Proofs of Writing for Robust Storage

Dan Dobre, Ghassan O. Karame, *Member, IEEE,* Wenting Li, Matthias Majuntke, Neeraj Suri, Marko Vukolić

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 state of the art robust storage protocols, including protocols that tolerate only crash faults.

Index Terms—Byzantine fault Tolerant Storage; Proofs of Writing; Commitment schemes.

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 [47] or high cost in performance and deployment compared to *crash-tolerant* protocols [31]. 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., [27]).
- Such a *robust* protocol should be *efficient*. For example, most BFT protocols resort to *data writebacks*, 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 [6] 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 [6]: *(i) high availability*, or *wait-freedom* [28], where read/write operations invoked by correct *clients* always complete, and *(ii) strong consistency*, or *linearizability* [29] of read/write operations.

At the heart of PoWerStore is a 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. Nevertheless, this "metadata writeback", performed by the writers, is powerful enough to spare readers of writing back the entire data, allowing them to also 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 writebacks* 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 [40]. Moreover, PoWerStore achieves these desirable properties with *optimal latency*: namely, we show that no robust single-writer protocol, including crash-tolerant ones, that uses a bounded number of servers and avoids data writebacks can achieve better latency

- *D. Dobre is with the Security and Networking, NEC Laboratories Europe, Heidelberg, Baden-Württemberg 69115, Germany. E-mail: dan.dobre@neclab.eu*
- *G. O. Karame, W. Li is with NEC Laboratories Europe, Heidelberg, Baden-Württemberg 69115, Germany. E-mail: {ghassan.karame,wenting.li}@neclab.eu*
- *M. Majuntke is with the Capgemini Deutschland, Berlin 10785, Germany. E-mail:matthias.majuntke@capgemini.com*
- *N. Suri is with the TU Darmstadt, Darmstadt 64289, Germany. E-mail: suri@cs.tu-darmstadt.de*
- *M. Vukolić is with IBM Research - Zurich, Rüschlikon 8803, Switzerland. E-mail: mvu@zurich.ibm.com*

1. As proved in [22], in any robust storage readers must "write", i.e., modify the state of storage servers.

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 prevent writers from exhausting the timestamp domain — a threat that was first outlined in [9].

This paper extends our previous work in [20]. More specifically, we present additional formal technical details about PoW, and we introduce formal proofs for the correctness of PoWerStore and M-PoWerStore. We additionally evaluate PoWerStore and demonstrate its superiority when compared to state of the art robust storage protocols such as DepSky [10]. Our results show that in typical settings, the peak throughput achieved by S-PoWerStore improves over DepSky by 27% for READ operations and by 17% for WRITE operations. On the other hand, our evaluation results show that M-PoWerStoreREAD is almost 2 times faster than the existing *crash-tolerant* robust atomic storage implementations.

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 [17], [46] 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 [50] where a distributed algorithm is a set of such processes. Processes that follow the algorithm are called *correct*. Similar to [3], [4], [21], [39], we assume that a setting where any number of *correct* writers may fail by crashing.

We assume that an unbounded number of readers and up to t servers are *Byzantine* [34] (or exhibit *arbitrary* [30]) faults. 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., [14], [24], [27], [35]. As a consequence, with Byzantine writers and asynchronous message schedule, the adversary can make a correct reader return arbitrary data at will. Here, appropriate access control mechanisms could prevent untrusted writers from having appropriate credentials to modify data.

In summary, we assume the following threat model:

- We assume a strong adversary that can coordinate up to t Byzantine servers and an unbounded number of readers.
- We 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 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.

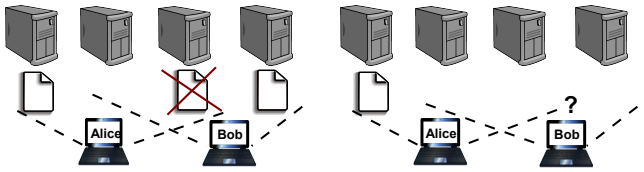
2.2 Atomic Storage

We focus on a read/write storage abstraction [33] 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. We model executions of an algorithm using *histories*, which are finite sequences of invocation and response events. In any history H, we say that a complete operation op_2 *follows* op_1 (resp., op_1 precedes op_2, denoted by $op_1 <_H op_2$) if the invocation of op_2 follows the response of $op1$ in that execution. A sequential history is a history in which an invocation of op is immediately followed by corresponding response. A sequential history S if read/write storage is *legal* if every READ op_{rd} in S returns a value written by the last WRITE in S that precedes op_{rd}, or $\perp$ if there is no such write. 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* [29] (or *atomicity* [33]) and *wait-freedom* [28]. Wait-freedom is originally defined [28] on concurrent data objects as a guarantee that any process can complete any operation in a finite number of steps. In our message passing model, we consider the following analogous definition of *wait-free liveness*: 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. More formally [29], a history H is linearizable if: (1) it can be extended (by appending zero or more response events) to some history H' which is equivalent to some legal sequential history S, and (2) relation $<_H$ is a subset of relation $<_S$.

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.



(a) Complete Write (b) Incomplete Write

Fig. 1. Complete vs. Incomplete writes ($t = 1$, $S = 4$). Alice and Bob are readers — dashed lines depict server quorums accessed by Alice and Bob. In a "complete" write of Figure (a), the value was previously written (by writer Charlie) into servers 1, 3 and 4 (skipping server 2). If server 3 becomes Byzantine and "forgets" the value, it becomes impossible for reader Alice to distinguish such a complete write from an "incomplete" write of Figure (b). Alice must "writeback" some information to be sure a subsequent reader Bob has enough information about a value Alice reads.

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. "mod. exp." refers to modular exponentiations.

	Construction costs	**Verification costs**
Hash-based PoW	1 hash	1 hash
Polynomial-based PoW	$O(\|t^3\|)$ mod. exp.	$O(\|t^2\|)$ mod. exp.
RSA signatures	$O(\|M\|)$ mod. exp.	$O(\|M\|)$ mod. exp.

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., [23].

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 reads some value already written (say by writer Charlie), and reader Bob reads but only *subsequently*, i.e., after Alice completes her read. Clearly, Bob should return the same value as Alice. However, reader Alice cannot tell apart a complete write (Fig. 1(a)) where one Byzantine server (server 3) 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 [6] 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 digital signature schemes required to preventing malicious readers from exploiting such write-backs to jeopardize the storage by over-writing 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 perform erasure coding on the 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.

At the heart of PoWerStore is a storage technique we call Proofs of Writing (PoW) which enables to achieve this differentiation more efficiently. PoW are inspired by commitment schemes [26]; 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. This process is sketched in Figure 2. 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 effectively constructed while incurring little cost when compared to digital signatures (Table 1).

Similar to commitment schemes [26], PoW consist of two main algorithms:

- $c \leftarrow \mathsf{commit}(s)$: Given a security parameter n, commit takes as input a string $s \in \{0,1\}^n$ and outputs a commitment c.
- $\{0,1\} \leftarrow \mathsf{verify}(s',c)$: On input a commitment c, and a string s', verify outputs 1 if $c \leftarrow \mathsf{commit}(s')$ and 0, otherwise.

PoW satisfies the following security properties:

- **Completeness**: Honest clients always accept a correct PoW verification. This is by construction; in fact $\mathsf{verify}(s,c)$ outputs 1 if and only if $c \leftarrow \mathsf{commit}(s)$.
- **Secrecy**: For any probabilistic polynomial time adversary $\mathscr{A}$, $\forall s \neq s' \in \{0,1\}^n$, $\mathscr{A}$ has a negligible advantage denoted by $\varepsilon(n)$, where ε is a negligible function of n, in distinguishing the probability ensembles $(\mathsf{commit}(s)|c)$ and $(\mathsf{commit}(s')|c)$.
- **Non-ambiguity**: $\forall s \neq s' \in \{0,1\}^n, \mathsf{commit}(s) \neq \mathsf{commit}(s')$.

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 (sampled uniformly at random from $\{0,1\}^N$, where N is a security parameter) 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 quorum. 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, i.e., including at least one confirmation from a correct server.

Since the writer keeps the nonce secret until the start of the second round, it is straightforward to show that our PoW construct based on cryptographic hashes satisfies the completeness, secrecy, and non-ambiguity properties. Notably, 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. Our construct is based on the seminal work by [44] that instantiates a threshold system based on polynomial evaluation. Namely, assume a polynomial $P(\cdot)$ of degree t with coefficients $\{\alpha_t, \ldots, \alpha_0\}$ chosen at random from $\mathscr{Z}_q$, where q is a public parameter. That is, $P(x) = \sum_{j=0}^{j=t} (\alpha_j x^j)$. $P(\cdot)$ can be used to instantiate a threshold system in such a way that [44]:

- The knowledge of any $t+1$ points of $P(.)$ makes $P(.)$ easily computable;
- The knowledge of any t or fewer points leaves $P(.)$ completely undetermined (i.e., all of its possible values are equally likely) [44].

We instantiate such a threshold system as follows: at the start of every WRITE operation, the writer constructs $P(x) = \sum_{j=0}^{j=t} (\alpha_j x^j)$. 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 *secure and 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 (the polynomial is completely undetermined in this case). 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. By relying on randomly chosen x_i, and the fact that correct servers never divulge their share, our construction prevents an adversary from fabricating $\hat{P(\cdot)}$. We point that, unlike our prior solution based on cryptographic hash functions, this PoW construction provides information-theoretic guarantees [44]. 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).

3.4 Optimality of PoWerStore

In this section, we prove that PoWerStore features optimal latency, by showing that writing in two rounds is necessary. In previous work [21], it was shown that, regardless of the "length" of a write a read needs to read in more than one round or else the number of required servers is proportional to the number of readers. Hence, in order to maintain a constant number of servers, regardless the number of readers, a read needs to read in at least two rounds.

We start by giving some informal definitions.

A distributed algorithm A is a set of automata [36], 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) [22]. Furthermore, we say that an operation is *fast* if it completes in a single round.

Theorem 3.1. 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 3.1 by contradiction assuming at most $4t$ servers. 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.

