PRSONA: Private Reputation Supporting Ongoing Network Avatars

Stan Gurtler*
sgurtler@cisco.com
Cisco Systems, Inc.
San Jose, California, USA

Ian Goldberg
iang@uwaterloo.ca
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

As an increasing amount of social activity moves online, online communities have become important outlets for their members to interact and communicate with one another. At times, these communities may identify opportunities where providing their members specific privacy guarantees would promote new opportunities for healthy social interaction and assure members that their participation can be conducted safely. On the other hand, communities also face the threat of bad actors, who may wish to disrupt their activities or bring harm to members. *Reputation* can help mitigate the threat of such bad actors, and there has been a wide body of work on privacy-preserving reputation systems. However, previous work has overlooked the needs of small, tight-knit communities, failing to provide important privacy guarantees or address shortcomings with common implementations of reputation. This work features a novel design for a privacy-preserving reputation system which provides these privacy guarantees and implements a more appropriate reputation function for this setting. Further, this work implements and benchmarks said system to determine its viability in real-world deployment. This novel construction addresses shortcomings with previous approaches and provides new opportunity to its target audience.

CCS CONCEPTS

• **Security and privacy → Pseudonymity, anonymity and untraceability**; *Privacy-preserving protocols*; Public key encryption; Social aspects of security and privacy.

KEYWORDS

anonymity, privacy, reputation

ACM Reference Format:
Stan Gurtler and Ian Goldberg. 2022. PRSONA: Private Reputation Supporting Ongoing Network Avatars. In *Proceedings of the 21st Workshop on Privacy in the Electronic Society (WPES '22), November 7, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3559613.3563197

*This work was conducted at the University of Waterloo.

1 INTRODUCTION

With the aid of the internet, individuals have become empowered to forge novel and powerful connections with each other. A variety of groups have been able to connect even from within geopolitical regions that are hostile towards them (as in the case of LGBTQ individuals) [30]. Online communities have made significant positive impact on countless individuals. However, such communities have also been targeted by malicious individuals who pose as legitimate members to aid their campaigns of stalking and harassment [24].

Meanwhile, places these communities currently gather online have proven inadequate for their needs in various ways. Members of such communities face abuse on social media sites and other gathering places from bad actors [3, 24, 29]. Social media platforms frequently lack robust methods for addressing abuse, and also may impose their own rules and policies that conflict with the standards the communities desire. For example, LGBTQ individuals may want to keep identifying information apart from their participation in such communities. The platforms themselves can also drive community dynamics in undesirable ways. Some advocates have called for adjustments toward "human-scale social media" [7, 1:29:16], explicitly because the sorts of engagement that social media algorithms currently drive towards can lead to unhealthy interactions [17].

In order to incentivize good behaviour and disincentivize bad behaviour, communities often choose to employ *reputation*. Platforms like Reddit provide a model. In such a system, reputation measures good-faith participation within the community. Though reputation has been used for a large portion of the internet's lifetime on sites like eBay and Amazon, the privacy needs of these communities require careful consideration. Further, previous implementations are frequently not well suited to address the problem at hand for such communities. These communities want to identify misbehaviour and incentivize members to correct it. However, with Reddit-style reputation, a user may amass a great amount of reputation that they can effectively "spend" to misbehave. Further, these systems do not always protect the privacy of their users. Metadata associated with votes and participation may be able to de-anonymize users via long-term linkage of their activities. Such de-anonymization can endanger users, by revealing to abusive individuals that said users have acted in some manner against them.

We propose a novel system, PRSONA, or Private Reputation Supporting Ongoing Network Avatars, in order to enable community protection and maintenance while addressing these previous shortcomings. Importantly, PRSONA is designed with usage in these community settings in mind; the privacy guarantees it provides and the reputation function (the method of combining ratings into a single score) it uses were selected for the opportunities they provide in such settings. The reputation function in particular is more

Table 1. Comparison of recent privacy-preserving reputation systems, excerpted from Gurtler and Goldberg's systematization [12, Table 1], and adding more recent work, as well as our own work, PRSONA

System		Structure				Trust					Privacy				Rep.			
Name	*Year*	*Centralization*	*Directionality*	*Rep. Scope*	*Rep. Ownership*	*Correctness*	*Unlinkable to TTP*	*# TTP for Setup*	*# TTP Ongoing*	*More via Anytrust*	*V-V Unlinkability*	*2V Unlinkability*	*R-U Unlinkability*	*Exact Rep. Blinding*	*Acc. Stars*	*Avg. Stars*	*STMC*	*LTMC*
PFWRAP [33]	2016	☍	↔	∃	⊙	●	●	0	0	-	●	●	-	-	-	-	●	●
Beaver [28]	2016	∴	→	∀	☽	◐	●	0	0	-	●	●	-	-	●	●	-	-
AnonRep [32]	2016	★	↔	∀	☽	◐	◐	2	2	●	●	●	●	●	●	●	-	-
EARS [19]	2020	★	↔	∀	☽	●	●	1	1	-	●	-	●	●	-	-	●	-
Dimitriou [8]	2021	∴	↔	∀	☽	●	●	1	0	●	●	●	●	●	●	●	-	-
PRSONA (honest-but-curious)	This Work	★	↔	∀	☽	-	●	2	2	●	●	●	●	●	-	-	●	●
PRSONA (covert)	This Work	★	↔	∀	☽	●	●	2	2	●	●	●	●	●	-	-	●	●

Reputation Attributes

Centralization:	★ = Third-Party Mediation	☍ = Ephemeral Mesh Topology	∴ = Proofs of Validity
Directionality:	→ = Simplex	↔ = Full-Duplex	
Scope:	∀ = Global	∃ = Local	
Ownership:	☽ = TTP-owned	⊙ = Voter-owned	
Correctness: via...	● = ...protocol guarantees	◐ = ...errors are traceable	- = ...TTP/miners
Trust Unlinkability: TTP can link...	● = ...nothing	◐ = ...misbehaviour	

complex than the simple ones commonly used in other systems, taking into account the user who is *giving* feedback when determining a user's reputation score — such a reputation function is termed a "voter-conscious reputation function" in our previous work [12]. Other common functions cannot do so and thus are not able to allow users to replace previous feedback, which can enable abuse of the reputation function. Further, we implement and benchmark this system; this implementation is available online at https://git-crysp.uwaterloo.ca/tmgurtler/PRSONA.

Section 2 of this paper discusses previous work, including previous systematization of the broader area. Section 3 describes the cryptographic tools we use to build PRSONA. Section 4 details the overall approach to designing PRSONA, as well as relevant assumptions. Section 5 outlines the design of PRSONA and its algorithms. Section 6 describes and benchmarks our implementation of PRSONA. Section 7 gives analysis of PRSONA's practicality for real-world deployment, and Section 8 concludes.

2 RELATED WORK

A number of privacy-preserving reputation systems have previously been proposed. Our previous work [12] provides a systematization of the space, and for purposes of standardization, we adopt the same terminology used there. For example, a *voter* expresses their opinion of a *votee* using a *vote*. Table 1 excerpts selected previous work from that systematization [12, Table 1], chosen due to their representativeness and relevance to PRSONA's targeted use case. We also add more recent work and PRSONA itself to the table.

In Table 1, we first identify four factors corresponding to a reputation system's structure — its overall approach to centralization, in what direction reputation flows, the scope of reputation (*i.e.*, whether every user's view of a specific user's reputation score is the same), and which entity holds on to the value that becomes a reputation score. Second, we identify five factors corresponding to the trust placed in third parties by a system — how correctness in the system is ensured, whether or not trusted third parties (TTPs) must be trusted with user privacy, how many TTPs are needed to first set up and then continuously run the system, and whether additional TTPs being added adds a potential point of failure or distributes the same amount of trust across more nodes. Third, we identify the privacy properties provided by each system. Fourth, we identify the reputation functions supported by each system.

Beaver [28] and the proposal by Dimitriou [8] both target transactional settings, like those of eBay or Amazon, and both use some form of append-only public ledger. In both, reputations are associated with the goods a votee sells. In Beaver's work, users provide feedback as an unidentified member of a known set of users who transacted with a votee. Dimitriou's proposal involves generating coins whose value constitutes the feedback. These are state-of-the-art transaction-based privacy-preserving reputation systems, but they miss key features for community usage (such as voter-conscious reputation functions and certain privacy properties).

Among Secure Multiparty Computation-based (SMC-based) reputation systems, PFWRAP [33] represents the state of the art. A reputation system is derived from secure (weighted) sum, and PFWRAP makes significant efficiency gains over previous similar approaches [2, 23, 31]. These systems are able to provide voter-conscious reputation systems, but miss key privacy properties and require that voters are always online when a user wants to see feedback on a votee.

EARS [19] is one of the few non-SMC-based systems to enable updating votes (and thus, voter-conscious reputation functions), doing so with blind and partially blind signatures. However, EARS requires significant amounts of trust in the issuer of these blind and partially blind signatures, which is undesirable in our setting.

AnonRep's [32] broad approach (using TTPs to perform a periodic verifiable shuffle of internal data to enable users to regularly change pseudonyms without linking themselves) inspired our own. However, AnonRep is incapable of enabling voter-conscious reputation functions. PRSONA is designed for communities where it is expected that any member could reasonably have interactions with and form opinions of every other member, a property we term "tight-knit". This property is related to Dunbar's number [9], the number of people with whom one person can maintain stable social relationships; recent estimates for this number range up to about 500 [20]. Tight-knittedness is directly modeled by the reputation function that PRSONA uses; every voter provides exactly one input on every votee. AnonRep, in contrast, cannot support such reputation functions. Its reputation function has all votes for a votee summed to calculate a score. A voter can provide multiple votes for

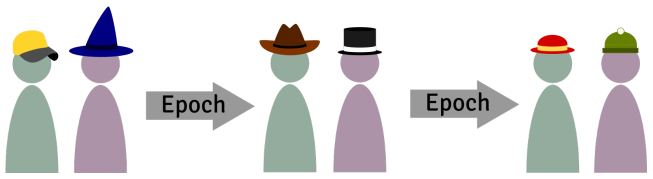

Figure 1. During each epoch, PRSONA users are assigned a new "fresh pseudonym" (public key X_t for epoch t, depicted here as hats) to interact with. Ratings from previous epochs are recalculated and carried over to the next epoch's pseudonyms.

a votee over time, weighting their input more heavily on the basis of their persistence, rather than appropriateness. Additionally, voters may be reticent to provide feedback in any cases but those of strong positive or negative reaction, knowing that a carelessly given vote will be credited to or held against a votee forever. These drawbacks allow more opportunity for a user to abuse the reputation function in AnonRep than in PRSONA.

3 CRYPTOGRAPHIC TOOLS

In order to accomplish its goals, PRSONA uses ElGamal and BGN encryptions in various places. For efficiency reasons, our implementation of PRSONA uses a prime-order version of BGN, and more detail on the exact libraries we build upon to do so is provided in Section 6.1.

