need to prove that all other operations crash to a state that satisfies the `recover` precondition, i.e., either *Synced* or *Rollback*. The interesting cases are the crashes during `append` and `recover`, represented by the dotted edges in Figure 4-10. The CHL specifications given in Figure 4-7 and Figure 4-11 are both for the state of the disk immediately *before* the crash, the source of the edges in the state diagram. We must analyze each case to prove that the state of the disk *after* the crash, the destination of the edges, satisfies the `recover` precondition.

According to the specification in Figure 4-7, `append(`*txn*`)` can crash in the *(Synced l)* or *(Extending l txn)* states, where *l* is the list of log entries already synced to disk and *txn* is the log entries we want to append. Clearly, the *Synced l* case satisfies the `recover` precondition.

Recall that the *(Extending l txn)* state includes outstanding writes to disk, including the log entries in *txn* and the matching new log header. There are three possible cases for what combination of these writes will make it to disk after the crash:

1. The new log header does not make it to disk. Since the log entries previously on disk were not affected, the old log header is still consistent with the rest of the log, so we are in *(Synced l)*.

2. The new log header makes it to disk, but not all of the *txn* data. The data on disk is not equal to the data in *txn*, so we are in *(Rollback l)*.

3. The new log header and *txn* data both make it to disk. The checksum is consistent with the data on disk, so we are in *(Synced (l ++ txn))*.

The nontrivial case for a crash during `recover`, specified in Figure 4-11, is the *(Recovering l)* case. Recall that this is the *(Rollback l)* case, but with an outstanding write to the log header that matches l, the previous log entries on disk. There are two cases for the on-disk log state after the crash:

1. The write does not make it to disk. The original header, whose checksum did not match the log data, is still on disk, so we are in *(Rollback l)*.

2. The write does make it to disk. The log header now includes the length of l as the `length` field and the correct checksum, since we computed it by reading l out from disk, so we are in *(Synced l)*.

We can also use this case analysis to prove idempotency of the recovery procedure. Specifically, if we crash during an operation and then crash any number of times during the recovery procedure, we want to show that we will always recover to the same log state. This is easy to see by tracing the cycle between *Rollback* and *Recovery* formed by the `recover` and crash transitions in the state diagram in Figure 4-10.

Finally, we examine the modifications necessary to the specifications of the higher-level logging layers to account for the changes in DiskLog. The original DiskLog states, those that do not appear during DiskLog's recovery procedure, abstract away the details of checksumming and remain unchanged besides the addition of a hashset argument. Since the specifications and proofs for the upper layers' non-recovery operations are built out of these DiskLog states, there are no changes necessary for these operations above the DiskLog layer.

For recovery, since the upper layers now call the new DiskLog `recover` method before executing their own recovery procedures, there are two modifications necessary. The first is the addition of a state equivalent to DiskLog's *Rollback* state in each of the upper layers' preconditions for recovery, to match DiskLog's `recover` specification. This is relatively simple, since we just add a *Rollback* state in each of the upper layers and define it to be the *Rollback* state of the layer below. For each layer's recovery method, we also add *Rollback* as a possible branch in the precondition.

The other modification is the crash-condition of the upper layers' recovery methods. Previously, since recovery was read-only, the crash-condition was simply the same as the precondition for layers above DiskLog. With the addition of checksumming, we define a new *Recovering* state in each of the upper layers that exactly matches the full DiskLog `recover` crash-condition, defined formally in Figure 4-11.

Most of the proof work is already taken care of within DiskLog. All upper layers' modified specifications are simple to prove using DiskLog's modified specification.

Chapter 5

Evaluation

As mentioned in chapter 4, RapidFSCQ is a sophisticated filesystem that includes several optimizations besides log checksumming, such as deferred writes, group commit. The system also supports file data writes that bypass the filesystem journal, which we call *log bypass*. This chapter answers the following questions about RapidFSCQ as a whole, with log checksum support.

- Do RapidFSCQ's theorems prevent bugs that previous systems could not?
- Does RapidFSCQ, with checksumming, indeed achieve good I/O performance?
- Are RapidFSCQ's high-level specifications correct and useful? That is, can applications use RapidFSCQ's specifications to prove their own correctness? One goal of having precise specifications is that applications can prove their own correctness.

5.1 What bugs are prevented?

We answer the question of whether RapidFSCQ's theorems prevent real bugs by presenting a case study of different kinds of bugs that have been discovered in the Linux ext4 file system. For each, we argue for whether the state-of-the-art prior work (FSCQ) or RapidFSCQ prevents them.