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, $v_R = v$.

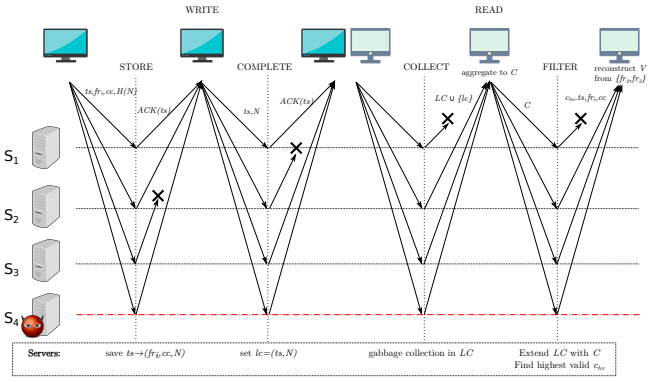


Fig. 2. Example depicting the case where a candidate collected in C is valid, then at least $t+1$ correct servers (S_2 and S_3) has saved $Hist[ts] = (fr_i, cc, \overline{N})$ and will return the data in FILTER step.

- 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 3.1 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.

4 PoWerStore

In this section, we provide a detailed description of the PoWerStore protocol and we analyze its correctness. We also 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.

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 sampled uniformly at random, 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.

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

```
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
```

Algorithm 1: Algorithm of the writer in PoWerStore.

In STORE, the writer erasure codes a value (in order to offload the storage servers and to reduce network traffic) by encoding V into S fragments fr_i ($1 \leq i \leq S$), such that V can be recovered from any subset of $t+1$ fragments. We achieve this by combining the use of parity fragments with Reed-Solomon coding. Here, the writer divides V to $t+1$ fragments and add $S-t-1$ parity fragments. Since the reader will only decode fragments from correct servers, the decoding can tolerate up to $S-t-1$ *erasures* and reconstruct the data from any $t+1$ correct 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 (see Algorithm 2). 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) included 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, i.e., tuples not greater than the highest *valid* timestamp among all candidates from LC using the GC procedure (see line 28, Alg. 2, where predicate *valid* is defined in line 41). 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

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: $Hist[\ldots]$: vector of $(fr, cc, \overline{N})$ indexed by ts, with all entries initialized to $(\text{NULL}, \text{NULL}, \text{NULL})$

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

Algorithm 2: Algorithm of server s_i in PoWerStore.

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. Note that although the client receives $S-t$ erasure-coded fragments, only $t+1$ fragments[4] are guaranteed to be available and correct. Therefore, the reconstruction of V requires an erasure code with dimension $t+1$. 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.

Definition 4.2 (Timestamps of operations). A READ operation *rd* by a correct reader has timestamp ts iff the reader in *rd* selected c in line 53 such that $c.ts = ts$. A WRITE operation *wr* has timestamp ts iff the writer increments its timestamp to ts in line 4.

Lemma 4.3 (Validity). Let *rd* be a completed READ by a correct reader. If *rd* returns value $V \neq \perp$ then V was written.

Proof: We show that if V is the value decoded in line 74, then V was indeed written. To show this, we argue that the fragments used to decode V were written. Note that prior to decoding V from a set of fragments, the reader establishes the correctness of each fragment as follows. First, in line 72, the reader chooses a cross-checksum that was received from $t+1$ servers. Since one of these servers is correct, the chosen cross-checksum was indeed written. Secondly, the reader checks in line 73 that each of the $t+1$ fragments used to decode V hashes to the corresponding entry in the cross-checksum. By the collision-resistance of H, all fragments that pass this check were indeed written. Therefore, if V is the value decoded from these fragments, we conclude that V was written.

Lemma 4.4 (Proofs of Writing). If c is a valid candidate, then there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(fr_i, cc, H(c.N))$.

Proof: If c is valid, then by Definition 4.1, $\mathsf{valid}(c)$ is true at some correct server s_j. Hence, $H(c.N) = Hist[c.ts].N_H$ holds at s_j. By the pre-image 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 after the STORE phase in *wr* completed. Hence, there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(fr_i, cc, H(c.N))$.

Lemma 4.5 (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 4.4 a there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(*, *, H(c.N))$. 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 phase are lower than $c.ts$, and so c is never excluded from C.

Lemma 4.6 (READ/WRITE Atomicity). Let *rd* be a completed READ by a correct reader. If *rd* follows some complete WRITE(V), then *rd* does not return a value older than V.

Proof: If ts is the timestamp of WRITE(V), it is sufficient to show that the timestamp of *rd* is not lower than ts. To prove this, we show that $\exists c' \in C$ such that *(i)* $c'.ts \geq ts$ and *(ii)* c' is never excluded from C.

By the time WRITE(V) completes, $t+1$ correct servers hold in lc a candidate whose timestamp is ts or greater. According to

4. The dimension is $t+1$ given $S = 3t+1$ in our system model (cf. Section 2). If $S > 3t+1$, then the quorum of correct servers that have saved the data after a complete WRITE is $S-2t$.

42: **Definitions:**
43: tsr: *num*, initially 0
44: Q, R: set of *pid*, initially $\emptyset$
45: C: set of (ts, N), initially $\emptyset$
46: $W[1 \ldots S]$: 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: $\mathsf{invalid}(c) \triangleq |\{i \in R : W[i].ts < c.ts\}| \geq S - t$

Algorithm 3: Algorithm of client r in PoWerStore.

lines 26, 30 of Algorithm 2, a correct server never changes lc to a candidate with a lower timestamp. Hence, when rd is invoked, $t+1$ correct servers hold candidates with timestamp ts or greater in lc. Hence, during the COLLECT phase in rd, some candidate received from a correct server with timestamp ts or greater is inserted in C. Such a candidate is necessarily valid because either the server received it directly from the writer, or the server checked its integrity in line 29. Let c' be the valid candidate with the highest timestamp in C. Then by Lemma 4.5, c' is never excluded from C. By line 53, no candidate c such that $c.ts < c'.ts$ is selected. Since $c'.ts \geq ts$, no candidate with a timestamp lower than ts is selected in rd.

***Lemma 4.7* (READ *atomicity*).** Let rd and rd' be two completed read operations by correct readers. If rd' follows rd that returns V, then rd' does not return a value older than V.

Proof: If c is the candidate selected in rd, it is sufficient to show that the timestamp of rd' is not lower than $c.ts$. We argue that C contains a candidate c' such that *(i)* $c'.ts \geq c.ts$ and *(ii)* c' is never excluded from C.

By the time rd completes, $t+1$ correct servers hold c in LC. As c was selected in rd in line 53, some correct server asserted that c is valid in line 29. According to Algorithm 2, if a correct server excludes c from LC in line 31, then the server changed lc to a valid candidate with timestamp $c.ts$ or greater in line 30. Consequently, $t+1$ correct servers hold in $LC \cup \{lc\}$ a valid candidate with timestamp $c.ts$ or greater. As such, during COLLECT in rd', a valid candidate c' such that $c'.ts \geq c.ts$ is included in C, and by Lemma 4.5, c' is never excluded from C. By line 53, no candidate with a timestamp lower than $c'.ts$ is selected. Since $c'.ts \geq c.ts$, no candidate with a timestamp lower than $c.ts$ is selected in rd'.

Theorem 4.8 (Linearizability/Atomicity). Algorithms 1, 2 and 3 are linearizable.

Proof: To prove linearizability of history H, consider an equivalent sequential history S in which all WRITES are (totally) ordered by their timestamps selected in line 4, Fig. 1, and all reads are (partially) ordered by the timestamp $c.ts$ (line 53, Alg. 3), or assigned timestamp 0 if they return $\perp$. If two reads rd_1 and $rd2$ are assigned the same timestamp, then we require $rd_1 <_S rd_2$ if $rd_1 <_H rd_2$, otherwise the order of rd_1 and rd_2 in S is irrelevant.

First, note that by Lemma 4.3, S is a legal sequential history.

It is left to show that $<_S$ is a subset of $<_H$. This clearly holds for two WRITES, as WRITE operations which are sequential in PoWerStore. We now show that a complete READ operation rd by a correct reader, which follows another read/write operation op (i.e., $rd <_H op$) cannot return an older value (i.e., $rd <_S op$). We distinguish two cases: 1) rd follows a write operation, and 2) rd follows another read operation. The first case follows from Lemma 4.6, whereas the second case follows from Lemma 4.7.

We now proceed to proving wait-free liveness.

Theorem 4.9 (Wait-free liveness). Algorithms 1, 2 and 3 satisfy wait-free liveness.

Proof: We show that no operation invoked by a correct client ever blocks. The wait-free liveness argument of the WRITE is straightforward; in every phase, the writer awaits acks from the least number $S - t$ of correct servers. The same argument holds for the COLLECT phase of the READ. Hence, in the remainder of the proof, we show that no READ blocks in the FILTER phase. By contradiction, consider a READ rd by reader r that blocks during the FILTER phase after receiving FILTER_ACK messages from all correct servers. We distinguish two cases: (Case 1) C includes a valid candidate and (Case 2) C includes no valid candidate.

- Case 1: Let c be the highest valid candidate included in C. We show that $\mathsf{highcand}(c) \wedge \mathsf{safe}(c)$ holds. Since c is valid, by Lemma 4.4, there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(fr_i, cc, H(c.N))$. Thus, during the FILTER phase, $\mathsf{valid}(c)$ holds at every server in Q. As no valid candidate in C has a higher timestamp than c, *(i)* all servers $s_i \in Q$ (at least $t+1$) responded with timestamp $c.ts$, corresponding erasure coded fragment fr_i, cross-checksum cc in line 39 and *(ii)* all correct servers (at least $S - t$) responded with timestamps at most $c.ts$. By *(i)*, c is

safe. By *(ii)*, every $c' \in C$ such that $c'.ts > c.ts$ became invalid and was excluded from C, implying that c is highcand.