3.1 ElGamal

The ElGamal cryptosystem was first proposed in 1985 [10]. In PRSONA, we use a scheme containing two variations from the originally proposed design. First, we put the message in the exponent of the group, so that we can encode integers. Second, in our version of the scheme, ciphertexts swap where a user's public key (X) and the group's generator ($\mathfrak{g}$) are used from the more typical ElGamal construction. This is necessary to support the way that PRSONA prevents servers from learning the linkage of a user's identity over time, and more detail on this is provided in Section 5.3.3. The full construction is as follows:

- **EGGroupKeyGen(1^λ):** Let $\mathcal{G}$ be a cyclic group generator. Compute $(\mathfrak{G}, \mathfrak{g}, q) \leftarrow \mathcal{G}(1^\lambda)$, where $\mathfrak{g}$ generates $\mathfrak{G}$, a cyclic group of order q. Choose $\mathfrak{h} \xleftarrow{\$} \mathfrak{G}$. Output the shared elements $E = (\mathfrak{G}, \mathfrak{g}, \mathfrak{h}, q)$.
- **EGUserKeyGen(E):** Choose $x \xleftarrow{\$} \mathbb{Z}_q^*$ and compute $X = \mathfrak{g}^x$. Output the public key $PK = (E, X)$ and the secret key $SK = x$.
- **EGEncrypt(PK, m):** Choose $r \xleftarrow{\$} \mathbb{Z}_q^*$. Output the ciphertext $C = (C_1, C_2) = (X^r, \mathfrak{g}^r \cdot \mathfrak{h}^m) \in \mathfrak{G}^2$.
- **EGDecrypt(SK, C):** Compute $y = x^{-1} \in \mathbb{Z}_q^*$. Note that, since $C_1 = X^r, C_1^y = \mathfrak{g}^r$. Output $m = \log_{\mathfrak{h}}(\frac{C_2}{C_1^y})$.

Note that although **EGDecrypt(SK, C)** involves calculating a discrete log, the range of values in use in PRSONA that would be encrypted in this manner is very limited (falling within $[0, 2n]$, for n the number of users in the system). With such a small range of values, brute forcing this discrete log is not challenging.

In PRSONA, we make one minor change: instead of one constant $\mathfrak{g}$ used in perpetuity in the system, PRSONA operates in epochs, as in Figure 1. In each epoch t, the PRSONA servers calculate a new $\mathfrak{g}_t$, in such a way that all user public key Xs are updated as well (denoted $X_t = \mathfrak{g}_t^x$). This is detailed further in Section 5.3.

3.2 Prime-order BGN

While the above variant of ElGamal is additively homomorphic, our application additionally requires (depth-1) multiplication. This additional property is provided by the BGN [5] cryptosystem.

The BGN cryptosystem involves three components: a group, a proper subgroup for which it is infeasible (without the private key) to determine whether an arbitrary member of the group is also a member of the subgroup, and a projection operator (using the private key) that removes the subgroup component from group elements. In the original construction, the group is an elliptic curve with order a product of two primes (and so thousands of bits long), and the projection operator removes one of the primes. For efficiency purposes, we implement a prime-order BGN construction that was first suggested by Freeman [11]. In this construction two order p^2 (for p prime) groups are selected (G, H; both consist of pairs of elements of order-p asymmetric pairing groups). Two subgroups (G_1 generated by g' and H_1 generated by h'; both are order p) and their corresponding projection operators (π_1, π_2) are chosen. Finally, defining a pairing e with elements of G and H results in elements of G_T; the subgroups G_1 and H_1 also implicitly define a subgroup G'_T of G_T and a corresponding projection operator π_T. The operation for generating these groups and projections is denoted $\mathcal{G}_{\mathcal{P}}(1^\lambda)$.

We make no significant changes from Freeman's construction. The full construction can be seen in Appendix A, and we highlight the portions of the construction relevant to our system as follows:

- **BGNKeyGen(1^λ):** Compute $(G, G_1, H, H_1, G_T, G'_T, e, \pi_1, \pi_2, \pi_T) \leftarrow \mathcal{G}_{\mathcal{P}}(1^\lambda)$. Choose $g \xleftarrow{\$} G$, $h \xleftarrow{\$} H$. Output the public key $PK = (G, G_1, H, H_1, G_T, e, g, h)$ and the secret key $SK = (\pi_1, \pi_2, \pi_T)$.
- **BGNEncrypt(PK, m):** Choose $g_1 \xleftarrow{\$} G_1$ and $h_1 \xleftarrow{\$} H_1$ and output the ciphertext $(C_A, C_B) = (g^m \cdot g_1, h^m \cdot h_1) \in G \times H$.
- **BGNMultiply(PK, C_A, C_B):** This algorithm takes as inputs two ciphertexts $C_A \in G$ and $C_B \in H$. Choose $g_1 \xleftarrow{\$} G_1$ and $h_1 \xleftarrow{\$} H_1$, and output $C = e(C_A, C_B) \cdot e(g, h_1) \cdot e(g_1, h) \in G_T$.
- **BGNAdd(PK, C, C'):** This algorithm takes as input two ciphertexts C, C' in one of G, H, or G_T. Choose $g_1 \xleftarrow{\$} G_1$ and $h_1 \xleftarrow{\$} H_1$, and do as follows:
  1. If $C, C' \in G$, output $C \cdot C' \cdot g_1 \in G$.
  2. If $C, C' \in H$, output $C \cdot C' \cdot h_1 \in H$.
  3. If $C, C' \in G_T$, output $C \cdot C' \cdot e(g, h_1) \cdot e(g_1, h) \in G_T$.
- **BGNDecrypt(SK, C):** The input ciphertext can be an element of G, H, or G_T.
  1. If $C \in G$, output $m \leftarrow \log_{\pi_1(g)}(\pi_1(C))$.
  2. If $C \in H$, output $m \leftarrow \log_{\pi_2(h)}(\pi_2(C))$.
  3. If $C \in G_T$, output $m \leftarrow \log_{\pi_T(e(g,h))}(\pi_T(C))$.

We further define two derivative functions that are helpful for our purposes:

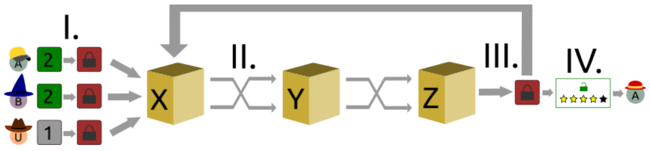

Figure 2. The data flow PRSONA uses to generate each user's reputation score. Note that this is done in parallel for all users. Step I depicts voting (Section 5.2.2). Voters may submit votes to any server or to none (in which case they would continue using a previously specified vote). Step II depicts score calculation, which is performed during epoch changeover (Section 5.3). More detail on this step can be observed in Figure 4. Step III depicts feedback, a component of our reputation function where previous reputation scores influence future calculations (Section 4.4). Step IV depicts publication, where a votee receives their score from the servers and may use it in interactions with other users (Section 5.2.1).

- **BGNEncrypt**$_G$**(**PK, m**):** Compute and output only the first component $C_A \in G$ of **BGNEncrypt(**PK, m**)**.
- **BGNEncrypt**$_H$**(**PK, m**):** Compute and output only the second component $C_B \in H$ of **BGNEncrypt(**PK, m**)**.

Note that although **BGNDecrypt(**SK, C**)** involves calculating a discrete log, the range of values in use in PRSONA that would be encrypted in this manner is limited (these values fall within $[0, 4n^2]$, for n the number of users in the system). Though this range is larger than that of **EGDecrypt()**, brute forcing this discrete log is still within reason, or it can be calculated in time $O(n)$ with the Pollard kangaroo method [25].

We assume that it is possible for the group homomorphisms π_1, π_2, and π_T (and thus the **BGNDecrypt** operation) to be generated distributively such that multiple entities must jointly calculate them. These homomorphisms involve raising group elements to specific combinations of integer factors. By distributing a specific subset of these factors across multiple parties, this is straightforward to calculate in a distributed manner.

4 OVERVIEW

In this section, we lay out the design choices made by PRSONA. A high-level workflow can be seen in Figure 2. We draw inspiration in part from a previous system, AnonRep [32]. Similar to AnonRep, PRSONA is specifically intended for use in forum-like settings, where members of a community interact and form opinions of one another regularly. PRSONA differentiates itself from AnonRep in particular by its choice of reputation function, which is better tailored for use by small, tight-knit communities.

4.1 Architecture

In PRSONA, there are two types of nodes: users, who act as both voters and votees, and servers, who maintain the accuracy of reputation. Users are untrusted and are not required to be constantly online. Servers have some trust placed in them (detailed in Section 4.2), and are assumed to be highly available. We expect servers to be operated independently. Given the forum and community settings that PRSONA is tailored to, these servers could logically be operated by various stakeholders within said community.

Users communicate with any server. Tracking their reputation, making and giving feedback, and verifying other users' reputations can all be done while interacting with only one server. Users confirm the correctness of a server's response with other servers.

4.2 Threat Model

In PRSONA, users are fully untrusted. They may collude with one another; when they do so, the system must not leak any information beyond what colluding users can conclude from their inputs and the output.

Servers in PRSONA may operate in one of two settings, per implementer's choice. First, in the honest-but-curious setting, servers may inspect user input, so long as they honestly follow the protocols of the system. This places a great deal of trust in the servers, but this trust comes with greater speed and efficiency of server operations, as servers perform fewer proofs of their correct behaviour than in our other setting. Second, in the covert setting (as first suggested by Aumann and Lindell [4]), servers may behave however they like, with one restriction. Servers may take no action for which they would (with non-negligible probability) be incriminated for deviating from the protocol, unlike a malicious adversary, who could take advantage of the probability that they are not immediately caught.

In addition to those restrictions, there are limitations on how many servers may collude. Like previous work, so long as any one server is honest, the privacy properties PRSONA guarantees will hold. However, the accuracy of scores can only be guaranteed in the covert setting as long as a majority of servers behaves honestly.

PRSONA protects against linking a user's score to their identity, but it does not protect the multiset of plaintext reputation scores. That is, PRSONA does not prevent any entity from learning what reputation scores are held by *some* user each epoch.

Though we separate out the two in our analysis, design, and implementation, a method exists to hybridize the covert and honest-but-curious settings. This hybrid approach provides security against a covert adversary, while only requiring an online cost in line with the honest-but-curious setting (along with an offline cost that is still in line with the covert setting). We examine this hybrid approach in greater detail in Section 7.

4.3 Security Goals

Anonymity. The primary goal of PRSONA is to allow a user to maintain ongoing participation in the system without allowing their activities to be linked together. To that end, we seek to provide four specific privacy guarantees to users. For the sake of standardization, we use the same terms as our previous work [12], with the same definitions (guaranteed for users against all other entities, server or otherwise):

Voter-Vote Unlinkability: a voter cannot be associated with a (plaintext) vote they cast, or with the fact that they voted for a particular votee.

Two-Vote Unlinkability: it is not possible to distinguish whether two (plaintext) votes were cast by the same voter or not.

Reputation-Usage Unlinkability: a votee can display or use their reputation to accumulate new votes without enabling others to identify that said votee was the user who performed another specific reputation use elsewhere.

Exact Reputation Blinding: a mechanism is provided for votees to display or use their reputation without giving an exact score.

