HARDFS: Hardening HDFS with Selective and Lightweight Versioning

Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin, Madison [†] University of Chicago

Abstract

We harden the Hadoop Distributed File System (HDFS) against fail-silent (non fail-stop) behaviors that result from memory corruption and software bugs using a new approach: selective and lightweight versioning (SLEEVE). With this approach, actions performed by important subsystems of HDFS (e.g., namespace management) are checked by a second implementation of the subsystem that uses lightweight, approximate data structures. We show that HARDFS detects and recovers from a wide range of fail-silent behaviors caused by random bit flips, targeted corruptions, and real software bugs. In particular, HARDFS handles 90% of the fail-silent faults that result from random memory corruption and correctly detects and recovers from 100% of 78 targeted corruptions and 5 real-world bugs. Moreover, it recovers orders of magnitude faster than full reboot by using micro-recovery. The extra protection in HARDFS incurs minimal performance and space overheads.

1 Introduction

Large-scale distributed storage systems [26, 33, 42, 49] are becoming a dominant platform for a variety of applications and services. These complex "cloud" systems often run on clusters of thousands of unreliable commodity machines and must handle all kinds of failures, while preserving the integrity of user data and system metadata [18, 25, 26, 30, 38, 40]. Making these systems robust is challenging.

At the individual machine level, two common failure modes that these systems face are machine crashes and disk failures. To deal with these failures, there is a rich body of literature describing detection and recovery mechanisms such as journaling [34], RAID [27, 45], and redundant hardware [17]. With these advancements, fail-stop machine and disk failures are no longer considered a single point of failure in many of today's cloud systems.

Many cloud systems are able to handle fail-stop failures, but they do face new challenges. First, the systems run at large scale [30, 38, 40]; thus, failures that used to be rare (*e.g.*, memory corruption) become more frequent [12, 48]. Second, modern software is increasingly complex, and thus software bugs are becoming more common. If not handled properly, errors resulting from memory corruption and software bugs become a single point of failure in today's systems. Observations from real systems show that these failures can lead to transient, non-deterministic errors, and make the system exhibit *fail-silent* behaviors (*e.g.*, send corrupt messages) rather than crashing; these fail-silent errors can lead to data loss, unavailability, and prolonged debugging effort [11, 12].

To effectively handle fail-silent errors, we propose that distributed systems be built with *selective* and *lightweight* versioning (SLEEVE). The goal of SLEEVE is to detect silent faults in select subsystems of a target system and to do so in a lightweight manner (with little space and performance overhead). For example, a developer can pick some important functionality (*e.g.*, file-system namespace management) and protect that functionality from fail-silent behaviors by developing a second lightweight implementation of the functionality. This approach essentially transforms a target system into an efficient two-version form that can detect (and recover from) fail-silent behaviors.

Using the SLEEVE approach, we harden the Hadoop file system (HDFS) [49], which is similar in structure to Google's file system, GFS [33]. Although HDFS already contains some mechanisms for detecting and recovering from errors (*e.g.*, replication and checksums), bugs have been found in these mechanisms, and our experiments show that HDFS is still susceptible to memory corruptions. Thus, additional hardening to prevent data loss is useful. We harden three pieces of HDFS functionality: namespace management, replica management, and the read/write protocol, creating three robust systems, called HARDFS-N, HARDFS-R, and HARDFS-D, respectively.

We evaluate the effectiveness of HARDFS by injecting random bit flips, corrupting targeted fields of important data structures, and by reintroducing known bugs. Our experimental results show that while HDFS silently misbehaves in many cases, HARDFS effectively isolates faulty behavior so that it remains within a single node. In particular, HARDFS handles 90% of the fail-silent faults that result from random memory corruption and correctly detects and recovers from 100% of 78 targeted corruptions and 5 real-world bugs that we reintroduce in our codebase. Since errors do not propagate to persistent storage or other nodes, previously fail-silent errors are transformed into fail-stop errors, enabling the

use of standard recovery mechanisms such as failover, single-machine reboot, or execution of *fsck*. Furthermore, HARDFS detection can often pinpoint corrupt data structures, enabling micro-recovery that repairs small portions of corrupted state. HARDFS is able to micro-recover in seconds instead of rebooting over many hours.

This paper is structured as follows. First, we discuss the SLEEVE approach and its usefulness for hardening HDFS (§2). Next, we describe the HARDFS design (§3) and our implementation (§4). We then measure the effectiveness and performance of HARDFS (§5), discuss related work (§6), and conclude (§7).

2 Extended Motivation

Modern software systems such as HDFS must deal with memory corruption and software bugs that are becoming more common. Therefore, we address a failure model where in-memory data can contain wrong values due to memory corruption and software bugs. If not handled properly, these errors lead to fail-silent behaviors that are hard to detect and can cause severe problems like data loss and service unavailability. We assume that the system is not malicious and that persistent storage is trusted.

Figure 1 illustrates some problems caused by fail-silent behaviors. Figure 1a shows a normal correct behavior of HDFS; a client writes a file `F` and the HDFS namenode replicates `F`'s data block, `D`, to two datanodes (in 2-way replication). However, silent memory corruption such as a bit flip can take place (*e.g.*, metadata `F` flips to `G` in Figure 1b). In this case, the user will not be able to read the file in the future. Subtle software bugs in HDFS (§5.1.3) could also lead to silent data loss or corruption. For example, in Figure 1c, a bug in the namenode silently deletes `F`'s data blocks in a background task.

In this work, we attempt to address this question: How should distributed storage systems such as HDFS deal with fail-silent behaviors? Many approaches such as Byzantine fault tolerance (BFT) [43], N-version programming [14, 16], and the use of ECC memory have been proposed. However, existing approaches either incur high performance overhead, hardware cost, or engineering effort (§6). In this paper, we propose a new approach: selective and lightweight versioning (SLEEVE).

2.1 Hardening HDFS with SLEEVE

The goal of the SLEEVE approach is to selectively protect some part of the target system against fail-silent behaviors and to do so in a lightweight manner (with little space and performance overhead). Figure 1d illustrates HARDFS, an HDFS system that employs the SLEEVE approach. The code of the HDFS system (which we call the *main version*) implements the complete functionality of the system. A developer can pick some important

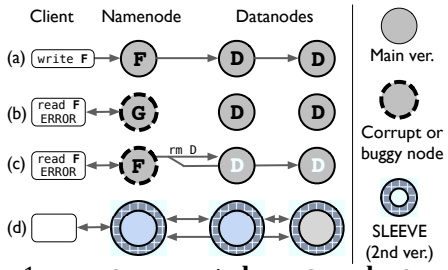

Figure 1: **HDFS, corrupted HDFS, and HARDFS.**

piece of functionality and create a "second version" of it, a variant of 2-version programming. This second, selective, and lightweight version models the state and logic of the main version. The model can detect misbehavior in the main version and trigger appropriate responses. We refer to systems that pair a complete main version with a modeled second version as *sleeved systems*.

Sleeved systems watch inputs and outputs of the main version (as illustrated in Figure 1d) to detect incorrect behaviors that deviate from the model. For example, memory corruption and software bugs in Figure 1b and 1c can easily be detected; HARDFS will catch the read `F` error and incorrect background data removal (`rm D`) as faulty behaviors. After detecting faulty behaviors, a sleeved system can perform an appropriate action, such as micro-recovery, to transform faulty states (*e.g.*, corrupt metadata in the main-version memory) into consistent states. Thus, a sleeved system isolates faulty behavior within a single node; faults are not propagated to persistent storage or other nodes.

