PoWerStore: Proofs of Writing for Efficient and Robust Storage

Dan Dobre[1] Ghassan O. Karame[1] Wenting Li[1] Matthias Majuntke[2] Neeraj Suri[3] Marko Vukolić[4]

[1]NEC Laboratories Europe
Heidelberg, 69115, Germany
{dan.dobre, ghassan.karame, wenting.li}@neclab.eu

[2]Capgemini Deutschland
Berlin, 10785, Germany
matthias.majuntke@capgemini.com

[3]TU Darmstadt
Darmstadt, 64289, Germany
suri@cs.tu-darmstadt.de

[4]EURECOM
Biot, 06410, France
vukolic@eurecom.fr

ABSTRACT

Existing Byzantine fault tolerant (BFT) storage solutions that achieve strong consistency and high availability, are costly compared to solutions that tolerate simple crashes. This cost is one of the main obstacles in deploying BFT storage in practice.

In this paper, we present PoWerStore, a robust and efficient data storage protocol. PoWerStore's robustness comprises tolerating network outages, maximum number of Byzantine storage servers, any number of Byzantine readers and crash-faulty writers, and guaranteeing high availability (wait-freedom) and strong consistency (linearizability) of read/write operations. PoWerStore's efficiency stems from combining lightweight cryptography, erasure coding and metadata write-backs, where readers write-back only metadata to achieve strong consistency. Central to PoWerStore is the concept of "Proofs of Writing" (PoW), a novel data storage technique inspired by commitment schemes. PoW rely on a 2-round write procedure, in which the first round writes the actual data and the second round only serves to "prove" the occurrence of the first round. PoW enable efficient implementations of strongly consistent BFT storage through metadata write-backs and low latency reads.

We implemented PoWerStore and show its improved performance when compared to existing robust storage protocols, including protocols that tolerate only crash faults.

Categories and Subject Descriptors

D.4.5 [**Operating Systems**]: Reliability – Fault tolerance.

Keywords

Byzantine-Fault Tolerance; Secure Distributed Storage; Strong Consistency.

CCS'13, November 4–8, 2013, Berlin, Germany.

http://dx.doi.org/10.1145/2508859.2516750.

1. INTRODUCTION

Byzantine fault-tolerant (BFT) distributed protocols have attracted considerable research attention, due to their appealing promise of masking various system issues ranging from simple crashes, through software bugs and misconfigurations, all the way to intrusions and malware. However, the use of existing BFT protocols is questionable in practice due to, e.g., weak guarantees under failures [41] or high cost in performance and deployment compared to *crash-tolerant* protocols [27]. This can help us derive the following requirements for the design of future BFT protocols:

- A BFT protocol should be *robust*, i.e., it should tolerate Byzantine faults and *asynchrony* (modeling network outages) while maintaining correctness (data consistency) and providing sustainable progress (availability) even under *worst-case* conditions that still meet the protocol assumptions. This requirement has often been neglected in BFT protocols that focus primarily on *common*, failure-free operation modes (e.g., [22]).
- Such a *robust* protocol should be *efficient*. For example, virtually all BFT protocols resort to *data write-backs*, in which a reader must ensure that a value it is about to return is propagated to a sufficient number of servers before a read completes—effectively, this entails repeating a write after a read [5] and results in performance deterioration. We believe that the efficiency of a robust BFT protocol is best compared to the efficiency of its crash-tolerant counterpart. Ideally, a robust protocol should not incur significant performance and resource cost penalty with respect to a crash-tolerant implementation, hence making the replacement of a crash-tolerant protocol a viable option.

In this paper, we present PoWerStore, a *robust* and *efficient* asynchronous BFT distributed read/write storage protocol. The notion of *robustness* subsumes [5]: *(i) high availability*, or *wait-freedom* [23], where read/write operations invoked by correct *clients* always complete, and *(ii) strong consistency*, or *linearizability* [24] of read/write operations.

At the heart of PoWerStore is a novel data storage technique we call *Proofs of Writing* (PoW). PoW are inspired by commitment schemes; PoW incorporate a 2-round write procedure, where the second round of write effectively serves to "prove" that the first round has actually been completed

before it is exposed to a reader. The second write round in PoW is lightweight and writes only metadata; however, it is powerful enough to spare readers of writing back the data allowing them to only write metadata to achieve strong consistency.[1] We construct PoW using cryptographic hash functions and efficient message authentication codes (MACs); in addition, we also propose an instantiation of PoW based on polynomial evaluation.

PoWerStore's efficiency is reflected in *(i) metadata write-backs* where readers write-back only metadata, avoiding expensive data write-backs, *(ii)* use of lightweight cryptographic primitives (such as hashes and MACs), *(iii) optimal resilience*, i.e., ensuring correctness despite the largest possible number t of Byzantine server failures; this mandates using $3t + 1$ servers [35]. Moreover, PoWerStore achieves these desirable properties with *optimal latency*: namely, we show that no robust protocol, including crash-tolerant ones, that uses a bounded number of servers and avoids data write-backs can achieve better latency than PoWerStore. More specifically, in the single writer (SW) variant of PoWerStore, this latency is two *rounds* of communication between a client and servers for both reads and writes.

In addition, PoWerStore employs *erasure coding* at the client side to offload the storage servers and to reduce network traffic. Furthermore, PoWerStore tolerates an unbounded number of Byzantine readers and unbounded number of writers that can crash.

Finally, while our SW variant of PoWerStore demonstrates the efficiency of PoW, for practical applications we propose a multi-writer (MW) variant of PoWerStore (referred to as M-PoWerStore). M-PoWerStore features 3-round writes and reads, where the third read round is invoked only under active attacks. M-PoWerStore also resists attacks specific to multi-writer setting that exhaust the timestamp domain [6]. We evaluate M-PoWerStore and demonstrate its superiority even with respect to existing crash-tolerant robust atomic storage implementations. Our results show that in typical settings, the peak throughput achieved by M-PoWerStore improves over existing robust crash-tolerant [5] and Byzantine-tolerant [34] atomic storage implementations, by 50% and 100%, respectively.

The remainder of the paper is organized as follows. In Section 2, we outline our system and threat model. Section 3 outlines the main intuition behind Proofs of Writing. In Section 4, we introduce PoWerStore and we analyze its correctness. In Section 5, we present the multi-writer variant of PoWerStore, M-PoWerStore. In Section 6, we evaluate an implementation of M-PoWerStore. In Section 7, we overview related work and we conclude the paper in Section 8.

2. MODEL

We consider a distributed system that consists of three *disjoint* sets of processes: a set *servers* of size $S = 3t + 1$, where t is the failure threshold parameter, containing processes $\{s_1, ..., s_S\}$; a set *writers* $w_1, w_2, ...$ and a set *readers* r_1, r_2, etc. The set *clients* is the union of writers and readers. We assume the *data-centric* model [12, 40] with bi-directional point-to-point channels between each client and each server. Servers do not communicate among each other, nor send messages other than in reply to clients' messages. In fact, servers do not even need to be aware of each other.

2.1 Threat Model

We model processes as probabilistic I/O automata [43] where a distributed algorithm is a set of such processes. Processes that follow the algorithm are called *correct*. Similar to [3, 4, 16, 34], we assume that any number of *honest* writers that may fail by crashing while allowing any number of readers and up to t servers to exhibit *Byzantine* [29] (or *arbitrary* [25]) faults.

We assume a strong adversary that can coordinate the Byzantine servers and readers to compromise the system. We further assume that the adversary controls the network and as such controls the scheduling of all transmitted messages in the network, resulting in *asynchronous* communication. However, we assume that the adversary cannot prevent the eventual delivery of messages between correct processes.

We opt not to focus on Byzantine writers because they can always obliterate the storage by constantly overwriting data with garbage, even in spite of (expensive) techniques that ensure that each individual write leaves a consistent state (e.g., [10, 19, 22, 31]. As a consequence, with Byzantine writers and asynchronous message schedule, the adversary can make a correct reader return arbitrary data at will. Therefore, we assume appropriate access control mechanisms to prevent untrusted writers from having appropriate credentials to modify data.

Finally, we assume that the adversary is computationally bounded and cannot break cryptographic hash functions or forge message authentication codes. In this context, we assume the existence of a cryptographic (i.e., one way and collision resistant) hash function $H(\cdot)$, and a secure message authentication function $MAC_k(\cdot)$, where k is a λ-bit symmetric key. We further assume that each server s_i pre-shares one symmetric group key with each writer in W; in the following, we denote this key by k_i.[2] Note that, in this paper, we do not address the confidentiality of the outsourced data.

2.2 Atomic Storage

We focus on a read/write storage abstraction [28] which exports two operations: WRITE(v), which stores value v and READ(), which returns the stored value. We assume that the initial value of a storage is a special value $\perp$, which is not a valid input value for a write operation. While every client may invoke the READ operation, we assume that WRITEs are invoked only by writers.

We say that an operation (invocation) *op* is *complete* if the client receives the *response*, i.e., if the client returns from the invocation. In any execution of an algorithm, we say that a complete operation op_2 *follows* op_1 if the invocation of op_2 follows the response of $op1$ in that execution. We further assume that each correct client invokes at most one operation at a time (i.e., does not invoke the next operation until it receives the response for the current operation).

