An Empirical Study on the Correctness of Formally Verified Distributed Systems

Pedro Fonseca Kaiyuan Zhang Xi Wang Arvind Krishnamurthy
University of Washington
{pfonseca, kaiyuanz, xi, arvind}@cs.washington.edu

Abstract

Recent advances in formal verification techniques enabled the implementation of distributed systems with machine-checked proofs. While results are encouraging, the importance of distributed systems warrants a large scale evaluation of the results and verification practices.

This paper thoroughly analyzes three state-of-the-art, formally verified implementations of distributed systems: IronFleet, Verdi, and Chapar. Through code review and testing, we found a total of 16 bugs, many of which produce serious consequences, including crashing servers, returning incorrect results to clients, and invalidating verification guarantees. These bugs were caused by violations of a wide-range of assumptions on which the verified components relied. Our results revealed that these assumptions referred to a small fraction of the trusted computing base, mostly at the interface of verified and unverified components. Based on our observations, we have built a testing toolkit called PK, which focuses on testing these parts and is able to automate the detection of 13 (out of 16) bugs.

1. Introduction

Distributed systems, complex and difficult to implement correctly, are notably prone to bugs. This is partially because developers find it challenging to reason about the combination of concurrency and failure scenarios. As a result, distributed systems bugs pose a serious problem for both service providers and end users, and have critically caused service interruptions and data losses [57]. The struggle to improve their reliability spawned several important lines of research, such as programming abstractions [5, 36, 44], bug-

Revised version (4/19/2017)

EuroSys '17 April 23-26, 2017, Belgrade, Serbia
© 2017 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-4938-3/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3064176.3064183

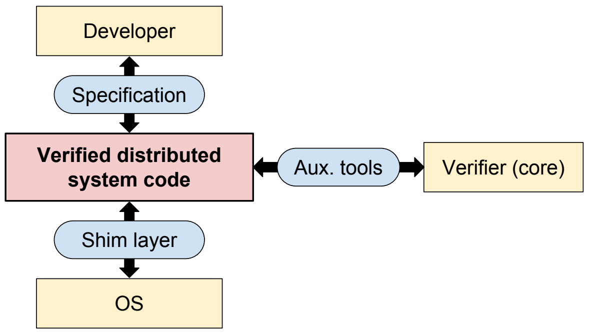

Figure 1: An overview of the workflow to verify a distributed system implementation.

finding tools [26, 37, 53, 54], and formal verification techniques [22, 29, 34, 52].

Formal verification, in particular, offers an appealing approach because it provides a strong correctness guarantee of the absence of bugs under certain assumptions. Over the last few decades, the dramatic advances in formal verification techniques have allowed these techniques to scale to complex systems. They were successfully applied to build large single-node implementations, such as the seL4 OS kernel [27] and the CompCert compiler [33]. More recently, they enabled the verification of complex implementations of distributed protocols, including IronFleet [22], Verdi [52], and Chapar [34], which are known to be non-trivial to implement correctly.

At a high level, verifying these distributed system implementations follows the workflow shown in Figure 1. First, developers describe the desired behavior of the system in a high-level *specification*, which is often manually reviewed and trusted to be correct. Developers also need to model the primitives, such as system calls provided by the OS, on which the implementation relies upon; we refer to this as the *shim layer*. Finally, developers invoke *auxiliary tools* (e.g., scripts) to communicate with a verifier and print results. The specification, the shim layer, and auxiliary tools, as well as the components they glue together, are part of the trusted computing base (TCB). If the verification check passes, it

	Protocol	**Consistency**	**Code size**
IronFleet	Multi-Paxos	Linearizability (⋆)	34K lines of Dafny/C#
Verdi	Raft	Linearizability	54K lines of Coq/OCaml
Chapar	[2, 38]	Causal	20K lines of Coq/OCaml

(⋆) IronFleet's specification does not guarantee exactly-once semantics. See §5.

Figure 2: Summary of the verified distributed systems we analyzed.

guarantees the correctness of the implementation, assuming the TCB is correct.

This paper conducts the first empirical study on the correctness of formally verified implementations of distributed systems. While formal verification gives a strong correctness guarantee under certain assumptions, our overarching research goal is to understand the effectiveness of current verification practices: what types of bugs occur in the process of verifying a distributed system, and where do they occur? We focus on possible bugs in distributed systems; bugs in external components, such as the OS, the verifier, or the hardware, are beyond the scope of this paper.

In particular, this paper addresses the following three research questions:

1. How reliable are existing formally verified distributed systems and what are the threats to their correctness?
2. How should we test the assumptions relied upon by verification?
3. How can we move towards real-world, "bug-free" distributed systems?

To answer these questions, we studied three state-of-the-art verified distributed systems (Figure 2). We acknowledge that these systems, although with a formal correctness proof, are research prototypes; our ultimate goal is *not* to find bugs in them, but rather to understand the impact of the assumptions made by formal verification practices when applied to building distributed systems. §3 will provide a detailed discussion of our methodology.

Our four main contributions follow. Surprisingly, we have found 16 bugs in the verified systems that have a negative impact on the server correctness or on the verification guarantees. Importantly, analyzing their causes reveals a wide range of mismatched assumptions (e.g., assumptions about the unverified code, unverified libraries, resources implicitly used by verified code, verification infrastructure, and specification). This finding suggests that a single testing technique would be insufficient to test all the assumptions that actually fail in real-world scenarios when building verified distributed systems; instead, developers need a similarly diversified testing toolkit.

Second, we observe that the identified bugs occur at the *interface* between verified and other components, namely in the specification, shim layer, and auxiliary tools, rather than in the rest of the system (e.g., the OS). These interface components typically consist of only a few hundred lines of source code, which represent a tiny fraction of the entire TCB (e.g., the OS and verifier). However, they capture important assumptions made by developers about the system; their correctness is vital to the assurances provided by verification and to the correct functioning of the system.

Third, *none* of these bugs were found in the distributed protocols of verified systems, despite that we specifically searched for protocol bugs and spent more than eight months in this process. This result suggests that these verified distributed systems *correctly* implement the distributed system protocols, which is particularly impressive given the notorious complexity of distributed protocols. The absence of protocol bugs found in the verified systems sharply contrasts with the results of an analysis we conducted of known bugs in *unverified* distributed systems. This analysis confirms that even mature, unverified distributed systems suffer from many protocol-level bugs. It suggests that these verification techniques are effective in significantly improving the reliability of distributed systems.

Finally, based on the evaluation results and with the goal of complementing verification techniques, we built the PK[1] testing toolchain that detects the majority of the bugs found. The toolkit can be generalized to find similar bugs in other verified systems. Inspired by the findings of our study, we limited testing to the components that were found to be the source of bugs for *real-world verified systems*. In particular, our toolchain does not test verified components; it tests only the TCB and, additionally, it *focuses* testing on the interface between verified and unverified components.

2. Background

This section provides background on verification techniques, replicated distributed protocols, and the verified distributed systems that we analyzed.

2.1 Machine-Checked Verification

An important technique to formally reason about systems relies on the programmer writing formal *proofs*. As opposed to pen-and-paper proofs, machine-checked proofs provide the assurance that each step in the proof is correct—a key factor given that proofs can be extensive and complex.

Verification provides a formal guarantee that the system satisfies a *specification*, which consists of: (1) a formal description of the properties (behavior) that the system must satisfy, and (2) the assumptions made about the environment, such as the network and file system.

The specification is a critical concept in verification. The specification is important to (a) informally convince developers that the system has the properties that they desire and (b) formally verify other systems through *compositional* verification techniques. The former application of the specification relies on the fact that the specification is often smaller and simpler than the implementation, increasing developer's confidence that it is correct through manual inspection.