- Case 2: Here, we show that $C = \emptyset$. As none of the candidates in C is valid, during the FILTER phase, the integrity check in line 29 failed for every candidate in C at all correct servers. Hence, at least $S-t$ servers responded with timestamp ts_0. Since ts_0 is lower than any candidate timestamp, all candidates were classified as invalid and were excluded from C.

Theorem 4.10 (Latency). Algorithms 1, 2 and 3 feature a latency of *two* communication rounds for the WRITE and *two* for the READ.

Proof: By Algorithm 1, the WRITE completes after two phases, STORE and COMPLETE, each taking one communication round. By Algorithm 3, the READ completes after two phases, COLLECT and FILTER, each incurring one communication round.

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 [9]. 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 [18]), which prevents adversary from swamping the point-to-point channels between correct processes, 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, which prevents the adversary from exhausting the timestamp domain, i.e., from skipping timestamps [9].

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 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 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 safe(c) holds, or until C is empty, after which FILTER returns. The predicate 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 $t+1$

80: **Definitions:**
81: Q: set of pid, (process id) initially $\emptyset$
82: ts: structure $(num, pid, MAC_{\{k_W\}}(num||pid))$, initially $ts_0 \triangleq (0,0,\text{NULL})$

83: **operation** WRITE(V)
84: $Q \leftarrow \emptyset$
85: $ts \leftarrow$ CLOCK()
86: $N \leftarrow \{0,1\}^\lambda$
87: $\overline{N} \leftarrow H(N)$
88: $vec \leftarrow [MAC_{\{k_i\}}(ts||\overline{N})_{1 \le i \le S}]$
89: STORE$(ts, V, \overline{N}, vec)$
90: COMPLETE(ts, N, vec)
91: return OK

92: **procedure** CLOCK()
93: send CLOCK$\langle ts \rangle$ to all servers
94: **wait until** $|Q| \ge S - t$
95: $ts.num \leftarrow ts.num + 1$
96: $ts \leftarrow (ts.num, w, MAC_{\{k_W\}}(ts.num||w))$
97: return ts

98: **upon** receiving CLOCK_ACK$\langle ts, ts_i \rangle$ from server s_i
99: $Q \leftarrow Q \cup \{i\}$
100: **if** $ts_i > ts \wedge$ verify(ts_i, k_W) **then** $ts \leftarrow ts_i$

101: **procedure** STORE$(ts, V, \overline{N}, vec)$
102: $\{fr_1, \ldots, fr_S\} \leftarrow$ encode$(V, t+1, S)$
103: $cc \leftarrow [H(fr_1), \ldots, H(fr_S)]$
104: **foreach** server s_i send STORE$\langle ts, fr_i, cc, \overline{N}, vec \rangle$ to s_i
105: **wait for** STORE_ACK$\langle ts \rangle$ from $S - t$ servers

106: **procedure** COMPLETE(ts, N, vec)
107: send COMPLETE$\langle ts, N, vec \rangle$ to all servers
108: **wait for** COMPLETE_ACK$\langle ts \rangle$ from $S - t$ servers

Algorithm 4: Algorithm of writer w in M-PoWerStore.

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 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. Without REPAIR subprotocol, the atomicity across READ operations could be violated (see Lemma 5.8 in Sec. 5.4).

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.

Definition 5.1 (Valid candidate). A candidate c is *valid* iff valid(c) is true at some correct server.

Definition 5.2 (Timestamps of operations). A READ operation rd by a non-malicious reader has timestamp ts iff the reader in rd selected c in line 143 such that $c.ts = ts$. A WRITE operation wr has timestamp ts iff the CLOCK procedure in wr returned ts in line 85.

Lemma 5.3 (Validity). Let rd be a completed READ by a correct reader. If rd returns value $V \neq \perp$ then V was written.

109: **Definitions:**
110: lc: structure (ts, N, vec), initially $c_0 \triangleq (ts_0, \text{NULL}, \text{NULL})$
111: $Hist[\ldots]$: vector of $(fr, cc, \overline{N}, vec)$ indexed by ts, with all entries initialized to $(\text{NULL}, \text{NULL}, \text{NULL}, \text{NULL})$

112: **upon** receiving CLOCK$\langle ts \rangle$ from writer w
113: send CLOCK_ACK$\langle ts, lc.ts \rangle$ to writer w

114: **upon** receiving STORE$\langle ts, fr, cc, \overline{N}, vec \rangle$ from writer w
115: $Hist[ts] \leftarrow (fr, cc, \overline{N}, vec)$
116: send STORE_ACK$\langle ts \rangle$ to writer w

117: **upon** receiving COMPLETE$\langle ts, N, vec \rangle$ from writer w
118: **if** $ts > lc.ts$ **then** $lc \leftarrow (ts, N, vec)$
119: send COMPLETE_ACK$\langle ts \rangle$ to writer w

120: **upon** receiving COLLECT$\langle tsr \rangle$ from client r
121: send COLLECT_ACK$\langle tsr, lc \rangle$ to client r

122: **upon** receiving FILTER$\langle tsr, C \rangle$ from client r
123: $c_{hv} \leftarrow \max(\{c \in C : \text{valid}(c)\} \cup \{c_0\})$
124: **if** $c_{hv}.ts > lc.ts$ **then** $lc \leftarrow c_{hv}$ //write-back
125: $(fr, cc, vec) \leftarrow \pi_{fr,cc,vec}(Hist[c_{hv}.ts])$
126: send FILTER_ACK$\langle tsr, c_{hv}.ts, fr, cc, vec \rangle$ to client r

127: **upon** receiving REPAIR$\langle tsr, c \rangle$ from client r
128: **if** $c.ts > lc.ts \wedge$ valid(c) **then** $lc \leftarrow c$ //write-back
129: send REPAIR_ACK$\langle tsr \rangle$ to client r

130: **Predicates:**
131: valid$(c) \triangleq (H(c.N) = Hist[c.ts].\overline{N}) \vee$
verify$(c.vec[i], c.ts, H(c.N), k_i)$

Algorithm 5: Algorithm of server s_i in M-PoWerStore.

Proof: Note that the reader protocol returns $V \neq \perp$ only using a RESTORE procedure of the single-writer version of PoWerStore. (line 144). Hence proof of this lemma (Validity) is identical to the proof of Lemma 4.3.

Lemma 5.4 (*WRITE *atomicity). Let op be a completed operation by a correct client and let wr be a completed WRITE such that op precedes wr. If ts_{op} and ts_{wr} are the timestamps of op and wr respectively, then $ts_{wr} > ts_{op}$.

Proof: By the time op completes, $t+1$ correct servers hold in lc a candidate whose timestamp is ts_{op} or greater. According to lines 118, 124, 128 of Algorithm 5, a correct server never updates lc with a candidate that has a lower timestamp. Hence, the writer in wr obtains from the CLOCK procedure a timestamp that is greater or equal to ts_{op} from some correct server s_i. Let c be the candidate held in lc by server s_i, and let $c.ts$ be the timestamp reported to the writer. We now argue that $c.ts$ is not fabricated. To see why, note that prior to overwriting lc with c in line 124 (resp. 128), server s_i checks that c is valid in line 123 (resp. 128). The valid predicate as defined in line 131 subsumes an integrity check for $c.ts$. Hence, $c.ts$ passes the integrity check in line 100; according to the WRITE algorithm, $ts_{wr} \ge (c.ts.num + 1, *, *) > c.ts \ge ts_{op}$.

Lemma 5.5 (Proofs of Writing). If c is a valid candidate, then there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(fr_i, cc, H(c.N), vec)$.

Proof: If c is valid, then by Definition 5.1, valid(c) is true at some correct server s_j. Hence, either $H(c.N) = Hist[c.ts].N_H$ or

```
132: Definitions:
133:    tsr: num, initially 0
134:    Q,R: set of pid, initially ∅
135:    C: set of (ts,N,vec), initially ∅
136:    W[1...S]: vector of (ts,fr,cc,vec), initially []
---------------------------------------------------------
137: operation READ()
138:    C,Q,R ← ∅
139:    tsr ← tsr+1
140:    C ← COLLECT(tsr)
141:    C ← FILTER(tsr,C)
142:    if C ≠ ∅ then
143:       c ← c' ∈ C : highcand(c') ∧ safe(c')
144:       V ← RESTORE(c.ts)
145:       REPAIR(c)
146:    else V ← ⊥
147:    return V
   ...
148: upon receiving COLLECT_ACK⟨tsr,c_i⟩ from server s_i
149:    Q ← Q∪{i}
150:    if c_i.ts > ts_0 then C ← C∪{c_i}
   ...
151: upon receiving FILTER_ACK⟨tsr,ts,fr,cc,vec⟩ from server s_i
152:    R ← R∪{i}; W[i] ← (ts,fr,cc,vec)
153:    C ← C \ {c ∈ C : invalid(c)}
   ...
154: procedure REPAIR(c)
155:    vec ← vec' s.t. ∃R' ⊆ R : |R'| ≥ t+1 ⋀
        (∀i ∈ R' : W[i].ts = c.ts ∧ W[i].vec = vec')
156:    if c.vec ≠ vec then
157:       c.vec ← vec                                  //repair
158:       send REPAIR⟨tsr,c⟩ to all servers
159:       wait for REPAIR_ACK⟨tsr⟩ from S−t servers
160: Predicates:
161:    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)
   ...
```

Algorithm 6: Algorithm of client r in M-PoWerStore.

verify$(c.vec[j], c.ts, H(c.N), k_j)$ must hold at s_j. By the pre-image resistance of H, no computationally bounded adversary can acquire $c.N$ from the sole knowledge of $H(c.N)$. Hence, $c.N$ stems from some writer in a WRITE operation wr with timestamp $c.ts$. By Algorithm 4, line 90, the value of $c.N$ is revealed after the STORE round in wr completed. Hence, there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(fr_i, cc, H(c.N), vec)$.

Lemma 5.6 (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 5.5 a there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(*, *, H(c.N), vec)$. Hence, validByHist(c) is true at every server in Q. Thus, no server in Q replies with a timestamp $ts < c.ts$ in line 126. 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.

