Gossiping with Append-Only Logs in Secure-Scuttlebutt

Anne-Marie Kermarrec
Erick Lavoie
anne-marie.kermarrec@epfl.ch
erick.lavoie@epfl.ch
EPFL, Lausanne, Switzerland

Christian Tschudin
christian.tschudin@unibas.ch
University of Basel, Basel, Switzerland

Abstract

Middlewares for building social applications, in which infrastructure is provided by participants, are currently developed by open-source communities. Among those, Secure-Scuttlebutt has pioneered the use of replicated *authenticated single-writer append-only logs*, i.e., chains of ordered immutable events specific to each participant, replicated by gossip algorithms that are driven by social signals, to build eventually-consistent social applications. The use of persistent append-only logs removes parameters traditionally required to be tuned for gossiping. We present two gossip models that can be used for replication: a new *open* model, simpler than the current SSB implementation, that works best in small and trusted groups; and the *transitive-interest* model practically deployed by SSB, that scales to thousands of participants and is spam- and sybil-resistant. We also present limitations of both to motivate further research.

CCS Concepts: • **Computer systems organization** → **Peer-to-peer architectures**; • **Human-centered computing**; • **Computing methodologies** → **Distributed algorithms**;

Keywords: gossip algorithms, append-only logs

ACM Reference Format:
Anne-Marie Kermarrec, Erick Lavoie, and Christian Tschudin. 2020. Gossiping with Append-Only Logs in Secure-Scuttlebutt. In *1st International Workshop on Distributed Infrastructure for Common Good (DICG'20), December 7–11, 2020, Delft, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3428662.3428794

1 Introduction

Over the last decade, open source communities have pioneered a redecentralization of major Internet services [7] with the promises, among others, of increased agency for participants over the data they produce and minimal infrastructure requirements beyond participants' devices. Among the ongoing projects, Dat [29], renamed Hypercore [14], has been used to efficiently share large scientific datasets between research teams [31]. Also, Secure-Scuttlebutt[1] (SSB) [37] is actively used for social applications, e.g., blogging and code development [33], and has active open-source mobile clients [21, 27]. Both Hypercore and SSB are supported by a combination of grassroot donations [1–3], grants [4], and a large number of volunteers.

Both projects pioneered the use of replicated *authenticated single-writer append-only logs*: chains of ordered immutable events, similar to blockchains [15] but specific to each participant and without the need for global consensus. In particular, SSB efficiently replicates logs using the Scuttlebutt gossip protocol [39], by only transmitting the latest missing events between pairs of replicas.

Compared to commonly-known gossip algorithms [13], SSB has less parameters to tune: (1) no temporary buffer size, logs are persisted because storage is abundant and cheap; (2) no retransmission, events are exchanged over reliable channels when found missing from meta-data; (3) no fan-out, i.e., number of neighbours to contact, participants exchange events when they connect, often after days offline [32].[2] The result is a simple yet practical messaging middleware, as shown by the hundreds of daily SSB participants [25].

Previous work on SSB has already described the secure handshake [35], the data format and communication protocols [5], and a general overview and comparison to other information-centric approaches [37]. However, the range and limitations of distributed abstractions that can be built with SSB have yet to be shown both in theory and in practice.

In this paper, we provide a first step in that direction by describing the core concepts of SSB that are used to provide *eventually-consistent* replication: when a store receives a new event (in a log), other stores eventually also receive a copy of that event, as long as all stores are transitively connected through a sequence of temporary connections. We present two gossip models providing eventual consistency: first, we

DICG'20, December 7–11, 2020, Delft, Netherlands
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8197-0/20/12.
https://doi.org/10.1145/3428662.3428794

[1] *Scuttlebutt* is marine slang for gossip: it refers to the water barrel sailors gather around for refreshment and socialising. Secure-Scuttlebutt is gossip, which contrary to real life, cannot be modified by intermediaries.

[2] One author successfully updated after months offline to find 4000 new messages.

propose a new *open model* that is simpler than currently used in SSB but helps explain the mechanics and is easier to implement for smaller and more trusted settings; second, we present the model of the current SSB implementation, which we call the *transitive-interest* model, that is used by thousands of Internet participants, and is spam- and sybil-resistant [10] by replicating only the logs participants are interested in.