[1] PK is an acronym for Panacea Kit

Bug	Component	Trigger	Incorrect results	Crash	Impact	Reported	Fixed	PK
Specification								
I1	High-level specification	Packet duplication	-	-	Void exactly-once guarantee	✓	-	✓
C4	Test case	-	-	-	Void client guarantee	✓	✓	-
Verification tool								
I2	Verification framework	Incompatible libraries	-	-	Verify incorrect programs	✓	✓	✓
I3	Verification framework	Signal delivered	-	-	Verify incorrect programs	✓	✓	-
I4	Binary libraries	-	-	-	Prevent verification	-	✓	✓
Shim layer								
V1	Client-server communication	Partial socket read	-	✓	Crash server	✓	-	✓
V2	Client-server communication	Client input	✓	✓	Inject commands	✓	-	✓
V3	Recovery	Replica crash	-	✓	Crash server	✓	-	✓
V4	Recovery	Replica crash	✓	✓	Crash server	✓	-	✓
V5	Recovery	OS error during recovery	✓	-	Incomplete recovery	✓	-	✓
V6	Server-server communication	Lagging replica	-	✓	Crash server	-	✓	✓
V7	Server-server communication	Lagging replica	-	✓	Crash server	-	✓	✓
V8	Server-server communication	Lagging replica	-	✓	Crash server	✓	-	-
C1	Server-server communication	Packet duplication	✓	-	Violate causal consistency	✓	-	✓
C2	Server-server communication	Packet loss	-	✓	Return stale results	✓	-	✓
C3	Server-server communication	Client input	✓	✓	Hang and corrupt storage	✓	-	✓

Figure 3: Bugs that our analysis found in the high-level specification, verification tool, and shim layer of verified distributed systems. Some bugs caused servers to crash or to produce incorrect results, and most bugs are detected by our testing toolchain (PK). We reported all listed bugs to developers, except bug `V6` and bug `V7`, which the developers had already fixed.

Importantly, all formal guarantees provided by machine-checked verification are valid as long as the *trusted computing base* (TCB) is correct. For verified systems, the TCB includes: the specification, the verification tools (e.g., verifier, compiler, build system), and the runtime infrastructure (e.g., libraries, OS, hardware). With a correct TCB, verification ensures that the implementation "bug-free."

2.2 Replicated Distributed Protocols

The systems we studied implement replicated distributed protocols. IronFleet and Verdi implement *replicated state machine* protocols (MultiPaxos [30] and Raft [45], respectively), while Chapar implements a replicated key-value store that provides *causal consistency* [2, 38]. This section provides background about these protocols.

Replicated state machine protocols. Replicated state machine (RSM) protocols replicate an arbitrary state machine over a set of replicas while providing the abstraction of a single server running a single state machine. Both MultiPaxos and Raft, which are leader-based, aim to provide fault-tolerance under the crash-fault model where replicas and clients communicate over asynchronous networks. MultiPaxos and Raft provide linearizable [23] semantics to clients—the strongest consistency guarantee [31].

Causal consistency protocols. Causal consistency, a weaker form of consistency, uses the notion of *potential causality* [28]. It imposes fewer restrictions on implementation behavior than linearizability, potentially improving performance while still providing intuitive semantics. Lloyd et al. [38] and Ahamad et al. [2] proposed two different algorithms for causal consistency.

2.3 Verified Systems Surveyed

We survey three state-of-the-art verified distributed systems: IronFleet, Verdi, and Chapar.

IronFleet. IronFleet proposes a methodology to verify distributed systems that relies on state machine refinement [1, 19, 29] and Hoare-logic verification [16, 24]. It provides a verified implementation of a MultiPaxos server library (§2.2) and an implementation of a counter that uses the library.[2] IronFleet aims at proving the safety (linearizability) and liveness of the MultiPaxos library.

IronFleet is implemented and verified using Dafny. The Dafny compiler and verifier [32] relies on a low-level compiler and verifier (Boogie [4]) and on an SMT solver (Z3 [14]). Some of IronFleet's non-verified implementation code is written in C#.

Verdi. Verdi's methodology to verify distributed systems relies on a *verified transformer* [52]. It verifies a server implementation of the Raft protocol (§2.2)[2] and seeks to prove its safety properties (linearizability). Verdi provides durability (operations are written to disk) and implements recovery (replicas can recover from a crash).

Verdi is verified and implemented using the Coq proof assistant [13]. It invokes Coq to translate its verified code into OCaml, which is then either interpreted by the OCaml interpreter or compiled into a binary by the OCaml compiler. Some of Verdi's non-verified code is written in OCaml.

Chapar. Chapar proposes a methodology to verify distributed systems with causal consistency semantics. It verifies a key-value store that implements the two causal consistency algorithms described in §2.2. Chapar seeks to prove the safety properties of both servers and clients.

[2] IronFleet and Verdi implement additional, simpler protocols. We consider these to be secondary and outside the scope of this paper.

Like Verdi, the Chapar server is implemented and verified using Coq and OCaml. Chapar also verifies the client application using model checking.

3. Methodology

This section describes the methodology we used in our study and discusses some of its limitations.

3.1 Scope

Our study analyzed two aspects of each verified distributed system:

1. Overall correctness. We studied the overall correctness of the server implementation (shim layer and verified code).[3] Thus, we did not restrict the analysis to verified components or verified properties.

2. Verification guarantees. We studied the specification and verification tools used to verify the systems. This analysis allowed us to understand the extent to which formal verification guarantees cover properties and components.

3.2 Analysis Techniques

We relied on the following methods to analyze the correctness of the implementations and their formal guarantees.

Analysis of code and documentation. We analyzed the verified systems' source code and specification. In addition, we leveraged existing documentation to understand their design. We identified assumptions the systems made and formulated hypotheses about missing or incorrect functions that could constitute bugs.

Testing of implementation. We tested the implementations using a network and file system fuzzer, and we developed test cases to check the correctness of different components. Furthermore, we applied traditional debugging techniques, such as debuggers and packet sniffers, to gain a better understanding of the implementations and to confirm or rebut our hypotheses throughout our study. We incorporated the testing tools we developed into our PK toolchain (§3.3).

Comparison of systems and interaction with developers. We cross-checked the different verified systems by checking whether bugs found in one such system also existed in the others. In addition, we checked whether bugs found in non-verified systems existed in the verified systems analyzed as well. We reported to developers of the verified systems the bugs we found using their issue trackers.

3.3 PK Testing Toolchain

Our analysis of the verified systems identified a series of specific bugs that impaired their overall correctness or verification guarantees. Importantly, these bug examples, which were gathered using the methods explained in §3.2, enabled us to develop the PK testing toolchain that *systematizes* the search for similar bugs. Figure 4 provides an overview of prescribed approaches to improve the reliability of verified systems. We adopted some of these approaches in our testing toolchain (§4.5, §5.2, and §6.2).

Shim layer
- Verifying additional components of the system (§4.4, §4.5, §8.2)
- Verifying resource usage and liveness properties (§4.4)
- Improving documentation of libraries (§4.4)
- Testing focused on the shim layer (§4.5)
- Testing implicit resource usage (§4.5)

Specification
- Proving specification properties (§5.2)
- Verifying applications using specifications of underlying layers (§5.2)
- Testing specifications (§5.2)

Verification tool
- Designing fail-safe verifiers (§6.2)
- Testing verifiers (§6.2)

Figure 4: An overview of prescribed approaches to improve the overall reliability of verified software.

3.4 Study Limitations

We focused our efforts on analyzing the source code of the implementations and the specification of the verified systems. Therefore, it is possible that our results could underrepresent bugs in other parts of the TCB.

As with other bug studies, it can be difficult to reason about the number of *false negatives*, bugs that may exist but were not found. This applies to bug studies that rely on bugs reported by users [17, 40, 50], bugs found by testing tools [6, 7, 12, 18], and bugs found through a mix of testing tools and manual inspection, like ours. Despite this challenge, we aimed to be systematic using two separate means. First, we cross-checked the bugs found in each verified system against the other verified systems. Second, we analyzed the verified systems by iteratively formulating and checking hypotheses, which included hypotheses based on bugs that were found in other non-verified distributed systems, such as those found by Scott et al. [47].