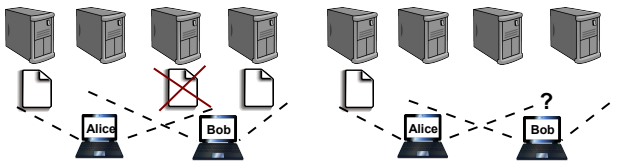
We focus on *robust* storage with the strongest storage progress consistency and availability semantics, namely *linearizability* [24] (or *atomicity* [28]) and *wait-freedom* [23].

[1] As proved in [17], in any robust storage readers must "write", i.e., modify the state of storage servers.

[2] Sharing group keys is not a requirement for the main functionality of the single-writer nor the multi-writer versions of PoWerStore. As we show in Section 5, this requirement is only needed to prevent a specific type of DoS attack where malicious readers exhaust the timestamp space.

	Construction costs	Verification costs
Hash-based PoW	1 hash	1 hash
Polynomial-based PoW	$O(\|t^3\|)$ modular exponentiations	$O(\|t^2\|)$ modular exponentiations
RSA signatures	$O(\|M\|)$ modular exponentiations	$O(\|M\|)$ modular exponentiations

Table 1: Construction and verification costs of our PoW instantiations. Here, t is the failure threshold, $M \gg t$ is the modulus used in RSA signatures.



(a) Complete Write (b) Incomplete Write

Figure 1: Complete vs. Incomplete writes ($t = 1$, $S = 4$).

Wait-freedom states that if a correct client invokes an operation *op*, then *op* eventually completes. Linearizability provides the illusion that a complete operation *op* is executed instantly at some point in time between its invocation and response, whereas the operations invoked by faulty clients appear either as complete or not invoked at all.

Finally, we measure the time-complexity of an atomic storage implementation in terms of number of *communication round-trips* (or simply *rounds*) between a client and servers. Intuitively, a *round* consists of a client sending the message to (a subset of) servers and receiving replies. A formal definition can be found in, e.g., [18].

3. PROOFS OF WRITING

In this section, we give the main intuition behind Proofs of Writing (PoW) and describe two possible instantiations of PoW: *(i)* a hash-based variant of PoW that offers computation security, and *(ii)* a polynomial evaluation-based PoW variant that provides information theoretic guarantees.

3.1 Intuition behind PoW

A distributed storage that tolerates failures and asynchrony must prevent clients from blocking while waiting for t possibly faulty servers to reply. As depicted in Figure 1(a), this implies that operations by clients must return after probing a quorum of $S - t$ servers. Intuitively, by looking at a strict subset of the servers, a reader cannot obtain a global view of the system state and in particular, differentiate a complete write from an incomplete write.

For example, in Figure 1, reader Alice cannot tell apart a complete write (Fig. 1(a)) where one Byzantine server deletes the data, from an incomplete write without Byzantine servers (Fig. 1(b)). To ensure strong consistency, i.e., that a subsequent read by Bob does not observe stale data (Fig. 1(b)), Alice must ensure that the data she is about to read is propagated to a sufficient number of servers before her read completes. Essentially, Alice must complete the write herself by performing a *data write-back*, involving a full write of the data she reads [5] in both executions. However, if she somehow knew that the write in Figure 1(a) was in fact complete, Alice could safely skip data writeback, since Bob would anyway observe recent data.

In the context of BFT storage, data write-backs by readers are undesirable for two reasons: *(i)* the overhead of cryptographic schemes related to preventing malicious readers from exploiting such write-backs to jeopardize the storage by overwriting data with garbage, and *(ii)* the inherent bandwidth and latency cost associated with writing-back data. In addition, when combined with erasure coded storage, data write-backs are computationally expensive since readers may need to erasure code data.

Essentially, since the data write-back technique is driven by readers' uncertainty in differentiating between a complete write and an incomplete write, we aim at an efficient technique that would allow readers to tell incomplete and complete writes apart. Such a technique would allow readers to safely discard incomplete writes altogether, obviating the need for writing-back data. Currently, the most widespread technique to achieve this differentiation would be to have writers send an authenticated acknowledgement (e.g., by using digital signatures) upon completion of the write.

At the heart of PoWerStore is a novel storage technique we call Proofs of Writing (PoW) which enables to achieve this differentiation more efficiently. PoW are inspired by commitment schemes [21]; similar to commitment schemes, PoW consist of two rounds: *(i)* in the first round, the writer commits to a random value of its choice, whereas *(ii)* in the second round, the writer "opens" his commitment. Unlike commitment schemes, PoW are constructed by honest writers and stored along with the data in a set of servers, enabling them to collectively convince a reader that the requested data has been stored in sufficiently many correct servers. Furthermore, we show that PoW can be constructed without relying on any public-key infrastructure while incurring little cost when compared to digital signatures (Table 1).

PoW obviate the need for writing-back data, allowing readers to write-back metadata. Metadata write-backs *(i)* help to prevent malicious readers from compromising the storage, and *(ii)* feature low communication latency and bandwidth usage even in worst-case conditions. By doing so, PoW provide an efficient alternative to digital signatures in BFT storage protocols. In what follows, we describe two efficient instantiations of PoW using cryptographic hashes, and polynomial evaluation; we also outline their relative performance gains when compared to digital signatures.

3.2 PoW based on Cryptographic Hashes

We start by outlining a PoW implementation that is based on the use of one-way collision-resistant functions seeded with pseudo-random input.

In the first WRITE round, the writer generates a pseudo-random nonce and writes the hash of the nonce together with the data in a quorum of servers. In the second WRITE round, the writer discloses the nonce and stores it in a quo-

rum. During the first round of a READ operation, the client collects the nonce from a quorum and sends (writes-back) the nonce to a quorum of servers in the second round. The server verifies the nonce by checking that the received nonce matches the hash of the stored nonce. If so, the server confirms the validity of the nonce by exposing the corresponding stored data to the client. The client obtains a PoW after receiving $t+1$ confirmations pertaining to a nonce.

Since the writer keeps the nonce secret until the start of the second round, it is computationally infeasible for the adversary to fabricate the nonce unless the first round of WRITE has completed, and hence the data is written. Thus, if the nonce received in the first READ round hashes to the stored hash at $t+1$ servers (one of which is necessarily correct), then this provides sufficient *proof* that the nonce has been disclosed by the writer, which implies that the data has been written.

3.3 PoW based on Polynomial Evaluation

In what follows, we propose an alternative construction of PoW based on polynomial evaluation. Here, at the start of every WRITE operation, the writer constructs a polynomial $P(\cdot)$ of degree t with coefficients $\{\alpha_t, \ldots, \alpha_0\}$ chosen at random from $\mathcal{Z}_q$, where q is a public parameter. That is, $P(x) = \sum_{j=0}^{j=t} \alpha_j x^j$.

The writer then constructs the PoW as follows: for each server s_i, the writer picks a random point x_i on $P(\cdot)$, and constructs the share (x_i, P_i), where $P_i = P(x_i)$. As such, the writer constructs S different shares, one for each server, and sends them to each server s_i over a *confidential channel*. Note that since there are at most t Byzantine servers, these servers cannot reconstruct the polynomial $P(\cdot)$ from their shares, even if they collude. In the second WRITE round, the writer reveals the polynomial $P(\cdot)$ to all servers. This enables a correct server s_i to establish that the first WRITE round has been completed by checking that the share (x_i, P_i) is indeed a point of $P(\cdot)$.

The argument of PoW relies on the assumption that the correct servers holding shares *agree* on the validity of the polynomial. Therefore, it is crucial to ensure that even after the disclosure $P(\cdot)$, the adversary cannot fabricate a polynomial $\hat{P(\cdot)} \neq P(\cdot)$ that intersects with $P(\cdot)$ in the share of a correct server. By relying on randomly chosen x_i, and the fact that correct servers never divulge their share, our construction prevents an adversary from fabricating $\hat{P(\cdot)}$.

Note that there exist other variant implementations for PoW that do not leak the polynomial to the servers [26]. We point that, unlike our prior solution based on cryptographic hash functions, this PoW construction provides information-theoretic guarantees [4, 30]. Table 1 illustrates the PoW construction and verification costs incurred in our PoW constructs. Owing to its reduced costs, we focus in this paper on hash-based PoWs (Sec. 3.2).

4. PoWerStore

In this section, we provide a detailed description of the PoWerStore protocol and we analyze its correctness. In Appendix A, we show that PoWerStore exhibits optimal worst-case latency.

4.1 Overview of PoWerStore

In PoWerStore, the WRITE operation performs in two consecutive rounds, called STORE and COMPLETE. Likewise, the READ performs in two rounds, called COLLECT and FILTER. For the sake of convenience, each round $rnd \in \{$STORE, COMPLETE, COLLECT, FILTER$\}$ is wrapped by a procedure *rnd*. In each round *rnd*, the client sends a message of type *rnd* to all servers. A round completes at the latest when the client receives messages of type *rnd*_ACK from $S-t$ correct servers. The server maintains a variable lc to store the metadata of the last completed WRITE, consisting of a timestamp-nonce pair, and a variable LC that stores a set of such tuples written-back by clients. In addition, the server keeps a variable $Hist$ storing the history, i.e., a log consisting of the data written by the writer[3] in the STORE round, indexed by timestamp.