Server misbehaviour detection. As a secondary goal, PRSONA intends to make all server misbehaviour detectable. Put another way, when considering the covert setting, whatever misbehaviour is not outright prevented will identify the perpetrating servers.

Non-goals. PRSONA does not make any attempt to protect against Sybil attacks directly. This is considered out of scope for PRSONA; in order to participate, it is assumed that there is some mechanism to ensure that any given user is only registered in PRSONA once. PRSONA does not make any effort to prevent network-level Denial-of-Service attacks. This is a robust area of study, and many defences [1, 13, 16, 18, 26, 27, 34] that have already been designed for such attacks could be deployed in concert with PRSONA. PRSONA does not attempt to mitigate any potential stylometry attacks [22]. PRSONA's focus is on preventing structural leakage of sensitive user data rather than any content-based leakage.

4.4 Reputation Function

In PRSONA, a user's reputation represents their behaviour within a particular community. A low reputation indicates that a user misbehaves or is otherwise making themselves unwelcome, while a high reputation indicates that a user exemplifies the behaviour a community wishes to see. As users become dissatisfied with the behaviour of an individual, said individual receives feedback that allows them to course-correct. Further, users with high reputations cannot ignore honestly given feedback without repercussion.

Users (or servers) may choose a threshold reputation score, below which users face restrictions. Significant enough misbehaviour, as recognized by enough other users, would put an individual into either a warning or a restricted state. This is intended to address trolling and abuse; an individual who comes to a community intending only to disrupt it must face consequences. Choosing an appropriate threshold must be done with care.

In order to generate reputation scores in line with our expectations, we chose "Short-Term Memory Consensus — Weighted" as PRSONA's reputation function. Long-Term Memory Consensus can also be implemented with only minor alterations. As we discuss in previous work [12], in Short-Term Memory Consensus, each voter has one direct input to the reputation score of every other user. This input (vote) may be updated at any time. Due to this update function, a votee can experience relatively quick swings in their score when several voters switch their votes from positive to negative (or vice versa). "Weighted" indicates that a voter's inputs are weighted by their own (current) reputation. Short-Term Memory Consensus is an example of a *voter-conscious* reputation system — one that takes some aspect of the voter (*e.g.*, their reputation) into account when calculating their contribution to a votee's score.

4.5 Data Types

During setup, the PRSONA servers run **EGGroupKeyGen**(1^λ) and **BGNKeyGen**(1^λ) to obtain and make public the ElGamal and BGN parameters $(\mathfrak{G}, \mathfrak{g}, \mathfrak{h}, q)$ and $(G, G_1, H, H_1, G_T, e, g, h)$, respectively. They simultaneously distribute (π_1, π_2, π_T) to each other such that a threshold of servers must cooperate to execute **BGNDecrypt()**. In our instantiation, we require said threshold to be all servers, to maintain our privacy guarantees.

PRSONA operates in rounds or *epochs*. The PRSONA servers collaboratively maintain an **Epoch Generator** (that is, the group generator corresponding to the epoch) $\mathfrak{g}_t \in \mathfrak{G}$, where t represents

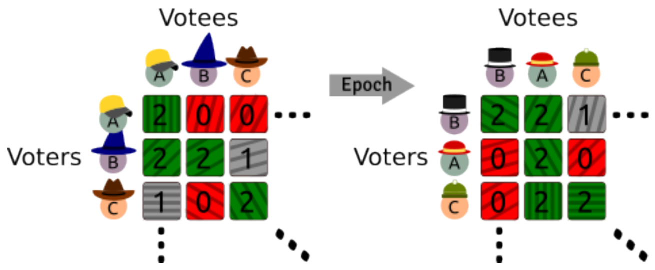

Figure 3. PRSONA uses a variant of Short-Term Memory Consensus (STMC) as its reputation function. In STMC, each user gets one replaceable vote for each other user. In this image, encryption is represented by the stripes on each vote. Here we can see the vote matrix shuffling and re-randomizing every vote between epochs, as well as the third row updating as voter C casts a new vote for votee A.

the current epoch. The PRSONA servers collaboratively generate $\mathfrak{g}_t$ once at the beginning of each epoch. When at least one server behaves honestly, the output $\mathfrak{g}_t$ itself is random.

A variety of data corresponds to PRSONA users during their participation in the system, some of which they hold directly, and some of which is held by the servers. A brief overview of this data follows; how it is actually maintained and used is elaborated in Section 5. Items marked (U) indicate they are held exclusively by users; (S) exclusively by servers; and (B) by both users and servers.

- **Long-Term Secret Key (U):** When users register to participate in the system, they must choose a long-term secret key $x \overset{\$}{\leftarrow} \mathbb{Z}_q^*$.
- **Fresh Pseudonym (B):** At all times, users have a pseudonym $X_t = \mathfrak{g}_t^x$ corresponding to the epoch t.
- **Plaintext Votes (U):** Voters give feedback for votees as votes. These votes in plaintext must be one of the following values: $\{0, 1, 2\}$ (*i.e.*, {"negative", "neutral", "positive"}). A voter is assumed to have a neutral opinion of any votee they have not explicitly rated.
- **Ciphertext Votes (B):** PRSONA servers receive encrypted votes, as outputs of **BGNEncrypt**$_H$**()**. These votes can only be decrypted by a threshold of servers. Each PRSONA server stores these in parallel as a **vote matrix**. The matrix is n-by-n, for n users in the system. Each row represents all the votes cast by one voter, and each column represents all the votes received by one votee. This vote matrix is permuted randomly each epoch. A visualization of the vote matrix can be seen in Figure 3.
- **Server-Encrypted Reputation List (S):** In order to calculate Short-Term Memory Consensus — Weighted, the previous epoch's reputation score for each user is required. The PRSONA servers store this as the output of **BGNEncrypt**$_G$**()**.
- **User-Encrypted Reputation List (B):** Users must be able to know and use their own reputation scores. These are generated as the output of **EGEncrypt()**, encrypted to the relevant user's new fresh pseudonym for the upcoming epoch.

5 DESIGN

As mentioned before, PRSONA operates in epochs. At the beginning of each epoch, users are assigned a new "fresh pseudonym", which is unlinkable to any previous or future fresh pseudonyms they have held or will hold. In practice, this fresh pseudonym is a public key, to which users are able to prove ownership of a corresponding long-term secret key with a straightforward zero-knowledge proof (ZKP): $\{(x) : X_t = \mathfrak{g}_t^x\}$. (We display all ZKPs in Camenisch-Stadler notation [6].)

With this fresh pseudonym, users are able to participate in a forum, by posting messages signed by their fresh pseudonym and evaluating other users according to their fresh pseudonyms. If a user posts multiple messages within an epoch, all such messages will be clearly associated with the same fresh pseudonym for that epoch. However, messages posted by the same user in different epochs (and thus under different fresh pseudonyms) are not able to be linked, unless the user explicitly chooses to link them through some external mechanism. If a user votes within an epoch, an adversary can learn that they updated their vote row, but will gain no information on whom they gave a vote for (if anyone), nor the content of any votes.

At the end of each epoch, the servers recalculate each user's reputation score. The servers are able to learn the distribution of scores, but not which user has which score. Simultaneously, the servers generate new fresh pseudonyms for each user and a new epoch generator, before beginning a new epoch.

5.1 User Registration

A new PRSONA user must register with the servers in order to participate. As mentioned previously, each epoch, the servers collaboratively calculate an epoch generator ($\mathfrak{g}_t \in \mathfrak{G}$, corresponding to epoch t) for the group the PRSONA servers generated at setup time via **EGGroupGenKey(1^λ)**. Before registering, a prospective user requests the epoch generator, signed by all servers. Then, the user k chooses $x_k \xleftarrow{\$} \mathbb{Z}_q^*$ and generates a key pair $\langle X_{k,t} = \mathfrak{g}_t^{x_k}, x_k \rangle$. $X_{k,t}$ is user k's fresh pseudonym for epoch t, and x_k is their long-term private key. The client then submits $X_{k,t}$ to a randomly selected server, along with the following ZKP: $\{(x_k) : X_{k,t} = \mathfrak{g}_t^{x_k}\}$.

Said server encrypts default (neutral) values for the user's votes on all existing users, all other existing users' votes for said new user, and the user's initial reputation score. As these default values are known to all, the randomness in the selection of $g_1 \xleftarrow{\$} G_1$ and $h_1 \xleftarrow{\$} H_1$ in **BGNEncrypt** can be omitted, simply setting g_1 to g' (the generator of G_1), and similarly for h_1. The other servers see the initial proof of a valid fresh pseudonym, confirm that the votes and score are correct (deterministic) encryptions of the appropriate defaults, and update their own data stores to add the new user.

It is possible for a covert server to ignore a user's request to be added. A user could easily detect such a case by requesting information from other servers to verify that their data stores have been updated. A user in this scenario would not be able to incriminate said server, but would not lose any privacy, and would be able to attempt registration again with a new server.

5.2 User Participation

Once a user is registered, they may participate in PRSONA in two ways: posting (which requires sufficient **Reputation**), and **Voting**.

5.2.1 *Reputation.*

Between each pair of epochs, a record of each user's reputation score is encrypted to their fresh pseudonym for the new epoch. This encrypted score is the output of **EGEncrypt()**, of the form $(C_1 = X_{k,t}^r, C_2 = \mathfrak{g}_t^r \cdot \mathfrak{h}^{z_k})$, for $r \xleftarrow{\$} \mathbb{Z}_q^*$ and z_k the user's score. They may request this record, and then use it to create a ZKP that their reputation score is above a given threshold: $\{(x_k', z_k) : X_t^{x_k'} = \mathfrak{g}_t \wedge z_k \in [\theta, 2n] \wedge C_2 = C_1^{x_k'} \cdot \mathfrak{h}_k^z\}$ where $x_k' = x_k^{-1}$, θ the publicly communicated threshold the user is above, and n the number of users. Note that $2n$ is the greatest possible score a user can have in PRSONA. The size and complexity of this ZKP is at worst logarithmic in the size of the range between θ and $2n$.

A verifier of this proof would request $(X_{k,t}, (C_1, C_2))$ directly from the servers; the response would be signed by a threshold of servers. Here, a majority of servers would be sufficient to prove correctness.

5.2.2 *Voting.*

Each post that a user makes will be tagged with their current fresh pseudonym. This allows other users to evaluate their behaviour and give feedback. These evaluations come in the form of votes, which are sent to servers as the output of **BGNEncrypt$_H$()** (such values having form $V = h^v \cdot h'^r$, for $v \in [0, 2]$ the plaintext vote and $r \xleftarrow{\$} \mathbb{Z}_p^*$).

When a user votes in PRSONA, they update all of their current votes. Their current votes are stored by the servers as a "vote row" encrypted to a threshold of the servers, and rerandomized and shuffled every epoch. Users know the current, but not the past, pseudonyms associated with each vote in their current row. For those users that a voter wishes to evaluate based on the most recent evidence, they replace existing votes, and for all others, they rerandomize existing votes.