In the rest of this paper, we first present the core concepts used by SSB in Section 2. We then present the *open* model in Section 3 and the *transitive-interest* model in Section 4: for both we present their gossip algorithm, the properties they provide, some limitations, and the kinds of applications for which they are suitable. We compare the design of SSB with other related work in Section 5 and conclude in Section 6.

2 Concepts

SSB organises events in logs, that are replicated between stores. Events may represent a user action or the result of processing operations. Logs organise events in a data structure for efficient replication and integrity. Stores hold many replicated logs either in the storage of an active process, or passively on a storage device. Stores are connected over reliable communication channels, such as TCP connections over the Internet or USB connections on a local machine. Many data formats and communication protocols can be used, they are therefore abstracted in this paper. We present events, logs, stores, a procedure to reconcile the state of logs, and briefly discuss the support of private content.

2.1 Event

An *event* is a tuple *(id, previous, index, signature, content)* where:

id	is the *publicKey* of the creator
previous	is the hash of the previous event in the log including the signature, *null* if none
index	(sequence number) is the position of the event in the log
content	is defined by applications
signature	is the cryptographic signature of *id*, *previous*, *index*, and *content*, obtained with the *privateKey* that corresponds to the *publicKey*

All fields except *content* represent the meta-data about events: they enable efficient replication and integrity of the event chain. Changing *id*, *previous*, *index* or *content* requires a change of *signature* and the signature is linked to *id*: events are therefore **immutable** for anyone but the owner of the *privateKey*.

2.2 Log

A *log* is an identifier *id* associated to a sequence of events, e.g., an empty log is $id : []$ and a log with two events is $id : [(id, null, 0, ..., ...), (id, hash_0, 1, ..., ...)]$, with the operations listed in Table 1.

All operations are implemented to ensure the log is: **secure**, i.e., only the owner of the corresponding *privateKey* can append new events; **monotonic**, i.e., information can only be added to the log by adding new (immutable) events; **linear (total order)**, i.e., no two events have the same predecessor (either *null* or a reference to a previous event); **single writer**, i.e., all log events have the same *id* that corresponds to the *id* of the log; **connected**: all log events are transitively connected, i.e., no gaps in the sequence. A log with those properties is said to be *correct*. The implementations are omitted for brevity.

2.3 Store and Frontier

A *store* is a set $\{log_i, ...\}$ that represents the logs stored locally. Adding or removing logs in the store can be the result of direct user actions or be indirectly the result of operations triggered by new events added to logs.

A *frontier* is a set $\{(log_i.id, log_i.last), ...\}$ that represents the latest known indexes about logs in a store. The difference between two frontiers, e.g., $\{(log_i.id, log_i.last), ...\} - \{(log'_i.id, log'_i.last), ...\}$ such that $log_i.id = log'_i.id$, represents the new events in one store that are not yet in the other. Two frontiers may differ in the logs they contain because two stores may contain only partially overlapping sets of logs: in that case the difference between the two frontiers only comprises the logs with *id*s present in both.

Stores and frontiers are linked to events and logs by the operations listed in Table 2.

2.4 Updating

Two stores in different locations may diverge because new events have been added locally. *Updating* is the process of propagating the new events in one store not present in the other. The following sequence of operations ensures that logs present in both stores, i.e., that have the same id, will both contain the same sequence of events:

Algorithm 1 $Update(store, store')$: update $store$ and $store'$ with missing new events present in the other.

```
1: $frontier \leftarrow store.frontier(store.ids)$
2: $frontier' \leftarrow store'.frontier(store.ids)$
3: $news \leftarrow store.since(frontier')$
4: $news' \leftarrow store'.since(frontier)$
5: $store \leftarrow store.update(news')$
6: $store' \leftarrow store'.update(news)$
```

Algorithm 1 provides: **consistency**, i.e., if two logs with the same id are present in two stores, then after an update both logs will contain the same events; **terseness**, i.e., only events present in either *store* or *store'* but not the other are