Regarding *false positives*, on the other hand, the fact that we reported the bugs and that nearly all of them were already either fixed or confirmed by the developers provided a degree of high confidence that these were indeed bugs. Our study analyzed a relatively small number of verified systems (3) and found a limited number of bugs (16); small values necessarily require care in generalizing results. Nevertheless, our study analyzed the largest number of verified implementations and the largest number of bugs, relative to all previous studies of which we are aware [55].

4. Shim Layer Bugs

We classified the shim layer bugs into three categories: (1) RPC implementation, (2) disk operations, and (3) resource limitations. All these bugs resulted from a discrepancy between the shim implementation and the expectation (also known as low-level specification) that the verified component held regarding the shim implementation.

[3] We focus on servers because they typically contain most of the complexity in distributed systems.

```
// Client A, B and C run on different servers
1: Client A: PUT("key", "NA")
2: Client A: PUT("key", "Request")
3: Client B: GET("key") = "Request"
4: Client B: PUT("key-effect", "Reply")

// Packet sent by request 1 is duplicated
5: Client C: GET("key-effect") = "Reply"
6: Client C: GET("key") = "NA"
```

Figure 5: Test case that violates causal consistency (Bug C1). Client C reads the effect event ("Reply") but not the cause event ("Request") that Client B read.

4.1 RPC Implementation

We found five shim layer bugs affecting the client-server and server-server communication. These bugs were all caused by mismatched assumptions about the network semantics, with respect to its failure model and limits, or the RPC input, with respect to its size and contents.

Bug V1: *Incorrect unmarshaling of client requests throws exceptions.*

The Verdi server used TCP to receive client requests but wrongly assumed that the `recv` system call would return the entire request in a single invocation. Because of this assumption, the server tried to unmarshal a partial client request, which caused it to throw an exception. The fix for this problem is to accumulate the received data in a buffer until the complete request is received.

Bug C1: *Duplicate packets cause consistency violation.*

We found that Chapar servers, which sent updates to each other through UDP, accepted duplicate packets and always reapplied the updates. As a result, we were able to construct a test case (Figure 5) that caused a client to see results that violate causal consistency [28]. Client C is able to see the effect of an event ("Reply") by reading "key-effect" but is unable to see the cause ("Request") by reading "key". In addition to violating causal consistency, accepting duplicate updates that are interleaved with other updates to the same key prevented monotonic reads, an important session guarantee property [51].

This bug resulted from the assumptions of Lloyd's algorithms [38] that were implemented by the server, which assume a reliable network. The updates sent between servers (reflecting the PUT operations from clients) contained a dependency vector that used Lamport clocks. Before applying a new update, the server checked that all causally preceding updates had already been applied. Unfortunately, given the use of UDP, this check is not enough in an unreliable network. An old update would satisfy this check and be reapplied, but applying it could overwrite later updates.

Bug C2: *Dropped packets cause liveness problems.*

The server shim layer implementation did not handle packet drops because it assumed that the network layer was reliable. Given that the server relied on UDP sockets to exchange updates, the practical result is that a single packet drop prevented clients on the receiver server from ever seeing the respective update. This problem was made worse because a single dropped packet could also prevent the clients from reading subsequent updates: the causal dependency check would prevent subsequent requests from being applied. The fix to this problem would be to implement retransmission and acknowledgment mechanisms.

```
// Initialize
PUT("key1", "value1")
PUT("key2", "value2")
PUT("key3", "value3")

// Inject GET("key2") request
GET("key1 - - \n132201621 216857 GET key2") = "value1"

// GET requests return wrong values
GET("key1") = "value2"   // Wrong value
GET("key2") = "value1"   // Wrong value
GET("key3") = "value2"   // Wrong value
```

Figure 6: Command injection vulnerability (Bug V2).

Bug V2: *Incorrect marshaling enables command injection.*

Figure 6 shows a sequence of requests that, when executed, allowed the client to cause the server to execute multiple commands by invoking a single command. As the example shows, this bug also causes subsequent requests to return incorrect results.

This bug resulted from incorrect marshaling; the server used meta-characters (newlines and spaces) to distinguish commands and command arguments but it did not escape the meta-characters. As a result, if the client invoked a command request with specially crafted arguments, it caused the RSM library to interpret that invocation as two or more distinct requests. In addition, after injecting commands, subsequent requests returned incorrect results because the client-server protocol expected each invocation to be followed by exactly one response (and had no other way to pair responses with replies). Further, due to this bug, some arguments crashed the server because they led to messages that did not comply with the format expected by the server, causing the marshaling function to throw an uncaught exception.

The fix to this problem would be to change, at *both ends*, the client-server communication protocol to ensure that any argument value can be sent by the client. This could be achieved either by escaping meta-characters or by adopting a length-prefix message format.

Bug C3: *Library semantics causes safety violations.*

Our tests demonstrated that, under certain conditions, the server could sent corrupted packets to other servers containing command arguments that were not provided by the client. This bug violated an expected safety property (namely, integrity) because the corrupted packets were ap-

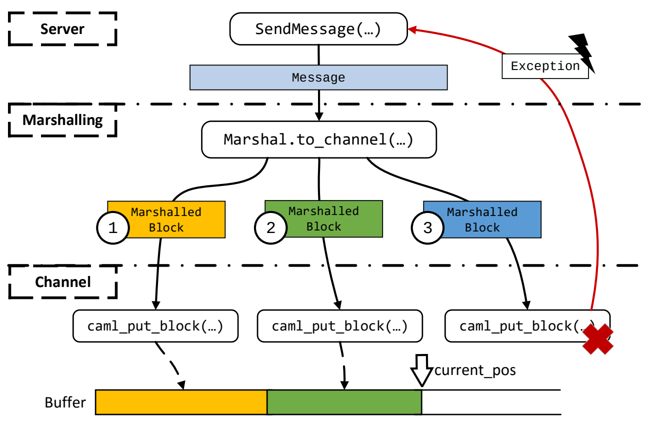

Figure 7: Exception while invoking the marshaling function left data in the internal buffers of the channel (Bug C3).

plied by the servers to their storage and were made visible to other users.

Interestingly, this bug resulted from the semantics of the OCaml library function `Marshal.to_channel()`, which Chapar uses to marshal messages sent between servers. It was triggered if the server tried to marshal and send a message that did not fit within the UDP packet limit. When this occurred, sending the packet would fail; more importantly, the headers and partial data would be kept in internal buffers of the OCaml library. In other words, the prefix of the message was internally buffered while the suffix was discarded. Our test case demonstrated that the buffered data, in turn, could be concatenated with subsequent requests to construct a packet that had the correct format but incorrect content.

As Figure 7 shows, the function `Marshal.to_channel()` broke down an OCaml object and serialized its subcomponents (which are the elements of a list in Chapar's case). After converting each subcomponent into a byte representation, the marshaling function invoked the channel write function, `caml_put_block()`. OCaml channels can have different types (e.g., UDP socket, TCP socket, or files). However, the channel write function internally buffered the writes and only wrote to the device (e.g., socket) if: (1) the buffer limit had been reached during the write, or (2) the developer explicitly flushed the buffer. If the channel attempted to write the buffer contents and failed, it would return, leaving the contents as they were. The error returned from the channel layer was caught by the marshaling function, which then returned to the code that invoked it; importantly, however, the prefix of the byte representation was left in the channel buffer and could be sent later due to other invocations.

Our example test case is complex, but we created simpler test cases that caused other problems. Besides causing servers to accept requests that no client issued, this bug can cause: requests to be silently discarded; large requests to not be sent when they exceed the UDP limit; and small requests to not be accepted by the receiver when they are concatenated with large requests.

