A Formally Verified Protocol for Log Replication with Byzantine Fault Tolerance

Joel Wanner, Laurent Chuat, and Adrian Perrig
Network Security Group, Department of Computer Science, ETH Zurich, Switzerland

***Abstract*—Byzantine fault tolerant protocols enable state replication in the presence of crashed, malfunctioning, or actively malicious processes. Designing such protocols without the assistance of verification tools, however, is remarkably error-prone. In an adversarial environment, performance and flexibility come at the cost of complexity, making the verification of existing protocols extremely difficult. We take a different approach and propose a formally verified consensus protocol designed for a specific use case: secure logging. Our protocol allows each node to propose entries in a parallel subroutine, and guarantees that correct nodes agree on the set of all proposed entries, without leader election. It is simple yet practical, as it can accommodate the workload of a logging system such as Certificate Transparency. We show that it is optimal in terms of both required rounds and tolerable faults. Using Isabelle/HOL, we provide a fully machine-checked security proof based upon the Heard-Of model, which we extend to support signatures. We also present and evaluate a prototype implementation.**

***Index Terms*—Byzantine fault tolerance, consensus algorithm, formal verification**

I. Introduction

The problem of Byzantine consensus has been the subject of a considerable amount of research over the past decades, giving rise to various Byzantine fault tolerant (BFT) protocols, most notably Practical Byzantine Fault Tolerance (PBFT) by Castro and Liskov [1]. In response to the publication of PBFT, there have been many attempts to improve on the performance and robustness of the protocol by focusing on different scenarios. For instance, Zyzzyva [2] is designed to be especially efficient in the absence of failures, whereas Aardvark [3], on the contrary, is designed to react gracefully when failures occur.

These protocols were designed for high-throughput, low-latency state-machine replication. Unfortunately, this is only possible at the cost of complexity [4]. The BFT-SMaRt [5] library, which implements a variant of PBFT, can serve as a benchmark with almost 25,000 lines of Java code. Even in a benign fault model, where nodes can only crash and messages may be lost but not modified, distributed systems are notoriously hard to design and implement. In the presence of possibly malicious participants, arguing about the correctness and security of such protocols is an even greater challenge, or in the words of Lamport et al. [6]: "We know of no area in computer science or mathematics in which informal reasoning is more likely to lead to errors than in the study of this type of algorithm." To guarantee the security of such complex systems, a formal treatment is thus essential. The traditional approach in the distributed systems community is to provide a pen-and-paper proof for the desired properties of the protocol. At best, such proofs provide some intuition about why the claimed properties hold, but since they lack the rigor that is required to argue about such systems, they cannot be reasonably used as a guarantee. Past experiences, such as the Chord protocol [7], which had all of its hand-proved properties refuted by model checking, have shown that proofs must be *machine-checked*. Such proofs tend to be much longer and more detailed than their hand-crafted counterparts, but manual error can be ruled out conclusively using *proof assistants*.

To the best of our knowledge, there exists no complete machine-checked proof for any authenticated BFT protocol. Most work on verifying distributed systems has instead focused on consensus algorithms like Raft [8] and Paxos [9], which only tolerate benign faults. The IronFleet project [10] can serve as a benchmark for the complexity of large-scale verification efforts, as it expended approximately 3.7 person-years for the proof of a Paxos-based distributed system and its implementation. Due to the tremendous complexity of a Byzantine fault model caused by the introduction of arbitrary behavior, it is reasonable to assume that the effort of verifying a complex BFT protocol would require even more resources.

The lack of formal verification makes general-purpose BFT protocols unsuitable for security-critical applications, even if they have been tested and deployed in practice. To the best of our knowledge, the only instance of complete formal verification of a BFT protocol is by Debrat and Merz [11], who verified two algorithms proposed by Biely et al. [12] using the Isabelle/HOL [13] proof assistant. However, the properties provided by these very simple algorithms are too weak for use in many realistic settings.

Instead of aiming to develop a general-purpose BFT system, we focus on the use case of *secure logging*, a critical component in a variety of systems: modern public-key infrastructures [14, 15], online voting systems [16, 17], secure timestamping services [18, 19], and more [20]. In this endeavor, we make the following contributions:

- We present Logres, a BFT protocol designed specifically for secure log replication, and provide machine-checked proofs of all its properties using the Isabelle/HOL proof assistant. Our protocol model and proofs consist of approximately 300 and 1000 lines of code, respectively, and are available online [21]. Although the protocol is simple, our verification revealed subtle flaws in its initial design, which have since been fixed.

- We extend the Heard-Of model [22] to capture the concept of digital signatures. Our extended model can be used to verify other BFT algorithms that make use of signatures.
- We evaluate the performance of a prototype implementation to demonstrate that our protocol can be used in practical scenarios.

II. Background: Secure Logging

Logging can trivially be performed by a single server, but this server must then be trusted to (a) accept all valid requests, (b) not remove existing entries from the log, and (c) show the same view of the log to all clients. Verifiable data structures based on cryptographic primitives (such as Merkle hash trees) [23, 24] enable the efficient auditing of logs. This is, most notably, the approach employed in the Certificate Transparency (CT) framework [14]. Verifiable logging by itself is not sufficient though, as a malicious log server can still choose to ignore requests and show different views to different clients [25]. A log server ignoring specific requests is particularly problematic, because such misbehavior is hard to demonstrate and reporting it to a third party has privacy implications [26].

Relying on a single server has obvious drawbacks: weakest-link security, no resilience to failure, and no censorship resilience. On the other hand, relying on a large collection of non-synchronized log servers makes monitoring difficult. Indeed, a client cannot simply query one CT log server to inspect all entries related to a given domain name, for example, but must instead rely on monitors that keep entire copies of several logs. In turn, monitors must be trusted to correctly display all relevant entries from all trusted logs, which has revealed to be a challenge in practice [27]. For these reasons, we propose a protocol that allows independent entities to maintain a single log, thus providing resilience to compromise, failure, and censorship, while facilitating the monitoring of the log's contents by resource-limited clients.

A large majority of CT log servers accepted by Google Chrome have a "maximum merge delay" of 24 hours at the time of writing. This means that log servers will typically append newly submitted certificates to their hash tree within 24 hours. In such a context, our protocol would operate on a timescale that is perhaps unconventional for a distributed system, with each round of the protocol lasting several hours. However, we evaluate Logres within much smaller timeframes as well in Section VI, and find it to be able to support substantial workloads even with a 1-minute period.

III. Problem Definition

A. Log Replication with Byzantine Fault Tolerance

BFT protocols are commonly designed to achieve *state-machine replication*, where processes agree on an ordered set of incoming requests from clients, creating an input log that is equal on all processes. Running a deterministic state machine on the log then produces the same results on each node. The design goals in this problem space are usually low latency and high throughput, enabling the protocol to handle a high volume of requests quickly.

This paper considers the related but slightly different problem of BFT *log replication*. In this problem, a set of n nodes, of which at most f may fail, periodically run a distributed algorithm to maintain a log. There is an arbitrary number of clients in the system that can send messages to the nodes, requesting entries to be added to the log. Moreover, the clients can obtain the most recently created log along with an authenticator and verify the validity of its entries locally. In order to achieve log replication, a protocol must satisfy the following properties:

Agreement All valid logs created during a run of the protocol must be equal.

