Integrita: Protecting View-Consistency in Online Social Network with Federated Servers

Sanaz Taheri-Boshrooyeh, Alptekin Küpçü, Öznur Özkasap
Department of Computer Engineering, Koç University, İstanbul, Turkey
{staheri14, akupcu, oozkasap}@ku.edu.tr

Abstract—Current designs of Online Social Networks (OSN) deploy a centralized architecture where a central OSN provider handles all the users' read and write requests over the shared data (e.g., Facebook wall or a group page). The historical incidents demonstrate that such centralization is leveraged for censorship and violating view-consistency; a corrupted provider deliberately displays different views of the shared data to different users. Integrita provides a data-sharing mechanism that protects view-consistency by replacing the centralized architecture with the federated-server model consisting of N providers, $N-1$ of which can be fully malicious and colluding. The state of the shared data is modeled by an append-only data structure, stored at the server-side, which contains the history of all the operations performed by the users. The consistency of the users' views towards the shared data depends on their accessibility to the intact log of operations. Integrita guarantees that the servers cannot manipulate the log without being detected by the users. Unlike the state-of-the-art, Integrita accomplishes this neither by using storage-inefficient data replication nor by requiring users to exchange their views. Every user, without relying on the presence of other users, can verify whether his operation has been added to the log and is visible to the rest of the users. We introduce and achieve a new level of view-consistency, named q-detectable consistency, where any inconsistency between users' views cannot remain undetected for more than q operations, where q is a function of the number of the servers. This level of consistency is stronger than what centralized counterparts offer. Also, our proposal reduces the storage overhead imposed by replication-based solutions by the multiplicative factor of $\frac{1}{N}$. Furthermore, the application of Integrita is not limited to OSNs and can be integrated into any log-based systems, e.g., version control systems.

Index Terms—View consistency, q-Detectable Consistency, Strong Consistency, Malicious Adversary, Collaborative Data Sharing, History Integrity, Log-based System.

1 Introduction

Motivation: Online Social Networks (OSNs) enable various methods of data sharing, e.g., via users' walls or social groups. Using a personal wall, a user may share her personal information (e.g., thoughts, images, and videos) with her social connections, e.g., friends or followers. The content of the wall can be updated by the user or her connections, e.g., by posting a message on the wall and commenting. A similar data-sharing paradigm appears in the context of social networking groups like Facebook groups where the group pages serve as a shared board on which members can post their content.

In the current designs of OSNs with a central provider, users' read and write operations over the shared data go through the central OSN server, who is supposed to be trusted and serves users as expected. However, no technique is deployed to enforce such trustworthiness of the OSN server. A corrupted server may add arbitrary content to the shared data and make users accept them as authentic, or hide some content from some users. As a historical example, in 2012, several bloggers claimed that Sina Weibo, a Chinese OSN, aimed to practice censorship by serving different views of the walls to their followers via hiding some of their posts [22]. Given such historical incidents, it is vital to tackle *view-consistency* of the shared data with a practical solution rather than trusting the service provider.

Problem Statement: To formalize the problem of *view-consistency*, we use the term shared *object* to indicate a collaborative data-sharing environment (such as a Facebook-like wall or a group-page) with a defined set of users with read/write access. Users can modify the *object* by inserting or deleting a post. The current content of each *object* is the result of a sequence of insertion and deletion of posts. For example, the content of a shared object after the following operations $\Delta = \{insert(post_1), insert(post_2), delete(post_1), insert(post_3)\}$ would be $\{post_1, post_3\}$. The view of a user towards an *object*, at any point in time, is defined as the sequence of operations seen by that user, e.g., $view = \{insert(post_1), insert(post_2)\}$ is the view of a user from Δ who has not yet fetched the last two operations. We assume that no concurrent operations/edits from any two users happen on the shared object; operations performed by the users are collectively sequential. This assumption has been similarly sought by Frientegrity [21], which addresses view-consistency in a centralized OSN. Users are said to have a consistent view of the shared *object* if the following two conditions are met: First, operations served to the users are only generated by the authorized users. This can be immediately addressed by deploying digital signatures. The second criteria regards the *history integrity* of an *object*, which is less recognized and studied in the literature. That is, all the users get to see an identical and intact log of operations, i.e., no operation is dropped or misplaced. For instance, the following sequence of operations $\Delta' = \{insert(post_1), delete(post_1), insert(post_3)\}$ is inconsistent w.r.t. Δ since $insert(post_2)$ is dropped from the original sequence.

Related Work: The view-consistency problem is addressed in the literature by two types of solutions: communication-based solutions that are sought by the centralized architectures, and replication-based solutions deployed by the distributed designs. We elaborate on each solution type next.

In a centralized architecture with non-communicating users, the best achievable level of view-consistency is fork-consistency [55], first defined by [41]. The fork-consistency is a weaker form of view-consistency in which a corrupted provider can split users into disjoint sets (fork them) and serve each set with a distinct view. However, the provider is forced to serve each set with a consistent view of the operations performed by the users of the same set. Identification of the forked views can only happen through users' communication. That is, the users must regularly communicate their views of the *object* (e.g., a wall) with all the other authorized users (e.g., friends) to catch any inconsistency. This approach would not be practical considering that a user of an OSN like Facebook has 338 friends on the average[1]. Hence, each user needs to communicate with almost 338*338= 114244 other users to monitor the view consistency of her wall and her friends' walls. Addressing view consistency using communication-based solution is sought in the context of secure OSNs [21], [23], and cloud computing [11], [10], [39], [8].

The replication-based solutions are deployed in peer-to-peer distributed OSNs [43], Authenticated Data Structures (ADS) [25], [26], [45], [52], [27], [44], Byzantine fault-tolerant protocols [37], [13], cloud storage [30], [49], [1], [53], [57] as well as certificate transparency [36], [35], [18]. The idea is to designate multiple entities for the storage of the *object* and let all the read and write requests go through all of the storage providers. In specific, the shared *object* (or some authenticated-metadata associated with it) must be replicated on N entities considering $N-1$ of them may act maliciously. Having only one honest repository suffices to always retrieve the intact content of the *object*. Replication-based solutions are not efficient concerning the storage overhead, where N copies of the *object* must be stored in the OSN.

Integrita: In Integrita, we achieve the best of both aforementioned solutions: a method to achieve view-consistency which is replication-free as well as communication-free, i.e., users do not have to directly exchange their views. Integrita utilizes N federated servers (governed by distinct authorities) who might be malicious/Byzantine; act arbitrarily, collude, and compromise the view-consistency by dropping, tampering with, and forging operations. Nonetheless, our approach guarantees that as long as one server is non-colluding, the view consistency is preserved. We assume that users shall act honestly and tend to achieve a consistent view. A similar assumption was sought in prior studies [15]. Note that the sole objective of Integrita is to address view-consistency but not data privacy. One can address privacy using well-practiced techniques like encryption [4], [34]. In summary, Integrita provides the following features:

- **q-Detectable-Consistency:** Integrita introduces a new level of view-consistency called *q-Detectable-Consistency* where the views of the users toward the *object* cannot diverge for more than a sequence of q operations without being detected. That is, if a user performs an operation, e.g., inserts a post, servers either honestly serve her post to the rest of users, or their misbehavior gets caught by the post owner within the next q operations. The value of q depends on the number of servers and the number of operations applied on the shared *object*. Section 6.2 covers a thorough analysis of this relation. The formal definition and proof of q-detectable consistency are presented in Section 8. We also discuss that after a certain number of operations on the *object*, the q-detectable consistency level converts to strong consistency (i.e., misbehavior is immediately detectable).
- **Communication-free:** Users do not have to exchange their views of the shared *object* to uncover inconsistencies. Instead, each user, independent of other users, can verify whether her operation is applied on the *object* (hence visible to all the other users) or catch the server's misbehavior.
- **Replication-free:** The shared *object* is not replicated over the servers, instead, the *object* is partitioned into disjoint sets such that each resides at one server[2]. Our numerical analysis asserts that using Integrita, an OSN like Facebook with 2.41 billion monthly active users [3] saves up to 2344 Terabytes of storage per year compared to a replication-based approach deploying 20 servers. In our replication-free solution, we trade strong consistency with the q-detectable consistency.
- **Efficient verification:** In Integrita, each read and write request is associated with a proof of correctness. Due to the lack of a central entity for the generation of the proof, users have to individually contact servers and construct the proof. Despite this, the overhead of users and servers w.r.t. data transmission, as well as the communication and computation complexity is identical to the centralized fork-consistent counterparts [22], [23] (while Integrita enforces a higher level of consistency).
- **Cross-server communication-free**: Servers do not need to communicate or to coordinate to resolve the users' read and write requests. Instead, all the communication happens solely between a user and the servers.

2 Related Work

The problem of view consistency in a collaborative data-sharing environment has been investigated in centralized

1. https://www.brandwatch.com/blog/facebook-statistics/

2. Note that each of the N storage providers is modeled as a data-center that would take care of a portion of the data assigned to it. While Integrita does not depend on data replication to achieve view-consistency, this does not contradict with the replication of data for the sake of *availability*. Namely, each data-center shall deploy its replication mechanism to maintain the availability of the data assigned to it. However, due to Integrita, the amount of data assigned to each data-center is $\frac{1}{N}$ of the data that would be otherwise assigned by using a replication-based solution.

3. https://www.statista.com/statistics/264810/number-of-monthly-active-Facebook-users-worldwide/

OSNs, peer-to-peer (p2p) OSNs, cloud storage platforms, Byzantine fault-tolerant protocols, and authenticated data structures. In the following, we elaborate on each one of these areas and shed light on their shortcomings compared to Integrita. Later in this section, we also discuss how to integrate Integrita in each one of the related studies.

Centralized OSN: Among the centralized OSNs, Frientegrity [22] and SPORC [23] address the view-consistency. However, their solutions do not eliminate the possibility of having forked views, and in principle, they only guarantee a fork-consistent level that is weaker than q-detectable consistency. In a fork-consistent system, while a corrupted provider is able to fork the users into disjoint sets, he is forced to serve each set with a consistent view of the operations performed by the users of the same set. This is enabled since the users embed their views of the object history in each post that they insert. Thus, as soon as the server forks the view of two users, he cannot show their operations to each other without risking detection. Users can detect the inconsistency of their views by exchanging them out of the band. The main shortcoming of fork-consistent systems is that the server's equivocation remains undetected till the users happen to contact out of the band. Thus, to ensure view-consistency, users must regularly communicate their views of the shared object. This approach would not be practical. For example, consider Facebook walls as a shared object to which both the user and her friends have access. Each Facebook user has 338 friends on the average[4]. Hence, each user needs to monitor the view consistency of 338 shared objects (her wall and her friends') where each wall is shared among 338 users (friends of each wall owner), thus on aggregate, she has to communicate with almost $338 * 338 = 114244$ other users to monitor the view consistency of *her wall* and *her friends' walls*.

Peer-to-Peer OSNs: In p2p OSNs [50], [28], [38], [9], [51], [54], [43], [48], [42], [54], there is no central server running the system; instead, the individual users, called peers, contribute a part of their computation and storage power to the system. Social networking services are enabled in a distributed manner relying on shared resources. As such, the storage of users' data is also distributed among the existing peers. The view consistency in p2p OSNs is usually addressed through replication or by leveraging users' trust. In the latter case, the object owner (e.g., the owner of a wall) is responsible to store and serve the content of her wall on her own or replicates it on some trusted peers like her friends. Subsequently, the view-consistency is guaranteed due to the trustworthiness of storage peers [50], [28], [38], [9], [51], [54]. However, if the storage responsibility is spread over the p2p network and the storing peers are untrusted, then view-consistency is met through replication [43]. In particular, suppose f is the number of potential dishonest peers. The object (or some units of the object like a post) should be replicated on $f + 1$ peers to ensure that at least one honest peer is among the replicas. Each requester reads each post from all the $f + 1$ replicas and identifies the latest content (e.g., using a version number). However, such a solution results in storage overhead and communication complexity which grow linearly by f. Note that other studies in the context of p2p OSNs also utilize replication but for the sake of data availability [48], [42], [54]. Namely, the storing nodes are supposed to be trusted and always serve the intact contents when available.

4. https://www.brandwatch.com/blog/facebook-statistics/

Byzantine Fault-Tolerant Protocols: In BFT protocols, a service is to be given to a set of clients while the execution of the service concerning the sequence of requests appears identical to all the clients (and this sequence preserves the temporal order of non-concurrent operations). Enabling consistency, BFT protocols also seek a replication-based solution [37], [13] where they deploy several servers each keeping a replica of the state of the intended service. Byzantine fault-tolerant systems behave correctly when no more than f out of $3f + 1$ replicas fail [37].

Cloud Storage Platforms: Similar to the centralized OSNs, the best level of view-consistency in the context of cloud storage is fork-consistency [11], [10], [39], [8], [55], due to the presence of a single corrupted service provider and non-communicating users. Addressing the fork issue, cloud storage platforms utilize replication over multiple servers [29], [37]. In some other studies replication is utilized for the sake of data availability [30], [49], [1], [53], [57] and the view-consistency is attempted in the presence of concurrent operations.

Authenticated Data Structures: In the Authenticated Data Structures (ADS), a data owner outsources her data to multiple untrusted repositories. The outsourced data is modeled by a data structure that enables performing queries on the data in a verifiable and authenticated manner. Repositories, on behalf of the data owner, are responsible to answer queries of users on the data structure and hand them with a proof of the validity of the answer. The same data structure is replicated over all the repositories and repositories need to keep themselves updated with the data owner in the case of update [25], [26], [45], [52], [44]. As such, one can assume view consistency of ADSs is guaranteed through replication, which is not storage efficient.

Certificate Transparency: In public-key cryptography, parties need to have access to each other's authentic public keys. To accomplish this, a centralized certificate authority (CA) is designated through which users insert and retrieve each other's public keys. To ensure that CAs hand on authentic public keys, Certificate transparency (CT) [36], [35], [18] techniques are developed. CT relies on a set of public, untrusted, append-only log servers that collect the certificates issued by CAs and serve the certificates to the users in a verifiable manner [17]. Moreover, the log servers are tracked by auditors/monitors. Auditors collect external information as well as constantly query the log servers to catch inconsistency. In the nutshell, protecting view-consistency in CT is done through replication (i.e., multiple log servers) as well as constant auditing. Replication is not storage efficient. Also, in collaborative data sharing environments like Facebook groups, making users to constantly audit the servers is neither desirable nor effective. Essentially, the servers know the set of authorized members, hence they still can answer queries deliberately and partition users' views (in contrast to the CT systems where auditors can be arbitrary entities). Some other CT proposals suggest users exchange their views of the log through gossiping protocols [12], [32], [46]. This immediately violates the assumption of non-communicating users that we set earlier.

Adoption of Integrita: Integrita can be integrated into any of the studies listed above as long as the following conditions are met in the given context. We refer to this list as *Integrita Adoption Requirements* or *IAR* for short.

1) There is a shared object to be accessed and modified by a known set of authorized users.
2) All the authorized users are acting honestly and willingly to achieve view consistency.
3) The shared object has an append-only nature.
4) Updates on the object come sequentially but not concurrently.
5) The system should deploy the federated-server architecture where the service is run by the collaboration of multiple independent service providers.
6) The number of deployed servers should be fixed and known beforehand. This is required to achieve q-detectable consistency and to have a well-defined transition point to strong consistency (more details are provided in section 6.2.4).

