David Froelicher*, Patricia Egger, João Sá Sousa, Jean Louis Raisaro, Zhicong Huang, Christian Mouchet, Bryan Ford, and Jean-Pierre Hubaux

UnLynx: A Decentralized System for Privacy-Conscious Data Sharing

Abstract: Current solutions for privacy-preserving data sharing among multiple parties either depend on a centralized authority that must be trusted and provides only weakest-link security (e.g., the entity that manages private/secret cryptographic keys), or leverage on decentralized but impractical approaches (e.g., secure multi-party computation). When the data to be shared are of a sensitive nature and the number of data providers is high, these solutions are not appropriate. Therefore, we present UnLynx, a new decentralized system for efficient privacy-preserving data sharing. We consider m servers that constitute a collective authority whose goal is to verifiably compute on data sent from n data providers. UnLynx guarantees the confidentiality, unlinkability between data providers and their data, privacy of the end result and the correctness of computations by the servers. Furthermore, to support differentially private queries, UnLynx can collectively add noise under encryption. All of this is achieved through a combination of a set of new distributed and secure protocols that are based on homomorphic cryptography, verifiable shuffling and zero-knowledge proofs. UnLynx is highly parallelizable and modular by design as it enables multiple security/privacy vs. runtime tradeoffs. Our evaluation shows that UnLynx can execute a secure survey on 400,000 personal data records containing 5 encrypted attributes, distributed over 20 independent databases, for a total of 2,000,000 ciphertexts, in 24 minutes.

Keywords: privacy, confidentiality, decentralized system, data sharing, differential privacy

DOI 10.1515/popets-2017-0032
Received 2017-02-28; revised 2017-06-01; accepted 2017-06-02.

1 Introduction

In our increasingly connected and data-driven world, the need to protect sensitive data and still be able to share them among multiple entities in a privacy-conscious way has become critical in numerous contexts. Five concrete examples include (i) *medical research*, where patients' sensitive data, from multiple institutions, need to be protected from an increasing number of cyber attacks [45] while remaining accessible to practitioners who want to better understand and treat complex diseases, (ii) *fraud detection*, where a tax authority needs to securely access foreign bank accounts in order to identify potential tax evaders, (iii) *public safety*, where security agencies from different countries need to protect their confidential information but also share it to design effective anti-terrorism strategies, (iv) *private surveys*, where institutions need to collect personal data from citizens or private companies, and (v) *commercial collaborations*, where corporations do not want to reveal confidential data but are willing to share some information for mutual benefit.

In the last few years, researchers have tried to address these needs by proposing different privacy-preserving solutions that enable several data providers to securely store and share their sensitive data in either a centralized or decentralized way [3, 11, 14, 26, 28, 30, 33, 39, 40, 42, 49, 50]. Yet, despite the acknowledgment and acceptance that most of these solutions have received in the research community of privacy and security, *only a few* have been converted into concrete *operational tools* and deployed in the real world [3, 14, 39]. The main reasons for such a low adoption stem from the fact that simpler solutions based on a centralized approach [30, 39] only provide weakest-link security (e.g., relying on a trusted third party managing cryptographic keys); and, decentralized (and more complex) solutions based on secret sharing (SS) [3, 14, 26, 49] or secure multi-party computation (SMPC) [11, 28, 33, 40, 42, 50] have intrinsic limitations in terms of control over data and scalability. For example, SS-based solutions result in data providers losing the full control over their data as, for security reasons, they require the data storage to be outsourced to independent servers. SMPC-based solutions, which theoretically enable multiple data providers to keep control over their private data and to jointly and securely compute any public function over them, become completely unpractical when the number of data providers increases to more than three.

As a result of the immaturity of these secure solutions, in most of the scenarios mentioned above, it is common practice to rely only on legal agreements rather than technical solutions. Stakeholders willing to share their sensitive

***Corresponding Author: David Froelicher:** EPFL, E-mail: david.froelicher@epfl.ch
Patricia Egger: EPFL, E-mail: patricia.egger@alumni.epfl.ch
João Sá Sousa: EPFL, E-mail: joao.gomesdesaesousa@epfl.ch
Jean Louis Raisaro: EPFL, E-mail: jean-raisaro@epfl.ch
Zhicong Huang: EPFL, E-mail: zhicong.huang@epfl.ch
Christian Mouchet: EPFL, E-mail: christian.mouchet@epfl.ch
Bryan Ford: EPFL, E-mail: bryan.ford@epfl.ch
Jean-Pierre Hubaux: EPFL, E-mail: jean-pierre.hubaux@epfl.ch

data can stipulate data-use agreements with a centralized trusted third party that becomes fully responsible for collecting, storing and managing the data for the whole ecosystem. Yet, this approach has proven to not be future-proof and to be particularly difficult to realize on a large scale for the following three main reasons:

- The trusted third party represents a single point of failure in the system. A breach caused by an external (hacker) or internal (insider) attack can compromise all data providers' data at once.
- Data providers are beginning to understand the value of their sensitive data and are increasingly willing to be masters of them, instead of giving control to a third party [8, 10].
- Privacy restrictions from different jurisdictions, such as the new European General Data-Protection Regulation [4], can prevent data providers from transferring their data across national boundaries, thus substantially limiting the scope of data-sharing.

Hence, it is more urgent than ever to develop new operational tools that enable *thousands* of data providers to *protect* and *efficiently share* their sensitive data while keeping control over them.

In this paper, we respond to this by presenting UnLynx, a new decentralized *operational* solution for efficiently protecting and querying a large amount of sensitive data that is distributed across a multitude of data providers. UnLynx outperforms state-of-the-art secure solutions, both in terms of security and efficiency, as it does not present a single point of failure and has the ability to scale up to thousands of data providers. It achieves this while guaranteeing (i) data confidentiality at rest and during processing, (ii) unlinkability between data providers and their data, (iii) correctness of secure computations, and (iv) private release of end-results.

UnLynx is a decentralized system where data providers are able to share their sensitive data without having to trust one single entity to protect their privacy and data confidentiality. In fact, trust is distributed among multiple entities that constitute a collective authority [44]. UnLynx achieves two distinct types of decentralization. The first is the decentralization of the data, i.e., there is no central repository for all data. Each data provider can store its data on its own premises thus maintaining control over them. The second is the decentralization of the computations, i.e., there is no central authority responsible for all the computations. Instead, even in the presence of a malicious adversary, a group of collective-authority servers is responsible for securely processing data from the different data providers.

In particular, UnLynx enables the end-user to perform SQL-like queries over encrypted distributed data to compute useful descriptive statistics (e.g., count/sum, averages, etc.) on a selected subset of data records in a privacy-preserving way. This subset selection is based on a set of Boolean conditions. Although these operations represent only a subset of those supported by alternative approaches based on SMPC or trusted third parties, it is also true that such a subset is enough to solve most of the data-sharing scenarios described above. Moreover, contrary to alternative approaches, our solution is highly parallelizable by design and can easily scale to thousands of data providers with millions of data records.

In UnLynx, during the secure processing, data are homomorphically encrypted under a collective key and shuffled by a set of servers, hence preventing any entity in the ecosystem from linking data back to their respective owners. By generating deterministically encrypted tags from probabilistically encrypted records, UnLynx is also able to filter encrypted records according to the set of Boolean conditions defined in the query. Finally, to prevent inferences based on the end result and to satisfy formal privacy notions (e.g., differential privacy), UnLynx provides a mechanism enabling the collective-authority servers to obliviously perturb the end result with noise (unknown to any party) sampled from a known probability distribution. All computations performed by UnLynx can be verified through the use of cryptographic zero-knowledge proofs.

To evaluate the performance of UnLynx, we built a working prototype implemented as a modular system where the different security features are represented by independent modules that can be activated depending on the application domain and on the privacy/efficiency requirements. An experimental evaluation, in a realistic simulation environment, shows that our prototype scales almost linearly with respect to the amount of data to be shared and the size of the collective authority. A query - with 2 Boolean conditions and 1 grouping criteria, over 400,000 records distributed among 20 data providers, and processed by 3 independent servers - can be executed in less than 24 minutes under the assumption of a strong adversary. By relaxing this assumption (e.g., by considering honest-but-curious servers and deactivating some of the security modules) the execution time of the same query can be reduced to, at best, 2.5 minutes.

Contributions

In this paper we make the following contributions:

- A flexible, decentralized, strongest-link security system for privacy-conscious data sharing among a multitude of distributed data providers, built on top of well-established security and privacy techniques, and secure even in a strong adversary model.
- A novel use of collective authorities combined with the use of homomorphic encryption and zero-knowledge proofs.
- A set of new secure and distributed protocols enabling deterministic tagging, key switching and collective ag-

gregating that, combined with a verifiable shuffle enable a set of collective-authority servers to compute on distributed sensitive data and produce zero-knowledge proofs of their work.
– A novel distributed protocol that enables collective-authority servers to obliviously perturb an aggregate query end-result in order to satisfy formal notions of privacy (e.g., differential privacy) and to mitigate inference attacks from a malicious querier.
– A thorough evaluation of our modular system and of the different privacy/efficiency tradeoffs in a realistic simulation environment.

