Hardening Cassandra Against Byzantine Failures[*]

Roy Friedman[1] and Roni Licher[2]

1 **Department of Computer Science, Technion, Haifa, Israel**
`roy@cs.technion.ac.il`

2 **Department of Computer Science, Technion, Haifa, Israel**
`ronili@cs.technion.ac.il`

Abstract

Cassandra is one of the most widely used distributed data stores. In this work, we analyze Cassandra's vulnerabilities when facing Byzantine failures and propose protocols for hardening Cassandra against them. We examine several alternative design choices and compare between them both qualitatively and empirically by using the Yahoo! Cloud Serving Benchmark (YCSB) performance benchmark.

Some of our proposals include novel combinations of quorum access protocols with MAC signatures arrays and elliptic curve public key cryptography so that in the normal data path, there are no public key verifications and only a single relatively cheap elliptic curve signature made by the client. Yet, these enable data recovery and authentication despite Byzantine failures and across membership configuration changes. In the experiments, we demonstrate that our best design alternative obtains roughly half the performance of plain (non-Byzantine) Cassandra.

1998 ACM Subject Classification C.2.4 Distributed Systems, K.6.5 Security and Protection, B.8.1 Reliability, Testing, and Fault-Tolerance

Keywords and phrases Cassandra, Byzantine Fault Tolerance, Distributed Storage

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2017.27

1 Introduction

Distributed data stores are commonly used in data centers and cloud hosted applications, as they provide fast, reliable, and scalable access to persistently stored data. Persistent storage is fundamental in most applications, yet developing an effective one is notoriously difficult.

Due to inherent tradeoffs between semantics and performance [8] as well as the desire to offer various flexible data management models, a plethora of products has been developed. These differ in the data access model, which can range from traditional relational databases, to wide-columns [12, 28], key-value stores [1, 17], as well as graph databases and more. Another axis by which such systems differ is the consistency guarantees, which can range from strong consistency [29] to eventual consistency [36] and a myriad of intermediate options.

In our work, we focus on Cassandra [28]. Cassandra follows the wide-column model, and offers very flexible consistency guarantees. Among open source data stores, it is probably the most widely used; according to the Cassandra Apache project page [5], more than 1,500 companies are currently using Cassandra, including, e.g., Apple, CERN, Comcast, eBay, GitHub, GoDaddy, Hulu, Instagram, Intuit, Microsoft, Netflix, Reddit, and more.

Cassandra can effectively withstand benign failures, but it was not designed to overcome Byzantine attacks, in which some nodes in the system may act arbitrarily, including in

[*] A full version of the paper is available at [22], `https://arxiv.org/abs/1610.02885`.

a malicious manner. Handling Byzantine failures requires sophisticated protocols and more resources. However, ever since the seminal PBFT work of Castro and Liskov [11], the practicality of building Byzantine fault tolerant replicated state machines has been demonstrated by multiple academic projects, e.g., [13, 24] to name a few. Interestingly, storage systems offer weaker semantics than general replicated state machines, and therefore it may be possible to make them resilient to Byzantine failures using weaker timing and failure detection assumptions, as proposed in [10, 31, 33]. Yet, to the best of our knowledge, we are the first to systematically harden Cassandra against Byzantine failures.

Contributions

We analyze Cassandra's structure and protocols to uncover their vulnerability to Byzantine behavior. We then propose alterations to Cassandra's existing protocols that overcome these failures. We examine several alternative solutions and compare between them qualitatively and quantitatively. Let us emphasize that one of our main design principles is to preserve Cassandra's basic interaction model as much as possible, to increase the likelihood of adoption and to minimize the number of lines of code we need to change. After all, our goal is to harden the existing system, not creating a new one.

We have benchmarked the original Cassandra and our hardened versions of Cassandra using the standard YCSB benchmark [14]. As expected, a key factor to obtain reasonable performance is in the type of cryptography used. E.g., using traditional RSA signatures dramatically lowers the performance. Conversely, using only (symmetric key) MAC signatures is challenging when data must survive configuration changes and be shared by multiple clients. To that end, here we propose a novel combination of vectors of MACs with the *Elliptic Curve Digital Signature Algorithm* (ECDSA) [27]. The normal data path of this proposal incurs no public key verifications and only a single relatively cheap ECDSA signature made by the client. Yet, these enable data recovery and authentication despite Byzantine failures and across membership configuration changes. The measured performance of the best configuration of our hardened Cassandra, utilizing our new approach, is only twice worse than the original Cassandra; an order of magnitude better than the naive approach.

Note that while using an array of MAC signatures instead of PKI has been done in the past, e.g., in [11], prior use of this idea was restricted to ordering decisions that are contained within a single configuration. In contrast, in our work we ensure data survivability across configuration changes, which requires the extra protection of the elliptic curve signature.

2 Related Work

The seminal work of Castro & Liskov [11] inspired multiple extensions. Clement et al. [13] introduced UpRight, a modular library for BFT replicated state machine. BFT-SMaRt [7] and Prime [3] have improved the performance when facing Byzantine behaviour, whereas Abstract [24] is the state of the art in BFT replicated state machine. It adds the ability to abort a client request when faults occur and then dynamically switch to another BFT protocol that produces better results under the new system conditions. The work of [35] explored adding BFT resilience to transactional systems that follow the scalable differed update replication model.

Malkhi & Reiter [33] were the first to discuss *Byzantine* quorum systems, i.e., using read and write quorums such that any two quorums intersect in at least one *correct* node. Furthermore, the system remains available in spite of having up to f Byzantine nodes.

Aguilera & Swaminathan [2] explored BFT storage for slow client-server links while relying on synchronized clocks. They designed a *linearizable abortable* register in which partial writes due to benign client failures do not have any effect by using unique timestamps and timestamp promotion when conflicts appear. Yet, they did not show an actual implementation nor performance analysis. As our work preserves Cassandra's semantics, we are able to design faster operations requiring lighter cryptography measures even when conflicts occur.

Byzantine clients might try to perform *split-brain-writes*, i.e., writing multiple values to different servers using the same timestamp. Preventing this can be done by obtaining a commitment from a quorum to bind a timestamp and a value on every write. In Malkhi & Reiter's approach [33], on every write, the servers exchange inter-servers messages agreeing on the binding. In Liskov & Rodrigues's approach [31], the servers transmit signed agreements to the client that are later presented to the servers as a proof for the quorum agreement. In contrast, we do not prevent split-brain-writes, but rather repair the object's state on a read request (or in the background).

Some BFT cloud storage systems provide *eventual consistency* [36]. ZENO [39] ensures *causal order consistency* [29] with at least $f+1$ correct nodes. DEPOT [32] tolerates any number of Byzantine clients and servers and guarantees *Fork-Join-Causal order consistency*.

Aniello et al. [4] explored Byzantine DoS attacks on gossip based membership protocols. They have demonstrated their attack on Cassandra [28] and presented a way to prevent it by using signatures on the gossiped data. Other more general solutions for BFT gossip membership algorithms were shown in FIREFLIES [26] and BRAHMS [9]. The first uses digital signatures, full membership view and a pseudorandom mesh structure and the latter avoids digital signatures by sophisticated sampling methods.

Sit & Morris [40] mapped classic attacks on *Distributed Hash Tables* (DHT). Some of the attacks can be disrupted by using SSL. Other attacks described in [40], such as storage and retrieval attacks, are addressed in our work.

Non malicious value and state error failures in Cassandra and other distributed storage systems have been explored in [15, 23]. Solutions to such problems can be more efficient than Byzantine resilient implementations, but inherently only protect against a more restrict failure scenario.

3 Model and Assumptions

We assume a Cassandra system consisting of *nodes* and *clients*, where each may be *correct* or *faulty* according to the Byzantine failure model [30]. A correct entity acts according to its specification while a faulty entity can act arbitrarily, including colluding with others.