Many studies listed above exhibit the aforementioned features yet sometimes seek additional objectives which are not covered by Integrita. Thus, we emphasize that the adoption of Integrita should take place only to achieve view consistency conditioned on IAR.

Applications that address view consistency by relying on replication (i.e., deploying replicated storage providers) can alternatively benefit from Integrita and enjoy its extended features[5]. Examples are P2P OSNs [43] as well as OSN services like [5], [48], [51][6]. Centralized OSNs like [22] and cloud storage providers like [23], [29], [37] can benefit from Integrita conditioned on utilizing multiple independent servers. BFT protocols like [37], [13] and cloud storage services of [11], [10], [39], [8], [55], [29], [37] also comply with IAR, though, they additionally deal with the linearization of concurrent operations, which is out of the scope of Integrita. The view consistency in CT can be resolved using Integrita noting that the number of log servers should be preset in order to achieve q-detectable consistency. Also, note that CT tackles with the problem of a fraudulent certificate, where a malicious user (e.g., CA) inserts a certificate to replace an existing one. Treating such issues are out of the scope of Integrita.

3 System Model

Entities: Integrita is comprised of N *servers* denoted by S_1,...,S_N (each operated by a distinct authority so that at least one of them does not collude with the others) and a set of users $U_1, ..., U_T$. The shared *object* resides at the server-side. For the sake of simplicity, users are assumed to have identical read/write access, though, more fine-grained access control can be enforced using the technique proposed by [22].

5. Note that sometimes replication is used for the sake of data availability, e.g., in the case of ADS, the data is replicated over multiple servers residing in multiple geographical locations to reduce the users access delay and allow scalability [40].

6. These are already run on the federated-server architecture hence the adoption of Integrita is straightforward.

Security Goals: The security goal of Integrita is to achieve q-detectable consistency; that is, any inconsistency between users' views cannot remain undetected for more than q operations. A formal definition is provided in Section 8.

Adversarial Model. We assume that $N-1$ servers are colluding and may act maliciously by showing users a different subset of operations. On the other side, users are honest and tend to achieve a consistent view of the shared *object*. The communication channels are authenticated but not necessarily secure; hence subject to eavesdropping. Note that data confidentiality is out of the scope of Integrita and can be addressed by well-studied solutions like encryption.

4 Definitions and Preliminaries

Negligible: A function f is called negligible if for all positive polynomials p, there exists a constant C such that for every value $c > C$ it holds that $f(c) < \frac{1}{p(c)}$.

Signature Scheme A signature scheme [24] Sig consists of three algorithms; key generation, sign and verify denoted by $Sig = (Gen, Sign, Vrfy)$. A pair of keys (sk, vk) is generated via $SGen$ where sk is the signature key and vk is the verification key. The signer signs a message m using sk by computing $\eta = Sign_{sk}(m)$. Given the verification key vk, a receiver of signature runs $Vrfy_{vk}(\eta, m)$ to verify.

A signature scheme $Sig = (Gen, Sign, Vrfy)$ is said to be existentially unforgeable under adaptive chosen message attack [24], [33] if $\forall$ probabilistic polynomial time adversaries A, there exists a negligible function $nelg(.)$ s.t.

$$\begin{aligned} Pr[(sk, vk) \leftarrow Gen(1^{\lambda}); (m, \sigma) \leftarrow A^{Sign_{sk}(.)}(vk) \\ \text{s.t. } m \notin Q \textit{ and } Vrfy_{vk}(m, \sigma) = accept] = negl(\lambda). \end{aligned} \tag{1}$$

$A^{Sign_{sk}(.)}$ indicates that adversary has oracle access to the signing algorithm. Q indicates the set of adversary's queries to the signature oracle. λ is the security parameter.

Collision-Resistant Hash Function: A hash function family $\Pi = (Gen, H)$ is collision-resistant if for all probabilistic polynomial time adversary A, there exists a negligible function $negl(\lambda)$ for which the following holds [16].

$$\begin{aligned} Pr[s \leftarrow Gen(1^{\lambda}); A(s) = x, x' | \\ H_s(x) = H_s(x') \text{ AND } x \neq x'] \leq negl(\lambda) \end{aligned} \tag{2}$$

History Tree: A history tree [15] is an append-only tamper-evident data structure modeled by a variant of the Merkle hash tree. The leaves of the tree hold data items and the intermediate nodes and the root node store the hash of their children. In such a structure, the root essentially covers the entire content of the tree. New data items can freely be added as the leaf nodes to the right side of the tree (appending). For each newly added item, the value of intermediate nodes and the root shall be recalculated. A sample history tree consisting of 4 and 5 data items (leaves) are demonstrated in Figure 1 and Figure 2, respectively. We use the term of *tree digest* or τ to refer to the root of a history tree and we write τ_i to indicate version-i tree, i.e., the root of the history tree with i data items.

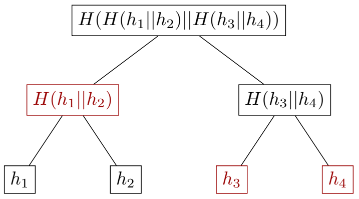
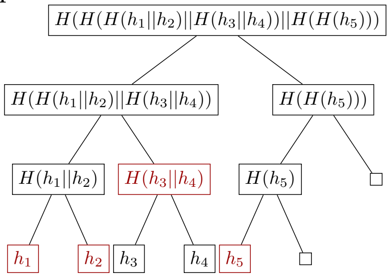


Fig. 1: A history tree with 4 leaves. h_is represent the data items. The colored parts represent the nodes required for the membership proof of h_3.



$H(H(H(h_1||h_2)||H(h_3||h_4))||H(H(h_5)))$
$H(H(h_1||h_2)||H(h_3||h_4))$
$H(H(h_5)))$
$H(h_1||h_2)$
$H(h_3||h_4)$
$H(h_5)$
h_1
h_2
h_3
h_4
h_5

Fig. 2: A history tree with 5 leaves. h_is represent the data items. The sub-trees with no data item are shown by $\Box$. The colored parts represent the nodes required for the incremental proof between the 2nd and 5th version the history tree.

The history tree exhibits the following properties (that are fundamental in efficiently preserving view consistency).

- Every tree digest τ_j uniquely defines a distinct ordered sequence of j data items. That is, the roots of history trees constructed from an identical set of data items but under different orders, e.g., $\{h_1, h_2, h_3\}$ and $\{h_2, h_1, h_3\}$, would be different. This is because the underlying hash function is collision-resistant.
- Proof of membership: Given a tree with j data items (and the root τ_j), the membership of a data item h_i at a certain position i (where $i <= j$) of the tree is efficiently verifiable in $O(log(j))$. The proof of membership includes the sequence of values stored at the siblings of the nodes (indicating whether it is a left or right sibling) on the path from the leaf node storing h_i to the root τ_j. Given the proof, one can recompute the root as τ'_j and compare against τ_j. For example, as shown in Figure 1, the membership proof of h_3 as the 3^{rd} data item of a history tree with tree digest τ_4 is h_3, h_4, $H(h_1||h_2)$. To verify the proof, one should reconstruct the tree digest TD'_4 recursively from the values included in the proof. If τ'_4 equals to τ_4, the membership proof is verified.
  Throughout the description of the paper, *MemVf* refers to the function defined below
  $$True/False = MemVf(\tau_j, \{(i, h_i), \cdots, (k, h_k)\}, proof)) \tag{3}$$
  MemVf takes a tree digest τ_j, a set of index and item pairs as $\{(i, h_i), \cdots, (k, h_k)\}$, and a membership proof $proof$. The function returns $True$ if the $proof$ can successfully verify the membership of $\{h_i, \cdots, h_k\}$ for the given positions $\{i, \cdots, k\}$ of the history tree with tree digest τ_j. We also write $\{h_i, \cdots, h_k\} \in \tau_j$ to indicate that there exists a $proof$ for which *MemVf* returns $True$.
- Incremental Proof: Given two tree digests τ_i and τ_j where $i < j$ one can check whether they share the same history regarding $h_1, \cdots, h_i$. The incremental proof between version-2 (τ_2) and version-5 (τ_5) is shown in Figure 2. Given the proof i.e., $h_1, h_2, H(h_3||h_4)$, and h_5, one can reconstruct τ'_2 and τ'_5 as below.
  $$\tau'_2 = H(h_1||h_2)$$
  $$\tau'_5 = H(H(H(h_1||h_2)||H(h_3||h_4))||H(H(h_5)))$$
  Note that for the calculation of τ'_2 only h_1 and h_2 are required, whereas for the τ'_5 all the nodes in the proof are used. If $\tau'_2 = \tau_2$ and $\tau'_5 = \tau_5$, then the consistency between τ_2 and τ_5 is successfully verified.
  Throughout the paper, we consider $IncrVf$ function as defined below [15]
  $$True/False = IncrVf(\tau_i, \tau_j, proof) \tag{4}$$
  It takes two tree digests τ_j and τ_j where $i < j$, and verifies whether $proof$ is a correct incremental proof between τ_i and τ_j or not. We write $\tau_i \implies \tau_j$ to indicate that there exists an incremental $proof$ for which $IncrVf(\tau_i, \tau_j, proof))$ returns $True$.

5 Integrita System Design

In the current section, we present our solution to address the view consistency issue in a collaborative data-sharing environment. Recall that the system contains a shared *object* whose content gets altered by a set of authorized non-communicating collaborators. The *object* resides at the servers side.

The shared *object* is comprised of smaller data units called operations. Each operation may contain an image, text, audio, etc. Users may add or remove operations. The sequence of users' activity (addition or deletion operations) is recorder in the activity log $\Delta = \{op_1, \cdots, op_j\}$ where each operation $op_{i\in[1,j]}$ indicates either an insertion operation or a deletion operation together with the operation's content. The activity log has an append-only nature,

Users have a consistent view of the shared object if they have access to the intact activity log. However, due to the servers' misbehavior, this may not be the case. We denote a user's view by δ_i which is a sequence of i operations, i.e., $\delta_i = \{op'_1, \cdots, op'_i\}$. A view δ_i is consistent with the activity log $\Delta = \{op_1, \cdots op_k\}$ (where $i <= k$) if it is a prefix of the activity log, i.e., $\Delta = \delta_i||\{op_{i+1}, \cdots, op_k\}$. This definition implies the following properties. A solution for the view-consistency must satisfy all these three properties and vice versa.

1) Authenticity: A consistent view contains operations that are made by authorized users.
2) Individual view-consistency: For a user with a consistent view, her past and current views should be also consistent, i.e., the past view is a prefix of the current view. To put it formally, for every two views δ_i and δ_j of the same user where $i < j$, we have $\delta_j = \delta_i||\delta_{i+1,j}$ where $\delta_{i+1,j}$ contains the last $j - i$ activities of the δ_j.
3) Cross-user view-consistency: For a user with a consistent view, her view at any point in time should share the same history with the view of every other authorized user. More precisely, consider δ_i^{Alice} and δ_j^{Bob} ($i < j$) are the views of Alice and Bob, two authorized users. It should hold that $\delta_j^{Bob} = \delta_i^{Alice}||\delta_{i+1,j}^{Bob}$ where $\delta_{i+1,j}^{Bob}$ holds the last $j - i$ operations of δ_j^{Bob}.

5.1 Solution Overview

We address the view-consistency problem by providing a solution for each of the aspects enumerated above, namely, authenticity, individual view-consistency, and cross-user view-consistency.

The authenticity of the log of activities is guaranteed through digital signatures where each operation shall be signed by the user who performs the operation. A similar approach is sought in [22], [21], [23], hence we skip the details of this part.

To protect individual view-consistency, we model the activity log by a history tree where the hashed operations constitute the leaves of the tree. Each user maintains the tree digest of the history tree as her local Status and keeps updating it to the latest version after each read and write request. The tree digest mirrors the content of the activity log in an abstract format. Using her local Status, each user monitors the consistency of her past and present view by verifying the result of her read and write requests using membership and incremental proofs.

Tackling cross-user view-consistency is beyond the simple deployment of a history tree. Prior studies utilizing history-tree have shown that users can still be forked into two disjoint groups and only be served by the operations of users in their own group in a consistent and verifiable manner [21], hence compromising cross-user view-consistency. In such a situation, unless users communicate and compare their views, the fork remains undetected [22]. Satisfying cross-user view-consistency efficiently, namely preventing and detecting forked views without relying on users' communication, is one of the core contributions of Integrita. There are two key methods to our solution for cross-user view-consistency.

1) Integrita replaces the central storage of the shared *object* (that is the major roadblock for satisfying cross-user view-consistency) by a federated server architecture that deploys multiple independent providers with conflicting interests (that is at least one of them will not collude with the rest) such as political parties of a country or the ISPs of multiple distinct countries. Similar setting has been similarly utilized in other security-concerned systems [31], [14], [2], [47], [6], [7]. The storage of the activity log, which is modeled by a history tree, is partitioned among the servers using an algorithmic approach offered by Integrita. All the nodes of the history tree, namely, tree digest, internal nodes, and leaves, are indexed (using a proposed indexing function) and grouped into disjoint sets, and each set is assigned to a distinct server. We remark that no replication takes place. The indexing function is public and users know where to locate a particular node of the history tree. To obtain a (membership/incremental) proof, users identify the nodes along a membership or incremental proof, then fetch them from the corresponding storage servers.
The distributed storage of nodes of the history tree is one of the keys to preventing malicious servers from forking users' views without being detected. The intuition is that when a fork happens, the users under each fork shall receive membership and incremental proofs that are different from the other fork. Recall that proofs in a history tree data structure are a subset of tree nodes. Thus, when a fork is created, users of each fork are handed with different (inconsistent) values of nodes along the proof paths. However, the storage of nodes is distributed among N servers, one of which is non-colluding. Once the users of the forked groups contact the non-colluding server (who is not involved in the malicious behavior of $N-1$ other servers), the server's response can only be consistent with the view of one of those forked groups but not both. Therefore, the proof for one of the groups fails and the inconsistency is revealed. Essentially, the $N-1$ malicious servers can fork users' views for the operations whose membership proofs reside at the malicious servers (but not the non-colluding server). However, our proposed indexing algorithm assures that for the nodes along the membership proof of every sequence of q operations, there will be at least one node that resides at the non-colluding server. Therefore, no fork can last for more than q consecutive operations.
2) The distribution of the storage of the activity log alone does not suffice to achieve a quantifiable and provable level of view-consistency. In fact, despite the presence of an honest server in every sequence of q operations, the dishonest servers, in particular circumstances which are not avoidable, can bypass the honest server and make the fork last a bit longer (more than q operations). They can do so, by exchanging operations, i.e., leaf nodes from one fork to another fork.
To mitigate this interchangeability mentioned above, we propose that each newly inserted operation (i.e., a leaf) as well as the associated root must be signed by the user who submits the operation. This way, each leaf is tied to the resultant tree. Hence, malicious servers have to borrow an operation and it's associated tree digest together from one fork to another. However, since the tree digests are representative of a history of the operations, they are not exchangeable across forks (since distinct forks do not share an identical history). By mitigating the interchangeability of operations, we can provably guarantee that no fork lasts for more than q consecutive operations.
Our solution enjoys another significant property due to which the value of q converges to 1 as the size of the activity log grows (an extensive analysis of this feature is supplied in Section 6.2). Having q equal to 1 indicates that no fork lasts for even 1 operation, thus, the system enters a strong level of consistency.

