Smart Casual Verification of the Confidential Consortium Framework

Heidi Howard[*], *Markus A. Kuppe*[*], *Edward Ashton*[*], *Amaury Chamayou*[*], *Natacha Crooks*[*‡]
Azure Research, Microsoft[*] *UC Berkeley*[†]

Abstract

The Confidential Consortium Framework (CCF) is an open-source platform for developing trustworthy and reliable cloud applications. CCF powers Microsoft's Azure Confidential Ledger service and as such it is vital to build confidence in the correctness of CCF's design and implementation. This paper reports our experiences applying *smart casual verification* to validate the correctness of CCF's novel distributed protocols, focusing on its unique distributed consensus protocol and its custom client consistency model. We use the term smart casual verification to describe our hybrid approach, which combines the rigor of formal specification and model checking with the pragmatism of automated testing, in our case binding the formal specification in TLA$^+$ to the C++ implementation. While traditional formal methods approaches require substantial buy-in and are often one-off efforts by domain experts, we have integrated our smart casual verification approach into CCF's CI pipeline, allowing contributors to continuously validate CCF as it evolves. We describe the challenges we faced in applying smart casual verification to a complex existing codebase and how we overcame them to find six subtle bugs in the design and implementation before they could impact production.

1 Introduction

The Confidential Consortium Framework (CCF) [28, 62] is a general-purpose platform for developing trustworthy and highly available cloud applications. CCF combines centralized compute with decentralized trust, supporting deployment on untrusted cloud infrastructure and transparent governance by mutually untrusted parties. CCF achieves this by leveraging hardware-based trusted execution environments (TEEs) for remotely verifiable confidentiality and code integrity [78, 79], coupled with state machine replication backed by an auditable immutable ledger for data integrity and high availability. CCF is trusted in production by services such as Azure Confidential Ledger [58], a tamper-proof append-only ledger, which is utilized for storing critical data like digests for integrity-protection with SQL Server [3, 64] and Azure Immutable Blob Storage [36]. Open source applications of CCF range from code transparency [17, 59] and mediation of multi-party data sharing [44], to decentralized identity [60], privacy-preserving ad auctions [63], and confidential storage [32, 57].

This paper summarizes our experience with verifying the correctness guarantees of the distributed protocols in CCF, a large-scale, production distributed system, using TLA$^+$ [46, 48]. We have five primary requirements that guided our approach to verification:

(1) Verify high-level distributed safety properties. We aim to check the correctness of our custom distributed consensus protocol, including its dynamic reconfiguration logic. CCF's consensus logic, though based on Raft [75] has been sufficiently modified such that it is now based on an unproven algorithm. This is a common problem: the Chubby authors [8] similarly note that the additional requirements imposed by a real-world system over a vanilla Paxos [47] implementation require significant changes. Moreover, distributed consensus protocols are notoriously hard to get right, and even well-known protocols, including PBFT [9], Egalitarian Paxos [67], and Zyzzyva [40], as well as the previously mentioned Raft and Chubby, have been found to contain subtle bugs [1, 2, 4, 56, 73, 83, 91]. The chosen verification strategy must therefore be able to check properties across distributed nodes, handling asynchrony, concurrency, and non-determinism in the order that events, such as timeouts and message delivery, occur as well as expected failures, like nodes crashing and message loss.

(2) Document and communicate the behavior of the system. A formal specification can act as compact and unambiguous documentation of system behavior for potential users. This gives developers and users clarity about the guarantees they can expect and rely on. The existence of such a spec also offers a succinct mechanism to communicate changes to these guarantees. Due to resource constraints on early TEEs [14], the consistency guarantees offered by CCF to clients can be subtle, confusing users and paper reviewers alike. In fact, even when consistency guarantees are seemingly simple, they can still be famously difficult to reason about [37]. We thus wish to formally define these guarantees, communicate them clearly, and check that these hold, even as failures occur.

(3) Increased confidence in the implementation & design. While many verification efforts focus solely on checking the design via reference specs, we also wish to validate that our concrete implementation corresponds to our specs. If, in the future, there were to be variants or even multiple implementations of CCF, we would like to verify that they are functionally equivalent in key areas.

(4) Integrates with the existing codebase. We do not wish, however, to rewrite CCF for the purpose of verification. At the time of writing, the implementation is already 63 kLoC in C++. Our support for Intel SGX [14] (via OpenEnclave SDK [19]) constrains us to C++. Moreover, since the project started seven years ago, we have invested significantly in adding functionality, improving performance, and supporting new hardware (AMD SEV-SNP [31]). We want our verification efforts to *improve* upon that existing investment, rather than impose a fresh implementation.

(5) Pragmatic and evolves with the implementation over time. CCF is an ongoing project which is continuously growing and evolving, with an average 16 pull requests merged every week. Since 1.0, the first release to be deployed to production, there have been four further major versions, with minor versions and patches released every 11 days on average. Any verification effort must thus be continuous, automatic, and sufficiently lightweight to integrate into the existing CI pipelines and software engineering workloads. The approach should also be approachable and pragmatic such that anyone contributing to the project can update the specs and debug discrepancies between the implementation and specs.

1.1 Approach

Full formal verification of distributed systems has been successfully applied in various research projects [21,23,24,92], for instance, by synthesizing an executable implementation from a formally proven spec. Unfortunately, while formal verification is a powerful tool, it requires a significant upfront investment in time and expertise. We thus chose a different approach. We already used traditional testing techniques (*casual verification*), but wanted to augment them with an approach that was more rigorous and complete. We thus chose to adopt *smart casual verification*[1], a pragmatic yet systematic approach to verification that combines the rigor of formal methods with the easy-of-use and flexibility of more casual methods. We combine a rigorous TLA$^+$ specification which we *tie* to our existing production implementation using *trace validation*.

We chose TLA$^+$ for several reasons. TLA$^+$ has been successfully utilized to verify the design of a number of production distributed systems [7,69,82,86,93]. Most notably for us, TLA$^+$ has been used to describe both consistency guarantees [22,26,87], and distributed consensus protocols, namely Paxos [47,88] and Raft [74]. The existence of the last of these was a significant factor in our decision, as it allowed us to start from a complete spec of Raft and adapt it to our protocol's specificities, rather than from scratch. Finally, the maturity of TLA$^+$ and its tools [15,38,43,94], the extensive examples available [52], and its active community [12] assured us of ongoing support and resources. Our increased investment in TLA$^+$, following initial successes, was supported by the availability of a recurring, two-day TLA$^+$ workshop [41] to train our team.

However, achieving our goals is not simply a matter of writing some high-level specs as this would not provide any guarantees about the production code itself. We bridged this gap by applying trace validation to validate implementation traces against the formal specs. In this paper, we describe our experience applying smart casual verification to CCF using TLA$^+$, focusing on the challenges we faced and how we overcame them. First, we present our TLA$^+$ specs of CCF from the perspective of its nodes (§4) and its clients (§5). Next, we present how we validate traces generated from the CCF implementation against our TLA$^+$ specs (§6). Finally, we reflect on our experiences applying smart casual verification to CCF (§7), documenting the six bugs we prevented, and the lessons learned along the way (§8). In contrast to some previous industrial efforts in this space, both our specifications and implementation are open source and actively maintained [65].

2 CCF

This section provides an overview of the distributed architecture of CCF, focusing on the components that are relevant to our verification efforts. Interested readers can find a more comprehensive description of CCF in [28].

At its core, CCF uses state-machine-replication (SMR) [80] and trusted execution to offer the abstraction of an always available application that remains robust to attacks, including from the nodes themselves. CCF assumes that neither node operators nor other applications running on the hardware can be trusted. The host, the OS, the hypervisor, the network and persistent storage are all assumed to be corruptible. Intuitively, SMR provides clients with the illusion that there is a single server that will, sequentially, execute individual application requests. SMR achieves this by replicating application logic on a set of nodes, a fraction of which may fail. The system maintains consistency across nodes by deciding on a totally ordered transaction log. Formally, SMR guarantees the following (def. from [75, Fig. 3]):

Property 1 (State Machine Safety) *If a node has applied a log entry at a given index to its state machine, no other node will ever apply a different log entry for the same index.*

[1] Smart casual refers to a style of dress that is neat and stylish, without the expense and discomfort associated with formal attire such as a business suit. Smart casual dress can easily be made more/less formal, for instance, with the addition/removal of a blazer or tie. [18]

Most SMR systems further ensure that the agreed-upon set of transactions will be applied in a way that guarantees *linearizability* [25] (or *strict serializability*): the resulting execution will be equivalent (equal read and write sets) to an execution in which each transaction was executed in sequence, and in an order that matches the order in which they were issued. Though appealing, strict serializability can be costly to enforce. SMR systems thus often relax this guarantee to read-only transactions specifically. They offer only *serializability* and allow read-only transactions to read stale state. CCF is no different: it offers strict serializability for committed read-write transactions and serializability for committed read-only transactions.

Serializability is a gold standard in system design, but is fairly pessimistic: a client must wait until a transaction has committed to learn any information about that transaction and its effects. This design was unfortunately at odds with CCF's initial SGX-related design constraint, which precluded keeping potentially large amounts of application-defined responses in the limited amount of in-enclave memory (128 MB). CCF thus provides configuration settings that clients can use to achieve good performance. They are useful in practice, but make it more challenging to formalize the consistency guarantees that users can expect.

In CCF, the leader node executes transactions as soon as they are received, and prior to them being replicated to other nodes. The leader then directly replies to the client with the result of the transaction without waiting for confirmation that the request has been replicated. As a consequence, a leader failure can cause the transaction to fail, even after a response has been returned to the client. Clients can then, on a per-response basis, decide whether they wish to wait for the transaction to be committed before proceeding or not. Either way, this cuts down the number of open client connections and pending responses, reducing memory footprint significantly.