Completeness If an entry is submitted by a client to a correct node, the node will include it in its next log produced.

Liveness A run of the protocol must always produce a new valid log for every correct node.

This problem is different from BFT state-machine replication in three ways:

- There exists an inherent total order on entries (e.g., alphabetical or chronological). Therefore, no coordination is required to determine an ordering, unlike in the state-machine replication problem.
- Clients are not limited to obtaining the output of a state machine. Instead, they can verify the integrity of the entire log, or parts of it.
- The system aggregates entries and produces new outputs in fixed intervals, not in response to each request.

Due to these differences, the log replication problem allows for less complex solutions, as client requests do not need to be processed individually with low latency. Nevertheless, this problem appears in various real-world systems, such as public-key infrastructures.

B. Assumptions

We use the standard Byzantine fault model, where up to f of the n nodes may crash, malfunction, or even be actively malicious (and colluding). We call these nodes *faulty*, and there are at least $n - f$ remaining nodes that we call *correct*.

The protocol relies on the following assumptions, which are common for BFT protocols:

(A1) There is a correct majority of nodes, i.e., the constraint $n > 2f$ must hold.

(A2) Messages sent between correct nodes are neither lost nor modified, i.e., communication is synchronous.

(A3) Every node has a key pair and knows the public keys of all other nodes.

We stress that assumption (A2) is attainable in practice using loose time synchronization combined with a transport protocol that provides reliable and authenticated communication. Moreover, if a link between two nodes fails, the nodes can simply be considered faulty, and if the parameter f is chosen large enough, the protocol will be able to continue operating without any issues.

IV. The Logres Protocol

A. Definitions and Notation

A log contains an ordered sequence of entries as well as an expiration timestamp. The makeLog(L, X) function creates a new log from the previous log L and a set of new entries X. A log is *valid* if and only if it has not expired and is signed by $f+1$ different nodes. These signatures serve to verify the authenticity of a log and can be computed over a digest of it (using the getDigest(L) function, which may be a simple hash function, or may return the root of a hash tree, for example). Since there are at most f faulty nodes in the system, the requirement of $f+1$ signatures ensures that they cannot collude to forge a log without obtaining a signature from a correct node.

We use $\sigma_i(x)$ to denote a signature for x created by process i using its private key. $\langle x \,|\, i \rangle$ is shorthand for a signed term, i.e., $\langle x \,|\, i \rangle := (x, \sigma_i(x))$. Analogously, we use $\langle x \,|\, W \rangle$ for a term x that is signed by a set of nodes W, which are called witnesses.

B. Protocol Overview

The protocol is organized into rounds of communication, which can be implemented in asynchronous settings assuming loose time synchronization.

A key insight into how the protocol can be kept simple while still achieving strong security properties is that it does not require leader election. This concept allows low-latency BFT protocols to process requests quickly by designating one process as a leader, also called *primary*. While this can reduce the number of messages required, electing a leader also introduces a high amount of complexity, as the system needs to be able to handle cases where the current primary crashes or is actively malicious. For this purpose, processes can initiate a "view change" phase to convince others to hold a new election.

At the core of Logres lies the Consensus subroutine, a distributed consensus algorithm. While this subroutine is also based on a primary, we avoid the process of leader election by running n instances in parallel, such that each node is the leader of exactly one thread. The goal of this technique is to achieve *interactive consistency*, which Pease et al. [28] define as follows. Each node i chooses an initial value v_i, and the following properties must hold:

(IC1) All correct nodes agree on the same vector V of n values.

(IC2) For a correct node i, all correct nodes agree on i's initial value: $V_i = v_i$.

Our protocol operation is specified in detail in Algorithm 1 and illustrated in Figure 1. It consists of three phases:

1) **Collection** (1 round)
   Clients can send requests for entries to be added to the log, where each request should be sent to $f+1$ distinct nodes to guarantee that at least one correct node receives it. Each node stores all entries that it has received and that have not been added locally. During this phase, no communication takes place between the nodes.

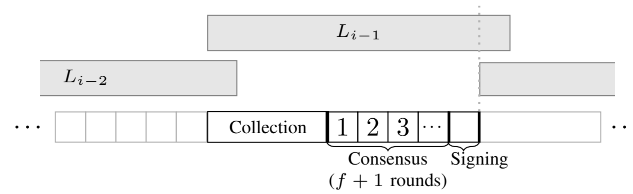

Fig. 1. An overview of the protocol operation. Each white rectangle represents a round of communication. Logs are characterized by their validity period. The exact definition of a validity period is left to application developers. Round lengths can also be tuned to each application.

Algorithm 1 Main protocol

1: **procedure** Logres
   Code for node i
   Collection phase:
2: $X \leftarrow$ collect entries from clients
   Consensus phase:
3: $\{X_1, \ldots, X_n\} \leftarrow \{\textsc{Consensus}(p, X) : 1 \leq p \leq n\}$
4: $X \leftarrow X_1 \cup \ldots \cup X_n$
   Signing phase:
5: $L \leftarrow$ log from previous epoch
6: $L' \leftarrow$ makeLog(L, X)
7: $h \leftarrow$ getDigest(L')
8: broadcast $\sigma_i(h)$
9: $\Sigma \leftarrow$ receive $\theta - 1$ valid signatures for h
10: publish $(L', \Sigma \cup \{\sigma_i(h)\})$

2) **Consensus** ($f+1$ rounds)
   With the set X of all cached entries from the previous phase as input, the Consensus subroutine is run in parallel n times. In each of these executions, a different node acts as the primary. The Consensus subroutine is explained in more detail in Section IV-C, but for the sake of this overview, we can assume that it achieves interactive consistency for initial values X and the result vector is $(X_1, \ldots, X_n)$. This implies that after line 4, all correct nodes obtain the same value for X, and their collected entries from the previous phase are contained in X.
3) **Signing** (1 round)
   Using the log from the previous run and the union of all new entries, each node constructs a new log.
   After the new log L' is constructed, each node broadcasts a signature for getDigest(L') to all other nodes.[1] Each node then collects signatures from f other participants that have also constructed the same log and publishes it along with all signatures.

In the following subsection, we describe in detail the Consensus subroutine, which is the main building block of the protocol, and explain how the consensus phase achieves interactive consistency.

[1]The log needs not be sent along with the signature, as the other correct nodes will have constructed the same log L'.

Algorithm 2 Consensus phase

1: **procedure** CONSENSUS(p, X)
Code for node i:
2: **if** $i = p$ **then** ▷ node is primary
3: broadcast $\langle X, p \,|\, i \rangle$
4: return X
5: **else** ▷ node is responder
6: $P \leftarrow \varnothing$ ▷ witnessed values
7: $d \leftarrow \varnothing$ ▷ decision value
8: **for** rounds $r = 1, \ldots, f+1$ **do**
9: $M \leftarrow$ receive messages
10: $M' \leftarrow \{\langle x, p \,|\, W \rangle \in M.\ p \in W \wedge |W| \geq r\}$
11: $P' \leftarrow \{x.\ \exists W.\ \langle x, p \,|\, W \rangle \in M'\}$
12: **if** $P' \setminus P \neq \varnothing$ **then**
13: **if** $|P \cup P'| = 1$ **then**
14: $d \leftarrow$ the only element of P'
15: **else**
16: $d \leftarrow \varnothing$
17: $S \leftarrow \varnothing$
18: **for** $x \in P' \setminus P$ **do**
19: $\langle x, p \,|\, W \rangle \leftarrow$ any element of M'
20: $S \leftarrow S \cup \{\langle x, p \,|\, W \cup \{i\} \rangle\}$
21: multicast S to other responders
22: $P \leftarrow P'$
23: **return** d

C. Consensus Phase

For the sake of abstraction, we will refer to a set of entries as a *value* and note that CONSENSUS is not limited to a specific type of value, but can be used more generally. The algorithm must satisfy the following two properties:

(C1) All correct nodes return the same value.
(C2) If the primary p is correct, all correct nodes return the primary's input value X.