Lemma 5.7 (READ/WRITE Atomicity). Let rd be a completed READ by a correct reader. If rd follows some complete WRITE(V), then rd does not return a value older than V.

Proof: If ts is the timestamp of WRITE(V), it is sufficient to show that the timestamp of rd is not lower than ts. To prove this, we show that $\exists c' \in C$ such that *(i)* $c'.ts \geq ts$ and *(ii)* c' is never excluded from C.

By the time WRITE(V) completes, $t+1$ correct servers hold in lc a candidate whose timestamp is ts or greater. According to lines 118, 124, 128 of Algorithm 5, a correct server never changes lc to a candidate with a lower timestamp. Hence, when rd is invoked, $t+1$ correct servers hold candidates with timestamp ts or greater in lc. Hence, during COLLECT in rd, some candidate received from a correct server with timestamp ts or greater is inserted in C. Such a candidate is necessarily valid by the integrity checks in lines 123, 128. Let c' be the valid candidate with the highest timestamp in C. Then by Lemma 5.6, c' is never excluded from C. By line 143, no candidate c such that $c.ts < c'.ts$ is selected. Since $c'.ts \geq ts$, no candidate with a timestamp lower than ts is selected in rd.

Lemma 5.8. *(*READ *atomicity).* Let rd and rd' be two completed read operations by correct readers. If rd' follows rd that returns V, then rd' does not return a value older than V.

Proof: If c is the candidate selected in rd, it is sufficient to show that the timestamp of rd' is not lower than $c.ts$. We argue that C contains a candidate c' such that *(i)* $c'.ts \geq c.ts$ and *(ii)* c' is never excluded from C.

As c is selected in rd in line 143 only if safe(c) holds, some correct server verified the integrity of $c.ts$ and $c.N$. In addition, in REPAIR, the reader in rd checks the integrity of $c.vec$. We distinguish two cases:

- Case 1: If $c.vec$ passes the integrity check in line 156, then the integrity of c has been fully established. Hence, by the time rd completes, $t+1$ correct servers validated c in line 123 and changed lc to c or to a higher valid candidate.
- Case 2: If vector $c.vec$ fails the integrity check in line 156, then in REPAIR, c is repaired in line 157 and subsequently written back to $t+1$ correct servers. Hence, by the time rd completes, $t+1$ correct servers validated c in line 128 and changed lc to c or to a higher valid candidate.

Consequently, in the COLLECT round in rd' a valid candidate c' such that $c'.ts \geq c$ is included in C, and by Lemma 5.6, c' is never excluded from C. By line 143, no candidate with a timestamp lower than c' is selected. Since $c'.ts \geq c.ts$, no candidate with a timestamp lower than $c.ts$ is selected in rd'.

Theorem 5.9 (Linearizability/Atomicity). Algorithms 4, 5 and 6 are linearizable.

Proof: To prove linearizability of history H, consider an equivalent sequential history S in which all WRITES are (totally) ordered by their timestamps selected in line 85, Fig. 4, and all reads are (partially) ordered by the timestamp $c.ts$ (line 140, Alg. 6), or assigned timestamp 0 if they return $\perp$. If two reads rd_1 and $rd2$ are assigned the same timestamp, then we require $rd_1 <_S rd_2$ if $rd_1 < Hrd_2$, otherwise the order of rd_1 and rd_2 in S is irrelevant.

First, note that by Lemma 5.3, S is a legal sequential history.

It is left to show that $<_S$ is a subset of $<_H$. For two WRITES, this follows from Lemma 5.4. We now show that a complete READ operation rd by a correct reader, which follows another

read/write operation op (i.e., $rd <_H op$) cannot return an older value (i.e., $rd <_S op$). We distinguish two cases: 1) rd follows a write operation, and 2) rd follows another read operation. THe first case follows from Lemma 5.7, whereas the second case follows from Lemma 5.8.

We now proceed to proving wait-free liveness.

Theorem 5.10 (Wait-free liveness). Algorithms 4, 5,and 6 satisfy wait-free liveness.

Proof: We show that no operation invoked by a correct client ever blocks. The wait-free liveness argument of the WRITE is straightforward; in every round, the writer awaits acks from the least number $S-t$ of correct servers. The same argument holds for the COLLECT and the REPAIR rounds of the READ. Hence, in the remainder of the proof, we show that no READ blocks in the FILTER round. By contradiction, consider a READ rd by reader r that blocks during the FILTER round after receiving FILTER_ACK messages from all correct servers. We distinguish two cases: (Case 1) C includes a valid candidate and (Case 2) C includes no valid candidate.

- Case 1: Let c be the highest valid candidate included in C. We show that highcand(c) $\wedge$ safe(c) holds. Since c is valid, by Lemma 5.5, there exists a set Q of $t+1$ correct servers such that each server $s_i \in Q$ changed $Hist[c.ts]$ to $(fr_i, cc, H(c.N), vec)$. Thus, during the FILTER round, validByHist(c) holds at every server in Q. As no valid candidate in C has a higher timestamp than c, *(i)* all servers $s_i \in Q$ (at least $t+1$) responded with timestamp $c.ts$, corresponding erasure coded fragment fr_i, cross-checksum cc and repair vector vec in line 126 and *(ii)* all correct servers (at least $S-t$) responded with timestamps at most $c.ts$. By *(i)*, c is safe. By *(ii)*, every $c' \in C$ such that $c'.ts > c.ts$ became invalid and was excluded from C, implying that c is highcand.
- Case 2: Here, we show that $C = \emptyset$. As none of the candidates in C is valid, during FILTER, the integrity check in line 123 failed for every candidate in C at all correct servers. Hence, at least $S-t$ servers responded with timestamp ts_0. Since ts_0 is lower than any candidate timestamp, all candidates were classified as invalid and were excluded from C.

Theorem 5.11 (Non-skipping Timestamps). Algorithms 4, 5 and 6 implement non-skipping timestamps.

Proof: By construction, a fabricated timestamp would fail the check in line 100. Hence, no fabricated timestamp is ever used in a WRITE. The Lemma then directly follows from the algorithm of WRITE.

Theorem 5.12 (Latency). Algorithms 4, 5 and 6 feature a latency of *three* communication rounds for the WRITE and *two* for the READ in the absence of attacks. In the worst case, the READ latency is *three* communication rounds.

Proof: By Algorithm 4, the WRITE completes after three rounds, CLOCK, STORE and COMPLETE, each taking one communication round. In the absence of attacks, by Algorithm 6, the READ completes after two rounds, COLLECT and FILTER, each taking one communication round. Under BigMac [18] attacks the READ may go to the REPAIR round, incurring one additional communication round.

TABLE 2
Default parameters used in evaluation.

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

6 Implementation & Evaluation

In this section, we describe an implementation modeling a file storage protocol based on M-PoWerStore. We then evaluate the performance of our implementation and we compare it to DepSky [10].

6.1 Implementation Setup

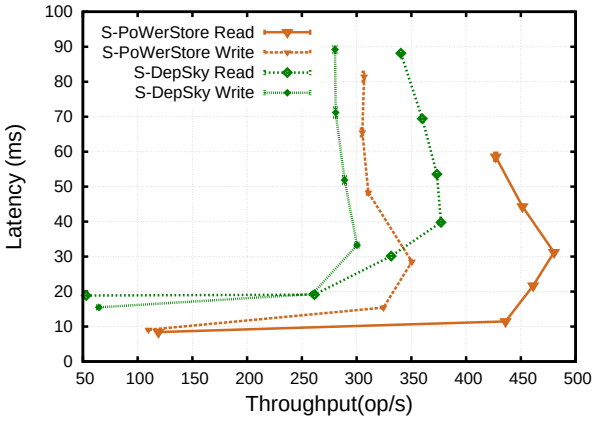
Our Java implementation is based on the Grpc library [2] and on the Backblaze library for constructing the dispersal codes [8] . To evaluate the performance of our S-PoWerStore, we additionally implemented a file storage abstraction based on DepSky. Recall that DepSky is a single-writer multi-cloud key value store that provides atomic semantics.

To evaluate the performance of M-PoWerStore we implemented a multi-writer variant of ABD [7] and Phalanx [38]. Recall that ABD is a crash-only atomic key-value store ($n = 2t+1$) that writes back the retrieved data in READ operation; Phalanx additionally provides Byzantine fault tolerance guarantees through the use of digital signatures. In a nutshell, Phalanx is a port of ABD to the Byzantine fault model, with all writes authenticated (digitally signed), and the number of servers adjusted to the Byzantine fault model ($2t+1$ in ABD vs. $3t+1$ in Phalanx).

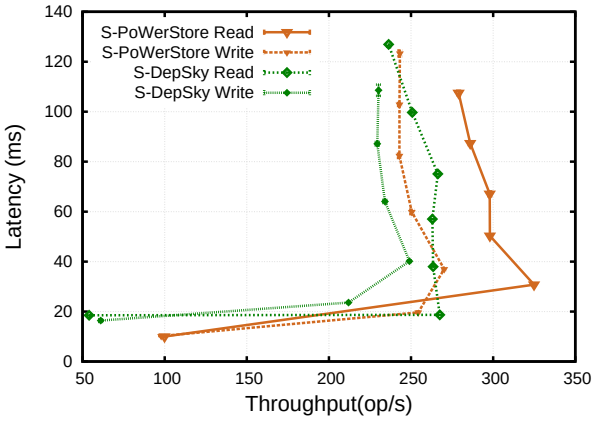
In our implementation, we adapted (n,k) Reed-Solomon coding scheme (with JErasure library [1]) to encode the uploaded data and tolerate up to $n-k$ erasures. The length of code n equals to the number of servers; and the dimension $k = t+1$, where t is the number of Byzantine faulty servers and $n = 3t+1$. We relied on SHA256 for hashing purposes, HMAC-SHA256 to implement MACs, and the signatures are generated with 2048-bit RSA for DepSky and M-Phalanx. For simplicity, we abstract away the effect of data origin authentication since it is typically handled as part of the access control layer in all implementations.