Algorithm 1 Algorithm of the writer in PoWerStore.

```
1: Definitions:
2:    ts : structure num, initially ts_0 ≜ 0
3: operation WRITE(V)
4:    ts ← ts + 1
5:    N ← {0,1}^λ
6:    N̄ ← H(N)
7:    STORE(ts, N̄, V)
8:    COMPLETE(ts, N)
9:    return OK
10: procedure STORE(ts, V, N̄)
11:   {fr_1, ..., fr_S} ← encode(V, t+1, S)
12:   cc ← [H(fr_1), ..., H(fr_S)]
13:   for 1 ≤ i ≤ S do send STORE⟨ts, fr_i, cc, N̄⟩ to s_i
14:   wait for STORE_ACK⟨ts⟩ from S − t servers
15: procedure COMPLETE(ts, N)
16:   send COMPLETE⟨ts, N⟩ to all servers
17:   wait for COMPLETE_ACK⟨ts⟩ from S − t servers
```

4.2 Write Implementation

The WRITE implementation is given in Algorithm 1. To write a value V, the writer increases its timestamp ts, generates a nonce N and computes its hash $\overline{N} = H(N)$, and invokes STORE with ts, V and $\overline{N}$. When the STORE procedure returns, the writer invokes COMPLETE with ts and N. After COMPLETE returns, the WRITE completes.

In STORE, the writer encodes V into S fragments fr_i $(1 \leq i \leq S)$, such that V can be recovered from any subset of $t+1$ fragments. Furthermore, the writer computes a cross-checksum cc consisting of the hashes of each fragment. For each server s_i $(1 \leq i \leq S)$, the writer sends a STORE$\langle ts, fr_i, cc, \overline{N}\rangle$ message to s_i. On reception of such a message, the server writes $(fr_i, cc, \overline{N})$ into the history entry $Hist[ts]$ and replies to the writer. After the writer receives $S-t$ replies from different servers, the STORE procedure returns, and the writer proceeds to COMPLETE.

In COMPLETE, the writer sends a COMPLETE$\langle ts, N\rangle$ message to all servers. Upon reception of such a message, the server changes the value of lc to (ts, N) if $ts > lc.ts$ and replies to the writer. After the writer receives replies from $S-t$ different servers, the COMPLETE procedure returns.

4.3 Read Implementation

The READ implementation is given in Algorithm 3; it consists of the COLLECT procedure followed by the FILTER procedure. In COLLECT, the client reads the tuples (ts, N) in-

[3]Recall that PoWerStore is a single-writer storage protocol.

Algorithm 2 Algorithm of server s_i in PoWerStore.

18: **Definitions:**
19: lc : structure (ts, N), initially $c_0 \triangleq (ts_0, \text{NULL})$ //last completed write
20: LC : set of structure (ts, N), initially $\emptyset$ //set of written-back candidates
21: $Hist[\dots]$: vector of $(fr, cc, \overline{N})$ indexed by ts, with all entries initialized to (NULL, NULL, NULL)

22: **upon** receiving STORE$\langle ts, fr, cc, \overline{N}\rangle$ from the writer
23: $Hist[ts] \leftarrow (fr, cc, \overline{N})$
24: send STORE_ACK$\langle ts\rangle$ to the writer

25: **upon** receiving COMPLETE$\langle ts, N\rangle$ from the writer
26: **if** $ts > lc.ts$ **then** $lc \leftarrow (ts, N)$
27: send COMPLETE_ACK$\langle ts\rangle$ to the writer

28: **procedure** GC()
29: $c_{hv} \leftarrow \max(\{c \in LC : \mathsf{valid}(c)\} \cup \{c_0\})$
30: **if** $c_{hv}.ts > lc.ts$ **then** $lc \leftarrow c_{hv}$
31: $LC \leftarrow LC \setminus \{c \in LC : c.ts \leq lc.ts\}$

32: **upon** receiving COLLECT$\langle tsr\rangle$ from client r
33: GC()
34: send COLLECT_ACK$\langle tsr, LC \cup \{lc\}\rangle$ to client r

35: **upon** receiving FILTER$\langle tsr, C\rangle$ from client r
36: $LC \leftarrow LC \cup C$ //write-back
37: $c_{hv} \leftarrow \max(\{c \in C : \mathsf{valid}(c)\} \cup \{c_0\})$
38: $(fr, cc) \leftarrow \pi_{fr,cc}(Hist[c_{hv}.ts])$
39: send FILTER_ACK$\langle tsr, c_{hv}.ts, fr, cc\rangle$ to client r

40: **Predicates:**
41: $\mathsf{valid}(c) \triangleq (H(c.N) = Hist[c.ts].\overline{N})$

cluded in the set $LC \cup \{lc\}$ at the server, and accumulates these tuples in a set C together with the tuples read from other servers. We call such a tuple a *candidate* and C a *candidate set*. Before responding to the client, the server removes obsolete tuples from LC using the GC procedure. After the client receives candidates from $S-t$ different servers, COLLECT returns.

In FILTER, the client submits C to each server. Upon reception of C, the server performs a write-back of the candidates in C (*metadata write-back*). In addition, the server picks c_{hv} as the candidate in C with the highest timestamp such that $\mathsf{valid}(c_{hv})$ holds, or c_0 if no such candidate exists. The predicate $\mathsf{valid}(c)$ holds if the server, based on the history, is able to verify the integrity of c by checking that $H(c.N)$ equals $Hist[c.ts].\overline{N}$. The server then responds to the client with a message including the timestamp $c_{hv}.ts$, the fragment $Hist[c_{hv}.ts].fr$ and the cross-checksum $Hist[c_{hv}.ts].cc$. The client waits until $S-t$ different servers responded and either *(i)* $\mathsf{safe}(c)$ holds for the candidate with the highest timestamp in C, or *(ii)* all candidates have been excluded from C, after which COLLECT returns. The predicate $\mathsf{safe}(c)$ holds if at least $t+1$ different servers s_i have responded with timestamp $c.ts$, fragment fr_i and cross-checksum cc such that $H(fr_i) = cc[i]$. If $C \neq \emptyset$, the client selects the candidate with the highest timestamp $c \in C$ and restores value V by decoding V from the $t+1$ correct fragments received for c. Otherwise, the client sets V to the initial value $\perp$. Finally, the READ returns V.

4.4 Analysis

In what follows, we show that PoWerStore is robust, guaranteeing that READ/WRITE operations are linearizable and wait-free. We start by proving a number of core lemmas, to which we will refer to throughout the analysis.

DEFINITION 4.1 (VALID CANDIDATE). *A candidate c is* valid *if $\mathsf{valid}(c)$ holds at some correct server.*

LEMMA 4.2 (PROOFS OF WRITING). *If c is a valid candidate, then there exists a set Q of $t+1$ correct servers such that for each server $s_i \in Q$, $H(c.N) = Hist_i[c.ts].\overline{N}$.*

Proof: If c is valid, then by Definition 4.1, $H(c.N) = Hist_j[c.ts].\overline{N}$ holds at some correct server s_j. By the preimage resistance of H, no computationally bounded adversary can acquire $c.N$ from the sole knowledge of $H(c.N)$. Hence, $c.N$ stems from the writer in a WRITE operation wr with timestamp $c.ts$. By Algorithm 1, line 8, the value of $c.N$ is revealed only after the completion of the STORE round in wr. Hence, by the time $c.N$ is revealed, there is a set Q of $t+1$ correct servers such that each server $s_i \in Q$ assigned $Hist_i[c.ts].\overline{N}$ to $H(c.N)$. □

LEMMA 4.3 (NO EXCLUSION). *Let c be a valid candidate and let rd be a* READ *by a correct reader that includes c in C during* COLLECT. *Then c is never excluded from C.*

Proof: As c is valid, by Lemma 4.2 there is a set Q of $t+1$ correct servers such that each server $s_i \in Q$, $H(c.N) = Hist_i[c.ts]$. Hence, $\mathsf{valid}(c)$ is true at every server in Q. Thus, no server in Q replies with a timestamp $ts < c.ts$ in line 39. Therefore, at most $S-(t+1) = 2t$ timestamps received by the reader in the FILTER round are lower than $c.ts$, and so c is never excluded from C. □

We now explain why linearizability is satisfied by arguing that if a READ follows after a WRITE(V) (resp. after a READ that returns V) then the READ does not return a value older than V.

Read/Write Linearizability.

Suppose a READ rd by a correct client follows after a completed WRITE(V). If ts is the timestamp of WRITE(V), we argue that rd does not select a candidate with a timestamp lower than ts. Since a correct server never changes lc to a candidate with a lower timestamp, after WRITE(V) completed, $t+1$ correct servers hold a valid candidate with timestamp ts or greater in lc. During COLLECT in rd, the client awaits to receive the value of lc from $S-t$ different servers. Hence, a valid candidate c with timestamp ts or greater is received from at least one of these $t+1$ correct servers and included in C. By Lemma 4.3 (no exclusion), c is never excluded from C and by Algorithm 3, rd does not select a candidate with timestamp lower than $c.ts \geq ts$.

Read Linearizability.

Suppose a READ rd' by a correct client follows after rd. If c is the candidate selected in rd, we argue that rd' does not select a candidate with a timestamp lower than $c.ts$. By the time rd completes, $t+1$ correct servers hold c in LC. According to Algorithm 2, if a correct server removes c from

Algorithm 3 Algorithm of client r in PoWerStore.