Bug category and example	**Possible in FSCQ?**	**Prevented by FSCQ?**	**Possible in RapidFSCQ?**	**Prevented by RapidFSCQ?**
Logging logic; write/barrier ordering [27, 36, 13]	Some (no checksumming)	Yes	Yes	Yes
Misuse of logging API [37, 34]	Some (no log bypass)	Yes	Yes	Yes
Bugs in recovery protocol [23, 30]	Yes	Yes	Yes	Yes
Improper corner-case handling [40]	Yes	Yes	Yes	Yes
Low-level bugs [31, 27, 39]	Some (memory safe)	Yes	Some (memory safe)	Yes
Concurrency [28, 35]	No	—	No	—

Figure 5-1: Representative bugs found in Linux ext4 and whether RapidFSCQ's specifications preclude them.

5.1.1 ext4 bugs case study.

We looked through the git logs for the Linux ext4 file system starting from 2013, and categorized the bugs fixed in those commits. Figure 5-1 shows the resulting categories along with representative bugs from each category. For instance, this table includes the bug that was mentioned in the introduction, where ext4 would disclose previously deleted file data after a crash [27]. The figure also shows whether each bug category could have occurred in the implementations of either FSCQ or RapidFSCQ; for instance, some bugs arise due to concurrent execution of system calls, which is impossible in both FSCQ and RapidFSCQ by design (i.e., they are not sophisticated enough to have such a bug). The figure also shows whether the theorems of FSCQ and RapidFSCQ prevent those bugs.

We make four conclusions from this case study. First, RapidFSCQ is sophisticated enough that its implementation could have had many of the bugs that were fixed in ext4, making verification important. Second, the state-of-the-art verified file system, FSCQ, was not sophisticated enough to even have many of these bugs, especially the trickier cases that included dealing log checksums. Third, RapidFSCQ's theorems preclude every bug category that was possible in its implementation. This suggests that RapidFSCQ's theorems are effective at preventing real bugs. Finally, the one category where RapidFSCQ is not sophisticated enough to have bugs is concurrency: RapidFSCQ is a single-threaded file system. Verifying a concurrent file system is an open problem and remains future work.

	mailbench		largefile	
ext4 bypass	32.2;	4.2	1.0;	1.0
ext4 logged	49.5;	7.2	4.1;	1.0
RapidFSCQ bypass	68.8;	15.5	1.2;	1.0
RapidFSCQ logged	34.1;	4.6	4.0;	1.0
FSCQ	86.3;	40.1	5.2;	4.1

Figure 5-2: I/O performance of RapidFSCQ compared to FSCQ and to Linux ext4. Each cell reports the number of writes and barriers, respectively, per application-level operation.

5.2 I/O performance

A primary goal of RapidFSCQ was to achieve good I/O performance by supporting deferred writes, group commit, and as discussed in this thesis, log checksums. To validate that RapidFSCQ's design indeed achieves its I/O efficiency goal, we run two benchmarks to stress both data and metadata aspects of RapidFSCQ: mailbench [10] and a modified LFS largefile benchmark [33]. mailbench performs many metadata operations by manipulating small files, and our modified largefile performs many data writes to an existing large file followed by `fdatasync` calls. To place RapidFSCQ's I/O efficiency in context, we also run the same benchmarks on the FSCQ file system, and on the Linux ext4 file system. We run both ext4 and RapidFSCQ in two different modes: one which implements log bypass for file data writes (using mount option `data=ordered` in ext4), and one which file data writes are logged (using mount option `data=journal` in ext4). In ext4, it is possible to enable log checksumming in the logged configuration (using mount option `journal_async_commit`), but not in the bypass configuration, due to a design issue [27].[1] We ran this experiment on a machine with a Samsung MZVKV512HAJH-000L1 NVMe SSD and an Intel Core i7-6600U 2.6 GHz CPU.

Figure 5-2 shows the results for this experiment, reporting the number of disk writes and disk write barriers issued by each of the file systems per application-level

[1]Amusingly, the patch for the design issue prevents the user from mounting a file system with `journal_async_commit,data=ordered` options, but if the user omits `data=ordered`, the check is bypassed but the kernel defaults to `data=ordered` anyway.

operation (delivering a mail message in mailbench and writing a 4KB block in a large file for largefile). We used the Linux `blktrace` support to trace the disk operations performed by each file system. We draw several conclusions.