In addition to reporting this bug to Chapar developers, we also reported it to OCaml developers. We did so because the current semantics of the OCaml library are difficult to use correctly, and this problem is not mentioned in the library function documentation. The OCaml developers confirmed the problem and, in response to our bug report, have been actively discussing possible workarounds and fixes.

One workaround discussed by OCaml developers is the following: (1) marshal the OCaml object into an external buffer, (2) check whether the length of the buffer is smaller than the UDP size, and (3) if it is smaller, then manually invoke the channel write channel function on the external buffer. Unfortunately, this workaround does not solve all the problems because of other types of errors that could prevent the write calls from succeeding (like those we discuss below in Bug V5). In practice, this bug may have a significant impact on verified systems reliability because of the number of such systems that use the OCaml library and may suffer from similar problems.

4.2 Disk Operations

We found three shim layer bugs related to disk operations. All these problems were caused by developers assuming that a single or set of disk operation(s) are atomic during crashes.

Bug V3: *Incomplete log causes crash during recovery.*

Bug V3 prevented replicas from recovering by causing them to repeatedly crash if the (disk) log were truncated. This problem was caused by the server code wrongly assuming that the entries in the log were always complete. This was not the case; the server used the `write` system call, which does not guarantee to write atomically when servers crash.

In a deployment situation, an administrator could overcome this problem by manually discarding the incomplete entry at the end of the log. This would be safe (i.e., not cause loss of data), assuming the rest of the server logic were correct because it would be equivalent to a server crash that occurs immediately before the write operation. Nevertheless, restoring service would require intervention from the administrator and a correct diagnosis.

Bug V4: *Crash during snapshot update causes loss of data.*

This bug caused the server to lose data due to defective code that wrote the disk snapshots. Verdi's shim layer implemented a snapshotting mechanism that, at every 1000 events, executed the following tasks: (1) wrote a new snapshot, (2) removed any previous snapshot, and (3) truncated the log. Unfortunately, data loss could occur because of the *unsafe* order in which the three tasks were executed. In particular, the implementation truncated the existing disk snapshot before it safely wrote the new one to disk. Thus, a crash between the truncation and write operations led to loss of data.

Because this bug caused data loss, it is more serious than V3, as the administrator cannot easily recover from the problem. Fixing this bug would require consistently ensuring that durable information remains on the disk; in particular, the old snapshot should be deleted *after* the new snapshot is written to disk.

Bug V5: *System call error causes wrong results and data loss.*

This bug affected servers that were recovering and was ultimately caused by the server not correctly distinguishing between situations where there was both a log and snapshot and those where there was only a log. The latter occurred if the server crashed before it executed 1000 events (i.e., when the first snapshot is created).

During recovery, the server tried to read the snapshot file and if it failed to open it, the server wrongly presumed that the snapshot file did not exist. In practice, this meant that a transient error returned by the open system call, such as insufficient kernel memory or too many open files, caused the server to silently ignore the snapshot.

Our testing framework generated a test case that caused the servers to silently return results as if no operations had been executed before the server crashed, even though they had. This bug could also lead to other forms of safety violations given that servers discard a prefix of events (the snapshot) but read the suffix (the log), potentially passing validation checks. Further, the old snapshot could be overwritten after a sufficient number of operations were executed.

4.3 Resource Limits

This section describes three bugs that involve exceeding resource limits.

Bug V6: *Large packets cause server crashes.*

The server code that handled incoming packets had a bug that could cause the server to crash under certain conditions. The bug, due to an insufficiently small buffer in the OCaml code, caused incoming packets to truncate large packets and subsequently prevented the server from correctly unmarshaling the message.

More specifically, this bug could be triggered when a follower replica substantially lagged behind the leader. This could occur if the follower crashed and stayed offline while the rest of the servers processed approximately 200 client requests. Then, during recovery, the follower would request the list of missing operations, which would all be combined into a single large UDP packet that exceeded the buffer size and crashed the server.

The fix to this problem was to simply increase the size of the buffer to the maximum size of the contents of a UDP packet. However, bugs Bug V7 and Bug V8, which we describe next, were also related to large updates caused by lagging replicas but these are harder to fix.

```
let rec findGtIndex orig_base_params raft_params0
                    entries i =
  match entries with
    | [] -> []
    | e :: es ->
      if (<) i e.eIndex
        then e :: (findGtIndex orig_base_params
                              raft_params0 es i)
        else []
```

Figure 8: OCaml code, generated from verified Coq code, that crashed with a stack overflow error (Bug V8). In practice, the stack overflow was triggered by a lagging replica.

Bug V7: *Failing to send a packet causes server to stop responding to clients.*

Another bug we found prevented servers from responding to clients when the leader tried to send large packets to a lagging follower. The problem was caused by wrongly assuming that there was no limit on the packet size and by incorrectly handling the error produced by the *sendto* system call. This bug was triggered when a replica that lagged behind the leader by approximately 2500 requests tried to recover.

In contrast to Bug V6, this bug was due to incorrect code on the sender side. In practice, the consequence was that a recovering replica could prevent a correct replica from working properly. The current fix applied by the developers mitigates this bug by improving error handling, but it still does not allow servers to send large state.

Bug V6 and Bug V7 were the only two that we did not have to report to developers because the developers independently addressed the bugs during our study.

Bug V8: *Lagging follower causes stack overflow on leader.*

After applying a fix for Bug V6 and Bug V7, we found that Verdi suffered from another bug that affected the sender side when a follower tried to recover. This bug caused the server to crash with a stack overflow error and was triggered when a recovering follower lagged by more than 500,000 requests.

After investigating, we determined that the problem was caused by the recursive OCaml function `findGtIndex()` that is generated from verified code. This function, which constructed a list of missing log entries from the follower, was executed before the server tried to send network data. This was an instance of a bug caused by exhaustion of resources (stack memory).

Figure 8 shows the generated code responsible for crashing the server with the stack overflow. This bug appeared difficult to fix as it would require reasoning about resource consumption at the verified transformation level (§2.3). It also could have serious consequences in a deployed setting because the recovering replica could iteratively cause all servers to crash, bringing down the entire replicated system.

4.4 Summary of Findings

Finding 1: *The majority (9/11) of shim layer bugs caused servers to crash or hang.*

Bugs that cause servers to crash or stop responding are particularly serious, especially for replicated distributed systems that have the precise goal of *increasing* service availability by providing fault-tolerance. Therefore, proving liveness properties is particularly important in this class of systems to ensure the satisfaction of user requirements.

Finding 2: *Incorrect code involving communication caused 5 of 11 shim layer bugs.*

Surprisingly, we concluded that extending verification efforts to provide strong formal guarantees on communication logic would prevent half of the bugs found in the shim layer, thereby significantly increasing the reliability of these systems. In particular, this result calls for composable, verified RPC libraries.

Finding 3: *File system operations were responsible for 3 of 11 shim layer bugs.*

File system semantics are notoriously difficult for developers to understand, especially in a crash-recovery model. The bugs we found were all located in the unverified recovery component of Verdi, the only system we studied that implements durability. This result confirms the importance of recent efforts to formalize file system semantics and verify file systems [3, 11, 49].

Furthermore, we found the official OCaml library reference documentation to be surprisingly terse and devoid of content. For instance, many functions provided by the basic operations module (`Pervasives` module) were documented with three or fewer sentences. This problem also affects file system functions which have particularly complex semantics due to possible error conditions. We look to have libraries, especially those relied upon by verified systems, with complete and accurate documentation.

Finding 4: *Three of 11 shim layer bugs were related to resource limits.*

`Bug V6` and `Bug V7` were caused by unreasonable assumptions about: (1) the buffer size limits, and (2) the maximum size of UDP packets. This suggests that explicitly reasoning about different types of resource limits is important for the reliability of systems. For example, IronFleet verified that message sizes always fit within a UDP packet, albeit, in this case, the state machine size was bounded to fit the UDP packet size. Ideally, verification should reason about resources and ensure reasonable bounds on their usage.