Table 1. Authenticated single-writer log operations

$log \leftarrow create(publicKey)$	create a log from a public key
$log \leftarrow log.append(content, privateKey)$	extend the log with a new event created locally (as owner) from *content*
$log \leftarrow log.update(events)$	extend the log with the subset of compatible *events* previously created remotely
$events \leftarrow log.get(start, end)$	get the set of events with index included between *start* and *end* (can be the same)
$index \leftarrow log.last$	get the index of the last event stored locally
$id \leftarrow log.id$	get the id (publicKey) of the log

Table 2. Store operations

$store \leftarrow store.add(log)$	add a log to the store
$store \leftarrow store.remove(id)$	remove the log with *id* from the store
$log \leftarrow store.get(id)$	get the log with *id* from the store (if present)
$ids \leftarrow store.ids$	get the set of ids of the logs in the store
$frontier \leftarrow store.frontier(ids)$	get the current *frontier* of the store only for *ids*
$events \leftarrow store.since(frontier)$	get the set of *events* that happened after *frontier*
$store \leftarrow store.update(events)$	update the logs in *store* with *events*

exchanged; **conservative replication**, i.e., only logs already present in *store* are updated, other logs are ignored.

Consistency provides a *network of stores* with **eventual consistency** on the logs they share, i.e., when a store receives a new event, other stores eventually also receive a copy, as long as the network is transitively connected through a sequence of temporary connections. *Terseness* minimizes bandwidth and the *conservative replication* of logs ensures that unknown logs and their events are only propagated after being first explicitly added to the local store.

2.5 Private Content

SSB uses Ed25519 key pairs both for signature and encryption [24]. Access to content is restricted by encrypting it using the *publicKey* of recipient(s) using a *private-box* [38], a scheme that hides the recipients, their number, and the content. When encrypted content arrives during an update, a recipient tries to decrypt it with their *privateKey*. If the decryption succeeds, the content was intended for them, otherwise not.

3 Open Gossip

In this section, we present an *open*, or unrestricted, gossip model to illustrate the basic mechanics. The goal of this model is to maximise the diffusion of events by replicating them in all participants.

Algorithm 2 provides the pseudo-code of one implementation: every time a pair of stores are connected, logs present in only one store are added to the other, and then both stores are updated using Algorithm 1. To be concise, the algorithm is written from a global perspective: in practice it is actually implemented as a two-parties protocol.

Algorithm 2 Open Gossip

```
1: loop
2:     Randomly pick store and store' from Participants
3:     for id' in store'.ids − store.ids do store.add(create(id'))
4:     for id in store.ids − store'.ids do store'.add(create(id))
5:     Update(store, store')
6: end loop
```

3.1 Properties

This model implements **total replication**, i.e., after new events are added in one store, all stores should eventually contain the same logs (and the same events) including the new events. Moreover, it provides **total persistence**, i.e., once an event is added to a store, it is kept forever.

Even in this open model, private operations are possible. For example, any participant can automatically backup private information by creating an event and encrypting its *content* using the *publicKey* of their log (and reading it back later with their *privateKey*). If their copy of the store is lost or destroyed, and they still have access to their *privateKey*, they can retrieve all the content from any other participant. Also, any participant can privately message other participants, without revealing the recipients, by encrypting the content of a message using the *id* (*publicKey*) of the corresponding logs [38]. More generally, any privacy requirement that only targets *restriction of access* to *content*, and can be satisfied with cryptographic protocols, is compatible with this model and the one in the next section.

3.2 Limitations

While simple and useful, the open model suffers from some limitations. For one, the memory usage of *each* store is proportional to the total activity, past and present, of all participants. For relatively small groups and organisations, and most activity recorded as text, audio, or images, the storage capabilities of today's integrated or removable drives suffices. However, large groups may generate too much data.

Second, within this model, participants have no *agency*. Perhaps, they are not *interested* in some content, or perhaps they actively *object* to replicating other logs. Yet, even if they remove unwanted logs from their local stores, replicas from other stores will be copied back. The same property enables *spam*: a spammy log added to a single store will be replicated in all others. This model is therefore applicable only within small groups of mutually trusting participants.