5.2 Distributed storage of the shared *object*

In Integrita, we propose a method to divide the storage of the shared *object* among N servers in a non-overlapping manner. We first present our new indexing mechanism for the nodes of a history tree. Then, we describe how to utilize those indices to distribute the storage of nodes among the servers.

Node Indexing. We define the insertion path of operation i to be the nodes of the history tree whose values get altered while inserting that operation into the tree. Figure

3 illustrates the insertion path of operations $1-4$, each with a different color. The index labels $I_{i,j}$ indicates the j^{th} node at level i. For a tree with M leaves, i ranges from 0 (indicating the leaves of the history tree) to $\lceil log(M) \rceil$ (indicating the root of the history tree). Since the history tree supports logarithmic path lengths from the root to the leaves, the insertion path of the i^{th} operation consists of $\lceil log(i) \rceil + 1$ nodes where the logarithm evaluation is in base 2. For example, the insertion path of op_1 is comprised of only one node $I_{1,1}$ ($\lceil log(1) \rceil + 1 = 1$) whereas the insertion path op_2 consists of two nodes $I_{1,2}$ and $I_{2,1}$ ($\lceil log(2) \rceil + 1 = 2$). The insertion paths of distinct operations may overlap, e.g., the insertion path of op_3, i.e., $\{I_{0,3}, I_{1,2}, I_{2,1}\}$, and the insertion path op_4, i.e., $\{I_{0,4}, I_{1,2}, I_{2,1}\}$, overlap in $I_{1,2}$ and $I_{2,1}$. However, the value of each of these nodes at the time of insertion of op_3 and op_4 are different, e.g., $I_{1,2}$, on the insertion path of op_3 contains $H(h_3 || \perp)$ whereas its value changes to $H(h_3||h_4)$ after the insertion of op_4.

Following the above intuition, in Integrita, we distinguish among different values of intermediate nodes and address them based on their location on the insertion path of each operation. That is, each node is addressed with a pair of integers (p, l) where l indicates the level of a node along the insertion path and p represents the operation number. We write $N_{p,l}$ to indicate the hash value of the node with address (p, l). Figure 4 demonstrates this new addressing semantic for operations $1-4$. The insertion paths of op_3 and op_4 are $\{N_{3,0}, N_{3,1}, N_{3,2}\}$ and $\{N_{4,0}, N_{4,1}, N_{4,2}\}$, respectively. By contrasting Figures 4 and 3, we see that in the new addressing mode, the node $I_{1,2}$ is given two separate index labels $N_{3,1}$ and $N_{4,1}$, corresponding to the values at the insertion of op_3 and op_4, respectively.

Following our new addressing mechanism, we define and distinguish three types of nodes of the history tree.

- **Tree digest**: The root of the tree after the insertion of each operation is called the *tree digest*. In Figure 4, nodes $N_{1,0}, N_{2,1}, N_{3,2}, N_{4,2}$ all represent the tree digests, which are the roots of the tree at the insertion time of op_1, op_2, op_3, and op_4, respectively. A node with the address (p, l) is a tree digest if Equation 5 below holds:
$$l = \lceil log(p) \rceil \tag{5}$$
- **Full node:** A full node is a node whose left and right sub-trees are full, i.e., insertion of further operations will not alter the value of a full node. In Figure 4, the nodes $N_{1,0}, N_{2,0}, N_{3,0}, N_{4,0}, N_{4,1}$ and $N_{4,2}$ are full. A node with the address (p, l) is a full node if Equation 6 below is met:
$$p \bmod 2^l = 0 \tag{6}$$
- **Temporary node:** Nodes whose left or right sub-trees are not full are called temporary nodes. We call them temporary since the insertion of further operations will change their hash values. For example, nodes $N_{3,2}$ and $N_{3,1}$ in Figure 4 are temporary as there is an empty node in their right sub-trees corresponding to $H(op_4)$. In general, $N_{p,l}$ is a temporary node if Equation 7 below holds:
$$p \bmod 2^l \neq 0 \tag{7}$$

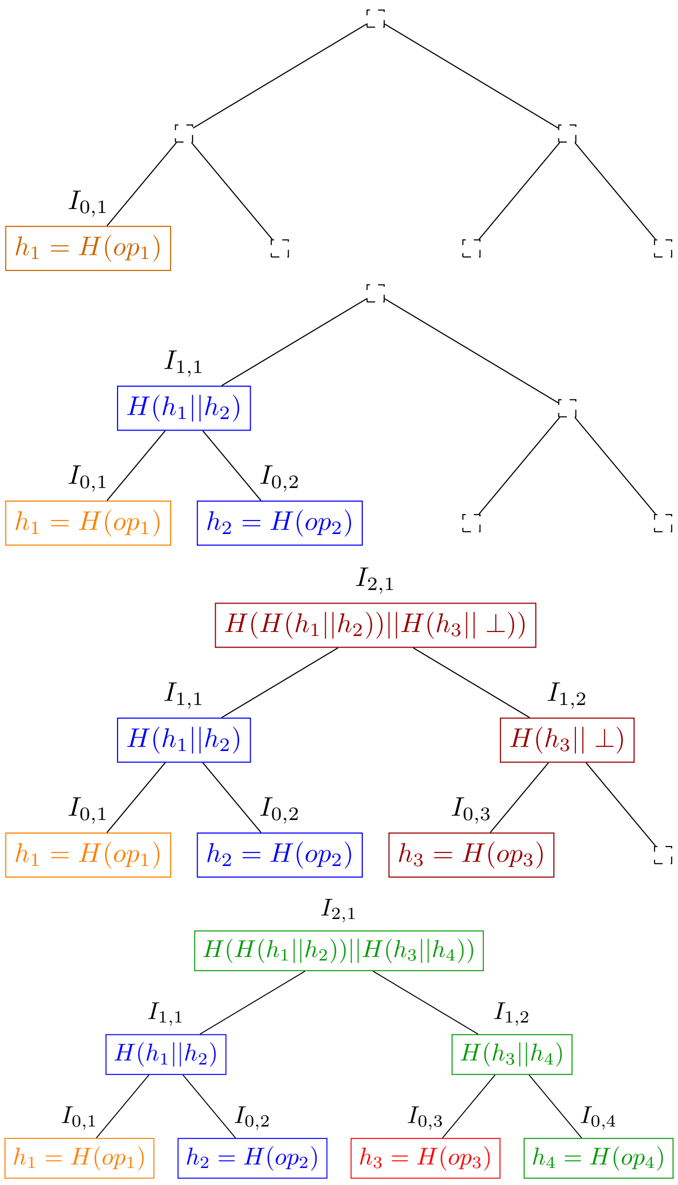


Fig. 3: Insertion paths of op_1, op_2, op_3, and op_4. Each insertion path is indicated by a distinct color.

Assignment of nodes to the servers. The index of the storage server of a node with the address (p, l) is determined by function F defined in Equation 8:
$$F(p, l) = [L(p, l) \bmod N] \tag{8}$$
where L is defined as in Equation 9 and N is the total number of servers. L can be seen as a deterministic labeling function that converts node addresses (p, l) to an integer value.
$$L(p, l) = 1 + l + \sum_{j=1}^{i-1} (\lceil log(j) \rceil + 1) \tag{9}$$
For example, in a system with 4 servers, the storage of $N_{1,0}, N_{3,1}, N_{4,2}, \ldots$ are given to the first server S_1. Nodes $N_{2,0}, N_{3,2}, \ldots$ are given to S_2, then S_3 gets to serve $N_{2,1}, N_{4,0} \ldots$, and S_4 stores $N_{3,0}, N_{4,1}, \ldots$.

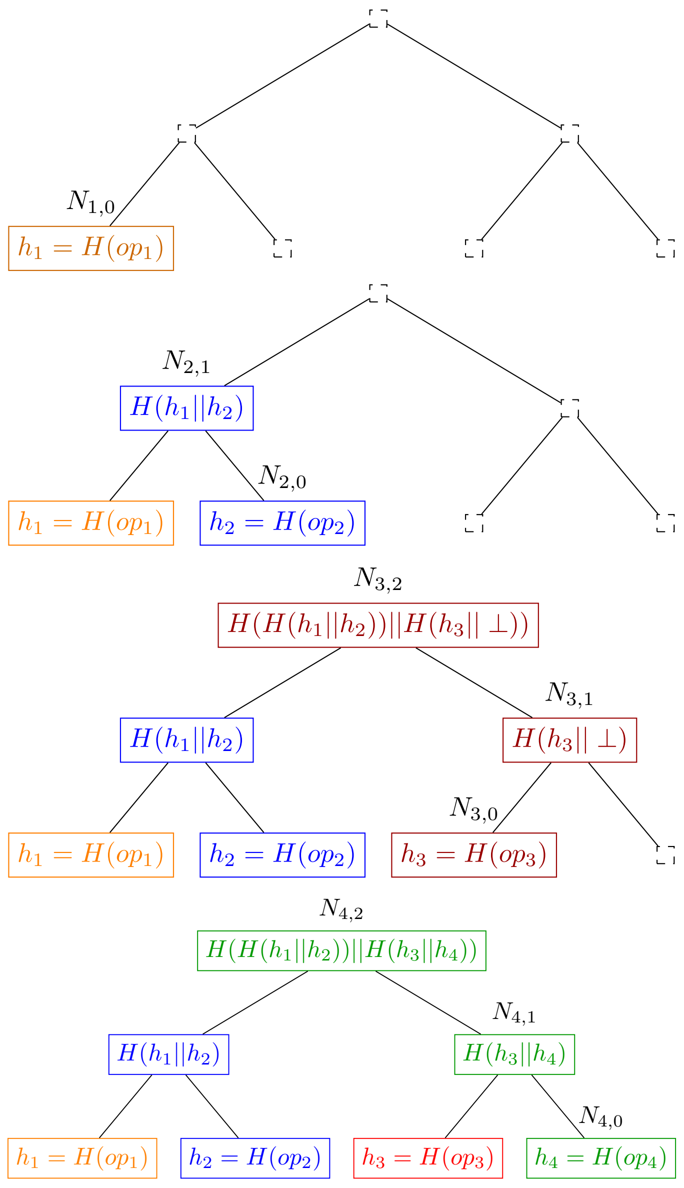


Fig. 4: Insertion paths of op_1, op_2, op_3, and op_4. Each insertion path is indicated by a distinct color. Each node is addressed with a pair of integers (p, l) as $N_{p,l}$ where i indicates the operation number and l stands for the level of node on the insertion path.

This results in two main observations. First, the assignment of the nodes circulates across the servers; that is $\{N_{1,0} \rightarrow S_1, N_{2,0} \rightarrow S_2, N_{2,1} \rightarrow S_3, N_{3,0} \rightarrow S_4, N_{3,1} \rightarrow S_1, N_{3,2} \rightarrow S_2, \ldots\}$, where $\rightarrow$ indicates the assignment and the pattern of S_1, S_2, S_3, S_4 recurs throughout the assignment. The first observation implies the second observation; that is, the labels of nodes assigned to each server are N distant. This is due to the fact that the storage of nodes is assigned to the servers with a round-robin pattern; hence the labels of two consecutive nodes received by a server are N distant. For example, consider the sequence of nodes stored at server S_1, i.e., $N_{1,0}, N_{3,1}, N_{4,2}, \ldots$ whose labels are $L(1,0) = 1, L(3,1) = 5, L(4,2) = 9, \ldots$, respectively; the difference between the label of each node and its preceding node is N, i.e., $L(3,1) - L(1,0) = 5 - 1 = 4$ and $L(4,2) - L(3,1) = 9 - 5 = 4$. In Section 8, we discuss how we enforce q-detectable-consistency using this circular storage distribution.

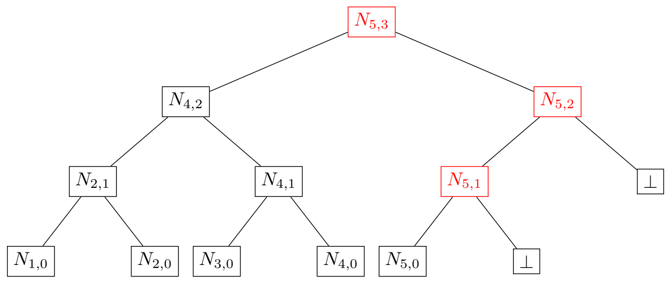


Fig. 5: 5th version of the shared *object*'s activity log. The temporary nodes are shown in red.

Fetching proofs in a distributed manner. In Integrita, there is no central entity holding a global view of the history tree associated with the activity log. As such, unlike the centralized system where one entity, i.e., the server, would create the membership and incremental proofs and shares with users, in Integrita, the user herself is responsible to determine the nodes along the proof path and fetch them from the corresponding storage providers (using Equation 8). Once the nodes are fetched, the correctness of the proof can be verified as described in Section 4 (more details can be also found in [20], [19]).

Identifying the nodes that are required for a membership or incremental proof is straightforward and exemplified in Section 4. However, due to the storage efficiency (explained in Section 7.1), Integrita makes a subtle change to that procedure. That is, in Integrita, the nodes of the tree that are temporary (with a non-full right or left sub-trees) never get stored at the server-side. Instead, if the value of a temporary node is required as part of a proof, it can be reconstructed from the full nodes with the highest level in it's left and it's right sub-trees. For example, consider node $N_{5,2}$ in Figure 5, which is a temporary node. The full nodes with the highest level in its left and right are $N_{5,0}$ and $\perp$, respectively. It is easy to verify that $N_{5,2}$ can be calculated by only having these two values, i.e., $N_{5,2} = H(H(N_{5,0}|| \perp)|| \perp)$. As such, if node $N_{5,2}$ is needed to construct a proof, the user should instead fetch $N_{5,0}$ from its corresponding storage server, i.e., $S_{F(5,0)} = S_3$ where F is defined in Equation 8 and the total number of servers is 4.

6 Integrita Construction

Integrita consists of four decentralized protocols, namely, *Create object*, *Update Status*, *Read*, and *Write*, which are run between a user and the servers. Through *Create Object* protocol, a user initiates the shared *object* and communicates necessary information about the *object* with the authorized users. A user runs *Update Status* protocol to fetch the root of the latest version of the history tree. A user inserts an operation to the history tree through the *Write* protocol. In the context of social networks, an operation may correspond to adding a photo to a profile, removing a video from the profile, or commenting on some post. Users engage in the *Read* protocol to read a subset of operations from the servers in a decentralized manner. The communication channel between the users and the servers are assumed to be authenticated.

The main protocols of Integrita rely on the following sub-protocols, which are essentially remote procedure calls (RPC) available at each server S_j where $j \in [1, N]$; namely, $S_j.Push$, $S_j.Pull$, and $S_j.GetStatus$. Each main protocol is a different orchestration of these RPC calls to a subset of storage servers.

6.0.1 Setup

Authorized users (with read and write access to the shared *object*) are each associated with a signature key pair. $FList = \{(U_j, vk_{U_j})\}_{j\in[1,T]}$ shall contain the username U_j and the signature verification key vk_{U_j} of each user where T is the total number of authorized users. $FList$ is publicly available to the servers and the authorized users. Moreover, the content that users operation on the *object* are all encrypted using symmetric key encryption where the key ek is given to the authorized users. For more fine-grained access control one can deploy the method proposed by [22], [56] or by leveraging attribute-based encryption similar to [3]. We assume that the users exchange $FList$ and the encryption key out of band; however, this can be outsourced to the server-side utilizing the method proposed by [22]. Also, for the ease of explanation, we assume that the set of authorized users is static, which can be extended to a dynamic version by deploying the proposal of [22]. In this paper, we rather focus only on view-consistency issues.