First, RapidFSCQ indeed achieves good I/O efficiency, issuing a similar number of write barriers and disk writes to ext4. For largefile, RapidFSCQ and ext4 have the same number of write barriers (1.0 per application-level block write) in both configurations. Both log-bypass configurations write each block to disk just once, whereas the logged configurations write 4 disk blocks for each application-level block write. This is because every application-level write turns into an on-disk transaction, which later must be applied separately. RapidFSCQ writes an average of 1.2 disk blocks per application-level write because of the startup phase of largefile, where it grows the file one block at a time. RapidFSCQ's GroupCommit flushes transactions to disk when it detects a write to a newly allocated file's data. Optimizing away this case is left to future work.

For mailbench, the number of disk barriers varies greatly due to application-level timing. Since mailbench is a multi-process application, the order of system calls seen by the file system can change the degree of batching. The low bound, based on back-of-the-envelope calculations, is 4 write barriers per message; both RapidFSCQ and ext4 come close to this bound. The variability is high in this experiment; adding even small sleep statements to mailbench produces different numbers of write barriers, ranging from 4 to 7.

Second, RapidFSCQ achieves far better I/O efficiency than the state-of-the-art verified FSCQ file system. This is due to RapidFSCQ's more sophisticated design which incorporates standard write-ahead logging optimizations, while FSCQ executes every system call synchronously, does not use checksums, and applies every transaction at commit time.

As mentioned in the introduction, one limitation of RapidFSCQ is that it is slower in terms of its CPU performance. For instance, RapidFSCQ in logged mode can deliver 19 messages per second when running mailbench, while Linux ext4 can deliver either 26 or 37 messages per second (in logged and bypass modes, respectively).

Profiling the RapidFSCQ user-space FUSE file server shows that this overhead is due to RapidFSCQ's reliance on Haskell to produce executable code. We hope to adopt ideas for better generation of certified assembly code in future work.

5.3 Are RapidFSCQ specs correct and useful?

To demonstrate that RapidFSCQ's specifications for its system calls are meaningful, we performed the following experiments.

5.3.1 fsstress.

We ran `fsstress` from the Linux Test Project to check if it finds any bugs in RapidFSCQ. When we first ran `fsstress`, it caused our FUSE file server to crash. However, after some investigation, we discovered that this was due to a bug in our Haskell FUSE bindings that sit between RapidFSCQ and the Linux FUSE interface. The bug was due to the developer thinking that some corner case could not be triggered, and calling the `error` function in Haskell to panic if that case ever executed. As it turns out, `fsstress` found a way to trigger that corner case. After fixing this bug, `fsstress` ran without problems and did not discover any bugs in RapidFSCQ's proven code.

5.3.2 Enumerating crash states.

We implemented the `crash_safe_update` program whose pseudocode was shown in Figure 4-3. Our specific implementation of the `crash_safe_update` program writes and syncs some data to a temporary file using `fdatasync`, then performs an atomic rename of the temporary file to a destination file using `fsync` on the directory. We ran the program on RapidFSCQ while monitoring all of the disk writes and barriers issued by RapidFSCQ. We then computed all possible subsets and re-orderings of RapidFSCQ's disk writes, subject to its barriers, to produce every possible state in which RapidFSCQ could have crashed. Finally, we re-mounted the resulting disk with RapidFSCQ and examined the file system state after RapidFSCQ performed its recovery. This experiment produced 182 possible disks after a crash, but only three distinct file system states after RapidFSCQ executed its recovery code: neither file existed, the temporary file existed with no contents, or the destination file existed with the written contents. All of these states are safe, since either the destination file didn't exist or it contained the correct data (the empty temporary file could be removed during recovery).

5.3.3 Certifying an application.

The above experiments suggest that RapidFSCQ specifications capture the right properties, because the implementation appears not to have bugs. However, to increase our confidence that the specifications themselves are correct (and not just the implementation), we wrote a formal specification for the `crash_safe_update` program, and proved its correctness based on the specification of RapidFSCQ.

Proving the correctness of `crash_safe_update` led us to discover several cases where the RapidFSCQ specification was too weak. For example, the `read` specification originally forgot to mention that the data returned by the system call is related to the contents of the file. Another example is the `fsync` system call, which forgot to promise a safety property required for log bypass. This also uncovered many cases where the specification was not as convenient to use as it could have been. None of these issues required changing the RapidFSCQ implementation, and we were able to re-prove the correctness of RapidFSCQ after fixing the specification.