Similarly, `Bug V8`, caused by assumptions on stack memory size, confirms that reasoning about resource limits is vital to prevent potentially serious bugs. In this case, the verification checks did not reason about the consumption of stack memory, which is an implicit resource.

Finding 5: *No protocol bugs were found in the verified systems.*

None of the bugs we found were due to mistakes in the implementation of distributed protocols (e.g., Paxos, Raft), which are well known to be complex and difficult to implement correctly. Our results suggest that verification does improve the reliability of the verified components: all the bugs described in this section were located in the unverified shim layer code, in the unverified shim layer library (`Bug C3`), or in the shim layer runtime (`Bug V8`). In fact, we found no shim layer bugs in IronFleet, which is the system studied with fewest unverified components.

4.5 PK Toolchain: Preventing Shim-layer Bugs

Our results demonstrate, with concrete examples, that overall system correctness crucially depends on making correct assumptions regarding the shim layer, which represents a small subset of the entire TCB. Motivated by these results, this subsection argues for the adoption of testing approaches that complement verification techniques by testing non-verified components that interface with verified ones. More specifically, as part of the verification methodology, the *shim layer* should be independently tested to detect bugs that arise from possible mismatches between assumptions made by verified code and the properties provided by the shim layer implementation.

We built several test cases that specifically targeted Verdi's *shim layer*; we incorporated these into our PK testing toolchain. Our test cases consist of three testing applications that we implemented in OCaml, which directly linked with Verdi's shim layer (i.e., excluded the verified code). Each of these applications checks a different property that was assumed by the verified code: (1) the integrity of messages sent between servers, (2) the integrity of messages sent between clients and servers, and (3) the integrity of the abstract state machine log during recovery. Even though neither Verdi nor Chapar aimed to prove liveness properties, using timeout mechanisms, our toolchain tests for liveness, which is necessary to detect serious classes of bugs, such as those that crash or hang servers.

In addition to test cases, we implemented a file system and network fuzzer that transparently, using `LD_PRELOAD`, modifies the environment, unmasking bugs that would otherwise remain undetected. For instance, our fuzzer emulates different behaviors permitted by OS semantics, such as reordering UDP packets, duplicating UDP packets, executing non-atomic disk writes, and producing spurious system call errors. Our experiments demonstrated that our testing infrastructure detects all shim layer bugs found, except for `Bug V8` which is caused by implicit resource usage.

Using formal verification techniques, reasoning about implicit memory usage and verifying that it is guaranteed to be within given bounds would prevent bugs like the stack overflow bug [8]. However, it is unclear how to apply these techniques to verify the resource usage of distributed systems. A middle-ground approach would be to design test cases using tools such as our fuzzer and to monitor the resource usage of verified components, checking whether it matches expected resource consumption models.

5. Specification Bugs

This section discusses two bugs that we found in the specification of the systems analyzed. Neither bug caused the current implementation of servers to crash or otherwise produce incorrect results (unlike the bugs discussed in §4); however, specification bugs partially void verification guarantees. In practice, both bugs would allow distributed system implementations that return incorrect results to pass verification checks.

Bug I1: *Incomplete high-level specification prevents verification of exactly-once semantics.*

We found that the high-level specification of IronFleet's RSM server did not ensure linearizability because it did *not specify* that the implementation had exactly-once semantics even though it *did implement* this functionality.

We demonstrated the problem by constructing a patch that modified the server. The patch disabled the deduplication functionality, which had been implemented, by modifying only seven lines of the implementation. Notably, the patched implementation still verified. Our patch demonstrates that an implementation bug could prevent the servers from providing exactly-once semantics and this problem would not be detected by the verification process.

RSM libraries usually implement exactly-once semantics by using per-client sequence numbers to identify the request. This mechanism lets servers distinguish duplicate requests, which can occur for several reasons. Duplicate requests can arrive at servers due to network semantics (i.e., the network can duplicate packets, and clients must retransmit packets if they suspect lost packets) and the fault model (clients need to resend requests if they suspect that a server might have crashed).

We reported this bug to developers, who confirmed that the specification did not provide exactly-once semantics. However, they stated that: their understanding of linearizability does not include exactly-once semantics; the state machine could implement de-duplication; and they had been aware of the absence of exactly-once semantics in the specification. In response to our bug report, the developers added a comment to the source code, clarifying that the specification does not cover exactly-once semantics.

We consider this a bug because generally applications expect replicated state machine libraries to provide exactly-once semantics and because the absence of this guarantee would cause incorrect results for applications that expect it [31]. Regardless of the definition of linearizability, this example demonstrates well the need to be clear about the exact specification.

```
(* Client 0 *)
   put (Photo, "NewPhoto");;
   put (Post, "Uploaded");;

(* Client 1 *)
   post <- get (Post);;
   if (string_dec post "Uploaded") then
      photo <- get (Photo);;
-     if string_dec post "" then       // Original
+     if string_dec photo "" then      // Fix
        fault
      else
        skip
```

Figure 9: Patch to fix Bug C4 in a client example of Chapar (`Clients.v`). The original Coq code is equivalent to `assert(post != "")`. Instead, the assertion should check that *photo* is non-null when the post is non-null.

Bug C4: *Incorrect assertion prevents verification of causal consistency in client.*

The client application `prog_photo_upload` had a bug in an assertion (see Figure 9). The check simply asserted the condition of the conditional branch (the if-condition). Therefore, the assertion always evaluated to true, which defeated its purpose. This bug was easily fixed by asserting that `photo` (instead of `post`) is non-null.

Besides verifying the server, Chapar verified clients using model checkers to detect causal consistency violations. In this example, no other assertion correctly performed the causal consistency check in the client application. Of the three systems analyzed, only Chapar verified clients, and this was the only bug that we found in a client. All other bugs were discovered in servers or in verification tools.

5.1 Summary of Findings

Finding 6: *Incomplete or incorrect specification can prevent correct verification.*

Even if verification tools are correct, specifications must be correct for verification to deliver its promise. Bug I1 showed an example of a subtle specification problem that could result in application bugs. Regardless of the exact definition of linearizability, the existence of different interpretations is already sufficient to lead to application bugs. In addition, we note that it was not initially obvious to us that the IronFleet specification did not guarantee exactly-once semantics. To complement the current verification methodologies, we need techniques to test specifications.

5.2 PK Toolchain: Preventing Specification Bugs

As we have discussed, verification crucially relies on the correctness of the specification. This section discusses two types of techniques that seek to validate *specifications* themselves, and we report our experience on applying them to IronFleet. The first technique, *negative testing*, tests by actively introducing bugs into the implementation and confirming that the specification can detect them during verification. The second technique, *specification checking*, relies on proving properties about the specification.

Negative testing. We built a testing tool that automatically modifies the implementation source code. Its goal is to help developers check whether the source code implements more functions than required by the specification, a problem that we found in IronFleet (`Bug I1`). This could indicate an incomplete specification.

Our tool performs three types of simple transformations to Dafny source code that disables code: (1) changing the values of sub-conditions, (2) preventing updates to structures, and (3) commenting out entire statements. These transformations sufficed to generate the changes necessary to modify the implementation in a manner functionally equivalent to the patch we discussed for `Bug I1`. Nevertheless, like the types of transformations used in mutation testing [15], other types of transformations could be considered, such as removing parts of statements or modifying statements in different ways.