If these properties hold for all parallel executions of CONSENSUS, it follows that the consensus phase satisfies (IC1) and (IC2). The main challenge in designing the algorithm does not lie in securing executions in which the primary is correct, but in preventing a faulty primary from causing disagreement between correct nodes. This is especially difficult because the primary may be actively malicious and colluding with all other faulty nodes.

Informally, the algorithm works as follows. Only the primary can propose new values, which it does by signing them. When a process receives such a value, it forwards the value to other nodes and testifies to witnessing it by appending its own signature. This is necessary for two reasons: (a) to inform nodes about the primary's value in case the message in the initial round was lost, and (b) to detect *equivocation* by the primary (i.e., sending different values in an attempt to create different views).

However, this technique alone is not enough to prevent attacks: A malicious primary could collude with responders and attempt to introduce a new value to some nodes in the last round, so the receivers will not have time to share it. To prevent this, correct nodes also apply the following policy: In round r, received values that have fewer than r witnesses are discarded. Any values sent in the last round must therefore be signed by $f+1$ nodes in order to be accepted, i.e., at least one correct node must have witnessed it previously.

After introducing the main concepts used in CONSENSUS, we now explain the subroutine in more detail and refer to the specification in Algorithm 2. In most situations, the execution of CONSENSUS is simple and requires only two rounds of communication. Such a normal case is depicted in Figure 2.

1) Primary: The procedure for the primary p is simple: p broadcasts its initial value in the first round and returns. A term of the form $\langle x, p \,|\, i \rangle$ represents the assertion "node i testifies to having witnessed value x from primary p". It is important to include p in the signature, as this prevents replay attacks using signatures received in different threads.

2) Responders: The other nodes, which we call responders, keep in their state the set P containing all values witnessed from the primary, as well as their current decision value d. In the following, we describe the behavior of a responder node with identifier i. During each round r, node i listens for any messages of the form $\langle x, p \,|\, W \rangle$ and stores them in the set M (discarding any invalid signatures). After a timeout, i advances to line 10 and selects from M all messages that are (a) signed by the primary, and (b) witnessed by a total of r nodes (including the primary). From these messages M', all distinct values are extracted and stored in P', which now contains all values that have been accepted this round.

If i has accepted any new values this round (i.e., $P' \setminus P \neq \varnothing$), it adjusts its decision based on the values it has accepted so far (line 13). If $|P \cup P'| = 1$ holds, i stores the new value in P' as its current decision. If there are more values in the two sets, i has proof of misbehavior by p because a correct primary will never send more than one value. In this case, i decides on the default value, which is the empty set.

In both of these two cases, i needs to testify to having witnessed the new values in $P' \setminus P$. For this purpose, a message m is assembled containing all new values with the corresponding signatures (lines 17–20). For each new value x, i adds a signature of its own to the message. m is then broadcast to all other responders.

Finally, after all $f+1$ rounds have completed, i concludes the consensus phase by returning its current decision d.

V. ANALYSIS

A. Informal Security Analysis

The goal of the following informal arguments is to provide intuition as to why our protocol satisfies the claimed properties. A more rigorous treatment using formal proofs is presented in Section VII.

The security arguments are structured such that the properties of CONSENSUS are discussed first. By building on these, we argue that LOGRES achieves agreement, completeness, and liveness. This is also the structure that our formal proof follows, as it allows for a more modular argument by treating the

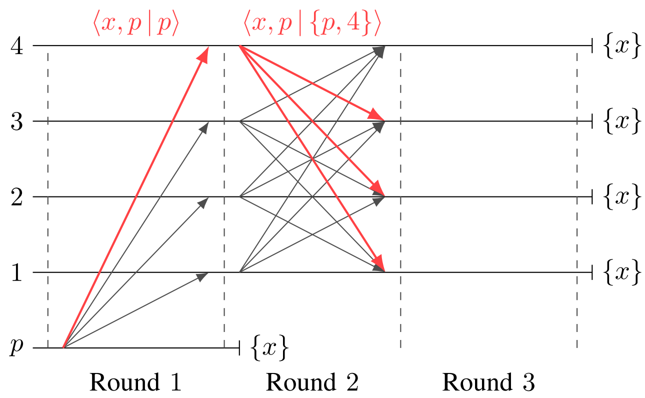

Fig. 2. The normal case of an execution of CONSENSUS with a correct primary p. There are $n = 5$ nodes, of which $f = 2$ may be faulty, which requires $f+1 = 3$ rounds of communication. In the first round, p broadcasts its initial value, and in the second round, each responder decides on this value and forwards it to others. No further communication is required in the last round.

two algorithms separately, with the properties (C1) and (C2) as the only interface between them.

1) Consensus phase: To show the validity property (C2), we can assume that the primary p is correct. In this case, all the primary does is broadcast its value X to all responders and decide on it. Since channels between correct nodes are reliable by Assumption (A2), all correct responders will receive X. Any other values will not be accepted by correct nodes, since the algorithm discards all values that are not signed by the primary. Therefore, all correct nodes will decide on X and validity is satisfied. This also implies the agreement property (C1) for the case of a correct primary.

We now show that agreement is also guaranteed in the presence of a faulty primary, which follows from two observations: (a) When a correct node accepts a new value, all other correct nodes will have accepted it in the next round. This is implied by a combination of (A2) with the fact that nodes sign and share all new values they accept. (b) No new values can be introduced into the system in the last round. This follows from the property that in this round, values require $f+1$ signatures to be accepted, so any new value must have been signed (and therefore shared with all other responders) by a correct node in an earlier round.

2) Logres: Property (C1) ensures that all correct nodes obtain the same set X. Along with the fact that makeLog(L, X) is a deterministic function, i.e., imposes some known order on the entries in X, this implies that all correct nodes produce the same log L'. There are at least $n-f \geq f+1$ correct nodes by assumption (A1), and because (A2) implies reliable channels between them, each of the correct nodes obtains signatures for L' from at least $f+1$ nodes (including themselves). This satisfies the Liveness property.

In order to satisfy the Agreement property, we not only need to show that all correct nodes produce the same log, but also that there can exist no other log. This follows from the assumption that there are at most f faulty nodes, and since $f+1$ are required for a valid log, this implies that they need the signature of at least one correct node to forge a log. However, since all correct nodes produce the same log, the faulty processes will not be able to create a different log.

Finally, we give some intuition about Completeness. Let i be a correct node to which the client sent its request. Then, i will include the requested entry x in its proposal X, which is given as input to the CONSENSUS subroutine. Using (C2), we have that x is contained in the set X_i. When combined with the agreement properties above, this implies that all correct nodes will include x in their new log.