We have three requirements for hardening HDFS which the SLEEVE approach satisfies: HARDFS should be effective at detecting and handling faults (§5.1), the additional protection should incur minimal performance and memory overhead (§5.2), and hardening HDFS should require reasonable engineering effort (§5.3).

2.1.1 Selective Versioning

While traditional N-versioning requires developers to re-implement *all* the functionality of the specification, *selective versioning* requires an additional version for only the most important functionality. The idea is that some functionality in the system is worth protecting more than other functionality, for several reasons.

First, some components are more *sensitive* to bugs and memory corruption. For instance, a bug in the HDFS namespace or replica management could cause irrecoverable data loss; a buggy transaction committed to the log can make the system crash permanently; corrupt internal state could make the system serve incorrect data. On the other hand, bugs in maintaining system statistics may be less harmful. Therefore, if one must prioritize, it is more appropriate to protect bug-sensitive functionalities first.

Other potential candidates for applying the SLEEVE approach are *new* or *frequently-changed* modules. Real-

world cases have shown that code that does not change frequently is relatively stable, and hence less likely to contain bugs, while new or frequently-changed code is more likely to be buggy [35, 53, 60].

Finally, some software systems already contain protection machinery for some modules. For example, HDFS on-disk data is already protected with checksums and replication [45, 57], and thus the second version could just protect the exposed in-memory system metadata.

HARDFS hardens namespace management, replica management, and the read/write protocol of HDFS.

2.1.2 Lightweight Versioning

With *lightweight versioning*, we avoid completely replicating the state maintained by the main version. This challenge particularly arises when a single node needs to store a large amount of state. For example, the HDFS namespace management could manage in-memory metadata of millions of files in one machine.

A naive approach for a 2-version system is to maintain the same amount of metadata in the second version as the main version. Although simple, this approach is unattractive because of its large memory overhead (potentially 100%). When memory is scarce, this design choice limits system scalability. For instance, doubling memory overhead could reduce the maximum number of files the system can manage [50]. Moreover, many systems may run on the same cluster (*e.g.*, Hadoop MapReduce [6], HBase [7], and HDFS), so doubling memory overhead is undesirable.

We exploit compact encoding techniques to minimize memory overheads. We have found that sleeved systems can be organized to ask boolean questions (*e.g.*, "Does file F really exist?"); therefore, we can use efficient encodings that answer boolean questions. HARDFS uses a Bloom filter to efficiently encode the file hierarchy for our sleeved namespace management functionality (§3.3).

2.1.3 Recovery

Detecting faults that are normally silent is the primary contribution of SLEEVE. Upon detection, a variety of standard recovery techniques or tools can be used, such as: restart, *fsck*, safemode or otherwise blocking dangerous actions, or failover.

In addition to simply detecting errors, SLEEVE can often pinpoint the problem, enabling sophisticated recovery options, such as *micro-recovery*, a fast alternative to full reboot. Fail-silent behaviors sometimes occur due to state corruption; with a second version of the internal state, the system can pinpoint and correct only the corrupt state. With the available redundancy, a sleeved system can initiate fast, fine-grained recovery as opposed to slow, coarse-grained recovery. HARDFS always attempts micro-recovery before resorting to full recovery (§3.5).

2.1.4 Soundness and Completeness

HARDFS is not sound: we do not attempt to guarantee that HARDFS never triggers recovery action unnecessarily. Like the main version, the second version is also subject to anomalous bit flips and bugs. As long as recovery actions have a small cost, occasional false positives are acceptable. HARDFS is not complete: we do not attempt to catch all faults with HARDFS. Our premise is that faults are more dangerous in some subsystems than others, and complete checking is not possible without a formal specification of behavior regardless. Although HARDFS fault detection is neither sound nor complete, our experiments show that HARDFS is quite useful for handling memory corruption and real software bugs (§5).

3 HARDFS Design

In this section, we describe our general approach to designing HARDFS with SLEEVE. In §4, we describe in detail how we implement the design to harden the HDFS namespace management, replica management, and the read/write protocol of datanodes with HARDFS-N, HARDFS-R, and HARDFS-D respectively.

3.1 Node Models

Like GFS clusters [33], HDFS clusters consist of a single *master* node and multiple *worker* nodes. The master is responsible for file-system metadata, including the namespace structure and the locations of block replicas. File metadata is kept in memory for fast operation, but for persistence and crash recovery, the master writes every namespace update to an on-disk log. The workers store block replicas on their local disks and keep block information in memory. Metadata and data operations are decoupled: while the master serves metadata operations, the workers serve read and write requests.

HDFS nodes can be described by a *behavioral model*: nodes perform *actions* in response to *events*. Events occur when a node receives *input* messages from other systems or when *periodic threads* trigger work. The actions a node performs include modifying the node's *memory state*, accessing persistent storage, and sending output messages based on the current state. In HARDFS, sleeved subsystems understand the behavioral model. A node is considered faulty if it performs *incorrect actions* (*i.e.*, actions that deviate from the model).

3.2 Hardened Subsystem Architecture

To harden a distributed storage system against incorrect actions, we augment each node in the system with a lightweight version that verifies node behavior. More specifically, we "sleeve" each node by interposing on message and file I/O without significantly changing the

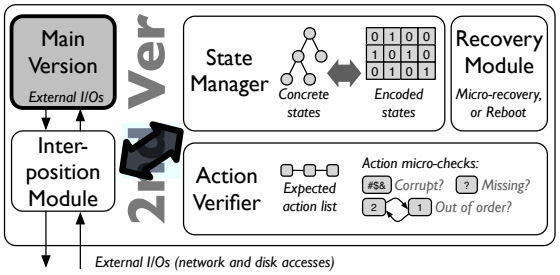

Figure 2: **Sleeved systems architecture.**

core implementation. With this approach, faulty behaviors are also isolated within a single node and not propagated to persistent storage or other nodes. As depicted in Figure 2, we use four major modules for each sleeved subsystem: an *interposition module*, a *state manager*, an *action verifier*, and a *recovery module*.

Interposition module: A sleeved system forwards all input messages to the main version, and forwards the messages relevant to the hardened functionality to the state manager. It also interposes on thread events to know when a periodic thread is triggered. This interposition is important because a periodic thread may trigger events that change the state of the main version; the second version must make equivalent changes to its own model. HARDFS uses AspectJ [2] to interpose on events without making major changes to the main version.

State manager: The state-manager module of HARDFS does the bookkeeping necessary to describe and check the data maintained in the main version. To be lightweight, the state manager keeps the state of the hardened functionality in *encoded states*. HARDFS encodes states with Bloom filters, but a variety of data structures could be used for this purpose. Since encoding techniques can incur high computational overhead during updates, HARDFS employs a small "cache" of *concrete states* for objects being actively modified (*e.g.*, the metadata for a currently open file). State management is further described in §3.3.

Action verifier: The action-verifier module detects faulty actions of the main version with a set of micro-checks. Using these checks, the sleeved system verifies every action of the hardened functionality before it impacts other components. We describe the challenges of verifying actions in §3.4.

Recovery module: After a fault has been identified by a sleeved system, the recovery module is triggered. Since faulty behavior has been isolated within a single node, recovery can be as simple as crashing and rebooting the faulty node. However, rebooting can take a significant amount of time; therefore, a sleeved system may optionally perform micro-recovery by semantically comparing every state object in the main version with the secondary version to recover only the corrupt objects. Recovery is described further in §3.5.