To rerandomize a vote, a user chooses a random factor $r' \xleftarrow{\$} \mathbb{Z}_p^*$. They then calculate $V' = V \cdot h'^{r'}$, which, assuming V is well formed, is equivalent to $h^v \cdot h'^{r+r'}$, for their previous vote v for this votee, and for some unknown $r \in \mathbb{Z}_p$. The assumption that V is well formed is sound: in the case that a server is accepting a voter's submission, said server will verify the vote's correctness via a ZKP that the voter provides. In the case that a server is rerandomizing a vote as part of epoch changeovers, servers are either trusted to rerandomize votes correctly (honest-but-curious), or are required to provide a ZKP of correct rerandomization (covert).

Once this has been done for all of a voter's votes, the voter generates a ZKP over all said votes to prove that each is either a new encryption of a valid value (within the range of accepted votes) or a rerandomization of the vote that was already there: $\bigwedge_{k=1}^{n} \{(x, v_k', r') : X_t = \mathfrak{g}_t^x \wedge ((V' = h^{v_k'} h'^{r'} \wedge v' \in [0, 2]) \vee V' = V h'^{r'})\}$. Note that this ZKP is proportional in size and complexity to the number of users in PRSONA.

The voter submits this proof and their (new and/or rerandomized) votes to a randomly selected server. Upon verification of this proof, this server forwards the votes and the proof to the other servers, who verify the proof and update their data stores with the new set of votes. As with user registration, a covert server that ignores a user's vote will be easily detected by said user, without loss of privacy to the user, and said user can resubmit their votes to a new server.

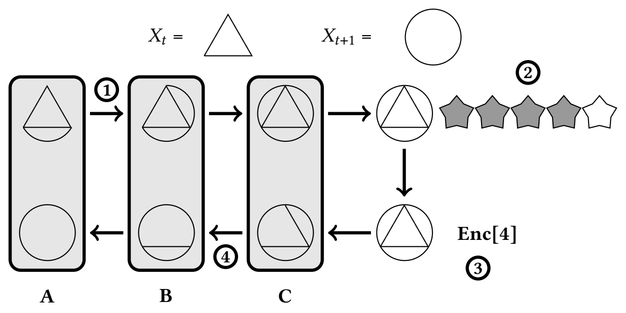

Figure 4. Following one user's pseudonym through an epoch changeover. In step 1 (Build-up Phase), servers A, B, and C add new factors to user X's epoch t fresh pseudonym. In step 2 (Decryption Phase), servers calculate the user's reputation. In step 3 (Re-encryption Phase), servers encrypt the user's reputation. In step 4 (Break-down Phase), servers remove a previous changeover's random factors to leave behind the epoch $t + 1$ fresh pseudonym.

5.3 Epoch Changeover

Between each pair of epochs, the servers must recalculate users' reputation scores, generate fresh pseudonyms for each user, and associate those reputations with the correct fresh pseudonym, without allowing fresh pseudonyms to be linked between epochs. Note that, while servers are computing the epoch changeover, certain functionalities of PRSONA are limited. Users can still make ZKPs that their reputation is above given thresholds, but new votes cannot be submitted, nor can new users be added.

As discussed in Section 4.5, each server holds several pieces of information. They track all of the fresh pseudonyms in a given epoch, along with the epoch generator. Note that the servers cannot know which fresh pseudonym in this epoch corresponds to which fresh pseudonym from any previous or future epoch, so long as any one server behaves honestly. The servers additionally track the matrix of encrypted votes and the previous epoch's calculated scores for each user, both of which are encrypted to the shared server BGN secret key (*i.e.*, they are the output of **BGNEncrypt**$_H$**()** and **BGNEncrypt**$_G$**()**, respectively). Finally, the servers also hold a list of the previous epoch's calculated scores for each user, where each encrypted score is encrypted to the user's fresh pseudonym (via **EGEncrypt()**).

In order to achieve PRSONA's functionality, subject to its privacy guarantees, the inter-epoch calculation proceeds in four rounds, as depicted in Figure 4:

1. **Build-up Phase:** The servers calculate the next epoch's epoch generator, and iteratively raise each user's previous epoch fresh pseudonym to new factors, putting the pseudonyms into a transitional state that cannot be linked to their values in the previous or following epoch. Each server also randomly shuffles the data.
2. **Decryption Phase:** The servers calculate the reputation scores for the next epoch and collaboratively decrypt them. These reputation scores will then be seen by the servers in plaintext, associated only with the transitional pseudonyms.
3. **Re-encryption Phase:** The servers re-encrypt the plaintext reputation scores into the output of **BGNEncrypt**$_G$**()** (used by the servers during the next Decryption Phase) and the output of **EGEncrypt()** (used by votees in reputation proofs). During this phase, the latter of these will be encrypted to the transitional pseudonyms, to be altered in the Break-down Phase such that they are correctly encrypted to the user's new fresh pseudonym. (This is necessary to avoid linking fresh pseudonyms to users' raw scores, which are observed during the Decryption and Re-encryption Phases.)
4. **Break-down Phase:** The servers iteratively raise the transitional pseudonym (and part of their encrypted reputation score) for each user to the inverse of the factors added during the previous inter-epoch calculations, leaving only the factors they used during *this* set of inter-epoch calculations. Once this is done, the resulting fresh pseudonyms (and encrypted reputation scores) are ready for the new epoch. Each server also randomly shuffles the data.

5.3.1 Build-up Phase. In the first phase, the servers "build up" on top of users' previous epoch fresh pseudonyms (we will call this epoch t), in addition to building up a new fresh generator for the next epoch ($t + 1$).

To do so, the servers begin with the current fresh pseudonyms and $\mathfrak{g}$, the generator of the group $\mathfrak{G}$ that was output by **EGGroup-GenKey()**. Each server operates in turns; the order does not strictly matter, but must be agreed upon ahead of time for purposes of synchonization. Each server chooses a random epoch factor $\mathfrak{r}_{k,t+1} \xleftarrow{\$} \mathbb{Z}_q^*$. The epoch generator for the next epoch is the result of each server iteratively applying (through exponentiation) their epoch factor to $\mathfrak{g}$.

The server also applies its epoch factor to each of the fresh pseudonyms (or the output transitional pseudonyms from the previous server, as appropriate). It then randomly shuffles the transitional pseudonyms (along with every piece of data tagged by the pseudonyms, with the same random shuffle). In the honest-but-curious setting, the server passes this shuffled data on to the next server. As the honest-but-curious algorithm is dominated by its need to rerandomize $O(n^2)$ elements of the vote matrix, and rerandomization is an $O(1)$ operation, we expect the honest-but-curious setting to have quadratic complexity. In the covert setting, server k generates a ZKP that it applied the same shuffle to all the data, and that the exponentiations and rerandomizations were all consistent, then passes the data and ZKP to all other servers, who verify the ZKP before the next server begins their turn.

The intuition for this ZKP is as follows. Imagine that the list of fresh pseudonyms is a vector. Next, imagine a permutation matrix — a matrix whose elements are all exclusively 0 or 1, and for which it is true that every row and every column sums to 1 — for which the matrix-vector product with the input list of pseudonyms is the list of pseudonyms in the desired shuffled order. In the ZKP, a server commits to such a permutation matrix, then proves that the output fresh pseudonyms are the result of raising each element of the matrix-vector product of that permutation matrix and the input fresh pseudonyms to a common value. Further, the server proves the other reordered datasets are the result of homomorphically adding encryptions of the value 0 to the matrix-vector product of the same permutation matrix and the appropriate input data.

We show the notation for the ZKP in parts. First, we must prove that we have a valid permutation matrix (**L**). Let us call each element of **L** $\ell_{i,j}$. We make a matrix of Pedersen commitments to this permutation matrix (**B**), whose elements are $B_{i,j} = \mathfrak{g}^{\ell_{i,j}}\mathfrak{h}^{s_{i,j}}$. We choose these $s_{i,j}$ values such that $\sum_{i=1}^{n} s_{i,j} = 0 \bmod q$ for each j and $\sum_{j=1}^{n} s_{i,j} = 0 \bmod q$ for each i. We prove that each $B_{i,j}$ is a commitment to 0 or 1: $\{(\ell_{i,j}, s_{i,j}) : B_{i,j} = \mathfrak{g}^{\ell_{i,j}}\mathfrak{h}^{s_{i,j}} \wedge \ell_{i,j} \in [0,1]\}$. With that, assuming that no server can know $d \in \mathbb{Z}_q$ such that $\mathfrak{g} = \mathfrak{h}^d$, a verifier can check that $\prod_{i=1}^{n} B_{i,j} \stackrel{?}{=} \mathfrak{g}$ for all j and $\prod_{j=1}^{n} B_{i,j} \stackrel{?}{=} \mathfrak{g}$ for all i. If this holds, each row and column of the matrix sums to 1.

Some stored data only needs to be rerandomized and reordered by the permutation matrix. This applies to the vote matrix (**V**) and to the server-encrypted reputation scores. In order to correctly permute both the rows and the columns of **V**, we specifically calculate $\mathbf{L}^T\mathbf{V}\mathbf{L}$. This calculation is carried out on each row (or column) individually, using the same process as any other input data vector we shuffle and rerandomize. Suppose $\vec{a} = (A_1, \ldots, A_j, \ldots A_n)$ is such a vector we want to reorder. (Recall that the elements of the vote matrix and the server-encrypted reputation scores will be in G and H, respectively). In addition to the Pedersen commitment permutation matrix from before, we generate an intermediary matrix **C**, whose elements are $C_{i,j} = A_j^{\ell_{i,j}} h'^{s'_{i,j}}$, where $s'_{i,j} \xleftarrow{\$} \mathbb{Z}_p^*$. With this in place, we include the following ZKP part:

$$\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n}\{(\ell_{i,j}, s_{i,j}, s'_{i,j}) : B_{i,j} = \mathfrak{g}^{\ell_{i,j}}\mathfrak{h}^{s_{i,j}} \wedge C_{i,j} = A_j^{\ell_{i,j}} h'^{s'_{i,j}}\}$$

Similar to the permutation matrix before, the verifier then calculates $\vec{c_i} = \langle \prod_{j=1}^{n} C_{i,j} \rangle_{i\in[1,n]}$. As there is only one 1 in each row of the permutation matrix, most of the $A_j^{\ell_{i,j}}$ terms drop away, and the remaining $h'^{s'_{i,j}}$ terms rerandomize the element. The resulting c_i values are the re-ordered and rerandomized vector we sought.

Note that the size and complexity of this ZKP scales quadratically with the number of elements being reordered. This is true of all the ZKP parts we describe, as they all involve transforming an n-element vector into an n-by-n element matrix (*i.e.*, to touch every element in the resulting matrix, it is necessarily the case that there are $O(n^2)$ operations). However, the overall size and complexity of the ZKP for a correct shuffle is actually cubic — this is because in order to reorder a vote matrix specifically, there are n different n-element vectors that must be transformed into n-by-n element matrices (that is, there are $O(n^2)$ operations per matrix, and n matrices, resulting in an $O(n^3)$ complexity).