42: **Definitions:**
43: tsr: num, initially 0
44: Q, R: set of pid, initially $\emptyset$
45: C: set of (ts, N), initially $\emptyset$
46: $W[1 \dots S]$: vector of (ts, fr, cc), initially []

47: **operation** READ()
48: $C, Q, R \leftarrow \emptyset$
49: $tsr \leftarrow tsr + 1$
50: $C \leftarrow$ COLLECT(tsr)
51: $C \leftarrow$ FILTER(tsr, C)
52: **if** $C \neq \emptyset$ **then**
53: $c \leftarrow c' \in C : \mathsf{highcand}(c') \wedge \mathsf{safe}(c')$
54: $V \leftarrow$ RESTORE$(c.ts)$
55: **else** $V \leftarrow \perp$
56: return V

57: **procedure** COLLECT(tsr)
58: send COLLECT$\langle tsr \rangle$ to all servers
59: **wait until** $|Q| \geq S - t$
60: return C

61: **upon** receiving COLLECT_ACK$\langle tsr, C_i \rangle$ from server s_i
62: $Q \leftarrow Q \cup \{i\}$
63: $C \leftarrow C \cup \{c \in C_i : c.ts > ts_0\}$

64: **procedure** FILTER(tsr, C)
65: send FILTER$\langle tsr, C \rangle$ to all servers
66: **wait until** $|R| \geq S - t \wedge$
$((\exists c \in C : \mathsf{highcand}(c) \wedge \mathsf{safe}(c)) \vee C = \emptyset)$
67: return C

68: **upon** receiving FILTER_ACK$\langle tsr, ts, fr, cc \rangle$ from server s_i
69: $R \leftarrow R \cup \{i\}$; $W[i] \leftarrow (ts, fr, cc)$
70: $C \leftarrow C \setminus \{c \in C : \mathsf{invalid}(c)\}$

71: **procedure** RESTORE(ts)
72: $cc \leftarrow cc'$ s.t. $\exists R' \subseteq R : |R'| \geq t + 1 \bigwedge$
$(\forall i \in R' : W[i].ts = ts \wedge W[i].cc = cc')$
73: $F \leftarrow \{W[i].fr : i \in R \wedge W[i].ts = ts \wedge H(W[i].fr) = cc[i]\}$
74: $V \leftarrow \text{decode}(F, t + 1, S)$
75: return V

76: **Predicates:**
77: $\mathsf{safe}(c) \triangleq \exists R' \subseteq R : |R'| \geq t + 1 \bigwedge$
$(\forall i \in R' : W[i].ts = c.ts) \bigwedge$
$(\forall i, j \in R' : W[i].cc = W[j].cc \wedge H(W[i].fr) = W[j].cc[i])$
78: $\mathsf{highcand}(c) \triangleq (c.ts = \max(\{c'.ts : c' \in C\}))$
79: $\mathsf{invalid}(c) \triangleq |\{i \in R : W[i].ts < c.ts\}| \geq S - t$

LC, then the server changed lc to a valid candidate with timestamp $c.ts$ or greater. As such, $t+1$ correct servers hold in $LC \cup \{lc\}$ a valid candidate with timestamp $c.ts$ or greater. During COLLECT in rd', the client awaits to receive the value of $LC \cup \{lc\}$ from $S-t$ different servers. As such, it includes a valid candidate c' with timestamp $c.ts$ or greater from at least one correct server. By Lemma 4.3 (no exclusion), c' is never excluded from C and by Algorithm 3, rd' does not select a candidate with timestamp lower than $c'.ts \geq c.ts$.

In the following, we argue that READ does not block in FILTER (line 66) while waiting for the candidate c with the highest timestamp in C to become $\mathsf{safe}(c)$ or C to become empty, which is the most sophisticated waiting condition.

Wait-freedom.

Suppose by contradiction that rd blocks during FILTER after receiving FILTER_ACK messages from all correct servers. We distinguish two cases: (Case 1) C contains a valid candidate and (Case 2) C contains no valid candidate.

- **Case 1:** Let c be the valid candidate with the highest timestamp in C. As c is valid, by Lemma 4.2, at least $t + 1$ *correct* servers hold history entries matching c. Since no valid candidate in C has a higher timestamp than $c.ts$, these $t + 1$ correct servers responded with timestamp $c.ts$, corresponding erasure coded fragment fr_i and cross-checksum cc such that $H(fr_i) = cc[i]$. Therefore, c is $\mathsf{safe}(c)$. Furthermore, all correct servers (at least $S-t$) responded with timestamps at most $c.ts$. Hence, every candidate $c' \in C$ such that $c'.ts > c.ts$ becomes $\mathsf{invalid}(c')$ and is excluded from C. As such, c is also $\mathsf{highcand}(c)$ and rd does not block.

- **Case 2:** Since no candidate in C is valid, all correct servers (at least $S - t$) responded with timestamp ts_0, which is lower than any candidate timestamp. As such, every candidate $c \in C$ becomes $\mathsf{invalid}(c)$ is excluded from C. Therefore, rd does not block.

5. M-PoWerStore

In what follows, we present the multi-writer variant of our protocol, dubbed M-PoWerStore. M-PoWerStore resists attacks specific to multi-writer setting that exhaust the timestamp domain [6]. Besides its support for multiple writers, M-PoWerStore protects against denial of service attacks specific to PoWerStore, in which the adversary swamps the system with bogus candidates. While this attack can be contained in PoWerStore by a robust implementation of the point-to-point channel assumption using, e.g., a separate pair of network cards for each channel (in the vein of [13]), this may impact practicality.

5.1 Overview

M-PoWerStore supports an unbounded number of clients. In addition, M-PoWerStore features optimal READ latency of two rounds in the *common case*, where no process is Byzantine. Under malicious attacks, M-PoWerStore gracefully degrades to guarantee reading in at most three rounds. The WRITE has a latency of three rounds, featuring non-skipping timestamps [6], which prevents the adversary from exhausting the timestamp domain.

The main difference between M-PoWerStore and PoWerStore is that, here, servers store and transmit a single candidate instead of a (possibly unbounded) set. To this end, it is crucial that servers are able to determine the validity of a written-back candidate without consulting the history. For this purpose, we enhance our original PoW scheme by extending the candidate with message authentication codes (MACs) on the timestamp and the nonce's hash, one for each server, using the corresponding group key. As such, a valid MAC proves to a server that the written-back candidate stems from a writer, and thus, constitutes a PoW that a server can obtain even without the corresponding history entry. Note that in case of a candidate incorporating corrupted MACs, servers might disagree about the validity of a written-back candidate. Hence, a correct client might not be able to write-back a candidate to $t + 1$ *correct* servers as needed. To solve this issue, M-PoWerStore "pre-writes" the MACs, enabling clients to recover corrupted candidates from the pre-written MACs.

To support multiple-writers, WRITE in M-PoWerStore comprises an additional distributed synchronization round, called

CLOCK, which is prepended to STORE. The READ performs an additional round called REPAIR, which is appended to COLLECT. The purpose of REPAIR is to recover broken candidates prior to writing them back, and is invoked only under attack by a malicious adversary that actually corrupts candidates.

Similarly to PoWerStore, the server maintains the variable $Hist$ to store the history of the data written by the writer in the STORE round, indexed by timestamp. In addition, the server keeps the variable lc to store the metadata of the last completed write consisting of the timestamp, the nonce and a vector of MACs (with one entry per server) authenticating the timestamp and the nonce's hash.

The full WRITE implementation is given in Algorithm 4. The implementation of the server and the READ operation are given in Algorithm 5 and 6. In the following, we simply highlight the differences to PoWerStore.

5.2 Write Implementation

As outlined before, M-PoWerStore is resilient to the adversary skipping timestamps. This is achieved by having the writer authenticate the timestamp of a WRITE with a key k_W shared among the writers. Note that such a shared key can be obtained by combining the different group keys; for instance, $k_W \leftarrow H(k_1||k_2||\dots)$.

To obtain a timestamp, in the CLOCK procedure, the writer retrieves the timestamp (held in variable lc) from a quorum of $S-t$ servers and picks the highest timestamp ts with a valid MAC. Then, the writer increases ts and computes a MAC for ts using k_W. Finally, CLOCK returns ts.

To write a value V, the writer, *(i)* obtains a timestamp ts from the CLOCK procedure, *(ii)* authenticates ts and the nonce's hash $\overline{N}$ by a vector of MACs vec, with one entry for each server s_i using group key k_i, and *(iii)* stores vec both in STORE and COMPLETE. Upon reception of a STORE$\langle fr_i, cc, \overline{N}, vec\rangle$ message, the server writes the tuple $(fr_i, cc, \overline{N}, vec)$ into the history entry $Hist[ts]$. Upon reception of a COMPLETE$\langle ts, N, vec\rangle$ message, the server changes the value of lc to (ts, N, vec) if $ts > lc.ts$.

5.3 Read Implementation

The READ consists of three consecutive rounds, COLLECT, FILTER and REPAIR. In COLLECT, a client reads the candidate triple (ts, N, vec) stored in variable lc in the server, and inserts it into the candidate set C together with the candidates read from other servers. After the client receives $S-t$ candidates from different servers, COLLECT returns.