In our proposed solutions, we assume that the maximal number of faulty nodes is bounded by f. We initially assume that all clients are *correct*, but later relax this assumption. When handling Byzantine clients, we do not limit the number of faulty clients nor change the assumption on the maximal number of f faulty nodes. Yet, we assume that clients can be authenticated so correct nodes only respond to clients that are allowed to access the system according to some verifiable *access control list* (ACL). We use the terms nodes and processes interchangeably and only to refer to Cassandra nodes.

We assume a fully connected partially synchronous distributed system. Every node can directly deliver messages to every other node and every client can directly contact any system node. We also assume that each message sent from one correct entity to another will eventually arrive exactly once and without errors. That can be implemented, e.g., on top of fair lossy networks, using retransmission and error detection codes. We do not assume any bound on message delay or computation time in order to support our safety and liveness properties. However, efficiency depends on the fact that most of the time messages and computation steps do terminate within bounded time [19].

Every system entity has a verifiable PKI certificate. We assume a trusted *system administrator* that can send signed membership configuration messages.

The system shares a loosely synchronized clock which enables detection of expired PKI certificates in a reasonable time but is not accurate enough to ensure coordinated actions. We discuss this clock in Appendix B.3.

4 Brief Overview of Cassandra

Cassandra stores data in tables with varying number of columns. Each node is responsible for storing a range of rows for each table. Values are replicated on multiple nodes according to the configurable *replication factor*.

Mapping data to nodes follows the *consistent hashing* principle, where nodes are logically placed on a virtual ring by hashing their ids. In fact, each node is represented as multiple *virtual nodes* [17]. Each virtual node generates a randomized key on the ring, called a *token*, representing its *place*. A virtual node is responsible for hashed keys that fall in the range from its place up to the next node on the ring, known as its *successor*. Each node also stores keys in the ranges of the $N - 1$ preceding nodes, where N is the replication factor parameter. The N nodes that should store a given value are called its *replication set*.

Cassandra uses a *full membership view*, where every node knows every other node. A node that responds to communication is considered *responsive* and otherwise it is *suspected*. Nodes exchange their views via *gossip* [41]. The gossip is disseminated periodically and randomly; every second, each node tries to exchange views with up to three other nodes: one responsive, one suspected, and a *seed* [28]. A new node starts by contacting seed nodes.

Cassandra provides tunable consistency per operation. On every operation, the client can specify the *consistency level* that determines the number of replicas that have to acknowledge the operation. Some of the supported consistency levels are: *one* replica, a *quorum* [25] of replicas and *all* of the replicas. According to the consistency level requested in the writes and in the respective reads, *eventual consistency* [36] or *strong consistency* can be achieved.

On each operation, a client connects to any node in the system. This selected node acts as a *proxy* on behalf of the client and contacts the relevant nodes using its view of the system. In the common configuration, the client selects a proxy among all system nodes in a *Round Robin* manner. The proxy node may contact up to N nodes that are responsible for storing the value according to the requested consistency level. If the required threshold of responses is satisfied, the proxy will acknowledge the write or forward the latest value, according to the stored timestamp, to the client. If the proxy fails to contact a node on a write, it stores the value locally and tries to update the suspected node at a later time. The stored value is called *hinted handoff* [16]. If a proxy receives multiple versions on a read query, it performs a *read repair* to update nodes that hold a stale version with the most updated one. As of Cassandra 1.0, this replaces the earlier *sloppy quorums* [17] approach.

If a node is unresponsive for a long time, hinted handoffs that were saved for this node may be deleted. Similarly, a hinted handoff may not reach its targeted node if the node that stores it fails. This is overcome with Cassandra's manual *anti-entropy* tool, where nodes compute and exchange *Merkle trees* for their values and sync the outdated ones.

The primary language for communicating with Cassandra is the *Cassandra Query Language* (CQL) [16]. Here, we focus on put and get commands as available in standard NoSQL key-value databases and ignore other options.

5 Hardened Cassandra

We identify Byzantine vulnerabilities in Cassandra and suggest ways to overcome them.

5.1 Impersonating

Cassandra supports SSL. Yet, sometimes messages need to be authenticated by a third party, e.g., a read response sent from a node to a client through a proxy node, which we support through digital signatures. In both SSL and digital signatures, we depend on PKI.

Digital signatures are divided into two main categories: *public/private keys* vs *MAC tags*. Public key signatures are more powerful as they enable anyone to verify messages. The downside of public key signatures is their compute time, which is about 2-3 orders of magnitude slower than MAC tags and these signatures are significantly larger, e.g., RSA 2048b versus AES-CBC MAC 128b.

5.2 Consistency Level

Recall that Cassandra offers a configurable replication factor N as well as the number of nodes that must acknowledge each read (R) and each write (W). This threshold can be one node or a quorum (majority in Cassandra) or all N nodes. When up to f nodes are Byzantine, querying fewer than $f+1$ nodes may retrieve old data (signed data cannot be forged), violating the consistency property. On the other hand, querying more than $N-f$ nodes may result in loss of availability. We present two approaches: (1) using Byzantine quorums for obtaining Strong Consistency and (2) using Cassandra quorums with a scheduled run of the anti-entropy tool for obtaining *Byzantine Eventual Consistency*.

Byzantine Quorums

By requesting that each read and each write will intersect in at least $f+1$ nodes, we ensure that every read will intersect with every write in at least one *correct* node. That is, $R+W \geq N+f+1$. As for liveness, we must require that $R \leq N-f, W \leq N-f$. By combining these requirements, we obtain: $N \geq 3f+1$.

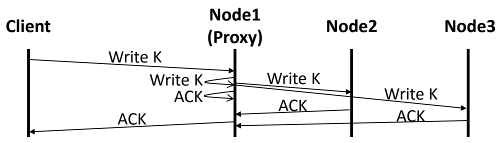
The last bound was formally proved by Malkhi & Reiter [33]. Cachin et al. [10] have lowered this bound to $2f+1$ by separating between the actual data and its *metadata*; storing the medadata still requires $3f+1$ nodes. The above separation was presented under the assumptions of benign writers and Byzantine readers.

Cachin's solution is beneficial for storing large data items. Yet, when storing small values, the method of [10] only increases the overhead. A system may offer either solution according to its usage, or employ both in a hybrid way, according to each value's size.

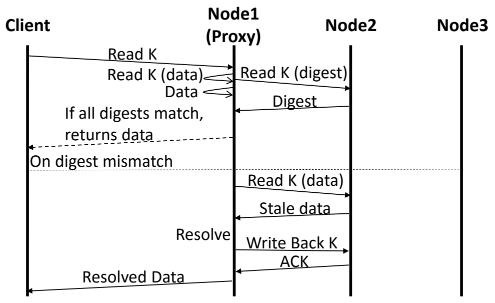
Byzantine Eventual Consistency

For eventual consistency, all replication set nodes must eventually receive every update. Further, writes order conflicts should be resolved deterministically. Here, there is no bound on the propagation time of a write, but it should be finite. In particular, if no additional writes are made to a row, eventually all reads to that row will return the same value.

Byzantine eventual consistency can be obtained through majority quorums. In this approach, the replication set is of size $2f+1$ nodes while write and read quorums are of size of $f+1$. Hence, each write acknowledged by $f+1$ nodes is necessarily executed by at least one correct node. This node is trusted to update the rest of the nodes in the background.



(a) The write algorithm in original Cassandra.



(b) The read algorithm in original Cassandra.

Figure 1 Original Cassandra. Configuration: N=3 and R=2.

As this node is correct, it will eventually use the anti-entropy tool to update the rest of the replication set. Recall that the client request is signed so the servers will be able to authenticate this write when requested.

Every read is sent to $f+1$ nodes and thus reaches at least one correct node. This correct node follows the protocol and accepts writes from proxy nodes and from the anti-entropy tool. So, eventually, it retrieves the latest update. Due to the cryptographic assumptions, a Byzantine node can only send old data and cannot forge messages. Hence, on receiving a value from the anti-entropy tool that does not pass the signature validation, we can use it as a Byzantine failure detector and notify the system administrator.