2 Background

We describe several tools that UnLynx is built upon. For the remainder of this paper, we assume elliptic curve notation in which $\mathcal{G}$ is an elliptic curve[1] and B designates a base point on $\mathcal{G}$.

2.1 Collective Authority

Applications and systems often rely on authorities that provide security-critical services. For example, certificate services might rely on an authority to attest the ownership of public keys by the subjects of the certificates. Because these services are security-critical, they are obvious targets for attackers. In order to provide stronger security and to distribute trust, authorities can be decentralized, i.e., composed of multiple collaborating entities, referred to as a collective authority. An example of a scalable collective-authority is described by Syta et al. [44].

Each server S_i from the collective authority possesses a private-public key pair (k_i, K_i). Using elliptic curve notation, $K_i = k_iB$, where k_i is a scalar and K_i is a point on $\mathcal{G}$. The collective authority generates a collective public key $K = K_1 + ... + K_m$ as the aggregation of all the servers' public keys. The corresponding private key is never constructed. Instead, in order to decrypt a message encrypted using K, each server in the collective authority must participate and partially decrypt it by using its own private key. Thus, the collective-authority key provides strongest-link security such that attackers have to compromise all m servers in order to decrypt. This can be assimilated to a (m, m)-threshold decryption.

2.2 ElGamal Cryptosystem

In UnLynx, data are encrypted using the probabilistic and additively-homomorphic ElGamal cryptosystem. Specifically, if P is a public key and x is a message mapped to a point on $\mathcal{G}$, the ElGamal encryption of x is the tuple $\mathrm{E}_P(x) = (rB, x+rP)$ where r is a random nonce. The additive homomorphic property states that $\mathrm{E}_P(\alpha x_1 + \beta x_2) = \alpha \mathrm{E}_P(x_1) + \beta \mathrm{E}_P(x_2)$ for any messages x_1 and x_2 and for any scalars α and β. In order to decrypt a ciphertext $(rB, x+rP)$, the holder of the corresponding private key p $(P = pB)$ multiplies rB and p yielding $p(rB) = rP$ and subtracts this point from $x+rP$. The result is the message x.

2.3 Zero-Knowledge Proofs of Correctness

In order to provide guarantees of correct computations by the collective authority, UnLynx makes use of zero-knowledge proofs for general statements about discrete logarithms, introduced by Camenisch et al. [16]. In general, these proofs enable a verifier to check that the prover knows the discrete logarithms y_1 and y_2 of the public values $Y_1 = y_1B$ and $Y_2 = y_2B$ and that they satisfy a linear equation

$$A_1y_1 + A_2y_2 = A, \tag{1}$$

where A, A_1, A_2 are public points on $\mathcal{G}$. This is done without revealing anything about y_1 or y_2. These proofs can be made non-interactive through the Fiat-Shamir heuristic [27].

In UnLynx, these proofs are used to ensure the integrity of each computation and are published in such a way that they can be verified by any entity.

2.4 Verifiable Shuffle

In UnLynx, we use the verifiable shuffle of sequences of ElGamal pairs described by Andrew Neff [34] to transform encrypted data in such a way that the outcome cannot be linked to the original encryption.

Formally, we consider the *SHUFFLE* protocol such that $SHUFFLE_{\pi, r''_{i,j}}$ takes as input multiple sequences of ElGamal pairs $(C_{1,i,j}, C_{2,i,j})$ forming a $a \times b$ matrix and outputs a shuffled matrix of $(\bar{C}_{1,i,j}, \bar{C}_{2,i,j})$ pairs such that for all $1 \le i \le a$ and $1 \le j \le b$,

$$(\bar{C}_{1,i,j}, \bar{C}_{2,i,j}) = (C_{1,\pi(i),j} + r''_{\pi(i),j}B, C_{2,\pi(i),j} + r''_{\pi(i),j}P)$$

where $r''_{i,j}$ is a re-randomization factor, π is a permutation and P is a public key.

Therefore, this verifiable shuffle permutes each row of ciphertexts of the form $(r_{i,j}B, x_{i,j} + r_{i,j}P)$ and transforms it into an indistinguishable list of ciphertexts of the form $\left(r'_{i,j}B, x_{i,j} + r'_{i,j}P\right)$ where $r'_{i,j} = r''_{i,j} + r_{i,j}$ is a new random nonce.

Andrew Neff provides a method to prove that such a shuffle is done correctly, i.e., that there exists a permutation π and re-randomization factors $r''_{i,j}$ such that output $= SHUFFLE_{\pi, r''_{i,j}}$(input), without revealing anything about π or $r''_{i,j}$. This is achieved by using honest-verifier zero-knowledge proofs that are discussed in detail by Neff [34, 35].

1 Elliptic Curves are known to require smaller key sizes for the same levels of security compared to other methods [1], hence enabling efficient computations.

2.5 Differential Privacy

Differential privacy is an approach for privacy-preserving reporting of results, first introduced by Cynthia Dwork [23]. This approach guarantees that a given randomized statistic, $\mathcal{M}(D)=R$, computed on a dataset D behaves similarly when computed on a neighbor dataset D' that differs from D in exactly one element. More formally we have that

$$\Pr[\mathcal{M}(D)=R] \leq \exp(\epsilon)\cdot\Pr\left[\mathcal{M}(D')=R\right]+\delta, \qquad (2)$$

where ϵ and δ are privacy parameters: the closer to 0 they are, the higher the privacy level is. The most straightforward method for achieving (ϵ, δ)-differential privacy [25] consists in perturbing the original output $f(D)$ with noise drawn from the Laplace distribution with mean 0 and scale $\frac{\Delta f}{\epsilon}$, where Δf is known as the *sensitivity* of the original real valued function f to be executed on the dataset:

$$\Delta f = \max_{D,D'} ||f(D)-f(D')||_1. \qquad (3)$$

3 System and Threat Models

We introduce our system model and discuss UnLynx's functionality, security/privacy and performance requirements. Then we sketch our threat model.

3.1 System Model

Our system, as depicted in Figure 1, consists of n data providers DP_1,..., DP_n, m servers S_1,..., S_m and a querier Q[2]. Together, the servers form a collective authority (CA) for privacy-preserving data sharing. The DPs combined constitute a distributed database that is used to answer queries. The querier and each of the DPs independently choose one server in the CA to communicate with. They can change this choice at any time. Data providers generate and/or store data that can pertain to either one or several individuals.

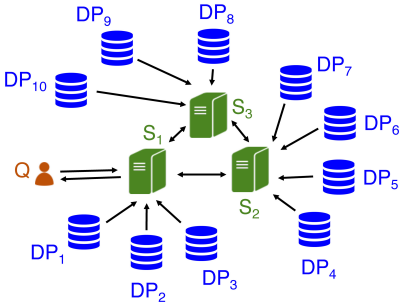


Fig. 1. Data providers (in blue), collective-authority servers (in green) and querier (in red). In this example, $m=3$ and $n=10$. The arrows show the information communication flow.

We assume a public immutable distributed ledger DL that is collectively managed by the servers and contains a complete view of the system, the access rights of the queriers, a list of available DPs, the query history and the system's global variables, such as the privacy parameters. Any change in the topology or querier access rights triggers an update of the public ledger.

We now discuss the functionality that UnLynx must provide along with its security/privacy and performance requirements.

Functionality. UnLynx should permit SQL queries of the form '`SELECT SUM(*)/COUNT(*) FROM` DP_1,...,DP_l `WHERE` $*$ `AND/OR` $*$ `GROUP BY` $*$' , where $l \leq n$ and we consider that '$*$' denotes an arbitrary number of attributes. We refer to the attributes involved in the `WHERE` clause and `GROUP BY` statement as filtering attributes. These queries can be executed on the distributed databases held by a set of l chosen DPs. Depending on its permission level, a querier can be limited to some types of queries. Finally, in some specific cases, UnLynx can also provide the possibility of '`SELECT` $*$' queries. This would, for example, enable DPs to query and decrypt their own database. However, this type of query is not suited for secure data-sharing as, it cannot be done on a distributed database held by multiple DPs, while ensuring data confidentiality and privacy. We further discuss this in Appendix B.