Each server has a signature key pair and a unique index in the range of $[1, N]$. We write S_i to indicate the server with index i. Servers publish the ordered list $SList = \{S_i\}_{i=1:N}$, where each server S_i's signature verification key is accessible through $S_i.vk$. Besides, the definition of the hash function H to be used in the history tree is publicly available. Each server also sets up a database DB to store the parts of the shared *object*'s activity log (history tree) for which it is responsible. Also, each server has a local $Status$ variable that reflects the address (p, l) (as demonstrated in Figure 4) of the latest node of the history tree stored (or seen) by that server.

Notation: Throughout our description, we distinguish between the data generated or operation performed by a server and a user using S and U subscript. That is, we write σ_{U_i} to indicate a signature generated by the i^{th} *user*. Likewise, $Sign_{U_i}(.)$, and $Verify_{U_i}(.)$ mean the execution of $Sign$ and $Verify$ algorithms using the signature key and verification key of the i^{th} *user*, respectively. Following the same pattern, we have σ_{S_i}, $Sign_{S_i}(.)$, and $Verify_{S_i}(.)$ for the i^{th} *server*.

6.1 Server-side Remote Procedure Calls

Each server S_j where $j \in [1, N]$ is available to the users through three RPCs namely, $S_j.Push$, $S_j.Pull$, and $S_j.GetStatus$. During the main protocols, the user may contact any of the servers to push or pull a node of the history tree. Each server has a local $Status$ variable that reflects the address (p, l) of the latest node of the history tree that is pushed to that server.

1) S_j.***Push***(U_i, (p, l), $in = (N_{p,l}, op, \sigma_{U_i})$): The details of the $Push$ RPC is shown in Algorithm 1. This function is invoked by a user to upload a single node of the history tree to the corresponding storage server. U_i indicates the calling user, (p, l) represents the address of the intended node (recall our addressing convention in Section 5.2), and in is the metadata about the intended node.

Algorithm 1 S_j.Push(U_i, (p, l) , $in = (N_{p,l}, op, \sigma_{U_i})$)

```
1:  if $U_i \notin FList$ then
2:      Return "Reject"
3:  if $F(p, l) \neq j$ then
4:      Return "Reject"
5:  if $L(p, l) - L(S_j.Status.p, S_j.Status.l) \neq N$ then
6:      Return "Reject"
7:  if $N_{p,l}$ is a leaf node then
8:      if $H(op) \neq N_{p,l}$ OR $Verify_{U_i}(N_{p,l}||p, \sigma_{U_i}) \neq$ accept then
9:          Return "Reject"
10: if $N_{p,l}$ is a tree digest then
11:     if $Verify_{U_i}(N_{p,l}||p, \sigma_{U_i}) \neq$ accept then
12:         Return "Reject"
13: Insert $(U_i, (p, l), in)$ into $DB$
14: if $S_j.Status.p \neq p$ then
15:     Remove the user signature $\sigma_{U_i}$ $\forall$ $\tau_i \in S_j.DB \setminus \tau_1$
16: $S_j.Status$=$(p, l)$;
17: if $N_{p,l}$ is a tree digest then
18:     Return $Sign_{S_j}(N_{p,l}||p)$
19: else
20:     Return "Accept"
```

Firstly, the server needs to perform several checks:

- The request is issued by an authorized user, which is $U_i \in FList$ (lines 1-2).
- The server is responsible for the storage the intended node (lines 3-4).
- The node is the next node that the server expects to receive (lines 5-6). Each server expects to receive a sequence of nodes whose labels are N distant. That is, $L(p, l)$ which is the label of the inserted node must equal to $L(S_j.Status.p, S_j.Status.l) + N$ where $F(S_j.Status.p, S_j.Status.l)$ is the label of the last seen node by the server. This is due to the fact that the storage of nodes are assigned to the servers with a circular pattern; hence the labels of two consecutive nodes received by a server are N distant (see Section 5.2).

Once all the checks are passed successfully, the server validates the metadata in. in has three parts:

- $N_{p,l}$ represents the hash value of the node. This parameter is non-empty except for the temporary nodes. That is, if the user is pushing a node with the address of (p, l) that belongs to a temporary node, then the user leaves $N_{p,l}$ empty. Recall from Section 5.2 that the temporary nodes are never saved at the server-side.
- *op*: This field is non-empty only for the leaf nodes.

op represents an operation of the activity log. The operation op can be deletion or insertion of a operation, e.g., $op = Insert||post$. Recall that in the history tree of the activity log, the leaves are the hash of the operations. Thus, if the inserted node is a leaf node, then the following should hold $N_{p,l} = H(op)$ (see Figure 4).

- σ_{U_i} is the user-generated signature over $N_{p,l}||p$. This field is set by the user if the inserted node is a leaf or a tree digest.

The server performs necessary verification over the content of in depending on the type of the node (lines 7-12). Upon successful verification, the server inserts the node and the associated metadata in to the database (line 13).

If the inserted node belongs to an operation that is newer than the last operation seen by the server (line 14), the server can erase the signatures that belong to the older tree digests. This will save storage for the server. Erasing the signatures will not affect provable view-consistency. This is formally proven in Section 8.

The server updates its Status variable (line 16) to reflect the address of the most recent node. Finally, if the inserted node is a tree digest, the server must sign the node (lines 17-18) and respond to the user accordingly (this signature is indeed a commitment from the server on the correct receipt of the tree digest). Otherwise, the server only acknowledges that the push operation is done (lines 19-20).

2) S_j***.Pull***(p, l): Algorithm 2 demonstrates the *Pull* procedure. This function gets the address (p,l) of a node and returns the information associated with that node. Initially, the server checks whether the calling user is authorized (line 1) as well as whether there is any record associated with that node in the database (line 3). If any of those fails, the algorithm outputs $\perp$ (lines 2 and 4). Otherwise, the server retrieves the record corresponding to the node from the DB (line 5). If the requested node is a tree digest, then the server generates a signature σ_{S_j} over $N_{p,l}||p$ (lines 6-7), otherwise leaves σ_{S_j} empty (lines 8-9). Finally, the server sends the $record$ and the signature (if any) to the user (line 10).

Algorithm 2 S_j.Pull(U_i, (p, l))

```
1: if $U_i \notin FList$ then
2:     Return $\perp$
3: if $DB.get((p,l)) = \perp$ then
4:     Return $\perp$
5: $record = DB.get(p,l)$
6: if $N_{p,l}$ is tree digest then
7:     $\sigma_{S_j} = Sign_{S_j}(N_{p,l})$
8: else
9:     $\sigma_{S_j} = \perp$
10: Return $(record, \sigma_{S_j})$
```

3) S_j***.GetStatus***(): Once this procedure is called, the server sends its signed $Status$ to the user (Algorithm 3).

Algorithm 3 S_j.GetStatus ()

```
1: $\sigma_{S_j} = Sign_{S_j}(S_j.Status)$
2: Return ($S_j.Status$, $\sigma_{S_j}$)
```

6.2 Main Protocols

6.2.1 Create object

During this protocol, the shared *object* gets initialized and the necessary information is communicated among the authorized users. The protocol is visualized in Figure 6.

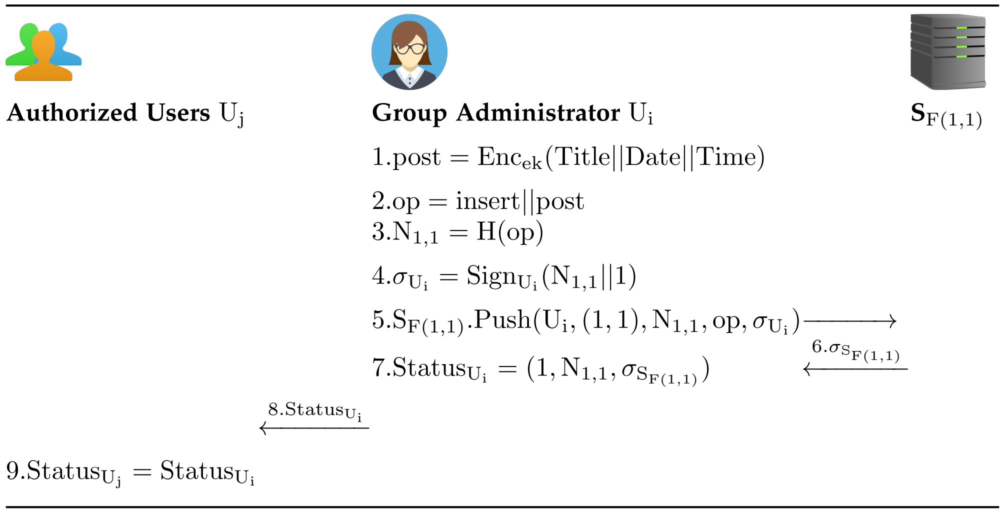
User: One of the authorized users U_i, e.g., the admin of a Facebook-like group page, runs this protocol. She inserts the very first operation op to the *object*'s activity log. The content of the initial operation must uniquely represent the *object*, e.g., the title of the group page together with its creation date and time. The group administrator proceeds as illustrated in steps 1-4 of Figure 6. Next, she submits $N_{1,1}$ to the corresponding storage server $S_{F(1,1)}$ by invoking its $Push$ function (step 5).

Server: The procedure of the $Push$ function is shown in Algorithm 1 and discussed in Section 6.1. Once the function is executed, and since the attempted node $N_{1,1}$ is a tree digest, the server signs $N_{1,1}||1$ and returns the signature $\sigma_{F(1,1)}$ back to the user (step 6).

Users: The group administrator updates her own Status (step 7) and also communicates her Status information with all the authorized users, e.g., group members (step 8). Notice that $Status$ is a local variable comprised of the following components $Status = (v, \tau_v, \sigma)$ where v reflects the last version of the shared *object* seen by the user, τ_v is the corresponding tree digest, and σ is the server-side signature over $\tau_v||v$. Note that for the initial operation, the group administrator sets the parameters as follows $v = 1$, $\tau_v = N_{1,1}$ and $\sigma = \sigma_{F(1,1)}$. All the other users initialize their $Status$ variable according to the group administrator's (step 9).

6.2.2 Update Status

In this protocol, demonstrated in Figure 7, the user interacts with N servers to find out and fetch the tree digest corresponding to the latest version of the activity log. To identify the latest version, one should find the index of the last operation inserted to the activity log. As such, the user collects the $Status$ variable of *all* the servers by invoking their $GetStatus$ function and determines the maximum and the minimum node addresses among those (steps 1-2). Identifying the minimum node address helps user to verify the consistency among the servers outputs. The outputs are consistent if the gap between the label of the max and min addresses, i.e., $L(p_{max}, l_{max}) - L(p_{min}, l_{min})$, is at most N. This is because the servers get to serve nodes of the activity log (history tree) in a circular manner; hence, the difference between the labels of the nodes seen by the servers is upper-bounded by N. If the constraint is satisfied, the user accepts p_{max} as the most up-to-date version of the activity log and pulls the corresponding tree digest from the respective storage server (steps 4-5). Since the user is pulling a tree digest, the output of the $Pull$ invocation will contain two signatures, one generated by the user (who has generated the p_{max}^{th} operation), and the server generated



Authorized Users $\mathrm{U_j}$
Group Administrator $\mathrm{U_i}$
$\mathbf{S}_{\mathrm{F(1,1)}}$
$1.\mathrm{post} = \mathrm{Enc_{ek}(Title||Date||Time)}$
$2.\mathrm{op} = \mathrm{insert||post}$
$3.\mathrm{N_{1,1}} = \mathrm{H(op)}$
$4.\sigma_{\mathrm{U_i}} = \mathrm{Sign_{U_i}(N_{1,1}||1)}$
$5.\mathrm{S_{F(1,1)}.Push(U_i, (1,1), N_{1,1}, op, \sigma_{U_i})} \longrightarrow$
$6.\sigma_{\mathrm{S_{F(1,1)}}}$
$7.\mathrm{Status_{U_i}} = (1, \mathrm{N_{1,1}}, \sigma_{\mathrm{S_{F(1,1)}}})$
$8.\mathrm{Status_{U_i}}$
$9.\mathrm{Status_{U_j}} = \mathrm{Status_{U_i}}$

Fig. 6: Create *object* protocol.

signature (recall Algorithm 2). Accordingly, the user verifies the associated signatures (steps 6-7). If any failure happens, the user aborts. Next, the user checks whether her local Status variable, i.e., τ_v is consistent with τ_{p^*}. This can be done through an incremental proof between τ_v and τ_{p^*} (steps 9-14). To this end, the user identifies the nodes along the incremental proof (step 9) and downloads them from the corresponding storage servers (step 10-11). Next, for each node on the path, she authenticates the associated signatures (step 12) and aborts in case the authentication fails. Once the proof is obtained, she verifies whether the proof can correctly assert the consistency between τ_v and τ_{p^*} (step 14). If the verification does not pass, the user aborts. Otherwise, she updates her $Status$ value (step 15).

6.2.3 *Read*

During the $Read$ protocol, which is illustrated in Figure 8, the user reads a certain range $R = [x, y]$ of operations, i.e., $op_x, \cdots, op_y$ (line 1). At first, the user updates her Status variable through Update Status protocol (line 2) to learn the latest version of the activity log. Next, she specifies the nodes along the membership proofs of the intended operations (line 3) and contacts the corresponding storage servers to fetch the nodes (lines 4-6). The leaf nodes of the proof should be appropriately signed by the issuing users (line 7). Also, the user should ensure that a leaf node is the hash of the corresponding operation (line 8). If any verification fails, the user aborts. Ultimately, the user verifies the correctness of the membership proofs of the operations against τ^* (line 10). If the result of the membership proof is false, then the user aborts (line 13).

6.2.4 *Write*

In this protocol, illustrated in Figure 9, a user interacts with the servers to insert her operation to the *object*. The user initially runs *Update Status* protocol to fetch the latest tree digest τ^* corresponding to the latest version of the *object* (line 1) also determines the index of the next operation (line 2).

The user sets her operation (line 3) and signs it (line 4). She identifies the nodes along the insertion path of her operation together with their siblings (line 5) and then fetches the values from the corresponding storage servers (lines 6-11). For the leaf nodes, the signature of the user who has generated that operation (line 9) as well as the computed hash values (line 10) must be checked. If anything goes wrong, the user aborts. Next, she recomputes the hash values of the nodes along the insertion path of her operation (line 12) and then attempts to submit them to the corresponding servers by invoking their $Push$ functions (lines 13-20). She builds the necessary metadata in for the submission of each node as explained below:

1) If the node is a leaf node (line 14), the user needs to submit the operation op_{p_c}, the hash of the operation, as well as the signature σ.
2) For a full node (line 15), only the hash value of the node is needed for the metadata.
3) For a temporary node (line 16), no metadata is required (all the fields of in are empty). The corresponding server only gets contacted to be informed about the insertion of the new operation.

If the response of any of the servers (line 17) is *reject* (line 18), then inconsistency is detected and the user aborts. Finally, the user submits the tree digest to the corresponding server (line 19), obtains the server-side signature, authenticates the signature (line 20), and updates her Status accordingly (line 21). If the server signature did not get verified, the user aborts.