Transactions in CCF can be in one of the following client-observable states: Committed, Pending, or Invalid. A transaction is Committed if it has been replicated by the leader to a majority of nodes in its current term. A transaction is Pending if it has been executed but not yet replicated. If the leader fails before replication is complete, CCF will mark the transaction as Invalid.

Pending transactions, can eventually become Invalid, but cannot return arbitrary results. They provide a guarantee akin to *fork-linearizability* [53] (or *fork sequential consistency* [6] when considering read-only transactions). A pending transaction observes a prefix of committed transactions and a sequence of pending transactions. Leader failures may cause the system to fork and generate multiple (locally linearizable) sequences of pending transactions. Only one forked sequence will eventually commit, thus ensuring that the set of committed transactions remains linearizable. All other sequences will be marked as invalid. In other words, if a pending transaction commits, the result it returned to clients is guaranteed to have been linearizable.

CCF makes extensive use of timestamps to help clients understand when transactions transition from Pending to Committed or Invalid. Each transaction is associated with a unique transaction identifier, consisting of a lexicographically ordered pair $\langle t.i \rangle$ of term t and log index i. The client can use this ID to quickly understand the system state. CCF, for instance, enforces *timestamp ordering*: if $txid < txid'$ and the two transactions are committed, then the transaction with $txid$ executed before $txid'$. Clients can further use this ID to learn the state of not only this transaction, but its ancestors. For instance, CCF guarantees that:

Property 2 *If* $\langle t.i \rangle$ *is committed then any transaction* $\langle t.j \rangle$ *where* $j \leq i$ *is also committed.*

Ensuring that CCF does indeed provide these specific guarantees requires care and adapting existing linearizability specs, which only reason about committed operations. They support neither reasoning about forks nor timestamp properties.

To tolerate node crashes and network asynchrony, SMR requires a crash fault-tolerant distributed consensus protocol or equivalent [10], such as Multi-Paxos [47] or Raft [75], to agree on a total-ordered transaction log. While the terminology varies, such protocols are typically leader-based and operate by electing one of the nodes to be the *leader* while the other nodes are *followers*. The leader is responsible for proposing new transactions for the log and the followers are responsible for replicating them. The leader will only consider a transaction to be committed once a strict majority of nodes have replicated the transaction at the same position in the log. When the leader fails, a new leader is elected from the remaining nodes and the protocol continues. We use *terms* to distinguish between the different periods of leadership and there should be at most one leader per term.

It is the responsibility of the leader-based consensus protocol to ensure that the new leader has knowledge of all previously committed transactions. In Raft, this is achieved by requiring followers to become *candidates* (transition ① in Fig. 1) before they can become leaders. A candidate will only become a leader (transition ② in Fig. 1) if it can obtain a strict majority of votes from the other nodes. A node will only vote for a candidate if it has not already voted in the candidate's term and the candidate's log is at least as up-to-date as its own log. The former ensures that at most one leader can be elected per term and the latter ensures that the new leader has knowledge of all committed transactions from previous terms. A more comprehensive description of Raft can be found elsewhere [75].

2.1 What makes CCF's distributed consensus protocol interesting?

CCF uses a custom consensus protocol, which evolved from Raft. Now we will outline some of the ways that the consensus protocol in CCF today differs from the description of the Raft protocol given the original paper. While some of these modifications may seem small, the combined effect is significant and the interactions between them have proven complex and subtle, leading to the bugs we later describe (§7).

Signature transactions Offline log integrity and transaction provenance are key requirements for CCF, neither of which is provided by Raft, nor by the AEAD mechanism used inside the CCF network. The offline guarantees crucially enable external audit, and disaster recovery. To implement them efficiently, CCF utilizes *signature transactions*, which include the root of a Merkle tree [55] over the whole log thus far, signed by the current leader. A transaction in the log is not considered committed unless a subsequent signature has been committed.

Messaging not RPCs CCF does not use RPCs to communicate between nodes, and instead it uses a uni-directional messaging layer. When a node receives a response to a message it has sent, it does not know which message the response corresponds to as we do not assume reliable or in-order delivery. Raft uses two main message types, APPENDENTRIES (AE) and REQUESTVOTE (RV). In the case of RVs, the term in the response is sufficient to handle the reply. In the case of AEs, the response in CCF contains an additional field, LASTINDEX. For positive responses to AE messages (AE-ACK), this is the index of the last transaction in the follower's log.

Optimistic acknowledgement In Raft, the leader replicates transactions to its followers using AE requests. The leader maintains a NEXTINDEX for each node to record which log entry to send next to the follower. This index is updated when a follower responds positively to an AE request (AE-ACK). If a leader receives a new transaction from a client, before it has received a response from a follower, it must either (1) wait for the reply for the previous transaction, potentially impacting liveness and performance as AEs cannot be pipelined, or (2) send both transactions in the next AE request, even though the follower is likely to already have the first transaction. CCF avoids this problem by allowing the leader to update the index, known in CCF as the SENTINDEX, as soon the AE message is sent. This therefore means that if the leader receives a negative response to its AE request (AE-NACK), it might need to rollback the SENTINDEX.

Express node catch up AE messages include a previous log index and term, which allows a follower to determine if it diverged from the leader. If a follower does not have the previous log index and term, either because it does not have, or has a different transaction at that index, it then responds with an AE-NACK. CCF uses an express catch up mechanism, where a leader makes a conservative estimate of how far behind a follower is, and sends a batch of transactions to the follower.

Partition leader step down A known limitation of the base Raft protocol is that partial/asymmetric network partitions can cause a loss of liveness [27, 33]. For instance, if a leader can no longer make progress because it cannot receive messages from the other nodes, it continues to send AE heartbeats to followers, preventing them from timing out and from electing a new leader who can make progress. CCF implements a known extension [72, pg. 69] to the Raft protocol, referred to as *CheckQuorum*, where a leader steps down if it does not hear back from a quorum of nodes within a specified time period (transition ③ in Fig. 1).

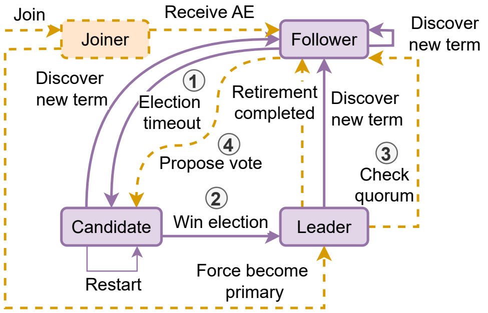

Figure 1: State transitions for CCF's consensus protocol. Solid boxes and lines show Raft's original states and transitions, CCF's additional states and transitions are shown with dashed lines.

Bootstrapping to retirement Reconfiguration is the process by which the set of nodes participating in consensus can be changed. Raft support two reconfiguration protocols: *joint consensus*, described in the original paper [75], and single-server reconfiguration, subsequently described in [72]. CCF's reconfiguration protocol is more similar to the former. Reconfigurations are recorded in CCF's log with *configuration transactions*, and are therefore ordered in the same total order. Logs always begin with an initial singleton configuration transaction followed by a signature transaction. To change the configuration, the leader proposes a new configuration transaction specifying the new set of nodes, which may be different in cardinality to the current set of nodes and may or may not be disjoint. To commit this transaction, the leader must obtain a quorum of AE-ACKs from both the previous and new configurations. Once the transaction is committed, the leader no longer needs quorum agreement from the previous configuration. If a node has been removed from the configuration, we refer to it as *retiring*. In order to complete its retirement and permanently switch off the node, a *retirement transaction* must be committed to ensure that any future leader will know that the reconfiguration which removed the node has been committed, and thus the node will never be needed. CCF also adds a message to the protocol, *ProposeVote* which is utilized by a retiring leader to nominate a successor, fast-tracking the usual leader election process (transition ④ in Fig. 1).

3 Primer on TLA$^+$

TLA$^+$ [46, 48] is a formal modeling language widely used to verify concurrent and distributed systems. It is easy to learn and use, as well as agnostic about system frameworks or implementation languages.

TLA$^+$ is a variant of linear temporal time logic with only two operators, *Always* (□) and *Eventually* (◇). A system is defined by a set of *behaviors*, each a sequence of pairs of states called *actions*, beginning at one of the system's initial states. A *state* is an assignment of values to variables. Syntactically, TLA$^+$ describes a system's state machine using a canonical (temporal) formula $Init \land \Box[Next]_{vars} \land L$. Here, $Init$ is a predicate defining the system's set of initial states. The system's next-state relation $Next$ is a first-order logic formula that is usually decomposed into a disjunct of actions which relates the values of the variables in the current state to the ones in the successor state. CheckQuorum (Listing 3), for instance, states that a node i can abdicate as a leader and become a follower; we change the value of the variable *role* to $Follower$ in the successor state, while the values of the other variables remain unchanged.[2] The tuple, $vars$, represents the spec's variables, and $[Next]_{vars}$ stipulates that either $Next$ is true, or the variable in $vars$ do not change. This asserts that TLA$^+$ specs are stuttering-insensitive, allowing a spec to always be refined by a more detailed, low-level one. The optional formula L is used to assert fairness, i.e., constraints on the system's behavior that ensure that certain actions eventually occur. Composition of actions allows us to change the *grain of atomicity* by defining more coarse-grained behaviors [48, §7.3]. Concretely, the composition $A \cdot B$ states that the two actions happen atomically; the intermediate state between A and B is not observable.