In FILTER, the client submits C to each server. Upon reception of C, the server chooses c_{hv} as the candidate in C with the highest timestamp such that $\mathsf{valid}(c_{hv})$ is satisfied, or c_0 if no such candidate exists, and performs a write-back by setting lc to c_{hv} if $c_{hv}.ts > lc.ts$. Roughly speaking, the predicate $\mathsf{valid}(c)$ holds if the server verifies the integrity of the timestamp $c.ts$ and nonce $c.N$ either by the MAC, or by the corresponding history entry. The server then responds to the client with the timestamp $c_{hv}.ts$, the fragment $Hist[c_{hv}.ts].fr$, the cross-checksum $Hist[c_{hv}.ts].cc$ and the vector of MACs $Hist[c_{hv}.ts].vec$.

The client awaits responses from $S-t$ servers and waits until there is a candidate c with the highest timestamp in C such that $\mathsf{safe}(c)$ holds, or until C is empty, after which FILTER returns. The predicate $\mathsf{safe}(c)$ holds if at least $t+1$ different servers s_i have responded with timestamp $c.ts$, fragment fr_i, cross-checksum cc such that $H(fr_i) = cc[i]$, and vector vec. If C is empty, the client sets V to the initial value $\perp$. Otherwise, the client selects the highest candidate $c \in C$ and restores value V by decoding V from the $t+1$ correct fragments received for c.

In REPAIR, the client verifies the integrity of $c.vec$ by matching it against the vector vec received from $t+1$ different servers. If $c.vec$ and vec match then REPAIR returns. Otherwise, the client repairs c by setting $c.vec$ to vec and invokes a round of write-back by sending a REPAIR$\langle tsr, c\rangle$ message to all servers. Upon reception of such a message, if $\mathsf{valid}(c)$ holds then the server sets lc to c provided that $c.ts > lc.ts$ and responds with an REPAIR_ACK message to the client. Once the client receives acknowledgements from $S-t$ different servers, REPAIR returns. After REPAIR returns, the READ returns V.

Algorithm 4 Algorithm of writer w in M-PoWerStore.

```
80: Definitions:
81:     Q: set of pid, (process id) initially ∅
82:     ts: structure (num, pid, MAC_{k_W}(num||pid)),
        initially ts_0 ≜ (0, 0, NULL)

83: operation WRITE(V)
84:     Q ← ∅
85:     ts ← CLOCK()
86:     N ← {0,1}^λ
87:     N̄ ← H(N)
88:     vec ← [MAC_{k_i}(ts||N̄)_{1≤i≤S}]
89:     STORE(ts, V, N̄, vec)
90:     COMPLETE(ts, N, vec)
91:     return OK

92: procedure CLOCK()
93:     send CLOCK⟨ts⟩ to all servers
94:     wait until |Q| ≥ S − t
95:     ts.num ← ts.num + 1
96:     ts ← (ts.num, w, MAC_{k_W}(ts.num||w))
97:     return ts

98: upon receiving CLOCK_ACK⟨ts, ts_i⟩ from server s_i
99:     Q ← Q ∪ {i}
100:    if ts_i > ts ∧ verify(ts_i, k_W) then ts ← ts_i

101: procedure STORE(ts, V, N̄, vec)
102:    {fr_1, ..., fr_S} ← encode(V, t + 1, S)
103:    cc ← [H(fr_1), ..., H(fr_S)]
104:    foreach server s_i send STORE⟨ts, fr_i, cc, N̄, vec⟩ to s_i
105:    wait for STORE_ACK⟨ts⟩ from S − t servers

106: procedure COMPLETE(ts, N, vec)
107:    send COMPLETE⟨ts, N, vec⟩ to all servers
108:    wait for COMPLETE_ACK⟨ts⟩ from S − t servers
```

5.4 Analysis

We argue that M-PoWerStore satisfies linearizability by showing that if a completed READ rd by a correct client returns V then a subsequent READ rd' by a correct client does not return a value older than V. The residual correctness arguments are similar to those of PoWerStore (Section 4.4), and therefore omitted.

Suppose rd' follows after rd. If c is the candidate selected in rd, we argue that rd' does not select a candidate with a lower timestamp. By assumption c is selected in rd by a correct client that checks the integrity of $c.vec$ (during REPAIR). If $c.vec$ passes the check, then each of the correct servers (at least $S-2t \geq t+1$) that received c during FILTER

Algorithm 5 Algorithm of server s_i in M-PoWerStore.

```
109: Definitions:
110:    lc: structure (ts, N, vec), initially c_0 ≜ (ts_0, NULL, NULL)
111:    Hist[...]: vector of (fr, cc, N̄, vec) indexed by ts, with all
        entries initialized to (NULL, NULL, NULL, NULL)

109: upon receiving CLOCK⟨ts⟩ from writer w
110:    send CLOCK_ACK⟨ts, lc.ts⟩ to writer w

111: upon receiving STORE⟨ts, fr, cc, N̄, vec⟩ from writer w
112:    Hist[ts] ← (fr, cc, N̄, vec)
113:    send STORE_ACK⟨ts⟩ to writer w

114: upon receiving COMPLETE⟨ts, N, vec⟩ from writer w
115:    if ts > lc.ts then lc ← (ts, N, vec)
116:    send COMPLETE_ACK⟨ts⟩ to writer w

117: upon receiving COLLECT⟨tsr⟩ from client r
118:    send COLLECT_ACK⟨tsr, lc⟩ to client r

119: upon receiving FILTER⟨tsr, C⟩ from client r
120:    c_hv ← max({c ∈ C : valid(c)} ∪ {c_0})
121:    if c_hv.ts > lc.ts then lc ← c_hv                    //write-back
122:    (fr, cc, vec) ← π_{fr,cc,vec}(Hist[c_hv.ts])
123:    send FILTER_ACK⟨tsr, c_hv.ts, fr, cc, vec⟩ to client r

124: upon receiving REPAIR⟨tsr, c⟩ from client r
125:    if c.ts > lc.ts ∧ valid(c) then lc ← c               //write-back
126:    send REPAIR_ACK⟨tsr⟩ to client r

127: Predicates:
128: valid(c) ≜ (H(c.N) = Hist[c.ts].N̄) ∨
                verify(c.vec[i], c.ts, H(c.N), k_i)
```

validates c (by verifying its own entry of $c.vec$) and sets lc to c unless it has already changed lc to a higher timestamped candidate. Otherwise, if $c.vec$ fails the integrity check, then the client in rd repairs $c.vec$ and subsequently writes-back c to $t+1$ correct servers or more. Hence, by the time rd completes, at least $t+1$ correct servers have set lc to c or to a valid candidate with a higher timestamp. Since during COLLECT in rd' the client receives the value of lc from $S-t$ different servers, a valid candidate c' such that $c'.ts \geq c.ts$ is included in C. By Lemma 4.3 (no exclusion), c' is never excluded from C and by Algorithm 6, rd' does not select a candidate with timestamp lower than $c'.ts \geq c.ts$.

6. IMPLEMENTATION & EVALUATION

In this section, we describe an implementation modeling a Key-Value Store (KVS) based on M-PoWerStore. To model a KVS, we use multiple instances of M-PoWerStore referenced by keys. We then evaluate the performance of our implementation and we compare it to: *(i)* M-ABD, the multi-writer variant of the crash-only atomic storage protocol of [5], and *(ii)* Phalanx, the multi-writer robust atomic protocol of [34] that relies on digital signatures.

6.1 Implementation Setup

Our KVS implementation is based on the JAVA-based framework Neko [2] that provides support for inter-process communication, and on the Jerasure [1] library for constructing the dispersal codes. To evaluate the performance of our M-PoWerStore we additionally implemented two KVSs based on M-ABD and Phalanx.

In our implementation, we relied on 160-bit SHA1 for hashing purposes, 160-bit keyed HMACs to implement MACs,

Algorithm 6 Algorithm of client r in M-PoWerStore.

```
129: Definitions:
130:    tsr: num, initially 0
131:    Q, R: set of pid, initially ∅
132:    C: set of (ts, N, vec), initially ∅
133:    W[1...S]: vector of (ts, fr, cc, vec), initially []

134: operation READ()
135:    C, Q, R ← ∅
136:    tsr ← tsr + 1
137:    C ← COLLECT(tsr)
138:    C ← FILTER(tsr, C)
139:    if C ≠ ∅ then
140:       c ← c' ∈ C : highcand(c') ∧ safe(c')
141:       V ← RESTORE(c.ts)
142:       REPAIR(c)
143:    else V ← ⊥
144:    return V
   ...
145: upon receiving COLLECT_ACK⟨tsr, c_i⟩ from server s_i
146:    Q ← Q ∪ {i}
147:    if c_i.ts > ts_0 then C ← C ∪ {c_i}
   ...
148: upon receiving FILTER_ACK⟨tsr,ts,fr,cc,vec⟩ from server s_i
149:    R ← R ∪ {i}; W[i] ← (ts, fr, cc, vec)
150:    C ← C \ {c ∈ C : invalid(c)}
   ...
151: procedure REPAIR(c)
152:    vec ← vec' s.t. ∃R' ⊆ R : |R'| ≥ t+1 ⋀
        (∀i ∈ R' : W[i].ts = c.ts ∧ W[i].vec = vec')
153:    if c.vec ≠ vec then
154:       c.vec ← vec                                       //repair
155:       send REPAIR⟨tsr, c⟩ to all servers
156:       wait for REPAIR_ACK⟨tsr⟩ from S − t servers

157: Predicates:
158:    safe(c) ≜ ∃R' ⊆ R : |R'| ≥ t+1 ⋀
        (∀i ∈ R' : W[i].ts = c.ts) ⋀
        (∀i, j ∈ R' : W[i].cc=W[j].cc ∧ H(W[i].fr)=W[j].cc[i]) ⋀
        (∀i, j ∈ R' : W[i].vec = W[j].vec)
   ...
```

and 1024-bit DSA to generate signatures[4]. For simplicity, we abstract away the effect of message authentication in our implementations; we argue that this does not affect our performance evaluation since data origin authentication is typically handled as part of the access control layer in all three implementations, when deployed in realistic settings (e.g., in Wide Area Networks).