Some stored data needs to be reordered by the permutation matrix while also applying the server's epoch factor $\mathfrak{r}_{k,t+1}$ to them, without doing any form of rerandomization. This is true of the fresh pseudonyms. As before, we will make use of the Pedersen commitment permutation matrix. We also use two new intermediary matrices, **D** (whose elements are $D_{i,j} = A_j^{\ell_{i,j}\mathfrak{r}_{k,t+1}}\mathfrak{h}^{s'_{i,j}}$) and **E** (whose elements are $E_{i,j} = \mathfrak{g}^{s'_{i,j}}$). In this case, the $s'_{i,j}$s are chosen such that $\sum_{j=1}^{n} s'_{i,j} = 0 \bmod q$ for each i. With this in place, we include the following ZKP part:

$$\bigwedge_{i=1}^{n}\bigwedge_{j=1}^{n}\{(\mathfrak{r}_{k,t+1}, \ell_{i,j}, s_{i,j}, s'_{i,j}) : B_{i,j} = \mathfrak{g}^{\ell_{i,j}}\mathfrak{h}^{s_{i,j}} \wedge D_{i,j} = A_j^{\ell_{i,j}\mathfrak{r}_{k,t+1}}\mathfrak{h}^{s'_{i,j}} \wedge E_{i,j} = \mathfrak{g}^{s'_{i,j}}\}$$

Again as before, the verifier then calculates $D_i = \prod_{j=1}^{n} D_{i,j}$ and $\prod_{j=1}^{n} E_{i,j} \stackrel{?}{=} \mathfrak{g}^0$. Assuming the $E_{i,j}$ check passes (confirming that the $s'_{i,j}$s canceled out), then the D_is are the reordered fresh pseudonyms with the $\mathfrak{r}_{k,t+1}$ factor applied. The user-encrypted reputation scores are not reordered during this phase, as they will be recalculated and replaced in the next two phases.

For all of these ZKP parts, proof batching techniques [14] can be used to make the proofs more efficient, at the expense of introducing a potential soundness error. Proof batching involves choosing a batch parameter λ, and the chance of error is typically something along the lines of $2^{-\lambda}$. As such, $\lambda = 50$ would indicate that a proof that is generated once per second would not be expected to incorrectly verify for 2^{24} years. In our implementation, we also implement proof batching, and test with $\lambda = 50$; this choice is appropriate for the usage expected with PRSONA in practical settings.

At the end of this process, the servers have collaboratively generated the epoch generator for epoch $t+1$, and a set of transitional pseudonyms that are not linkable to either the fresh pseudonyms for the previous or next epoch (we use the notation $X_{\mathrm{I/II}}$ to refer to such a pseudonym for user X between Epochs I and II). In this state, none of the users and none of the servers individually are able to determine which of the transitional pseudonyms apply to which user.

5.3.2 *Decryption and Re-encryption Phases.*

The next two phases are closely related to each other. Here, there is no difference between the covert setting and the honest-but-curious setting. Once the transitional pseudonyms have been "built up", the servers collaboratively calculate the next epoch's reputation scores for each user. Votes in the vote matrix are encrypted as elements in H, and associated with both the voter's and votee's transitional pseudonyms. Reputation scores from Epoch t are encrypted as elements in G, and associated with the transitional pseudonym of the user they apply to; let $\vec{z}$ be the vector form of these scores when in plaintext, and $\vec{a}$ when in ciphertext. Each server has its own copy of this data.

First, as a setup step, servers independently compute (in encrypted form) the matrix-vector product of the vote matrix and the reputation scores from Epoch t by using **BGNMultiply()** and **BGNAdd()** as appropriate (this step uses no secrets, so using fixed g_1 and h_1 values to ensure that the servers calculate identical output values is fine). Specifically, each element in the vote matrix is multiplied by the reputation score of its voter, then added with the other votes for the same votee, to achieve the weighting of our reputation function. The resulting vector consists of elements of G_T, in the same order as $\vec{z}$ (and therefore still associated with the correct transitional pseudonyms).

Note that PRSONA reputation scores are limited to the range $[0, 2n]$. However, these output values can be in range $[0, 4n^2]$ (the maximum occurring when every voter has reputation score $2n$ and gives a positive vote to the same votee). To resolve this, output values are scaled according to the maximum score that was possible to achieve during the epoch. To do so, servers at this point also independently homomorphically calculate the encryption of the sum of all elements of $\vec{z}$ by multiplying the ciphertexts in$\vec{a}$, then squaring the result to calculate the encryption of $m = 2 \sum_{Z_k \in \vec{z}} Z_k$, the maximum possible score. Because this calculation involves no multiplications, the resulting value is an element of G, just as the members of $\vec{a}$ are.

Next, the Decryption Phase begins. Servers confirm that they hold the same values in the output vector and for the encryption of m, then collaboratively execute **BGNDecrypt()** on these values. From there, each server has a plaintext version of the output vector, with each element tagged with the transitional pseudonym of the user it applies to, and a plaintext version of the maximum possible score. The servers thus learn the multiset of reputation scores held by users each epoch. Servers then independently calculate $z'_k = \left\lfloor \frac{2nz_k}{m} \right\rfloor$, for z_k the plaintext element of the output vector corresponding to user k. The resulting $\vec{z'}$ represents the plaintext reputation scores scaled into the range $[0, 2n]$, as they will apply to each user in the next epoch.

Once these scaled plaintext values have been calculated, servers move forward to the Re-encryption Phase. In this phase, servers encrypt the plaintext scores in two ways. First, they encrypt them as elements of G using **BGNEncrypt$_G$()**, to serve as the weights in the reputation function calculation during the next inter-epoch period. As PRSONA does not prevent learning the multiset of reputation scores, servers use fixed g_1 in this encryption, so that each server can obtain an identical copy of these newly encrypted values. If it is desired to prevent users that do not collude with servers from learning this multiset, the servers may instead agree on a random g_1 (whether at the time or in advance). Second, servers encrypt the scores as elements of $\mathfrak{G}^2$ using a variation of **EGEncrypt()**.

Ordinarily, the output of **EGEncrypt()** is encrypted to a pseudonym $X_{k,t}$ with form $(X^r_{k,t}, \mathfrak{g}^r_t \mathfrak{h}^{z'_k})$, for $r \in \mathbb{Z}^*_q$ some random blinding factor and z'_k the scaled score of the user in plaintext. However, the servers have not yet calculated the fresh pseudonyms X_{t+1}. Fortunately, they did calculate $\mathfrak{g}_{t+1}$ during the Build-up Phase. With this, the servers encrypt a score z_k like so: $(X^r_{k,t/t+1}, \mathfrak{g}^r_{t+1} \mathfrak{h}^{z'_k})$. Any individual server can compute this encryption, but a majority of servers must agree that it represents the correct score.

Users cannot normally decrypt these values, because $X_{k,t/t+1} \neq \mathfrak{g}^{x_k}_{t+1}$. During the Break-down Phase, the servers will manipulate these encrypted values to put them back into a form that users will be able to correctly decrypt, while also rerandomizing the encryptions with r' blinding factors that the other servers do not know. Due to this rerandomization aspect, these encryptions will not be possible for other servers to decrypt when associated with users' fresh pseudonyms, as long as any one server honestly rerandomized the scores. As with the outputs of **BGNEncrypt()**, appropriate values of r can be agreed upon at the time or pre-emptively, so that each server has an identical copy of the outputs of **EGEncrypt()**.

5.3.3 Break-down Phase. In the final phase, the servers "break down" transitional pseudonyms into the fresh pseudonyms for the next epoch (along with the user-encrypted reputation scores, in the same manner).

During Epoch t, servers each chose a (random) epoch factor $\mathfrak{r}_{k,t}$ to apply to $\mathfrak{g}$, in order to collaboratively construct $\mathfrak{g}_t$. The transitional pseudonyms are related to the fresh pseudonyms for Epoch t, as they still have those epoch factors from Epoch t applied to them. As in the Build-up Phase, each server operates in turns, and again the order does not strictly matter, but must be agreed upon ahead of time for purposes of synchronization. During server k's turn, first, it calculates $\mathfrak{r}^{-1}_{k,t} \bmod q$. Then, k applies this inverse epoch factor to each of the transitional pseudonyms (or the output transitional pseudonyms from the previous server, as appropriate). As in the Build-up Phase, the server also randomly shuffles the transitional pseudonyms (along with every piece of data tagged by the pseudonyms, with the same random shuffle).

In the honest-but-curious setting, the server passes this shuffled data on to the next server. As with the Build-up Phase, rerandomizing the vote matrix dominates the honest-but-curious Break-down Phase algorithm, resulting in an expected quadratic computational complexity. Across all phases, the greatest complexity any phase faces in the honest-but-curious setting is a quadratic complexity, and each phase executes independently. This leads us to conclude that the honest-but-curious setting should expect quadratic complexity for the epoch changeover calculations as a whole.

In the covert setting, server k generates a ZKP that it applied the same shuffle to all the data, and that the exponentiations and rerandomizations were all consistent, then passes the data and the ZKP to all other servers, who verify the ZKP before the next server begins their turn. This ZKP is almost identical to that of the Build-up Phase. Wherever the Build-up Phase refers to $\mathfrak{r}_{k,t+1}$, it can simply be replaced with $\mathfrak{r}^{-1}_{k,t} \bmod q$ without issue. The ZKP for the Break-down Phase has one notable addition on top of the ZKP for the Build-up Phase. During the Build-up Phase, user-encrypted reputation scores were ignored, because they are recalculated during the Decryption and Re-encryption Phases. During the Break-down Phase, however, user-encrypted reputation scores cannot be ignored. Their shuffle requires a slightly altered ZKP from the two types of data that have ZKPs during the Build-up Phase; this ZKP simultaneously rerandomizes shuffled data and applies a consistent exponent to each element.

Recall that user-encrypted reputation scores are (usually) the output of **EGEncrypt()** with form $(X^r_t, \mathfrak{g}^r_t \mathfrak{h}^z)$, for X_t the fresh pseudonym of some user during Epoch t, r a random blinding factor, and z the user's reputation score. Recall also that the "user-encrypted reputation scores" obtained during the Re-encryption Phase actually have form $(X^r_{t/t+1}, \mathfrak{g}^r_{t+1} \mathfrak{h}^z)$, because servers did not have access to the fresh pseudonyms for Epoch $t+1$ during the Re-encryption Phase. As the ciphertext is a tuple, there are two different operations that need to be done on each piece of the tuple. The first part, much like the fresh and transitional pseudonyms during the Build-up Phase, need to have a factor applied to them in addition to being reordered (specifically, $\mathfrak{r}^{-1}_{k,t} \bmod q$). The second part, much like the vote matrix and server-encrypted reputation scores, needs to be reordered and rerandomized — and importantly, the same rerandomization needs to *also* be applied to the first part of the tuple, so that the encrypted value can still be properly decrypted after the shuffle operation.