Security and privacy. UnLynx should be able to filter query responses based on the Boolean conditions of the query, i.e. attributes in the `WHERE` clause, and to group responses according to the `GROUP BY` statement without revealing any information about any of the attributes or to which groups the responses belong to. The ***confidentiality*** of raw data must be protected at rest and during processing. Moreover, no entity should be able to trace a query response back to its provider, i.e. ***unlinkability*** between DPs and their data must be guaranteed. UnLynx's primary goals are to enable data sharing, ensure the DPs' privacy and avoid any data leakage. Hence, UnLynx does not intend to protect queriers' privacy. UnLynx should permit any entity to check the ***correctness*** of the system's computations and it should ensure that any entity that computes incorrectly can be identified and excluded from future computations. UnLynx must ensure ***(ϵ, δ)-differential privacy*** for any individual sharing his data. Finally, it should guarantee that only the querier is able to decrypt the result of its query.

Performance. We require UnLynx to scale linearly with the number of DPs, the amount of data and the size of the CA. It should also provide a shorter response time by relaxing some of the security/privacy requirements.

3.2 Threat Model

We define UnLynx's threat model by discussing the role of each entity in the system.

Collective authority servers. We assume an Anytrust Model [46] for the CA servers. In other words, to

2 There can be several queriers in the system, but as they do not interact with each other, without loss of generality, we consider the protocol for a single one.

achieve the functionality and the security/privacy guarantees described in Section 3.1, UnLynx does not require any particular server to be globally trusted or to be honest-but-curious. Instead, as long as there *exists* at least one server that is not malicious, these properties are guaranteed.

Data providers. We assume DPs to be honest-but-curious and discuss the impact of having malicious DPs in Section 5. UnLynx does not protect against false information coming from the DPs, i.e., we do not ensure data integrity coming from the DPs. However, in Section 6.1, we propose a mechanism that enables servers to verify that an input is within a range of values, hence mitigating the effect of DPs sending altered data. Each DP can independently choose one server in the CA to trust, i.e., each DP can choose a different server. Finally, a DP cannot collude with any other entity.

Queriers. We assume queriers to be malicious. They can try to infer sensitive information about DPs and can collude between themselves or with a subset of the CA servers.

We assume that all network communication is authenticated and encrypted. This can be ensured by standard cryptographic techniques, e.g., using TLS protocol. The number of queries accepted per minute by UnLynx is limited and if a server does not respond, it can be removed from the CA.

4 Overview of UnLynx

In order to achieve privacy-preserving sharing of sensitive data, we developed a modular system that enables the use of sub-protocols as building blocks according to security, privacy and performance requirements. In Section 4.1, we describe our decentralized data-sharing protocol that is depicted in Figure 2. This protocol combines sub-protocols that we describe in more detail in Section 4.2.

4.1 Decentralized Data-Sharing Protocol

The protocol starts with a querier who wants to retrieve some aggregate information about sensitive data stored by multiple DPs. The query is sent to a chosen server that broadcasts it to the other servers in the CA. Then, the servers broadcast the request to all DPs that respond with the requested sensitive data in encrypted form. These data are securely and privately processed by the servers, before the query result is sent to the querier, who is then able to decrypt the final result. Table 1 contains the notation and symbols used throughout the proposed protocol.

Step 0. In the initialization phase, each server S_i in the CA creates its own ElGamal key pair $(k_i,\ K_i)$. The CA constructs its public key $K = \sum_{i=1}^{m} K_i$ that is used by data providers to encrypt their sensitive data. A server of the CA generates the probability distribution that corresponds to the predefined differential privacy parameters of

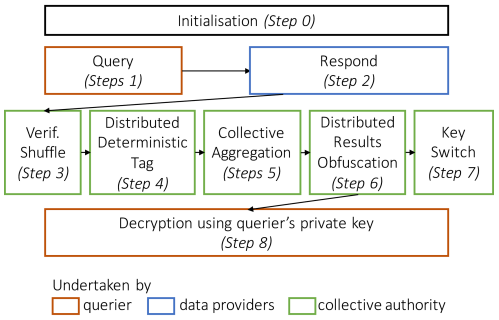


Fig. 2. Decentralized Data Sharing Protocol from beginning (DPs encrypt their data using the CA's public key) to end (querier decrypts final aggregate answers using its private key). The initialization step is executed once at the setup of the system.

the system, ϵ and δ, and it uniformly samples some points from this distribution, based on the probability quantum parameter θ. We assume that these parameters are chosen before the initialization of our system. This choice highly depends on the application domain. The resulting samples are then used as obfuscation noise in Step 6. This process is explained in detail in Section 4.2.3.

Symbol/abbreviation	Description
$\mathcal{G}$; B	Elliptic curve; base point on $\mathcal{G}$
$S_1,...,S_m$	m collective-authority servers
$DP_1,...,DP_n$	n data providers
$\mathsf{E}_P()$	ElGamal encryption using key P
$(k_i,\ K_i)$	Server i's private-public key
$K = K_1 + ... + K_m$	Collective authority's public key
$(u,\ U)$	Querier's private-public key
$(C_1,\ C_2) = (rB,\ x + rK)$	ElG encryption of x with K
s_i, v_i	Server i's short-term secrets
DL	Immu. distributed ledger
$(\epsilon,\ \delta)$; Δf	Privacy parameters; Query sensitivity
θ	Proba. quantum parameter

Table 1. Commonly used symbols and abbreviations.

Step 1. Querier Q sends its query to a server S_i in the CA. An example of a query could be '`SELECT SUM(`*employed*`) FROM` $DP_1, \ldots, DP_d$ `WHERE` *age* $= E_K(46)$ `AND` *married* $= E_K(1)$ `GROUP BY` *gender*'. In order for the servers to privately process the query, the attribute values in the `WHERE` clause are encrypted with the CA's public key K.

Finally, S_i broadcasts the query to all the other collective-authority servers, and each of them sends the query to a different set of DPs such that each DP in the `FROM` clause receives the query. We assume there is no error during the broadcast, hence there is no query duplicate. Before forwarding the query to its DPs or to other servers, each server verifies the access rights of the querier in the distributed ledger DL.

Step 2. The DPs select, for each of their data records, the encrypted or clear text values of the attributes specified

in the query and send them back to their respective server. If the query contains a 'SELECT COUNT' statement, the DPs append an attribute with value $E_K(1)$ to each of their responses. In order to prevent the servers from knowing the count value, the DPs add dummy records (containing $E_K(0)$) to their responses. A DP can choose to not respond to a query, e.g., if it is too intrusive. Each DP digitally signs its set of responses, publicly logs it in the DL, and appends the signature to its message to ensure authenticity.
If some DPs respond with filtering attributes (attributes involved in the WHERE clause and GROUP BY statement) in clear text, the servers locally aggregate the responses for each different combination of filtering attributes, thus reducing the number of responses to be processed.

Step 3. The CA launches a verifiable shuffle sub-protocol in order to break the link between the DPs and their data. In particular, each server sequentially performs a verifiable shuffle on the data, as described in Section 2.4. Any clear text data are encrypted during the verifiable shuffle process. Eventually, each DP's data will be shuffled among themselves and with other DPs' data, once by each server.

Step 4. In order to execute the query, all the servers run a distributed deterministic-tag sub-protocol on the filtering attributes. This protocol appends a deterministic tag to each filtering attribute of each DP's response.

The outputs of this protocol are used to filter the responses based on the query WHERE clause and to group the responses according to the GROUP BY statement. For example, if the query is 'SELECT SUM(*employed*) FROM $DP_1, \ldots, DP_d$ WHERE $age{=}E_K(46)$ AND $married = E_K(1)$ GROUP BY *gender*', the WHERE clause is transformed to 'WHERE $age{=}DT(E_K(46))$ AND $married{=}DT(E_K(1))$' where $DT(x)$ is the deterministic tag derived from x. The derivation of this tag is explained in Section 4.2.1. The server will then check the query predicate for each response by verifying if '$DT(age_{resp_{ij}}) == DT(E_K(46))$ AND $DT(married_{resp_{ij}}) == DT(E_K(1))$' holds. Here, $married_{resp_{ij}}$ and $age_{resp_{ij}}$ are the values of the married and age attributes of DP_i's response j. If the predicate is not verified, the response is discarded. Finally, the remaining responses are grouped based on the deterministic-tag values derived from their GROUP BY statement attribute values, and the SUM/COUNT attribute values are aggregated in each group. This results in one aggregated response per group.

Step 5. The servers perform a collective aggregation protocol, presented in Section 4.2.2. When this is done, one server has the total aggregate response for each group, encrypted under the CA's public key K.

Step 6. One server executes an oblivious results obfuscation sub-protocol and begins by verifying in the public distributed ledger DL if the same query has already been performed. We discuss how this verification can be executed with encrypted queries in Section 4.2.3. If the same query has already been executed, the server adds the same noise value that was used for the first query to the results of the new one. If this is not the case, the server is responsible for starting a verifiable shuffle protocol (similar to Step 3) on the list of noise values generated in the initialization phase. The server chooses the first element in the shuffled list and adds this noise to the query results.