3.3 State Manager Module

We describe how the state manager operates, specifically how internal state is selected from the main version, derived from incoming messages and actions, and encoded in a lightweight manner.

3.3.1 Selective State Management

We selectively model a subset of the functionality and state of the main version. For instance, to verify namespace integrity (*e.g.*, correct file hierarchy) and corresponding operations (*e.g.*, file creation and deletion), HARDFS maintains directory entries without storing less important information such as access and modification times. State management is flexible: new information can be added incrementally to meet current needs (*e.g.*, one could add permission information for security checks if desired). HARDFS uses the same file formats for on-disk structures as vanilla HDFS, so upgrading HDFS to HARDFS or adding new memory state only requires a restart; copying data to a new file system is unnecessary.

In addition to storing the selected state, HARDFS needs logic for how state should be updated based on interposed messages; this logic acts as the second version. In order to implement this logic, we needed to understand the semantics of various protocol messages. For instance, for namespace management, upon a successful file creation message, HARDFS adds the corresponding file name to the maintained state.

Properly handling thread events that are periodically triggered is also necessary to keep both versions synchronized; if the second version were not aware of the thread events, it could not verify actions triggered by the threads. For example, when a periodic thread in the master node wakes and detects dead workers, the master may perform a block-replication action. The second version must be aware of this transition in order to verify the resulting actions correctly.

3.3.2 Lightweight State with Bloom Filters

We now discuss how our sleeved systems can manage state in an efficient and lightweight manner. While there are many ways to do this, in HARDFS, we use counting Bloom filters [21]. A Bloom filter is a probabilistic data structure that allows testing whether a data element is a member of a set. It is space efficient: the overhead does not depend on the state objects stored.

Our intuition for the use of Bloom filters is that sleeved systems typically only need to answer boolean questions (*e.g.*, does file `F` exist?) rather than answering non-boolean questions (*e.g.*, what are all files under directory `D`?). Thus, a Bloom filter is a fitting solution for

compressing file-system metadata. The challenges that arise are dealing with non-boolean verification, excessive CPU overhead, and false positives.

Dealing with non-boolean verification: Although using a Bloom filter is space efficient, one challenge is to represent non-boolean information, in particular information that changes and must be updated. For example, consider the case where both the main and second versions agree that file `F` is `100` bytes long. If a client appends the file and the worker tells the master that `F` is now `200` bytes long, then the second version must update its state regarding `F`. However, the second version cannot overwrite the old entry {`F`,`100`} previously stored in the Bloom filter with a new entry {`F`,`200`}. Instead, it must perform two operations: delete the old entry {`F`,`100`}, and then insert the new entry {`F`,`200`}. To delete the old entry the second version must know the value of the old entry, but Bloom filters cannot answer non-boolean questions (in this example, what is the current length of `F`?).

To deal with this, we use an *ask-then-check* technique. That is, the secondary version asks a non-boolean question of the main version to determine the previous value for an entry before the main version's event handler executes. Because the returned result cannot be trusted, the second version then checks the previous value with a boolean question to the Bloom filter. In the above example, the secondary version first asks the main version for the length of `F` (which is `100`) and then checks via the Bloom filter that `F` is indeed `100` bytes long. With this verified and correct information, the secondary version performs the deletion (we use a counting Bloom filter to support this operation) and hence the overwrite.

Dealing with excessive CPU overhead: While Bloom filters are space efficient, in some cases they can lead to excessive CPU overheads. To remedy this problem, HARDFS keeps a small "cache" of states being actively modified in *concrete* form (in contrast to the *compressed* form in the Bloom filter). In addition, HARDFS can optionally keep all data in concrete form, trading space efficiency for less CPU overhead. In the future, we plan to investigate policies for converting data between concrete and compressed forms based on run-time measurements.

Dealing with false positives: The last challenge is the presence of false positives from two sources: Bloom filters and corrupted state or bugs in the sleeved code itself. First, Bloom filters fundamentally can return false positives [20]. A Bloom filter can "lie" that it contains file `F`, when in fact it does not. Fortunately, the false positive rate is relatively small and configurable. For instance, the probability of a false positive in a Bloom filter with 10 hash functions and 32 bits per data element is approximately 2 per million [31]. Doubling the number of bits per data element to 64 leads to a false positive rate of 4 per billion; at this rate, a cluster processing 100 ops/second would experience about one false positive per month. Second, the state maintained by the sleeved version itself can be corrupt due to memory problems or bugs. Fortunately, in crash-tolerant systems, false positives are benign from a correctness perspective because they only result in unnecessary recovery. The only danger is the degenerate case where a Bloom filter always generates a false positive for a particular element, resulting in repeated recovery. A small cache of concrete states solves this problem; the recovery mechanism remembers troublesome elements and pins them in the cache.

3.4 Action Verifier Module

The action verifier module detects incorrect actions performed by the main version that relate to the properties of interest. We classify incorrect actions into four types: corrupt, missing, orphan, and out-of-order. We believe it is important to detect all four types of incorrect actions. In our study of the HDFS bug reports, we find that all these types of incorrect actions occur [8].

The following sections describe the four types of incorrect actions that could occur after a file creation request as illustrated in Figure 3. Figure 3a represents the correct behavior of a file creation; here the file does not exist, and thus the master accepts the request and writes an appropriate transaction to its persistent operation log.

3.4.1 Corrupt Actions

The first type of incorrect action is a corrupt action. Consider the scenario shown in Figure 3b where a client sends a request to create a file `F`; if the file did not previously exist, then the request should be accepted. However, if the main version of the master behaves incorrectly (*e.g.*, the in-memory pathname is corrupted), then the main version will wrongly reject the request, while the second version accepts.

However, when there is *disagreement* between the secondary and main versions, the secondary version cannot be trusted to be the correct version. Thus, whenever disagreement occurs for any of the actions described below, the action verifier simply catches the incorrect action and takes additional steps to resolve the problem. These steps are described in more detail in §3.4.5.

3.4.2 Missing Actions

Missing actions represent the case where the main version should generate a specific action but fails to do so. For example, in Figure 3c, the master accepts the file creation request but forgets to write the corresponding transaction to the operation log.

To check for missing actions, the action verifier maintains an *expected action* list and generates expected ac-

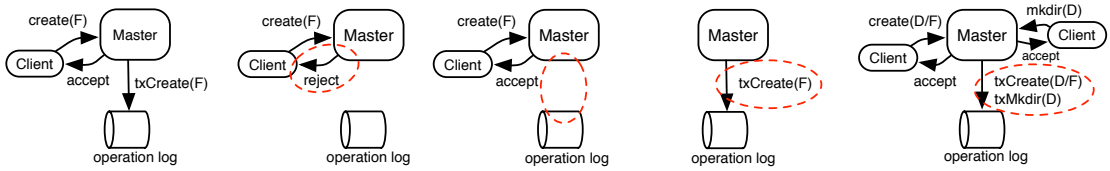

Figure 3: **Types of incorrect actions.**

tions for incoming requests or state changes that require a certain action. For example, a write to the operation log is expected to follow every accepted client-write. Expected-action entries describe both the action the main version should perform and when the action needs to be performed. Many actions are expected to occur before the main version's event handler returns, but in some cases, it is only possible to detect missing actions using timeouts. For example, replication in HDFS is throttled, so a namenode might not immediately send a replication command upon detecting an under-replicated block. Waiting too long, however, is incorrect behavior that could lead to data loss.