3.3 Applications

Within small and highly trusting groups, it is often most important to simplify the process of adding new participants to make sure they quickly have access to all the information. This model enables new participants to bootstrap their empty store from any other participant.

The complete replication on each store enables offline operation, similar to *local-first applications* [22]. This is ideal, for example, to implement a team messaging application where clients update over a local network and only a single group uses that network. In that setting, security and privacy is usually provided by the network policy in a local or virtual private network (VPN).

When used with removable drives, this model is well suited, e.g., to associations that only need to record contributions, and quickly share past archives and recent updates.

4 Transitive-Interest Gossip

Some larger organisations, such as a university department, may host hundreds and even thousands of participants. This may become too many for the *open* model. In that case, the replication of logs can be directed by the *interest* of participants towards one another. Doing so, the number of social relationships participants can meaningfully track, estimated to 150 people on average [11] and empirically validated on Twitter [17], restricts the number of logs replicated. This is the model currently used in SSB, with more than 10,000 identities created [37] and most participants having up to a few hundred logs they actively follow.[3] When such a model is used for social applications, two additional problems arise.

First, some applications, e.g., conversations between many participants, are spread over multiple logs. To ensure participants receive all messages, the logs in the *transitive* graph of interests should therefore be replicated. Additionally, this may serve to *discover* new interesting people to follow, similar to how a friend may introduce a friend to another one.

Second, all participants may not follow the same way as those they follow. Sometimes they are not interested in the same people. Other times, past disagreements result in one of the participants not wanting to receive information about another through their common friend.[4] It is therefore necessary to also allow participants to *block* other logs from being replicated in their store.

[3]There is one extreme case of an SSB user that decided to manually follow every other identities.

[4]Authors witnessed both multiple times in the last years.

Similar to prior work on membership management [12], *the logs participating in the gossip algorithm also propagate the interest/disinterest information*. Doing so, both the content and where it should (and shouldn't) flow propagate at the same time through the same gossip. However, because participants can have inconsistent interest/disinterests, each participant can only have a *subjective perspective* of the entire set of participants and global state of content.

Participants signal interest/disinterest by recording a *follow*/*block* event in their log, that includes the target *id*. Follow and block can be cancelled by recording an *unfollow*/*unblock* event. Which ids are transitively followed or blocked, is computed from the perspective of a specific participant's *id* and the other logs contained in a store. The additional operations introduced by this model are listed in Table 3.

Table 3. Interest operations. *privateKey* abbreviated *privK*.

$log \leftarrow follow(log, privK, id)$	publicly record in *log* active interest in *id*
$log \leftarrow unfollow(log, privK, id)$	publicly record in *log* lost interest in *id*
$log \leftarrow block(log, privK, id)$	publicly record in *log* active disinterest in *id*
$log \leftarrow unblock(log, privK, id)$	publicly record in *log* drop of disinterest in *id*
$ids \leftarrow followed(store, id)$	retrieve *ids* transitively followed from *id* in *store*
$ids \leftarrow blocked(store, id)$	retrieve *ids* transitively blocked from *id* in *store*

Using the subjective *followed* and *blocked* operations requires an explicit *id* to be associated with the stores of participants for gossiping, as shown in Algorithm 3. Upon a connection, the followed set is first computed and the missing logs added in the store. The *ids* that are blocked, and present in the store, are then removed. After both participants have done the same, only the shared *ids* are updated.

Algorithm 3 Transitive-Interest Gossip

```
        ▷ Participants abbreviated P and store abbreviated st.
1: loop
2:     Randomly pick (id,st) and (id,st') from P
3:     for f in followed(st,id) − st.ids do st.add(create(f))
4:     for b in blocked(st,id) ∩ st.ids do st.remove(b)
5:     ...                                    ▷ Same for store st'
6:     Update(st, st')
7: end loop
```

Different algorithms can be used to compute the *followed* and *blocked* sets. The current version of SSB uses the *ssb-friends* module [36] that: (1) is set to transitively follow 2 hops (i.e., friends-of-friends); (2) ignores transitive blocks if transitive follows are present, which keeps controversial participants discoverable, and (3), does not follow beyond an *id* that is transitively blocked to avoid sybil attacks. Other alternatives are possible, e.g., only following *id*s not blocked by anyone, to increase the level of trust between participants.