Audit: As discussed before, each user is responsible to ensure that her operation is correctly inserted to the *object* and is visible to all the other users. As such, every write operation must be followed by the audit procedure (lines 23-24) during which the user waits for q or more operations to be inserted into the object. q depends on p_c and N and is formulated by Equation 10 i.e., the user waits for $q = Q(p_c, N)$ operations (this dependency is explained below, under the audit threshold subsection). Then, the user checks whether the tree digest at version p_c is consistent with the tree digest at version $p_c + q$ of the object. She

```
User U                                                                              S_{j∈[1,N]}
1. $(p_{max}, l_{max}) = \max_{i\in[1,N]} \{S_i.GetStatus()\}$                                   $\Longleftrightarrow$
2. $(p_{min}, l_{min}) = \min_{i\in[1,N]} \{S_i.GetStatus()\}$                                   $\Longleftrightarrow$
3. If $(L(p_{max}, l_{max}) - L(p_{min}, l_{min}) >= N)$ then Abort
4. $p^* = p_{max} \quad l^* = \lceil \log(p_{max}) \rceil + 1$
5. $(U', (p^*, l^*), (\tau_{p^*}, \perp, \sigma_{U'}), \sigma_{S_{F(p^*,l^*)}}) = S_{F(p^*,l^*)}.Pull(U, (p^*, l^*))$   $\Longleftrightarrow$
6. If $Verify_{U'}(\tau_{p^*} || p^*, \sigma_{U'}) ==$ False then Abort
7. If $Verify_{S_{F(p^*,l^*)}}(\tau_{p^*} || p^*, \sigma_{S_{F(p^*,l^*)}}) ==$ False then Abort
8. Retrieve $\tau_v$ from U.Status
9. path =Nodes $(p_i, l_i)$ along the incremental proof of $\tau_v$ and $\tau_{p^*}$
10. For $(p_i, l_i) \in$ path
11.     $(U_k, (p_i, l_i), (N_{p_i,l_i}, op, \sigma_{U_k}), \sigma_{S_{F(p_i,l_i)}}) = S_{F(p_i,l_i)}.Pull(U, p_i, l_i)$ $\Longleftrightarrow$
12.     If $N_{p_i,l_i}$ is a leaf
            If $Verify_{U_k}(N_{p_i,l_i} || p_i, \sigma_{U_k}) ==$ False then Abort
13.     proof.insert$((p_i, l_i), N_{p_i,l_i}, op)$
14. If $(IncVrfy(\tau_v, \tau^*, proof) ==$ False$)$ then Abort
15. $U.Status = (p^*, \tau^*, \sigma_{S_{F(p^*,l^*)}})$
```

Fig. 7: Update Status protocol. The arrows indicate the user's interaction with the servers.

```
User U                                                                              S_{j∈[1,N]}
1. Select a range $R = [x, y]$
2. Run Update Status and fetch the latest tree digest$\tau^*$                        $\Longleftrightarrow$
3. path = Nodes $(p_i, l_i)$ along the membership proof of $op_x \cdots op_y$
4. For $(p_i, l_i) \in$ path
5.     $(U_k, (p_i, l_i), (N_{p_i,l_i}, op, \sigma_{U_k}), -) = S_{F(p_i,l_i)}.Pull(U, p_i, l_i)$      $\Longleftrightarrow$
6.     If $N_{p_i,l_i}$ is a leaf
7.             If $Verify_{U_k}(N_{p_i,l_i} || p_i, \sigma_{U_k}) ==$ False then Abort
8.             If $N_{p_i,l_i} \neq H(op)$ then Abort
9.     proof.insert$((p_i, l_i), N_{p_i,l_i}, op)$
10. If $(MemVrfy(\tau^*, \{(j, N_{j,0})\}_{j\in R}, proof) ==$ False$)$ then Abort
```

Fig. 8: Read protocol. The arrows indicate the user's interaction with the servers.

```
User U                                                                                    $\mathbf{S}_{j\in[1,N]}$
1. Run Update Status and fetch $(p^*, \tau^*, \sigma_{S_{F(p^*,\lceil\log(p^*)\rceil)}})$          $\Longleftrightarrow$
2. $p_c = p^* + 1$ , $l_c = \lceil\log(p_c)\rceil$
3. $op_{p_c}$ = Craft the operation
4. $\sigma = Sign_{u_i}(H(op_{p_c})||p_c)$
5. path = Nodes $(p_i, l_i)$ along the insertion path of $op_{p_c}$ and their siblings
6. For $(p_i, l_i) \in$ path
7.     $(U_k, (p_i, l_i), (N_{p_i,l_i}, op, \sigma_{U_k}), -) = S_{F(p_i,l_i)}.Pull(U, (p_i, l_i))$          $\Longleftrightarrow$
8.     If $N_{p_i,l_i}$ is a leaf
9.             If $Verify_{U_k}(N_{p_i,l_i}||p_i, \sigma_{U_k})$ == False then Abort
10.            If $N_{p_i,l_i} \neq H(op)$ then Abort
11.     proof.insert($(p_i, l_i), N_{p_i,l_i}, op$)
12. Using proof, recalculate the insertion path $\{N_{p_c,l_i}\}_{l_i\in[0,l_c]}$
13. For $N_{p_c,l_i} \in \{N_{p_c,0}, \cdots, N_{p_c,l_c-1}\}$
14.     If $N_{p_c,l_i}$ is a leaf node: in = $(N_{p_c,l_i}, op_{p_c}, \sigma)$
15.     If $N_{p_c,l_i}$ is a full node: in = $(N_{p_c,l_i}, \perp, \perp)$
16.     If $N_{p_c,l_i}$ is a temporary node: in = $(\perp, \perp, \perp)$
17.     Res = $S_{F(p_c,l_i)}.Push(U_i, (p_c, l_i), in)$          $\Longleftrightarrow$
18.     If Res == Reject then Abort
19. $\sigma_{S_{F(p_c,l_c)}} = S_{F(p_c,l_c)}.Push(U_i, (p_c, l_c), (N_{p_c,l_c}, \perp, \sigma))$          $\Longleftrightarrow$
20. If $Verify_{S_{F(p_c,l_c)}}(N_{p_c,l_c}, \sigma_{S_{F(p_c,l_c)}})$ == False then Abort
21. $U_i$.Status = $(N_{p_c,l_c}, \sigma_{S_{F(p_c,l_c)}})$
 Audit Procedure
23. Wait for T operations to be inseterted s.t. $T \geq Q(p_c, N)$
24. Run Update Status, if Abort occurs then Abort
```

Fig. 9: Write protocol. The arrows indicate the user's interaction with the servers.

carries out this consistency check by the execution of the update Status protocol (during which the incremental proof between τ_{p_c} and τ_{p_c+q} is checked). If *Update Status* terminates successfully, then the user knows that her operation is consistently visible to all the other users. Otherwise, servers must have equivocated about the user's operation, which is caught by the user.

Audit Threshold: The value of q is formulated in Equation 10 and is a function of the *object* version v at which the write operation takes place (i.e., $v = p_c$ in Figure 9) and the number of servers N. We refer to q as the *audit threshold*.

$$Q(v, N) = min(q) \text{ s.t. } \left[\sum_{j=0}^{q} \lceil log(v + j)\rceil + 1 \geq N\right] \quad (10)$$

As a concrete example, assume a system with $N = 8$ servers. A user who inserts the second operation $v = 2$ shall execute Update Status after the insertion of $Q(2, 8) = 2$ more operations into the *object*, i.e., at the version $v' = v + q = 2 + 2 = 4^{th}$, or any later version i.e., $v' \geq 4$ of the *object*. If the *Update Status* protocol does not end successfully, then there is a view inconsistency, e.g., servers attempted to drop her operation or replace it with another one.

Below, we provide intuition as to why auditing the *object* after $Q(p, N)$ operations will result in achieving q-detectable consistency. Moreover, in Section 8, we provide a formal security definition for a q-detectable consistent system followed by a proof asserting that Integrita is q-consistent relying on our proposed auditing strategy.

For the insertion of each operation i, the servers that are located on the insertion path will be informed about the insertion regardless of the type of nodes they are responsible for (see Figure 9, lines 13-19). We assume that at least one of the servers is honest, as this is a minimal requirement. We call an operation "frozen" if one or some of the nodes along its insertion path are assigned to the honest server.

It is named frozen since due to the presence of the honest server, no other operation with the same index will exist; the honest server will not accept the insertion of two operations with the same index (as indicated in line 5 of Algorithm 1). This implies that for the frozen operation with the index of f, there would be only one tree digest τ_f in the system, representing a unique history (sequence of operations) of the *object*. We call a tree digest corresponding to a frozen operation as a frozen tree digest. All the other tree digests τ_j created as the result of further write operations $j > f$ will comply with the history that τ_f represents (this is due to the incremental proof check in Step 11 of the Update Status protocol). Thus, if an operation i where $i < f$ belongs to the sequence of operations that a frozen tree digest τ_f represents, then it will certainly belong to all the future versions of the *object*. Thus, to ensure view-consistency, the user needs to perform a consistency check between the tree digest at the time of insertion of her operation and the very next frozen tree digest. To determine the index of the next frozen tree digest, we need to know the index of the honest server. However, there is no presumption about which server will act honestly. As such, after insertion of each operation i, the user shall wait for q many operations to be inserted *as the result of which all the servers get contacted at least once*. As the storage of nodes is assigned to the servers under a round-robin fashion, if the sum of the length of the insertion path of the next q operations exceeds N, it means that all the N servers, including the honest server whose index is unknown, are contacted at least once. Equation 10 calculates q, i.e., the total number of operations (inserted after i^{th} operation) whose insertion paths' lengths in total exceeds N. Recall that the number of nodes located on the insertion path of j^{th} operation is $\lceil log(j) \rceil + 1$ which means $\lceil log(j) \rceil + 1$ distinct servers get contacted as the result of insertion of the j^{th} operation.

Analysis of Audit Threshold: Figure 10 shows the audit threshold computed based on the function $Q(v, N)$ (Equation 10) under different number of servers N and operation number v. The audit threshold for a particular operation number will increase with the number of servers, e.g., the audit threshold for operation number 65 for $N = 8, 16, 24$, and 32 are $0, 1, 2$, and 3, respectively.

After a certain version of the *object*, each inserted operation is a frozen one since all the servers get contacted along the insertion of each operation. This is because the insertion path length exceeds the number of servers for each operation. Indeed, the *object* enters its strong consistent version where no fork can happen in users' views. We call that version of the *object* as the *transition point* of the *object*. For a given N, the transition point is computed based on Equation 11.

$$TP(N) = 2^{N-2} + 1 \tag{11}$$

For example, with $N = 8$, the strong view consistency starts at version 65, whereas with $N = 16$ the transition point is 16385. Thus, the higher the number of servers, the later the *object* transits to its strong consistent version. The transition points of the different numbers of servers (1-20) are illustrated in Figure 11.

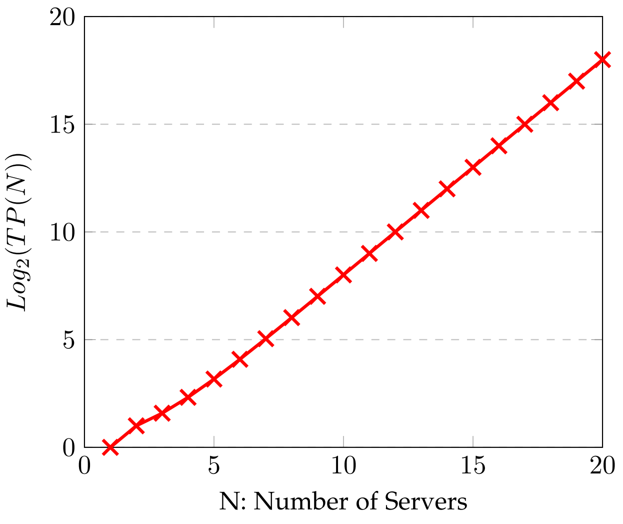


Fig. 11: The *object*'s transition point for various number of servers as defined in Equation 11. The x axis represents the number of servers whereas the y axis shows the binary logarithm of the transition point.

7 Complexity and Performance

In this section, we analyze the asymptotic performance of Integrita with respect to the storage overhead (Section 7.1), as well as the round and communication complexity (Section 7.2) for both the servers and the users. Throughout this section, we consider $|H|$ to be the bit-length of hash values whereas $|\sigma|$ indicates the bit-length of each signature.

7.1 Storage Complexity

User: Users have to store constant amount of data for their Status variable.

Server: Servers are responsible for storing the object's posts, and the activity log. Note that in the history tree representing the activity log, only leaves, full nodes, and tree digests are maintained by the servers but no temporary nodes.

An activity log with M operations (M being a power of two) consists of M leaves, $M - 1$ full nodes, and M tree digests. However, out of the M tree digests, some of them overlap with the full nodes hence are already saved in the system. Indeed, out of the M tree digests (for M operations), $log(M)$ many of them (associated with the operations with the indices 2^0, 2^1, 2^2,..., $2^{log(M)}$) are full nodes. Therefore, the total number of full nodes and tree digests stored is $M + M - 1 - log(M)$. Additionally, each leaf node is attached to a user-side signature, thus, M signatures shall be maintained by the servers on aggregate. Similarly, tree digests are also signed by the users, though, only the signature of the $O(N)$ most recent tree digests are kept in the system (lines 14-15 of Algorithm 1).

The overall storage consumption by the N servers is as shown in Equation 12 which is of $O(M)$.

$$|H| \cdot (\underbrace{M}_{\text{leaves}} + \underbrace{M + M - 1 - log(M)}_{\text{tree digests and full nodes}}) + |\sigma| \cdot (M + N) \tag{12}$$

Comparison with related work: In the replication-based solutions [37], [13], [25], [26], [45], [52], [44], the activity

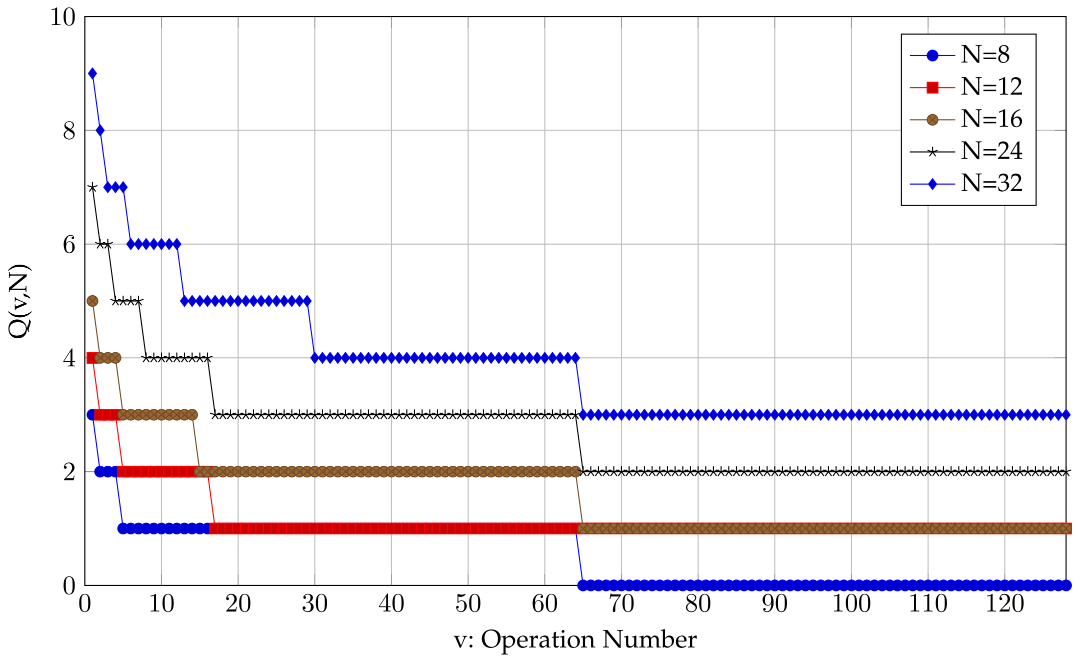


Fig. 10: The Audit Threshold for a various number of servers each demonstrated by a different color. The x axis represents the index of operation whereas the y axis shows the audit threshold computed based on function $Q(v, N)$ given in Equation 10.

log must be replicated over all the N servers. The history tree associated with the activity log with M operations has $2 \cdot M - 1$ nodes. Also, all the M leaves are attached to a user-generated signature. Thus, the storage of a single activity log takes up $|H| \cdot (2 \cdot M - 1) + |\sigma| \cdot M$ bits. Each server has one copy of the activity log, which on aggregate results in $N \cdot [|H| \cdot (2 \cdot M - 1) + |\sigma| \cdot M]$ space complexity for the entire system. On the other side, in the centralized architectures [22], [23], there is only one central server which holds a single copy of the activity log which imposes $|H| \cdot (2 \cdot M - 1) + |\sigma| \cdot M$ storage overhead.