Proving `crash_safe_update` also led us to discover a number of corner cases in `crash_safe_update` itself. For example, we discovered that `crash_safe_update` cannot perform a safe update on a file with the same file name as the temporary file that it uses. After fixing the specification to take into account these corner cases, we were able to prove the correctness of `crash_safe_update` when running on top of RapidFSCQ.

Chapter 6

Conclusion

Even well-studied and widely used filesystems like ext4 have a long history of bugs, some of which aren't discovered until after they have already caused disastrous data loss or disclosure. FSCQ was the first filesystem to formally verify the absence of such bugs. However, formally verified systems like FSCQ have a long way to go before they can be used in practice, mostly due to the difficulty of verifying optimizations.

This thesis presents the first formally verified logging system to use checksumming to improve performance. Although some progress has been made towards formally modeling hash function behavior, this is the first example of a practical application of a formally defined hash function to a system that supports disk I/O, the ability to crash and recover, and so on.

We focus the work in this thesis first towards a formal and logically sound definition of hashing execution semantics. Next, we demonstrate their practicality by building short programs that use these semantics to prove simple specifications. Finally, we use these simple hashing procedures to build checksums into a fully verified, highly optimized logging system. In our evaluation, we show that RapidFSCQ, a filesystem built on top of the logging system, achieves the desired goal of I/O efficiency similar to that of ext4, while formally guaranteeing crash safety properties that can be empirically checked.

Of course, there is still much to be done in the area of formally verified filesystems, as well as verified systems in general. Although the evaluation in this thesis shows that I/O efficiency is achievable through careful system design, practical performance is still out of reach. The Haskell executable extracted from the Coq implementation of RapidFSCQ is bottlenecked by CPU, not disk I/O. This, as well as the addition of the Haskell runtime to the trusted computing base, remains a problem in general Coq-based systems.

General system concurrency, something application developers often take for granted, is also notoriously difficult to formalize and verify. With the wrong specification, adding concurrency to a system can exponentially increase proof effort. Defining the correct specification for a concurrent filesystem, as well as multithreaded systems in general, remains an interesting open problem.

Although there is a long way ahead before formally verified systems can be widely adopted, recent developments in this field have laid the foundation for future work. In this thesis alone, we can see the progress being made towards specifying and proving common systems features that until now remained unverified and prone to error. We hope that the work described here will further the construction of other verified, practical computer systems.

Bibliography

[1] *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, October 2014.

[2] *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[3] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, Joseph Tuong, Gabriele Keller, Toby Murray, Gerwin Klein, and Gernot Heiser. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* [2], pages 175–188.

[4] Konstantine Arkoudas, Karen Zee, Viktor Kuncak, and Martin Rinard. Verifying a file system implementation. In *Proceedings of the 6th International Conference on Formal Engineering Methods*, Seattle, WA, November 2004.

[5] Gilles Barthe, Cédric Fournet, Benjamin Grégoire, Pierre-Yves Strub, Nikhil Swamy, and Santiago Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *Proceedings of the 41st ACM Symposium on Principles of Programming Languages (POPL)*, San Diego, CA, January 2014.

[6] William R. Bevier and Richard M. Cohen. An executable model of the Synergy file system. Technical Report 121, Computational Logic, Inc., October 1996.

[7] William R. Bevier, Richard M. Cohen, and Jeff Turner. A specification for the Synergy file system. Technical Report 120, Computational Logic, Inc., September 1995.

[8] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and checking file system crash-consistency models. In *Proceedings of the 21th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* [2], pages 83–98.

[9] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[10] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–17, Farmington, PA, November 2013.

[11] Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.5pl1*. INRIA, April 2016. http://coq.inria.fr/distrib/current/refman/.

[12] Jonathan Corbet. ext4 and data loss. http://lwn.net/Articles/322823/, March 2009.

[13] Lukas Czerner. [PATCH] ext4: Fix data corruption caused by unwritten and delayed extents. https://lwn.net/Articles/645722, April 2015.

[14] Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, Stanford, CA, USA, 1981.

[15] Gidon Ernst, Gerhard Schellhorn, Dominik Haneberg, Jorg Pfähler, and Wolfgang Reif. Verification of a virtual filesystem switch. In *Proceedings of the 5th Working Conference on Verified Software: Theories, Tools and Experiments*, Menlo Park, CA, May 2013.

[16] Miguel Alexandre Ferreira and Jose Nuno Oliveira. An integrated formal methods tool-chain and its application to verifying a file system model. In *Proceedings of the 12th Brazilian Symposium on Formal Methods*, August 2009.