We also state desired safety (*something bad never happens*) and liveness (*something good eventually happens*) properties in TLA$^+$. For example, the safety properties LogInv, AppendOnlyProp, and MonoLogInv (Listing 3, described in §4) assert properties of the *log*. Properties are checked and verified using, random state space exploration (simulation), model checking, or theorem proving. TLC [94], an explicit-state model checker, verifies that a *finite* model of a spec satisfies its properties by enumerating all reachable states. A symbolic model-checker [38] is especially well-suited for the verification of inductive invariants of finite systems. The TLA$^+$ Proof System [15, 89] mechanically verifies a deductive proof of the properties of an infinite system. These tools complement each other, enabling a combination of model-checking and deductive proofs to verify specs. This approach allows for varying levels of verification depth, from push-button model checking to fully mechanized safety and liveness proofs [39]. Our companion paper [11, §2.1] provides a more detailed summary of TLA$^+$.

4 Distributed Consensus Specification

Fig. 2 summarizes our consensus verification architecture. The consensus specification, shown as ① in Fig. 2, consists of 17 actions to describe the transitions over 13 variables. The first 12 variables are local and track consensus state (currentTerm, log, commitIndex, ...). The last variable instead tracks the set of in-transit messages, allowing support for different network abstractions (ordered/unordered delivery, etc.) and is shown as ② in Fig. 2. Each action models a node taking a single step within the protocol, updating its local state and optionally the collection of in-transit messages, (i.e. CheckQuorum in Listing 3). Actions can either be initiated from a message receipt or by a node itself. Actions in the latter class include adding a signature transaction, stepping up as a candidate or stepping down as a leader. Such actions are always enabled, to model the fact that we make no assumptions about clock synchrony; each node's opinion of the progression of time is independent. Later we discuss how action-weighted simulation enabled us to find bugs in the prototype despite the state space explosion inherent with such an approach. The spec is parameterized by the set of nodes available in the service. The initial states of the spec include every non-empty subset of nodes in the initial configuration with any node in that initial configuration as an initial leader.

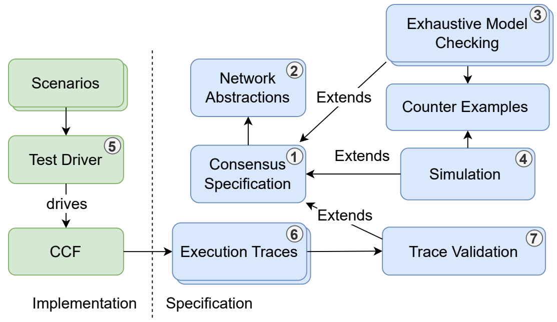

Figure 2: Components of our verification architecture for consensus

$$\begin{aligned}
&CheckQuorum(i) \triangleq \\
&\quad \land role[i] = Leader \\
&\quad \land role' = [role \text{ EXCEPT } ![i] = Follower] \\
&\quad \land \text{UNCHANGED } \langle currentTerm, log, votedFor, ... \rangle
\end{aligned}$$

$$\begin{aligned}
&LogInv \triangleq \forall i,j \in Nodes: \\
&\quad \lor IsPrefix(Committed(i), Committed(j)) \\
&\quad \lor IsPrefix(Committed(j), Committed(i))
\end{aligned}$$

$$\begin{aligned}
&AppendOnlyProp \triangleq \\
&\quad \Box[\forall i \in Nodes: \\
&\qquad IsPrefix(Committed(i), Committed(i)')]_{vars}
\end{aligned}$$

$$\begin{aligned}
&MonoLogInv \triangleq \forall i \in Nodes: log[i] \neq \langle\rangle \Rightarrow \\
&\quad \forall k \in 1 \,..\, Len(log[i]) - 1: \\
&\qquad \lor log[i][k].term = log[i][k+1].term \\
&\qquad \lor \ \ \land log[i][k].term < log[i][k+1].term \\
&\qquad\qquad \land log[i][k].contentType = Signature
\end{aligned}$$

Listing 3: Excerpt of our consensus spec.

Our key correctness property is State Machine Safety (Property 1) which we check with invariant LogInv and action property AppendOnlyProp (Listing 3). LogInv states that all pairs of committed logs must be consistent and AppendOnlyProp states that each node can only extend its committed log. LogInv checks for safety violations across nodes (in space) and AppendOnlyProp checks for safety violations within a node (in time). We also checked a further 27 invariants/properties to ensure the correctness of our con-

[2] The value of the variable *role* is a mapping from all node identifiers to their roles such as *leader*, *follower*, and *candidate*.

sensus protocol and of our understanding (see [65]). One such example is MonoLogInv (Listing 3), which is a stronger variant of the property that log terms are monotonically increasing which we depend upon extensively. More concretely, MonoLogInv states that terms in the log can only increase after a signature and remains the same otherwise.

Our consensus spec is unbounded, there is always the possibility of a new transaction being proposed or a timeout triggering a new election. To exhaustively model check our spec, we extend our consensus spec, shown as ③ in Fig. 2, to restrict the state space by adding additional constraints to the actions limiting the max term, number of client requests, and the sequence of reconfigurations. We found exhaustive model checking too time-consuming to run with all but the strongest state constraints in our CI pipeline, particularly after the spec was updated to describe reconfiguration accurately (discussed further in §8). We therefore developed an extension to our spec for simulation as a lightweight alternative to exhaustive state exploration. Our simulation spec, shown as ④ in Fig. 2, takes a time quota and explores as many behaviors as possible, up to a given depth, within that time. To expand the coverage of simulation, specifically to explore behaviors where the system exhibits more forward progress, we manually weighted failure actions to reduce the likelihood of them being chosen. We also implemented Q-Learning proposed by [68] in TLC to automatically weight actions to increase coverage of simulation. However, we were unable to find the right set of variables as input to Q-Learning's state hash function $\mathcal{H}$ that achieved better coverage than manual weighting.

5 Client Consistency Specification

Following our success with the consensus spec and a discussion regarding the linearizability of read-only transactions [61], we decided to formalize the possible externally visible behaviors of a CCF service in TLA$^+$ to better understand the guarantees provided. Our aim with this spec was to keep it as high-level as possible, focusing solely on the possible safe interactions between clients and the service. By design this spec does not model the internal details of the service itself such as the state of individual nodes or the messages exchanged between nodes as such low level details are already modeled by our consensus spec.

Our consistency spec uses just two variables. The first is History, an append-only sequence which records the messages exchanges between clients and the services. Five messages are supported, read-only/read-write transaction requests and responses as well as a transaction status messages. Note that since we are focused on safety, we omit messages that cannot impact correctness but can increase the state space, for instance, we do not track when a client requested a transaction status, nor do we track status responses of Pending. In order to stress the consistency guarantees, we modelled an application where all transactions operate on a simple value, reading the current value, appending a new identifier to the value and writing back the new value. All transactions conflict and each transaction observes every transaction that has been executed before it. To succinctly record the current state of the service we use LogBranches, an append-only two-dimensional sequence, where the sequence at index i corresponds with the local log of the leader of term i if such a node still exists. This representation does not therefore need to be parameterized over the number of nodes in the service and usefully models the fact that there can be multiple leaders at one time (although they will have different terms).

$$\begin{array}{l}
PrevCommittedInv \triangleq \\
\quad \forall i,j \in \{x \in \text{DOMAIN } history : history[x].type = Status\} : \\
\quad \land history[i].status = Committed \\
\quad \land history[i].term = history[j].term \\
\quad \land history[j].index \leq history[i].index \\
\quad \Rightarrow history[j].status = Committed
\end{array}$$

$$\begin{array}{l}
ObservedRoInv \triangleq \\
\quad \forall i \in RwResCommitIndexes : \\
\quad\quad \forall j \in RoReqCommitIndexes : \\
\quad\quad\quad \forall k \in RoResCommitIndexes : \\
\quad\quad\quad\quad history[k].tx = history[j].tx \land i < j \\
\quad\quad\quad\quad\quad \Rightarrow Contains(history[k].observed, history[i].tx)
\end{array}$$

Listing 4: Two properties over histories checked by our consistency spec.

To populate these histories, we defined a set of actions to describe how the history can be extended depending on the log branches. Initially, the history and log branches are empty. The possible actions are: append a transaction request or response to the history, execute a transaction by appending it to any log branches, append a transaction status to the history, or starting a new log branch to simulate leader election. Note that when a transaction is executed, it can be appended to any log branch, this simulates the fact that a transaction can be handled by any node that believes itself to be the leader, even if it is not the latest leader. At any time, a new log branch can be started. This new log branch can be any prefix of any existing log branches, provided it includes the last committed transaction. Having established an approach to generate histories, we formalized in TLA$^+$ the properties that we expect to hold over these histories and check them with TLC. For instance, PrevCommittedInv (Listings 4) formalizes Property 2 by stating that for any pair of transaction status responses from the same term, if the one with the greater (or equal) index is Committed, then the other status response must also be Committed (as Pending states are not modelled). ObservedRoInv (Listings 4) states that if a committed read-write transaction received an initial response (event i), before a committed read-only transaction was started (event j) then the read-only transaction response (event k) must observe the read-write transaction.

6 Trace Validation

Conceptually, trace validation checks that every observed implementation trace matches a behavior of the system's high-level specification. We must therefore first collect implementation traces before validating them against our high-level consensus TLA$^+$ specification. A more formal discussion of trace validation, including insights from applying trace validation to systems other than CCF, is provided in [11].