3) Denial of Service: A common problem with BFT algorithms is that malicious actors may attempt to introduce a large number of values into the system in order to decrease the effective throughput of legitimate requests. Concretely, an attacker may send an excessive number of requests to a node such that it is not able to broadcast their value to all receivers within the duration of a round.

Mitigation against such attacks is specific to the application deployed on top of the protocol and is thus considered out of scope, but we envision countermeasures like rate limiting, client request authentication and duplicate suppression. These are all common techniques to strengthen resilience of systems to denial-of-service attacks.

B. Relation to Theoretical Bounds

Pease et al. [28] showed that Byzantine consensus for at most f faults requires at least $f+1$ rounds of communication. An intuition for this bound is as follows. A faulty node may send different values to different nodes (i.e., perform a split-world attack, also called equivocation). This is not necessarily a problem, since correct nodes can communicate with each other in the next round to try to reach the same state. However, we want to avoid that faulty nodes keep proposing new values ad infinitum. Therefore, we must require that messages be signed by more nodes in further rounds (i.e., at least i nodes in round i). Even with the above requirement, in any round $i \leq f$, faulty nodes can still make one correct node accept a new value. That correct node must be able to show the new value to other correct nodes in the next round, and the only thing it can do is add its own signature. Therefore, in round $i+1$, nodes should accept messages with just $i+1$ signatures. As a result, in the worst case, consensus will only be reached in round $f+1$.

Our protocol attains this lower bound because consensus is reached entirely within the consensus phase, which consists of $f+1$ rounds. The last round (the signing phase) only serves to collect enough signatures to publish the list that the nodes have agreed upon.

Moreover, Pease et al. [28] proved that if signatures can be used, consensus is possible for any $n > f$. While we assume that $n > 2f$, the consensus phase also reaches consensus for the weaker bound of $n > f$ (which can be shown by modifying the assumptions in our Isabelle/HOL proof). However, it is intuitively clear that the secure log replication problem is not solvable in the case of $n \leq 2f$: Consensus can be reached among the servers, but clients cannot reliably obtain the correct result in the presence of adversaries. Since a client cannot distinguish between faulty and correct nodes, the only way to ensure that the faulty participants do not forge a result from the protocol is to require a quorum of nodes to verify it. This is the purpose of requiring $\theta = f + 1$ signatures for a log to be valid. Therefore, the system always requires a correct majority of nodes, and the $n > 2f$ bound holds.

We thus conclude that our protocol is optimal with respect to both the number of rounds and the number of faulty nodes tolerated, considering the theoretical bounds on the problem of secure log replication.

VI. Evaluation

We have implemented a prototype of Logres in Go consisting of approximately 600 lines of code and used it in a series of experiments to evaluate the practical performance of the protocol. The experiments were conducted in a setting similar to that of Certificate Transparency (CT). In our setup, a varying number of nodes coordinate to keep an identical log of certificate entries. The workload is simulated using randomly generated strings of length equal to the average size of a CT log entry, which we determined to be approximately 1570 bytes. The period length (duration of a protocol run) was chosen as one minute. In order to determine if Logres can support a realistic workload, we compare our results to the average throughput of one of the largest CT logs to date, Google Argon 2019 [29].

The nodes were run in a virtual network created using Mininet on a single machine with two 8-core Intel Xeon E5-2680 processors running at 2.70 GHz and with 32 GB of RAM. In the virtual topology, each node was connected to the other hosts by a link with capacity 100 Mb/s.

A. Scaling

In this experiment, we measure the throughput of the system as the number of nodes increases, in order to evaluate the scalability of Logres. The results are displayed in Figure 3. We observe that performance diminishes as more nodes are added, which can be explained by the fact that messages need to be broadcast to other nodes. The uplink capacity limits the amount of data that can be sent during a round, and if this capacity must be distributed among more nodes, the maximum possible throughput decreases. Despite these effects, we observe that our system is able to support the load of a real-world CT log with up to 25 nodes.

We used a constant number $f = 2$ of tolerated faulty nodes in this experiment. This parameter constitutes a typical trade-off between performance and security (as discussed in Section VI-B).

B. Malicious Nodes

The parameter f determines how many faults can be tolerated and may be chosen up to a maximum of $\lceil \frac{n}{2} \rceil - 1$. However, larger values of f are also detrimental to performance. Moreover, even though we have proven the protocol secure for

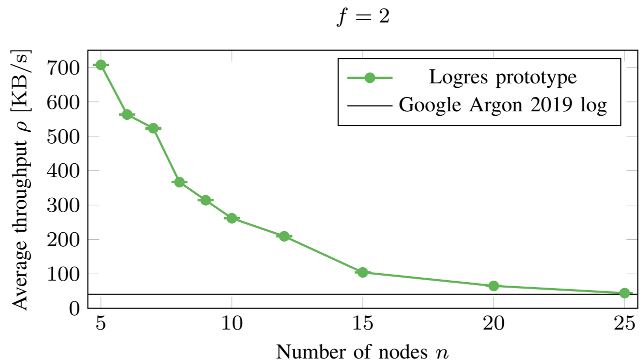

Fig. 3. Average throughput of our Logres prototype with $f = 2$, depending on the number of replicas n. For comparison, we indicate the average throughput of the (non-replicated) Google Argon 2019 CT log server, which processes 92,824 certificates per hour on average at the time of writing.

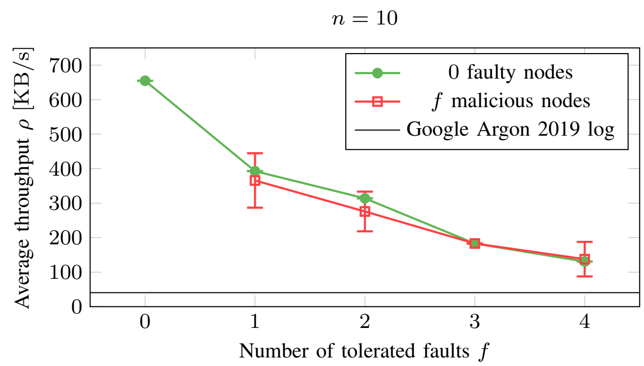

Fig. 4. The impact of increasing the parameter f for a fixed n in the case where all nodes are correct, compared to a simulation of malicious behavior.

these choices of f, adversaries may still be able to deteriorate the performance of the system by causing communication overhead. The purpose of this experiment is to measure the impact of these factors. For this purpose, we use a setup with $n = 10$ nodes, where measurements are conducted both with correct nodes and with a simulation of malicious behavior. The results are displayed in Figure 4.

1) Correct nodes: In the absence of faulty nodes, the parameter f has two effects on performance: The consensus phase consists of $f + 1$ rounds, so the rounds become shorter as f is increased. This limits the total size of entries that can be proposed with the same network bandwidth. Secondly, the clients are also affected by this parameter. In order to ensure that a request is not dropped by faulty nodes, a client should replicate it to $f + 1$ nodes, which amplifies the total data volume that needs to be exchanged during the protocol run.