Our tool requires developers to specify the functions to be tested and an upper bound on the number of transformations per function. Our evaluation shows that when applying it to `HandleRequestBatchImpl`, `LProposerProcessRequest`, and `ProposerProcessRequest` in IronFleet, the tool requires on average 377, 18, and 8015 iterations to generate the patches, respectively; these modified functions disabled the de-duplication function but still passed the verification checks. Currently, our tool simply picks random transformations and applies them at random source code locations. If not guided by the developer, this process could be expensive given that several matching modifications must be made to pass the verification checks. For instance, IronFleet relies on a consistent protocol and implementation layer; therefore, to pass the verification checks, the transformed source code needs matching modification at both layers.

Specification checking. A non-programmatic approach to find specification bugs relies on proving specification properties. We explored this approach by writing a test case in Dafny that combined the specification of IronFleet with the specification of the counter application it provides. By composing the two, we constructed a machine-checked lemma that confirmed the possibility of reaching any counter state after executing a single counter operation. This formally confirmed that the specification did not prevent duplicate execution of operations (`Bug I1`).

Ideally, such tests should be built by developers that did not write the specification—adding a level of redundancy—and should be reused across projects. Alternatively, verifying the implementation of applications and formally composing it with the verified distributed system library layer could also increase confidence in the correctness of the distributed system specification. However, it would still leave open the correctness of the top-most specification.

6. Verification Tool Bugs

This section analyzes four bugs we found in verification tools. Like specification bugs (§5), none of these bugs crashed the server implementation or otherwise produced incorrect results. However, they invalidated verification guarantees. In general, verification tool bugs can cause incorrect server implementations to pass the verification check even if the specification is correct. All the problems reported were found in either auxiliary tools or at the perimeter of the verifier's core functionality.

`Bug I2:` *Prover crash causes incorrect validation.*

A bug in IronFleet's tools caused the verifier to falsely report that *any* program passed verification checks, including programs that asserted false. In addition, the verifier built binaries for incorrect programs.

This bug was caused by a defect in NuBuild, a component of the verification infrastructure like Unix `make`. NuBuild repeatedly invoked Dafny for each source code file to verify and, if it verifies, compile it. For each invocation, NuBuild parsed the output produced by Dafny and aborted the build process if it detected an error; otherwise, it continued verifying and eventually built the binary.

Unfortunately, NuBuild incorrectly parsed the output of Dafny (see Figure 10). Dafny invoked both Boogie (the verifier) and Z3 (the prover) and emitted a diagnostic message regarding the verification process and an additional message if the prover crashed. NuBuild's parsing function mistakenly terminated after consuming just the first message even if there were additional message regarding a prover crash. We found this bug because it was triggered by another bug that caused the prover, Z3, to crash (`Bug I4`). However, this bug could also be triggered in other situations that caused Z3 to abruptly terminate, such as insufficient memory or other system errors.

In this case, several aspects combined to increase the potential for an unsuspecting developer to be tricked by the incorrect verifier: (1) no error or warning was made visible to the user, (2) the verifier built the program binary (and updated it if the source code changed), and (3) when the bug was triggered, the build duration did not change drastically.

This bug is not necessarily triggered on *every* verifier invocation. Instead, it can be triggered by "transient" problems, such as the termination of the prover by the OS due to insufficient resources (e.g., memory). Furthermore, because verification is computationally expensive and slow, NuBuild can offload verification to a cluster of remote machines. Under this setting, Dafny will run on many different machines and potentially different NuBuild installations. This aggravates the impact of the NuBuild bug because the verification process could sporadically and silently fail.

```
noTimeouts = new Regex("Dafny␣program␣verifier␣finished␣
    with␣(\\d*)␣verified,␣(\\d*)␣errors*");
proverDied = new Regex("Prover␣error:␣Prover␣died");

void parseOutput(out verificationFailures,
                 out parseFailures) {
  ...
  match = noTimeouts.Match(output);
  if (match.Success) {
    verificationFailures = match.Groups[2];
    return;    // <=== Returns when the prover dies
  }
  ...
  match = proverDied.Match(output);
  if (match.Success) {
    parseFailures = 1;  // <=== Not executed
    return;
  }
```

Figure 10: Simplified version of the NuBuild code responsible for Bug I2. The proverDied regular expression was never matched because the other regular expression matched first and returned from parseOutput(). Furthermore, Dafny only included errors *found* by the prover in the error count matched by the first regular expression, not errors *executing* the prover.

```
 void WaitForOutput() {
   ...
   try {
     outcome = thmProver.CheckOutcome(cce.NonNull(handler)
     );
   }
   catch (UnexpectedProverOutputException e) {
     outputExn = e;
   }
+  catch (Exception e) {
+    outputExn = new UnexpectedProverOutputException(e.
     Message);
+  }
```

Figure 11: Patch to fix Bug I3 in Boogie source code (Check.cs). Adding a general exception handler caught all exceptions thrown while the prover was executing.

This bug was confirmed by developers and fixed with our patch proposal. Our patch changed the order of the regular expressions used in the parsing function parseOutput().

Bug I3: *Signals cause validation of incorrect programs.*

When executing the verifier on Linux and macOS, we found that if the user sent a SIGINT signal, verification was interrupted (as expected), but Dafny misleadingly reported that no error occurred, and that the files were verified. This bug is similar to Bug I2 except that the verifier did not build the binary.

This bug was caused by Boogie (a low-level verifier invoked by Dafny) not handling correctly the exception thrown when the prover is interrupted. The exception handling code handled only UnexpectedProverOutputException exceptions, but SIGINT threw a different type of exception. Boogie's developers have fixed the problem by patching the verifier (Figure 11).

Bug I4: *Incompatible libraries cause prover crash.*

The prover included in the IronFleet distribution failed to execute, with an error (0xc00000007b). The problem was caused by the inclusion of incompatible libraries in the package; the included z3.exe binary was built for 64-bit architectures, while the binary libraries included were built for 32-bit architectures.

The problem can be more serious than it appears because the prover was not invoked directly by users; rather, it was invoked by other verifier components, some of which had defective error detection mechanisms. In fact, this problem triggered Bug I2. The solution to this bug is simple: after we reported it, the developers fixed it by updating the libraries with matching architectures.

6.1 Summary of Findings

Finding 7: *There were critical bugs in current verification tools that could compromise the verification process.*

Verification tools are complex and increasingly automated. In addition, they evolve quickly given their growing popularity. Thus, it is not surprising that they contain bugs. However, it is surprising that we found a combination of bugs (Bug I2 and Bug I4) that could mislead unsuspecting developers, potentially with serious impact on the correctness of verified programs. The correctness of verification tools becomes even more relevant if the programmer is an adversary [43].

Finding 8: *All critical verifier bugs were caused by functions that were not part of the core components of the verifier.*

Surprisingly, the critical bugs found in verification tools (Bug I2 and Bug I3) were not caused by the verifier's core components (i.e., the parts that reason about proofs). Instead, they were found in auxiliary tools (Bug I2) and in the verifier's exception handling (Bug I3). These results call for better methodologies to *design* and *compose* the various components of verification infrastructures to ensure either correct or, at least, fail-safe operation (e.g., reporting a verification error rather than success if there is any exception).

6.2 PK Toolchain: Preventing Verification Tool Bugs

As the bugs we found attest, verification infrastructures can contain serious bugs that potentially compromise the verification process. The problem of verifier correctness has been studied in the context of traditional verifiers, and several techniques have been proposed, including verified code extraction [42].

We developed verifier test cases, consisting of sanity checks, that deliberately caused the verification process to fail under different scenarios. This process determined whether the verification infrastructure could detect certain classes of verification problems. Although simple, these tests enabled our testing toolchain to detect the bugs that affected NuBuild and Z3. We argue that sanity checks should be con-

	Protocol	Consistency	Code size
LogCabin	Raft	Linearizability	27K lines of C++
ZooKeeper	Primary-backup	Linearizability	142K lines of Java
etcd	Raft	Linearizability	269K lines of Go
Cassandra	Paxos	Linearizability	374K lines of Java

Figure 12: Summary of the unverified distributed systems.

ducted to the verification infrastructure and under its actual execution environment—at the very least, when generating the system binaries that will be deployed.