4.1 Properties

This model implements **subjective interest-based replication**: stores can diverge in the logs they contain, driven by the choices of individual participants. Stores are also **eventually consistent** but only on shared logs.

The use of interest graphs provides **spam-** and **sybil-resistance**. Any participant may create as many identities as they wish. However, each of those identities has to be explicitly followed to be replicated. Convincing other participants to follow new identities has a significant cost: in the SSB community it implies creating regular compelling content for others. Moreover, an attacker has to convince other participants to *keep following* the sybil identities. If the social graph had clusters of identities with few links to the rest of the social graph, i.e. a 'cut' [40], this would eventually be detected and the connecting accounts would be blocked.

4.2 Limitations

Participants have limited control over the propagation of their data: once their events are replicated, a blocked id can still get updates from other participants that do not block them. The root of the problem stems from the unilateral direction of *follow* and *block* signals: it only takes one side of a relationship to express interest or disinterest. However, it takes both sides to establish *mutual trust*. Moreover, trust works in the opposite direction as interest. In terms of flow of data, trust means *allowing* data to flow to another participant, while interest means *wanting* data to flow to oneself. Mutual interest therefore does not imply mutual trust. A more private implementation would restrict propagation of updates only through mutually trusted participants.

Moreover, interests are revealed during the update step. Participants can therefore learn the social graph by recording which *ids* are requested by others during updates.

4.3 Applications

Transitive-interest gossip is very useful for publishing, e.g., blogs: the load on the original publisher is limited to the other stores with which they update. All further replication is performed by the rest of the participants. It can therefore scale to a large number of participants.

However, if it is used for group conversations or for enabling comments on blog posts, then the number of participants will likely be smaller because the original poster needs to follow the other participants to see their comments.

Some privacy issues make this model inappropriate for at-risk populations, such as investigative journalists who may risk they lives by revealing their interests. However, for participants for whom these issues are not critical, this model provides a practical online social space free of trolls, spammers, and unwanted advertisement without relying on centrally managed infrastructure.

5 Related Work

The structure of *append-only logs* is similar to that of *blockchains* for crypto-currencies [15]: they both consist of a sequence of immutable events that can only be extended by adding new events, and in which the integrity of earlier events is guaranteed by the chain of hashes from later events. However, for the gossip algorithms presented in this paper, each participant has their own log and the system is not trying to reach consensus on a single global state. The practical adoption of SSB shows that *eventual consistency* is sufficient to build many decentralised social applications [33]. Append-only logs are also popular in industrial data stores as Kappa Architecture [30] and have been used previously to provide accountability [20] and for securing timelines [26].

Compared to many *peer-to-peer systems* implementing, e.g., a global file system [6], or a global market place [34], SSB is not aiming at building *global*, *online* platforms for *anonymous* participants. Similar to decentralised online social networks [9, 19] using social overlays [18], SSB is tailored for *local* or *interest-driven* deployments for particular communities covering a variety of applications, with different communities forming disjoint networks. The immutability and wide replication of logs from participants fosters trust by providing reliable signals on past behaviour.

Complementary to the algorithms we presented, other results decrease storage usage while preserving availability [23], and optimise the speed of propagation of updates based on participant interactions [8] and relationships [28].

The transitive interest model of Section 4 uses ideas similar to previous work for spam and sybil resistance. SybilLimit [40] also leverages social interest signals to detect and prevent sybil attacks, and Reliable Email [16] also uses friend-of-friend whitelisting, gathered from email contacts, but to unconditionally accept emails from a trusted network.

In summary, there is prior work for the different ideas presented in this paper. The main contribution of SSB is a novel, and in our opinion *elegant*, *synthesis* with an active community of users and contributors. It is therefore a valuable testbed for technical and social experimentations.