Concrete Storage: Table 1 summarizes the comparison of the storage overhead between Integrita and the related studies. For the concrete overhead, we consider Facebook walls as the shared object and the average number of posts on users walls as the total number of operations in the activity log (though, we believe the true number of operations including the deletion operations would be higher than the number of posts). On Facebook, each wall on average has 1241 posts (per year)[7]. Thus we set $M = 1241$. Moreover, the number of monthly active users on Facebook is approximately 2.41 billion[8] which means 2.41 billion walls (activity logs) should be maintained by the OSN. The values reported in Table 1 are the storage required in total for 2.41 billion activity logs each with $M = 1241$ operations. We assume the deployment of SHA-3 as the hash function with a 512-bit output length and RSA signature scheme with a 2048-bit signature length. The number of servers is set to $N = 20$. We further examine the storage overhead for $N \in [20, 40]$ in Figures 12 and 13, which illustrate the total and per-server storage overhead, respectively.

By inspecting Figure 12, we draw two observations regarding the superiority of Integrita over the existing solutions. First, the storage consumption of Integrita is independent of the number of servers. This makes Integrita an efficient solution compared to the replication-based methods where the overhead grows linearly with the number of servers (recall that in replication-based solutions servers are the replica of the activity logs). Second, Integrita storage requirement is very close to the centralized architecture (only 10% more) yet can guarantee a stronger level of view consistency i.e., q-detectable consistency without relying on users communication nor replication.

With respect to the per-server storage requirement, as Figure 13 illustrates, Integrita outperforms both counterparts by incurring only $O(\frac{1}{N})$th of their overhead. This is expected since, in Integrita, the activity log gets split across the servers, and the proposed distributed storage algorithm comes with an innate load balancing property. This gives great flexibility to the system designer to adjust the number of servers with respect to the storage capacity of individual servers. For example, if the total storage requirement is 13504TB and each server has 421TB capacity, then the number of required servers will be 32 (find 421TB on the y-axis of Figure 13 which is 8.72 ($2^{8.72} \approx 421$), and then spot the corresponding x-value on the Integrita's curve i.e., 32).

7. https://blog.wishpond.com/post/115675435109/40-up-to-date-facebook-facts-and-stats

8. https://www.businessinsider.com/facebook-grew-monthly-average-users-in-q1-2019-4

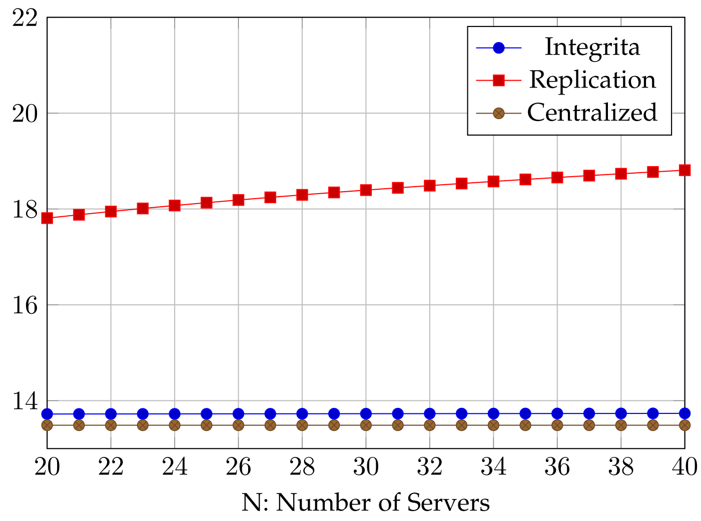


Fig. 12: Total storage overhead. y axis represents binary logarithm of the storage consumption in Terabytes.

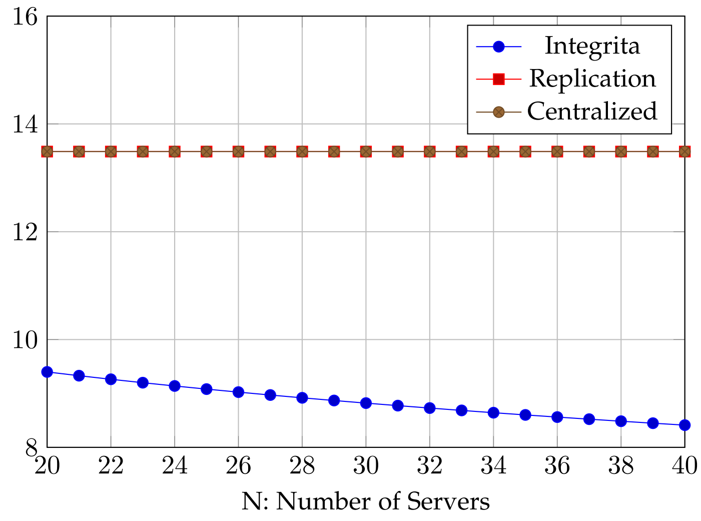


Fig. 13: Storage overhead per server. y axis represents binary logarithm of the storage consumption in Terabytes.

7.2 Round Complexity and Communication Complexity

In this section, we analyze the round complexity as well as the communication complexity, i.e., the number of bits communicated between parties during the main protocols. We consider each communication round to be one send and one receive operation. If multiple send and receive actions can be done concurrently (since they are independent), then we count them as one round. The results are summarized in Table 2 and Table 3. M and N stand for the total number of operations in the activity log and the total number of servers, respectively.

- *Update Status:* In this protocol as shown in Figure 7, the user communicates with the servers at 3 different steps (1,5, and 11). The result of step 1, where the user gets the Status of all the servers, is necessary for the next two communications. Thus, step 1 should precede steps 5 and 11. However, steps 5 and 11, during which the user downloads the tree digest and the incremental proof, respectively, can be executed in parallel. Thus, the overall round complexity of *Update Status* is 2 for the users. The same holds for the servers who may get contacted twice during this protocol, once to share their latest status and the second time when the server may be holding the tree digest or it is located on the proof path.
  The communication complexity of *Update Status* is due to the download of the signed Status of the servers (step 1) as well as fetching the most recent tree digest (step 5) and the incremental proof (step 11). The transmission of Status variables incurs $SSzie \cdot N$ bits data exchange for N server generated signatures. The response to the pull request of the tree digest embodies $|H| + 2 \cdot SSize$ bits data (the tree digest and two signatures, one server-generated, and the other user-generated). The incremental proof will approximately contain $2 \cdot log_2(M)$ of tree nodes where at most 4 of them are leaves. The metadata associated with these nodes will be $2 \cdot log_2(M) \cdot |H| + 4 \cdot SSize$ bits (where $2 \cdot log_2(M) \cdot |H|$ portion is for the hash values of the nodes and $4 \cdot SSize$ correspond to the user signatures for the 4 leaves). Thus, on aggregate, the communication complexity at the user side is at most $|\sigma| \cdot (N+6) + |H| \cdot 2 \cdot log(M)$. The communication complexity for each server on the average is $\frac{SSize \cdot (N+6) + |H| \cdot 2 \cdot log(M)}{N} \approx SSize + |H| \cdot \frac{2 \cdot log(M)}{N}$.
- *Read:* During the Read protocol, shown in Figure 8, the user communicates with the servers at two phases, once at step 2 to fetch the latest tree digest by executing Update Status protocol and another time at step 5, to pull the nodes along the membership proofs. The former has the round complexity of 2 whereas for the latter the user can connect to all the servers simultaneously; hence, she can fetch all the nodes in 1 round of communication. The round complexity can be enhanced further by combining the second round of Update Status protocol with step 5 of the Read protocol. This is doable since, in both phases, the user pulls a set of nodes from the servers where the indices of nodes are known and independent. Therefore, the overall round complexity of the Read protocol for the user is 2. Similarly, each server may get contacted at most 2 times.
  To read the range $[i, j]$ of operations, the user downloads $R = j - i + 1$ many leaf nodes with their user-side signatures, which requires $R \cdot [SSize + |H|]$ bits of transmission. The membership proof of R operations entails at most $2 \cdot log(M)$ hash values. On aggregate, the user communicates $|H| \cdot (2.log(M) + R) + |\sigma| \cdot R$ bits data. Consequently, the average data transfer for each server is $|\sigma| \cdot R + \frac{|H| \cdot (2.log(M)+R))}{N}$. Note that this is in addition to the data exchange incurred by the Update Status protocol that is run at the beginning of the Read protocol.
- *Write:* At the beginning of this protocol, as illustrated in Figure 9, the user obtains the latest tree digest via Update Status protocol, which imposes 2 rounds of communication. Next, at step 7, the user fetches the nodes on the insertion path of her operation. This can be handled in 1 round of communication with concurrent connections to the servers. Step 7 can be merged with the second round of communication in Update Status protocol since all the nodes fetched at these two phases

Design	Centralized	Replication-based	Integrita
Asymptotic	$\|H\| \cdot (2 \cdot M - 1) + \|\sigma\| \cdot M$	$N \cdot [\|H\| \cdot (2 \cdot M - 1) + \|\sigma\| \cdot M]$	$\|H\| \cdot (3M - 1 - Log(M)) + \|\sigma\| \cdot (M + N)$
Concrete	11483TB	229663TB	13504TB
Per-Server	11483TB	11483TB	675TB
View Consistency Level	Fork-consistency	Strong consistency	q-detectable consistency

TABLE 1: Storage overhead of Integrita vs. related work. N: the number of servers. M: the number of operation in the activity log. $|\sigma|$ and $|H|$ represent the bit-length of the signatures and hashed values, respectively.

are independent of each other and can be downloaded concurrently. At steps 17 and 19, the user recomputes the values for the nodes on the insertion path of her operation and pushes the new values to the servers. This also counts as 1 round of communication. Thus, in total, the *Write* protocol costs 3 rounds of communication for the user. Subsequently, each server may get involved in at most 3 rounds.

From the communication complexity perspective, the insertion path of the current operation requires a download of at most $2 \cdot log(M)$ hash values whereas pushing back the updated nodes requires the exchange of $|H| \cdot log(M)$ bits. As such, the user communicates $|H| \cdot 3 \cdot log(M)$ bits with the servers. The data transfer at the server-side, on the average, shall be $\frac{|H| \cdot 3 \cdot log(M)}{N}$. Note that, in this calculation we excluded the data exchange incurred by the Update Status protocol that is run at the beginning of the *Write* protocol.

8 Security

8.1 q-Detectable View Consistency and Inconsistency Interval

In this section, we provide a different interpretation of q-Detectable view-consistency. Recall that we name this level of consistency as q-detectable since servers' misbehavior in perturbing views cannot last for longer than q operations. Another way to look at this definition is that in a q-detectable consistent system, where the shared *object* is in its v^{th} version (v operations are applied to the activity log), the users are guaranteed to have consistent views towards all the operations in the activity log except the last δ operations, i.e., $op_{v-\delta}, ..., op_v$. We use the term *inconsistency interval* to refer to the range of operations $[v-\delta, v]$ where no guarantee about consistency is in place. The views of the users for any history of the *object* preceding the $v-\delta^{th}$ version of the *object* is guaranteed to be the same. In Integrita, δ is a function of the *object* version v and the number of servers N, and its value is computed using the function $\Delta(v, N)$ formulated in Equation 13.

$$\Delta(v, N) = max(q \in [0, v]) \text{ s.t. } \left[\sum_{j=0}^{q} \lceil log(v-j) \rceil + 1 \geq N\right] \tag{13}$$

As a concrete example, assume a system with $N = 8$ servers and an *object* at its 5^{th} version. The inconsistency interval is 2 ($\Delta(5, 8) = 2$), that is, the view consistency holds for all the operations except the 4^{th} and the 5^{th} operation. As such, the following two views $View_5 = \{op_1, op_2, op_3, op_4, op_5\}$ and $View'_5 = \{op_1, op_2, op_3, op'_4, op'_5\}$ are q-consistent since, the consistency holds for all the operations out of the inconsistency interval. However, the following two views $View_5 = \{op_1, op_2, op_3, op_4, op_5\}$ and $View'_5 = \{op_1, op'_2, op_3, op'_4, op'_5\}$ do not satisfy q-consistency because there is an inconsistency at the second operation which is out of the inconsistency interval.

Audit threshold vs inconsistency interval: The audit threshold Q (given in Equation 10) and inconsistency interval Δ (given in Equation 13) measure the number of operations whose aggregated insertion path lengths exceeds N (N is the total number of servers). But there is a slight difference; Q measures this value for the future of an operation whereas Δ measures this value for the recent past of an operation. To be more specific, the audit threshold and inconsistency interval help answering distinct questions given below.

- For the newly inserted operation v, what is the future version v' of the tree at which users are guaranteed to have consistent view of the operation v? the answer is $v' = v + Q(v, N)$ where $Q(v, N)$ indicates the audit threshold.
- Given the current version v' of the tree, what is the latest operation number v preceding v' that is consistently visible to all the users (in other words, the audit threshold of v has passed)? the answer is $v = v' - \Delta(v', N)$ where $\Delta(v', N)$ represents the inconsistency interval.

