Secure Replication for Client-centric Data Stores

Anonymous Author(s)

Abstract

Decentralized, peer-to-peer systems using Conflict-free Replicated Data Types (CRDTs) can offer a more privacy-friendly alternative to centralized solutions that are often used by Big Tech. However, traditional CRDTs assume that all replicas are trusted, which is not necessarily the case in a peer-to-peer system. This paper presents a protocol for secure state-based CRDTs which provide fine-grained confidentiality and integrity by using encryption per field in every (sub)-document. Our protocol guarantees Strong Eventual Consistency despite any Byzantine replicas. It provides a fine-grained, dynamic membership and key management system, without violating Strong Eventual Consistency or losing concurrent updates. Our evaluation shows that the protocol is suitable for use in interactive, collaborative applications.

1 Introduction

In the last decade, personal data has been stored in the cloud, rather than on a local computer. From many perspectives, this is beneficial for end-users. Data is accessible everywhere and collaboration with anyone in the world is made easy. Users also do not need to worry about data loss due to malfunction, or security breaches. However, the reality today often does not match this ideal. Few large tech companies and governments have access to vast amounts of data. They can potentially misuse it and invade the privacy of their customers or citizens to gain more money or harm political dissidents. Moving to another vendor is often very hard, if not impossible. The data is also not secure, as we hear about new security breaches almost every month, and most breaches probably even go undetected.

One solution is to move to a more decentralized and client-centric approach [2, 5]. The primary copy of the data is stored under the control of the user on their local device. Data can then be replicated peer-to-peer to all other user devices and collaborators. However, a true peer-to-peer approach of end-user devices is not very durable and available. Devices are often not online at the same time, do not have a large amount of storage space, and can fail more easily or more frequently compared to a server inside a data center.

Having some kind of centralized server can be beneficial to aid the client-centric vision. The server is most of the time online, and all clients can use this server to replicate their data to each other. Even when they are never online simultaneously. Ideally, this server does not belong to a big-tech company but is under the control of the end user. One such approach is the Solid Platform [8]. With Solid, every person manages their own Personal Online Datastore (pod), either self-hosted or hosted with a third party pod-provider. Each application will store all user data inside the user's pod, and the user is in control to decide who has access to it. This also makes it easy to switch to a different application. However, the majority of the users will not choose to host their pod themselves. Instead, they will rely on a third-party company or the government to provide them with a pod. This might lead to an even bigger problem of surveillance capitalism, where few companies provide pods to their customers and gain immediately access to even more data. These providers with all data of a large number of users will also be an interesting target for hackers.

The solution we propose is a hybrid approach of a peer-to-peer network of mostly client devices and some centralized servers to improve availability and durability. An example is shown in Figure 1. While we trust the centralized server to keep data available, we do not want them to read or modify the actual data. A similar durability can be reached by creating a larger peer-to-peer network with friends or family, and replicating all data between all these devices. However, it should be avoided that peers are able to look into all personal data of other peers. A secure replication protocol without having access to the plain data is required.

In such systems, eventual consistency is the most pragmatic and only viable option. As devices are often offline, reaching a global consensus to have strong consistency would be nearly impossible or beyond a user-friendly time window. With strong consistency, making updates on an offline device would be impossible, and latency will be bad as clients are often only connected via WiFi or a mobile network. By opting for eventual consistency, we need a way to make sure all replicas converge to the same state after they have received all operations. One option is to use Conflict-free Replicated Data Types (CRDTs) [11]. CRDTs are data structures that guarantee eventual consistency without explicit coordination. However, classical CRDTs do not encrypt their data

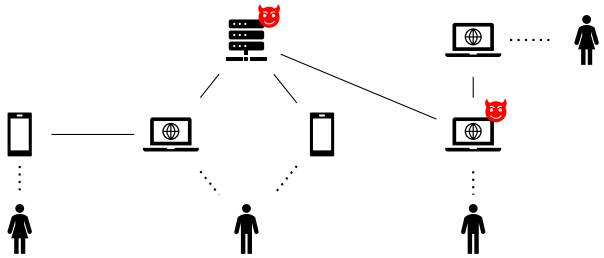


Figure 1. Hybrid architecture of a peer-to-peer network with a centralized server. Some users have one device, while others have multiple. Some devices have access to the server, others connect peer-to-peer. Some devices can be malicious.

and are not resilient against an attacker trying to prevent convergence.