Step 7. The CA launches a key switch sub-protocol to obtain the query results encrypted under the querier's public key U, instead of under the CA's public key K.

Step 8. The total aggregated responses per group are sent to the querier who is able to decrypt them by using its private key u.

4.2 Sub-protocols

Here, we provide details about the distributed deterministic-tag, collective aggregation, distributed results obfuscation, key switch and dynamic collective-authority sub-protocols that we designed to be independent building blocks. They can be combined to achieve privacy-conscious data sharing with different levels of security, privacy and performance. For the distributed deterministic-tag, distributed results obfuscation and key switch, the collective-authority servers are organized into a cycle as these sub-protocols are performed sequentially. For the collective aggregation, however, the CA can be organized into a tree to increase the protocol's efficiency. After each sub-protocol, we show how zero-knowledge proofs can be used to guarantee computation integrity.

4.2.1 Distributed Deterministic-Tag

The distributed deterministic-tag sub-protocol, or DDT, consists in tagging an ElGamal ciphertext of a message x, encrypted using the CA key K, with a deterministic value related to x without ever decrypting the ciphertext. This sub-protocol is executed in two successive rounds. We start with $E_K(x) = (C_1, C_2){=}(rB, x{+}rK)$, the ciphertext tuple corresponding to an ElGamal encryption of message x that uses the CA's public key K.

In the first round, each server sequentially generates a fresh secret s_i and adds the value derived from its secret s_iB to C_2. This eliminates the possibility of having a deterministic tag of 0 as an output of the protocol when the input message is zero. After this first round, the encrypted message is $(C_1, C_2){=}(rB, x{+}rK{+}\sum_{i=1}^{m} s_iB)$. Let $(\tilde{C}_{1,0}, \tilde{C}_{2,0}) = (C_1, C_2)$ be a ciphertext resulting from the first round.

In the second round, each server partially and sequentially modifies this ciphertext. More specifically, when server S_i receives the modified ciphertext $(\tilde{C}_{1,i-1}, \tilde{C}_{2,i-1})$

from server S_{i-1}, it computes $(\tilde{C}_{1,i}, \tilde{C}_{2,i})$ as

$$\tilde{C}_{1,i} = s_i \tilde{C}_{1,i-1} \tag{4}$$

and

$$\tilde{C}_{2,i} = s_i \left(\tilde{C}_{2,i-1} - \tilde{C}_{1,i-1} k_i\right) \tag{5}$$

Once all of these computations are done, we discard the first component $\tilde{C}_{1,m}$ and obtain

$$\tilde{C}_{2,m} = sx + \sum_{i=1}^{m} s_i s B \tag{6}$$

where $s = \prod_{i=1}^{m} s_i$ is a short-term collective secret corresponding to the product of each server's fresh secret. $\tilde{C}_{2,m}$ is the deterministic tag collectively computed from the original ciphertext (C_1, C_2).

In fact, each server S_i uses the same s_i for all the ciphertexts for a given query. Thus, if two messages x_a and x_b are the same, then the corresponding tags will be the same. In our case, this sub-protocol is used to verify the query conditions, namely the `WHERE` clause and `GROUP BY` statement.

Zero-knowledge proofs for the distributed deterministic-tag. Each time a server adds a secret value or computes Equations (4) and (5), it must also compute a zero-knowledge proof to prove that the computations were done correctly. In the second round, when computing $(\tilde{C}_{1,i}, \tilde{C}_{2,i})$, server S_i is the prover and any entity can act as a verifier. Coming back to Equation (1) in Section 2.3, which we recall is $A_1 y_1 + A_2 y_2 = A$, it is easy to see that for Equation (5), $y_1 = s_i$, $y_2 = k_i s_i$ are the discrete logarithms of $s_i B$ and $k_i s_i B = s_i K_i$, respectively. The points $s_i K_i$, $s_i B$, $A = \tilde{C}_{2,i}$, $A_1 = \tilde{C}_{2,i-1}$ and $A_2 = -\tilde{C}_{1,i-1}$ on $\mathcal{G}$ are made public and are part of the proof. The publication of $s_i B$ also guarantees that server S_i has used the same secret s_i for all data during a given query. Similar proofs can be obtained for the first round of the protocol and for Equation (4) and are sketched in Appendix A.1.

4.2.2 Collective Aggregation

Given the ElGamal ciphertext tuples $(C_{1,i}, C_{2,i})$ held by each server S_i, the CA will produce one ciphertext $(C_{1,\text{aggr.}}, C_{2,\text{aggr.}})$ as an aggregation of all ciphertexts $(C_{1,i}, C_{2,i})$. This aggregation is possible due to the additive homomorphic property of the ElGamal cryptosystem.

In order to improve performance, the CA can be organized into a tree structure, in which each server will wait to receive the ciphertext tuples from its children and sum them before passing the result on to its own parent.

Zero-knowledge proofs for the collective aggregation. Here, a zero-knowledge proof consists in publishing the ciphertexts and the result of their aggregation. Due to the confidentiality property of the ElGamal cryptosystem, publishing the ciphertexts does not leak any information about the underlying plaintexts. In order to verify these proofs, a verifier can simply sum all of the ciphertexts and check that it corresponds to the published output.

4.2.3 Distributed Results Obfuscation

The distributed results obfuscation sub-protocol (DRO) enables the CA to collectively and homomorphically add noise, sampled from a probability distribution satisfying the differential privacy requirements, to the query results. This ensures (ϵ, δ)-differential privacy for DPs without revealing to any entity the amount of noise added. This sub-protocol is composed of two phases: the initialization phase, executed by the CA in the setup of the system; and the runtime phase, performed by the servers in order to respond to each query.

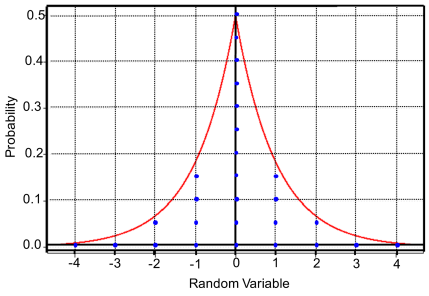


Fig. 3. Example of a quantization of Laplace distribution that is then used to derive noise values. The Laplace curve is in red and the quantas are in blue.

In the initialization phase, to generate the probability-distribution curve, a server in the CA uses the globally applicable predefined and publicly available differential privacy parameter ϵ. The same server uses the predefined probability quantum parameter θ in order to quantize an approximate representation of the distribution curve. This enables the server to derive a list of noise values and to randomly choose a value that can then be added to the query result to ensure (ϵ, δ)-differential privacy. An example of the quantization of a Laplace distribution with $\theta = 0.05$ is depicted in Figure 3. We use the number of quantas (blue dots) that fit under the curve in order to approximate the distribution and to create the list of noise values. In our example, the noise value list contains 11 '0's, 4 '1's and '-1's, 2 '2's and '-2's, ...

In the runtime phase, one server starts by verifying if the given query has already been answered by the system. We assume that no entity can know that two different queries yield the same results. To compare two queries, the server first retrieves, in the DL, all the queries that have been executed with the same attributes. For each of these, the server verifies if the values for the encrypted attributes match those in the new query. In order to do this, it subtracts both queries and tags the result along with a 0 by executing a DDT sub-protocol. Finally, the server verifies

if $\forall j,\ \ DT(E_K(Q_{new,j} - Q_{i,j}) = DT(E_K(0))$, where $Q_{i,j}$ is the jth attribute of query i and *new* is the new query. If all these equalities hold, the two queries are equal and the server adds the same noise to the new query results. This ensures that the noise cannot be averaged out. If the same query has not already been answered, the server starts the verifiable shuffle sub-protocol, described in Step 3, on the list of noise values. The re-randomization in the verifiable shuffle sub-protocol encrypts the clear text noise values. The zero-knowledge proof of the verifiable shuffle ensures the correctness of the computations. This results in a verifiable encryption and shuffling of the noise values. Then, the same server chooses the first element in the shuffled list as the noise value to be added. The verifiable shuffle sub-protocol ensures that the noise is chosen randomly from the proper distribution and that no entity knows its value. This noise is added to the query results and stored along with the query in the DL.

Zero-knowledge proofs for the distributed results obfuscation. In the initialization phase, the correctness of the computations can be verified by checking the value of the parameters and the generated noise values that are all publicly stored in the DL. For the runtime phase, the integrity of the computations is ensured by the zero-knowledge proofs of both the verifiable shuffle presented in Section 2.4 and the homomorphic addition described in Section 4.2.2. Given the quantization of the distribution, we prove the following theorem.