Interestingly, there has been significant recent interest in increasing the level of automation of modern verifiers. As a consequence, recent verifiers have become extremely complex. Whereas traditional verifiers (e.g., SAT solvers) relied on relatively simple artifacts, the Dafny verifier, for example, relies on Boogie, which in turn relies on the Z3 SMT solver, which itself is more complex than traditional verifiers. Furthermore, to mitigate the impact on verification time caused by the high degree of automation, verifiers are becoming distributed systems that rely heavily on caching across multiple machines. This trend suggests that some of the techniques and methodologies that have been developed to improve the robustness of other systems should now be considered for modern verifiers.

7. Response from Developers

The developers confirmed the existence of all problems we reported. However, as discussed earlier, they did not consider Bug I1 to be a bug because of their different understanding of linearizability. The developers agreed to apply the patch we proposed for Bug C4. Regarding Bug C1-3, developers stated that causal consistency is guaranteed if the explicit communication properties in the semantics hold and suggested different fixes to improve the implementation (not using UDP, modeling reordering, using acknowledgments, and limiting input size). Bug V1-5 and Bug V8 were confirmed by the developers. As shown in Figure 3, the rest of the bugs have already been fixed.

8. Toward "Bug-Free" Distributed Systems

To gain insight into how we can move towards "bug-free" distributed systems, we tried to understand what are the components and sources of reliability problems in modern *deployed* distributed systems. Most of these systems implement a large set of features that have yet to be verified in any distributed systems analyzed, although some of these features are particularly complex (thus potentially bug prone) and important for real-world users.

8.1 Methodology

Our analysis relied on the inspection of reports of *known bugs* by sampling bugs from the issue trackers of each unverified system (Figure 12). Due to the large volume of bug reports, we restricted our analysis to confirmed reports open between March 2015 and March 2016 (a 1-year span). In addition, we discarded low-severity bugs and bugs that did not affect functional correctness. Figure 13 presents an overview of the results which support the findings in §8.2.

	LogCabin	ZooKeeper	Etcd	Cassandra	Total
Communication	4	1	3	9	17
Recovery	0	1	0	7	8
Logging / snapshot	5	5	6	5	21
Protocol	1	1	2	8	12
Configuration	1	2	0	0	3
Client library	1	23	11	7	42
Reconfiguration	1	6	8	17	32
Management tools	1	22	21	116	160
Single-node storage	1	18	11	200	230
Concurrency	3	1	2	18	24
Total	23	80	65	387	555

Figure 13: Sample of known bugs from the bug reports of *unverified* distributed systems.

Limitations. We do not intend to compare the bug count between the verified and unverified systems due to their significant differences. These unverified systems are not research prototypes; they implement numerous and complex features, have been tested by innumerable users, and were built by large teams. Furthermore, the analysis methodologies differ because the scale of the unverified system would make it impractical for us to manually find undiscovered bugs, as we did for verified systems. Instead of aiming at a direct comparison, our analysis of unverified systems was motivated by the need to understand how future verification efforts can improve the reliability and robustness of real-world distributed systems.

8.2 Results

Finding 9: *No protocol bugs were found in verified systems, but 12 bugs were reported in the corresponding components of unverified systems.*

This result suggests that recent verification methods developed for distributed systems improve the reliability of components that, despite decades of study, are still not implemented correctly in real-world deployed systems. In fact, *all* unverified systems analyzed had protocol bugs in the one-year span of bug reports we analyzed.

Finding 10: *Most of the bugs in unverified systems were found in management (160) and storage layers (230).*

In part due to optimizations, the complexity of management tools and storage layers explains the unreliability of these components. This result strongly suggests that the application of verification techniques to these tools and layers could significantly improve the reliability of distributed systems. Interestingly, much recent interest focused on the verification of file systems [11, 48]. Our observations support the interest in this research direction.

Finding 11: *A total of 24 bugs in these systems were caused by multi-threaded concurrency.*

Only 4.3% of the bugs (24 out of 555) reported in deployed systems were due to local concurrency. A closer analysis of these bug reports showed that almost all concurrency bugs were in the storage layer. This relatively low number could be caused by the absence of concurrency or the use of coarse-granularity concurrency mechanisms in most components. Another contributing factor could be that concurrency bugs are often under-reported [40].

Although real-world distributed systems often rely on concurrency, none of the verified systems that we analyzed implemented multi-threaded concurrency. In fact, the verification of shared-memory concurrent software, in general, is an active area of research that is considered extremely challenging [21, 25]. Thus, it remains unclear how researchers will address this challenge in the context of complex distributed systems and, accordingly, which testing techniques should be adopted.

8.3 Discussion

Correctly writing complex software is hard for developers. In traditional unverified systems, a *single* mistake made by developers when writing code can lead to serious bugs that immediately compromise the correctness of the application, causing crashes or incorrect results. Efforts verifying implementations significantly improve this by adding a level of redundancy.

Verified components can only fail, regarding verified properties, if developers introduce both an implementation bug *and* a verification bug (i.e., specification or verifier bug). Furthermore, those two bugs have to match: the verification bug has to cause the verification process to miss the implementation bug. This extra level of redundancy helps explain why we did not find any protocol-level bugs in any of the verified prototypes analyzed, despite that such bugs are common even in mature unverified distributed systems. Next we discuss different paths to improve the reliability of verified components that are protected by this redundancy and unverified components that remain vulnerable:

Verifiers. We believe the routine application to verifiers of general testing techniques (e.g., sanity checks, test-suites, and static analyzers) and the adoption of fail-safe designs should become established practices. Due to the reliance on increasingly complex SMT solvers and caching mechanisms and because verifiers are becoming distributed systems, testing and correctly implementing verifiers is expected to become increasingly challenging. This increased complexity calls for the development of scalable testing techniques and improved verifier designs.

Specification. In addition to verifier bugs, specification bugs could invalidate verification guarantees. Proving properties about the specification or reusing specifications are two important ways to increase the confidence that they are correct. The latter is likely to occur naturally with the expansion of verification to other systems but the former would require the adoption of best practices that mandate the inclusion of test cases for specifications.

Shim layer. As our study demonstrates, bugs in non-verified components, such as the shim layer, remain a serious threat to the overall reliability of systems. Because these components are not covered by verification guarantees, a single implementation bug could compromise the overall system correctness. Furthermore, the shim layer is often not reused across projects—all surveyed verified systems had custom-built shim layers. Therefore, existing shim layers are likely to contain undiscovered bugs. Building reusable and well documented shim layers that are applicable to different applications would contribute to improve the reliability of verified systems. In addition, formally specifying the properties of shim layer, expected by the verified components, allows testing tools, as we showed with PK toolchain, to test the properties of the shim layer without having to test the verified components. Isolating the unverified components could significantly improve the scalability of testing, as compared with testing unverified systems, by reducing the amount of code that needs to be tested and by ensuring the required properties are clearly defined.

9. Related Work

Much work has been done to analyze the correctness of unverified systems [10, 12, 18, 35, 39, 40, 46, 50]. In stark contrast, Yang et al. [55] conducted the only other study, to our knowledge, that analyzed the correctness of a *formally verified implementation*. By testing 11 compilers, they found more than 325 bugs, 11 of which were located in a verified compiler (CompCert [33]) [56]. Like our study, their work concluded that verification, while effective in reducing compiler bugs, does not replace testing—specifications are complex and seldom check end-to-end guarantees. In contrast to Yang's study, our study targets a different class of verified implementations. Interestingly, we found examples of problems in the verification tools themselves that their study did not uncover.