We deployed our implementations on a private network consisting of a 4-core Intel Xeon E5443 with 4GB of RAM, a 4 core Intel i5-3470 with 8 GB of RAM, an 8 Intel-Core i7-37708 with 8 GB of RAM, and a 64-core Intel Xeon E5606 equipped with 32GB of RAM. In our network, the communication between various machines was bridged using a 1 Gbps switch. All the servers were running in separate processes on the Xeon E5606 machine, whereas the clients were distributed among the Xeon E5443, i7, and the i5 machines. Each client invokes operation in a closed loop, i.e., a client may have at most one pending operation. In all KVS implementations, all WRITE and READ operations are served by a local database stored on disk.

We evaluate the peak throughput incurred in M-PoWerStore in WRITE and READ operations, when compared to M-ABD and Phalanx with respect to: *(i)* the file size (64 KB, 128

[4]Note that our benchmarks feature balanced reads/writes in which case balanced sign/verify DSA performance seems more appropriate for Phalanx than RSA [14].

Parameter	Default Value
Failure threshold t	1
File size	256 KB
Probability of Concurrency	1%
Workload Distribution	100% READ 100% WRITE

Table 2: Default parameters used in evaluation.

KB, 256 KB, 512 KB, and 1024 KB), and *(ii)* to the server failure threshold t (1, 2, and 3, respectively). For better evaluation purposes, we vary each variable separately and we fix the remaining parameters to a default configuration (Table 2). We also evaluate the latency incurred in M-PoWerStore with respect to the attained throughput.

We measure peak throughput as follows. We require that each writer performs back to back WRITE operations; we then increase the number of writers in the system until the aggregated throughput attained by all writers is saturated. The peak throughput is then computed as the maximum aggregated amount of data (in bytes) that can be written/read to the servers per second.

Each data point in our plots is averaged over 5 independent measurements; where appropriate, we include the corresponding 95% confidence intervals. as data objects. On the other hand, READ operations request the data pertaining to randomly-chosen keys. For completeness, we performed our evaluation *(i)* in the Local Area Network (LAN) setting comprising our aforementioned network (Section 6.2) and *(ii)* in a simulated Wide Area Network (WAN) setting (Section 6.3). Our results are presented in Figure 2.

6.2 Evaluation Results within a LAN Setting

Figure 2(a) depicts the latency incurred in M-PoWerStore when compared to M-ABD and Phalanx, with respect to the achieved throughput (measured in the number of operations per second). Our results show that, by combining metadata write-backs with erasure coding, M-PoWerStore achieves lower latencies than M-ABD and Phalanx for both READ and WRITE operations. As expected, READ latencies incurred in PoWerStore are lower than those of WRITE operations since a WRITE requires an extra communication round corresponding to the CLOCK round. Furthermore, due to PoW and the use of lightweight cryptographic primitives, the READ performance of PoWerStore considerably outperforms M-ABD and Phalanx. On the other hand, writing in M-PoWerStore compares similarly to the writing in M-ABD.

Figure 2(b) depicts the peak throughput achieved in M-PoWerStore with respect to the number of Byzantine servers t. As t increases, the gain in peak throughput achieved in M-PoWerStore's READ and WRITE increases compared to M-ABD and Phalanx. This mainly stems from the reliance on erasure coding, which ensures that the overhead of file replication among the servers is minimized when compared to M-ABD and Phalanx. In typical settings, featuring $t = 1$ and the default parameters of Table 2, the peak throughput achieved in M-PoWerStore's READ operation is almost twice as large as that in M-ABD and Phalanx.

In Figure 2(c), we measure the peak throughput achieved in M-PoWerStore with respect to the file size. Our findings clearly show that as the file size increases, the performance gain of M-PoWerStore compared to M-ABD and Phalanx becomes considerable. For example, when the file size is 1 MB, the peak throughput of READ and WRITE operations in M-PoWerStore approaches the (network-limited) bounds of 50 MB/s[5] and 45 MB/s, respectively.

6.3 Evaluation Results within a Simulated WAN Setting

We now proceed to evaluate the performance (latency, in particular) of M-PoWerStore when deployed in WAN settings. For that purpose, we rely on a 100 Mbps switch to bridge the network outlined in Section 6.1 and we rely on NetEm [37] to emulate the packet delay variance specific to WANs. More specifically, we add a Pareto distribution to our links, with a mean of 20 ms and a variance of 4 ms.

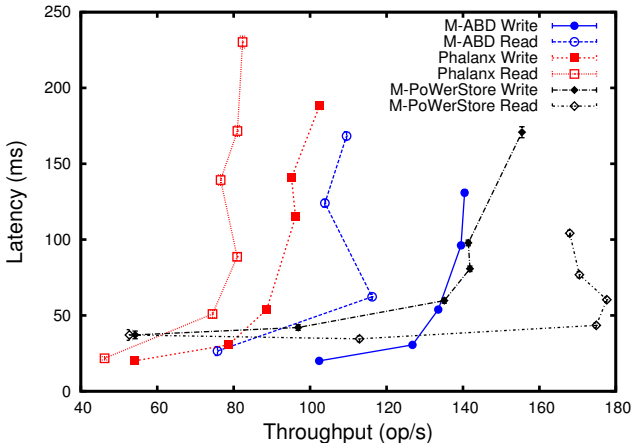
Our measurement results (Figure 2(d)) confirm our previous analysis conducted in the LAN scenario and demonstrate the superior performance of M-PoWerStore compared to M-ABD and Phalanx in realistic settings. Here, we point out that the performance of M-PoWerStore incurred in both READ and WRITE operations does not deteriorate as the number of Byzantine servers in the system increases. This is mainly due to the reliance on erasure coding. In fact, the overhead of transmitting an erasure-coded file F to the $3t+1$ servers, with a reconstruction threshold of $t+1$ is given by $\frac{3t+1}{t+1}|F|$. It is easy to see that, as t increases, this overhead asymptotically increases towards $3|F|$.

7. RELATED WORK

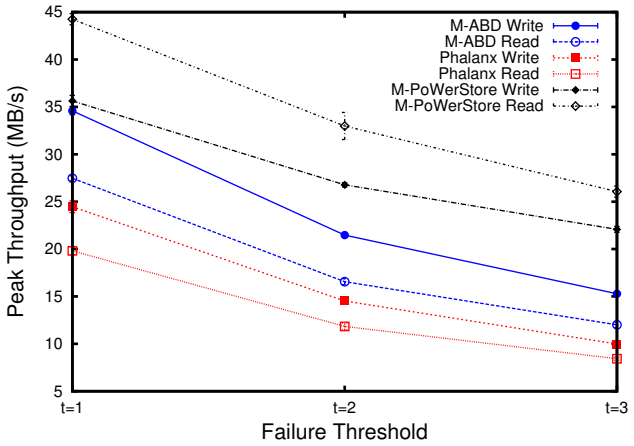
HAIL [8] is a distributed cryptographic storage system that implements a multi-server variant of Proofs of Retrievability (PoR) [9] to ensure integrity protection and availability (retrievability) of files dispersed across several storage servers. Like PoWerStore, HAIL assumes Byzantine failure model for storage servers, yet the two protocols largely cover different design space. Namely, HAIL considers a mobile adversary and a single client interacting with the storage in a synchronous fashion. In contrast, PoWerStore assumes static adversary, yet assumes a distributed client setting in which clients share data in an asynchronous fashion. Multiple clients are also supported by IRIS [42], a PoR-based distributed file system designed with enterprise users in mind that stores data in the clouds and is resilient against potentially untrustworthy service providers. However, in IRIS, all clients are pre-serialized by a logically centralized, trusted portal which acts as a fault-free gateway for communication with untrusted clouds. In contrast, PoWerStore relies on the highly available distributed PoW technique, which eliminates the need for any trusted and/or fault-free component. Notice that data confidentiality is orthogonal to all of HAIL, IRIS and PoWerStore protocols.