6 Conclusion

We have presented gossiping in Secure-Scuttlebutt (SSB), through its core operations and algorithms organised around *authenticated single-writer append-only logs*. We explained an *open gossip* and a *transitive-interest gossip* models that respectively work for small and medium-scale communities. Both models provide *eventual consistency* for participants.

Both are also compatible with cryptographic protocols, e.g., for private messaging, that use participants' id (public key) for restricting access to the content of logged events.

The *open* model is the simplest and is especially suited for small groups and organisations running behind networks that provide privacy and security. However, it is vulnerable to spammers in larger and more open settings. The *transitive interest-based model*, which corresponds to the model used by the current SSB implementation, enables larger participation and tolerates spam. It is currently practically used with hundreds of daily active participants, and thousands of occasional ones. However, it still has limited privacy guarantees that are not suitable for at-risk populations, such as investigative journalists with sensitive interests.

Future research on more private designs, different social signalling algorithms, and efficient gossip algorithms could make new variants of SSB useful in more settings.

References

[1] 2018. Dat Open Collective. Retrieved 2020-08-08 from https://opencollective.com/dat.
[2] 2018. Manyverse Open Collective. Retrieved 2020-08-08 from https://opencollective.com/manyverse.
[3] 2018. SSB Open Collective. Retrieved 2020-08-08 from https://opencollective.com/secure-scuttlebutt-consortium.
[4] 2019. Dat Organization and Funding. Retrieved 2020-08-08 from https://github.com/datproject/organization.
[5] 2019. Scuttlebutt Protocol Guide: How Scuttlebutt peers find and talk to each other. Retrieved 2020-09-02 from https://ssbc.github.io/scuttlebutt-protocol-guide/.
[6] Juan Benet. 2014. IPFS - Content Addressed, Versioned, P2P File System. *CoRR* abs/1407.3561 (2014). arXiv:1407.3561
[7] Irina Bolychevsky and Gerben. 2013. Redecentralize. Retrieved 2020-08-08 from https://redecentralize.org/.
[8] Marco Conti, Andrea De Salve, Barbara Guidi, and Laura Ricci. 2014. Epidemic Diffusion of Social Updates in Dunbar-based DOSN. In *European Conference on Parallel Processing*. Springer, 311–322.
[9] Anwitaman Datta, Sonja Buchegger, Le-Hung Vu, Thorsten Strufe, and Krzysztof Rzadca. 2010. Decentralized Online Social Networks. In *Handbook of social network technologies and applications*. Springer.
[10] John R Douceur. 2002. The sybil attack. In *International workshop on peer-to-peer systems*. Springer, 251–260.
[11] Robin Dunbar. 2014. *Thinking big: How the evolution of social life shaped the human mind.* Thames & Hudson.
[12] P. Th. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. 2003. Lightweight Probabilistic Broadcast. *ACM Trans. Comput. Syst.* 21, 4 (Nov. 2003), 341–374.
[13] Patrick T Eugster, Rachid Guerraoui, A-M Kermarrec, and Laurent Massoulié. 2004. Epidemic information dissemination in distributed systems. *Computer* 37, 5 (2004), 60–67.
[14] Paul Frazee, Andre Wosh, and Mathias Buus Madsen. 2020. Hypercore Protocol. Retrieved 2020-08-08 from https://hypercore-protocol.org/.
[15] Weichao Gao, William G Hatcher, and Wei Yu. 2018. A survey of blockchain: techniques, applications, and challenges. In *ICCCN 2018.* IEEE, 1–11.
[16] Scott Garriss, Michael Kaminsky, Michael J Freedman, Brad Karp, David Mazières, and Haifeng Yu. 2006. RE: Reliable Email.. In *NSDI*, Vol. 6. 22–22.