6.1 Trace Collection

When we began the trace validation work, CCF already had extensive unit, functional, and end-to-end testing. Unit testing of CCF's consensus layer consisted of approximately 1.6 kLoC. End-to-end testing covered complex scenarios such as reconfigurations, node failover, and network partitions, spread over more than 2 kLoC of Python tests and infrastructure code. Consensus functional testing was done through a scenario driver, shown as ⑤ in Fig. 2, that serialized execution deterministically across nodes, and isolated the consensus layer by mocking unrelated CCF components, such as governance, networking etc. This driver allowed the injection of network faults such as partitions, delays, reorderings, and message drops, and provides observability. Core correctness invariants and properties were checked at designated execution steps in 13 manually written scenario tests exercising replication, election, and reconfiguration under controlled fault conditions. Additionally, an initial prototype to fuzz-test the consensus layer through generated inputs and faults was developed but ultimately abandoned since it failed to generate interesting behaviors that would achieve satisfactory coverage.

The driver allows us to replace the node's wall clocks with a single global clock. If this global clock had not been possible, a distributed clock, such as Lamport or vector clocks, would have also provided the necessary event ordering to establish the happen-before relationships. However, a distributed clock would have required changes to the network message format. We enhanced system observability by incorporating an additional 15 log statements to capture consistent system states at well-defined, side-effect-free linearization points including (i) the sending and receipt of network messages, and (ii) transitions in a node's high-level state, such as moving from candidate to leader[3]. It is important to note that the driver logs only those values that remain constant in space. For instance, the driver records the length of the logs but not the entries themselves, which would be impractically large. Still, the logging is disabled at compile time for production builds, and thus does not impact CCF's performance.

Before validation, implementation traces, shown as ⑥ in Fig. 2, are preprocessed to exclude and de-duplicate events from the initial bootstrapping phase of a CCF network, as this phase is not modeled in our consensus spec.

6.2 Consensus Trace Validation

Trace validation, shown as ⑦ in Fig. 2, ensures that an implementation trace is consistent with the specification. More formally, it verifies whether the set of behaviors $\mathcal{T}$, which encapsulates the values and events from a trace, intersects with the set of behaviors $\mathcal{S}$ derived from the high-level spec, thereby checking that $\mathcal{T} \cap \mathcal{S} \neq \emptyset$. While TLC can construct $\mathcal{S}$, it cannot directly generate $\mathcal{T}$ from the trace. Thus, we write a new TLA$^+$ spec, *Trace*, reusing many actions of the definitions from the high-level spec. However, the actions are only enabled iff the current event in the trace matches an action. Likewise, the actions are parameterized by the values taken from the trace, effectively constraining successor states.

```
VARIABLE ln

IsSendAppendEntries ≜
    Enablement conditions on current state.
  ∧ IsEvent(ln, "sndAE")
  ∧ commitIndex[ln.snd] = ln.commit_idx
  ∧ ...
    High-level spec actions.
  ∧ AppendEntries(ln.snd, ln.rcv)
    Assertions on successor states.
  ∧ ∃m ∈ Network!Messages′ :
     ∧ Network!OneMoreMessage(m)
     ∧ IsAppendEntriesRequest(m, ln)
  ∧ ...

IsRcvAppendEntries ≜
    Enablement conditions on current state.
  ∧ IsEvent(ln, "recvAE")  ∧ ...
    High-level spec actions.
  ∧ ∃m ∈ Network!MessagesToFrom(ln.rcv, ln.snd) :
     IsAppendEntriesRequest(m, ln) ∧
     ∨ HandleAppendEntriesReq(ln.rcv, ln.snd, m)
       Impl optimization: Piggyback term on AppendEntries.
     ∨ UpdateTerm(ln.rcv, ln.snd, m) ·
           HandleAppendEntriesReq(ln.rcv, ln.snd, m)
     ∨ ...
    Assertions on successor states.
  ∧ ...

IsSendAppendEntriesResponse ≜
    Enablement conditions on current and successor state.
  ∧ IsEvent(ln, "sndAER")  ∧ ...
    High-level spec actions.
  ∧ UNCHANGED vars

IsFault ≜ ∃s ∈ MultiPowerset(network) :
     network′ = s ∧ UNCHANGED AllVarsExceptNetwork

Spec ≜ Init ∧ □[IsFault · Next]_⟨vars, ln⟩
```

Listing 5: Excerpt of the *Trace* spec for trace validation.

Consider for example *IsSendAppendEntries* (Listing 5). This action is enabled iff the current line of the trace, denoted by *ln*, is a *sndAE* event, and the *commitIndex* of the sending node (denoted by *ln.snd*) matches the trace's *commit_idx*. The action then reuses the definition of the *AppendEntries* action from the consensus spec, parameterized by *ln.snd* and

[3] State changes are logged immediately after acquiring a global lock.

ln.rcv from the trace. Given that the number of entries in the high-level action *AppendEntries* is chosen nondeterministically within defined limits, *IsSendAppendEntries* determines the successor state by asserting the presence of an additional *AppendEntriesRequest* in the network with a matching number of entries. Note that while the consensus spec modeled the network as a set of messages, thereby not altering the network upon a resend of an *AppendEntriesRequest*, *Trace* redefines the network as a multi-set. This allowed the spec to account for the addition of messages in the network, even during resend events. Subsequently, this approach to address the impedance mismatch was expanded to verify, with TLC, the impact of various message delivery guarantees, such as ordering, duplication, and other message loss patterns.

6.2.1 Aligning Grains Of Atomicity

The granularity of some actions in the consensus spec did not align with the granularity of events in the traces. The action *IsRcvAppendEntries* outlines the alignment of such different *grains of atomicity*. Like many Raft implementations, CCF minimize network roundtrips by piggybacking *term* updates on *AppendEntries* messages. This optimization was not reflected in the consensus spec. Instead, the spec modeled term updates with an action that increases the term upon a pending *AppendEntriesRequest*, while leaving the network unchanged. In effect, a term update might nondeterministically occur before the receipt of the *AppendEntriesRequest*. To reconcile these different grains of atomicity, we composed the actions *UpdateTerm* and *HandleAppendEntriesReq*, allowing them to occur atomically in a single action.

Another important application of action composition addresses events that are omitted from the trace, such as dropping messages. Although our consensus specs explicitly modeled loosing messages, message loss was not recorded in the trace. Therefore, to account for faults at any step of a behavior, we composed an *IsFault* action with *Trace*'s next-state relation. Conversely, aligning a single high-level action with multiple implementation events is addressed by introducing finite stuttering that does not change the high-level variables. For instance, the action *IsSendAppendEntriesResponse* is enabled iff *ln* is a *sndAER* event. However, it leaves the high-level variables unchanged.

6.2.2 Deriving Consensus Trace Validation

We began to derive *Trace* by mapping the trace of a straightforward happy-path test line by line to the consensus spec. To proactively catch discrepancies, we added as many assertions as possible to *Trace*. Whenever we discovered discrepancies, we investigated them by examining relevant sections of the implementation's source code. Adding cross-references between the implementation and the spec proved useful, particularly when the terminology——such as variable and function names——differed between the two. We further debugged the *Trace* using the TLA$^+$ debugger [42] in tandem with implementation debugging. Upon detecting discrepancies, we corrected either the implementation or the spec, subsequently rerunning verification on the revised consensus spec, and executing tests on the updated implementation.

6.3 Debugging Discrepancies

Bogus logging, incorrect mappings from implementation to spec state, or true discrepancies between the spec and the implementation, resulted in $\mathcal{T} \cap \mathcal{S} = \emptyset$, i.e., the verdict that a trace is invalid. In either case, contrary to ordinary model checking, a failure to validate a trace has no counterexample. However, the behaviors within $\mathcal{T}$ helped explain why a trace fails to validate. We typically compared the final state of the longest behaviors and the corresponding line in the trace to identify the source of the mismatch. The TLA$^+$ debugger was instrumental in this process, as it allowed us to step through the evaluation of formulas and compare variables at the current and successor states with the trace values. To determine if *Trace* is overly restrictive, we implemented a new *unsatisfied breakpoint*. It triggers for each state in $\mathcal{T}$ that is found to be unreachable, often due to the assertions in *Trace*. Furthermore, $\mathcal{T}$ can be visualized as a graph that not only includes all unreachable states but also references the subformula responsible for the state being unreachable.

6.4 Scalability of Trace Validation

The cardinality of $\mathcal{T}$, the set of potential system behaviors, can become prohibitively large due to nondeterminism resulting from incomplete traces. Recognizing that it suffices to find a single behavior in the intersection of $\mathcal{T}$ and $\mathcal{S}$ to check the validity of a trace, we implemented depth-first search (DFS) in TLC. This method mitigated the issue of state-space explosion, making trace validation orders of magnitude faster compared to enumerating all behaviors with breadth-first search (BFS). For instance, validating a trace against our consistency spec started to take less than a second using DFS, compared to about an hour with BFS.

6.5 Trace Validation Effort

The enhancements to the test driver and the addition of detailed logging were completed in approximately one day. The effort to derive a version of the *Trace* spec that validated the majority of the traces required approximately two engineer-months, spread over four months. The primary tasks included enhancing the TLC model checker to support trace validation, which involved implementing support for action composition, DFS, improved debugging support, and visualizing the state graph. The second major tasks was diagnosing if the root cause of discrepancies arose from bugs in the reverse-engineered spec, the implementation, or both. This frequently required consulting the original Raft paper and discussions with the CCF experts to elucidate differences between Raft and CCF. In this context, the shared vocabulary developed during the TLA$^+$ workshops, along with TLA$^+$ counterexamples from trace validation, simulation, and model checking proved invaluable. The third major task involved finding modeling patterns to bridge impedance mismatches between the high-level TLA$^+$ design and low-level system behavior. Writing the 400 LoC *Trace* spec itself was a minor task. The introduction of trace validation resulted in 88 fine-grained commits to the *Trace* spec, while the consensus spec underwent 107 changes.