The main observation from the the arguments above is that for two operation numbers v and v' where $v' = v+Q(v, N)$, we have $v = v' - \Delta(v, N)$. In other words, the audit threshold for the operation number v is equal to the inconsistency interval of the operation number $v' = v + Q(v, N)$ i.e., $\Delta(v + Q(v, N), N) = Q(v, N)$. To capture the notion of q-consistency, we define the following game to be played between an adversary $\mathcal{A}$ and a challenger $Chal$. The adversary controls $N - 1$ servers, whereas the challenger gets to play for the authorized users $U_i \in FList$ and the honest server. To represent servers under the control of the adversary we write $S_{i \in \mathcal{A}}$ whereas for the one server running by the challenger we denote S_{Chal}. The adversary can specify a user to run the read or write protocol. The challenge for the adversary is to make two users U and U' accept two q-inconsistent views of the *object*.

q-Detectable Consistency Experiment q-Det-Cons(1^λ)

1) The challenger gives the security parameter 1^λ to the adversary. Next, the challenger and the adversary exchange the signature verification keys of the parties they control. The challenger hands in the adversary $vk_{S_{Chal}}$ and

Entity\Overhead	Update Status	Read	Write
User	2	2	3
Servers	2	2	3

TABLE 2: Integrita Round Complexity

Entity\Overhead	Update Status	Read	Write
User	$\lvert\sigma\rvert \cdot (N+6) + \lvert H\rvert \cdot 2 \cdot log(M)$	$\lvert\sigma\rvert \cdot R + \lvert H\rvert \cdot (2.log(M) + R)$	$\lvert H\rvert \cdot 3 \cdot log(M)$
Servers	$SSize + hSzie \cdot \frac{2 \cdot log(M)}{N}$	$\lvert\sigma\rvert \cdot R + \frac{\lvert H\rvert \cdot (2.log(M)+R)}{N}$	$\frac{\lvert H\rvert \cdot 3 \cdot log(M)}{N}$

TABLE 3: Integrita Communication Complexity. N: the number of servers. M: the number of posts on the object. $|\sigma|$: the size of each signature in bit length. $|H|$: the bit-length of the hash output. R: the number of consecutive posts to be read from the object. In the values reported for the *Read* and *Write* protocols, the communication complexity of the *Update Status* protocol (which is executed at the beginning of these protocols) is not included.

$\{vk_{U_1}, \cdots, vk_{U_T}\}$ that are the verification keys of the honest server and the authorized users, respectively. Likewise, the adversary delivers a set of signature verification keys $\{vk_{S_i}\}_{i \in \mathcal{A}}$ for the corrupted servers.

2) The adversary designates the group administrator U_i. The challenger initiates the shared *object* on behalf of U_i through *Create object* protocol.
Steps 3 and 4 can be repeated polynomial times by the adversary, in any order.
3) The adversary specifies a user U_i to run the $Write$ protocol and insert an operation *op* to the *object*'s activity log. The content of *op* is determined by the adversary. The challenger runs the $Write$ protocol accordingly. Note that the challenger shall act upon the Audit procedure (which is a subroutine of the Write protocol) without needing the adversary's request.
4) The adversary determines a range of operations $R = [l, r]$ to be read by a particular user U_i. $Chal$ runs the *Read* protocol accordingly.
5) The adversary specifies two users U and U', a version number j, and the index j^* of an operation such that j^* is out of the inconsistency interval of the j^{th} version of the shared *object*, i.e., $j^* < j - \Delta(j, N)$. Note that when j^* is out of the inconsistency interval, the corresponding operation op_{j^*} is no longer subject to inconsistency and all the users including U and U' must have a consistent view of it unless the adversary could successfully fork the views and stay unnoticed (this is going to be determined in the rest of the game). The challenger attempts to read the j^{th} operation on behalf of both U and U' via the execution of the Read protocol. $\mathcal{A}$ wins if
   a) the *Read* protocol terminates successfully for both U and U' while the Status variable of both users point to the j^{th} version of the shared *object*
   b) and U and U' have obtained two different values for the j^{*th} operation. That is, U and U' have read op_{j^*} and op'_{j^*}, respectively while $op_{j^*} \neq op'_{j^*}$.

Definition 8.1. A storage system provides q-detectable consistency if the success probability of any probabilistic polynomial time (PPT) adversary in the q-Det-Cons(1^λ) experiment is negligible in the security parameter λ.

Theorem 1. If the deployed signature scheme is existentially unforgeable under adaptive chosen message attack and the hash function is collision-resistant, then Integrita provides q-detectable consistency.

Proof of Theorem 1: If there exists a PPT adversary $\mathcal{A}$ who wins q-Det-Cons(1^λ) with non-negligible probability ϵ, then we construct a PPT simulator B who finds a forgery for the underlying signature scheme.

Proof Overview: Before diving into the full proof, we provide an overview. The success of the adversary indicates that U and U' have read two different values for the j^* operation where $j^* < j - \Delta(j, N)$. Therefore, the system undertook a fork at the j^* operation due to which users are split into two groups depending on whether they are shown op_{j^*} or op'_{j^*}. Also, based on step 5.a of the game, the presence of a fork has stayed unnoticed and continued successfully till the j^{th} version of the object. This means that each fork should have a successful chain of write operations from i^{*th} to j^{th} version of the object. Recall that within this interval, there is at least one frozen operation (recall that a frozen operation is the one that the honest server sits on its insertion path). Let k where $j^* < k < j^* + Q(j^*, N) < j$ denote the index of that operation. The honest server accepts only one operation with index k, thus there will be only one valid tree digest at version k. This further implies that only one fork will get to grow. For the other fork (namely the second fork) to grow, the corrupted servers need to bypass the honest server. For this to happen the corrupted servers have two choices:

- The corrupted servers need to convince the users of the second fork that the last operation on the object has an index higher than k so that they won't attempt insertion of the k^{th} operation. However, this would only happen if the corrupted servers can generate an authenticated operation and tree digest on behalf of an authorized user. Thus, if the simulator B can guess for which authorized user this forgery takes place, B will exploit this forgery and breaks the unforgeability of the underlying signature scheme.

- Alternatively, the corrupted servers can bypass the honest server if they can successfully convince the users of the second fork to accept τ_k and op_k (generated by the first fork). This implies that the servers have to craft a valid incremental proof to τ_k from an older tree digest seen by the users of the second fork. Since the Status variable of all the users in the second fork contains a tree digest that is consistent with τ'_{j^*}, a valid crafted incremental proof would assert that τ_k is consistent with τ'_{j^*}. However, τ_k is generated by the users of the first fork and hence complies with the history represented by τ_{j^*} (but not τ'_{j^*}). This means that if users of the second fork verify τ_k as a valid tree digest, then τ_k must represent two different sequences of posts, a sequence with op_{j^*} and the other one with op'_{j^*}. This indicates that for an index h where $j^* \leq h \leq k$ the tree digests τ_h and τ'_h, generated by the first and second fork, respectively, collide, i.e., a collision is found for the underlying hash function.

Based on the above arguments, breaking the q-detectable consistency is equivalent to breaking the security of the underlying signature scheme or finding a collision in the hash function.

Note that the q-Detectable consistency experiment involves only one shared object yet can be further extended to include multiple objects shared across overlapping sets of users. It is straightforward to prove that Integrita provides q-detectable view-consistency under the extended experiment as well. For the sake of simplicity, we only sketch the proof idea of the extended game and eliminate the details. Essentially, the accessibility of the adversary to more than one shared object does not give it the advantage to win the game. This is because every two distinct shared objects e.g., group pages always differ in their first operation which is the insertion of the title of the group concatenated with the time of creation (the time and title might be encrypted though it does not affect our argument). Thus the history tree of each group page has a unique value for its first node (unless two group pages with identical names get created simultaneously, which is unlikely, or even in that case more metadata can be included in the first operation like the page owner, etc.). As such, two distinct shared objects can be seen as two forks of the same shared object where the fork has happened since the very first operation. As such, their operations are not exchangeable at any point in the future. This non-exchangeability of the forked views has been extensively discussed in the formal proof of the regular q-Detectable consistency game and relies on the fact that forked views never get to converge unless the adversary manages to forge signatures or find a collision in the hash function. As such, Integrita protects q-detectable consistency in the extended experiment with multiple shared objects as well.

In the following, we provide the formal security proof of Integrita based on q-Det-Cons(1^λ) . The hash function is assumed to be collision-resistant hence the security proof is tied to the security of the signature scheme. The proof leverages the following lemma that is due to [15].

Lemma 2. Given that the hash function that underlies the history tree is collision-resistant, if there is a valid incremental proof between two tree digests τ_i and τ_j, then for every operation op_k where $k < i$ for which there is a valid membership $proof$, s.t. $True \leftarrow MEMBERSHIP.VF(k, \tau_i, op_k, proof))$, and op'_k s.t. there is a $proof'$ for which $True \leftarrow MEMBERSHIP.VF(k, \tau_j, op'_k, proof))$, then op_k must be equal to op'_k. Namely, if two tree digests are consistent, then they both represent the same sequence of operations for their shared past [15].

Formal Proof of Theorem 1: If there exists a PPT adversary $\mathcal{A}$ who wins q-Det-Cons(1^λ) with non-negligible probability ϵ, then we construct a PPT simulator B who finds a forgery for the underlying signature scheme. The internal code of B is given below. B is given the security parameter 1^λ as well as a signature verification key vk' from the challenger of the signature scheme.

1) B gives the security parameter 1^λ to the adversary. B runs the signature key generation algorithm for the honest server as $(sk_{S_{Chal}}, vk_{S_{Chal}}) \leftarrow Gen(1^\lambda)$ and hands the $vk_{S_{Chal}}$ to the adversary. B selects the index of a user randomly i.e., $\beta \leftarrow [1, T]$ and sets the signature verification key of U_β to vk' whereas for the rest of users, B generates the signature key pairs as normal. Essentially, at this step, B guesses the user U_β for which the adversary may successfully forge a signature, hence B can leverage the adversary's success probability to win the signature unforgeability game. B sends $FList = \{(U_1, vk_1), \cdots, (U_\beta, vk'), \cdots, (U_T, vk_T)\}$ to the adversary. The adversary communicates a set of signature verification keys for the corrupted servers $\{vk_{S_i}\}_{i \in \mathcal{A}}$ to B.
2) The adversary specifies a user U_i as the group administrator to initiate the shared *object* through the invocation of *Create object* protocol. The adversary also determines the content of the first post, i.e., title, date, and time. The challenger performs as indicated in Figure 6. If the designated group administrator is $i = \beta$ (for which the challenger does not have the signature key), then to generate the required signatures, B queries the signing oracle of the signature challenger and stores the set of queried messages and signatures in a $QSign$ set. Otherwise, B acts as in the *Create object* protocol.
3) The adversary specifies a user U_i to carry out a write operation op on the *object*. B runs the $Write$ protocol accordingly.
   First, B runs the Update Status and fetches the latest tree digest τ_{p^*}. As the result of running Update Status, B fetches an incremental $proof = \{(N_{p,l}, op, \sigma_{U_k})\}$ for some p and l and k. B runs Algorithm 4 on the $proof$ to identify and output a signature forgery to the signature challenger, if any. During this procedure, B checks whether there is a node in the $proof$ that is signed by U_β but was never generated by her i.e., $\notin QSign$; which means that signature must have been forged by the adversary.
   B fetches the required nodes for the insertion of the new operation as $proof = \{(N_{p,l}, op, \sigma_{U_k})\}$. B runs

Algorithm 4 and if a successful signature forgery is found, returns it to the signature challenger.

Algorithm 4 Signature forgery identification. This process checks whether there is a node in the $proof$ that is signed by U_β but was never generated by her; which means that signature must have been forged by the adversary.

```
    Input: $proof = \{(N_{p,l}, op, \sigma_{U_k})\}$, $QSign$
1:  for $N_{p,l} \in proof$ do
2:      if $N_{p,l}$ is a leaf node OR $N_{p,l}$ is a tree digest then
3:          if $Vrfy_{vk_{U_\beta}}(N_{p,l}||p, \sigma_{U_k}) == accept$ AND $N_{p,l}||p \notin QSign$ then
4:              Return $(N_{p,l}||p, \sigma_{U_\beta})$
```

B recalculates the nodes on the insertion path of her operation. B signs the leaf node $H(op)||p^* + 1$ and the tree digest τ_{p^*+1} using the signature key of U_i. If $i = \beta$, then B queries the signing oracle and inserts the queried message and the obtained signature to $QSign$. If the $Write$ protocol terminates with abort, then B immediately aborts; in which case the adversary loses. Note that after each write operation, B shall act upon the audit procedure, i.e., B runs the Update Status protocol at the $p^* + Q(p^*, N)^{th}$ version (or some later version) of the *object*. As the result of running Update Status, B pulls some of the tree nodes as $proof = \{(N_{p,l}, op, \sigma_{U_k})\}$. B performs Algorithm 4 on the $proof$ to find any signature forged by the adversary, in which case B outputs the forgery to the signature challenger. If the Update Status results in an abort, then B aborts.

4) The adversary specifies a range $R = [l, r]$ to be read by a particular user U_i. B runs the *Read* protocol accordingly. B aborts in case that the *Read* protocol concludes with abort. Otherwise, as the result of running *Read* protocol, B pulls some of the nodes of the tree as $proof = \{(N_{p,l}, op, \sigma_{U_k})\}$. B performs Algorithm 4 on the $proof$ to identify any signature forged by the adversary, in which case B outputs the forgery to the signature challenger.
5) The adversary specifies two users U and U', a version number j, and a operation index j^* where $j^* < j - \Delta(j, N)$. B runs the *Read* protocol for U and U' separately. B acts identical to the step 4 (above) to run the $Read$ protocol. If the protocol concludes with abort, then B also aborts. Otherwise, B proceeds as follows. Let τ_j and τ'_j indicate the tree digests inside the Status variable of U and U' after the $Read$ protocol execution. Also let op_{j^*} and op'_{j^*} indicate the content of the i^{th} operation that are read by U and U', respectively. If $op_{j^*} \neq op'_{j^*}$ then B finds a signature forgery as we discuss below.

Note that the inconsistency between op_{j^*} and op'_{j^*} means that there will be two different tree digests τ_{j^*} (with op_{j^*} as its i^{*th} operation) and τ'_{j^*} (with op'_{j^*} as its i^{*th} operation). As such, from version j^* onward, the users will be divided into two groups G and G' depending on whether they are shown op_{j^*} (τ_{j^*}) or op'_{j^*} (τ'_{j^*}). More precisely, a group G of users whose further Status variables (i.e., τ_f where $f \geq j^*$) are consistent with τ_{j^*} (i.e., $\tau_{j^*} \rightarrow \tau_f$) and the other group G' whose further Status variables (i.e., τ'_f where $f \geq j^*$) are consistent with τ'_{j^*} (i.e., $\tau'_{j^*} \rightarrow \tau'_f$).

Since the users are divided in two groups G and G', there will be two separate chains of operations (after the i^{*th} operations) generated by group G and G', i.e., op_i $i \in [j^*, j]$ uploaded by group G and op'_i $i \in [j^*, j]$ performed by users of group G'. Assume $k \in [j^*, j^*+Q(i, N)]$ is the index of the frozen node (a node of the tree for which the honest server stores one of the nodes along its insertion path). Assume that a user from group G attempts the insertion of op_k earlier than a user from a group G'. Since the honest server appears on the insertion path of op_k, it gets informed about the inclusion of the k^{th} operation and updates its $Status$ accordingly. When a user from the group G' holding a Status variable τ'_i wants to insert op'_k, it first runs the Update Status to fetch the latest version of the *object* and perform consistency check between τ'_i and the current version of the *object*. During the status update protocol, the adversary may try to act dishonestly which we discuss next.