2) Malicious behavior: The simulation of malicious behavior is based on a number of observations. The first is that in executions of Consensus with a correct primary, an adversary cannot cause the correct nodes to deviate from their regular behavior. This is because any value that is not signed by the primary is rejected immediately. Moreover, a malicious

TABLE I
TIME REQUIRED FOR A REQUEST TO BE PROCESSED, DEPENDING ON THE NUMBER OF TOLERATED FAULTS f, WITH $n = 5$ NODES.

f	Lower Bound	Measured Latency
0	$40.34\,\mathrm{ms}$	$44\,\mathrm{ms}$
1	$60.51\,\mathrm{ms}$	$66\,\mathrm{ms}$
2	$80.68\,\mathrm{ms}$	$88\,\mathrm{ms}$

primary can be blocked immediately by the responders once equivocation is detected, since the signatures created can be used to hold the primary accountable. Finally, we have discussed in Section V-A3 how denial-of-service attacks should be mitigated by the application running on top of Logres.

From these observations and the properties proved for the protocol, we can rule out many behavior patterns that allow the correct nodes to block the adversaries. The attack strategy we implemented works as follows: The faulty nodes generate some values, share them among each other and sign them. These values are then sent to the correct nodes in a later round. Since the messages carry more signatures than usual, this causes some communication overhead at the correct nodes.

In our implementation, signatures are 512 bytes long and thus small in comparison to the large volume of values that are exchanged. Therefore, we do not expect this attack to have a great impact. Our hypothesis is confirmed in Figure 4, which shows no significant difference in average system throughput between the normal case and the attack scenario. We therefore conclude that Logres performs well even in the presence of malicious participants that actively try to sabotage the protocol execution.

C. Latency

Finally, we investigate the minimum latency attainable by the protocol. For the previous experiments, the system was configured with a period length of one minute, but a shorter interval can be chosen for applications that are subject to stricter latency requirements.

We will determine the minimum time that is required for a request to be processed, i.e., from the moment the request is received by a server until it is added to a new version of the log. A lower bound on the round length can be computed using the network parameters between nodes and the size of requests. In this experiment, we choose links with a latency of $20\,\mathrm{ms}$ and a bandwidth $100\,\mathrm{Mb/s}$. Again, we use $1570\,\mathrm{B}$ values, which results in a minimum bound on round length of approximately $20.17\,\mathrm{ms}$ (the time required for transmission of a single value).

To measure the actual latency of the system, the clients submit a single request for each period, and we decrease the round length successively until the minimum value (for which the requested value is still processed reliably) is reached. We expect the measurements to exhibit slightly larger values than the theoretical lower bound due to the computational overhead due to message processing and signature verification.

The results from the experiment are displayed in Table I for 5 nodes and different values of f. The results show that our implementation is able to achieve small latencies, close to the theoretical lower bound.

VII. VERIFICATION

Our verification is based on the Heard-Of model, which was introduced by Charron-Bost and Schiper [22]. We first provide a brief overview of this model and how we extended it to account for signatures. Finally, this section contains a brief overview of our results and experiences with constructing the proof. We refer to the appendix for more details about the protocol formalization (Appendix A) and the proof methodology (Appendix B) we used in the process.

A. The Heard-Of Model

The Heard-Of model provides a framework for modeling consensus algorithms in a lock-step model of the system, where communication is divided into rounds. As the original model only accounted for benign faults, Biely et al. [12] extended it to capture malicious behavior. Our proofs are based on an implementation of this model in Isabelle/HOL by Debrat and Merz [11].

In this context, we will refer to nodes more generally as processes. The set of n processes is represented by Π, and M denotes the set of possible messages. We define $M_\perp := M \cup \{\perp\}$, where $\perp$ stands for the absence of a message. Each $i \in \Pi$ is associated with a process specification $P_i = (\mathcal{S}_i, \mathcal{S}_i^0, S_i, T_i)$ consisting of the following components:

- $\mathcal{S}_i$: the set of i's *states*, and $\mathcal{S}_i^0$: the set of possible *initial states* of process i.
- $S_i : \mathbb{N} \times \mathcal{S}_i \times \Pi \to M$: the *message sending function*, where $S_i(r, s, j)$ denotes the message sent by i to j in round r, given that i is in state s.
- $T_i : \mathbb{N} \times \mathcal{S}_i \times M_\perp^\Pi \times \mathcal{S}_i \to \{\text{true}, \text{false}\}$: the *next-state predicate*, where $T_i(r, s, \mu, s')$ evaluates to true if and only if s' is a state that i can reach from state s in round r, given the vector μ of messages that were received by i in round r.

The collection of process specifications P_i is called an *algorithm* on Π.

A *run* of such an algorithm is defined, for each process i, by a sequence of states $s_i^0, s_i^1, \ldots$ that satisfies the following conditions:

- s_i^0 is a valid initial state: $s_i^0 \in \mathcal{S}_i^0$
- For each $r \in \mathbb{N}$, it holds that the next-state predicate $T_i(r, s_i^r, \mu_i^r, s_i^{r+1})$ is true for a message vector μ_i^r collected from the messages sent by the other processes.

We now define μ_i^r, which not only depends on the specification of the algorithm, but also on the communication assumptions. For each process i and each round r, we define two subsets of Π:

- HO_i^r: the *heard-of set*, which places the following constraint on the message vector:

$$j \in HO_i^r \iff \mu_i^r[j] \neq \perp$$

This condition states that some message arrives at i from each process in HO_i^r, which may or may not conform to the process specification.

- SHO_i^r: the *safe heard-of set*, the set of processes whose message arrives unchanged:

$$j \in SHO_i^r \iff \mu_i^r[j] = S_j(r, s_j^r, i)$$

The message received by i is therefore guaranteed to be the one given by the protocol specification for j.

B. Extension to Signatures

The standard Byzantine fault model can be encoded by defining a subset $F \subseteq \Pi$ of Byzantine faulty nodes with $|F| \leq f$ and choosing the heard-of sets as $SHO_i^r = \Pi \setminus F \subseteq HO_i^r$. This corresponds to the assumption that the messages of correct nodes arrive according to process specification, and that faulty nodes may send any message in M or none at all. However, this property is generally too weak for a protocol that relies on digital signatures. This cryptographic primitive is commonly used under the assumption that signatures are unforgeable, which implies that faulty nodes must not be allowed to simply send arbitrary elements from the set M.

Instead, we introduce a restriction on messages sent by faulty nodes that encodes the unforgeable nature of signatures. While this could also be achieved by modeling Dolev-Yao intruder knowledge, we chose a simpler approach that requires fewer modifications to the original model. Let Sig denote the set of signatures and for each process i, let $\sigma_i : \Pi \mapsto 2^{Sig}$ be a function that given a message m, extracts the set of signatures contained in m that are signed by i. Furthermore, Σ_i^r is the set of all signatures sent by i in round r:

$$\Sigma_i^r := \bigcup_{j \in \Pi} \sigma_i(S_i(r, s_i^r, j))$$

From this, we define M_r as the set of all messages that can be constructed from signatures sent out until round r:

$$M_r := \{m \in M : \ \forall i \in C_r.\ \sigma_i(m) \subseteq \bigcup_{r'=0}^{r} \Sigma_i^r\}$$

where C_r denotes the set of processes that have not failed yet, i.e., processes that have been contained in all secure heard-of sets up to this round. Intuitively, this definition means that an agent that has stored all messages up to round r can use its knowledge to construct any message in M_r (e.g., including signatures received from correct nodes). Note that malicious nodes are not in the set C_r, and therefore signatures from them may appear in any messages. This allows for collusion between attackers, which may share their private keys over out-of-band channels.