We deployed our implementations on a private network consisting of 1 machine with Xeon E5-2640 with 24 vCore and 32 GB RAM to host all servers, and 5 machines with Xeon E3-1240 V5 8 vCore and 32 GiB RAM as client nodes. In our network, the communication between various machines was bridged using a 1 Gbps switch. We assume a well-formed behavior from the clients: each client invokes a new operation only after the previous operation that it invoked is complete, i.e., a client may have at most one pending operation. In our implementations, WRITE and READ operations are served by a local database stored on disk.

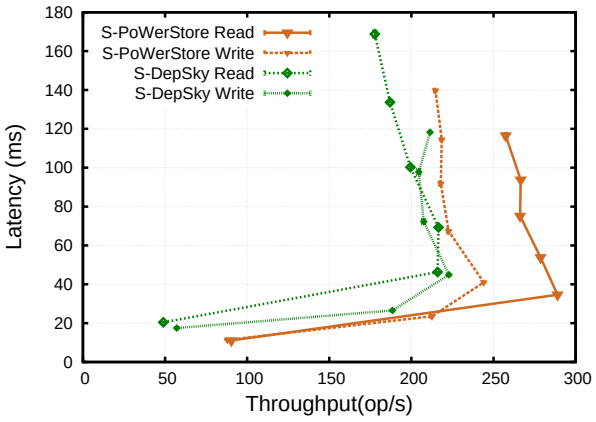
We evaluate the peak throughput incurred in M-PoWerStore in WRITE and READ operations with respect to the server failure threshold t. We measure peak throughput as follows. We require that each writer performs back to back WRITE operations; we then increase the number of writers in the system until the 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. Unless otherwise specified, we rely on the default parameters listed in



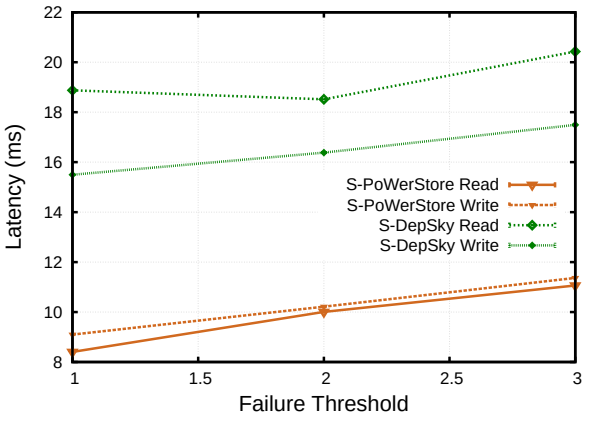
(a) Single-writer throughput vs. latency in a LAN setting for $t = 1$.



(b) Single-writer throughput vs. latency in a LAN setting for $t = 2$.



(c) Single-writer throughput vs. latency in a LAN setting for $t = 3$.



(d) Single-writer latency vs the failure threshold in a simulated LAN setting.

Fig. 3. Evaluation of S-PoWerStore vs. DepSky in a LAN setting. Data points are averaged over 10 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

Table 2 in our evaluation.

In our single-writer protocols, each writer writes data associated with uniquely generated keys; to emulate multi-writer protocols, all the writers and readers access the data pertaining to the same key. For multi-writer protocols, we also allow contention over the operations; namely, we allow a subset of writers/readers access files with the same key. For completeness, we performed our evaluation *(i)* in the Local Area Network (LAN) setting comprising our aforementioned network and *(ii)* in a simulated Wide Area Network (WAN) setting. We analyse the observed performance in Section 6.2 for single-writer storage protocol and Section 6.3 for multi-writer. Our results are presented in Figure 3 and Figure 5.

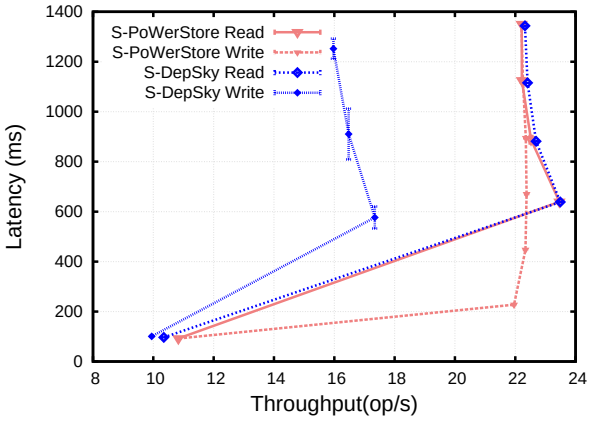
6.2 Evaluation Results of Single-Writer Protocols

Figures 3(a), 3(b), and 3(c) depict the latency incurred in S-PoWerStore when compared to DepSky in the LAN setting, with respect to the achieved throughput (measured in the number of operations per second) with different threshold value. As shown in Figure 3(a), both S-PoWerStore READ and WRITE achieve lower latency and higher peak throughput when compared to DepSky. When $t = 1$, S-PoWerStore READ (WRITE) requires 8 ms (9 ms) of latency and is able to reach a peak throughput of 480 op/s (350 op/s) with file size 256KB, equivalent to 120 MB/s (87 MB/s). Meanwhile, DepSky READ (WRITE) requires 18 ms (15 ms) for latency and can only reach a peak throughput of 377 op/s (300 op/s). Similar observations also hold when $t = 2$ (cf. Figure 3(b)) and $t = 3$ (cf. Figure 3(c)).

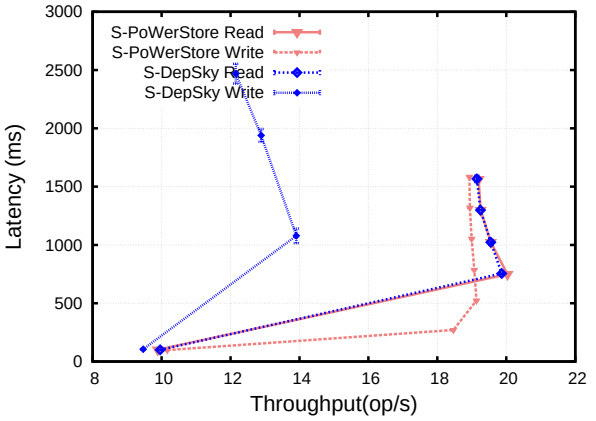
Notice that S-PoWerStore outperforms DepSky due to the use of lightweight cryptographic primitives. Namely, DepSky relies on digital signatures in the metadata retrieval and write-back round while S-PoWerStore only uses lightweight hash-based commitment schemes. We also observe that for both protocols, the latency incurred in READ and WRITE operations is rather similar, as both operations requires two rounds of communication: one round for data transmission and the other for metadata transmission. However, we see that READ always achieves higher throughput (37% more when $t = 1$, 22% when $t = 2$ and 33% when $t = 3$ in S-PoWerStore) when compared WRITE operation. This is because the READ operation just need to receive $n - t$ erasure-coded data partitions, while WRITE always sends n partitions to the servers.

Figure 3(d) shows the achieved latency of a single operation with respect to the threshold t. As t increases, the witnessed end-to-end latency for all operations increases slightly due to the hash computation on more data fragments that are transmitted to the servers. For example, READ and WRITE operations in S-PoWerStore require around 11 ms when $t = 3$; this latency increases to 20 ms and 17 ms, respectively, in the case of DepSky.

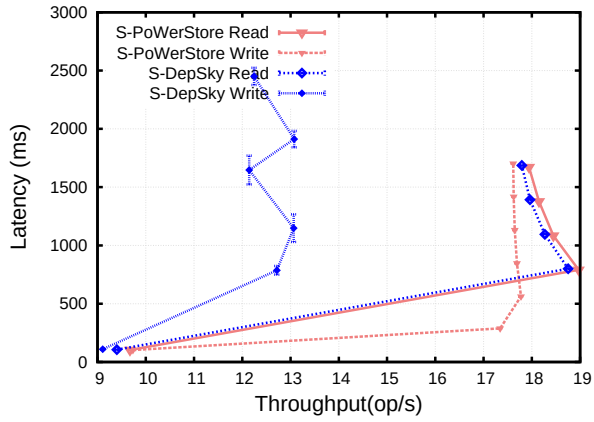
Similarly, we measured the latency w.r.t. throughput of both protocols in the WAN setting. For that purpose, we rely on NetEm [42] to emulate the packet delay variance specific to WANs. More specifically, we restrict the bandwidth from 1 Gbps to 100 Mbps and add a Pareto distribution to our links, with a mean



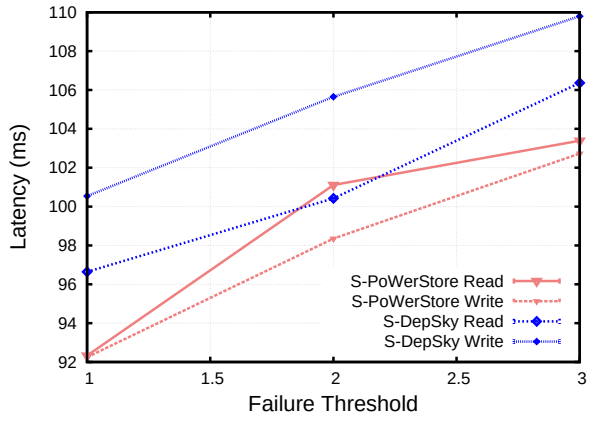
(a) Single-writer throughput vs. latency in a WAN setting for $t = 1$.



(b) Single-writer throughput vs. latency in a WAN setting for $t = 2$.



(c) Single-writer throughput vs. latency in a WAN setting for $t = 3$.



(d) Single-writer latency vs the failure threshold in a simulated WAN setting.

Fig. 4. Evaluation of S-PoWerStore vs. DepSky in a simulated WAN setting. Data points are averaged over 10 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

of 20 ms and a variance of 4 ms.