[17] Leo Freitas, Jim Woodcock, and Andrew Butterfield. POSIX and the verification grand challenge: A roadmap. In *Proceedings of 13th IEEE International Conference on Engineering of Complex Computer Systems*, pages 153–162, March–April 2008.

[18] Philippa Gardner, Gian Ntzik, and Adam Wright. Local reasoning for the POSIX file system. In *Proceedings of the 23rd European Symposium on Programming*, pages 169–188, Grenoble, France, 2014.

[19] Bogdan Gribincea et al. Ext4 data loss. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781, January 2009.

[20] Wim H. Hesselink and M.I. Lali. Formalizing a hierarchical file system. In *Proceedings of the 14th BCS-FACS Refinement Workshop*, pages 67–85, December 2009.

[21] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[22] William A. Howard. The formulas-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980. Reprint of 1969 article.

[23] Ben Hutchings. [PATCH 3.2 027/115] jbd2: fix fs corruption possibility in jbd2_journal_destroy() on umount path. `https://lkml.org/lkml/2016/4/26/1230`, April 2016.

[24] Dave Jones. Trinity: A Linux system call fuzz tester, 2014. `http://codemonkey.org.uk/projects/trinity/`.

[25] Rajeev Joshi and Gerard J. Holzmann. A mini challenge: Build a verifiable filesystem. *Formal Aspects of Computing*, 19(2):269–272, June 2007.

[26] Eunsuk Kang and Daniel Jackson. Formal modeling and analysis of a Flash filesystem in Alloy. In *Proceedings of the 1st Int'l Conference of Abstract State Machines, B and Z*, pages 294–308, London, UK, September 2008.

[27] Jan Kara. [PATCH] ext4: Forbid journal_async_commit in data=ordered mode. `http://permalink.gmane.org/gmane.comp.file-systems.ext4/46977`, November 2014.

[28] Jan Kara. ext4: fix crashes in dioread_nolock mode. `http://permalink.gmane.org/gmane.linux.kernel.commits.head/575311`, February 2016.

[29] Eric Koskinen and Junfeng Yang. Reducing crash recoverability to reachability. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 97–108, St. Petersburg, FL, January 2016.

[30] Kamal Mostafa. [PATCH 3.13 075/103] jbd2: fix descriptor block size handling errors with journal_csum. `https://lkml.org/lkml/2014/9/30/747`, September 2014.

[31] Kamal Mostafa. ext4: fix null pointer dereference when journal restart fails. `https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=9d506594069355d1fb2de3f9104667312ff08ed3`, June 2016.

[32] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 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)* [1], pages 433–448.

[33] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*, pages 1–15, Pacific Grove, CA, October 1991.

[34] Eric Sandeen. [PATCH] ext4: fix unjournaled inode bitmap modification. http://permalink.gmane.org/gmane.comp.file-systems.ext4/35119, October 2012.

[35] Theodore Ts'o. ext4: fix race between truncate and __ext4_journalled_writepage(). https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/commit/?id=bdf96838aea6a265f2ae6cbcfb12a778c84a0b8e, June 2015.

[36] Theodore Ts'o. [PATCH] ext4, jbd2: add req_fua flag when recording an error flag. http://permalink.gmane.org/gmane.comp.file-systems.ext4/49323, July 2015.

[37] Theodore Ts'o. [PATCH] ext4: use private version of page_zero_new_buffers() for data=journal mode. https://lkml.org/lkml/2015/10/9/1, October 2015.

[38] Markus Wenzel. Some aspects of Unix file-system security, August 2014. http://isabelle.in.tum.de/library/HOL/HOL-Unix/Unix.html.

[39] Darrick J. Wong. jbd2: Fix endian mixing problems in the checksumming code. http://lists.openwall.net/linux-ext4/2013/07/17/1, July 2013.

[40] Darrick J. Wong. [PATCH] ext4: fix same-dir rename when inline data directory overflows. http://permalink.gmane.org/gmane.comp.file-systems.ext4/45594, August 2014.

[41] Junfeng Yang, Can Sar, Paul Twohey, Cristian Cadar, and Dawson Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the 27th IEEE Symposium on Security and Privacy*, pages 243–257, Oakland, CA, May 2006.

[42] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 273–287, San Francisco, CA, December 2004.

[43] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. eXplode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 131–146, Seattle, WA, November 2006.

[44] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla+ specifications. In *Correct Hardware Design and Verification Methods*, pages 54–66. Springer, 1999.

[45] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)* [1], pages 449–464.