In the context of distributed storage asynchronously shared among multiple fault-prone clients across multiple servers without any fault-free component, a seminal crash-tolerant storage implementation (called ABD) was presented in [5]. ABD assumes a majority of correct storage servers, and achieves strong consistency by having readers write back the data they read. As shown in [17], server state modifications by readers introduced in ABD are unavoidable in *robust* storage such as ABD, where robustness is characterized by both strong consistency and high availability. However,

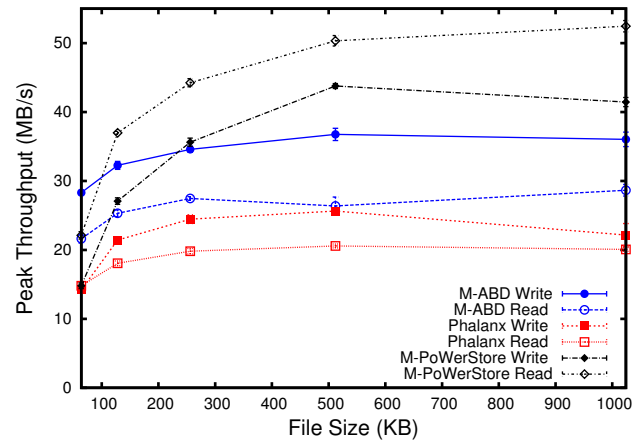
[5]Note that an effective peak throughput of 50MB/s in M-PoWerStore reflects an actual throughput of almost 820 Mbps when $t = 1$.



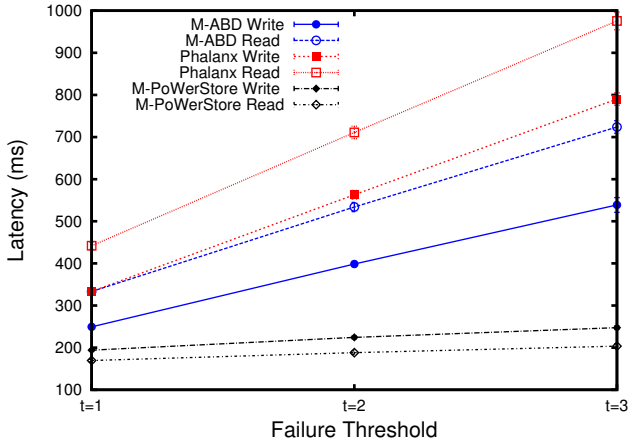
(a) Throughput vs. latency in a LAN setting.



(b) Peak throughput with respect to the failure threshold in a LAN setting.



(c) Peak throughput with respect to the file size in a LAN setting.



(d) Latency vs the failure threshold in a simulated WAN setting.

Figure 2: Evaluation Results. Data points are averaged over 5 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

robust storage implementations differ in the writing strategy employed by readers: in some protocols readers write-back data (e.g., [4,5,15,18,20,34]) whereas in others readers only write metadata to servers (e.g., [11,16,17]).

Existing robust storage protocols in which readers write only metadata, either do not tolerate Byzantine faults [11, 17], or require a total number of servers linear in number of readers to tolerate t Byzantine servers [16], and hence are prohibitively expensive. PoWerStore is hence the first robust BFT protocol that uses a bounded number of storage servers and has readers write only metadata to servers.

Clearly, most distributed BFT storage implementations have been focusing on using as few servers as possible, ideally $3t+1$, which defines optimal resilience in the asynchronous model [35]. This was first achieved by Phalanx [34], a BFT variant of ABD [5]. Phalanx uses digital signatures, i.e., *self-verifying data*, to port ABD to the Byzantine model, maintaining the latency of ABD, as well as its data write-backs. However, since digital signatures introduce considerable overhead [33, 38], protocols that feature lightweight authentication, or no data authentication at all (unauthenticated model) have been designed. Unfortunately, in the unauthenticated model, optimal resilience in BFT storage incurs considerable latency penalties: at least two rounds of communication between clients and servers for writes [3] and at least four rounds[6] for reads [15], even in the single writer case. To avoid such a considerable overhead, some robust BFT storage protocols (e.g., PASIS [19]) store unauthenticated data across $4t+1$ servers.

Clearly, there is a big gap in efficiency (and, in particular, communication latency and the number of servers) between storage protocols that use self-verifying data and those that assume no authentication. Loft [22] aims at bridging this gap and implements erasure-coded optimally resilient linearizable storage while optimizing the failure-free case. Loft uses homomorphic fingerprints and MACs; it features 3-round wait-free writes, but reads are based on data write-backs and data might be unavailable in case of heavy read/write concurrency. Similarly, our *Proofs of Writing*

[6]Under constant number of write rounds.

Protocol	Data Authentication	Data Dispersal	Read/Write Latency	no. of readers/writers	no. of messages	Byz. clients
Phalanx [34]	signatures	replication	2D/(D+d)	any/any	$O(t)$	readers
[31]	signatures	replication	2D/(D+2d)	any/any	$O(t)$	all
[15]	none	replication	(2D+2d)/(D+d)	any/1	$O(t)$	no
[10]	signatures	erasure coding	>(D+2d) / >(3D+d)	any/any	$O(t^2)$	all
Loft [22]	hash/MAC	erasure-coding	∞/D+2d	any/any	$O(t^2)$	all
PoWerStore	hash/MAC	erasure-coding	(D+d)/(D+d)	any/1	$O(t)$	readers
M-PoWerStore	hash/MAC	erasure-coding	(D+2d)/(D+2d)	any/any	$O(t)$	readers

Table 3: Comparison of properties of existing strongly consistent optimally resilient BFT storage protocols. Shown latency is worst case and distinguishes between data rounds (D) and metadata rounds (d), where typically $D >> d$.

(PoW) incorporate lightweight authentication that is, however, sufficient to achieve optimal latency and to facilitate metadata write-backs while guaranteeing optimal resilience, high-availability and strong consistency. We find PoW to be a fundamental improvement in the light of BFT storage implementations that explicitly renounce strong consistency in favor of weaker consistency notions due to the high cost of data write-backs (e.g., [7]). A summary of the key properties of existing BFT storage protocols when compared to our protocols is shown in Table 3.

A separate line of research aims at a family of so-called forking semantics (e.g., [36]), which relax atomic semantics, yet require no trusted components whatsoever. Systems guaranteeing forking semantics guarantee that after a single atomicity violation by the service, the views seen by two inconsistent clients can never again converge. PoWerStore avoids the drawbacks of fork-consistent systems (reflected in, e.g., difficulties in understanding forking semantics and exploiting them in practice [39]), by offering easily understandable, fully linearizable, (i.e., atomic) semantics.

8. CONCLUSION

In this paper, we presented PoWerStore, an efficient robust storage protocol that achieves optimal latency, measured in maximum (worst-case) number of communication rounds between a client and storage servers. We also separately presented a multi-writer variant of our protocol called M-PoWerStore. The *efficiency* of our proposals stems from combining *lightweight cryptography*, *erasure coding* and *metadata writebacks*, where readers write-back only metadata to achieve linearizability. While robust BFTs have been often criticized for being prohibitively inefficient, our findings suggest that efficient and robust BFTs can be realized in practice by relying on lightweight cryptographic primitives without compromising *worst-case* performance.

At the heart of both PoWerStore and M-PoWerStore protocols are *Proofs of Writing (PoW)*: a novel storage technique inspired by commitment schemes in the flavor of [21], that enables single-writer PoWerStore to write and read in 2 rounds which we show optimal. Similarly, by relying on PoW, multi-writer M-PoWerStore features 3-round writes/reads where the third read round is only invoked under active attacks. Finally, we demonstrated M-PoWerStore's superior performance compared to existing crash and Byzantine-tolerant atomic storage implementations.

9. ACKNOWLEDGEMENTS

This work is supported in part by the EU CLOUDSPACES (FP7-317555) and SECCRIT (FP7-312758) projects and by LOEWE TUD CASED.

10. REFERENCES

[1] Jerasure. `https://github.com/tsuraan/Jerasure`, 2008.

[2] The Neko Project. `http://ddsg.jaist.ac.jp/neko/`, 2009.

[3] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. *Distributed Computing*, 18(5):387–408, 2006.

[4] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded Wait-free Implementation of Optimally Resilient Byzantine Storage Without (Unproven) Cryptographic Assumptions. In *Proceedings of DISC*, 2007.

[5] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing Memory Robustly in Message-Passing Systems. *J. ACM*, 42:124–142, January 1995.

[6] Rida A. Bazzi and Yin Ding. Non-skipping Timestamps for Byzantine Data Storage Systems. In *Proceedings of DISC*, pages 405–419, 2004.

[7] Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *Proceedings of EuroSys*, pages 31–46, 2011.

[8] Kevin D. Bowers, Ari Juels, and Alina Oprea. Hail: a high-availability and integrity layer for cloud storage. In *CCS*, pages 187–198, 2009.

[9] Kevin D. Bowers, Ari Juels, and Alina Oprea. Proofs of retrievability: theory and implementation. In *CCSW*, pages 43–54, 2009.

[10] Christian Cachin and Stefano Tessaro. Optimal Resilience for Erasure-Coded Byzantine Distributed Storage. In *Proceedings of DSN*, pages 115–124, 2006.

[11] Brian Cho and Marcos K. Aguilera. Surviving congestion in geo-distributed storage systems. In *Proceedings of USENIX ATC*, pages 40–40, 2012.

[12] Gregory Chockler, Dahlia Malkhi, and Danny Dolev. Future directions in distributed computing. chapter A data-centric approach for scalable state machine replication, pages 159–163. 2003.

[13] Allen Clement, Edmund L. Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *Proceedings of NSDI*, pages 153–168, 2009.

[14] Wei Dai. Crypto++ 5.6.0 benchmarks. Website, 2009. Available online at `http://www.cryptopp.com/benchmarks.html`.