In this paper, we present a secure state-based CRDT protocol that extends classical state-based CRDTs with:

- Fine-grained encryption per field in every (sub)-document, to preserve confidentiality and integrity of all user data,
- Byzantine Fault Tolerance, to guarantee Strong Eventual Consistency even with Byzantine parties,
- Dynamic membership and fine-grained key management, without breaking Strong Eventual Consistency, leaking extra data, or losing updates.

Compared to other state-of-the-art approaches for secure CRDTs [1, 4], we provide the first framework to allow both concurrent data updates, as well as concurrent updates to the access control policy. This means that a user can share a document, or revoke access to a document, without losing concurrent updates to that document.

Being able to change the encryption key to give or revoke access, or to rotate the encryption key when it might be compromised is especially important for collaborative applications. For a single user, that user can easily coordinate a key rotation by bringing all his devices together, halting the system, and updating the key. However, for collaborative applications with several users, this process should be done online, without halting the system or explicit coordination between all collaborators. The protocol presented in this paper supports this.

This paper is structured as follows. Section 2 describes the system- and adversary-model. We explain our protocol for secure CRDTs in Section 3. We evaluate our protocol in Section 4. Section 5 presents related work. We conclude in Section 6.

2 System model

In this paper, we consider a peer-to-peer network of replicas connected by an asynchronous network (Figure 1). Replicas do not have a direct connection to every other replica, and they do not necessarily know the full set of replicas. Messages can be delayed, dropped, or delivered out of order, but eventually, some messages will be received. Honest replicas will follow the protocol exactly, Byzantine replicas can behave arbitrarily. There is no limit on the number of Byzantine replicas. Every user has an asymmetric key-pair, and other users are able to retrieve the public key of other users in a secure way, outside our protocol. We assume attackers are computationally bounded and it is infeasible to reverse the used symmetric encryption without the secret, forge the used asymmetric signatures or find collisions for the used cryptographic hash functions.

Given this system model, our protocol provides the following properties in the face of an active adversary:

- **Confidentiality**: Only users who have been given access can read the content.
- **Integrity**: Only users who have been given access can edit the content.
- **Attributability**: Each edit is attributable to the user who made the modification.
- **Availability**: As long as at least two honest replicas are available, they can work together and replicate correctly between each other.
- **Eventual delivery**: An update delivered at some correct replica is eventually delivered to all correct replicas.
- **Strong convergence**: Correct replicas that have delivered the same updates have equivalent state.
- **Termination**: All method executions terminate.

The last three properties together deliver Strong Eventual Consistency [11]. All the properties are kept intact, even when the adversary has been given access to the actual content. The adversary is then able to arbitrarily change the content, in a way that might not make sense for the application or end-user. However, all replicas will still converge to the same end-state, and the bad updates will be attributable to the Byzantine user. The other users can then decide to revoke access if necessary.