The discovery of a serious safety bug through trace validation (*Commit advance on AE-NACK* §7) led to increased investment in trace validation. Substantial changes were made to the consensus spec to accurately reflect the implementation. For example, the bootstrapping of a CCF network was modeled with greater fidelity. Additionally, the spec was expanded to include all node states, especially those related to node retirement. These comprehensive changes necessitated substantial revisions to the test driver and the development of new tests. This uncovered a serious liveness bug (*Premature node retirement* §7). Once the consensus spec was validated to accurately mirror the implementation, we transitioned to a spec-driven development, wherein the spec served as the source of truth. Notably, this phase included the integration of the *ProposeVote* messages (described in §4; used for transition ④ in Fig. 1).

Later, trace validation was also applied to the consistency spec, where the effort required was substantially less. The consistency spec was considerably less complex, was written with the implementation in mind, and we had already gained experience validating our consensus spec. Moreover, TLC had already been enhanced to support trace validation. No instrumentation of CCF's source code was necessary for the consistency trace validation, which was performed simply by calling client endpoints. Like with consensus, we had to address impedance mismatches. For instance, the consistency spec assumed knowledge of the transactions of all clients, whereas a trace is limited to the transactions of a single client. This required introducing logic to reconstruct all transactions based on observed transaction IDs. Yet, applying trace validation to consistency did not require the involvement of the formal methods expert and was almost entirely carried out by CCF experts. The effort required to apply trace validation to the consistency spec was approximately one engineer-week, spread over two weeks.[4]

7 Results

This section summarizes the core results of our efforts, focusing on state coverage as well as the bugs found. Table 1 compares the sizes of the TLA$^+$ specs against the implementation and test infra, to give a sense of scale and illustrate the level of detail necessary to execute trace validation. We find that verification of our TLA$^+$ specs is an extremely efficient way to achieve state coverage. Comparing state exploration between implementation and specs is straightforward, because traces can also be collected in end-to-end tests, and one log line is largely equivalent to a spec action. It is immediately apparent that verification of our consensus spec explores orders of magnitude more states at a higher rate than implementation testing. The fact that verification matches unit testing in state throughput magnitude despite checking a substantial set of invariants and properties is remarkable. While code size is not a direct measure of cost, the overall size of the spec and its models are not out of proportion compared to the tests. Importantly, the consistency spec required very little infrastructure to verify, and to validate traces against the implementation. The cost of writing formal documentation of the log's consistency guarantee was thus low, and validating it and keeping it in sync with the implementation is equally affordable.

Table 1: Scale of specifications and state coverage.

				Approx states	
	Item	LoC	Vars	/min	Total
Consensus	Specification	1134	13		
	Model Checking	158		10^6	10^8
	Simulation	69		10^6	10^8
	Trace Validation	369			
	Implementation	2174	25		
	Unit Tests	1691		10^8	10^6
	Functional Tests	2579		10^5	10^3
	End-to-end Tests	2815		10^3	10^4
Consistency	Specification	375	2		
	Model Checking	70		10^6	10^5
	Simulation	0		10^5	10^3
	Trace Validation	111			
	Functional Tests	123			

All numbers measured on an Azure DC8s v3 VM.

Table 2 lists the most serious bugs found in our consensus protocol as part of the verification work. These bugs affected both the safety and the liveness of CCF, and were uncovered at several stages of the process by each tool in our verification wardrobe. In the rest of this section we describe the bugs which were found and corrected during this process, explaining how they were uncovered by our combination of smart casual and classical testing methods. All these bugs were fixed before they affected any end-users or resulted in customer bug reports. Thus, although this formalization work took place after the system was initially developed and deployed, it still provided a core benefit of spec-driven development; the identification and removal of critical bugs before they impact production, proving that it is never too late to benefit from smart casual verification.

Incorrect election quorum tally 48 hours of exhaustive model checking of the consensus spec on a 128 core machine revealed that CCF was tallying election quorums against the union of active configurations (the current configuration plus any pending reconfigurations), rather than against each individual active configuration (compare §2.1). The initial imple-

[4] The work on trace validation has been tracked in milestones 18 and 20 at https://github.com/microsoft/CCF/milestones/.

Table 2: Bugs found in CCF's consensus protocol before they could impact production.

Name	Violation	High-Level Description (Issues)
Incorrect election quorum tally	Safety	Quorum was tallied against union of active configurations, rather than against each individual active configuration. (#3837, #3948, #4018)
Commit advance for previous term	Safety	Leaders could advance commit for historical terms without extending log in the current term. (#3828, #3950, #3971, #5674)
Commit advance on AE-NACK	Safety	Variable reuse could cause the leader's commit index to advance when receiving an AE-NACK (#5324, #5325)
Truncation from early AE	Safety	Followers could roll back committed entries, after a sequence triggered by stale AE-NACK messages. (#5927, #5991, #6016)
Inaccurate AE-ACK	Safety	AE-ACK could report an index beyond the end index of AE received, despite the suffix potentially being incompatible. (#6001, #6016)
Premature node retirement	Liveness	Nodes could stop participating in consensus too early during retirement, leading to diminished fault tolerance. (#5919, #5973)

mentation had correctly used the majority in each term, as described in [75, §5.2], but had not been updated appropriately when reconfiguration was implemented. This meant that a node could be elected leader in a term without having a quorum in one of the active configurations, potentially allowing two leaders being elected in the same term, violating a core safety property. The issue was reproduced with functional and end-to-end testing, a fix was applied, and resolved the problem in both tests and model checking. This was the first bug identified by TLC and motivated our further investment in TLA$^+$.

Commit advance for previous term While assessing the work required to align the implementation and the spec, we discovered that the implementation omitted a check described in Raft. Our implementation allowed a leader to advance its commit index based solely on receiving a quorum of AE-ACKs confirming a given log entry, and missed the additional requirement that this entry must have been appended by the current leader. This restriction is fully explained in [75, §5.4.2], along with the corresponding risk to safety, but had been accidentally omitted in the implementation. We added a scenario test based on [75, Fig. 9] to confirm that the implementation was faulty. An initial fix emptied the node's set of indices eligible for commit (because they are signature transactions) when becoming a leader. This fix passed all existing and amended tests and was thus integrated into the codebase. Our work on trace validation, several months later, required us to revise the spec to represent committable indices accurately. Subsequent simulation revealed a safety violation caused by the initial fix; the *fix* broke an implicit property that committable indices contains *all* signatures. A second fix was implemented and tested as well as verified with TLC. This bug illustrated that even deterministic testing (compare §6.1) is insufficient to guarantee the correctness of changes. Moreover, it confirmed that trace validation is effective at guiding the alignment of the spec and the implementation to enable verification.

Commit advance on AE-NACK Trace validation discovered that the spec defined a leader's *matchIndex* to remain unchanged after receiving a follower's AE-NACK, whereas the implementation allowed it to decrease. This difference was due to an aggressive implementation of an optimization proposed in [75, §5.3, last paragraph]. After a single LoC change to align the spec with the implementation, subsequent simulation found a 34-state counterexample violating one of the spec's main correctness properties. This counterexample was manually translated into a 150 LoC functional test, confirming that the implementation could incorrectly advance its commit index. We also noted that [75, fig. 2, p. 4] implicitly states that *matchIndex* should never decrease, except after a leader election. Adding this property to the spec allowed model checking to find a shorter counterexample. The combination of functional testing and model checking allowed us to fix this bug quickly and confidently.

Truncation from early AE Once a subset of our initial scenarios passed trace validation, investigating why the remaining scenarios failed trace validation uncovered a safety violation. Log entries necessary to the persistence of committed transactions could be rolled back by a follower. This bug was introduced by an optimization, and existed in the implementation for some time. When the suffix of a follower's log is incompatible with that of the leader, there is a need to find the last agreement point. The Raft paper describes an iterative reverse search of the sequence numbers, but we instead implemented a suggested optimization to skip entire terms of divergence. CCF thus finds an agreement point after a sequence of roundtrips bounded by the number of divergent terms, rather than sequence numbers. We implemented this optimization by changing the semantics of the AE-NACK message, in which followers now include a safe best-estimate of an agreement point communicated using existing fields in the AE-NACK message.

Because the leader cannot distinguish these estimate messages from stale AE-NACKs emitted in previous terms, it may respond with an AE starting before the end of the follower's log. This, coupled with an insufficiently defensive code path in the follower's code, would cause the AE to be treated as a conflicting suffix, and trigger a roll back preceding the application of the AE, potentially violating Leader Completeness (defined in [75, Figure 3]). The fix proved simple: rather than rolling back optimistically on an AE in a new term, the follower should only do so on true conflicts.

Notably, this bug was triggered by existing functional tests, producing output that did not pass trace validation, but the tests' assertions were not strong enough. The reproduction scenario we wrote produced a trace 305 events long at the point the bug manifests itself. This bug showed the important benefit of trace validation to check the invariants in every state, compared to the traditional approach of manually inserting assertions in scenarios at specific points.

Inaccurate AE-ACK Fixing this bug directly led to the discovery of the AE-ACK index issue. Because CCF uses unidirectional messages rather than RPCs (§2.1), the code responsible for responding to messages sends values from local state where possible, rather than values specific to the message they responded to. The AE-ACK handler did so for the *last_idx* field, which is the index of the last entry in the appended entries, without correctly checking that the log was also compatible. This was discovered while conducting trace validation on the previous issue, and a specific scenario was added to test it in isolation. Here too, the fix was simple, and involved constraining the *last_idx* in AE-ACKs to the last index contained within the received AE.