Theorem 1. *Let Laplace*$(0, b)$ *with* $b = \frac{\Delta f}{\epsilon}$ *be the Laplace distribution from which the noise is to be sampled,* θ *the unit quanta,* $[-T, T]$ *the range of integer noises (*T *is the integer bound), and* L *the length of the generated noise list in the initialization phase. Our mechanism* $\mathcal{M}$ *provides* (ϵ, δ)*-differential privacy where* $\delta = \frac{1}{L}$*, if we choose* $\theta = \frac{1}{2b}e^{-T/b}$.

Proof. Let w be the noisy output, μ_1 the original output for dataset D_1, and μ_2 the original output for dataset D_2. We have

$$\begin{aligned}
\frac{Pr[\mathcal{M}(D_1) = w]}{Pr[\mathcal{M}(D_2) = w]} &= \frac{\lceil \frac{1}{2b}e^{-|w-\mu_1|/b}/\theta \rceil / L}{\lceil \frac{1}{2b}e^{-|w-\mu_2|/b}/\theta \rceil / L} \\
&\leq \frac{\frac{1}{2b}e^{-|w-\mu_1|/b}/\theta + 1}{\lceil \frac{1}{2b}e^{-|w-\mu_2|/b}/\theta \rceil} \\
&\leq \frac{\frac{1}{2b}e^{-|w-\mu_1|/b}/\theta}{\frac{1}{2b}e^{-|w-\mu_2|/b}/\theta} + \frac{1}{\lceil \frac{1}{2b}e^{-|w-\mu_2|/b}/\theta \rceil} \\
&= e^{(|w-\mu_2| - |w-\mu_1|)/b} + \frac{1}{\lceil \frac{1}{2b}e^{-|w-\mu_2|/b}/\theta \rceil} \\
&\leq e^{|\mu_1-\mu_2|/b} + \frac{1}{\lceil \frac{1}{2b}e^{-|w-\mu_2|/b}/\theta \rceil} \\
&\leq e^{\epsilon} + \frac{1}{\lceil \frac{1}{2b}e^{-|w-\mu_2|/b}/\theta \rceil}.
\end{aligned} \tag{7}$$

Therefore, we have

$$Pr[\mathcal{M}(D_1) = w] \leq e^{\epsilon} Pr[\mathcal{M}(D_2) = w] + \frac{1}{L}. \tag{8}$$

When w is at the boundary of $\mathcal{M}(D_1)$, w might be the output for D_1 but is not a possible output for D_2, hence $Pr[\mathcal{M}(D_2) = w] = 0$. For example, if $\mu_1 = \mu_2 - 1$, we have $\mathcal{M}(D_1) \in [\mu_1 - T,\ \mu_1 + T]$, and $\mathcal{M}(D_2) \in [\mu_2 - T,\ \mu_2 + T]$, and thus $Pr[\mathcal{M}(D_2) = \mu_1 - T] = 0$. If we choose $\theta \geq \frac{1}{2b}e^{-T/b}$, then a boundary noise is sampled with probability $\frac{1}{L}$, hence the formula $Pr[\mathcal{M}(D_1) = w] \leq e^{\epsilon} Pr[\mathcal{M}(D_2) = w] + \frac{1}{L}$ still holds with $Pr[\mathcal{M}(D_2) = w] = 0$. Nevertheless, we should choose T large enough such that $\frac{1}{L}$ is sufficiently low to achieve strong differential privacy. □

4.2.4 Key Switch

The key switch sub-protocol enables the conversion of an ElGamal ciphertext of a message x encrypted under the CA's public key K to one of the same message x encrypted under any known public key, e.g., the querier's public key U, without ever decrypting. The sub-protocol is described below.

We start with $\mathrm{E}_K(x) = (C_1, C_2) = (rB, x + rK)$, a ciphertext tuple corresponding to the ElGamal encryption of message x using the CA's public key K. Let $(\tilde{C}_{1,0}, \tilde{C}_{2,0}) = (0, C_2)$ be the initial modified ciphertext tuple. Each server partially and sequentially modifies this element. More specifically, when server S_i receives $(\tilde{C}_{1,i-1}, \tilde{C}_{2,i-1})$ from server S_{i-1}, it generates a fresh random nonce v_i and computes $(\tilde{C}_{1,i}, \tilde{C}_{2,i})$ as

$$\tilde{C}_{1,i} = \tilde{C}_{1,i-1} + v_i B \tag{9}$$

and

$$\begin{aligned}
\tilde{C}_{2,i} &= \tilde{C}_{2,i-1} - (rB)k_i + v_i U \\
&= \tilde{C}_{2,i-1} - rK_i + v_i U.
\end{aligned} \tag{10}$$

where $v = v_1 + \ldots + v_m$. Once all of these are computed, we obtain the new ciphertext that corresponds to x encrypted under the public key U, $(\tilde{C}_{1,m}, \tilde{C}_{2,m}) = (vB,\ x + vU)$ from the original ciphertext (C_1, C_2). At this point, the ciphertext $(\tilde{C}_{1,m}, \tilde{C}_{2,m})$ can be decrypted only by the holder of the private key u, where $U = uB$.

Zero-knowledge proofs for the key switch. To prove that the computations have been done correctly, each time a server computes Equations (9) and (10), it must also compute a zero-knowledge proof. Again, at each step i, server S_i is the prover and any entity can be the verifier. Coming back to Equation (1) in Section 2.3, it is easy to see that for Equation (10), $y_1 = k_i$, $y_2 = v_i$ are the discrete logarithms of $k_i B = K_i$ and $v_i B$, respectively. All points K_i, $v_i B$, $A = \tilde{C}_{2,i} - \tilde{C}_{2,i-1}$, $A_1 = -rB$ and $A_2 = U$ are made public and do not leak any information about underlying secrets. A similar proof can be obtained for equation (9) and is sketched in Appendix A.2.

4.2.5 Dynamic Collective Authority

This sub-protocol enables us to add/remove a server to/from the collective authority. On the one hand, adding more servers strengthens the privacy guarantees; on the other hand, detecting misbehavior, for example through the use of zero-knowledge proofs, should lead to the culprits' exclusion from the CA. When a server joins/leaves the CA, this server has to collaborate with all the DPs in order to change their data encryption from the CA's previous public key to the new one. We assume, here, that DPs want to outsource computations and that the joining/leaving server participates in the protocol.

Without loss of generality, let S_m be the server that needs to be added/removed. in order to have its data encrypted under the CA's new public key K_{new}, any DP storing data encrypted using the CA's previous public key K_{prev} must execute the following protocol. When adding a new server S_m to the collective authority $S_1,\ldots,S_{m-1}$, $K_{\text{prev}} = K_1 + \ldots + K_{m-1}$ and $K_{\text{new}} = K_1 + \ldots + K_m$. When removing server S_m from $S_1,\ldots,S_m$, $K_{\text{prev}} = K_1 + \ldots + K_m$ and $K_{\text{new}} = K_1 + \ldots + K_{m-1}$. Starting from a message x encrypted under K_{prev}, $(C_1, C_2) = (rB, x + rK_{\text{prev}})$, server S_m multiplies C_1 by its private key k_m

$$C_1 k_m = (rB)k_m = rK_m \tag{11}$$

and adds/removes, the result to/from, C_2

$$\tilde{C}_2 = C_2 \pm rK_m = x + rK_{\text{prev}} \pm rK_m = x + rK_{\text{new}}.8 \tag{12}$$

Component $\tilde{C}_1$ remains the same, i.e., $\tilde{C}_1 = C_1$. The result is the new ciphertext tuple $(\tilde{C}_1,\ \tilde{C}_2) = (rB,\ x + rK_{\text{new}})$ corresponding to the same message x encrypted under the CA's new public key. Hence, it is possible to expand the CA and update the corresponding encryptions without needing to decrypt any of the ciphertexts. Finally, the DPs that trusted the server who left the CA can choose another server to trust or leave the system.

Zero-knowledge proofs for the dynamic collective authority. Using Equation (1) in Section 2.3, we see that for Equation (12), $y_1 = k_m$ is the discrete logarithm of $k_m B = K_m$. All points K_m, $A = \tilde{C}_2 - C_2$ and $A_1 = -C_1$ are made public and do not leak any information about underlying secrets. A similar proof can be obtained for Equation (11) and is sketched in Appendix A.2

5 Security and Privacy Analysis

We analyze UnLynx's security and privacy by studying each step of our decentralized data-sharing protocol presented in Section 4.1 and by showing, for each step, how UnLynx guarantees the security and privacy requirements presented in Section 3.1.