3.4.3 Orphan Actions

Orphan actions represent the case where the main version performs unexpected actions. For instance, in Figure 3d, the master node writes to the operation log that file `F` is created although there is no origin for this request. To detect orphan actions, the action verifier leverages the expected action-list. Specifically, it signals an error when the action has no match in the expected-action list.

Orphan actions also cover the case of *duplicate* actions. For example, consider the block re-replication procedure due to dead worker nodes. If the master sends too many block re-replication commands, then the first re-replication command will be considered correct, while the subsequent ones will be considered orphans.

3.4.4 Out-of-order Actions

An action may depend on another one. For example, a transaction creating a new file `F` (*op2*) cannot precede the transaction making the parent directory `D` (*op1*). If the main version executes *op2* before *op1* (as in Figure 3e), the operation log will be corrupt, which may lead to severe consequences such as data loss or the master crashing permanently during checkpoint recovery. To address this challenge, action dependencies are tracked. Tracking action dependencies is challenging and domain specific. We present a specific solution in §4.1.1.

3.4.5 Handling Disagreement

By detecting incorrect actions as explained above, the action verifier can identify disagreements, but with only two versions to compare, it cannot know which version is wrong; therefore, the action verifier resolves disagreements using domain-specific information and falls back on the safety of recovery from trusted state.

As an example, consider the request originally shown in Figure 3b, where the main and secondary versions disagree about the success of a file creation. It is entirely possible that the main version (correctly) rejected the request because a space quota was exceeded; if the second version does not incorporate knowledge about space quotas in its selective model, then it will (incorrectly) accept the request. Thus, the action verifier cannot conclude that the main version behaves incorrectly.

In several cases, we have found it much easier to implement a simplified secondary version that naively accepts requests that the complete main version rejects. To avoid false alarms in these cases, the action verifier examines the error code returned from the main version and ignores disagreements when the secondary version is not equipped to generate those cases. In the out-of-quota example, the action verifier agrees with the main version to reject the request and operation continues without recovery. Unfortunately, if the main version incorrectly reports "out-of-quota", HARDFS will not detect it. There is a tradeoff: writing logic for more cases improves reliability, but increases engineering effort.

For some situations, the action verifier needs a mechanism to detect repeated disagreement. If a transient fault causes disagreement, the same discrepancy will not reappear after recovery, and normal operation will resume. However, one of the versions may have a bug that causes permanent disagreement. In this case, the developer is notified, and policy determines how to proceed until the code is fixed; entering HDFS safemode is one option (safemode prevents all file and block modifications).

3.5 Recovery Module

Once the action verifier detects a failure, the recovery module can apply many different techniques. One simple approach is crash and reboot (suitable for crash tolerant systems). A more fine-grained technique is micro-recovery where the recovery module pinpoints and recovers only the corrupt state. In addition to doing repair, the recovery module can also thwart destructive actions to prevent fault propagation to other nodes. We discuss the three techniques used by HARDFS below.

3.5.1 Reboot

Crash and reboot is a safe mechanism to prevent the propagation of corrupt state due to transient and non-deterministic failures. Upon reboot, the main version can safely reload its in-memory state from other trusted sources, such as persistent storage or other nodes across the network; the states of the secondary version are also reloaded since it interposes on these inputs.

3.5.2 Micro-Recovery

When rebooting a node is expensive (*e.g.*, rebooting a master node may take hours [22]), a sleeved system can instead quickly identify and repair only the corrupted state. We call this technique micro-recovery (similar to micro-rebooting [23]). In micro-recovery, when a fault is detected, the node is frozen to prevent changes to the system state. The recovery module then identifies the corrupted state by semantically comparing the secondary and main version state, and recovers it from a trusted source (*e.g.*, persistent storage).

For example, if the two versions disagree about the length of a file `F`, then micro-recovery reconstructs just `F`'s metadata from the checkpoint file and the operation log on disk. These sources can be trusted for two reasons: data is never written to them unless both versions agree, and solutions for preventing and detecting corruption to persistent storage are well known [15, 17, 19, 27, 34, 36, 41, 44, 45, 51, 52].

Disagreement can happen because of corruption in either secondary or main version state (or both). Repairing corrupt main version state is relatively easy because the recovery module can overwrite the corrupt state "in place". Repairing encoded state in a Bloom filter is more challenging. Consider a corrupted entry {`F`,`374`}, incorrectly indicating F is 374 bytes long. To repair the corrupted entry, the recovery module must delete the encoded entry and insert the correct entry, but it does not know that `F`'s length has been corrupted to 374. The solution described in §3.3.2 does not work because there is no entity that knows the corrupt value. Therefore, our solution is to begin with an empty Bloom filter instance and add entries as they are verified, either from main-version state or persistent storage, without a full reboot.

If micro-recovery does not find any disagreement, it means the detected faults might involve corruption in non-hardened functionality or bugs in the software logic, and thus the recovery module falls back to full reboot. If recovery is continuously repeated, an error report is generated as discussed in §3.4.5.

3.5.3 Thwarting Destructive Actions

Repairing a local node is of limited value if the faulty node causes permanent damage to other nodes before it recovers. HDFS workers send regular heartbeat messages to the master, and the master replies with messages directing workers to perform various actions. Some of these directives, such as "delete replica" or "decommission", can cause irrecoverable data loss if misguided.

Our sleeved subsystems drop messages containing destructive directives if there is any disagreement between the main version and the secondary model about the objects in question. Our policy here is conservative; it is safer to potentially waste storage space than to risk deleting data unintentionally.

4 Implementation

In this section, we describe the specific details of the three HARDFS subsystems, HARDFS-N, HARDFS-R, and HARDFS-D, which harden the HDFS namespace management, replica management, and the read/write protocol of datanodes, respectively.

4.1 Namespace Management: HARDFS-N

Namespace management is a critical functionality in HDFS. The architecture of HDFS has a dedicated master, the *namenode*, which stores all file-system metadata in memory for fast operations. When the namenode executes a client request that changes the namespace, it writes an appropriate transaction to the on-disk operation log before responding back to the client. Periodically, the namenode replays this operation log to produce an on-disk checkpoint file that contains the complete namespace structure. HDFS splits files into 64MB blocks, which are replicated across *datanodes*.

To protect namespace integrity, HARDFS-N guards the in-memory namespace structures that are necessary for reaching data: the file-tree hierarchy, file-to-block mapping, and block-length information. With this protection, HARDFS-N detects namespace-related problems such as accidental file truncations, unreachable directories, and corrupt file-to-block mappings. When these problems are detected by HARDFS-N, faulty actions are not propagated to the client, persistent storage, or datanodes.

4.1.1 Maintaining State and Checking Actions

After interposing on incoming and outgoing messages, HARDFS-N can update its state (both Bloom filters and the expected-action list) and verify observed actions. Its logic for updating state from incoming messages is shown in Table 1. For example, in the first row of the table, at the entry path of the request `create(F)`, HARDFS-N records this fact by calling `insert(F)` to the Bloom filter. Table 1 can be seen as a concrete example of how a developer programs a sleeved service.