5.3 Proxy Node

Figures 1a and 1b present the current write and read flows in Cassandra, including the role of proxies. A Byzantine proxy node can act in multiple ways, such as (1) respond that it has successfully stored the value without doing so, (2) perform a split-brain-write, and (3) respond that the nodes are not available while they are. We augment the existing flows of writing and reading in Cassandra to overcome these vulnerabilities below.

Naive Hardened Write Operation

Our modified write algorithm appears in Figures 2 and 8. Here, when writing a new value, the client signs the value and a node will store it only if it is signed by a known client according to the ACL with a timestamp that is considered fresh (configurable). On each store, the storing node signs an acknowledgment that the client can verify. The signed acknowledgment also covers the timestamp provided by the client, preventing replay attacks by the proxy. A client completes a write only after obtaining the required threshold of signed responses, which now the proxy cannot forge. If one proxy fails to respond with enough signed acknowledgments in a configurable reasonable time, the client contacts another node to serve as an alternative proxy for the operation. After contacting at most $f+1$ proxy nodes (when needed), the client knows that at least one *correct* proxy node was contacted.

Naive Hardened Read Operation in Details

The read algorithm has three parts: (1) Reading data from one node and only a digest from the rest. As an optimization, the read may target only a known live quorum instead of all relevant nodes. (2) On digests mismatch, a full read is sent to all contacted nodes from the

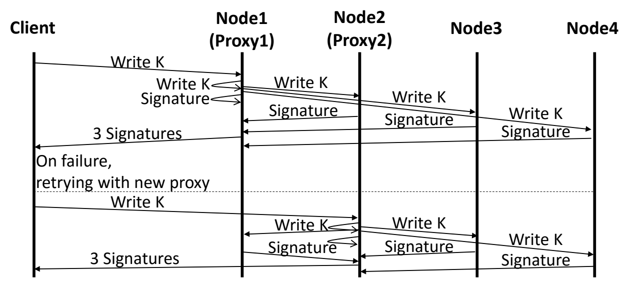


Figure 2 Illustrating our write algorithm from Figure 8 where the proxy verifies each store acknowledgment. Configuration: N=4 and W=3.

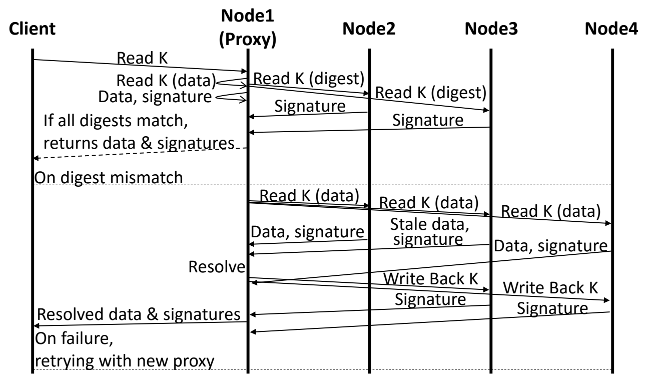


Figure 3 Illustrating our read algorithm from Figure 9 where the proxy verifies. N=4 and R=3.

first phase, retrieving the data. (3) The proxy resolves the conflict by creating a row with the most updated columns according to their timestamps, using lexicographical order of values as tie breakers when needed. The resolved row is sent back to out-dated nodes.

Figures 3 and 9 present our modified read algorithm, including the following changes: (1) In case the first phase is optimized by addressing only a known live quorum of nodes, if a failure occurs, we do not fail the operation but move to a full read from all nodes. Thus, if a Byzantine node does not respond correctly, it does not fail the operation. (2) If there is a digest mismatch in the first phase, we do not limit the full read only to the contacted nodes from the first phase but rather address all replication set nodes. Hence, Byzantine nodes cannot answer in the first phase and fail the operation by being silent in the second phase. (3) During resolving, the nodes issue a special signature, notifying the client about the write back. The proxy then supplies the original answers from the first phase to the client, all signed by the nodes. This way, the client can authenticate the resolving's correctness.

The motivation for forwarding the original answers from the proxy to the client relates to the fast write optimization of Cassandra, where all writes are appended to a commit log and reconciled in the background or during a following read request. Hence, while stale values are saved, if there is already a newer value, a stale value would not be served by any read. Unfortunately, this optimization exposes an attack by which a Byzantine proxy would request a quorum of nodes to store an old value. By providing the client with the original answers, it can verify that the write-back was necessary and correct.

5.3.1 Targeting Irrelevant Nodes

Byzantine proxies may direct read requests to irrelevant nodes, who will return a verifiable empty answer. We consider three remedies for this: (1) Using clients that have full membership view, already supported by Cassandra, so a client knows which nodes have to respond. (2) Using an authentication service that is familiar with the membership view and can validate each response. A client can use this service to authenticate answers. (3) Configure the nodes to return a signed 'irrelevant' message when requested a value that they are not responsible for. Here, clients avoid counting such 'irrelevant' answers as valid responses.

A Byzantine proxy can forward updates only to members of a single write quorum to decrease long term availability. The anti-entropy tool is periodically invoked to overcome this. It requires each value to be accompanied by a correct client signature for authentication.

5.3.2 Proxy Acknowledgments Verification

In our proposed solution as presented so far, we have requested the proxy to verify the nodes acknowledgments and accept a response only if it is signed correctly. Yet, both hardened read and write protocols forward to the client digitally signed acknowledgments so the latter can authenticate the completion of the operation by the nodes. This motivates shifting the entire verification from the proxy to the client, to reduce the proxy load. Below, we identify the challenges in enabling this optimization and offer solutions:

1. Consider a correct proxy and f Byzantine nodes. The Byzantine nodes may reply faster with bad signatures (they need not verify signatures nor sign). The proxy then returns to the client $f + 1$ good signatures and f bad signatures. Contacting an alternative proxy now might produce the same behavior.
2. Consider a colluding Byzantine proxy that is also responsible to store data itself. On a write, the proxy asks the Byzantine nodes to produce a correct signature without storing the value. The proxy also asks one correct node to store the data and produces false f signatures for some nodes. The client will get $f + 1$ correct signatures and f bad signatures, while only one node really stored the value.

To overcome the above, we let the client contact the proxy again in case it is not satisfied with the $2f + 1$ responses it obtained. On a read, the client requests the proxy to read again without contacting the nodes that supplied false signatures. On a write, the client requests the proxy to fetch acknowledgments from additional nodes.

As mentioned above, the motivation for this alternative is that signatures verification is a heavy operation. In the proxy verification option, on every write, the proxy is required to perform at least $2f + 1$ signature verifications. In the alternative client only verification option, the latency penalty will be noticed only when Byzantine failures are present and could be roughly bounded by the time of additional RTT (round-trip-time) to the system and f parallel RTT's inside the system (counted as one), multiplying it all by f (the number of retries with alternative proxies). Assuming that in most systems Byzantine failures are rare, expediting the common correct case is a reasonable choice.

The client only verification option also enables using MAC tags instead of public signatures, since only the client verifies signatures. To that end, a symmetric key for each pair of system node and client should be generated. Every client has to store a number of symmetric keys that is equal to the number of system nodes. Every node has to store a number of symmetric keys that is equal to the number of (recently active) clients. These keys can be pre-configured by the system administrator or be obtained on the first interaction through a secure SSL line. This yields significant speedups both for the node signing and for the client verification. The exact details appear in [22].

5.3.3 Proxy Resolving vs. Client Resolving

Recall that when Cassandra's read operation detects an inconsistent state, a resolving process is initiated to update outdated replicas. This way, the chance for inconsistency in future reads decreases. In plain Cassandra as well as in our solution as presented so far, the proxy is in charge of resolving such inconsistent states. An alternative option is to let the client resolve the answers and write back the resolved value using a write request that specifies to the proxy which replicas are already updated.