3 Secure CRDTs

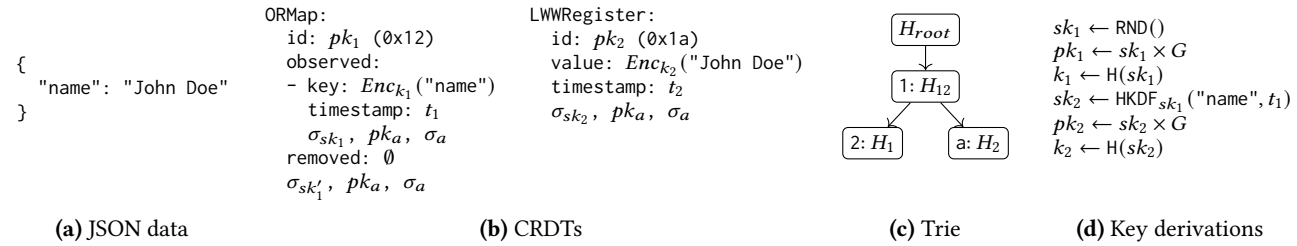
This section explains the protocol for our secure CRDTs. We use the term key to refer to a key from a key-value pair. When we are referring to cryptographic keys, we will always specify this as a *secret key* (k) for symmetric encryption and as a *private key* (sk) or *public key* (pk) for asymmetric encryption or signatures.

3.1 Encrypted CRDT

We now present two encrypted CRDT protocols. These two data structures are enough to encode a JSON tree with only maps and values into a CRDT. Arrays are not yet supported. Figure 2a shows an example of a JSON document and Figure 2b shows how it will be represented internally by the protocol explained in this section.

State-based CRDTs have a merge-function, which takes as input two states of the same CRDT and produces a new state. Mathematically, these states form a join semi-lattice, and the resulting state of the merge function is the smallest state that is larger or equal to the two input states according to the partial order of the lattice. To replicate this data structure, a replica needs to send its state to another replica. This receiving replica can use the merge function with its local state and the received state to end up with the merged state.

Each CRDT is associated with an asymmetric key-pair. The public key is included in the CRDT and also functions as unique ID to reference the CRDT. The private key is only shared with users who have read-write access to the data.



(a) JSON data **(b)** CRDTs **(c)** Trie **(d)** Key derivations

Figure 2. Example of how a JSON data structure can be translated into a secure CRDT data structure, consisting of two CRDTs. These CRDTs are put inside the Modified Merkle-Patricia Trie. At the right, we show how keys can be derived starting from one root key: sk_1.

LWWRegister. A LWWRegister [11] is a data structure that holds one single value. When updating the value, the new value is associated with the current timestamp. Conflicts are resolved by selecting the value associated with the highest timestamp. If the timestamps are equal, the value with the lexicographically largest hash value will be selected. Since the actual value is not used to perform a merge, the value can be encrypted using any symmetric encryption protocol, and the resulting data structure is still a CRDT.

ORMap. An ORMap [11] is a data structure that holds a mapping of keys to values. In practice, it consists of two sets: the observed-set and the removed-set. When a new key-value pair is added to the ORMap, it is associated with a unique ID and added to the observed-set. When removing the key-value pair, it is added to the removed set. The key-value pairs included in the ORMap are all pairs included in the observed-set, which are not present in the removed-set. Thanks to the unique ID, it is possible to remove an item and add it again later. In our protocol, the values are references to other CRDTs: either a LWWRegister or another ORMap. The keys, however, need to be encrypted to maintain confidentiality. The unique ID also has to be protected against Byzantine replicas [4]. If the replica itself is responsible for generating a new random ID, a Byzantine replica can easily generate duplicate IDs. Therefore, we will generate IDs deterministically based on the update. The ID of a new key-value pair is derived from the secret key linked to the ORMap and the key from the key-value pair. Since it must be possible to remove and add a key-value pair again, we also add a timestamp to each key-value pair. This timestamp is also used as input to derive the ID. There is no need to store this ID, every replica that has access to the secret key can compute the ID itself. Since the IDs and keys are therefore only available to replicas that have access to the secret key, replicas without access do not know when two items have the same ID or key, and they will therefore not be able to propagate the merge to the child CRDTs. Instead, two copies of these similar key-value pairs will be stored in the ORMap, and any other replica which does have access to the secret key can perform the merge later. This derived ID is also the value of the key-value pair: i.e., it is the ID of the child CRDT. Since this ID is only available to replicas with access to the secret key, the structure of the data is also hidden from replicas that do not have access. This means that a replica that has no access at all, will only be able to see how many individual ORMaps and LWWRegisters there are, without knowing how they belong together.