Step 0. In the initialization step, each server i builds its own key pair $(k_i,\ K_i)$ and, as long as one server keeps its secret key k_i hidden, the CA's secret key $k = \sum_{i=1}^{m} k_i$ remains unknown. As a result, data confidentiality is ensured for DPs through the use of the CA's public key $K = \sum_{i=1}^{m} K_i$ to encrypt their data. The correctness of the noise-value generation is ensured by the fact that all the parameters and values are stored in a public, immutable, distributed ledger DL, and the computations can be verified by any entity.

Step 1. UnLynx rules out unauthorized queries by checking the querier's permission in the DL. Moreover, because DPs publicly log the fact that they respond to a query, a server's malicious behavior, such as excluding a DP from a query or impersonating a DP, will be caught.

Step 2. Data authenticity is ensured by the DPs' digital signatures on the responses. Moreover, UnLynx enables an optional upper bound that can be used to hide the amount of data sent by DPs. More specifically, DPs either discard some records or add dummy responses that consist of responses with the SELECT attributes, all equal to $E_K(0)$; and the filtering attributes uniformly distributed over the range of possible values. This prevents an adversary from inferring any information (e.g., using traffic analysis) from the amount of data sent by a given DP.
As described in Section 3.2, we consider DPs to be honest-but-curious, which means that we assume they provide correct responses to a query and do not collude with any other entity. Otherwise, if malicious DPs collude with the querier or some of the servers, they could infer some information about other (non-colluding) DPs. Nevertheless, UnLynx would still offer some protection when this is the case. In fact, when colluding with the querier, malicious DPs would only have access to an approximation of the query results of their target(s) because of the oblivious noise addition done in Step 6. When colluding with one or multiple servers, malicious DPs would be able to observe the output of the Distributed Deterministic Tagging sub-protocol (Step 4). In the worst case, if there were only malicious DPs connected to a malicious server, they would be able to infer the mapping between an attribute and its deterministic tag. Yet, the data would be shuffled (Step 3) and mixed with dummy records (Step 2) beforehand. This would still ensure the unlinkability of the data and the secrecy of the honest DPs responses distribution.

Step 3. Unlinkability is guaranteed by the verifiable shuffle sub-protocol. Data confidentiality is maintained as the data are never decrypted during the protocol.

Step 4. The tag $(sx + \sum_{i=1}^{m} s_i sB)$ is a collective encryption of the filtering attribute x because each s_i is known only to server S_i. Hence, the confidentiality of the filtering attributes is guaranteed. Each step of this protocol is publicly verifiable due to the zero-knowledge proofs.

Step 5. A misbehaving server can be caught due to the zero-knowledge proofs for homomorphic additions.

Step 6. The input list of noise values is publicly known, and the shuffling and aggregating operations can be verified. Any entity can check that the noise added is one of the values in the list, without learning which one. Moreover, the queries are publicly logged and the same noise is used for identical queries. By providing (ϵ, δ)-differential privacy, UnLynx is resilient against (colluding) queriers/servers trying to infer information from query outputs and against other types of attacks such as set difference attacks presented by Souza et al. [19]. UnLynx maintains a global privacy budget, defined by Dwork [24], which is updated at each executed query by computing the sensitivity or the privacy loss of the query and by subtracting this value to the global budget. In fact, with (ϵ, δ)-differential privacy, the privacy loss is additive and the privacy budget imposes a limit on the cumulative values ϵ and δ after which new queries are not allowed. The choice of this parameter highly depends on the application domain and is out of the scope for this paper.

Step 7. Each step of the key switch sub-protocol can be verified and the result can be decrypted only by the querier. Each server can check that the encryption is indeed switched to the querier's public key and, if this is not the case, any server can block the process.

Step 8. If the querier does not receive the final result from the server to which it is connected, e.g., because the server is unresponsive, it can simply send the same request to another server.

In conclusion, data are encrypted during the whole protocol execution, therefore data confidentiality is not compromised at any step. All computations are publicly verifiable due to the use of zero-knowledge proofs. Finally, UnLynx provides (ϵ, δ)-differential privacy and unlinkability between DPs and their data through to the use of new distributed and secure sub-protocols, namely distributed deterministic-tag, distributed results obfuscation and verifiable shuffle sub-protocols.

6 UnLynx's Possible Extensions

By analyzing UnLynx and its set of protocols, we can identify two potential improvements: (1) data integrity/input validation and (2) a way to exclude a non-cooperative server from the CA. We propose two new solutions that can be implemented in the next version of this system.

6.1 Input-Range Validation

In this paper, we assume DPs are honest-but-curious and we do not ensure the correctness of the data they provide. Nevertheless, to limit the effect of DPs who introduce invalid data, we propose an input-range validation technique. Adding input-data validation in UnLynx is not intended for, and will not help in, situations where a DP injects a large amount of invalid data (many data records), but it can limit the damage to cumulative query results if only a few of a DP's records (inadvertently or maliciously) contain invalid data, e.g., a data-entry error by an organization employee. This would enable us to relax the assumption that DPs input only correct data. Camenisch et al. [15] present simple and efficient zero-knowledge proofs to prove that an encrypted value is in a specific range/set. If the encrypted value is an integer I, a zero-knowledge proof consists of proving that $I = \sum I_j \cdot b^j$ and that each b-ary digit of this integer is between $[0, b-1]$. For the first part of the proof we can adapt Equation (1), and as for the second part, we can use the set membership proofs provided by Camenisch et al. [15].

6.2 Enabling a Dynamic Collective Authority

In Section 4.2.5, we propose a sub-protocol that enables the system to remove/add a server from/to the CA when this server collaborates. However, this is not always the case and UnLynx might want to exclude a misbehaving server that refuses to leave.

The first solution is to require the DPs to re-encrypt their data with the CA's new public key, assuming that DPs keep off-line backups of their data. When this is not possible, we propose that a threshold of t (out of $m-1$) servers reconstruct the secret key of the leaving server S_m through the use of a $(t,\ m-1)$-verifiable secret-sharing (VSS) scheme [18]. In such a scheme, a potentially dishonest dealer can share S_m's secret key k_m, among the $m-1$ remaining servers, in such a way that any t semi-honest servers can reconstruct k_m but any subset of $t-1$ servers learn nothing about k_m. This secret sharing should be done for all servers when they join the CA. In this way, when S_m is removed from the CA, its private key can be reconstructed by the remaining CA servers. This VSS weakens the threat model defined in Section 3.2 but enhances the dynamism of the CA by enabling it to discard a misbehaving or unresponsive server. In fact, by using a $(t,\ m-1)$-verifiable secret-sharing scheme, the security of the scheme is guaranteed, as long as t out of $m-1$ servers are honest or honest-but-curious, instead of only 1 in the Anytrust model.

7 Performance Evaluation

We start with a theoretical analysis of UnLynx's computation and communication complexities. Then, we discuss our experimental setup and evaluate UnLynx's performance. We consider the performance when producing the results for *one* single query. We demonstrate that UnLynx yields acceptable performance and is scalable with the amount of data and the size of the CA.

7.1 UnLynx Complexity

We discuss the time and communication complexities for each of our sub-protocols. We denote by m the number

of servers in the CA, r the total number of records sent by all the DPs, and f and g the number of attributes in the WHERE clause and in the GROUP BY statement, respectively. The number of different combinations of GROUP BY attributes is denoted gd and we usually have $r >> gd$. Finally, t is the size of the noise-values list, and a is the number of attributes in the query SELECT statement. We discuss the complexity of each sub-protocol and UnLynx's overall complexity.

Verifiable Shuffle. In this sub-protocol, all the ciphertexts have to be shuffled and re-randomized by all the servers, resulting in a computation and communication complexity of $\mathcal{O}(m{\cdot}r{\cdot}(f{+}g{+}a))$.

Distributed Deterministic Tag. In this sub-protocol, only the attributes in the WHERE clause and in the GROUP BY statement are processed. Both the computation and communication complexities are therefore $\mathcal{O}(m{\cdot}r{\cdot}(f{+}g))$.

Collective Aggregation. Before executing this sub-protocol, the responses are locally filtered and aggregated by each server, which means that the number of responses is reduced from r to $m \cdot gd$. Moreover, the size of the responses is reduced from $g{+}f{+}a$ to $g{+}a$, as the WHERE clause attributes are no longer useful and can be discarded. The servers can be organized in a binary tree structure, such that each server aggregates the results of its children and its own. Hence, the computation complexity is $\mathcal{O}(log_2(m){\cdot}gd{\cdot}(g{+}a))$ and the communication is $\mathcal{O}((m{-}1){\cdot}gd{\cdot}(g{+}a))$.

Distributed Results Obfuscation. Both the computation and communication complexities of this sub-protocol are $\mathcal{O}(m{\cdot}t)$ and correspond to the complexity of shuffling the noise values. Other parts of the protocol incur a negligible workload.

Key Switch. The complexity depends mainly on the number of servers and on the size of the responses. Both the computation and communication complexities are $\mathcal{O}(m{\cdot}gd{\cdot}(g{+}a))$.