To prevent Byzantine nodes from manipulating the resolver with false values, the resolver must verify the client signature on each version. When combining the client resolving option with using a proxy that is not verifying nodes acknowledgments (as discussed in Section 5.3.2), the proxy no longer needs to verify any client signatures, improving its scalability. The exact details appear in [22].

5.3.4 From Public Key Signatures to MAC Tags

The use of public key signatures has a major performance impact while switching to MAC tag is not trivial. In Section 5.3.2, we described how to switch from public key signatures to MAC tags on messages sent from nodes to clients.

Supporting MAC tags on messages sent from clients to nodes poses the following challenges: (1) Joining new nodes to Cassandra. These nodes have to fetch stored values for load-balancing. As a client does not know who these future nodes are, it cannot prepare MAC tags for them. One solution is that a new node will only store values that were found in at least $f + 1$ nodes. Alternatively, the client may re-write all values that the new node has to store. (2) Using the anti-entropy tool and resolving consistency conflicts need to ensure the correctness of the values by contacting at least $f + 1$ nodes that agree on the values. Alternatively, every node will have to store a vector of MAC tags for each responsible node. Storing a signature vector poses another challenge: a Byzantine proxy can manipulate the signatures vector sent to each node, leaving only the node's signature correct and corrupting all other nodes' signatures (making the stored vector useless). This can be overcome by adding another MAC tag on the signatures vector, proving to the node that the tags vector was not modified by the proxy.

Due to these issues and in order to speed up the write path, we suggest a hybrid solution as presented in Figure 4. A write is signed with a public key signature, which is covered by MAC tags, one for each node. A node then verifies only the MAC tag and stores only the public key signature. Hence, in the common case, we use a single public key signature with no public key verifications at all. When things go bad, we fall back to the public key signature. Furthermore, use of ECDSA [27] offers much faster signing than *RSA* [37].

Finally, when using MAC tags on the client to node path, the client should know what are the relevant nodes for that key. One solution is to ensure clients are updated about the nodes tokens. This way, on every write, the client knows what keys to use. Since our solution has a fall back option, even if there is a topology change that the client is late to observe, the new node (targeted by the proxy) can still use the public signature and not fail the write. On the write acknowledgment, the new node can update the client about the (signed) topological change.

5.3.5 Comparing The Variants

Tables 1, 2 and 3 summarize the alternatives proposed in this section. We focus on the number of signing and verification operations of digital signatures as these are the most

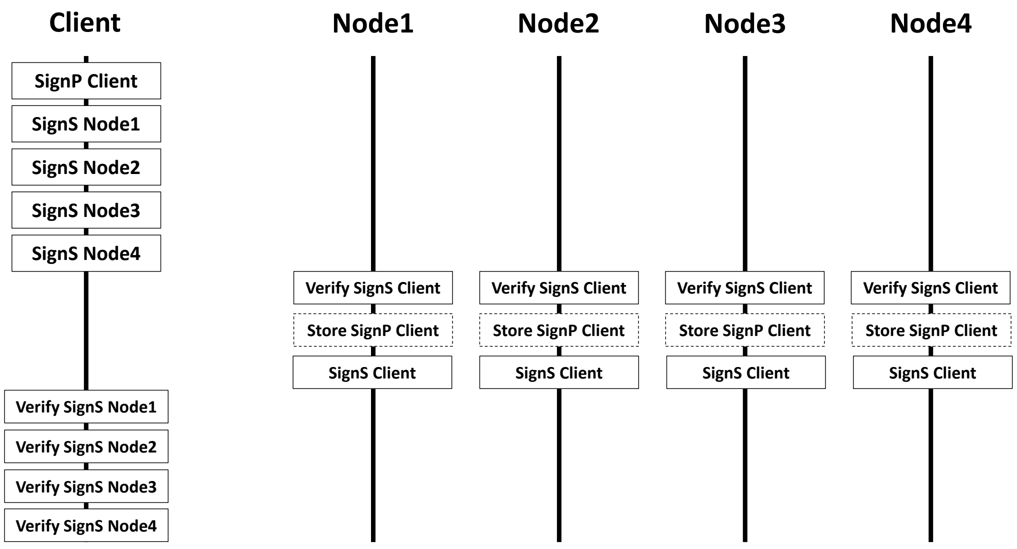


Figure 4 Illustration of our hybrid signing solution. The *SignP* stands for public key signature, using the private key of the signing entity. The *SignS* stands for MAC tag.

Table 1 Comparing the variants of our solution with the most optimist assumptions. C is the number of columns, (p) indicated public key signatures and (s) MAC tags. When the proxy does not verify, we refer both to the proxy resolves and client resolves modes. We assume that on a read, the proxy uses the optimization in the first phase and contacts only a Byzantine quorum and not all replicas. For example, the forth row presents a proxy that does not verify acknowledgments and MAC tags are used from client to nodes and from nodes to client. In this variant, the client signs the C columns using public key signatures and adds $3f + 1$ MAC tag, one for each node. All nodes $(3f + 1)$ have to store it and they verify only their MAC tags. All nodes issue verifiable acknowledgments $(3f + 1)$ and the client verifies only a Byzantine quorum $(2f + 1)$.

Proxy Verifies?	Op	MAC Tags	Signatures	Verifications
Yes	Write	None	Client: C(p) Nodes: $3f + 1$(p)	Nodes: $(3f + 1) \cdot C$(p) Proxy: $2f + 1$(p) Client: $2f + 1$(p)
No	Write	None	Client: C(p) Nodes: $3f + 1$(p)	Nodes: $(3f + 1) \cdot C$(p) Client: $2f + 1$(p)
No	Write	Nodes to client	Client: C(p) Nodes: $3f + 1$(s)	Nodes: $(3f + 1) \cdot C$(p) Client: $2f + 1$(s)
No	Write	Both ways	Client: C(p) & $3f + 1$(s) Nodes: $3f + 1$(s)	Nodes: $3f + 1$(s) Client: $2f + 1$(s)
Yes	Read	None	Nodes: $2f + 1$(p)	Proxy: $2f + 1$(p) Client: $2f + 1$(p)
No	Read	None	Nodes: $2f + 1$(p)	Client: $2f + 1$(p)
No	Read	Nodes to client	Nodes: $2f + 1$(s)	Client: $2f + 1$(s)

time consuming. We divide our analysis in three: (1) best case and no failures, (2) a benign mismatch on a read that requires resolving, and (3) worst case with f Byzantine nodes.

5.4 Handling Byzantine Clients

Notice that some Byzantine clients actions are indistinguishable from correct clients behaviors. For example, erasing data or repeatedly overwriting the same value. Yet, this requires the client to have ACL permissions.

Here, we focus on preserving data consistency for correct clients. I.e., a correct client should not observe inconsistent values resulting from a split-brain-write nor should it read older values than values returned by previous reads.

Specifically, we guarantee the following semantics, as in plain Cassandra: (1) The order between two values with the same timestamp is their lexicographical order (breaking ties

Table 2 Comparing the variants in the read flow in case of a benign mismatch that requires resolving. C is the number of columns, M is the number of outdated replicas in the used quorum, (p) indicated public key signatures and (s) MAC tags. We assume that the proxy uses the optimization in the first phase and contacts only a Byzantine quorum. For example, the first row presents a proxy that verifies the acknowledgments and resolves conflicts when mismatch values are observed. MAC tags are not in use. On a read request, a Byzantine quorum of nodes $(2f + 1)$ have to retrieve the row and sign it. The proxy verifies their signatures $(2f + 1)$ and detects a conflict. Then, the proxy requests all relevant nodes (except for the one that returned data in the first phase) for the full data ($3f$ nodes sign and the proxy verifies only $2f$). The proxy resolves the mismatch (verifies C columns) and sends the resolved row to the M outdated nodes (write-back). These nodes verify the row (C) and sign the acknowledgments that are later verified by the proxy. The proxy supply the client with the original $2f + 1$ answers and the resolved row signed also by M nodes that approved the write-back.