Our results are depicted in Figures 4(a), 4(b), 4(c), and 4(d). We observe that the operations now take around 100 ms in latency due to the restricted bandwidth and the additional link latency in all rounds. As a result, the peak throughput of READ (WRITE) is reduced to 23 op/s (22 op/s) for S-PoWerStore and 23 op/s (17 op/s) for DepSky when $t = 1$. Our results suggest therefore that, in the WAN setting, the difference between the S-PoWerStore and DepSky becomes less significant, since the network latency dominates the metadata computation.

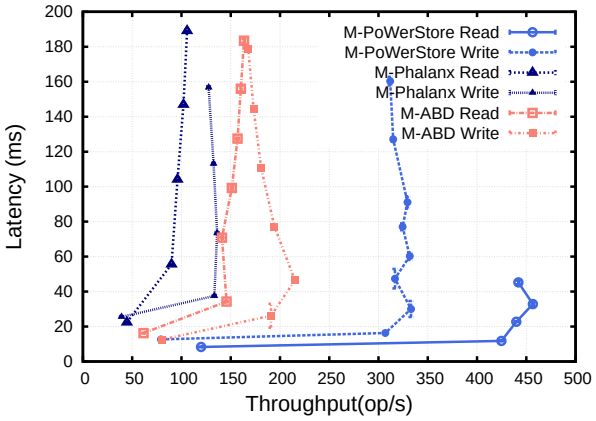
6.3 Evaluation Results of Multiple-Writer Protocols

We now proceed to evaluate the performance of M-PoWerStore comparing to M-ABD and M-Phalanx. Our results are depicted in Figures 5 and 6. We can see that M-PoWerStore achieves the lowest latency and highest peak throughput among all three protocols, followed by M-ABD and M-Phalanx, respectively. For example, when $t = 1$ in the LAN setting, M-PoWerStore performs similarly to S-PoWerStore and exhibits a latency of 8 ms and peak throughput of 457 op/s for READ and latency 12 ms and peak throughput 333 op/s for WRITE. We contrast this to the 164 op/s peak throughput achieved by M-ABD in READ (and 215 op/s in WRITE operations), and to the 106 op/s peak throughput achieved by M-Phalanx in READ (and 136 op/s in WRITE) (cf. Figure 5(a)).

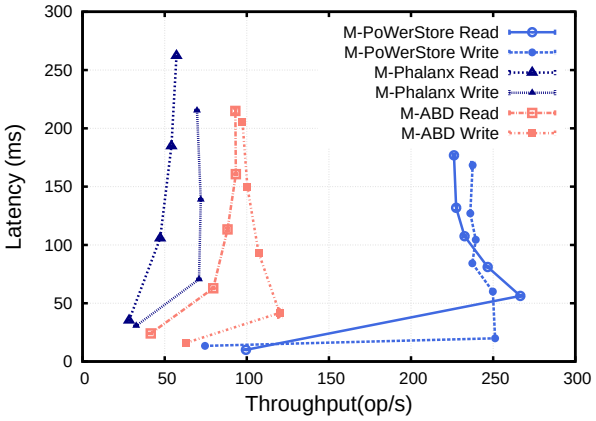
Contrary to M-PoWerStore, M-ABD and M-Phalanx READ operation is slower than WRITE operation since both M-ABD and M-Phalanx need to write back the data after the read round. M-Phalanx achieves even lower peak throughput as it further uses digital signature in addition to data write-back.

In the WAN setting, the performance of M-PoWerStore is even more pronounced when compared to M-ABD and M-Phalanx. For example, when $t = 1$, M-ABD and M-Phalanx exhibit a READ latency of 255 ms and 287 ms, respectively, which is more than two times slower than that of M-PoWerStore. Namely, READ in M-PoWerStore completes in only 90 ms (cf. Figure 5(a)). Recall that in order to synchronize multiple writers, the WRITE operation of the multi-writer protocols needs an additional round when compared to the READ operation, which results in higher WRITE latency of almost 118 ms when $t = 1$ (cf. Figure 5(a)). Finally, we notice that the failure threshold has less impact on the performance of M-PoWerStore in the WAN setting since the network latency dominates the additional local computation cost. This is why the latency of ABD READ is the double of that of ABD WRITE in the WAN setting; this effect is less visible in the LAN setting where network speed is no longer the major contributor in the overall latency.

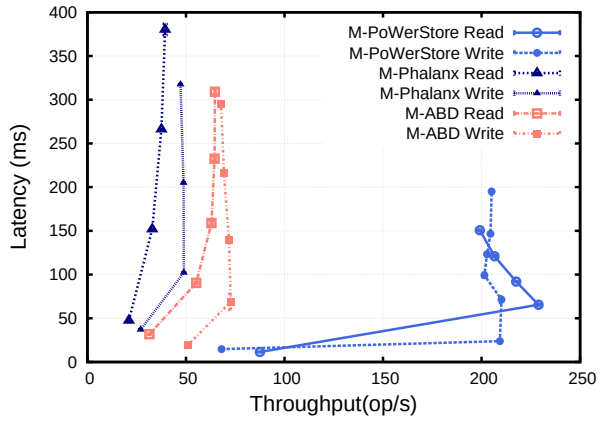
In Figure 7, we profile resource usage both on the client side and server side. Here, we measure the amount of time incurred for client-side computation, server-side computation, and communication latency (both in LAN and WAN settings), respectively for each operation. Our results show that M-ABD incurs negligible computation compared to communication latency,



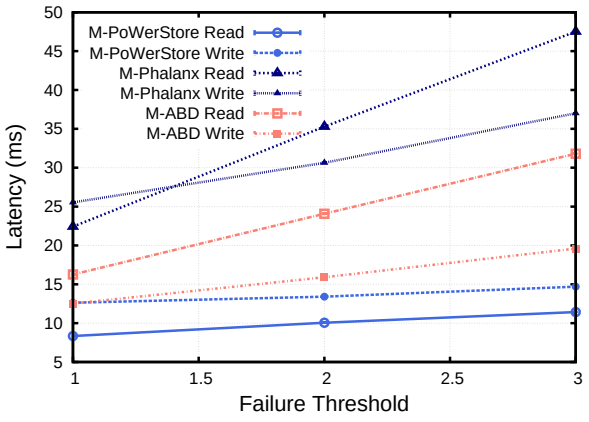
(a) Multi-writer throughput vs. latency in a LAN setting for $t = 1$.



(b) Multi-writer throughput vs. latency in a LAN setting for $t = 2$.



(c) Multi-writer throughput vs. latency in a LAN setting for $t = 3$.



(d) Multi-writer latency vs the failure threshold in a simulated LAN setting.

Fig. 5. Evaluation of M-PoWerStore in the LAN setting. Data points are averaged over 10 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

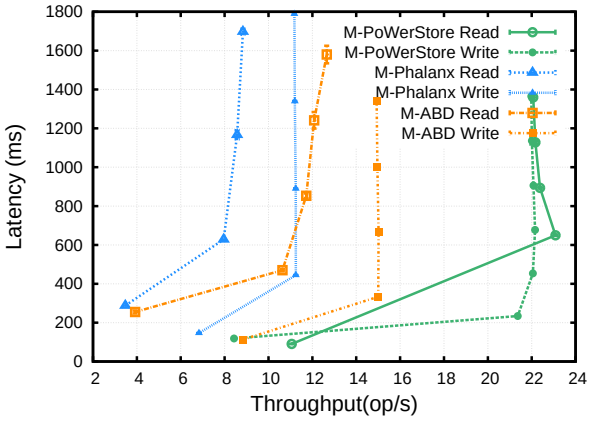
while M-Phalanx WRITE consumes the most computation resources both on the client side and the server side. We also see that for M-PoWerStore the computation overhead is mostly borne on the client side while the server-side computation is negligible. This is the case since in M-PoWerStore the client needs to select a safe candidate among the candidate list and perform erasure-coding. Our results also show that the servers in M-PoWerStore require the least communication and computational overhead; this confirms our claim that M-PoWerStore scales better than M-ABD and M-Phalanx. Namely, we see that the network latency dominates the operation performance in the WAN setting. Since M-PoWerStore uses PoW metadata write-back combining with erasure-coding optimization, it incurs the least network latency when compared to the other protocols, especially in READ operations.

In Figure 7(d), we profile the memory usage on the servers. We see that M-PoWerStore requires more memory than the other schemes since the servers need to maintain the history of the PoW. However, since PoW only consists of metadata (i.e., nonce and the hash of nonce), we argue that this overhead does not significantly impact the performance when the history is properly pruned with checkpoints. In fact, even when $t = 3$, the total amount of memory used by 13 servers only accounts for 1.8% of the total available memory (i.e., 32 GB in our case).

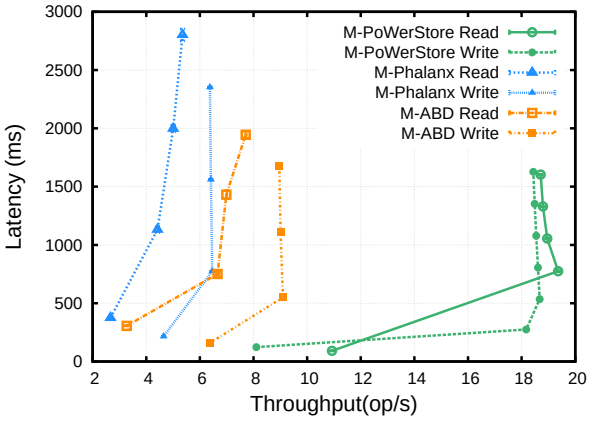
Notice that our previous evaluation in Figure 3-6 only focuses on single operation with full contention. In what follows, we vary the client requests to take into account different write/read workloads and contention levels.

We first consider different ratios of write/read operations in the system and measure the latency incurred in these operations. More specifically, we consider scenarios when the WRITE operations compose of 20% and 40% respectively of the total amount of client requests. Figures 8(a)-8(c) show that the latency of both READ and WRITE operations in M-ABD and M-Phalanx decreases when there are more writers in the system. This is due to the fact that the WRITE operation is faster than READ in these schemes; as a result, there are more system resources available to process the requests when there are more writers. On the other hand, M-PoWerStore WRITE latency increases only slightly while the READ latency almost remains intact as the number of writers increase in the system. Moreover, our results show that when the number of servers increases, the performance of WRITE operations in M-ABD and M-Phalanx are more affected by the READ since in these two protocols, READ consumes more resources than WRITE, which is even further exacerbated when the fault tolerance threshold increases.