Premature node retirement Network configuration in CCF is stored in a map (compare §2.1). Adding or removing nodes happens through updates to this map, which produce write sets that are also replicated and handled like any other transaction, and are therefore part of the totally ordered log. The consensus logic is notified through hooks, which can be called when a transaction is ordered and/or committed. As a result, the reconfiguration logic is not isolated from the consensus code, and was initially mocked with low fidelity in the scenario driver, and correspondingly simplified in the TLA$^+$ spec. This was known to be a significant discrepancy between the implementation and the spec, which we decided to address by making the driver more realistic, and by improving the scenario coverage used for trace validation. As the spec was aligned, simulation produced counterexamples where a reconfiguration would leave the CCF network permanently unable to make progress; a retiring node stopped responding before all future leaders were aware of its retirement. We translated the counterexamples into new functional tests to reproduce the issue. A fix, leveraging an existing mechanism to shut down retired nodes safely, was proposed, verified, and implemented together.

Non-linearizability of read-only transactions Thus far we have described bugs founds in our consensus protocol. In this last example we describe how we identify an ambiguity in CCF's docs, aided by the consistency spec. Recall that linearizability requires that transaction execution is consistent with the real-time ordering of client requests and responses. Recall that OBSERVEDROINV (Listings 4) specifies that any committed read-only (ro) transaction must observe any previously committed read-write (rw) transactions. Model checking found a 12-step counterexample to OBSERVEDROINV in four seconds. A ro transaction is handled by an old, yet active leader that has not added a rw transaction to its log since the new leader was elected. This is rare in practice, a leader has to be falsely replaced when it is still active. The leader's logs must be identical, then, in the short window of time before the old leader retires, it needs to handle a ro transaction.[5] Currently, we have no plans to change this behavior, as serializability for ro transactions is sufficient for most applications, however, we hope that our consistency spec will help us to more clearly communicate this guarantee to developers.

8 Lessons Learned

We were surprised that, despite our extensive testing, *many bugs were first spotted during the development of the spec and subsequent alignment with the implementation.* The development and refinement of the consensus and consistency specs forced us to think deeply about our protocol and its invariants. During the process of spec development, the implementation was very closely scrutinized (with the target invariants in mind) and many bugs were first identified. In some cases, someone would become suspicious of a particular part of the code but would be unable to confirm that the current behavior led to a violation. This was particularly true for situations including multiple reconfigurations and failure handling (see Incorrect election quorum tally §7 and Commit advance for previous term §7), where counterexamples require many steps and were no longer feasible to work through on a whiteboard. This is where verification was invaluable, as it allowed us to quickly check the behavior against the expected invariants.

While finding implementation bugs by gradually and manually aligning a formal spec with its implementation is possible, we found that *trace validation is a more systematic and efficient approach*. Prior to our trace validation efforts, which began in Spring 2023, we were not confident that our consensus spec matched the implementation. Different team members worked on the spec and the implementation, and as such, they reflected different understandings of how the consensus worked. Moreover, we corrected safety violations present only in the spec, while the implementation contained bugs that spec verification could not find. Trace validation, and its inclusion in our CI pipeline, proved to be a turning point in our verification efforts, as it finally allowed us to systematically identify and fix these discrepancies.

Our results show that *that software verification is beneficial even for systems that have already been "proven in production"*. During the time that the incorrect election

[5] The counterexample can be found and interactively explored online [66].

quorum tally (§7) was present in CCF, the operators added and removed nodes one at time. This meant that the bug did not lead to a safety violation and thus remained undiscovered, but could have surfaced if the operators had changed their reconfiguration strategy. Similarly, that the incorrect fix for the commit advance for previous term bug (§7) did not lead to a production incident can be attributed to chance.

Despite modern hardware and advances in tooling, we found that *exhaustive model checking, at our level of abstraction, took significant time to complete*. This led us to limit the state space more than we would have liked. We could have used a proof system, such as the TLA$^+$ Proof System, instead of a model checker. By opting for model checking, we chose to prioritize developer time and accessibility over compute time. As interactive theorem provers continue to advance [13, 76], along with AI-assisted verification [45], this trade-off may no longer be necessary. However, *simulation proved effective at quickly finding bugs*, especially when combined with action weighting (described in §4).

9 Related Work

Validating the correctness of distributed systems is a widely studied problem with approaches ranging from rigorously verified implementations, *formal verification*, to the many flavors of software testing, *casual verification*, including *smart casual verification* which sits between the two.

Formal verification of distributed systems provides the strongest guarantees of correctness, but is often impractical for real-world systems due to the high cost of development and expertise required. For example, IronFleet [24] and Verdi [92] both proved implementations of Raft correct, but, to the best of our knowledge, have not been used outside of an academic setting. Moreover, they are not easily amenable to systems already implemented in a general-purpose programming language. PGo [21] follows a related approach in which one could prove the correctness of a TLA$^+$ spec, and then extract a Go implementation using their PGo compiler. Again, an approach that works best for greenfield projects.

On the other hand, there is a broad spectrum of approaches to testing distributed systems (casual verification) [7, 34, 54], which tend to follow the same pattern: (i) orchestrate the creation of one or more configurations of the system, (ii) schedule workloads, and (iii) inject faults, such as network partitions, node failures, clock skew etc. As the extent of the system and its dependencies being orchestrated increases, it becomes more difficult to maintain determinism and repeatability. Test times for equivalent scenarios also tend to grow longer, and the likelihood of spurious failures goes up.

CCF's consensus spec is the latest addition in a recent tradition of modeling consensus algorithms in TLA$^+$. This began with Lamport's description of Paxos [47] in TLA$^+$, as a refinement of higher-level specs [88]. This was followed by the formalization in TLA$^+$ of other consensus algorithms including Paxos variants [20, 29, 30, 49–51], vanilla Raft [72, 74, 75, 81] and MongoDB's variant [82], Zab [35, 93] and Tendermint [86].

Tasiran et al. [85] were the first to extract and validate traces obtained from a hardware simulator against a TLA$^+$ spec, demonstrating the practical applicability of trace validation. The adoption of TLA$^+$ among distributed system practitioners, spurred by Newcombe et al. [70], and the formalization of trace validation as a refinement check by Pressler [77], caused trace validation to be applied to real-world distributed systems. For instance, Davis et al. [16] applied the technique to MongoDB, discovering a non-trivial implementation bug. However, they faced challenges in consistently logging implementation state, and aligning different grains of atomicity, which we attribute to them not leveraging TLA$^+$'s non-determinism to infer implementation state, and action composition to align atomicity. Niu et al. [71] also validated traces of Zookeeper, ensuring that its implementation corresponds to its spec. Similarly, Wang et al. [90] revealed several implementation bugs by replaying TLA$^+$ behaviors against instrumented implementations. Likewise, SandTable is capable of replaying behaviors but, additionally, provides a generic testing framework for distributed systems [84]. SandTable intercepts network communication at the POSIX layer, making it incompatible with systems like CCF that encrypt communications at the application layer. Furthermore, Wang et al. and SandTable serve as examples of the challenges of aligning the grains of atomicity, illustrated by the authors identifying two bugs in Ongaro's well-established Raft [74] spec. We contend that these are, in fact, common TLA$^+$ modeling patterns and can be handled with action composition. Nevertheless, all efforts found non-trivial bugs in real-world systems by comparing implementation traces to high-level TLA$^+$ behaviors; a testament to the effectiveness of this approach.

More pragmatic verification efforts are not limited to TLA+; Amazon's S3 ShardStore service was recently augmented with *lightweight verification* [5] using reference models written in Rust, which are simplified instantiations of program components that can be used to track program state under different input conditions. Like CCF, the primary goals are usability, and the ability to ensure correctness as both the implementation and the spec evolve over time. Unlike the CCF approach however, state exploration is limited to what the test harness is able to reach.

10 Conclusion

This report details our journey with smart casual verification of the distributed protocols in CCF using TLA$^+$. Our experience demonstrates that TLA$^+$ can be used in industrial settings to verify extensive and nuanced distributed protocols, and that the verification process can be integrated into the development workflow of a production codebase. We have seen that TLA$^+$ can be effectively utilized to find bugs in both the design and implementation of these protocols and to communicate understanding of complex and subtle distributed systems like CCF.

References

[1] Ittai Abraham, Guy Gueta, Dahlia Malkhi, Lorenzo Alvisi, Rama Kotla, and Jean-Philippe Martin. Revisiting fast practical byzantine fault tolerance, 2017. URL: https://arxiv.org/abs/1712.01367, arXiv:1712.01367.

[2] Brandon Amos and Huanchen Zhang. 15-812 term paper: Specifying and proving cluster membership for the Raft distributed consensus algorithm, 2015. [Last accessed: 2023-Nov-03]. URL: https://www.cs.cmu.edu/~aplatzer/course/pls15/projects/bamos.pdf.

[3] Panagiotis Antonopoulos, Arvind Arasu, Kunal D. Singh, Ken Eguro, Nitish Gupta, Rajat Jain, Raghav Kaushik, Hanuma Kodavalla, Donald Kossmann, Nikolas Ogg, Ravi Ramamurthy, Jakub Szymaszek, Jeffrey Trimmer, Kapil Vaswani, Ramarathnam Venkatesan, and Mike Zwilling. Azure SQL database always encrypted. In *Proceedings of the 2020 International Conference on Management of Data*, SIGMOD '20, pages 1511–1525, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3386141.