HARDFS-N uses Bloom filters as a space-efficient data structure for encoding the file namespace. Only three

Message	Logic of the secondary version

`create(F)`
client requests NN to create file F
	`Entry:`
	`  If exists(F) Then reject;`
	`  Else`
	`  insert(F);`
	`  generateAction(txCreate[F]);`
	`Return: check response;`

`addBlk(F)`
client requests NN to allocate a block to file F
	`Entry: F:X = ask-then-check(F);`
	`Return:`
	`  B = addBlk(F);`
	`  If exists(F) & !exists(B) Then`
	`  X′ = X ∪ {B};`
	`  update(F:X, F:X′);`
	`  insert(B@0);`
	`  Else declare error;`

`blkRcvd(B,100)`
DN informs NN of received 100-byte block B
	`Entry:`
	`  B@L = ask-then-check(B);`
	`  update(B@L, B@100);`
	`Return: check response;`

`complete(F)`
client informs NN of write completion on file F
	`Entry:`
	`  If exists(F) Then`
	`  SIZES = empty-list`
	`  F:X = ask-then-check(F);`
	`  for B in X:`
	`  B@L = ask-then-check(B);`
	`  SIZES.append(B@L);`
	`  generateAction(txClose[F,SIZES]);`
	`Return: check response;`

Table 1: **SLEEVE for namespace management.** *The table shows how the secondary version derives the semantics of input messages from a client or datanode (DN) to the namenode (NN), manages its state using Bloom filter APIs, and generates expected actions.* `update(x1, x2)` *represents a* `delete(x1)` *followed by an* `insert(x2)`*; "Check response" means that the secondary version compares returned results and handles disagreement if any;* `F:B` *represents a mapping from file* `F` *to block* `B`*;* `B@x` *indicates that block* `B` *is* `x` *bytes long. Multiple Bloom filters (not shown) are used to encode different facts.*

APIs are needed: `insert(x)`, `delete(x)`, and `exists(x)`, where `x` is a variable-length byte array.

To encode a file hierarchy, the most straight-forward approach would be to perform `insert("d/f")` to indicate that there exists a directory `d` with a child `f`. However, this scheme leads to inefficient performance if a directory has many entries and is frequently renamed [39]. Imagine there exist many entries `d/f1`, `d/f2`, `d/f3`, and so on, and directory `d` is renamed to `n`; since Bloom filters do not support overwrite (§3.3.2), the system would need to perform many ask-then-check operations to delete all `d/*` entries, and then insert all new `n/*` entries. Our solution is to introduce another level of indirection (*e.g.*, `keyOf(d)/f`). If the main version maintained a unique inode number for each directory, we could just use that information directly. Unfortunately, there is no such information. Instead, we use the hash code of the memory address for the Java object that represents the directory.

To catch orphan, missing, and out-of-order actions, HARDFS-N maintains an expected-action list. For example, in the first row of Table 1, upon an incoming `create(F)` request, a future action `txCreate` is expected. To detect out-of-order transactions, HARDFS-N uses domain-specific knowledge. Specifically, a completed transaction (a successful `txCreate(D)`) implies that the associated object (`D`) is committed to the on-disk log and that subsequent child additions to `D` are also allowed. With this knowledge, out-of-order transactions can be detected (*e.g.*, if `txCreate(D/F)` is sent to the disk but `txCreate(D)` is still in the expected-action list).

4.1.2 Recovery

HARDFS-N could recover from detected errors by rebooting the namenode and reconstructing all state. For faster recovery, HARDFS-N attempts micro-recovery first. Here, we describe further how corrupt states can be recovered from persistent storage.

HARDFS-N repairs corrupted states in memory (*e.g.*, bad `F`'s metadata) using states stored in the namenode's checkpoint file. Since we assume persistent storage is trusted (§3.5.2), the checkpoint file is expected to have "good" states. To obtain the latest checkpoint file, HARDFS-N forces the namenode to start a checkpointing process by replaying the operation log. However, HDFS checkpointing is relatively slow and I/O-intensive: it requires reading the old checkpoint file in its entirety, as well as the operation log, before it can write out a new checkpoint file. To optimize this, HARDFS-N avoids forcing a checkpoint when possible. HARDFS-N first scans the (relatively small) operation log to find the correct values for any of the relevant corrupted state (*e.g.*, `F`'s latest metadata). If no relevant transactions are found, HARDFS-N performs an efficient binary search on the checkpoint file for the needed information; the checkpoint file is already sorted based on pathname.

4.2 Replica Management: HARDFS-R

HDFS replica management involves the block-to-node mapping structure for tracking the node locations and number of replicas for every block. Since datanodes in a cluster may arrive and leave at any time, a block can be over- or under-replicated. Replica management ensures that each block has the intended number of replicas by sending deletion and regeneration commands to different datanodes. When a block is created/regenerated, the datanode sends a `blkRcvd` message to the namenode. Every datanode also sends periodic `blockReport` messages containing the list of blocks managed by that datanode.

HARDFS-R hardens the namenode replica management functionality by protecting the integrity of block-mapping states (*e.g.*, no blocks will be accidentally deleted and no incorrect block locations will be returned to the client). Since many of the basics are similar to HARDFS-N, we focus on the differences.

4.2.1 Maintaining State and Checking Actions

HARDFS-R uses two Bloom filters to encode block-to-node mappings and replica-count information with simple formats such as `insert(BlkID:NodeID)` and `insert(BlkID:Count)`. For every block regeneration/deletion command sent by the main version, HARDFS-R performs various checks. For example, deleting a block replica should not make the block under-replicated; a regeneration command should only be performed on a valid block.

HDFS uses a periodic thread to detect dead nodes. When the thread is triggered, HARDFS-R is informed (§3.3) so that it can replicate the functionality for block accounting and manage its expected-action list (*e.g.*, HARDFS-R expects to observe a block regeneration command if the block is under-replicated).

4.2.2 Recovery

HDFS namenode does not maintain block-to-node maps in its persistent storage; therefore, full recovery is done by requesting block mappings from all the datanodes (specifically by requesting `blockReport` commands). However, to perform micro-recovery on a corrupt block-to-node mapping (either in the main or secondary version), HARDFS-R only requests a block report from the corresponding node. If micro-recovery fails (§3.5.2), HARDFS-R falls back to full recovery.

4.3 Read/Write: HARDFS-D

Our final subsystem, HARDFS-D, hardens the datanode's metadata for reading and writing blocks. HARDFS-D can detect data access problems such as returning incorrect data or appending data at a wrong offset.

4.3.1 Managing State and Checking Actions

In each datanode, HARDFS-D protects two pieces of information: the list of blocks maintained by the datanode and the length of each block. In an append-only storage system such as HDFS, the block length is especially important since it defines the location of the next write; a corrupt length could lead to accidental overwrites. HARDFS-D uses two Bloom filters to protect the information (*e.g.*, `insert(B)` and `insert(B,100)`).

HARDFS-D verifies both disk and network actions. First, HARDFS-D checks that all disk accesses performed by the datanode are to the correct files and to the correct offsets. Second, HARDFS-D checks all outgoing network messages to ensure that any local corruption does not propagate to another datanode; this network check is vital because writes are preformed in a pipelined fashion.

4.3.2 Recovery

A corrupted and faulty datanode can be recovered with a simple reboot. Fortunately, because each block is typically replicated across multiple datanodes, rebooting a datanode does not affect data availability. In addition, as we will show in our evaluation, rebooting a datanode is fast, taking only a few seconds (§5.2.3). Therefore, we do not investigate micro-recovery for HARDFS-D.

Outcome	HDFS	HARDFS
No problem observed	728	460
Detect and reboot	-	140
Detect and micro-recover	-	107
Hang	22	16
Crash	133	268
Silent failure	**117**	**9**
(Corrupt pathname)	95	0
(Corrupt replication)	1	0
(Corrupt blocksize)	12	1
(Corrupt permission)	3	0
(Corrupt modification time)	6	8
Total	1000	1000

Table 2: **Outcomes of random memory corruption.**

5 Evaluation

We now evaluate HARDFS. Specifically, we present experimental results that answer the following questions:

- Is HARDFS effective at detecting and recovering from fail-silent faults caused by memory corruption and real-world bugs (§5.1)?
- How much time and space overhead does the additional bookkeeping incur? Does micro-recovery substantially improve recovery time (§5.2)?
- Does hardening HDFS require a reasonable amount of engineering effort (§5.3)?