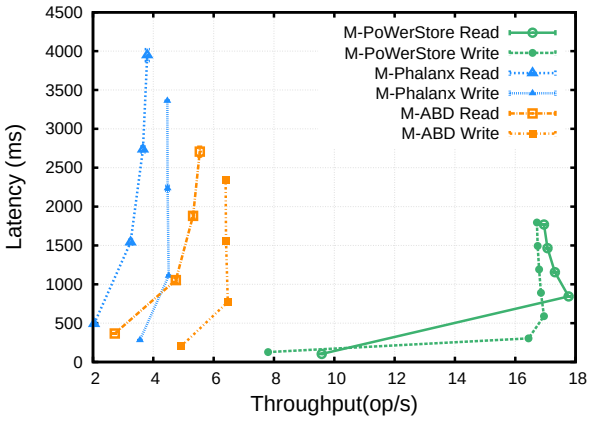
We also consider the case when a subset of the client requests involves concurrent data access. In our subsequent experiments, we allow 0%, 10%, 20% and 100% respectively of the clients to access the same data (i.e., indexed by the same key) and measure the differences in the peak throughput. The results are shown in Figures 8(d)-8(f). Our results show that the contention level does not influence the performance of all three protocols due to memory



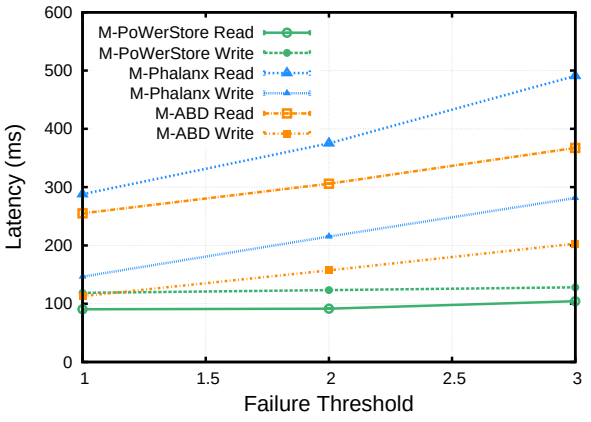
(a) Multi-writer throughput vs. latency in a WAN setting for $t = 1$.



(b) Multi-writer throughput vs. latency in a WAN setting for $t = 2$.



(c) Multi-writer throughput vs. latency in a WAN setting for $t = 3$.



(d) Multi-writer latency vs the failure threshold in a simulated WAN setting.

Fig. 6. Evaluation of M-PoWerStore in a simulated WAN setting. Data points are averaged over 10 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

caching.

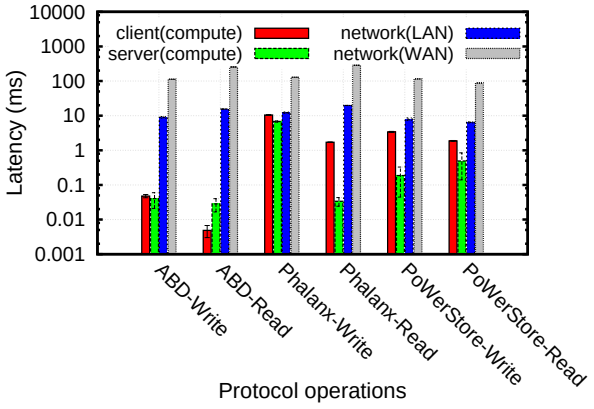
7 Related Work

HAIL [11] is a distributed cryptographic storage system that implements a multi-server variant of Proofs of Retrievability (PoR) [12] 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 [48], 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.

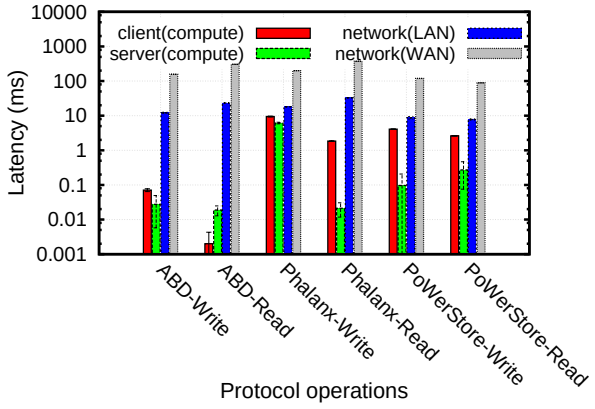
In the context of distributed storage asynchronously shared among multiple fault-prone clients across multiple servers without any fault-free component, a seminal crash-tolerant storage implementation (called ABD) was presented in [6]. ABD assumes a majority of correct storage servers, and achieves strong consistency by having readers write back the data they read. As shown in [22], 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, robust storage implementations differ in the writing strategy employed by readers: in some protocols readers write-back data (e.g., [4], [6], [19], [23], [25], [39]) whereas in others readers only write metadata to servers (e.g., [15], [16], [21], [22]).

Previous robust storage protocols in which readers write only metadata, either do not tolerate Byzantine faults [15], [16], [22], or require a total number of servers linear in number of readers to tolerate t Byzantine servers [21], 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.

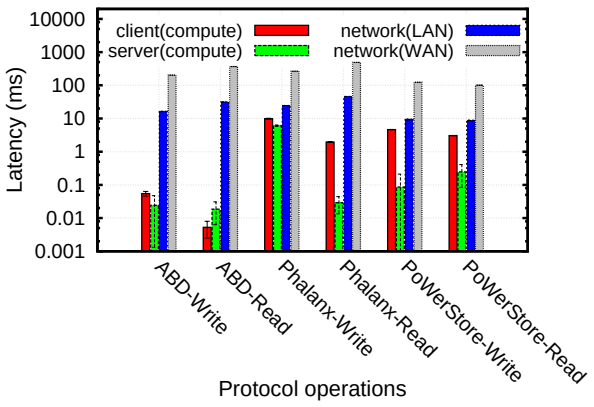
Clearly, most distributed BFT storage implementations have been focusing on using as few servers as possible, ideally $3t + 1$, which defines optimal resilience in the asynchronous model [40]. This was first achieved by Phalanx [39], a BFT variant of ABD [6]. Phalanx uses digital signatures, i.e., *self-verifying 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 [37], [43], protocols that feature lightweight authentication, or no data authentication at all (unauthenticated



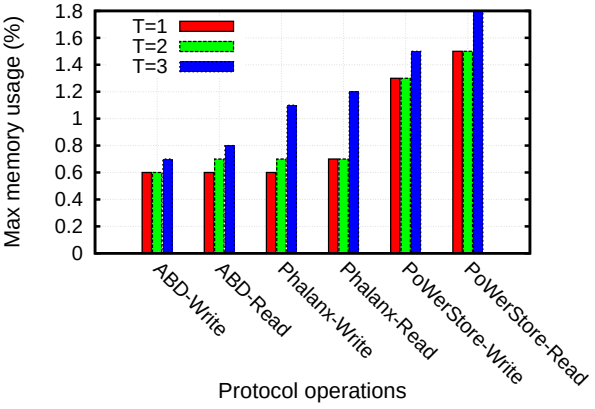
(a) Multi-writer operation latency profiling in a LAN/WAN setting for $t = 1$.



(b) Multi-writer operation latency profiling in a LAN/WAN setting for $t = 2$.



(c) Multi-writer operation latency profiling in a LAN/WAN setting for $t = 3$.



(d) Multi-writer server memory usage profiling.

Fig. 7. Evaluation of M-PoWerStore in the LAN and WAN setting. Data points are averaged over 10 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

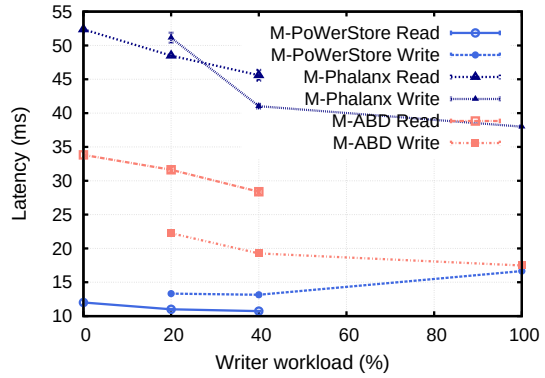
model) have been designed. Unfortunately, in the unauthenticated model, optimal resilience in BFT storage incurs considerable latency penalties: at least two rounds of communication between clients and servers for writes [3] and at least four rounds[5] for reads [19], even in the single writer case. To avoid such a considerable overhead, some robust BFT storage protocols (e.g., PASIS [24]) 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 [27] 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* (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., [10]).

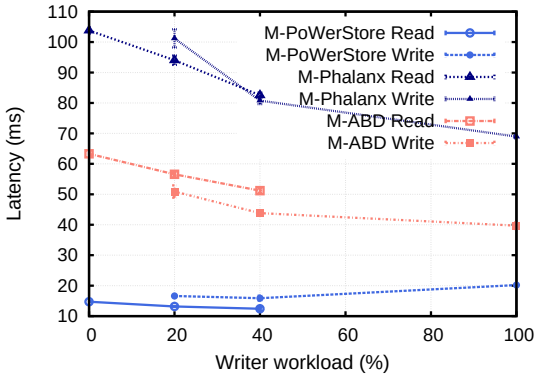
After the publication of our preliminary work on PoWerStore [20], several systems and storage protocols that target atomic semantics and optimize the storage of metadata were proposed. Hybris [49] is a multi-writer atomic key value store that stores metadata in an atomic wait-free coordination service (Apache ZooKeeper) and data across cloud-based key-value stores. Unlike PoWerStore, Hybris provides only finite-write termination which is a weaker notion that wait-freedom guaranteed by PoWerStore. Furthermore, Hybris works in a different network model, requiring partial synchrony, whereas PoWerStore is an asynchronous protocol. MDStore [13] and AWE [5] remove the partial synchrony requirement from Hybris-like 2-layer architectures separating data and metadata, and guarantee wait-freedom, but their latency is not optimal as latency of PoWerStore. Furthermore, [15] proposes CASGC which is basically a version of our M-PoWerStore restricted to the crash failure model. The other notable difference between CASGC and M-PoWerStore is that CASGC does not incure infinite storage costs, whereas M-PoWerStore stores the entire history of the shared object indexed by writer timestamps. It is interesting future work to attempt to combine both approaches and obtain a version of M-PoWerStore that does not store the entire history of the shared object.