Using these definitions, we can finally define the restriction on message vectors μ_i^r for all rounds r and processes i and j:

$$\mu_i^r[j] \in M_r \cup \{\perp\}$$

In summary, our extension of the HO model encodes the standard assumption that signatures are unforgeable, while still allowing faulty processes to sign arbitrary values and colluding with each other.

C. Verification Results

Our Isabelle/HOL verification of all security properties claimed in this paper is available online [21]. It consists of approximately 300 lines of code for the protocol model and 1000 lines of code for the proofs.

The verification process revealed two flaws in previous versions of the protocol. The first was based on missing isolation between instances of the CONSENSUS subroutine, which are invoked in parallel by Logres. Since signatures were not bound to a specific subroutine, a sophisticated attacker could have been able to replay legitimate signatures obtained from one thread in another thread, thereby preventing nodes from creating a valid log. This vulnerability was fixed by binding all signatures from a CONSENSUS routine to the corresponding primary.

In the second attack, an adversary could cause some nodes to exit from the consensus phase prematurely and thus again prevent the creation of a valid log.

These insights from our proofs provide further anecdotal evidence towards the conclusion that proofs must be machine-checked to provide strong guarantees. The vulnerabilities described above were not discovered during pen-and-paper construction of the proof, but only during the process of mechanizing it. This was detected when the proof assistant rejected some proof steps that were believed to be correct after informal reasoning.

The lessons learnt in this process and the practices we are recommending in Appendix B have also been identified in other verification projects [8, 30]. Since the advent of large-scale mechanized proofs, significant progress has been made in the direction of systematic proof engineering.

VIII. RELATED WORK

Numerous protocols have been designed to solve Byzantine agreement or BFT state-machine replication [1, 2, 3, 31, 32, 33, 34, 35], and the Abstract framework proposed by Guerraoui et al. [4] represents a notable step towards making the development of these protocols more modular and systematic. The problem of Byzantine agreement has been known since 1980, when it was first introduced by Pease et al. [28] as the "Byzantine Generals Problem". Dolev et al. [36] worked on signature-based protocols to solve this problem and introduced the principle of requiring values to have more signatures in later rounds, which was also used by Biely et al. [12].

To the best of our knowledge, there exists no full machine-checked proof of a state-of-the-art BFT protocol. The most notable steps in this direction have been proofs of safety properties for PBFT by Lamport [37] and more recently Rahli et al. [38]. Lamport used the TLA+ language in conjunction with the TLAPS proof system, while Rahli et al. developed an extension of EventML in the theorem prover Coq to verify the protocol. The main issue with both of these proofs is that they omit any liveness properties, which are commonly more challenging to prove than safety properties. We stress that these properties are just as important to the security of the protocol, especially when deployed in a environment that demands a reliable system. Another approach to verifying PBFT is based on Event-B and Rodin, but makes the assumption that messages cannot be forged [39]. A proof in this model can therefore not provide any guarantees in the Byzantine fault model.

More success has been achieved for consensus algorithms that tolerate benign (i.e., non-Byzantine) faults. In a breakthrough effort, the IronFleet project [10] fully verified a Paxos-based distributed system and its implementation using the Dafny verifier. The Raft [40] protocol is designed to achieve similar goals as the classic Paxos algorithm, but with the main design goal of understandability. Many properties of the system and its implementation have been formally verified in the Verdi framework for the Coq proof assistant, but liveness properties were not proved [8].

In the only instance of a complete proof for a BFT protocol that we are aware of, Debrat and Merz [11] verified two simple Byzantine consensus protocols in the Heard-Of model using the Isabelle/HOL theorem prover. Our work builds on these results, showing that similar methods can be applied to a more practical protocol that makes use of digital signatures.

Model checking is another method of verifying properties of protocols that can be used as an alternative to formal proofs. A model checker explores the state space of a system specification, which can be very useful for exposing design flaws without requiring a high amount of manual effort. However, this technique is ineffective at providing assurance of security for a system due to the constraints that combinatorial exploration places on the complexity of the model. Zieliński [41] applied model checking to Byzantine consensus protocols, but due to the tremendous size of the state space caused by Byzantine faults, the scope of the verification was limited to only a single round of communication.

In order to address problems related to logging schemes such as Certificate Transparency, Syta et al. [20] proposed to leverage multisignatures and distribute trust among many collective authorities or "cothorities". However, the rather complex protocols used by cothorities are proposed without any formal verification.

IX. Conclusion

This paper presents Logres, a secure log replication protocol. To the best of our knowledge, Logres is the first practical BFT protocol to be accompanied by a complete machine-checked security proof. This makes the protocol particularly suitable for security-critical applications that demand high resilience and reliability.

We have demonstrated that our protocol can achieve sufficient throughput for practical applications such as certificate logging, even in the presence of actively malicious participants. Concretely, Logres is able to support the load of the Certificate Transparency system for up to 25 nodes. Our system could be further improved by using a hash-tree based data store such as Trillian [42]. A threshold signature scheme [43] would also reduce the size of the signed logs produced by the protocol.

Our extension of the Heard-Of model to signatures can be used in formal verification of other consensus protocols that rely on this cryptographic primitive. Future work could capture other primitives such as message authentication codes and hash functions. Other interesting paths for future work include producing a verified implementation of Logres and considering application-specific optimizations.

X. Acknowledgments

We gratefully acknowledge support from ETH Zurich, and from the Zurich Information Security and Privacy Center (ZISC). Furthermore, we thank Kartik Nayak, Dahlia Malkhi, and David Kozhaya for their valuable feedback.

References

[1] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems*, vol. 20, no. 4, 2002.

[2] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative Byzantine fault tolerance," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[3] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti, "Making Byzantine fault tolerant systems tolerate Byzantine faults," in *NSDI*, 2009.

[4] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić, "The next 700 BFT protocols," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2010.

[5] A. Bessani, J. Sousa, and E. E. P. Alchieri, "State machine replication for the masses with BFT-SMART," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2014.

[6] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, Jul. 1982.

[7] P. Zave, "Using lightweight modeling to understand Chord," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 2, Mar. 2012.

[8] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson, "Planning for change in a formal verification of the Raft consensus protocol," in *Proceedings of the ACM Conference on Certified Programs and Proofs (CPP)*, 2016.

[9] L. Lamport, "Paxos made simple," *ACM Sigact News*, vol. 32, no. 4, 2001.

[10] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. Setty, and B. Zill, "IronFleet: proving practical distributed systems correct," in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2015.

[11] H. Debrat and S. Merz, "Verifying fault-tolerant distributed algorithms in the Heard-Of model," *Archive*

of Formal Proofs, 2012. [Online]. Available: https://isa-afp.org/entries/Heard_Of.html

[12] M. Biely, B. Charron-Bost, A. Gaillard, M. Hutle, A. Schiper, and J. Widder, "Tolerating corrupted communication," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, 2007.

[13] T. Nipkow, M. Wenzel, and L. C. Paulson, Eds., *Isabelle/HOL*. Springer Berlin Heidelberg, 2002.

[14] B. Laurie, A. Langley, and E. Kasper, "Certificate transparency," RFC 6962, Jun. 2013.

[15] D. Basin, C. Cremers, T. H.-J. Kim, A. Perrig, R. Sasse, and P. Szalachowski, "ARPKI: attack resilient public-key infrastructure," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2014.