5.1 Detection and Recovery

We evaluate the ability of HARDFS to detect faults and recover using three sets of experiments. We first randomly corrupt memory by injecting bit flips in the namenode's address space. To further understand the effect of such corruptions, we perform memory corruptions that target various fields in important data structures. Finally, we reintroduce real bugs to the codebase and measure how well HARDFS can prevent data loss.

5.1.1 Random Memory Corruption

We study how random memory corruptions affect the operation of vanilla HDFS and HARDFS by injecting random bit flips in the namenode's address space. Specifically, for each system we performed 1000 runs, each of which involved: (1) creating 10,000 files, (2) injecting random bit flips into the namenode's writable address space, and (3) recording if the system crashes or if `stat()` returns unexpected metadata (*i.e.*, a silent failure has occurred).

We focus on namenode corruptions because it is a single point of failure in the system and has more potential to propagate errors. Each bit has one chance in 50 million of being flipped. With our injection methodology,

both the main implementation and the secondary model are subject to the random injections, so discrepancies can arise when state in either is corrupted.

Table 2 summarizes the experimental results. With HARDFS, the number of silent failures is reduced by a factor of 10 (from 117 to 9) because the failures are mostly detected and recovered (140 reboots and 107 micro-recovery instances). However, the JVM crashes twice as often (because additional bookkeeping increases the chance of pointer corruptions that lead to crashes). All the crashes were due to dangling pointers, memory protection errors, or illegal instructions. HARDFS trades availability for correctness and data safety.

The breakdown of silent failures illustrates the result of selective protection. In HDFS, corrupt pathnames are most common (95 cases) followed by corrupt blocksizes (12 cases). In contrast, the most common silent failure for HARDFS is a corruption of the modification time (8 cases), which arises because we selectively chose not to protect this inode field. For HARDFS, there is only one dangerous failure, a corruption of blocksize; although HARDFS protects this field, it is possible that either our aggressive injections caused the logic checks to misbehave, or perhaps the same corruption happened to both the main version and secondary model, leading to this false negative. It is difficult to reproduce this case.

5.1.2 Targeted Memory Corruption

We also conduct targeted memory corruption because the random memory corruption experiment gives us little information about which part of memory is corrupted and its corresponding effect. To do this, we pick a field of the namespace data structure (*e.g.*, pathname, block ID, etc.), change it to an unexpected value (*e.g.*, from `f0` to `f1`), and run a simple workload (*e.g.*, file creation).

Table 3 summarizes our experimental results. We see that, despite employing on-disk replication, vanilla HDFS is quite fragile to memory corruption (many $\times$ and $\bigcirc$ outcomes). For instance, a block ID corruption can cause the namenode to remove all replicas of a block; faulty transactions can be written to the on-disk operation log, eventually leading to unsuccessful checkpoints and reboots; corrupted states can propagate, leading to global corruption. HARDFS-N, on the other hand, correctly detects and recovers from all of the 54 injected faults without propagating the faults to the disk or other nodes.

We also investigate whether HARDFS-N can handle the secondary version behaving incorrectly. Specifically, we repeat the experiment in Table 3 where we corrupt the pathname and the file-to-block mapping, but this time within the Bloom filters. We find (not shown) that the recovery module successfully recreates HARDFS-N's internal state by reconstructing a new instance of the Bloom filter (as described in §3.5.2). The time to populate a new instance of the Bloom filter is negligible: it takes only 2 seconds for a namespace of 200K files.

	HDFS								HARDFS							
Message	P	C	S	R	B	I	G	L	P	C	S	R	B	I	G	L
`mkdir`	⊗	⊗	.	.	.	.	.	.	√	√	.	.	.	.	.	.
`create`	⊗[a]	⊗[b]	×	×	.	.	.	.	√	√	√	√	.	.	.	.
`append`	⊗[c]	⊗[c]	⊗[c]	⊗	⊗[c]	⊗[c]	⊗[b]	⊗[b]	√	√	√	√	√	√	√	√
`addBlk`	○	○	.	○	.	.	.	.	√	√	.	√	.	.	.	.
`blkRcvd`	.	.	.	.	.	⊗[b]	⊗[b]	.	.	.	.	.	.	√	√	.
`fsync`	⊗[a]	.	⊗[c]	⊗	⊗[c]	⊗[c]	⊗[b]	⊗[c]	√	.	√	√	√	√	√	√
`complete`	⊗[a]	.	⊗[c]	⊗	⊗[c]	⊗[c]	⊗[b]	⊗[c]	√	.	√	√	√	√	√	√
`delete`	⊗	⊗	.	.	⊗[b]	⊗[b]	.	.	√	√	.	.	√	√	.	.
`rename`	⊗	⊗	.	.	.	.	.	.	√	√	.	.	.	.	.	.
`setRep`	⊗[a]	.	.	×	.	.	.	.	√	.	.	√	.	.	.	.
`setTimes`	⊗[a]	.	.	.	.	.	.	.	√	.	.	.	.	.	.	.
`getInfo`	○	○	○	○	.	.	.	.	√	√	√	√	.	.	.	.
`getListing`	○	○	.	.	.	.	.	.	√	√	.	.	.	.	.	.
`getBlks`	⊗[a]	⊗	.	.	○	○	○	○	√	√	.	.	√	√	√	√

Table 3: **Namespace memory corruption experiments.** *Corrupted metadata fields are: (`P`) pathname, (`C`) child pointer, (`S`) default block size, (`R`) replication factor, (`B`) block pointer, (`I`) block ID, (`G`) block generation stamp, and (`L`) actual block length. Each cell presents the resulting actions from the combination of input message (e.g., mkdir) and corrupted internal state. Possible outcomes are: ($\times$) faulty transaction, ($\bigcirc$) incorrect response, ($\surd$) correct transaction and response, (.) inapplicable. Footnotes:* [a] *the namenode fails to reboot and crashes permanently;* [b] *data loss;* [c] *inconsistency.*

Bug	Year	Priority	Description
HADOOP-1135	2007	Major	Blocks in block report wrongly marked for deletion
HADOOP-3002	2008	Blocker	Blocks removed during safemode
HDFS-900	2010	Blocker	Valid replica deleted rather than corrupt replica
HDFS-1250	2010	Major	Namenode processes block report from dead datanode
HDFS-3087	2012	Critical	Decommission before replication during namenode restart

Table 4: **Software bugs.**

To measure the benefits of HARDFS-D, we corrupt replica metadata during block reads and writes (not shown). Although vanilla HDFS datanodes handle faults better than the namenode in our last experiment, half of the trials still resulted in data loss, corruption, or incorrect responses. The data replication in HDFS is useless if corruption can spread. HARDFS-D, however, detects faulty behaviors immediately and reboots the faulty node. In every trial, the fault is isolated, operations continue successfully, and no data is lost or corrupted.

5.1.3 Real Software Bugs

In this section, we explore how well HARDFS handles real software bugs. We chose five bugs from Hadoop and HDFS bug repositories [5, 8]; Table 4 gives a summary.