Signatures. Only users who have been given access to the secret key should be able to modify data. For this reason, every update has to be signed by the private key of the CRDT. For a LWWRegister one signature is sufficient. An ORMap will have one signature per key-value pair in the observed- and removed-set. Since the public key is also included in the CRDT, anyone can verify that an update came from a party with access to the private key. These signatures also ensure it is safe to use a public key as ID for the CRDT in a context with Byzantine actors. You cannot reuse the same ID if you do not have access, and if you do have access, using the same ID will lead to a merge of those two CRDTs. This is equivalent to a write to the first CRDT, which you are allowed to do as you do have access to the private key.

Each update is also signed by the private key of the user who makes the update. This way, each update is attributable to the user who made the edit. The first signature (σ_{sk} in Figure 2b) with the private key of the CRDT proves that you have the right to modify it, the second signature (σ_a in Figure 2b) with your own private key proves who you are. If attributability is not required, it is possible to leave out the second signature with no other changes to the protocol.

3.2 Modified Merkle Patricia Trie

All individual CRDTs are stored inside a Modified Merkle Patricia Trie [13] (Figure 2c). A Patricia Trie is a tree-shaped data structure in which items associated with a key with a common prefix, will share the same path in the tree for that prefix. A Merkle-tree [9] is a tree-shaped data structure of hashes, in which the hash of a parent node is based on the hash of the hashes of the child nodes. This way, large data structures can quickly be compared or verified based on the hash in the root node of the tree. A Modified Merkle Patricia

Trie combines both a Patricia Tree and a Merkle-tree. Each node in the trie also carries a hash value. This data structure is also used by Ethereum to store the state of the Ethereum blockchain [13].

The key to insert a CRDT into the trie is the ID of the CRDT. Since the ID is also a public key, they are random and therefore the trie will be relatively well-balanced. By using the Merkle-tree, two replicas can efficiently exchange updates between each other. The replicas can compare the root hash of the trie. If the hashes match, the two tries are exactly the same, and no replication is required. If the hashes do not match, the replicas will descend in the tree and send the hashes of the next level in the tree. This process continues until it reaches the leaves of the tree. At this time, the updated CRDTs can be sent and merged. This process is similar to the replication process in OWebSync [3].

3.3 Key derivation and rotation

In the previous two sub-sections, we created a trie of individual CRDTs which contain signatures and are partially encrypted by the respective private and secret key of the CRDT. The secret key can be derived from the private key by using a key derivation function, for example HKDF [7]. This leads to one encryption key to manage per CRDT. However, as already indicated in the paragraph on ORMaps, the key material for children is derived from the parent key. Instead of directly deriving the ID for a key-value pair in an ORMap, we derive a private key. We can then use this private key to derive the secret key and public key. This public key is also the ID. A user who has access to the full document tree only needs access to the private key of the root and can derive all other keys from this single key. This makes sharing a document and key management easy. An example of this derivation process is shown in Figure 2d.

When access is revoked from a user, the encryption key will have to be updated. Otherwise, that user still has access to the secret key, and would still be able to read and write. A new private key is derived from the parent private key, the key (from the path in the tree), and the current timestamp. Because the timestamp will be different, a brand new private key is generated and the CRDT can be re-encrypted with the corresponding secret key. Since the private key is changed, the public key will also be different and the CRDT will be stored under a different ID in the trie.

Because these CRDTs end up in different places of the trie, they can co-exist for while. This means that replicas that are not yet informed about the key rotation can still perform updates on the old version, while other replicas can do updates on the new version. Any replica that has access to both the old and the new version knows those two CRDTs are in fact the same CRDT and can perform a merge operation as usual. Replicas that do not have access to both private keys are unaware they belong to the same CRDT and will threat them as two separate CRDT structures.