[16] D. A. Gritzalis, "Principles and requirements for a secure e-voting system," *Computers and Security*, vol. 21, no. 6, 2002.

[17] N. Chondros, B. Zhang, T. Zacharias, P. Diamantopoulos, S. Maneas, C. Patsonakis, A. Delis, A. Kiayias, and M. Roussopoulos, "A distributed, end-to-end verifiable, internet voting system," *arXiv:1507.06812*, 2015.

[18] H. Massias, X. S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirement," in *Proceedings of the Symposium on Information Theory in the Benelux*, 1999.

[19] B. Gipp, N. Meuschke, and A. Gernandt, "Decentralized trusted timestamping using the crypto currency Bitcoin," *arXiv:1502.04015*, 2015.

[20] E. Syta, I. Tamas, D. Visher, D. I. Wolinsky, and B. Ford, "Decentralizing authorities into scalable strongest-link cothorities," *CoRR*, 2015.

[21] https://www.dropbox.com/s/w679b3xem11gjx4/proofs.tar.gz?dl=1, Isabelle/HOL source code.

[22] B. Charron-Bost and A. Schiper, "The Heard-Of model: computing in distributed systems with benign faults," *Distributed Computing*, Apr. 2009.

[23] A. Eijdenberg, B. Laurie, and A. Cutter, "Verifiable data structures," https://github.com/google/trillian/blob/master/docs/papers/VerifiableDataStructures.pdf.

[24] R. Dahlberg, T. Pulls, and R. Peeters, "Efficient sparse Merkle trees," in *Proceedings of the 21st Nordic Conference on Secure IT Systems (NordSec)*, 2016.

[25] L. Chuat, P. Szalachowski, A. Perrig, B. Laurie, and E. Messeri, "Efficient gossip protocols for verifying the consistency of certificate logs," in *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, September 2015.

[26] S. Eskandarian, E. Messeri, J. Bonneau, and D. Boneh, "Certificate Transparency with privacy," in *Proceedings on Privacy Enhancing Technologies*, 2017.

[27] B. Li, J. Lin, F. Li, Q. Wang, Q. Li, J. Jing, and C. Wang, "Certificate Transparency in the wild: Exploring the reliability of monitors," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[28] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM*, vol. 27, no. 2, Apr. 1980.

[29] Cloudflare, "Merkle Town," https://ct.cloudflare.com.

[30] G. Klein, M. Norrish, T. Sewell, H. Tuch, S. Winwood, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, and R. Kolanski, "seL4: formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS symposium on Operating systems principles (SOSP)*. ACM Press, 2009.

[31] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-scalable Byzantine fault-tolerant services," *ACM Operating Systems Review*, vol. 39, no. 5, 2005.

[32] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira, "HQ replication: a hybrid quorum protocol for Byzantine fault tolerance," in *Proceedings of the Symposium on Operating Systems Design and Implementation (OSDI)*, ser. OSDI '06, Berkeley, CA, USA, 2006.

[33] I. Abraham, D. Malkhi, K. Nayak, L. Ren, and M. Yin, "Sync hotstuff: Simple and practical synchronous state machine replication," Cryptology ePrint Archive, 2019.

[34] J. Katz and C.-Y. Koo, "On expected constant-round protocols for Byzantine agreement," in *Annual International Cryptology Conference*, 2006.

[35] I. Abraham, S. Devadas, D. Dolev, K. Nayak, and L. Ren, "Synchronous Byzantine agreement with expected O(1) rounds, expected O(n^2) communication, and optimal resilience," in *International Conference on Financial Cryptography and Data Security*, 2019.

[36] D. Dolev, M. J. Fischer, R. Fowler, N. A. Lynch, and H. R. Strong, "An efficient algorithm for Byzantine agreement without authentication," in *Information and Control*, 1982.

[37] L. Lamport, "Byzantizing Paxos by refinement," in *Lecture Notes in Computer Science*, 2011.

[38] V. Rahli, I. Vukotic, M. Völp, and P. Esteves-Verissimo, "Velisarios: byzantine fault-tolerant protocols powered by Coq," in *Programming Languages and Systems*, 2018.

[39] R. Krenický and M. Ulbrich, "Deductive verification of a Byzantine agreement protocol," Karlsruhe Institute of Technology, Department of Computer Science, Tech. Rep., 2010.

[40] D. Ongaro and J. K. Ousterhout, "In search of an understandable consensus algorithm," in *USENIX ATC*, 2014.

[41] P. Zieliński, "Automatic verification and discovery of Byzantine consensus protocols," in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2007.

[42] A. Eijdenberg, B. Laurie, and A. Cutter, "Verifiable data structures," Google Research, Tech. Rep., 2015.

[43] V. Shoup, "Practical threshold signatures," in *Advances in Cryptology (EUROCRYPT)*, 2000.

Appendix

A. Protocol Formalization

In the following, we show how Logres can be specified within the extension of the Heard-Of model presented in Section VII-B and provide formal specifications of the security properties stated in Section III.

There are two types of messages that are exchanged between nodes in Logres:

1) The first and most common one is used during the consensus phase and contains a list of value sets. Let $\mathcal{X}$ denote the domain of entries, and let $\mathcal{V}$ be the set of signatures over sets of values from $\mathcal{X}$, which can be modeled formally using the definition $\mathcal{V} := \Pi \times (\Pi \times 2^{\mathcal{X}})$ (consisting of the author, the identifier of the primary and the value to be signed). We call these signatures *votes*. The set of possible messages of this type is then given by $M_1 := \Pi \times 2^{\mathcal{V}}$, i.e., it carries a (possibly empty) set of votes for each primary node.
2) During the signing phase, each node sends out a single signature over a log. We refer to the domain of logs over entries from $\mathcal{X}$ as $\mathcal{L}_X$ and model the signatures over logs as $M_2 := \Pi \times \mathcal{L}_X$.

These two message domains are combined in the set $M := M_1 \cup M_2$.

Next, we specify the node states. Since every thread of the CONSENSUS subroutine requires its own state, we first define the possible *thread states* as $\mathcal{T} := 2^{\mathcal{X}} \times 2^{(2^{\mathcal{X}})} \times 2^{\mathcal{V}}$. This corresponds to the definitions in Algorithm 2, which consist of (a) the current decision $d \in 2^{\mathcal{X}}$, (b) the sets of values $P \in 2^{(2^{\mathcal{X}})}$ witnessed from the primary, and (c) the votes $M' \in 2^{\mathcal{V}}$ from the current round. A node state stores such a thread state for each primary in Π, as well as the set of entries collected from clients, the current version of the log and some signatures (from the set Σ) for it:

$$\begin{aligned} \mathcal{S}_i := (\!|\ & \mathit{threads} \in \Pi \times \mathcal{T}, \\ & \mathit{entries} \in 2^{\mathcal{X}}, \\ & \mathit{log} \in \mathcal{L}_{\mathcal{X}}, \\ & \mathit{sigs} \in 2^{M_2}\ |\!) \text{ for } i \in \Pi \end{aligned}$$

To simplify the notation, this is specified as a set of records, which are tuples whose components are named. We use the syntax $s.f$ to access the field with name f of a state s.