[4] Christian Berger, Hans P. Reiser, and Alysson Bessani. Making reads in BFT state machine replication fast, linearizable, and live. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 1–12, 2021. doi:10.1109/SRDS53918.2021.00010.

[5] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 836–850, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483540.

[6] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. Rollback and forking detection for trusted execution environments using lightweight collective memory. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 157–168, USA, 2017. IEEE Computer Society. URL: https://arxiv.org/pdf/1701.00981.pdf, doi:10.1109/DSN.2017.45.

[7] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 463–478, USA, 2020. USENIX Association. URL: https://www.usenix.org/system/files/nsdi20-paper-brooker.pdf.

[8] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 335–350, USA, 2006. USENIX Association. URL: https://www.usenix.org/legacy/event/osdi06/tech/full_papers/burrows/burrows.pdf.

[9] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, page 173–186, USA, 1999. USENIX Association. URL: https://pmg.csail.mit.edu/papers/osdi99.pdf.

[10] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, mar 1996. doi:10.1145/226643.226647.

[11] Horatiu Cirstea, Markus A. Kuppe, Benjamin Loillier, and Stephan Merz. Validating Traces of Distributed Programs Against TLA$^+$ Specifications, September 2024. arXiv:2404.16075.

[12] TLA Community. TLA+ google group. URL: https://groups.google.com/g/tlaplus.

[13] Coq. Coq formal proof management system, 2024. [Last accessed: 2024-Mar-25]. URL: https://github.com/coq/coq.

[14] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Paper 2016/086, 2016. URL: https://eprint.iacr.org/2016/086.

[15] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA$^+$ Proofs. *arXiv:1208.5933 [cs]*, August 2012. arXiv:1208.5933.

[16] A. Jesse Jiryu Davis, Max Hirschhorn, and Judah Schvimer. eXtreme Modelling in Practice. *Proceedings of the VLDB Endowment*, 13(9):1346–1358, May 2020. URL: https://dl.acm.org/doi/10.14778/3397230.3397233, doi:10.14778/3397230.3397233.

[17] Antoine Delignat-Lavaud, Cédric Fournet, Kapil Vaswani, Sylvan Clebsch, Maik Riechert, Manuel Costa, and Mark Russinovich. Why should I trust your code? Confidential Computing enables users to authenticate code running in TEEs, but users also need evidence this code is trustworthy. *Queue*, 21(4):94–122, sep 2023. doi:10.1145/3623460.

[18] Cambridge English Dictionary. Definition for smart casual, 2024. [Last accessed: 2024-May-02]. URL: https://dictionary.cambridge.org/dictionary/english/smart-casual.

[19] Open Enclave. Open Enclave SDK, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/openenclave/openenclave.

[20] Eli Gafni and Leslie Lamport. Disk Paxos. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, page 330–344, Berlin, Heidelberg, 2000. Springer-Verlag. URL: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Disk-Paxos.pdf.

[21] Finn Hackett, Shayan Hosseini, Renato Costa, Matthew Do, and Ivan Beschastnikh. Compiling Distributed System Models with PGo. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 159–175, Vancouver BC Canada, January 2023. ACM. doi:10.1145/3575693.3575695.

[22] Finn Hackett, Joshua Rowe, and Markus Alexander Kuppe. Understanding inconsistency in Azure Cosmos DB with TLA+. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '23, page 1–12. IEEE Press, 2023. doi:10.1109/ICSE-SEIP58684.2023.00006.

[23] Travis Hance, Andrea Lattuada, Chris Hawblitzel, Jon Howell, Rob Johnson, and Bryan Parno. Storage systems are distributed systems (so verify them that way!). In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. URL: https://www.usenix.org/system/files/osdi20-hance.pdf.

[24] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 60(7):83–92, jun 2017. doi:10.1145/3068608.

[25] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, jul 1990. doi:10.1145/78969.78972.

[26] Lorin Hochstein. Reading the Herlihy & Wing linearizability paper with TLA+, 2022. [Last accessed: 2024-May-05]. URL: https://github.com/lorin/tla-linearizability.

[27] Heidi Howard and Ittai Abrabam. Raft does not guarantee liveness in the face of network faults, 2020. [Last accessed: 2024-Feb-23]. URL: https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/.

[28] Heidi Howard, Fritz Alder, Edward Ashton, Amaury Chamayou, Sylvan Clebsch, Manuel Costa, Antoine Delignat-Lavaud, Cédric Fournet, Andrew Jeffery, Matthew Kerner, Fotios Kounelis, Markus A. Kuppe, Julien Maffre, Mark Russinovich, and Christoph M. Wintersteiger. Confidential Consortium Framework: Secure multiparty applications with confidentiality, integrity, and high availability. *Proc. VLDB Endow.*, 17(2):225–240, oct 2023. doi:10.14778/3626292.3626304.

[29] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible Paxos: Quorum Intersection Revisited. In *20th International Conference on Principles of Distributed Systems (OPODIS 2016)*, volume 70 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:14, Dagstuhl, Germany, 2017. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2016.25, doi:10.4230/LIPIcs.OPODIS.2016.25.

[30] Guanzhou Hu. Practical SMR-style TLA+ specification of the MultiPaxos protocol, 2024. [Last accessed: 2024-Apr-26]. URL: https://www.josehu.com/technical/2024/02/19/practical-MultiPaxos-TLA-spec.html.

[31] Intel. AMD SEV-SNP: Strengthening VM isolation with integrity protection and more, January 2020. [Last accessed: 2023-Oct-06]. URL: https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[32] Andrew Jeffery, Julien Maffre, Heidi Howard, and Richard Mortier. LSKV: A confidential distributed datastore to protect critical data in the cloud, 2024. URL: https://arxiv.org/abs/2407.12623, arXiv:2407.12623.

[33] Chris Jensen, Heidi Howard, and Richard Mortier. Examining Raft's behaviour during partial network failures. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*, HAOC '21, page 11–17, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3447851.3458739.

[34] Jepsen-io. Jepsen, 2024. [Last accessed: 2024-May-02]. URL: https://github.com/jepsen-io/jepsen.

[35] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, DSN '11, pages 245 – 256, USA, 2011. IEEE Computer Society. doi:10.1109/DSN.2011.5958223.

[36] Shubhra Sinha Kamath. Integrity protect your Azure blob storage data with Azure Confidential Ledger, 2024. [Last accessed: 2024-Feb-09]. URL: https://techcommunity.microsoft.com/t5/azure-storage-blog/integrity-protect-your-azure-blob-storage-data-with-azure/ba-p/4050754?WT.mc_id=DT-MVP-5001664.

[37] Martin Kleppmann. A critique of the CAP theorem. *CoRR*, abs/1509.05393, 2015. URL: http://arxiv.org/abs/1509.05393, arXiv:1509.05393.

[38] Igor Konnov, Jure Kukovec, and Thanh-Hai Tran. TLA+ model checking made symbolic. *Proc. ACM Program. Lang.*, 3(OOPSLA), oct 2019. doi:10.1145/3360549.

[39] Igor Konnov, Markus Kuppe, and Stephan Merz. Specification and Verification with the TLA$^+$ Trifecta: TLC, Apalache, and TLAPS. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*, volume 13701, pages 88–105. Springer International Publishing, Cham, 2022. doi:10.1007/978-3-031-19849-6_6.

[40] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. *ACM Trans. Comput. Syst.*, 27(4), jan 2010. doi:10.1145/1658357.1658358.

[41] Markus A. Kuppe. Teaching TLA$^+$ to Engineers at Microsoft. In Catherine Dubois and Pierluigi San Pietro, editors, *Formal Methods Teaching*, pages 66–81, Cham, 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-27534-0_5.

[42] Markus A. Kuppe. The TLA$^+$ Debugger. In Paolo Masci, Cinzia Bernardeschi, Pierluigi Graziani, Mario Koddenbrock, and Maurizio Palmieri, editors, *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops*, volume 13765, pages 174–180. Springer, 2023. doi:10.1007/978-3-031-26236-4_15.

[43] Markus Alexander Kuppe, Leslie Lamport, and Daniel Ricketts. The TLA+ Toolbox. In Rosemary Monahan, Virgile Prevosto, and José Proença, editors, *Proceedings Fifth Workshop on Formal Integrated Development Environment, F-IDE@FM 2019, Porto, Portugal, 7th October 2019*, volume 310 of *EPTCS*, pages 50–62, 2019. doi:10.4204/EPTCS.310.6.

[44] Hyperledger Labs. Hyperledger private data objects, 2022. URL: https://github.com/hyperledger-labs/private-data-objects.

[45] Guillaume Lample, Timothee Lacroix, Marie-Anne Lachaux, Aurelien Rodriguez, Amaury Hayat, Thibaut Lavril, Gabriel Ebner, and Xavier Martinet. Hypertree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems*, volume 35, pages 26337–26349, 2022. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/a8901c5e85fb8e1823bbf0f755053672-Paper-Conference.pdf.

[46] Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, may 1994. doi:10.1145/177492.177726.

[47] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 16(2):133 – 169, may 1998. doi:10.1145/279227.279229.

[48] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Professional, Pearson Education, USA, 2002. [Last accessed: 2023-Oct-06]. URL: https://lamport.azurewebsites.net/tla/book.html.

[49] Leslie Lamport. Fast Paxos. *Distrib. Comput.*, 19(2):79–103, oct 2006. doi:10.1007/s00446-006-0005-x.

[50] Leslie Lamport. Byzantizing Paxos by refinement. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, page 211–224, Berlin, Heidelberg, 2011. Springer-Verlag. doi:10.5555/2075029.2075058.