As with the ZKP parts defined during the Build-up Phase, for this ZKP part we will make use of the Pedersen commitment permutation matrix **B**. In this ZKP part, A'_j describes the first element of the j-th tuple in the encrypted values arranged as a vector, and A''_j describes the second element of said tuple. $X_{k,t/t+1}$ refers to user k's transitional pseudonym (or the appropriate output transitional pseudonym of the previous server). We make use of intermediate vectors **E** (whose elements are $E_{i,j} = \mathfrak{g}^{s'_{i,j}}$), **F** (whose elements are $F_{i,j} = A_j'^{\ell_{i,j}\mathfrak{r}_{k,t}^{-1}} X_{j,t/t+1}^{\ell_{i,j}\mathfrak{r}_{k,t}^{-1}s'_{i,j}} \mathfrak{h}^{s'_{i,j}}$), and **H** (whose elements are $H_{i,j} = A_j''^{\ell_{i,j}} \mathfrak{g}_{t+1}^{\ell_{i,j}s'_{i,j}} \mathfrak{h}^{s'_{i,j}}$). As with the Build-up Phase, the $s'_{i,j}$ values are chosen such that $\sum_{j=1}^{n} s'_{i,j} = 0 \bmod q$ for each i. With this in place, we include the following ZKP part, which proves that the output user-encrypted reputation scores are consistent with rerandomizing and removing the server's previous epoch factor from the input user-encrypted reputation scores:

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \{(\mathfrak{r}_{k,t}^{-1}, \ell_{i,j}, s_{i,j}, s'_{i,j}) : B_{i,j} = \mathfrak{g}^{\ell_{i,j}} \mathfrak{h}^{s_{i,j}} \wedge E_{i,j} = \mathfrak{g}^{s'_{i,j}} \wedge$$
$$F_{i,j} = A_j'^{\ell_{i,j}\mathfrak{r}_{k,t}^{-1}} X_{j,t/t+1}^{\ell_{i,j}\mathfrak{r}_{k,t}^{-1}s'_{i,j}} \mathfrak{h}^{s'_{i,j}} \wedge H_{i,j} = A_j''^{\ell_{i,j}} \mathfrak{g}_{t+1}^{\ell_{i,j}s'_{i,j}} \mathfrak{h}^{s'_{i,j}}\}$$

As was done in the Build-up Phase, the verifier then calculates $\prod_{j=1}^{n} E_{i,j} \overset{?}{=} \mathfrak{g}^0$, $\vec{f_i} = \langle \prod_{j=1}^{n} F_{i,j} \rangle_i$, and $\vec{h_i} = \langle \prod_{j=1}^{n} H_{i,j} \rangle_{i \in [1,n]}$. In the $\vec{f_i}$ calculations, the $h^{s'_{i,j}}$ terms drop away as expected, all but one of the $A_j^{\ell_{i,j}\mathfrak{r}_{k,t}^{-1}}$ terms drop away, and the one $X_{j,t/t+1}^{\ell_{i,j}\mathfrak{r}_{k,t}^{-1}s'_{i,j}}$ term that does not drop away rerandomizes the first part of the tuple appropriately. In the $\vec{h_i}$ calculations, the $h^{s'_{i,j}}$ terms again drop away as expected, all but one of the $A'_{j^{\ell_{i,j}}}$ terms drop away, and the one $\mathfrak{g}_{t+1}^{\ell_{i,j}s'_{i,j}}$ term that does not drop away rerandomizes the second part of the tuple (with the same random factor as was applied to the first part of the tuple). The resulting $\vec{f_i}$ and $\vec{h_i}$ values are the re-ordered and rerandomized elements of the ciphertexts.

Because the vote matrix portion of the ZKP still dominates in the Break-down Phase, we expect the overall size and complexity of this ZKP to be cubic. As each phase operates independently, this leads us to conclude that we should expect cubic complexity for the epoch changeover calculations as a whole in the covert setting.

6 EVALUATION

In this section, we discuss our implementation of PRSONA and evaluate it on suitable benchmarks.

6.1 Implementation

We implemented a functional prototype of PRSONA, consisting of approximately 11400 lines of novel C++ code. Our implementation makes use of an open-source C++ library implementing prime-order BGN from Herbert *et al.* [15], upon which we have made significant extensions (totalling approximately 2700 lines of C++ code). That library in turn uses an open-source C library implementing various elliptic curve and pairing primitives from Naehrig *et al.* [21]. Our prototype implements the complete design of PRSONA, including all relevant zero-knowledge proofs. The source code of our prototype is available at https://git-crysp.uwaterloo.ca/tmgurtler/PRSONA, and the source code of the modifications we made to the BGN library is available at https://git-crysp.uwaterloo.ca/tmgurtler/BGN2.

The prime-order BGN library that we use implements prime-order BGN (as described in Section 3.2) over a Barreto-Naehrig curve — the library specifically builds upon a previous implementation by Naehrig *et al.* [21]. We also implement ElGamal encryption (as described in Section 3.1) over this same curve (note that BN pairings are Type III, so this is secure).

In Section 5.3, we mention that proof batching techniques [14] can be applied to the ZKPs used in our proofs of correct shuffles. Our implementation includes such techniques. As mentioned before, our benchmarks in this chapter that use proof batching are measured with $\lambda = 50$ (for λ the negative log of the soundness error induced by batching).

6.2 Evaluation

We deployed servers and clients on the CrySP RIPPLE Facility; each individual machine had 1 TiB of RAM and 80 cores and was used for multiple clients and servers. The maximum RAM usage for servers in our largest experiments never exceeded 30 GiB, and the clients' cryptographic calculations use no more than 3 MiB of RAM. In all timing calculations we perform, the effect of bandwidth is small (each machine is connected to the others through a 1 Gbps network). Timings accounting for lower bandwidth may be simulated via the recorded data for how much bandwidth was used, but as will be seen, CPU usage has a much more significant impact on the time any operation takes than the time necessary to transmit the data PRSONA requires during these operations.

Throughout our experiments, we benchmarked the system with $n = \{5, 10, 15, 20, 25, 30, 40, 50\}$ users connected. These benchmarks were run as a proof of concept and to confirm our analysis of PRSONA's asymptotic behaviour; with the optimizations (the hybrid honest-but-curious and covert approach, as well as probabilistic proofs) that will be discussed in Section 7, we expect that PRSONA could reasonably support several hundred users. As a rough estimate, using the hybrid approach, an epoch length of one day, and requiring servers to prove a random 1% of their covert proofs, we estimate PRSONA could support 600 users with no further optimizations. Our expected use case involves small, tight-knit communities. Recent statistical re-analysis of Dunbar's number gives its 95% confidence interval upper bound at 520 [20]; as such, 600 users seems to be an appropriate limit for PRSONA's use case.

In our experiments, we benchmark three key operations of PRSONA. We are concerned with making proofs that reputations are above a threshold (done by clients), with making votes (done by clients), and with the calculations involved in turning over to a new epoch (done by servers).

Reputation Proofs (client side). As mentioned in Section 5.2.1, the size and time needed to create reputation proofs is logarithmic in the difference between the chosen threshold θ and $2n$. We find that both generating and verifying a reputation proof takes a very small amount of time (less than 0.025 s in all cases tested). Proof sizes are very small as well (less than 3 KiB in all cases we tested). Scaling up beyond proof-of-concept cases, we estimate making a reputation proof would take less than 0.04 s in the 600-user case, with the size of a new reputation proof at approximately 5 KiB.

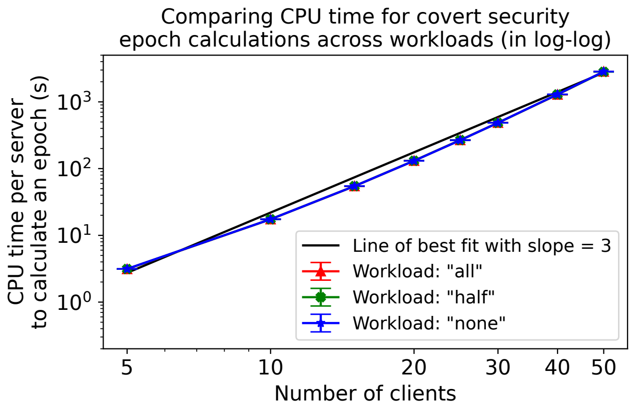

Figure 5. Comparing the CPU time required to perform epoch calculations with covert security to a cubic curve. On a log-log graph, epoch calculation CPU time is roughly a straight line with slope ≈ 3. A 2-server setup was used in this experiment. Proof batching was used in this experiment, with $\lambda = 50$.

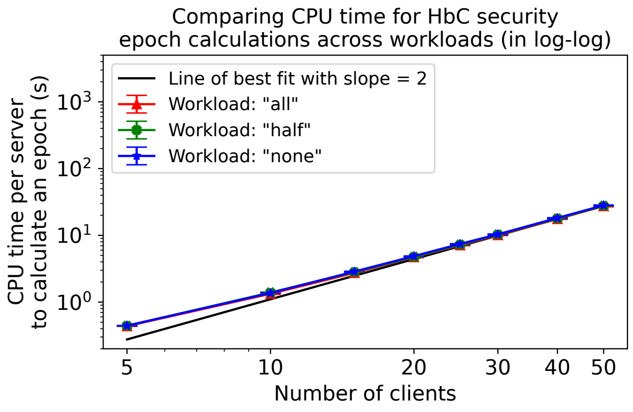

Figure 6. Comparing the CPU time required to perform epoch calculations with honest-but-curious security to a quadratic curve. On a log-log graph, epoch calculation CPU time is roughly a straight line with slope ≈ 2. A 2-server setup was used in this experiment. Proof batching is not relevant in the honest-but-curious setting.

Even when scaling up, these operations are very reasonable for users to engage with regularly as needed.

New Votes (client side). As mentioned in Section 5.2.2, the size and time needed to create new rows of votes is linear in the number of users in the system as a whole. We find that in our largest cases, making new vote rows takes less than 1.0 s, and the sizes of their proofs are less than 350 KiB. Scaling up beyond proof-of-concept cases, we estimate making a new vote row (once per epoch) would take approximately 11 s in the 600-user case, with the size of a new vote row proof at approximately 4 MiB. Even as we expect users of the system to have less powerful machines or less bandwidth than servers, making new vote rows will be more or less unnoticeable to the average user.

Epoch Calculations (server side). When benchmarking epoch changeover calculations, we tested three workloads, corresponding to differing numbers of voters voting in each epoch. In the "none" workload, no clients vote during the epoch, then an epoch changeover was calculated, then one reputation proof was made. In the "half" workload, half the clients are randomly selected to vote during the epoch, with the other two steps remaining the same. In the "all" workload, all the clients vote during the epoch, and the other two steps stayed the same. When comparing these workloads under covert security (Figure 5), the proof batching techniques discussed in Section 5.3 were used, with batch soundness parameter $\lambda = 50$. Proof batching is not relevant in the honest-but-curious setting (HbC) (Figure 6). We repeated each experiment 25 times.

As can be seen in Figures 5 and 6, epoch calculations in both covert and honest-but-curious security had virtually no difference in CPU time between workloads. All workload curves take the same shape, with only very minor differences between them; for most values of the number of clients observed, their 95% confidence intervals overlap. We note that, as expected, the honest-but-curious cases (Figure 6) have significant performance improvements over the covert cases (Figure 5) in CPU time. These lower costs may make the system more attractive to deploy, at the cost of requiring more trust in the central servers. Alternatively, in the hybrid approach that will be discussed in Section 7, these represent lower *online* costs with no extra trust in the central servers. We estimate that in a 300-user case, servers could execute a full covert proof in one week, and in a 600-user case, servers could execute a random 1% of the covert proof in one day. The proofs themselves could also be invoked only probabilistically (on average say once per month for the full 300-user proof, and once per week for the probabilistic 600-user proof).