3.4 Global time

Common wisdom in the field of distributed systems is that you cannot have a global time in a distributed system. While this is true, a course-grained global timestamp is still possible. The Ethereum blockchain, for example, includes a timestamp in every block header. In the Geth implementation, a timestamp of a new block has to be larger than the timestamp of the previous block and less than 15 seconds in the future of the current time of a replica. Similar timestamps and rules are present in other blockchains.

We use similar rules for the timestamps used in our protocol. A timestamp may only be at most one minute in the future, otherwise, the replica will not accept it and stop communication with the other replica. It is the task of the replicas to keep their clocks correct. These days, internet-connected devices automatically synchronize their time with an internet time server and are generally correct within one minute.

A Byzantine replica can re-use a timestamp without problem since the lexicographic order of the hash value will then be used as a tie-breaker. Such a replica can also get an edge over other replicas by always using a timestamp one minute in the future. Because its timestamps are generally larger, when an update is done simultaneously, its update is more likely to win in the last-writer-wins conflict resolution. This is however only possible for Byzantine replicas that have access to the private key, i.e., replicas with write access. Replicas without access can never change anything. Hence, such replicas do not get to choose a timestamp. This edge that a Byzantine replica has is only present for short intervals. On larger intervals, the correct user intention will be kept. For example, when user A makes a change in the morning, and another user B changes the same data in the afternoon, the change of user B will be chosen. User A can of course keep increasing the timestamp of his update, but this is equivalent to a new write by a replica that has write access, so this is allowed.

3.5 Discussion

This section presented a novel protocol for secure and confidential CRDTs. Since replication is state-based, there are no client-specific identifiers kept for the replication. Replicas do not need to know every other replica. Only the replica modifying the access control policy has to know the public key of all users with read and write access. This makes the protocol extremely robust against network failures and long-term disconnects [3]. Centralized servers which are only there to improve the availability and durability of the replication between clients, do not need any private key material to function.

The current protocol will keep both old and new versions of a CRDT after a key rotation forever. This is not required, once the new version is created, the old one can be removed. With concurrent edits, it is possible that the old version will

resurface again, but after each merge with the new version, it is removed again. After some time, all replicas will know about the key rotation and all updates will be applied to the new version and the old version will never resurface. If the replica that has been revoked access by the key rotation makes an update, it will not be merged in the new version, but simply be discarded. This is possible due to the course-grained timestamps. So, there is a small interval of less than a minute in which its updates will still be accepted. For most application cases that already opted for eventual consistency, this is acceptable.

To be able to determine whether an update from an old version of the CRDT should be merged with the new version, a list of all users having access to it is required. This is a list of public keys, and only needs to be kept at the point in the JSON tree where you give access to other users. This can be encrypted as well, as only replicas with access to the actual data will have to use the list to potentially merge data updates across key updates. Replicas that decide to rotate a key can also use this list to determine who should have access to the new key. Replicas with no access to the data do not use this list and instead rely on the key-pair of each CRDT to determine whether access was correctly granted.

The only cryptographic protocols used are plain symmetric encryption (e.g. AES), public-key cryptography (e.g. RSA or ECDSA), and hashing (e.g. SHA256). Furthermore, we use a key derivation algorithm based on these protocols (HKDF). As these are older, well-tested protocols, we can be more certain of their correctness and safety. There are also more well-tested and maintained libraries available, making it possible to implement our protocol in multiple languages. Also, the availability of hardware support for some of these will be good for the performance on client devices.

4 Evaluation

We implemented the protocol in a JavaScript-based web application, without browser plugins. For our experiments, we launched up to 30 virtual machines in the Azure public cloud (F2s_v2 with 2 vCPU and 4 GB RAM) in the same data center. To emulate geographically distributed users, we use the Linux `tc` tool to increase the network latency between each VM to an average of 100 ms with 50 ms jitter. Each VM contains one Chromium browser. Every client makes one write every second. We are interested in the interactive latency, i.e., after one client makes an update, how long does it take for other clients to receive it. To compare the overhead of our encrypted and Byzantine fault-tolerant approach to a regular approach without security, we also performed the same experiments with the open-source version of OWebSync. OWebSync [3] is a state-based CRDT framework, in which all clients are trusted.