Proxy Verifies?	**Mismatch Resolving**	**MAC tags**	**Signatures**	**Verifications**
Yes	Proxy	No	Nodes: $5f+1+M$(p)	Nodes: $M \cdot C$(p) Proxy: $4f+1+C+M$(p) Client: $2f+1+M$(p)
No	Proxy	No	Nodes: $5f+1+M$(p)	Nodes: $M \cdot C$(p) Proxy: C(p) Client: $2f+1+M$(p)
No	Proxy	Yes	Nodes: $5f+1+M$(s)	Nodes: $M \cdot C$(p) Proxy: C(p) Client: $2f+1+M$(s)
No	Client	No	Nodes: $5f+1+M$(p)	Nodes: $M \cdot C$(p) Client: $2f+1+C+M$(p)
No	Client	Yes	Nodes: $5f+1+M$(s)	Nodes: $M \cdot C$(p) Client: $2f+1+M$(s) & C(p)

Table 3 Comparing the variants of the read and write flows in the worst case f Byzantine nodes. Due to the wide options of Byzantine attacks and the fact that every Byzantine node can waste other node's cycles, we compare the variants only from the point of view of a correct client. C is the number of columns, M is the number of outdated replicas in the used quorum, (p) indicated public key signatures and (s) MAC tags. For example, the second row presents a proxy that does not verify the acknowledgments in a write operation. MAC tags are not in use. On a write request, the client signs the C columns and sends it to the proxy. The client receives from the proxy responses from a Byzantine quorum of nodes $(2f + 1)$ and detects that one is incorrect. The client requests the proxy f more times for the missing signature and every time gets a false signature. Then, the client uses alternative proxies repeatedly f additional times. At last, the client successfully retrieves all $2f + 1$ correct signatures due to our assumption on f.

Proxy Verifies?	**Op**	**Mismatch Resolving**	**MAC Tags**	**Signatures**	**Verifications**	**Client-Proxy Requests**
Yes	Write	-	None	C(p)	$(2f+1)(f+1)$(p)	$f+1$
No	Write	-	None	C(p)	$(3f+1)(f+1)$(p)	$(f+1)(f+1)$
No	Write	-	Nodes to client	C(p)	$(3f+1)(f+1)$(s)	$(f+1)(f+1)$
No	Write	-	Both ways	C(p)	$(3f+1)(f+1)$(s)	$(f+1)(f+1)$
Yes	Read	Proxy	None	None	$(2f+1+M)(f+1)$(p)	$(f+1)$
No	Read	Proxy	None	None	$(2f+1+M)(f+1)(f+1)$(p)	$(f+1)(f+1)$
No	Read	Client	None	None	$(2f+1)(f+1)(f+1)$ $+C+(M+f)(f+1)$(p)	$(f+1)(f+1)+(M+f)(f+1)$
No	Read	Proxy	Nodes to client	None	$(2f+1+M)(f+1)(f+1)$(s)	$(f+1)(f+1)$
No	Read	Client	Nodes to client	None	$(2f+1)(f+1)(f+1)+(M+f)(f+1)$(s) & C(p)	$(f+1)(f+1)+(M+f)(f+1)$

according to their value). (2) A write of multiple values with the same timestamps is logically treated as multiple separate writes with the same timestamp. (3) Deleting values is equivalent to overwriting these values with a *tombstone*. (4) A read performed by a correct client must return any value that is not older (in timestamp order) than values returned by prior reads. (5) A read performed after a correct write must return a value that is not older (in timestamp order) than that value.

As mentioned, if the proxy handling a read observes multiple versions from different nodes, it resolves the mismatch and writes the resolved value back to the nodes. The resolved version will be a row with the most updated columns according to their timestamps. If the proxy observes two values with the same timestamp, it will use the lexicographical order of the values as a tie breaker.

For split-brain-writes, Byzantine clients colluding with Byzantine proxies may sign multiple values with the same timestamp. Proxies can send these different values with the same timestamps to different nodes for a split-brain-write. Notice that a split-brain-write could occur spontaneously in plain Cassandra by two correct clients that write in parallel since in Cassandra clients provide the write's timestamp, typically by reading their local clock.

Consider a Byzantine client colluding with a proxy to perform a split-brain-write. Due to the resolve mechanism, a read that sees both values returns only the latest value in lexicographical order. Further, on a client read, the proxy resolves the conflict and updates a quorum of servers with that version, restoring this value's consistency.

If a Byzantine client and a colluding proxy try to update only part of the nodes with a certain v, a read may return two kinds of values: (1) If the read quorum will witness v, it will be resolved and propagated to at least a quorum of nodes meaning that v will be written correctly. As a result of this resolve, every following read will return v (or a newer value). (2) If a read will not witness v, the most recent version of a correct write will be returned. Thus, the hardened system protects against such attempts.

Finally, if a Byzantine client is detected by the system administrator and removed, its ACL and certificate can be revoked immediately. This way any potentially future signed writes saved by a colluder will be voided.

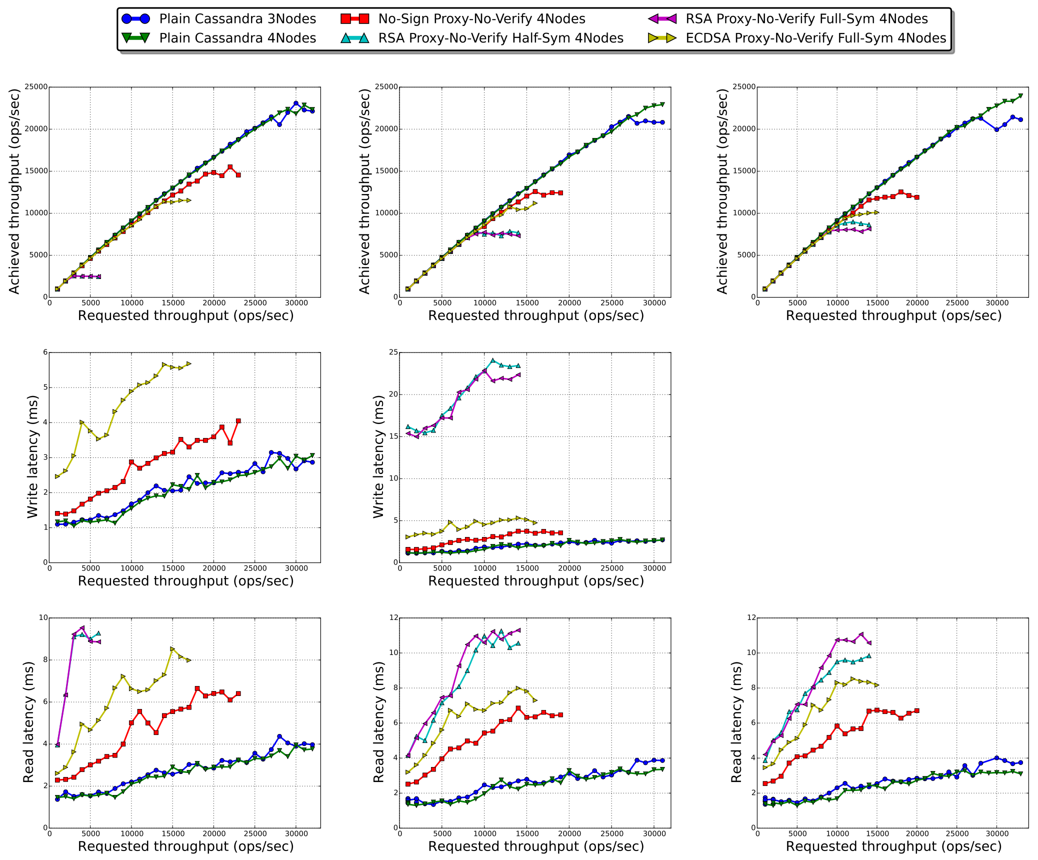
5.5 Deleting Values

In Cassandra, deleting a value is done by replacing it with a *tombstone*. The tombstone is replied to any request for this value to indicate its deletion. Occasionally, a garbage collector removes all tombstones that are older than a configurable time (10 days by default).