Overall Complexity. The overall complexity of our protocol can be reduced to the complexity of the verifiable shuffle sub-protocol, that is responsible for most of the computation and communication.

We recall that even if most of the sub-protocols need to be executed sequentially by the servers, the computations performed locally by each server are highly parallelizable.

7.2 System Implementation

We implemented UnLynx in Go [5], and the experimental code is publicly available at https://github.com/lac1/unlynx [7]. We relied on Go's native crypto library and the public advanced crypto DeDiS library [2]. The latter includes an implementation of a verifiable shuffle of sequences of ElGamal pairs [34] and zero-knowledge proofs for general statements about discrete logarithms [16]. We used ElGamal encryption on the Ed25519 elliptic curve [12] with 128-bit security. More specifically, our prototype implements all the sub-protocols described in Section 4. In all of these sub-protocols, we assume the existence of a distributed ledger for which the implementation is future work. This means that the proofs of correctness are stored in global variables and the query logging and equality checking are not implemented yet. The communication between different participants relies on TCP. Finally, in order to allow for an easy deployment of UnLynx in different environments, we implemented an application that automatically handles the creation of the CA on multiple servers and provides queriers with an easy way to query the system.

7.3 System Evaluation

We used Mininet [6] to simulate a realistic virtual network between servers. Each CA server ran on a separate machine and was connected to the others by a 1Gbps link with a communication delay of 10ms. For each of our servers, we used machines with two Intel Xeon E5-2680 v3 CPUs with a 2.5GHz frequency, 256GB RAM that supports 24 threads on 12 cores. In our performance evaluation, we study the execution time of Steps 3 to 7 presented in Section 4. We do not include the time needed to initialize the system (Step 0) or for the data providers to encrypt their data since these operations are done once and offline. The time needed for the querier to build the query (Step 1), for the DPs to send their responses (Step 2), and for the querier to decrypt the results (Step 8) are also left out. For Step 1 and Step 8, the runtime is negligible in comparison to the whole process, whereas the time needed for Step 2 depends almost entirely on the communication links between the servers and the DPs. In the following, we describe the default parameters used in our experiments and we observe the influence of each parameter on the overall system separately.

Parameter	Default Value
# of servers	3
# of responses in tot.	15,000
# of filtering attributes	2
# of possible groups	10
# of aggregating attributes	10
# of noise values	1,000

Table 2. Default parameter values used for the evaluation of UnLynx.

We simulated distributed computations on 15,000 responses, evenly distributed among 3 servers. A response was considered to contain 2 filtering attributes, e.g., one in the WHERE clause and one in the GROUP BY statement, and 10 aggregating attributes. We assume 10 different groups, i.e., GROUP BY attributes can form up to 10 different group combinations. We chose to use a list of 1,000 obfuscating noise values. The default parameters are summarized in Table 2. In the following graphs, each measurement is averaged over 10 independent runs.

We begin our evaluation by showing how our decentralized data-sharing protocol is collectively executed by three servers (S_1, S_2, S_3) with the default parameters described above. The results are shown in Figure 4.

Recall, each server has to run the verifiable shuffle and

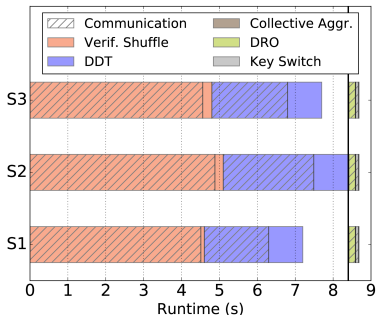


Fig. 4. Runtime for the different servers ($S1$, $S2$, $S3$) in the CA.

distributed deterministic-tag sub-protocols on the data received from their DPs. A server can run these two sub-protocols sequentially, without having to synchronize with the others. Nevertheless, they are still required to participate in the sub-protocols when requested. As a consequence, both sub-protocols can be executed in parallel by the CA, hence we can efficiently distribute the workload among the servers. Once these first two steps are finished, the server responsible for initially processing the query begins a collective aggregation. Finally, the same server sequentially executes the distributed results obfuscation sub-protocol and then the key switch sub-protocol.

By further analyzing the graph, we see that the first two sub-protocols are the most time consuming, as they are required to process all the DPs' responses. In contrast, the last two sub-protocols are executed significantly faster, as they are only required to process aggregate responses. The DRO execution time is constant for a given number of servers.

Using these observations, we often group the two first sub-protocols - "Ver. Shuffle + DDT" - and the three last sub-protocols - "Other" - in our experimental results. Moreover, we always separate the runtime of the sub-protocols and their respective proofs, as the proofs can be verified offline. At runtime, servers are required to save all the information needed (e.g, the ciphertexts and the public values derived from the secret/ephemeral keys used) to create the proofs in order to be able to generate them when requested. Finally, we observe that the communication is the most time consuming process and accounts for 75% of the overall execution time. We now study the scalability of UnLynx against different parameters.

Varying the number of responses. To show that UnLynx scales almost linearly with the total number of responses, we begin by increasing the number of responses processed by each server in the collective authority.

Figure 5(b) shows the time it takes for *all* servers to produce *all* zero-knowledge proofs, as well as the time needed for an entity[3] to verify these proofs. We do not consider the time it takes for the servers to publish their proofs and for a verifier to download the data necessary to verify them.

Results from Figure 5(a) show evidence of UnLynx's scalability. In fact, UnLynx is able to satisfy a request with 150K responses in less than 73 seconds. However, in Figure 5(b), we observe that the zero-knowledge proofs incur a non-negligible computational overhead. For example, a query with 15K responses is answered in less than 9 seconds when it does not include proofs, whereas the complete execution takes almost 64 seconds. This represents an expanding factor of 7.

Varying the response size. We study the runtime of UnLynx against the number of attributes in each response. We consider that half of the attributes in a response are filtering attributes. Figure 5(c) shows that the runtime increases almost linearly with respect to the size of the responses, and the proofs bear a non-negligible overhead as shown in Figure 5(d).

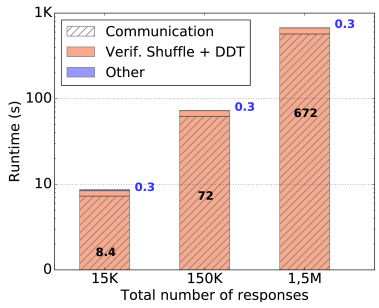
Varying the number of groups. We observe UnLynx's runtime when increasing the number of possible groups or, in other words, the number of different combinations of `GROUP BY` attributes. This is plotted in Figures 5(e) and 5(f). As the number of responses is constant, the execution time for the verifiable shuffle and DDT (Steps 3 and 4) are constant and, combined, they take less than 9 seconds on average. The proof creation and verification also remain constant at 29 and 15 seconds, respectively. The runtime, both with and without proofs, increases with the number of groups but remains within acceptable boundaries. For example, with 1,000 possible groups, the execution time without proofs takes approximately $9+3.3=12.3$ seconds.

Varying the number of servers. We assess the effect of the size of the CA on UnLynx's runtime. Figure 6 shows that the latency increases slightly with an increasing number of servers. This is explained by the fact that the workload and data are distributed among a larger number of servers, thus improving the parallelization of UnLynx. Nevertheless, adding a server increases the number of steps needed to complete each of the sub-protocols, which hinders the positive effect of improved parallelization.

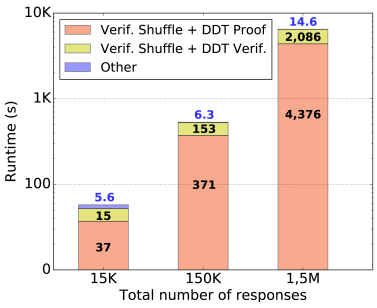
7.3.1 Storage Overhead

We now discuss the storage overhead induced by ElGamal encryption. We recall that our encryption relies on an el-

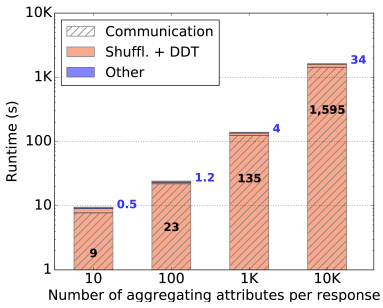
3 We assume that this entity has the same computing power as one of the servers.



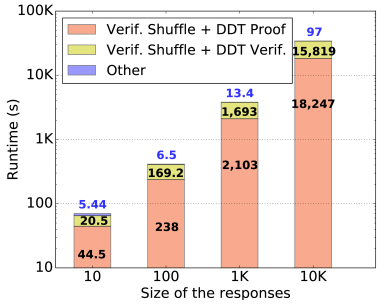
(a) Runtime vs. total number of responses.