The performance results are shown in Table 1. We compare the protocol from this paper (secure CRDTs) with OWebSync (baseline) for three different numbers of active replicas. With 30 different replicas, each making one request per second, the average latency is 1.6 seconds before an update is visible to other replicas. With smaller network sizes, the latency is lower. OWebSync has a much lower latency, of 0.5 seconds, even for the larger network sizes as no cryptographic operations are required there. Overall, the latency is low enough to be considered interactive when multiple users are collaboratively working on the same document. The storage overhead of the protocol ranges from 16 to 19 times, compared to the size of the raw data. For OWebSync this overhead is only 4 times. The overhead comes from the extra metadata required for state-based CRDTs, but also from the signatures and encrypted data. This leads to a bandwidth usage of 830 kbps for 30 replicas, which is readily available on any mobile network. Interestingly, the network usage for our baseline, OWebSync is a factor 5 higher, even though the actual storage size is much lower. The explanation for this is two-fold. First, OWebSync uses a Merkle-tree which is based on the actual tree structure of the data, while our protocol uses a much better balanced Merkle-Patricia Trie. This allows replicas to propagate updates more fine-grained, i.e., when only a leaf in the JSON data changes, we do not need to replicate the intermediate nodes of the JSON tree. Second, as the latency of OWebSync is much better, it does more traversals of the Merkle-tree, while our protocol does less as more updates can be batched in the same tree traversal given the higher latency. This second point also explains the discrepancy in CPU usage for the network with 30 replicas, as we would expect that our protocol always has a higher CPU usage compared to a solution without any signatures and cryptography.

Table 1. Performance characteristics of the proposed protocol, compared to a baseline protocol without any security.

# replicas		10	20	30
Latency [s]	secure CRDTs	0.62	0.76	1.59
	baseline	0.56	0.50	0.54
Storage overhead	secure CRDTs	×16	×16	×19
	baseline	×4	×4	×4
Bandwidth [kbps]	secure CRDTs	229	806	830
	baseline	231	1382	4160
CPU usage [%]	secure CRDTs	19	56	72
	baseline	13	42	83

To conclude, we have shown that our protocol for secure CRDTs, which tolerates Byzantine replicas, and which supports very fine-grained access control by encrypting every field in every (sub)-document with a different key, can be used for interactive, collaborative document editing. The price to pay is a significant increase in the size of the data (up to 19 times).

5 Related work

This section covers related work that also tries to reach eventual consistency in an adversarial context.

Snapdoc [6] is a collaborative peer-to-peer text editing protocol. New replicas can be added to the network by only sending a snapshot of the data, including a cryptographic proof of the integrity. They can keep the edit history private for new replicas, but new replicas can still attribute all changes, as well as verify the integrity. This is made possible by using RSA accumulators and Merkle-proofs. However, the new replica can only accept operations that are created after the snapshot. When an operation, not included in the snapshot, was created before or concurrent to a snapshot, the new replica will have to request a new snapshot and do the verification process again.

In [12], van der Linde et al. present a system that protects against rational misbehaving clients in causal consistency. However, servers are considered trusted, and the focus is on detecting the Byzantine client, rather than avoiding divergence at all.

In [1], Barbosa et al. extend standard CRDTs with cryptographic protocols. The paper focuses on a client-server context, where servers are unable to see the actual data. The same approach can most likely also be used in a peer-to-peer setting. However, the provided algorithms only work as long as the same cryptographic key is used. Switching to a new key will require coordination between the replicas. Furthermore, the approach focuses on confidentiality and does not tolerate an active Byzantine replica.