[51] Leslie Lamport. Byzantine Paxos, 2020. [Last accessed: 2023-Nov-03]. URL: https://lamport.azurewebsites.net/tla/byzpaxos.html.

[52] Leslie Lamport, Markus A. Kuppe, Stephan Merz, Andrew Helwer, William Schultz, Jeff Hemphill, Mariusz Ryndzionek, Igor Konnov, Thanh Hai Tran, Josef Widder, Jim Gray, Murat Demirbas, Guanzhou Hu, Giuliano Losa, Ron Pressler, Younes Akhouayri, Luming Dong, Zhi Niu, Lim Ngian Xin Terry, Gaurav Gandhi, Isaac DeFrain, Martin Harrison, Santhosh Raju, Cherry G. Mathew, Fransisca Andriani, and Ludovic Yvoz. TLA+ Examples. Version 1.0.0. URL: https://github.com/tlaplus/Examples.

[53] David Mazières and Dennis Shasha. Building secure file systems out of byzantine storage. In *Proceedings of the*

Twenty-First Annual Symposium on Principles of Distributed Computing, pages 108–117, Monterey California, July 2002. ACM. doi:10.1145/571825.571840.

[54] Ruijie Meng, George Pîrlea, Abhik Roychoudhury, and Ilya Sergey. Greybox fuzzing of distributed systems. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 1615–1629, New York, NY, USA, 2023. Association for Computing Machinery. doi:10.1145/3576915.3623097.

[55] Ralph C. Merkle. A digital signature based on a conventional encryption function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, CRYPTO '87, pages 369–378, Berlin, Heidelberg, 1987. Springer-Verlag. doi:10.1007/3-540-48184-2_32.

[56] Ellis Michael, Dan R. K. Ports, Naveen Kr. Sharma, and Adriana Szekeres. Recovering Shared Objects Without Stable Storage. In Andréa Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 36:1–36:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops-dev.dagstuhl.de/entities/document/10.4230/LIPIcs.DISC.2017.36, doi:10.4230/LIPIcs.DISC.2017.36.

[57] Microsoft. LSKV, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/LSKV.

[58] Microsoft. Microsoft Azure Confidential Ledger, 2022. [Last accessed: 2023-Oct-06]. URL: https://learn.microsoft.com/en-us/azure/confidential-ledger/overview.

[59] Microsoft. scitt-ccf-ledger, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/scitt-ccf-ledger.

[60] Microsoft. W3C DID for Confidential Consortium Framework, 2022. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/did-ccf.

[61] Microsoft. CCF - more efficient support for linearizable reads (#5636), 2023. [Last accessed: 2024-Apr-19]. URL: https://github.com/microsoft/CCF/issues/5636.

[62] Microsoft. Confidential Consortium Framework, 2023. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/CCF.

[63] Microsoft. Privacy sandbox key management system (kms) for azure, 2023. [Last accessed: 2024-Apr-19]. URL: https://github.com/microsoft/azure-privacy-sandbox-kms.

[64] Microsoft. SQL Server - ledger overview, 2023. [Last accessed: 2023-Nov-03]. URL: https://learn.microsoft.com/en-us/sql/relational-databases/security/ledger/ledger-overview.

[65] Microsoft. TLA+ specifications for the Confidential Consortium Framework, 2023. [Last accessed: 2023-Oct-06]. URL: https://github.com/microsoft/CCF/tree/main/tla.

[66] Microsoft. CCF - document non-linearizability of read-only transactions in rare system condition (#6167), 2024. [Last accessed: 2024-May-07]. URL: https://github.com/microsoft/CCF/pull/6167.

[67] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 358–372, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2517349.2517350.

[68] Suvam Mukherjee, Pantazis Deligiannis, Arpita Biswas, and Akash Lal. Learning-based controlled concurrency testing. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–31, November 2020. doi:10.1145/3428298.

[69] Chris Newcombe. Why Amazon chose TLA+. In *Proceedings of the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 8477*, ABZ 2014, page 25–39, Berlin, Heidelberg, 2014. Springer-Verlag. doi:10.1007/978-3-662-43652-3_3.

[70] Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, and Michael Deardeuff. How Amazon Web Services uses formal methods. *Communications of the ACM*, 58(4):66–73, March 2015. doi:10.1145/2699417.

[71] Zhi Niu, Luming Dong, Yong Zhu, and Li Chen. Verifying Zookeeper based on model-based runtime trace-checking using TLA$^+$. In *Proceedings of the 7th International Conference on Cyber Security and Information Engineering*, pages 13–18, Brisbane QLD Australia, September 2022. ACM. doi:10.1145/3558819.3558822.

[72] Diego Ongaro. *Consensus: Bridging Theory and Practice*. PhD thesis, Stanford, 2014. [Last accessed:

2023-Oct-06]. URL: https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf.

[73] Diego Ongaro. Email to raft-dev mailing list, subject line: bug in single-server membership changes, 2015. [Last accessed: 2023-Nov-03]. URL: https://groups.google.com/g/raft-dev/c/t4xj6dJTP6E.

[74] Diego Ongaro. TLA+ specifications for raft, 2020. [Last accessed: 2023-Nov-03]. URL: https://github.com/ongardie/raft.tla.

[75] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305 – 320, USA, 2014. USENIX Association. URL: https://www.usenix.org/system/files/conference/atc14/atc14-paper-ongaro.pdf.

[76] Lawrence C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1990.

[77] Ron Pressler. Conjunction Capers: A TLA$^+$ Truffle. https://conf.tlapl.us/2020/, September 2020. URL: https://www.youtube.com/watch?v=TP3SY0EUV2A&list=PLWLcqZLzY8u8EA8UlsZ5xKMvtUoeGr5R_&index=1.

[78] Mark Russinovich. Confidential computing: Elevating cloud security and privacy. *Commun. ACM*, 67(1):52–53, December 2023. doi:10.1145/3624577.

[79] Mark Russinovich, Manuel Costa, Cédric Fournet, David Chisnall, Antoine Delignat-Lavaud, Sylvan Clebsch, Kapil Vaswani, and Vikas Bhatia. Toward Confidential Cloud Computing. *Commun. ACM*, 64(6):54–61, May 2021. doi:10.1145/3453930.

[80] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, dec 1990. doi:10.1145/98163.98167.

[81] William Schultz, Edward Ashton, Heidi Howard, and Stavros Tripakis. Scalable, interpretable distributed protocol verification by inductive proof slicing, 2024. URL: https://arxiv.org/abs/2404.18048, arXiv:2404.18048.

[82] William Schultz, Ian Dardik, and Stavros Tripakis. Formal verification of a distributed dynamic reconfiguration protocol. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 143–152, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3497775.3503688.

[83] Pierre Sutra. On the correctness of Egalitarian Paxos. *Information Processing Letters*, 156:105901, 2020. URL: https://www.sciencedirect.com/science/article/pii/S002001901930184X, doi:https://doi.org/10.1016/j.ipl.2019.105901.

[84] Ruize Tang, Xudong Sun, Yu Huang, Yuyang Wei, Lingzhi Ouyang, and Xiaoxing Ma. SandTable: Scalable distributed system model checking with specification-level state exploration. In *Proceedings of the Nineteenth European Conference on Computer Systems*, pages 736–753, Athens Greece, April 2024. ACM. doi:10.1145/3627703.3650077.

[85] Serdar Tasiran, Yuan Yu, Brannon Batson, and Scott Kreider. Using formal specifications to monitor and guide simulation: Verifying the cache coherence engine of the Alpha 21364 microprocessor. In *Proceedings of the 3rd IEEE Workshop on Microprocessor Test and Verification, Common Challenges and Solutions*, 2002. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2002/06/mtv2002-alpha.pdf.

[86] Tendermint. Tendermint specifications, 2021. [Last accessed: 2024-Apr-26]. URL: https://github.com/tendermint/tendermint/tree/master/spec.

[87] TLA+. High-level TLA+ specifications for the five consistency levels offered by Azure Cosmos DB, 2023. [Last accessed: 2023-Nov-03]. URL: https://github.com/tlaplus/azure-cosmos-tla.

[88] TLA+. TLA+ specifications for Paxos, 2023. [Last accessed: 2023-Nov-03]. URL: https://github.com/tlaplus/Examples/tree/master/specifications/Paxos.

[89] TLA+. TLA+ proof manager (tlapm), 2024. [Last accessed: 2024-Mar-25]. URL: https://github.com/tlaplus/tlapm.

[90] Dong Wang, Wensheng Dou, Yu Gao, Chenao Wu, Jun Wei, and Tao Huang. Model checking guided testing for distributed systems. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 127–143, Rome Italy, May 2023. ACM. doi:10.1145/3552326.3587442.

[91] Michael Whittaker. EPaxos dependency set compaction bug, 2021. [Last accessed: 2023-Nov-03]. URL: https://github.com/mwhittaker/bipartisan_paxos/blob/master/epaxos_bugs/epaxos_dependency_bug.pdf.

[92] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for

implementing and formally verifying distributed systems. *SIGPLAN Not.*, 50(6):357–368, jun 2015. doi:10.1145/2813885.2737958.

[93] Jia-Qi Yin, Hui-Biao Zhu, and Yuan Fei. Specification and verification of the Zab protocol with TLA+. *J. Comput. Sci. Technol.*, 35(6):1312–1323, nov 2020. doi:10.1007/s11390-020-0538-7.

[94] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA$^+$ specifications. In Laurence Pierre and Thomas Kropf, editors, *10th IFIP WG 10.5 Conf. Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer. URL: https://www.microsoft.com/en-us/research/uploads/prod/2016/12/Model-Checking-TLA-Specifications.pdf.