Even in a benign environment, a deleted value might reappear, e.g., when a failed node recovers after missing a delete operation and passing all garbage collection intervals in other nodes. Mimicking this, a Byzantine node can ignore delete operations and propagate the deleted values to correct nodes after the garbage collection interval.

We define the following counter-measures: (1) Every delete is signed by a client as in the write operation. This signature is stored in the tombstone. A client completes a delete only after obtaining a Byzantine quorum of signed acknowledgments. (2) During each garbage collection interval, a node must run at least once the anti-entropy tool among a Byzantine quorum of nodes, fetching all missed tombstones. (3) A node will accept writes of values that are not older than the configured time for garbage collection interval. Since the node runs the anti-entropy tool periodically, even if a deleted value is being fetched, the tombstone will overwrite it. (4) A node that receives a store value older than the configured garbage collector time will issue a read for the value and accept it only if a Byzantine quorum approves that the value is live.

We explore a few additional attack vectors, including membership, clocks and network attacks in Appendix B.



(a) Workload A **(b)** Workload B **(c)** Workload C

Figure 5 Comparing the best variants against plain Cassandra and the algorithm with *No-Sign* using workloads A, B and C. In the write latency of (a), we left the RSA variants out as they rapidly grew to ≈65ms latency.

6 Performance

We implemented our algorithms[1] as patches to Cassandra 2.2.4[2]. We evaluated the performance of the variants of our solution and compared them to the original Cassandra using the standard YCSB 0.7[3] benchmark [14], adjusted to use our BFT client library[4]. We used Datastax's Java driver 2.1.8[5] on the client side. There are nearly 390K *LOC* (lines of code) in Cassandra. Our patch added about 3.5K LOC to the servers code and about 4K LOC to the client code (including YCSB integration), which uses the client driver as is. Our entire code adds less than 2% LOC.

All experiments were run on 4 to 5 machines (Ubuntu14, dual 64-bit 6 core 2.2GHz Intel Xeon E5 CPUs, 32GB of RAM, 7200 RPM hard drive and 10Gb ethernet), one for the client and three to four for the Cassandra nodes.

1 https://github.com/ronili/HardenedCassandra
2 https://github.com/apache/cassandra/tree/cassandra-2.2.4
3 https://github.com/brianfrankcooper/YCSB/tree/0.7.0
4 https://github.com/ronili/HardenedCassandraYCSB
5 https://github.com/datastax/java-driver/tree/2.1.8

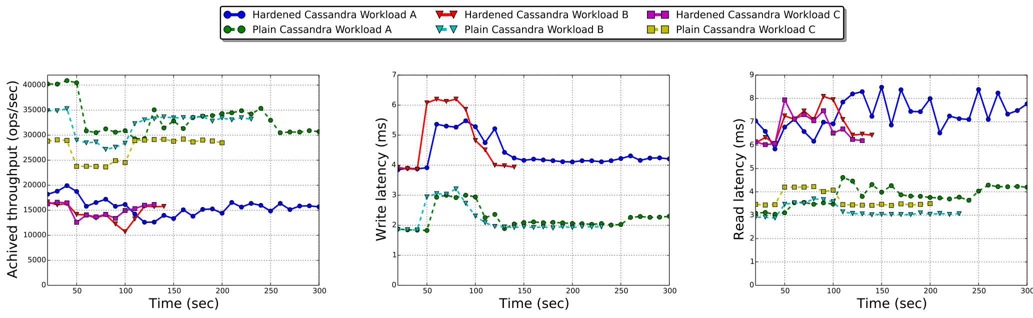


Figure 6 Comparing our best solution (ECDSA Proxy-No-Verify Full-Sym) against plain Cassandra in a benign failure of one node in a 4 nodes setup.

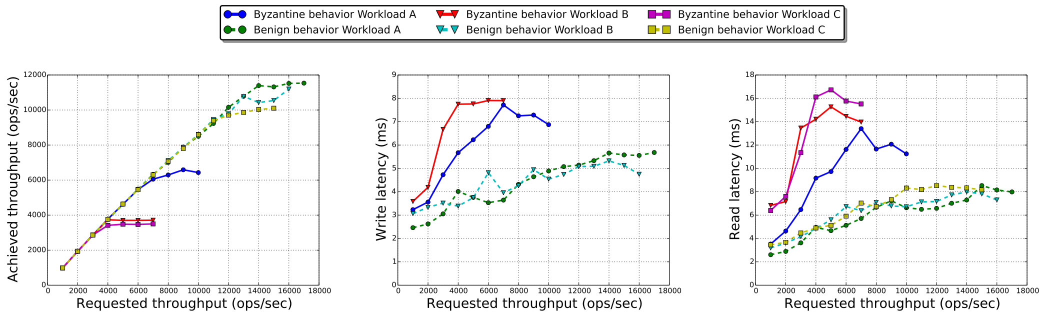


Figure 7 Comparing the best solution in benign behavior and having one node only replying with bad signatures.

We pre-loaded the database with 100,000 rows and benchmarked it with five YCSB workloads as follows: (1) Workload A - 50/50 reads/writes. (2) Workload B - 95/5 reads/writes. (3) Workload C - only reads. (4) Workload D - 95/5 reads/writes, where the reads are for the latest inserts (and not random). (5) Workload F - 50/50 writes/Read-Modify-Writes. In all workloads other than D, the write is for one column while in workload D it is for the entire row. Every workload ran with 100 client threads, performing a total of 100,000 operations with varying throughput targets. The tables consisted of 10 columns (default in YCSB) as well as tables consisting of one value, modeling a key-value datastore. Each value is of size 100 bytes while the key size is randomly chosen in the range of 5-23Bytes. Thus, each record/line with 10 columns has an average length of 1014Bytes. As YCSB throttles the requests rate to the achieved maximum throughput, we run each experiment until obtaining a stable throughput.

We implemented the algorithms of Figures 8 and 9 where the proxy authenticates the acknowledgments; denote these as *Proxy-Verifies*. We also implemented the variant where the proxy does not verify the acknowledgments and lets the client fetch more acknowledgments in case it is not satisfied; denote it *Proxy-No-Verify*. We ran that last algorithm in two modes, one where the proxy is in charge of resolving inconsistent reads, and one where the client is. We present only the former as both behaved similarly.

We analyzed MAC tags in two steps: (1) using MAC tags on messages from nodes to client, denoted *Half-Sym* and (2) using it for both ways, denoted *Full-Sym*.

We used two types of private key signatures: (1) RSA with keys of length of 2048b and (2) ECDSA with keys of length 256b and the *secp256r1* curve. For symmetric keys, we used keys of length of 128b with the *HMAC* MAC algorithm and *SHA256* [20] for hashing.

To evaluate the cost of our algorithm without cryptographic overhead, we ran them also without any signatures. That is, we swapped the signing methods with a *base64* encoded on a single char, referred to as *No-Sign*.

6.1 Performance with No Failures

Figures 5 presents the performance results for the multi-column model for workloads A-C. Workloads D and E are qualitatively similar and can be found in [22]. Our best solution is the variant where the proxy does not verify the acknowledgments, and we use ECDSA and MAC tags for both ways (ECDSA Proxy-No-Verify Full-Sym 4Nodes). The slowdown here is roughly a factor of 2-2.5 in terms of the maximum throughput, 2.5-3 in the write latency and 2-4 in the read latency. The No-Sign experiment represents the BFT algorithmic price including larger quorums, extra verifications and storing signatures. The ECDSA experiment adds the cryptography cost. Notice that RSA has a significant negative performance impact.