A separate line of research aims at a family of so-called forking semantics (e.g., [41]), which relax atomic semantics, yet require no trusted components whatsoever. Systems guaranteeing forking semantics guarantee that after a single atomicity violation by the

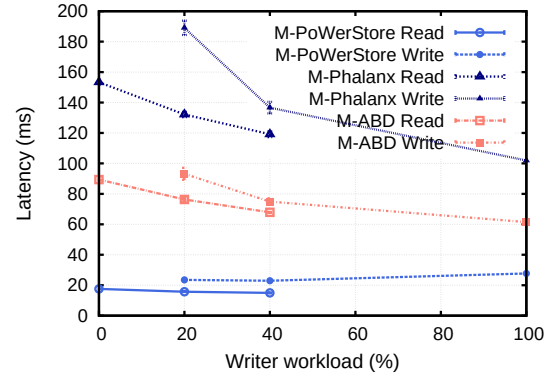
5. Under constant number of write rounds.



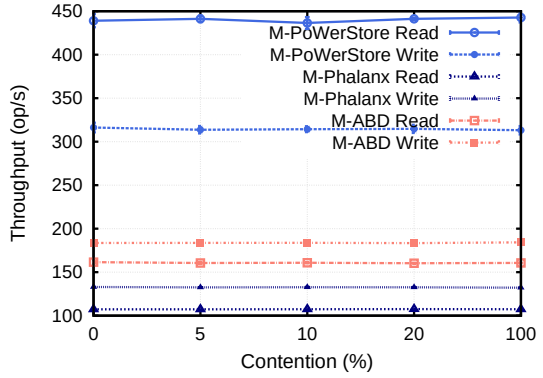
(a) Multi-writer latency vs. writer workload in a LAN setting for $t = 1$.



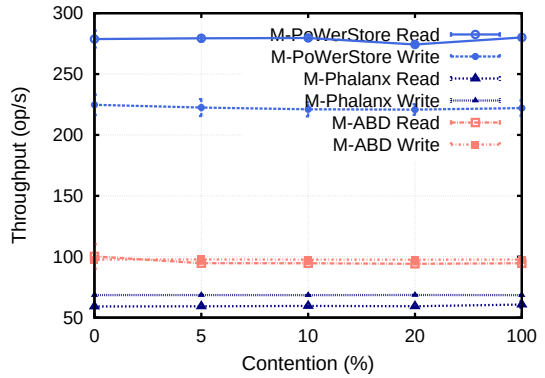
(b) Multi-writer latency vs. writer workload in a LAN setting for $t = 2$.



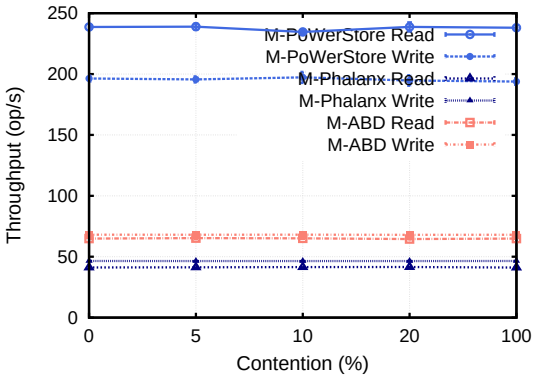
(c) Multi-writer latency vs. writer workload in a LAN setting for $t = 3$.



(d) Multi-writer peak throughput vs. data contention in a LAN setting for $t = 1$.



(e) Multi-writer peak throughput vs. data contention in a LAN setting for $t = 2$.



(f) Multi-writer peak throughput vs. data contention in a LAN setting for $t = 3$.

Fig. 8. Evaluation of M-PoWerStore in the LAN setting. Data points are averaged over 10 independent runs; where appropriate, we include the corresponding 95% confidence intervals.

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 [45]), 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 in the single writer setting, 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. At the heart of both PoWerStore and M-PoWerStore protocols are *Proofs of Writing (PoW)*: a storage technique inspired by commitment schemes in the flavor of [26], that enables single-writer PoWerStore to write and read in 2 rounds which we show optimal. 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.

Our work has promising practical potential, for being deployed in

the context of multiple clouds [32], which views clouds from multiple providers to be seen as independently failing but potentially untrusted. As future work, we highlight as very interesting the problem of combining crash-tolerant storage smilax to M-PoWerStore (CASGC [15]) which does not incur unbounded storage costs, with Byzantine fault-tolerance of M-PoWerStore. This would pave the path to practical deployments of M-PoWerStoreacross multiple untrusted clouds, in the context of a distributed key value store, akin to Amazon S3, building a dependable and robust cross-cloud storage.

REFERENCES

[1] JErasure 0.2.0. https://libraries.io/maven/de.uni-potsdam.hpi.jerasure:JErasure, 2013.

[2] GRPC. Available online at http://www.grpc.io, 2019.

[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] Elli Androulaki, Christian Cachin, Dan Dobre, and Marko Vukolić. Erasure-coded byzantine storage with separate metadata. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings*, pages 76–90, 2014.

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

[7] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *Journal of the ACM (JACM)*, 42(1):124–142, 1995.

[8] Backblaze. JavaReedSolomon. Available online at https://github.com/Backblaze/JavaReedSolomon, 2017.

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

[10] 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.

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

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

[13] Christian Cachin, Dan Dobre, and Marko Vukolić. Separating data and control: Asynchronous BFT storage with 2t + 1 data replicas. In *Stabilization, Safety, and Security of Distributed Systems - 16th International Symposium, SSS 2014, Paderborn, Germany, September 28 - October 1, 2014. Proceedings*, pages 1–17, 2014.

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

[15] Viveck R. Cadambe, Nancy A. Lynch, Muriel Médard, and Peter M. Musial. A coded shared atomic memory algorithm for message passing architectures. *Distributed Computing*, 30(1):49–73, 2017.

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

[17] 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.

[18] 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.

[19] 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.

[20] Dan Dobre, Ghassan Karame, Wenting Li, Matthias Majuntke, Neeraj Suri, and Marko Vukolic. Powerstore: proofs of writing for efficient and robust storage. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 285–298, 2013.

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

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

[23] 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.

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

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

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

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

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

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

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

[31] 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.

[32] Marc Lacoste, Markus Miettinen, Nuno Neves, Fernando M. V. Ramos, Marko Vukolić, Fabien Charmet, Reda Yaich, Krzysztof Oborzynski, Gitesh Vernekar, and Paulo Sousa. User-centric security and dependability in the clouds-of-clouds. *IEEE Cloud Computing*, 3(5):64–75, 2016.

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

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

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

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

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

[38] Dahlia Malkhi and Michael K. Reiter. Byzantine Quorum Systems. *Distributed Computing*, 11(4):203–213, 1998.

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

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

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

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

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

[44] Adi Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, November 1979.

[45] 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.

[46] 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.

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

[48] 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.

[49] Paolo Viotti, Dan Dobre, and Marko Vukolić. Hybris: Robust hybrid cloud storage. *TOS*, 13(3):27:1–27:32, 2017.

[50] 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.

Dan Dobre is a Patent Examiner in European Patent Office in Munich since 2014. He received his PhD degree from TU Darmstadt, Germany, in 2010. Then he worked as a senior researcher in NEC Laboratories Europe, with a focus on fault-tolerance and distributed storage. This work was done while Dan Dobre was working at NEC Laboratories Europe.

Ghassan O. Karame is a Manager and Chief researcher of Security Group of NEC Laboratories Europe. He received his Masters of Science from Carnegie Mellon University (CMU) in December 2006, and his PhD from ETH Zurich, Switzerland, in 2011. Until 2012, he worked as a postdoctoral researcher in ETH Zurich. He is interested in all aspects of security and privacy with a focus on cloud security, SDN/network security and Bitcoin security. He is a member of the IEEE and of the ACM. More information on his research at http://ghassankarame.com/.

Wenting Li is a Senior Software Developer at NEC Laboratories Europe. She received her Masters of Engineering in Communication System Security from Telecom ParisTech in September 2011, and a dipl.ing. degree in Information Security from Shanghai JiaoTong University in 2009. She is interested in system security with a focus on distributed system and IoT devices.

Matthias Majuntke is an Engagement Manager in Capgemini, Berlin. He received his PhD in September 2012 at DEEDS Groups, Computer Science Department, Technische Universität Darmstadt. Before that, he received his Diploma degree in Computer Science from RWTH Aachen University in 2006. This work was done while Matthias Majuntke was working at TU Darmstadt.

Neeraj Suri received his Ph.D. from the University of Massachusetts at Amherst. He currently holds the Chair Professorship in “Dependable Systems and Software” at TU Darmstadt, Germany. His earlier appointments include the Saab Endowed Chair Professorship, faculty at Boston University and multiple sabbaticals at Microsoft Research. His research interests focus on design, analysis and assessment of trustworthy (dependable & secure) distributed systems and software.

Marko Vukolić joined IBM Research in January 2015 as a Research Staff Member and earlier was a Post-Doc and Intern in IBM. Before that, he was a faculty in EURECOM, and a visiting professor at Systems Group @ ETH Zurich. He obtained a Doctor of Science (PhD) degree in Distributed Systems from EPFL in the Distributed Programming Laboratory (LPD) in 2008. His research interests lie in the broad area of distributed systems, more specifically fault-tolerance, blockchain and distributed ledgers, cloud computing security and distributed storage.