In Figures 5 and 6, we compare the graphs with a simple cubic curve (a straight line with slope 3 on the log-log plot) in the covert setting, and a simple quadratic curve (slope 2) in the honest-but-curious setting. In the covert setting, the graph is roughly a straight line with slope ≈ 3; in the honest-but-curious setting, the graph is roughly a straight line with slope ≈ 2. This indicates, as expected in Section 5.3, that the epoch calculation has approximately cubic complexity in the covert setting, and approximately quadratic complexity in the honest-but-curious setting.

We also consider cases with different number of servers, as in Figures 7 and 8. In Figure 7, we observe that the per-server CPU time to calculate an epoch changeover has some small additional overhead among greater numbers of clients, but this overhead is barely noticeable with smaller numbers of clients. In Figure 8, we similarly observe little discernable difference between different number of servers in per-server bandwidth. Comparing both CPU usage and server-to-server outgoing bandwidth to a cubic curve on a log-log graph, as in previous cases in the covert setting, graphs are roughly straight lines with slope ≈ 3, indicating approximately cubic complexity. Importantly, all lines have approximately the same slope; this indicates that the relative overhead increases with the same complexity. Together, this tells us that distributing trust across more entities does not require exorbitant costs.

7 DISCUSSION

Conducting the epoch changeover calculations in the covert setting is computationally demanding. The bandwidth usage we require of servers and the times involved in calculating epoch changeovers can be relatively high. As mentioned elsewhere, we target PRSONA for usage in smaller, tight-knit communities, where we expect group

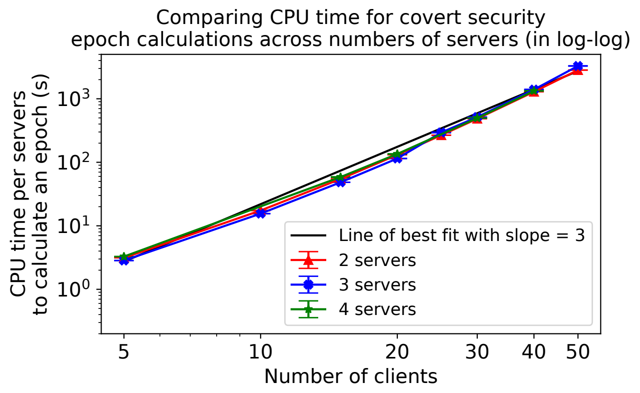

Figure 7. Comparing the covert security epoch calculation CPU time of different numbers of servers to a simple cubic curve. On a log-log graph, epoch calculation CPU time is roughly a straight line with slope ≈ 3. Datapoints represent a mix of workloads. Proof batching was used in this experiment, with $\lambda = 50$.

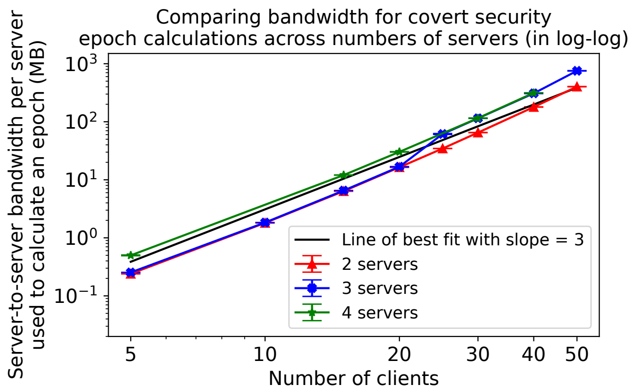

Figure 8. Comparing the covert security epoch calculation bandwidth of different numbers of servers to a simple cubic curve. On a log-log graph, epoch calculation bandwidth is roughly a straight line with slope ≈ 3. Datapoints represent a mix of workloads. Proof batching was used in this experiment, with $\lambda = 50$.

sizes to not be exceptionally large. For the largest of our test cases, PRSONA epoch changes do take significant amounts of time in the covert setting (several hours of total CPU time). This is still fairly unoptimized — there are a considerable number of matrix operations that could be parallelized, for example. On the other hand, in the honest-but-curious setting, epoch changes take only a few minutes of total CPU time.

Though we have considered the honest-but-curious setting and the covert setting separately to this point, we note that there is a way to combine the two modes for epoch calculations. If a server conducts their shuffle in the honest-but-curious setting, but holds on to the random shuffle orderings they used in the Build-up and Break-down Phases, as well as the blinding factors used to rerandomize elements, they would be able to retroactively produce the proof they would have needed for the covert setting. Thus, servers can conduct the shuffle using the honest-but-curious setting in real time, preventing the system from being blocked for long periods of time. Afterwards, they can retroactively produce the proofs of the covert setting, and verify them amongst each other, to confirm that all servers carried out their shuffle correctly. Because a server who did not conform to the protocol would be eventually incriminated, a covert server would not be able to act maliciously in this case. We also consider probabilistic proofs of two forms. First, we consider cases where servers perform all the quadratic portions and a (jointly selected by all servers) random fraction of the cubic portions of the covert proof. In particular, the 600-user case we discuss in Section 6.2 performs this probabilistic proof with a random 1% of the cubic proofs. Second, a proof may be asked for probabilistically; that is, it may be asked for randomly with some parameterized probability. An implementer may choose to set how frequently they desire servers to actually produce their covert setting proofs. For both, a covert server would still be prevented from cheating, because of the non-negligible probability of their proof being required. Note that, under these conditions, a malicious server would not have been so prevented.

With this hybrid approach, an implementer gains security against a covert adversary while only requiring the online cost of the honest-but-curious setting. Though there is still an offline cost in line with the non-hybridized covert setting, the system would not block for long periods of time, meaning that users would be unable to vote on each other for only a few minutes during each epoch change. If the epoch is set to change every day, or every other day, users would receive the benefits of fresh pseudonyms on a timescale appropriate for the speed of forum conversations, and the system would be able to support a reasonable number of users. Given what we have observed from these benchmarks, we conclude that PRSONA is appropriate for use in such settings.

8 CONCLUSION

Naturally, there remains room for improvement in the design of reputation systems. Our approach focuses on smaller communities, but aspects of our design can still be applied to larger contexts. In particular, larger contexts do not often employ Reputation-Usage Unlinkability, as PRSONA does. The ability for users to use reputation without linking themselves can be very valuable, regardless of the size of the userbase, and future work exploring this opportunity could be fruitful.

PRSONA itself is designed to assist tight-knit communities in empowering their members to connect with one another with the freedom and safety of privacy and anonymity, while balancing against the risk of bad actors abusing those privileges. PRSONA implements its guarantees with straightforward cryptographic tools, and its experimental performance supports its viability for its target audience. PRSONA and similar systems can help enable more human-scale social media, supporting people to connect and grow.

ACKNOWLEDGMENTS

We thank NSERC (CRDPJ-534381) and the Royal Bank of Canada for supporting this work. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

REFERENCES

[1] Sharad Agarwal, Travis Dawson, and Christos Tryfonas. DDoS mitigation via regional cleaning centers. Technical report, Sprint ATL, January 2004.

[2] Carlos Aguilar Melchor, Boussad Ait-Salem, and Philippe Gaborit. A collusion-resistant distributed scalar product protocol with application to privacy-preserving computation of trust. In *2009 Eighth IEEE International Symposium on Network Computing and Applications*, pages 140–147, July 2009.

[3] Jay Allen. The invasion boards that set out to ruin lives. https://boingboing.net/2015/01/19/invasion-boards-set-out-to-rui.html, January 2015.

[4] Yonatan Aumann and Yehuda Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In Salil P. Vadhan, editor, *Theory of Cryptography*, pages 137–156, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[5] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In Joe Kilian, editor, *Theory of Cryptography*, pages 325–341, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[6] Jan Camenisch and Markus Stadler. Efficient Group Signature Schemes for Large Groups. In *CRYPTO 1997*, pages 410–424, 1997.

[7] CBS News. Facebook whistleblower Frances Haugen testifies before Senate committee | full video. https://www.youtube.com/watch?v=juZEkeTjTRY, October 2021.

[8] Tassos Dimitriou. Decentralized reputation. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy*, CODASPY '21, pages 119–130, New York, NY, USA, 2021. Association for Computing Machinery.

[9] Robin Ian MacDonald Dunbar. Neocortex size as a constraint on group size in primates. *Journal of Human Evolution*, 22(6):469–493, 1992.

[10] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985.

[11] David Mandell Freeman. Converting pairing-based cryptosystems from composite-order groups to prime-order groups. In Henri Gilbert, editor, *Advances in Cryptology — EUROCRYPT 2010*, pages 44–61, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[12] Stan Gurtler and Ian Goldberg. SoK: Privacy-preserving reputation systems. *Proceedings on Privacy Enhancing Technologies*, 2021(1):107–127, 2021.

[13] Sufian Hameed and Hassan Ahmed Khan. Leveraging SDN for collaborative DDoS mitigation. In *2017 International Conference on Networked Systems (NetSys)*, pages 1–6, 2017.

[14] Ryan Henry and Ian Goldberg. Batch proofs of partial knowledge. In Michael Jacobson, Michael Locasto, Payman Mohassel, and Reihaneh Safavi-Naini, editors, *Applied Cryptography and Network Security*, pages 502–517, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[15] Vincent Herbert, Bhaskar Biswas, and Caroline Fontaine. Design and implementation of low-depth pairing-based homomorphic encryption scheme. *Journal of Cryptographic Engineering*, 9(2):185–201, June 2019.

[16] Aapo Kalliola, Kiryong Lee, Heejo Lee, and Tuomas Aura. Flooding DDoS mitigation and traffic management with software defined networking. In *2015 IEEE 4th International Conference on Cloud Networking (CloudNet)*, pages 248–254, 2015.

[17] Karen Hao. The Facebook whistleblower says its algorithms are dangerous. Here's why. https://www.technologyreview.com/2021/10/05/1036519/facebook-whistleblower-frances-haugen-algorithms/, October 2021.

[18] Soon Hin Khor and Akihiro Nakao. DaaS: DDoS mitigation-as-a-service. In *2011 IEEE/IPSJ International Symposium on Applications and the Internet*, pages 160–171, 2011.

[19] Vishnu Teja Kilari, Ruozhou Yu, Satyajayant Misra, and Guoliang Xue. EARS: Enabling private feedback updates in anonymous reputation systems. In *2020 IEEE Conference on Communications and Network Security (CNS)*, pages 1–9, 2020.

[20] Patrick Lindenfors, Andreas Wartel, and Johan Lind. 'Dunbar's number' deconstructed. *Biology Letters*, 17(5):1–4, April 2021.