We have also studied the performance in the key-value model, i.e., a table with one non-key column. For lack of space, the results are in the full version [22].

6.2 Performance Under Failures

Figure 6 presents the performance of our best solution in a single stopped node scenario. We run workload A on a four nodes setup, with maximum throughput. After 50 seconds, we stopped one node for 30 seconds and then restarted it. It took the node between 20 to 30 seconds to start, after which the other nodes started retransmitting the missed writes to the failed node. In our best solution, the distributed retransmitting took about 250 seconds and in the plain Cassandra, about 170 seconds. We repeated this test with workloads B and C with one change, failing the node in $t = 40$ instead of $t = 50$. In this experiment too our solution performs similarly to plain Cassandra.

In Figure 7, we present the performance of our best solution when one node always returns a bad signature. This impacts the entire system as on every failed signature verification, the client has to contact the proxy again. Further, on every read that addresses the Byzantine node, a resolving and a write-back process is initiated.

We have also explored a stalling proxy attack, where the proxy waits most of the timeout duration before supplying the client with the response. See Appendix C.1.

7 Conclusion

Cassandra's wide adoption makes it a prime vehicle for exploring various aspects of distributed data stores. In our work, we have studied Cassandra's vulnerabilities to Byzantine failures and explored various design alternatives for hardening it against such failures.

We have also evaluated the attainable performance of our design alternatives using the standard YCSB benchmark. The throughput obtained by our best Byzantine tolerant design was only 2-2.5 times lower than plain Cassandra while write and read latencies are only a 2-3X and 2-4X higher, respectively, than in the benign system.

Performance wise, the two most significant design decisions are the specific cryptographic signatures and resolving all conflicts during reads only. Our novel design of sending a vector of MAC tags, signed by itself with the symmetric key of the client and target node, plus the ECDSA public key signature, means that the *usual path* involves no public key verifications and only one elliptic curve signature, but no costly RSA signatures.

References

1 Riak. http://basho.com/products/riak-kv/.

2 Marcos K Aguilera and Ram Swaminathan. Remote storage with byzantine servers. In *Proc. of ACM SPAA*, pages 280–289, 2009.

3 Yair Amir, Brian Coan, Jonathan Kirsch, and John Lane. Prime: Byzantine replication under attack. *IEEE Trans. on Dependable and Secure Computing*, 8(4):564–577, 2011.

4 Leonardo Aniello, Silvia Bonomi, Marta Breno, and Roberto Baldoni. Assessing Data Availability of Cassandra in the Presence of non-accurate Membership. In *Proc. of the 2nd ACM International Workshop on Dependability Issues in Cloud Computing*, 2013.

5 Apache. Cassandra. http://cassandra.apache.org/.

6 Roberto Baldoni, Marco Platania, Leonardo Querzoni, and Sirio Scipioni. A peer-to-peer filter-based algorithm for internal clock synchronization in presence of corrupted processes. In *Proc. of the IEEE PRDC*, pages 64–72, 2008.

7 Alysson Bessani, João Sousa, and Eduardo EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *Proc. of the Annual IEEE/IFIP DSN*, pages 355–362, 2014.

8 Ken Birman and Roy Friedman. Trading Consistency for Availability in Distributed Systems. Technical Report TR96-1579, Cornell University, 1996.

9 Edward Bortnikov, Maxim Gurevich, Idit Keidar, Gabriel Kliot, and Alexander Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 53(13):2340–2359, 2009.

10 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*, pages 1–17. Springer, 2014.

11 Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, 2002.

12 Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM TOCS*, 26(2), 2008.

13 Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *22nd ACM SOSP*, pages 277–290, 2009.

14 Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. of the ACM Symposium on Cloud Computing*, pages 143–154, 2010.

15 Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical Hardening of Crash-tolerant Systems. In *Proc. of the USENIX Annual Technical Conference*, ATC, pages 41–41, 2012.

16 Inc DataStax. Apache Cassandra 2.2. http://docs.datastax.com/en/cassandra/2.2/.

17 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM OSR*, 41(6):205–220, 2007.

18 John R Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

19 Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *J. ACM*, 35(2):288–323, 1988.

20 D Eastlake and Tony Hansen. US secure hash algorithms (SHA and HMAC-SHA). Technical report, RFC 4634, July, 2006.

21 Christof Fetzer and Flaviu Cristian. Integrating external and internal clock synchronization. *Real-Time Systems*, 12(2):123–171, 1997.

22 Roy Friedman and Roni Licher. Hardening cassandra against byzantine failures. *CoRR*, abs/1610.02885, 2016. URL: http://arxiv.org/abs/1610.02885.

23 Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proc. of the 15th Usenix Conference on File and Storage Technologies*, FAST, pages 149–165, 2017.

24 Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proc. of the 5th ACM European Conference on Computer Systems*, pages 363–376, 2010.

25 Maurice Peter Herlihy. Replication methods for abstract data types. Technical report, DTIC Document, 1984.

26 Håvard D Johansen, Robbert Van Renesse, Ymir Vigfusson, and Dag Johansen. Fireflies: A secure and scalable membership and gossip service. *ACM TOCS*, 33(2), 2015.

27 Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

28 Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS OSR*, 44(2):35–40, 2010.

29 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

30 Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

31 Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *Distributed Computing*, pages 487–489. Springer, 2005.

32 Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. *ACM TOCS*, 29(4), 2011.

33 Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

34 David Mills, Jim Martin, Jack Burbank, and William Kasch. Network time protocol version 4: Protocol and algorithms specification. *IETF RFC5905, June*, 2010.

35 F. Pedone, N. Schiper, and J. E. Armendáriz-Iñigo. Byzantine Fault-Tolerant Deferred Update Replication. In *Proc. of the 5th Latin-American Symposium on Dependable Computing (LADC)*, pages 7–16, 2011.

36 Karin Petersen, Mike Spreitzer, Douglas B. Terry, Marvin Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In Michel Banâtre, Henry M. Levy, and William M. Waite, editors, *Proceedings of the Sixteenth ACM Symposium on Operating System Principles, SOSP 1997, St. Malo, France, October 5-8, 1997*, pages 288–301. ACM, 1997. `doi:10.1145/268998.266711`.

37 Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

38 Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*, 2006.

39 Atul Singh, Pedro Fonseca, Petr Kuznetsov, Rodrigo Rodrigues, Petros Maniatis, et al. Zeno: Eventually consistent byzantine-fault tolerance. In *NSDI*, pages 169–184, 2009.

40 Emil Sit and Robert Morris. Security considerations for peer-to-peer distributed hash tables. In *Peer-to-Peer Systems*, pages 261–269. Springer, 2002.

41 Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Middleware*, pages 55–70. Springer, 1998.

A Pseudocode

The pseudocode of the basic hardened write protocol appears in Figure 8 while the basic hardened read protocol is in Figure 9.

```
 1: function NodeWrite($k, v, ts, clientSign, clientID$)
 2:     if clientSign is valid then
 3:         $nodeSign \leftarrow ComputeSignature(clientSign)$
 4:                                              ▷ The client signature covers a fresh ts
 5:         Store locally $< k, v, ts, clientSign, clientID >$
 6:         return $nodeSign$                    ▷ A verifiable acknowledgment
 7:     end if
 8: end function
 9:
10: function ProxyWrite($k, v, ts, clientSign, clientID$)
11:     for each node $n$ that is responsible for $k$ do        ▷ N nodes
12:         Send the write request to $n$
13:     end for
14:     Wait for $2f + 1$ verified acknowledgements OR timeout
15:                                   ▷ Verified in the manner of correct node signature
16:     return responses
17: end function
18:
19: function ClientWrite($k, v$)
20:     $ts \leftarrow$ Current timestamp          ▷ From a secure synchronized clock
21:     $clientSign \leftarrow$ ComputeSignature($k$ || $v$ || $ts$)
22:     $p \leftarrow$ Some random system node
23:     Send write with <$k$, $v$, $ts$, $clientSign$, $clientID$> to $p$
24:     Wait for acknowledgments OR timeout
25:     if $|validResponses| \geq 2f + 1$ then
26:         return Success
27:     end if
28:     if $p = \perp$ OR $contactedNodes > f$ then
29:         return Failure
30:     end if
31:     goto line 22                              ▷ Use another node as proxy
32: end function
```

Figure 8 Our hardened write algorithm. ClientWrite is invoked on the client for each write. ProxyWrite is invoked on the proxy node by the client. NodeWrite is invoked on a node that has the responsibility to store the value. *Store locally* appends the write to an append log without any read. When k is queried, the latest store (by timestamp) is retrieved.