The importance of distributed systems has prompted significant work on analyzing and improving their reliability. For instance, Yuan et al. [57] sampled and studied 198 bugs in five popular implementations of distributed systems. Their results showed that many serious bugs can be detected by testing the error-checking code. Guo et al. [20] studied cascading recovery failures that can bring down entire distributed systems. In the context of minimizing execution traces, Scott et al. [47] found several bugs in an unverified implementation of the Raft protocol.

The verification of distributed system protocols has been an important line of research with a long history. For instance, many protocol proposals provide a pen-and-paper proof: the RSM protocols (e.g., Paxos [30], Raft [45], and

PBFT [9]) are notable examples. To prevent mistakes in pen-and-paper proofs [58], others have gone further and proposed machine-checked proofs [41]. As an alternative to proof-based methods, bounded model-checking techniques have been leveraged to increase confidence in the correctness of distributed systems [26, 53, 54, 58]. More recently, verified distributed system implementations, which we studied, have been proposed to extend formal guarantees to actual implementations [22, 34, 52].

10. Conclusion

This work presents the first comprehensive study on the correctness of formally verified implementations of distributed systems. Our study found 16 bugs that were caused by a wide-range of incorrect assumptions. We thoroughly analyzed these bugs and their underlying causes, which suggest that only a small fraction of the TCB was responsible for these problems; hence, this subset should be the focus of special attention. Our analysis suggested that verification was effective at preventing protocol bugs that occur in unverified systems. We conclude that verification, while beneficial, posits assumptions that must be tested, possibly with testing toolchains similar to the PK toolchain we developed.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Neeraj Suri, for their feedback. In addition, we are grateful to the authors of the verified systems studied for their help in analyzing our reports and clarifying our questions. In particular, we thank Manos Kapritsos, Mohsen Lesani, Jay Lorch, Zachary Tatlock, James R. Wilcox, and Doug Woos, with whom we had insightful conversations that greatly enhanced this paper. This work was supported in part by DARPA under contract FA8750-16-2-0032.

References

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[3] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. Murray, G. Klein, and G. 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)*, pages 175–188, Atlanta, GA, Apr. 2016.

[4] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal methods for Components and Objects*, pages 364–387. Springer, 2005.

[5] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1): 39–59, Feb. 1984.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS)*, pages 322–335, Alexandria, VA, Oct.–Nov. 2006.

[7] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 209–224, San Diego, CA, Dec. 2008.

[8] Q. Carbonneaux, J. Hoffmann, T. Ramananandro, and Z. Shao. End-to-end verification of stack-space bounds for C programs. In *Proceedings of the 2014 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–281, Edinburgh, UK, June 2014.

[9] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 173–186, New Orleans, LA, Feb. 1999.

[10] H. Chen, Y. Mao, X. Wang, D. Zhou, N. Zeldovich, and M. F. Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems*, Shanghai, China, July 2011. 5 pages.

[11] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. 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, Oct. 2015.

[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 73–88, Chateau Lake Louise, Banff, Canada, Oct. 2001.

[13] Coq development team. *Coq Reference Manual, Version 8.4pl5*. INRIA, Oct. 2014. http://coq.inria.fr/distrib/current/refman/.

[14] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Budapest, Hungary, Mar.–Apr. 2008.

[15] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, Apr. 1978. ISSN 0018-9162.

[16] R. W. Floyd. Assigning meanings to programs. In *Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, volume 19, pages 19–31, 1967.

[17] P. Fonseca, C. Li, V. Singhal, and R. Rodrigues. A study of the internal and external effects of concurrency bugs. In *Proceedings of the 40th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 221–230, Chicago, IL, June 2010.

[18] P. Fonseca, C. Li, and R. Rodrigues. Finding complex concurrency bugs in large multi-threaded applications. In *Proceedings of the ACM EuroSys Conference*, pages 215–228, Salzburg, Austria, Apr. 2011.

[19] S. J. Garland and N. A. Lynch. Using I/O automata for developing distributed systems. *Foundations of Component-Based Systems*, 13:285–312, 2000.

[20] Z. Guo, S. McDirmid, M. Yang, L. Zhuang, P. Zhang, Y. Luo, T. Bergan, P. Bodik, M. Musuvathi, Z. Zhang, and L. Zhou. Failure recovery: When the cure is worse than the disease. In *Proceedings of the 14th Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Ana Pueblo, NM, May 2013.

[21] A. Gupta, C. Popeea, and A. Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL)*, pages 331–344, Austin, TX, Jan. 2011.

[22] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[23] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

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

[25] J. Hoenicke, R. Majumdar, and A. Podelski. Thread modularity at many levels: A pearl in compositional verification. In *Proceedings of the 44th ACM Symposium on Principles of Programming Languages (POPL)*, pages 473–485, Paris, France, Jan. 2017.

[26] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 243–256, Cambridge, MA, Apr. 2007.

[27] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, M. Norrish, R. Kolanski, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, Big Sky, MT, Oct. 2009.

[28] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[29] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.

[30] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[31] C. Lee, S. J. Park, A. Kejriwal, S. Matsushita, and J. Ousterhout. Implementing linearizability at large scale and low latency. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, pages 71–86, Monterey, CA, Oct. 2015.

[32] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 348–370, Dakar, Senegal, Apr.–May 2010.

[33] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009.

[34] M. Lesani, C. J. Bell, and A. Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)*, pages 357–370, St. Petersburg, FL, Jan. 2016.

[35] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 289–302, San Francisco, CA, Dec. 2004.

[36] B. Liskov. Primitives for distributed computing. In *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*, pages 33–42, Pacific Grove, CA, Dec. 1979.

[37] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D^{3}S: Debugging deployed distributed systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 423–437, San Francisco, CA, Apr. 2008.

[38] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 313–328, Lombard, IL, Apr. 2013.

[39] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu. A study of Linux file system evolution. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–44, San Jose, CA, Feb. 2013.

[40] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 329–339, Seattle, WA, Mar. 2008.

[41] T. Lu. Formal verification of the Pastry protocol using TLA+. In *Proceedings of the 1st International Symposium on Dependable Software Engineering: Theories, Tools, and Applications*, pages 284–299, Nov. 2015.

[42] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, Jan. 1997.

[43] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI)*, pages 229–243, Seattle, WA, Oct. 1996.

[44] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *Computer*, 24(8):52–60, Aug. 1991.

[45] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference*, pages 305–319, Philadelphia, PA, June 2014.

[46] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *IEEE*

Transactions on Software Engineering, 31(4):340–355, 2005.

[47] C. Scott, V. Brajkovic, G. Necula, A. Krishnamurthy, and S. Shenker. Minimizing faulty executions of distributed systems. In *Proceedings of the 13th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 291–309, Santa Clara, CA, Mar. 2016.

[48] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, Nov. 2016.

[49] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, Savannah, GA, Nov. 2016.

[50] M. Sullivan and R. Chillarege. A comparison of software defects in database management systems and operating systems. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 475–484, 1992.

[51] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. W. Welch. Session guarantees for weakly consistent replicated data. In *Proceedings of the 3rd IEEE International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 140–149, Washington, DC, Sept. 1994.

[52] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 357–368, Portland, OR, June 2015.

[53] M. Yabandeh, N. Knežević, D. Kostić, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the 5th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 229–244, San Francisco, CA, Apr. 2008.

[54] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MoDist: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI)*, pages 213–228, Boston, MA, Apr. 2009.

[55] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 2011 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 283–294, San Jose, CA, June 2011.

[56] X. Yang, Y. Chen, E. Eide, and J. Regehr. Csmith/BUGS_REPORTED.TXT. https://github.com/csmith-project/csmith/blob/master/BUGS_REPORTED.TXT, Nov. 2013. Accessed: 2017-04-19.

[57] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain, and M. Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 249–265, Broomfield, CO, Oct. 2014.

[58] P. Zave. Using lightweight modeling to understand Chord. *SIGCOMM Comput. Commun. Rev.*, 42(2):49–57, Mar. 2012. ISSN 0146-4833.