The initial states are defined as follows:

$$\begin{aligned} \mathcal{S}_i^0 := (\!|\ & \mathit{threads} = \Pi \times \{(\varnothing, \varnothing, \varnothing)\}), \\ & \mathit{entries} = 2^{\mathcal{X}}, \\ & \mathit{log} = \{L\}, \\ & \mathit{sigs} = \varnothing\ |\!) \text{ for } i \in \Pi \end{aligned}$$

where L is some initial log that is the same for all nodes.

For the sake of brevity, we omit definitions such as the message sending functions and state transition predicates. These are contained in the Isabelle theories, which are available online [21] and were used in constructing the proofs.

Using the above definitions, it is now possible to formalize the security properties of the protocol (defined in Section III):

Agreement *All valid logs created during a run of the protocol must be equal.*

$$\forall i, j \in \Pi \setminus F.\ s_i^{f+2}.\mathit{log} = s_j^{f+2}.\mathit{log}$$

Since the protocol consists of $f+2$ rounds, s_i^{f+2} is the final state of the protocol run for a node i.

Completeness *If an entry is submitted by a client to a correct node, the node will include it in its next log produced.*

$$\forall i \in \Pi \setminus F.\ s_i^1.\mathit{entries} \subseteq s_i^{f+2}.\mathit{log}$$

Clients and their requests are not modeled explicitly. Instead, each node uses the $\mathit{entries}$ field to store all received requests. We can therefore express the property as follows: all requested entries that a node has received before the end of the collection phase (i.e., $s_i^1.\mathit{entries}$) will be included in the log at the end of the protocol run.

Liveness *A run of the protocol must always produce a new valid log for every correct node.*

$$\forall i \in \Pi \setminus F.\ |s_i^{f+2}.\mathit{sigs}| \geq f+1$$

A log is considered valid if it is signed by at least $f+1$ distinct nodes.

B. Proofs in Isabelle/HOL

1) Using a Proof Assistant: Proof assistants enable the construction of complex models and machine-checked proofs. Their greatest advantage over other verifications tools is generality: they can be applied to practically any system. This is done by constructing proofs that are analogous to handwritten proofs, but provide much higher assurance since the proof assistant verifies its correctness, checking that each step of the proof is sound. Isabelle/HOL is a particular instance of this type of program, and its effectiveness has been proven in large-scale verification efforts such as that of the seL4 microkernel [30].

In order to provide some intuition about how proofs are constructed with the assistance of Isabelle/HOL, we show the code that is used to prove a simple statement about set theory, which is also used during our verification of Logres. It states that for any two sets A and B that are subsets of a finite domain C, if the sum of their cardinalities is greater than that of the domain, then A and B must overlap.

Isabelle can encode statements in higher-order logic (HOL), using the functional programming language ML, which is enriched by a large variety of standard mathematical definitions. In Isabelle's meta-language, the lemma stated above can be formalized as follows:

```
lemma overlap:
  assumes finC: finite C
  and subset: A ⊆ C ∧ B ⊆ C
  and cardAB: card A + card B > card C
  shows A ∩ B ≠ {}
```

The keywords **assumes**, **and**, **shows** are part of Isabelle's meta-language. They identify the premises and conclusions of the lemma, which can be named as in this example to improve readability of the code. Isabelle supports reasoning using declarative proofs in its proof language Isar, which closely resembles the structure of handwritten proofs. Moreover, Isabelle provides a rich library of common lemmas that can be used in proofs.

For this lemma, we can use a proof by contradiction. First, we assume that $A \cap B = \{\}$, from which we can deduce two contradicting results:

- $|A \cup B| > |C|$. Given that A and B do not intersect, $|A| + |B| = |A \cup B|$ holds. This follows from the built-in rule $card\text{-}Un\text{-}disjoint$, which is not defined here for the sake of brevity.
- $|A \cup B| \leq |C|$. This follows from the fact that since A and B are both subsets of C, it holds that $A \cup B \subseteq C$. The built-in rule $card\text{-}mono$ is used to convert this into the equation about cardinalities.

From these two statements, we can show $False$, which is a contradiction. The code for the complete proof in Isabelle is shown below:

```
proof
  assume nonempty: A ∩ B = {}
  from finC subset have finite A ∧ finite B
      using finite-subset by auto
  with nonempty cardAB have card (A ∪ B) > card C
    by (simp add: card-Un-disjoint)
  moreover have card (A ∪ B) ≤ card C
    using subset finC by (simp add: card-mono)
  ultimately show False by simp
qed
```

The detailed syntax and semantics of the proof are out of scope for this brief overview of Isabelle, but thanks to the structure that resembles the language of handwritten proofs, the steps are fairly comprehensible. For each step of the proof, an automated proof strategy (such as $auto$, $simp$, $\ldots$) is given using a **by** keyword, providing a hint to the proof assistant about how the step can be automatically verified. Once the machine is able to verify all individual steps, the lemma has been proved.

This particular lemma was chosen as an example due to its simplicity, but is also a part of our proofs, where we use it to show that a set of $f + 1$ nodes always contains at least

2) Proof Methodology: In the following, we provide a high-level description of the process of constructing our proofs for Logres. Initially, we formalized the protocol specification (given by pseudocode) in the HO model and specified the desired properties formally. We have found it an effective strategy for the construction of a sophisticated proof to proceed along the following basic steps:

one correct node. Our Isabelle theory contains a total of 104 lemmas, most of which are more complex to express and prove than the one shown above. However, this example serves as an overview of what to look for when performing high-level inspection of the proofs.

1) Write down a sequence of informal arguments for why the properties hold (similar to our analysis in Section V-A). The goal of this step is to identify key intermediate lemmas around which the proof will be constructed.
2) Formalize the intermediate lemmas from the previous step to obtain a rough outline in the proof assistant. Insert placeholders where the proof is incomplete (e.g., using the **sorry** keyword in Isabelle) to enable the assistant to check this outline.
3) Attempt to complete the unfinished proof steps, using existing lemmas and creating new ones where required. Repeat this step until all statements are proved.

This strategy is similar to a backward search technique, where the goal is inspected first and lemmas are created whenever they are needed to solve open goals.

Naturally, this description of the process is fairly idealized, as it is often necessary in practice to modify the original outline or even the protocol itself upon discovering that certain steps of the proof cannot be completed as expected. This is a common occurrence, as it is extremely difficult for a human to anticipate the precise outline of a proof before carrying out the steps in detail.

It is therefore necessary to ensure that the proofs are robust to changes and highly automated, such that a small change in some part of the proof does not require rewriting all of the remaining steps. For this purpose, we employ the design concept of modularization, which is standard folklore in the software engineering domain thanks to its benefits in creating maintainable code. In the case of a formal proof, this concept implies that the interfaces at which certain components of the model interact must be specified clearly and kept as abstract as possible. Isabelle provides useful concepts for this, called *locales* and *definitions*, which fulfill a similar role as abstract interfaces in software engineering. Well-designed modular proofs enable changing details of the protocol or lemmas without completely rewriting the proof.

An example of how our proof utilizes this concept is the CONSENSUS subroutine, which is invoked in parallel by Logres. It is natural to first prove the properties of the subroutine, providing an interface from which a proof for the properties for Logres to be constructed. While this composition appears to be simple, we discovered a possible attack where it was possible for an adversary to replay legitimate signatures across different threads, preventing the correct nodes from creating valid logs.