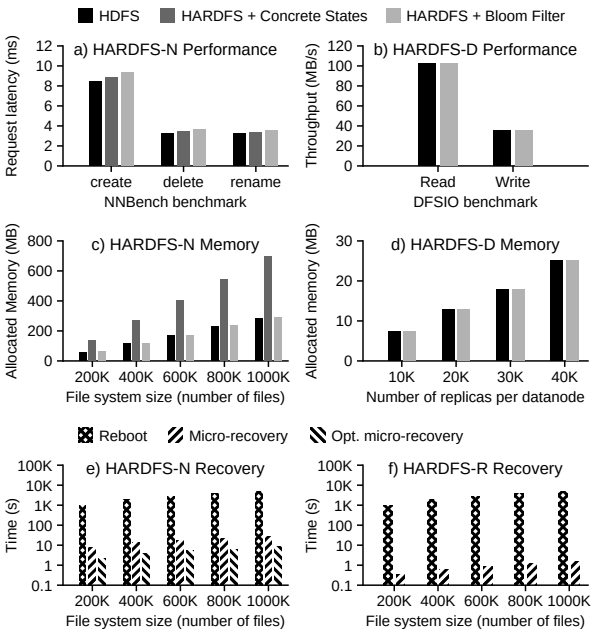

Figure 4: **Performance: time, space, and recovery.**

The bugs have the following characteristics: they affect at least one of the subsystems we hardened, the bugs received a rank "major" or greater, and the bugs result in data loss under certain circumstances.

The bugs were discovered over a number of years, ranging from 2007 to 2012. The older bugs tend to be simple programmer oversights. For example, deletion of valid blocks can be triggered by a poorly written loop that processes block reports incorrectly [1], or because of missing safemode checks [9]. The newer bugs tend to be more subtle. For example, blocks could be deleted or under replicated due to complex ordering of thread executions, messages, operator actions and failures [3, 4, 10].

For each bug, we made controlled modifications to our codebase to reproduce it. We limited ourselves to reintroducing the buggy code, injecting delays to reorder events, and dropping messages. As these bugs lead to behaviors that deviate from the expected model, HARDFS was able to detect the problem in each case and take appropriate action. In one case, HARDFS was able to restore proper state by restarting the namenode, and in four cases, HARDFS prevented data loss by simply thwarting the destructive directives (§3.5.3).

5.2 Efficiency

We now evaluate the performance impact, space overhead and recovery time of each HARDFS system. All experiments were conducted in a cluster of 21 machines, each having 8GB of memory and a 2.66GHz CPU.

5.2.1 Performance Impact

We evaluate the time overhead of HARDFS-N using the namenode benchmark (NNBench) in the Hadoop distribution (Figure 4a). This benchmark stresses many metadata requests by creating, renaming, and deleting files. HARDFS-N imposes acceptable overhead: 4% or 8%, with concrete state or Bloom filters, respectively.

In an experiment designed to stress HARDFS-R containing heavy client write activity and significant background processing of block reports, we found that the performance overhead of HARDFS-R is negligible.

Finally, to evaluate the performance of HARDFS-D, we run the DFSIO benchmark with 3-way replication on a cluster containing one dedicated namenode and 20 datanodes and measure the average throughput of read and write operations. The results in Figure 4b show that the overhead of HARDFS-D is negligible for both workloads.

5.2.2 Memory Overhead

We measure the memory allocated for the namenode in both HDFS and HARDFS-N as the number of managed files is varied. Figure 4c shows that concrete states in HARDFS-N lead to memory overheads near 100%; this accentuates the need for lightweight data structures such as Bloom filters. As desired, the memory overhead of HARDFS with Bloom filters is negligible (2.6%).

We measure the memory allocated for both HDFS and HARDFS-D by varying the number of replicas a datanode manages (Figure 4d). The space efficiency of Bloom filters makes the memory overhead of HARDFS-D less than 1%. Finally, HARDFS-R with Bloom filters incurs less than 2% memory overhead (not shown).

5.2.3 Recovery Time

We measure the time for HARDFS-N to recover corrupt states using three approaches: simple reboot, micro-recovery, and optimized micro-recovery. Normal micro-recovery creates a new checkpoint based on the last checkpoint and the operation log; optimized micro-recovery computes only the needed state by efficiently scanning the log and last checkpoint (§4.1.2). Figure 4e summarizes the results. Although crash-and-reboot works correctly, it is prohibitively expensive: more than an hour is required to reboot a namenode managing 1 million files; most of this time is spent processing block reports from datanodes [22]. Fortunately, micro-recovery is highly efficient and more than two orders of magnitude faster. Our optimized version can recover corrupted state in less than 10 seconds, even when the namenode is managing 1 million files. Figure 4f shows similar benefits of using micro-recovery for HARDFS-R.

HARDFS-D does not utilize micro-recovery, as a datanode reboot is relatively quick. A datanode reboot involves reading a block list from local disks and sending the list to the namenode. In our experiments, it takes about 2 seconds to reboot a datanode storing 40,000 blocks (around 2.5TB) of data (not shown).

Subsystem	HDFS	HARDFS	
Namespace management	10114	1751	(17%)
Replica management	2342	934	(40%)
Read/Write protocol	5050	944	(19%)
Others	13339	0	(0%)
Total	30845	3629	(12%)

Table 5: **Engineering effort in lines of code.**

5.3 Engineering Effort

Table 5 compares the effort of implementing HDFS to the effort of hardening HDFS. By selecting three key modules where correctness is most important, we were able to focus our efforts on 57% of the codebase. Our lightweight second versions are much smaller than the original versions (17% to 40% of the main-version sizes); overall, our changes only increase the codebase by 12%. Although implemented in Java, HARDFS could be implemented in declarative languages [13, 46] in order to further reduce the engineering effort. We leave that for future work.

6 Related Work

In this section we discuss related work on which HARDFS is based and other approaches to addressing memory corruption and software bugs.

HARDFS is primarily based on two related works: N-version programming (NVP) [14] and Micro-reboot [23]. Traditional NVP systems require high engineering effort to develop multiple versions of a software system. In addition, coordinating different implementations often requires complex machinery and incurs significant overhead [16]. HARDFS reduces engineering costs by only protecting select subsystems with redundant implementations. HARDFS minimizes overhead by making data structures lightweight via lossy compression.

Lossy compression occasionally causes unnecessary recovery due to error-detection false positives; we make this acceptable by making recovery inexpensive with Micro-reboot, which advocates that systems should be designed with the ability to reboot partial components [23]. Micro-reboot has been useful in other systems, allowing OS drivers and file systems to be restarted without a full OS reboot [54, 55, 56].

A common way to address memory corruption is to add detection machinery at the hardware and software layer (*e.g.*, using ECC memory and page checksums). These approaches do not protect the system from bugs introduced by complex software in many layers. An end-to-end approach to handling corruption is provided by PASC [29], a library that makes it easy for developers to maintain two replicas of the main state and execute the program logic twice on both replicas. PASC involves minimal engineering effort since developers do not need to implement the same functionality twice; however, simply executing the same code twice makes the system vulnerable to bugs in that code. Furthermore, keeping two complete state replicas is costly.

One way to address bugs (but not memory corruption) is to perform offline testing driven by sophisticated model checkers [37, 58, 59]. Model checking is complementary to SLEEVE. It is more desirable to find and fix a bug during testing than to tolerate the bug during deployment; however, offline testing can only address bugs that arise in the situations selected by the model checker's execution-exploration and state-exploration algorithms. By contrast, SLEEVE performs checking in every situation that arises in deployment.