[15] Dan Dobre, Rachid Guerraoui, Matthias Majuntke, Neeraj Suri, and Marko Vukolić. The Complexity of

Robust Atomic Storage. In *Proceedings of PODC*, pages 59–68, 2011.

[16] Partha Dutta, Rachid Guerraoui, Ron R. Levy, and Marko Vukolić. Fast Access to Distributed Atomic Memory. *SIAM J. Comput.*, 39:3752–3783, December 2010.

[17] Rui Fan and Nancy Lynch. Efficient Replication of Large Data Objects. In *Proceedings of DISC*, pages 75–91, 2003.

[18] Chryssis Georgiou, Nicolas C. Nicolaou, and Alexander A. Shvartsman. Fault-tolerant Semifast Implementations of Atomic Read/Write Registers. *J. Parallel Distrib. Comput.*, 69(1):62–79, January 2009.

[19] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient Byzantine-Tolerant Erasure-Coded Storage. In *Proceedings of DSN*, 2004.

[20] Rachid Guerraoui and Marko Vukolić. Refined quorum systems. *Distributed Computing*, 23(1):1–42, 2010.

[21] Shai Halevi and Silvio Micali. Practical and provably-secure commitment schemes from collision-free hashing. In *Proceedings of CRYPTO*, pages 201–215, 1996.

[22] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead Byzantine fault-tolerant storage. In *Proceedings of SOSP*, pages 73–86, 2007.

[23] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[24] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[25] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant Wait-free Shared Objects. *J. ACM*, 45(3), 1998.

[26] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Proceedings of ASIACRYPT*, volume 6477, pages 177–194, 2010.

[27] Petr Kuznetsov and Rodrigo Rodrigues. Bftw^3: Why? When? Where? workshop on the theory and practice of Byzantine fault tolerance. *SIGACT News*, 40(4):82–86, 2009.

[28] Leslie Lamport. On Interprocess Communication. *Distributed Computing*, 1(2):77–101, 1986.

[29] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[30] Harry C. Li, Allen Clement, Amitanand S. Aiyer, and Lorenzo Alvisi. The Paxos Register. In *Proceedings of SRDS*, pages 114–126, 2007.

[31] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine Faulty Clients in a Quorum System. In *Proceedings of ICDCS*, 2006.

[32] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[33] Dahlia Malkhi and Michael K. Reiter. A High-Throughput Secure Reliable Multicast Protocol. *J. Comput. Secur.*, 5(2):113–127, March 1997.

[34] Dahlia Malkhi and Michael K. Reiter. Secure and Scalable Replication in Phalanx. In *Proceedings of SRDS*, pages 51–58, 1998.

[35] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine Storage. In *Proceedings of DISC*, pages 311–325, 2002.

[36] David Mazières and Dennis Shasha. Building secure file systems out of byantine storage. In *PODC*, pages 108–117, 2002.

[37] NetEm. NetEm, the Linux Foundation. Website, 2009. Available online at `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[38] Michael K. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of CCS*, pages 68–80, 1994.

[39] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: verification for untrusted cloud storage. In *CCSW*, pages 19–30, 2010.

[40] Alexander Shraer, Jean-Philippe Martin, Dahlia Malkhi, and Idit Keidar. Data-centric reconfiguration with network-attached disks. In *Proceedings of LADIS*, pages 22–26, 2010.

[41] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *Proceedings of NSDI*, pages 189–204, 2008.

[42] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *ACSAC*, pages 229–238, 2012.

[43] Sue-Hwey Wu, Scott A. Smolka, and Eugene W. Stark. Composition and behaviors of probabilistic i/o automata. In *Proceedings of CONCUR*, pages 513–528, 1994.

APPENDIX

A. OPTIMALITY OF PoWerStore

In this section, we prove that PoWerStore features optimal latency, by showing that writing in two rounds is necessary and we refer the reader to [16] for the necessity of reading in two rounds. We start by giving some informal definitions.

A distributed algorithm A is a set of automata [32], where automaton A_p is assigned to process p. Computation proceeds in steps of A and a *run* is an infinite sequence of steps of A. A *partial run* is a finite prefix of some run. We say that a (partial) run r *extends* some partial run pr if pr is a *prefix* of r. We say that an implementation is *selfish*, if clients write-back metadata to achieve linearizability (instead of the full value) [17]. Furthermore, we say that an operation is *fast* if it completes in a single round.

THEOREM A.1. *There is no* fast WRITE *implementation I of a multi-reader* selfish *robust storage that makes use of less than $4t + 1$ servers.*

Preliminaries. We prove Theorem A.1 by contradiction assuming at most $4t$ servers. An illustration of the proof is given in Figure 3. We partition the set of servers into four distinct subsets (we call *blocks*), denoted by T_1, T_2, T_3 each of size exactly t, and T_4 of size at least 1 and at most t. Without loss of generality we assume that each block contains at least one server. We say that an operation *op skips* a block T_i, $(1 \leq i \leq 4)$ when all messages by *op* to

T_i are delayed indefinitely (due to asynchrony) and all other blocks T_j receive all messages by *op* and reply.

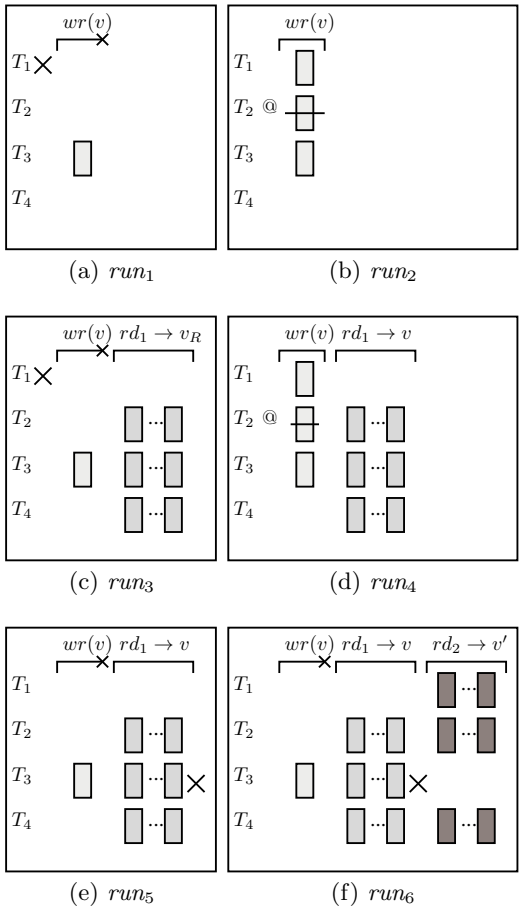


Figure 3: Sketch of the runs used in the proof of Theorem A.1.

Proof: We construct a series of runs of a linearizable implementation I towards a partial run that violates linearizability, i.e., that features two consecutive READ operations by distinct readers that return different values.

- Let run_1 be the partial run in which all servers are correct except T_1 which crashed at the beginning of run_1. Let wr be the operation invoked by the writer w to write a value $v \neq \perp$ in the storage. The WRITE wr is the only operation invoked in run_1 and w crashes after writing v to T_3. Hence, wr skips blocks T_1, T_2 and T_4.
- Let run'_1 be the partial run in which all servers are correct except T_4, which crashed at the beginning of run'_1. In run'_1, w is correct and wr completes by writing v to all blocks except T_4, which it skips.
- Let run_2 be the partial run similar to run'_1, in which all servers except T_2 are correct, but due to asynchrony, all messages from w to T_4 are delayed. Like in run'_1, wr completes by writing v to all servers except T_4, which it skips. To see why, note that wr cannot distinguish run_2 from run'_1. After wr completes, T_2 fails Byzantine by reverting its memory to the initial state.
- Let run_3 extend run_1 by appending a complete READ rd_1 invoked by r_1. By our assumption, I is wait-free. As such, rd_1 completes by skipping T_1 (because T_1 crashed) and returns (after a finite number of rounds) a value v_R.
- Let run_4 extend run_2 by appending rd_1. In run_4, all servers except T_2 are correct, but due to asynchrony all messages from r_1 to T_1 are delayed indefinitely. Moreover, since T_2 reverted its memory to the initial state, v is held only by T_3. Note that r_1 cannot distinguish run_4 from run_3 in which T_1 has crashed. As such, rd_1 completes by skipping T_1 and returns v_R. By linearizability, $v_R = v$.
- Let run_5 be similar to run_3 in which all servers except T_3 are correct but, due to asynchrony, all messages from r_1 to T_1 are delayed. Note that r_1 cannot distinguish run_5 from run_3. As such, rd_1 returns v_R in run_5, and by run_4, $v_R = v$. After rd_1 completes, T_3 fails by crashing.
- Let run_6 extend run_5 by appending a READ rd_2 invoked by r_2 that completes by returning v'. Note that in run_5, *(i)* T_3 is the only server to which v was written, *(ii)* rd_1 did not write-back v (to any other server) before returning v, and *(iii)* T_3 crashed before rd_2 is invoked. As such, rd_2 does not find v in any server and hence $v' \neq v$, violating linearizability. □

It is important to note that Theorem A.1 allows for self-verifying data and assumes clients that may fail only by crashing. Furthermore, the impossibility extends to crash-tolerant storage using less than $3t + 1$ servers when deleting the Byzantine block T_2 in the above proof.