In [4], Kleppmann shows how operation-based CRDTs can be adapted to tolerate Byzantine replicas. The paper lists four techniques that are together sufficient to make most operation-based CRDT tolerate Byzantine replicas. The techniques are: constructing a hash-graph of all updates, with links to predecessor; ensuring eventual delivery, which could be done by using the hash graph; constructing unique IDs, which cannot be controlled by an attacker; and ensuring that replicas only look at the predecessors of an update to check the validity of it. However, the paper only focuses on maintaining eventual consistency, and not on confidentiality.

6 Conclusion and future work

In this paper, we presented a protocol for secure state-based CRDTs. We have shown that Strong Eventual Consistency can be reached even in settings with Byzantine replicas. We have also shown that a key rotation does not have to break Strong Eventual Consistency and that you can do this concurrently, while other users are still making updates with the old keys. The key idea to support this is to store all CRDTs inside a Merkle-Patricia Trie, and only allow replicas that have access to both the old and the new secret key to merge two different versions of the same CRDT.

In future work, we will extend this protocol with online pruning to remove old versions which are not necessary anymore from the trie. Arrays are also not yet supported. The current protocol uses basic, state-of-practice cryptography. More research is required to evaluate whether newer cryptography protocols such as attribute-based encryption [10] can offer any benefits.

References

[1] Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. 2021. Secure Conflict-Free Replicated Data Types. In *International Conference on Distributed Computing and Networking 2021 (ICDCN '21)*. ACM, USA, 6–15. https://doi.org/10.1145/3427796.3427831

[2] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. 2019. The Web Browser as Distributed Application Server: Towards Decentralized Web Applications in the Edge. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking (EdgeSys '19)*. ACM, USA, 7–11. https://doi.org/10.1145/3301418.3313938

[3] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. 2021. OWebSync: Seamless Synchronization of Distributed Web Clients. *IEEE Transactions on Parallel & Distributed Systems* 32, 9 (2021), 2338–2351. https://doi.org/10.1109/TPDS.2021.3066276

[4] Martin Kleppmann. 2022. Making CRDTs Byzantine Fault Tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '22)*. ACM, USA, 8–15. https://doi.org/10.1145/3517209.3524042

[5] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2019)*. ACM, USA, 154–178. https://doi.org/10.1145/3359591.3359737

[6] Stephan A. Kollmann, Martin Kleppmann, and Alastair R. Beresford. 2019. Snapdoc: Authenticated snapshots with history privacy in peer-to-peer collaborative editing. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 210–232. https://doi.org/10.2478/popets-2019-0044

[7] Hugo Krawczyk and Pasi Eronen. 2010. HMAC-based Extract-and-Expand Key Derivation Function (HKDF). RFC 5869. https://doi.org/10.17487/RFC5869

[8] Essam Mansour, Andrei Vlad Sambra, Sandro Hawke, Maged Zereba, Sarven Capadisli, Abdurrahman Ghanem, Ashraf Aboulnaga, and Tim Berners-Lee. 2016. A Demonstration of the Solid Platform for Social Web Applications. In *Proceedings of the 25th International Conference Companion on World Wide Web (WWW '16 Companion)*. WWW, CHE, 223–226. https://doi.org/10.1145/2872518.2890529

[9] Ralf Merkle. 1988. A Digital Signature Based on a Conventional Encryption Function. In *Advances in Cryptology (CRYPTO '87)*. Springer, Berlin, Heidelberg, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[10] Amit Sahai and Brent Waters. 2005. Fuzzy Identity-Based Encryption. In *Advances in Cryptology – EUROCRYPT 2005*. Springer, Berlin, Heidelberg, 457–473. https://doi.org/10.1007/11426639_27

[11] Marc Shapiro, Nuno Perguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*. Springer, Berlin, Heidelberg, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[12] Albert van der Linde, João Leitão, and Nuno Preguiça. 2020. Practical Client-Side Replication: Weak Consistency Semantics for Insecure Settings. *Proc. VLDB Endow.* 13, 12 (2020), 2590–2605. https://doi.org/10.14778/3407790.3407847

[13] Gavin Wood. 2014. *Ethereum: a secure decentralized generalized transaction ledger*. Yellow paper. ethereum.org.