1) The adversary may attempt to send an incorrect Status value to the user and make her accept a lower version $< k$ of the *object*. However, due to the presence of the honest server (who has witnessed the insertion of op_k), the adversary does not succeed as the honest server will communicate its intact Status value, i.e., k with the user.
2) The adversary may attempt sending a Status value x, where $x \geq k$, for which the adversary also needs to come up with a valid tree digest τ'_x where $\tau'_x \implies \tau'_i$[9] (τ'_i is the Status of the user from group G' while inserting op'_k) in order to pass the Update Status protocol successfully. To come up with a valid τ'_x, the adversary has the following choices:
    a) The adversary may use the tree digest τ_x that is signed and generated by one of the members of the group G. However, any tree digest τ_x generated by a member of group G will be consistent with τ_{j^*} but not with τ'_{j^*}, i.e., $\tau_{j^*} \not\Longrightarrow \tau_x$. This means that there will be no valid incremental proof between τ'_{j^*} and τ_x.
    b) The other choice for the adversary is to generate a op'_x and forge a signature on $H(op'_x||x)$ (the leaf node) on behalf of an authorized user.
    c) The adversary uses op_x generated by one of the members of G and computes the tree digest τ'_x accordingly. $\mathcal{A}$ also needs to generate a valid signature over τ'_x on behalf of the user who issued op_x.

    The above argument indicates that for a member of group G' to accept that the latest version of *object* as x where $x \geq k$ and successfully pass the Update Status protocol, the adversary needs to forge a signature on behalf of an authorized user U'' either on the leaf node $H(op'_x||x)$ or the tree digest $\tau'_x||x$. Thus, B shall figure out this forgery while fetching the incremental proof on behalf of a member of the group G'.

B can win the signature game if the forgery of the adversary is from U_β. Recall that the probability of $\mathcal{A}$ winning the q-Det-Cons(1^λ) is $\epsilon(\lambda)$ and the total number of users T

9. We write $\tau_i \implies \tau_j$ to indicate that there exists an incremental $proof$ for which $IncrVf(\tau_i, \tau_j, proof))$ returns $True$.

is polynomial in the security parameter (i.e., $poly(\lambda)$). Thus, we have

$$\begin{aligned}
&Pr[B \text{ breaks the signature scheme}] = \\
&Pr[\text{q-Det-Cons}(1^{\lambda}) \ = 1 \text{ AND } U^{''} = U_{\beta}] \\
&= Pr[\text{q-Det-Cons}(1^{\lambda}) \ = 1 | U^{''} = U_{\beta}] \cdot Pr[U^{''} = U_{\beta}] \\
&\geq \epsilon(\lambda) \cdot \frac{1}{T} \\
&= \epsilon(\lambda) \cdot \frac{1}{poly(\lambda)} \qquad (14)
\end{aligned}$$

If $\epsilon(\lambda)$ is non-negligible, then B also breaks the signature scheme with non-negligible probability. Since the signature scheme is assumed to be existentially unforgeable under adaptive chosen message attacks, then Integrita provides q-detectable consistency. ■

9 Conclusion

In Integrita, we address the view consistency in a collaborative data-sharing environment such as Facebook group pages and walls. The shared object is comprised of a sequence of posts that can be generated by any of the authorized users. The view consistency concerns that all the authorized users are shown the same set of posts and with the intact order. To accomplish this, the log of operations performed on the shared object (called activity log) is modeled by a history tree, which is an append-only data structure. Operations constitute the insertion and deletion of posts. Having access to the intact activity log naturally guarantees users' view-consistency. Our design relies on the federation of N servers.

In Integrita, we introduce a new level of consistency called *q-detectable consistency* where any inconsistency between users' view (toward the activity log) cannot remain undetected for more than q operations. The q-detectable consistency holds as long as one server does not collude with the rest of the servers. The value of q is quantified based on the number of operations in the activity log as well as the number of servers. Our proposal outperforms the state of the art in two major directions. First, unlike the replication-based solutions, Integrita operates only on one instance of the shared object that is maintained collaboratively by all the servers. As such, Integrita saves 2344 Terabytes of annual storage for a social network like Facebook with 2.3 billion users, when presumably running on the federation of 20 servers. This is enabled by trading the strong consistency with q-detectable consistency. Nevertheless, q-detectable consistency will ultimately converge toward strong consistency as the size of the activity log elevates. Moreover, unlike the centralized OSNs where the inconsistency detection relies on the users' direct communication, Integrita detects any fork in the users' views regardless of users' direct communication.

As future work, Integrita can be extended to further support q-detectable view-consistency in a malicious adversarial model where a subset of users may get corrupted and conspire with the servers.

Acknowledgements

We acknowledge the support of the Turkish Academy of Sciences.

References

[1] Aldin, H. N. S., Deldari, H., Moattar, M. H., and Ghods, M. R. Strict timed causal consistency as a hybrid consistency model in the cloud environment. *Future Generation Computer Systems 105* (2020), 259–274.

[2] Ananth, P., Chandran, N., Goyal, V., Kanukurthi, B., and Ostrovsky, R. Achieving privacy in verifiable computation with multiple servers–without fhe and without pre-processing. In *International Workshop on Public Key Cryptography* (2014), Springer, pp. 149–166.

[3] Baden, R., Bender, A., Spring, N., Bhattacharjee, B., and Starin, D. Persona: an online social network with user-defined privacy. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication* (2009), pp. 135–146.

[4] Bethencourt, J., Sahai, A., and Waters, B. Ciphertext-policy attribute-based encryption. In *IEEE symposium on security and privacy (SP'07)* (2007), IEEE, pp. 321–334.

[5] Bielenberg, A., Helm, L., Gentilucci, A., Stefanescu, D., and Zhang, H. The growth of diaspora-a decentralized online social network in the wild. In *2012 Proceedings IEEE INFOCOM Workshops* (2012), IEEE, pp. 13–18.

[6] Boshrooyeh, S. T., Küpçü, A., and Özkasap, Ö. PPAD: Privacy Preserving Group-Based ADvertising in Online Social Networks. In *IFIP Networking 2018* (Zurich, Switzerland, 2018), IEEE, pp. 541–549.

[7] Boshrooyeh, S. T., Küpçü, A., and Özkasap, Ö. Privado: Privacy-preserving group-based advertising using multiple independent social network providers. *ACM Transactions on Privacy and Security (TOPS) 23*, 3 (2020), 1–36.

[8] Brandenburger, M., Cachin, C., and Knežević, N. Don't trust the cloud, verify: Integrity and consistency for cloud object stores. *ACM Transactions on Privacy and Security (TOPS) 20*, 3 (2017), 8.

[9] Buchegger, S., Schiöberg, D., Vu, L.-H., and Datta, A. Peerson: P2p social networking: early experiences and insights. In *Proceedings of the Second ACM EuroSys Workshop on Social Network Systems* (2009), ACM, pp. 46–52.

[10] Cachin, C., Keidar, I., and Shraer, A. Fork sequential consistency is blocking. *Information Processing Letters 109*, 7 (2009), 360–364.

[11] Cachin, C., and Ohrimenko, O. Verifying the consistency of remote untrusted services with conflict-free operations. *Information and Computation 260* (2018), 72–88.

[12] Chuat, L., Szalachowski, P., Perrig, A., Laurie, B., and Messeri, E. Efficient gossip protocols for verifying the consistency of certificate logs. In *2015 IEEE Conference on Communications and Network Security (CNS)* (2015), IEEE, pp. 415–423.

[13] Civit, P., Gilbert, S., and Gramoli, V. Polygraph: Accountable byzantine agreement. *IACR Cryptology ePrint Archive* (2009), 587.

[14] Cramer, R., Damgård, I., and Nielsen, J. B. Multiparty computation from threshold homomorphic encryption. In *International Conference on the Theory and Applications of Cryptographic Techniques* (2001), Springer, pp. 280–300.

[15] Crosby, S. A., and Wallach, D. S. Efficient data structures for tamper-evident logging. In *USENIX Security Symposium* (2009), pp. 317–334.

[16] Damgård, I. B. A design principle for hash functions. In *Conference on the Theory and Application of Cryptology* (1989), Springer, pp. 416–427.

[17] Eskandarian, S., Messeri, E., Bonneau, J., and Boneh, D. Certificate transparency with privacy. *Proceedings on Privacy Enhancing Technologies 2017*, 4 (2017), 329–344.

[18] Etemad, M., and Küpçü, A. Efficient key authentication service for secure end-to-end communications. In *International Conference on Provable Security* (2015), Springer, pp. 183–197.

[19] Etemad, M., and Küpçü, A. A generic dynamic provable data possession framework. *IACR Cryptology ePrint Archive 2016* (2016), 748.

[20] Etemad, M., and Küpçü, A. Verifiable database outsourcing supporting join. *Journal of Network and Computer Applications 115* (2018), 1–19.

[21] Feldman, A. J., Blankstein, A., Freedman, M. J., and Felten, E. W. Privacy and integrity are possible in the untrusted cloud. *IEEE Data Eng. Bull. 35*, 4 (2012), 73–82.

[22] Feldman, A. J., Blankstein, A., Freedman, M. J., and Felten, E. W. Social networking with frientegrity: privacy and integrity with an untrusted provider. In *Presented as part of the*

21st {USENIX} Security Symposium ({USENIX} Security 12) (2012), pp. 647–662.

[23] Feldman, A. J., Zeller, W. P., Freedman, M. J., and Felten, E. W. Sporc: Group collaboration using untrusted cloud resources. In *OSDI* (2010), vol. 10, pp. 337–350.

[24] Goldwasser, S., Micali, S., and Rivest, R. L. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing 17*, 2 (1988), 281–308.

[25] Goodrich, M. T., Lentini, J., Shin, M., Tamassia, R., and Cohen, R. Design and implementation of a distributed authenticated dictionary and its applications. Tech. rep., Technical report, Center for Geometric Computing, Brown University, 2002.

[26] Goodrich, M. T., and Tamassia, R. Efficient authenticated dictionaries with skip lists and commutative hashing, Aug. 14 2007. US Patent 7,257,711.

[27] Goodrich, M. T., Tamassia, R., and Hasić, J. An efficient dynamic and distributed cryptographic accumulator. In *International Conference on Information Security* (2002), Springer, pp. 372–388.

[28] Graffi, K., Gross, C., Stingl, D., Hartung, D., Kovacevic, A., and Steinmetz, R. Lifesocial. kom: A secure and p2p-based solution for online social networks. In *2011 IEEE Consumer Communications and Networking Conference (CCNC)* (2011), IEEE, pp. 554–558.

[29] Han, S., Shen, H., Kim, T., Krishnamurthy, A., Anderson, T., and Wetherall, D. Metasync: File synchronization across multiple untrusted storage services. In *2015 {USENIX} Annual Technical Conference ({USENIX} {ATC} 15)* (2015), pp. 83–95.

[30] Hsu, T.-Y., Kshemkalyani, A. D., and Shen, M. Causal consistency algorithms for partially replicated and fully replicated systems. *Future Generation Computer Systems 86* (2018), 1118–1133.

[31] Jakobsen, T. P., Nielsen, J. B., and Orlandi, C. A framework for outsourcing of secure computation. In *Proceedings of the 6th edition of the ACM Workshop on Cloud Computing Security* (2014), ACM, pp. 81–92.

[32] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and Van Steen, M. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS) 25*, 3 (2007), 8.

[33] Katz, J., and Lindell, Y. *Introduction to modern cryptography*. CRC press, 2014.

[34] Kurosawa, K., and Desmedt, Y. A new paradigm of hybrid encryption scheme. In *Annual International Cryptology Conference* (2004), Springer, pp. 426–442.

[35] Langley, A., Kasper, E., and Laurie, B. Certificate transparency. *Internet Engineering Task Force (IETF)* (2013).

[36] Laurie, B., Langley, A., and Kasper, E. Certificate transparency. *ACM Queue 12*, 8 (2014), 10–19.

[37] Li, J., and Maziéres, D. Beyond one-third faulty replicas in byzantine fault tolerant systems. In *NSDI* (2007).

[38] Loupasakis, A., Ntarmos, N., Triantafillou, P., and Makreshanski, D. exo: Decentralized autonomous scalable social networking. In *CIDR* (2011), pp. 85–95.

[39] Mahajan, P., Setty, S., Lee, S., Clement, A., Alvisi, L., Dahlin, M., and Walfish, M. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems (TOCS) 29*, 4 (2011), 12.

[40] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., and Stubblebine, S. G. A general model for authenticated data structures. *Algorithmica 39*, 1 (2004), 21–41.

[41] Mazieres, D., and Shasha, D. Building secure file systems out of byzantine storage. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing* (2002), ACM, pp. 108–117.

[42] Narendula, R., Papaioannou, T. G., and Aberer, K. Privacy-aware and highly-available osn profiles. In *2010 19th IEEE International Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises* (2010), IEEE, pp. 211–216.

[43] Nilizadeh, S., Jahid, S., Mittal, P., Borisov, N., and Kapadia, A. Cachet: a decentralized architecture for privacy preserving social networking with caching. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), ACM, pp. 337–348.

[44] Palazzi, B. *Outsourced storage services: Authentication and security visualization*. PhD thesis, Ph. D. thesis, Roma Tre University, 2009.

[45] Polivy, D. J., and Tamassia, R. Authenticating distributed data using web services and xml signatures. In *XML Security* (2002), pp. 80–89.

[46] Ryan, M. D. Enhanced certificate transparency and end-to-end encrypted mail. In *NDSS* (2014), pp. 1–14.

[47] Schoenmakers, B., and Veeningen, M. Universally verifiable multiparty computation from threshold homomorphic cryptosystems. In *International Conference on Applied Cryptography and Network Security* (2015), Springer, pp. 3–22.

[48] Shakimov, A., Lim, H., Cáceres, R., Cox, L. P., Li, K., Liu, D., and Varshavsky, A. Vis-a-vis: Privacy-preserving online social networking via virtual individual servers. In *Third International Conference on Communication Systems and Networks* (2011), IEEE, pp. 1–10.

[49] Son, S. H. Semantic information and consistency in distributed realtime systems. *Information and Software Technology 30*, 7 (1988), 443–449.

[50] Strufe, T. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine 95* (2009).

[51] Studi, P., Mohomed, I., Balakrishnan, M., Mao, Z. M., Ramasubramanian, V., Terry, D., and Wobber, T. Contrail: Enabling decentralized social networks on smartphones. In *ACM/IFIP/USENIX International Conference on Distributed Systems Platforms and Open Distributed Processing* (2011), Springer, pp. 41–60.

[52] Tamassia, R. Authenticated data structures. In *European symposium on algorithms* (2003), Springer, pp. 2–5.

[53] Tang, Y., Sun, H., Wang, X., and Liu, X. Achieving convergent causal consistency and high availability for cloud storage. *Future Generation Computer Systems 74* (2017), 20–31.

[54] Tegeler, F., Koll, D., and Fu, X. Gemstone: empowering decentralized social networking with high data availability. In *2011 IEEE Global Telecommunications Conference-GLOBECOM 2011* (2011), IEEE, pp. 1–6.

[55] Williams, P., Sion, R., and Shasha, D. E. The blind stone tablet: Outsourcing durability to untrusted parties. In *NDSS* (2009), Citeseer.

[56] Yuan, L., McNally, D., Küpçü, A., and Ebrahimi, T. Privacy-preserving photo sharing based on a public key infrastructure. In *Applications Of Digital Image Processing XXXVIII* (2015), vol. 9599, International Society for Optics and Photonics, p. 95991I.

[57] Zhao, X., and Haller, P. Replicated data types that unify eventual consistency and observable atomic consistency. *Journal of Logical and Algebraic Methods in Programming* (2020), 100561.