[21] Michael Naehrig, Ruben Niederhagen, and Peter Schwabe. New software speed records for cryptographic pairings. In *Proceedings of the First International Conference on Progress in Cryptology: Cryptology and Information Security in Latin America*, LATINCRYPT'10, pages 109–123, Berlin, Heidelberg, 2010. Springer-Verlag.

[22] Arvind Narayanan, Hristo S. Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul Richard Shin, and Dawn Xiaodong Song. On the feasibility of internet-scale author identification. *2012 IEEE Symposium on Security and Privacy*, pages 300–314, 2012.

[23] Elan Pavlov, Jeffrey S. Rosenschein, and Zvi Topol. Supporting privacy in decentralized additive reputation systems. In Christian Jensen, Stefan Poslad, and Theo Dimitrakos, editors, *Trust Management*, pages 108–119, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[24] Margaret Pless. Kiwi Farms, the web's biggest community of stalkers. https://nymag.com/intelligencer/2016/07/kiwi-farms-the-webs-biggest-community-of-stalkers.html, July 2016.

[25] John M Pollard. Monte Carlo methods for index computation (mod p). *Mathematics of Computation*, 32(143):918–924, 1978.

[26] Rishikesh Sahay, Gregory Blanc, Zonghua Zhang, and Hervé Debar. Towards autonomic DDoS mitigation using software defined networking. In *SENT 2015: NDSS Workshop on Security of Emerging Networking Technologies*, San Diego, Ca, United States, February 2015. Internet Society.

[27] Rishikesh Sahay, Gregory Blanc, Zonghua Zhang, and Hervé Debar. ArOMA: An SDN based autonomic DDoS mitigation framework. *Computers & Security*, 70:482–499, 2017.

[28] Kyle Soska, Albert Kwon, Nicolas Christin, and Srinivas Devadas. Beaver: A decentralized anonymous marketplace with secure reputation. Cryptology ePrint Archive, Report 2016/464, 2016. https://eprint.iacr.org/2016/464.

[29] Adam Steinbaugh. Kevin Bollaert sentenced to 18 years over revenge porn site "You Got Posted". http://adamsteinbaugh.com/2015/04/03/kevin-bollaert-sentenced-to-years-over-revenge-porn-site-you-got-posted/, April 2015.

[30] Jonathan Wells. Tyler Oakley: How the internet revolutionised LGBT life. https://www.telegraph.co.uk/men/thinking-man/tyler-oakley-how-the-internet-revolutionised-lgbt-life/, November 2015.

[31] Danfeng Yao, Roberto Tamassia, and Seth Proctor. Private distributed scalar product protocol with application to privacy-preserving computation of trust. In Sandro Etalle and Stephen Marsh, editors, *Trust Management*, pages 1–16, Boston, MA, 2007. Springer US.

[32] Ennan Zhai, David Isaac Wolinsky, Ruichuan Chen, Ewa Syta, Chao Teng, and Bryan Ford. AnonRep: Towards tracking-resistant anonymous reputation. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 583–596. USENIX Association, March 2016.

[33] Mingwu Zhang, Yong Xia, Ou Yuan, and Kirill Morozov. Privacy-friendly weighted-reputation aggregation protocols against malicious adversaries in cloud services. *International Journal of Communication Systems*, 29(12):1863–1872, 2016.

[34] Luying Zhou, Huaqun Guo, and Gelei Deng. A fog computing based approach to DDoS mitigation in IIoT systems. *Computers & Security*, 85:51–62, 2019.

A FREEMAN'S PRIME-ORDER BGN

The prime-order BGN construction we use in PRSONA was first suggested by Freeman [11]. First, we quote Freeman's definition of a bilinear group generator (Freeman's Definition 2.1 [11, p. 48]):

- A *bilinear group generator* is an algorithm $\mathcal{G}$ that takes as input a security parameter λ and outputs a description of five abelian groups G, G_1, H, H_1, and G_T, with $G_1 \subset G$ and $H_1 \subset H$. We assume that this description permits efficient (*i.e.*, polynomial time in λ) group operations and random sampling in each group. The algorithm also outputs an efficiently computable map (or "pairing") $e : G \times H \to G_T$ that is
  - Bilinear: $e(g_1 g_2, h_1 h_2) = e(g_1, h_1) e(g_1, h_2)$ $e(g_2, h_1) e(g_2, h_2)$ for all $g_1, g_2 \in G$ and $h_1, h_2 \in H$; and
  - Nondegenerate: for any $g \in G$, if $e(g, h) = 1$ for all $h \in H$, then $g = 1$ (and similarly with G, H reversed).

Note that, in the above definition, G and H are not assumed to be prime order (and in fact, they could not be, as described). This will be addressed shortly.

Next, we quote Freeman's definition of a projecting bilinear group generator (Freeman's Definition 3.1 [11, p. 52]):

- Let $\mathcal{G}$ be a bilinear group generator (Def. 2.1). We say $\mathcal{G}$ is *projecting* if it also outputs a group $G'_T \subset G_T$ and three group homomorphisms π_1, π_2, π_T mapping G, H, G_T to themselves, respectively, such that
  1. G_1, H_1, G'_T are contained in the kernels of π_1, π_2, π_T, respectively, and
  2. $e(\pi_1(g), \pi_2(h)) = \pi_T(e(g, h))$ for all $g \in G$, $h \in H$.

The original BGN cryptosystem [5] formed the hidden subgroups $G_1 \subset G$ and $H_1 \subset H$ by having $G = H$ be an elliptic curve group with a symmetric pairing, and whose order is an RSA number $N = p_1 p_2$, and $G_1 = H_1$ are the order-p_1 subgroups. (The factorization of N is part of the private key of the scheme.) The projections π_1, π_2, and π_T are then exponentiations by p_1. Unfortunately, this requires the order of the group to be an RSA-sized integer (meaning thousands of bits for 128-bit security), which makes ciphertexts large and slow to process. Freeman's adaptation instead uses a standard prime-order asymmetric pairing setup, as can be seen in Freeman's Example 3.3 [11, p. 53]:

- Let $\mathcal{P}$ be a prime-order bilinear group generator. Define $\mathcal{G}_{\mathcal{P}}$ to be a bilinear group generator that on input λ does the following:
  1. Let $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, \hat{e}) \xleftarrow{\$} \mathcal{P}(1^\lambda)$, and let $G = \mathbb{G}_1^2$, $H = \mathbb{G}_2^2$, $G_T = \mathbb{G}_T^4$.
  2. Choose generators $g \xleftarrow{\$} \mathbb{G}_1$, $h \xleftarrow{\$} \mathbb{G}_2$, and let $\gamma = \hat{e}(g, h)$.
  3. Choose random $a_1, b_1, c_1, d_1, a_2, b_2, c_2, d_2 \xleftarrow{\$} \mathbb{F}_p$ such that $a_1 d_1 - b_1 c_1 = a_2 d_2 - b_2 c_2 = 1$.
  4. Let G_1 be the subgroup of G generated by $g' = (g^{a_1}, g^{b_1})$, let H_1 be the subgroup of H generated by $h' = (h^{a_2}, h^{b_2})$, and let G'_T be the subgroup of G_T generated by
  $$\{\gamma^{(a_1a_2, a_1b_2, b_1a_2, b_1b_2)}, \gamma^{(a_1c_2, a_1d_2, b_1c_2, b_1d_2)}, \gamma^{(c_1a_2, d_1b_2, c_1a_2, d_1b_2)}\}.$$
  5. Define $e : G \times H \to G_T$ by
  $$e((g_1, g_2), (h_1, h_2)) := (\hat{e}(g_1, h_1), \hat{e}(g_1, h_2), \hat{e}(g_2, h_1), \hat{e}(g_2, h_2)).$$
  6. Let $A = \begin{pmatrix} -b_1c_1 & -b_1d_1 \\ a_1c_1 & a_1d_1 \end{pmatrix}$, $B = \begin{pmatrix} -b_2c_2 & -b_2d_2 \\ a_2c_2 & a_2d_2 \end{pmatrix}$, and define
  $$\pi_1((g_1, g_2)) := (g_1, g_2)^A = (g_1^{-b_1c_1} g_2^{a_1c_1}, g_1^{-b_1d_1} g_2^{a_1d_1})$$
  $$\pi_2((h_1, h_2)) := (h_1, h_2)^B = (h_1^{-b_2c_2} h_2^{a_2c_2}, h_1^{-b_2d_2} h_2^{a_2d_2})$$
  $$\pi_T((\gamma_1, \gamma_2, \gamma_3, \gamma_4)) := (\gamma_1, \gamma_2, \gamma_3, \gamma_4)^{A \otimes B}$$
  7. Output the tuple $(G, G_1, H, H_1, G_T, G'_T, e, \pi_1, \pi_2, \pi_T)$.

Due to the prime-order assymetric pairing setup described above, the ciphertexts in use are smaller and faster to operate on, while still admitting depth-1 multiplication, as can be seen in Freeman's full construction (from Freeman's Section 5 [11, p. 57]):

- **KeyGen(1^λ):** Let $\mathcal{G}$ be a projecting bilinear group generator (Definition 3.1). Compute $(G, G_1, H, H_1, G_T, G'_T, e, \pi_1, \pi_2, \pi_T) \leftarrow \mathcal{G}(1^\lambda)$. Choose $g \xleftarrow{\$} G, h \xleftarrow{\$} H$, and output the public key $PK = (G, G_1, H, H_1, G_T, e, g, h)$ and the secret key $SK = (\pi_1, \pi_2, \pi_T)$.
- **Encrypt(PK, m):** Choose $g_1 \xleftarrow{\$} G_1$ and $h_1 \xleftarrow{\$} H_1$. (Recall that the output of $\mathcal{G}$ allows random sampling from G_1 and H_1.) Output the ciphertext $(C_A, C_B) = (g^m \cdot g_1, h^m \cdot h_1) \in G \times H$.
- **Multiply(PK, C_A, C_B):** This algorithm takes as inputs two ciphertexts $C_A \in G$ and $C_B \in H$. Choose $g_1 \xleftarrow{\$} G_1$ and $h_1 \xleftarrow{\$} H_1$, and output $C = e(C_A, C_B) \cdot e(g, h_1) \cdot e(g_1, h) \in G_T$.
- **Add(PK, C, C'):** This algorithm takes as input two ciphertexts C, C' in one of G, H, or G_T. Choose $g_1 \xleftarrow{\$} G_1$ and $h_1 \xleftarrow{\$} H_1$, and do the following:
  1. If $C, C' \in G$, output $C \cdot C' \cdot g_1 \in G$.
  2. If $C, C' \in H$, output $C \cdot C' \cdot g_2 \in H$.
  3. If $C, C' \in G_T$, output $C \cdot C' \cdot e(g, h_1) \cdot e(g_1, h) \in G_T$.
- **Decrypt(SK, C):** The input ciphertext is an element of G, H, or G_T.
  1. If $C \in G$, output $m \leftarrow \log_{\pi_1(g)}(\pi_1(C))$.
  2. If $C \in H$, output $m \leftarrow \log_{\pi_2(h)}(\pi_2(C))$.
  3. If $C \in G_T$, output $m \leftarrow \log_{\pi_T(e(g,h))}(\pi_T(C))$.