[17] Bruno Gonçalves, Nicola Perra, and Alessandro Vespignani. 2011. Modeling users' activity on twitter networks: Validation of dunbar's number. *PloS one* 6, 8 (2011), e22656.
[18] Barbara Guidi, Tobias Amft, Andrea De Salve, Kalman Graffi, and Laura Ricci. 2016. DiDuSoNet: A P2P architecture for distributed Dunbar-based social networks. *Peer-to-Peer Networking and Applications* (2016).
[19] Barbara Guidi, Marco Conti, Andrea Passarella, and Laura Ricci. 2018. Managing social contents in decentralized online social networks: a survey. *Online Social Networks and Media* 7 (2018), 12–29.
[20] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. 2007. PeerReview: Practical Accountability for Distr. Sys.. In *SOPS 2007.* ACM.
[21] Evan Henshaw-Plath, Amanda Hickman, Henry Bubert, Christian Bundy, Martin Dutra, and Sebastian Heit. 2018. Planetary. Retrieved 2020-08-08 from https://planetary.social/.
[22] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-First Software: You Own Your Data, in Spite of the Cloud. In *Onward! 2019.* ACM.
[23] David Koll, Jun Li, and Xiaoming Fu. 2014. SOUP: An Online Social Network by the People, for the People. In *Middleware 2014.* ACM.
[24] A. Langley, M. Hamburg, and S. Turner. 2016. RFC 7748: Elliptic Curves for Security. Retrieved 2020-09-10 from https://tools.ietf.org/html/rfc7748.
[25] Charles E. Lehner. 2020. Daily Active Users. Retrieved 2020-09-06 from https://ssb.celehner.com/activity/.
[26] Petros Maniatis and Mary Baker. 2002. Secure History Preservation through Timeline Entanglement. *CoRR abs/cs/0202005* (2002).
[27] Andre Medeiros, Gordon Martin, Luandro, Robert P. Levy, Charles E. Lehner, Jacob Karlsson, and David Gomez. 2018. Manyverse. Retrieved 2020-08-08 from https://manyver.se.
[28] Giuliano Mega, Alberto Montresor, and Gian Pietro Picco. 2011. Efficient dissemination in decentralized social networks. In *2011 IEEE International Conference on Peer-to-Peer Computing*. IEEE, 338–347.
[29] Maxwell Ogden, Karissa McKelvey, and Mathias Buus Madsen. 2017. Dat - Distributed Dataset Synchronization And Versioning. https://doi.org/10.31219/osf.io/nsv2c
[30] Milinda Pathirage. 2014. kappa-architecture.com/. Retrieved 2020-09-09 from https://milinda.pathirage.org/kappa-architecture.com/.
[31] Danielle C Robinson, Joe A Hand, Mathias Buus Madsen, and Karissa R McKelvey. 2018. The Dat Project, an open and decentralized research data tool. *Scientific data* 5, 1 (2018), 1–4.
[32] Staltz, André. 2017. An Off-Grid Social Network. Retrieved 2020-09-10 from https://staltz.com/an-off-grid-social-network.html.
[33] T. 2017. Secure Scuttlebutt Applications. Retrieved 2020-08-08 from https://handbook.scuttlebutt.nz/applications.
[34] Taaki, Amir and Hoffman, Brian. 2016. https://openbazaar.org/. Retrieved 2020-09-10 from https://openbazaar.org/.
[35] Dominic Tarr. 2015. Designing a secret handshake: authenticated key exchange as a capability system. *self published* (2015), 1–13.
[36] Dominic Tarr. 2017. ssb-friends. Retrieved 2020-09-09 from https://github.com/ssbc/ssb-friends.
[37] Dominic Tarr, Erick Lavoie, Aljoscha Meyer, and Christian Tschudin. 2019. Secure Scuttlebutt: An Identity-Centric Protocol for Subjective and Decentralized Applications. In *ICN 2019.* ACM.
[38] Tarr, Dominic. 2015. private-box. Retrieved 2020-09-10 from https://github.com/auditdrivencrypto/private-box.
[39] Robbert van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas. 2008. Efficient Reconciliation and Flow Control for Anti-Entropy Protocols. In *LADIS 2008.* ACM.
[40] Haifeng Yu, Phillip B. Gibbons, Michael Kaminsky, and Feng Xiao. 2010. SybilLimit: A near-Optimal Social Network Defense against Sybil Attacks. *IEEE/ACM Trans. Netw.* (June 2010).