Some systems, like HARDFS, attempt to address both bugs and memory corruption. Recon [32] interposes on all disk writes by the file system, and prevents any writes that would break *fsck*'s consistency rules. Although relatively lightweight, Recon only checks for consistency, not correctness in general. Byzantine Fault Tolerance (BFT) [24, 43, 47] is a heavyweight solution which protects software systems from malicious behaviors like corruption, bad inputs, and wrong computation. Unfortunately, BFT requires a high degree of replication ($3f+1$ replicas to tolerate f failures), does not handle cases where the logic of the software is buggy, and may be difficult to deploy (*e.g.*, requires significant changes to the HDFS replication policies [28]).

7 Conclusion

Distributed systems fail, and worse, sometimes they fail silently. We propose SLEEVE, a new approach that encourages developers to harden their systems against fail-silent behaviors with minimal engineering effort. Central to our approach is the idea of building a lightweight version that protects important components of the system. Applying the SLEEVE approach, we harden HDFS and show that it can detect and recover from a wide range of fail-silent behaviors caused by memory corruptions and software bugs. We hope that the SLEEVE approach can be applied to distributed systems beyond HDFS.

8 Acknowledgments

We thank the anonymous reviewers and our shepherd Ashvin Goel for their feedback, which helped improve the quality of the paper. This material was supported by funding from NSF grants CNS-1218405, CCF-0937959, CSR-1017518, CCF-1016924, CCF-1017073, as well as generous support from NetApp and Google. Tyler Harter is supported by the Facebook Fellowship and NSF Fellowship. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] A block report processing may incorrectly cause the namenode to delete blocks. https://issues.apache.org/jira/browse/HADOOP-1135.

[2] AspectJ. www.eclipse.org/aspectj.

[3] Corrupt replicas are not tracked correctly through block report from DataNode. https://issues.apache.org/jira/browse/HDFS-900.

[4] Decommissioning on NN restart can complete without blocks being replicated. https://issues.apache.org/jira/browse/HDFS-3087.

[5] HADOOP JIRA. http://issues.apache.org/jira/browse/HADOOP.

[6] Hadoop MapReduce. http://hadoop.apache.org/mapreduce.

[7] HBase. http://hbase.apache.org.

[8] HDFS JIRA. http://issues.apache.org/jira/browse/HDFS.

[9] HDFS should not remove blocks while in safemode. https://issues.apache.org/jira/browse/HADOOP-3002.

[10] Namenode accepts block report from dead datanodes. https://issues.apache.org/jira/browse/HDFS-1250.

[11] Amazon s3 availability event. http://status.aws.amazon.com/s3-20080720.html, 2008.

[12] Observations on errors, corrections, and trust of dependent systems. http://perspectives.mvdirona.com/, 2012.

[13] Peter Alvaro, Tyson Condie, Neil Conway, Khaled Elmeleegy, Joseph M. Hellerstein, and Russell C Sears. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *EuroSys '10*.

[14] Algirdas A. Avižienis. The Methodology of N-Version Programming. In Michael R. Lyu, editor, *Software Fault Tolerance*, chapter 2. John Wiley & Sons Ltd., 1995.

[15] L. Bairavasundaram, Garth R. Goodson, Bianca Schroeder, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*, pages 223–238.

[16] L. Bairavasundaram, Swaminathan Sundararaman, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Tolerating File-System Mistakes with EnvyFS. In *USENIX '09*.

[17] Wendy Bartlett and Lisa Spainhower. Commercial Fault Tolerance: A Tale of Two Systems. *IEEE Transactions on Dependable and Secure Computing*, 1(1):87–96, January 2004.

[18] Ken Birman, Gregory Chockler, and Robbert van Renesse. Towards a Cloud Computing Research Agenda. *ACM SIGACT News*, 40(2):68–80, June 2009.

[19] Dina Bitton and Jim Gray. Disk shadowing. In *VLDB 14*, pages 331–338, Los Angeles, CA, August 1988.

[20] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13:422–426, 1970.

[21] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An improved construction for counting bloom filters. In *Proceedings of the 14th conference on Annual European Symposium - Volume 14*, ESA'06, pages 684–695, London, UK, UK, 2006. Springer-Verlag.

[22] Dhruba Borthakur. Hadoop avatarnode high availability. http://hadoopblog.blogspot.com/2010/02/hadoop-namenode-high-availability.html, 2010.

[23] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI '04*, pages 31–44.

[24] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *OSDI '99*, New Orleans, LA, February 1999.

[25] Tushar Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live - An Engineering Perspective. In *PODC '07*.

[26] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System for Structured Data. In *OSDI '06*, pages 205–218.

[27] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[28] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP '09*.

[29] Miguel Correia, Daniel Gomez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical hardening of crash-tolerant systems. In *USENIX '12*.

[30] Jeffrey Dean. Underneath the Covers at Google: Current Systems and Future Directions. In *Google I/O '08*.

[31] Li Fan, Pei Cao, J. Almeida, and A.Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281 –293, jun 2000.

[32] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying file system consistency at runtime. In *FAST '12*.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *SOSP '03*, pages 29–43.

[34] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[35] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. Has the bug really been fixed? In *ICSE (1)*, pages 55–64, 2010.

[36] H. Gunawi, V. Prabhakaran, S. Krishnan, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *SOSP '07*.

[37] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Peter Alvaro, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Koushik Sen, and Dhruba Borthakur. FATE and DESTINI: A Framework for Cloud Recovery Testing. In *NSDI '11*.

[38] James Hamilton. On Designing and Deploying Internet-Scale Services. In *LISA '07*.

[39] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: understanding the i/o behavior of apple desktop applications. In *SOSP '11*.

[40] Alyssa Henry. Cloud Storage FUD: Failure and Uncertainty and Durability. In *FAST '09*.

[41] Gordon F. Hughes and Joseph F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives Using SATA Disk Drives. *ACM Transactions on Storage*, 1(1):95–107, February 2005.

[42] Avinash Lakshman and Prashant Malik. Cassandra - a decentralized structured storage system. In *LADIS '09*.

[43] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[44] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John K. Ousterhout, Mendel Rosenblum, and Mendel Rosenblum. Fast crash recovery in ramcloud. In *SOSP '11*.

[45] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, pages 109–116, Chicago, IL, June 1988.

[46] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, Charles Killian, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *NSDI '06*.

[47] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. Base: using abstraction to improve fault tolerance. In *SOSP '01*.

[48] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: A Large-Scale Field Study. In *SIGMETRICS '09*, Seattle, Washington, June 2007.

[49] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop Distributed File System. In *MSST '10*, 2010.

[50] Konstantin V. Shvachko. Hdfs scalability: The limits to growth. *;login:*, 35, 2010.

[51] Muthian Sivathanu, V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *FAST '04*, pages 15–30, San Francisco, CA, April 2004.

[52] Muthian Sivathanu, V. Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST '03*, pages 73–88, San Francisco, CA, April 2003.

[53] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *ACM SIGSOFT Software Engineering Notes*, pages 1–5, 2005.

[54] Swaminathan Sundararaman, Sriram Subramanian, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Michael M. Swift. Membrane: Operating System Support for Restartable File Systems. In *FAST '10*.

[55] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *SOSP '03*.

[56] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. In *OSDI '04*, pages 1–16.

[57] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[58] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI '09*.

[59] Junfeng Yang, Can Sar, and Dawson Engler. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *OSDI '06*.

[60] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi Bairavasundaram. How do fixes become bugs? – a comprehensive characteristic study on incorrect fixes in commercial and open source operating systems. In *FSE '11*.