B Additional Attack Vectors

B.1 Column Families vs. Key-Value semantics

The algorithms described so far reflect only key-value semantics. Yet, we also supports Cassandra's column family semantics. In the latter, a client signs each column separately, producing a number of signatures that is equivalent to the number of non-key columns. This is needed for reconciling partial columns writes correctly according to Cassandra's semantics. For example, consider a scheme with two non-key columns A and B. One node can hold an updated version of A and a stale version of B while another node might hold the opposite state. A correct read should return one row containing the latest columns for both.

Nodes acknowledgments can still include only a single signature covering all columns. This is because the purpose of signatures here is to acknowledge the operation.

B.2 Membership View

The membership implementation of Cassandra is not Byzantine proof as faulty nodes can temper other's views by sending false data [4]. In addition, Byzantine seed nodes can partition the system into multiple subsystems that do not know about each other. This is by exposing different sets of nodes to different nodes.

To overcome this, each node's installation should be signed by the trusted system administrator with a logical timestamp. The logical timestamp enables nodes to verify they are using an updated configuration. Each node must contact at least $f + 1$ seed nodes in order to get at least one correct view. This requires the system administrator to pre-configure

```
 1: function NodeRead($k, clientTs$)
 2:     if $k$ is stored in the node then
 3:         $< v, ts, clientSign, clientID >\leftarrow$ Newest (timestamp manner) stored timestamp, value and client
    signature with $k$
 4:     else
 5:         $clientSign \leftarrow EMPTY$
 6:     end if
 7:     $nodeSign \leftarrow ComputeSignature(k||h(v)||clientSign||clientTs)$
 8:     if isDigestQuery then
 9:         return $< h(v), ts, clientSign, clientID, nodeSign >$
10:                                                              ▷ The hash is matched in the proxy
11:     else
12:         return $< v, ts, clientSign, clientID, nodeSign >$
13:     end if
14: end function
15:
16: function ProxyRead($k, clientTs$)
17:     $targetEndpoints \leftarrow allRelevantNodes$ for $k$ OR a subset of $2f+1$ fastest relevant nodes ▷ Optimization
18:     $dataEndpoind \leftarrow$ One node from $targetEndpoints$
19:     Send read data to $dataEndpoind$
20:     Send read digest to $targetEndpoints \setminus \{dataEndpoind\}$
21:     Wait for $2f+1$ verified responses OR timeout
22:     if timeout AND all nodes were targeted in line 17 then
23:         return $\perp$
24:     end if
25:     if got data response AND all digests match then
26:         return $< v, nodesSign >$
27:     end if
28:     Send read data to all nodes in $allRelevantNodes$                                ▷ N nodes
29:     Wait for $2f+1$ verified responses OR timeout
30:     if timeout then
31:         return $\perp$
32:     end if
33:     $resolved \leftarrow$ Latest $v$ in $responses$ that is clientSign verified.
34:     Write-back $resolved$ to $allRelevantNodes$
35:                                                  ▷ except those that are known to be updated
36:     Wait to have knowledge about $2f+1$ verified updated nodes OR timeout ▷ Returned updated data or a
    write back ACK
37:     if timeout then
38:         return $\perp$
39:     end if
40:     return $< resolved, nodesSigns, allNodesValues >$
41: end function
42:
43: function ClientRead($k$)
44:     $clientTs \leftarrow$ Current timestamp                                          ▷ Fresh timestamp
45:     $p \leftarrow$ Some random system node
46:     Send read request with $< k, clientTs >$ to $p$
47:     Wait for responses OR timeout
48:     if $|validNodesSign| \geq 2f+1$ then
49:         ▷ If write-back is observed, the resolved row is verified with the original read answers to ensure it was
    required
50:         return data
51:     end if
52:     $p \leftarrow$ Random node that was not used in this read as a proxy
53:     if $p = \perp$ OR $contactedNodes > f$ then
54:         return Failure
55:     end if
56:     goto line 46
57: end function
```

Figure 9 Our hardened read algorithm. ClientRead is invoked by the client for each read. ProxyRead is invoked on the proxy by the client. NodeRead is invoked on a node that is responsible to store the value.

manually the first $f+1$ nodes view as they cannot trust the rest. Let us emphasize that Byzantine seeds cannot forge false configurations. Rather, they can only hide configurations by not publishing them.

Here, we adopt the work on BFT push-pull gossip by Aniello et al. [4]. Their solution solves the dissemination issues by using signatures on the gossiped data.

B.3 Synchronized Clock

In plain Cassandra, as well as in our case, each write includes a wall-clock timestamp used to order the writes. With this, strong consistency cannot be promised unless local clocks are perfectly synchronized. For example, consider two clients that suffer from a clock skew of Δ. If both clients write to the same object in a period that is shorter than Δ, the later write might be attached with a smaller timestamp, i.e., the older write wins.

In a benign environment, when ensuring a very low clock skew, for most applications, these writes can be considered as parallel writes so any ordering of them is correct. For time synchronization, Cassandra requires the administrator to provide an external solution such as NTP. In our work, we follow this guideline using the latest version of NTP that can tolerate Byzantine faults when ensuring the usage of SSL and authentication measures [6, 34]. We configure this service so that all servers could use it as is and clients would be able only to query it, without affecting the time.

Alternatively, one could use external clocks such as GPS clocks, atomic clocks or equivalents [21], assuming Byzantine nodes can neither control them nor the interaction with them. Finally, Cassandra nodes can ignore writes with timestamps that are too far into the future to be the result of a normal clock's skew.

B.4 Other Network Attacks

Cassandra might suffer known overlay networks attacks, such as *Sybil attacks* [18] and *Eclipse attacks* [38]. In a Sybil attack, attackers create multiple false entities. In Cassandra, multiple node ids may lead to the same node, thereby fooling a client into storing its data only on a single Byzantine replica. As suggested in [18], here we rely on a trusted system administrator to be the sole entity for approving new entities that can be verified using PKI.

In an Eclipse attack, attackers divert requests towards malicious entities. E.g., a proxy might try to target only Byzantine replicas. To overcome this, clients request verifiable acknowledgments and count the number of correct repliers. If a proxy fails to provide these, alternative proxies are contacted until enough correct nodes have been contacted. Additionally, Section 5.3.1 explains how we handle a proxy that diverts requests to irrelevant nodes.

C Additional Performance Issues

C.1 Stalling Proxy

In the stalling proxy performance attack, following a correct execution of an operation, the proxy waits most of the timeout duration before supplying the client with the response. Consequently, the system's performance might decrease dramatically. Since the attack effects vary depending on the timeout configuration, the attack can be mitigated by lowering the timeout as low as possible. On the contrary, a tight timeout might fail correct requests during congestion times. The right optimization of timeouts relies on several deployment factors, e.g., the application requirements, the connection path of the client to system, the network topology of the nodes and more. Therefore, we have no definitive insights when facing this case. Finally, we would like to point out that the client can be configured to contact the fastest responding nodes first and thus reduce the effect of this attack.