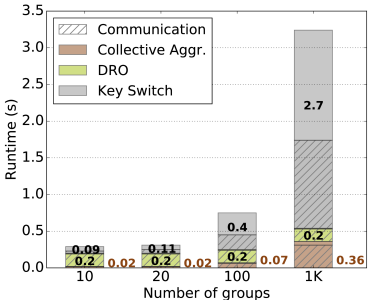
(b) Proof creation and verification runtime vs. total number of responses.



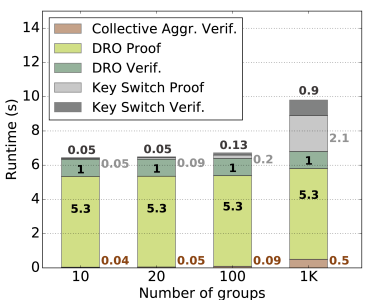
(c) Runtime vs. size of the responses (half of each response is filtering attributes and other half is aggregating attributes).



(d) Proof creation and verification runtime vs. responses' size.



(e) Runtime vs. number of groups.



(f) Proof creation and verification runtime vs. number of groups.

Fig. 5. Performance evaluation of UnLynx (using the default parameters presented in Table 2). (a), (c), and (e) do not include proof creation or verification. In (b), (d) and (f), *all* proofs from *all* servers are computed and verified.

liptic curve with 128-bit security and that each encrypted message is a pair of points on the curve. Each point is encoded with 32 bytes, hence each encrypted message is 64 bytes long. Therefore, the encryption of an integer (4 bytes) in clear text yields an expansion factor of $64/4=16$. For example, assume each DP's database contains 10,000 lines and 120 columns. Each line contains data belonging to one individual, and the columns correspond to attributes. Here, the amount of data stored by each DP is 4.8MB if it is stored in clear, and 73.25MB if it is encrypted. Finally, storage overhead on each CA server is negligible, as data can be temporarily stored and discarded after locally aggregating the data.

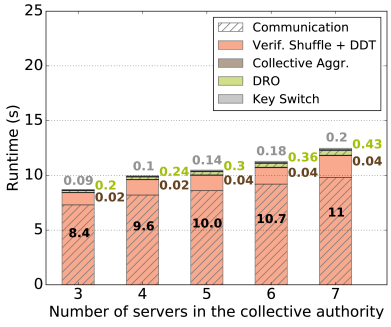


Fig. 6. Runtime for a varying number of servers in the CA.

7.3.2 Communication Overhead

As shown by our performance evaluation, most of the sub-protocol's execution time is dedicated to communication. We use the same example as in the previous section and assume that a query requests 120 columns, half of which are filtering attributes. In this case, each DP sends 73.25MB to the servers. Considering six DPs, this results in 439.5MB of data to be processed by the CA. The communication overhead for each sub-protocol is given below.

Verifiable Shuffle. All the data have to be sent through all three servers, resulting in a communication overhead of $439.5\times3=1315.5$MB.

Distributed Deterministic Tag. In this sub-protocol, only the attributes in the `WHERE` clause and in the `GROUP BY` statement need to be sent, which totals to $\frac{439.5}{2}\times3=659.25$MB.

Collective Aggregation. Before executing this sub-protocol, the responses are locally filtered and aggregated by each server. With 10 possible groups, we obtain a communication overhead of only 0.14MB for when all the filtering attributes are in the `GROUP BY` statement (worst case scenario). We recall that `WHERE` clause attributes can be discarded after the responses have been filtered.

Distributed Results Obfuscation. The amount of traffic for this sub-protocol depends on the number of servers and the size of the noise value list that, in this case,

is 1,000 clear text integers. Therefore, for this particular example, we send around 0.004MB ×3=0.012MB of data.

Key Switch. In this sub-protocol, only the final results of the query are processed. With 10 possible groups, the highest possible communication overhead is 0.2MB.

Finally, in order to privately process 439.5MB of data, UnLynx needs to send 1315.5+659.25+0.14+0.012+0.2= 1975.1MB, which is 4.5 times the original amount of data.

In Figure 7, we observe the influence of the bandwidth capacity and communication delay between the servers. By using the default parameters presented in Table 2 and with 1Gbps links, the maximum communication rate observed was 80Mbps. We show, for increasing latency, the complete runtime of UnLynx when the communication rate is not limited (80Mbps) and when it is reduced to 40Mbps and 20Mbps. As expected, the computation time is constant, around 1.3 seconds, and the communication time increases with both the bandwidth and the transmission delay. We observe that when the delay increases, reducing the bandwidth from 80Mbps to 40Mbps does not have a significant effect on the overall runtime. This occurs because when the delay increases the maximum communication rate also decreases.

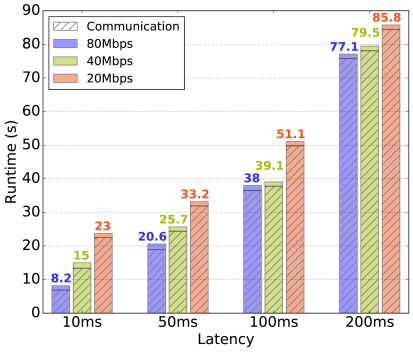


Fig. 7. Runtime for different values of bandwidth capacity and latency for the links between the servers.

7.3.3 Dynamic Collective Authority

We observe the latency incurred by adding/removing a server to/from the CA. We do not include the time to transfer the data between DPs and servers. We assume that the server leaving/joining the CA is willing to participate in the process.

The results are shown in Figure 8 and depict an almost linear increase in runtime with the total number of ciphertexts collectively held by the DPs.

8 Example Application: Secure Survey

To illustrate that UnLynx is usable in practice, we present a realistic use case of a secure distributed survey. We also further assess our system's performance and study the possible tradeoffs that can be made in order to improve UnLynx's response time.

We show that our system could improve and simplify the process of carrying out a survey on sensitive personal data. In fact, such surveys are usually done on data that are anonymized, hence reducing the precision of the results. Moreover, and as demonstrated by D. Bogdanov et al. [13], obtaining the permission to access such sensitive data is administratively heavy, requires the participation of multiple data-protection entities (local government, European Union, ...) and is extremely time and money consuming. UnLynx enables us to do a secure and privacy-preserving survey on data that are encrypted at rest and during computations, and that remain under the DPs' control throughout the process. This can tremendously facilitate the access to data and the achievement of such surveys.

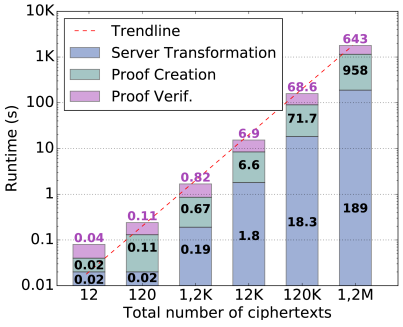


Fig. 8. Runtime for adding/removing a server to/from the CA.

We consider a realistic example where a statistical institute wants to compute the average salary of the top 20 biotechnology companies in the United States. The query is '`SELECT AVG(`*salaries*`) FROM` C_1,...,C_{20} `WHERE` *age in* [40: 50] `AND` *ethnicity* = *Caucasian* `GROUP BY` *gender*'. A `SELECT AVG(*)` query is executed by combining the results of the '`SELECT SUM(`*salaries*`), COUNT(*)`' query on the same filtering attributes. In our example, the DPs are the companies, the querier is the statistical institute and the collective authority corresponds to three servers handled by the statistical institute, a consortium of the companies and the US government (who wants to ensure that data are protected). We assume that each company has 20K employees.

In this scenario, which we refer to as our baseline, all data are encrypted and all the zero-knowledge proofs are created and verified. Then, we discuss two possible tradeoffs that can be used in UnLynx and compare their performance.

The first tradeoff is to consider all servers as non-malicious, hence to not ask for proofs of correctness. This reduces the runtime by at least 90%, as shown in Figure 9. A verifier can also randomly ask for - and check - part of the proofs or request proofs from a single server. In both cases, this can be done offline and the proof-execution time decreases proportionally to the amount of omitted proofs.

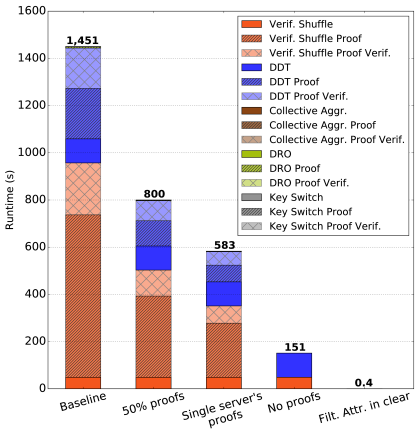


Fig. 9. Protocol runtime considering the tradeoffs between security/privacy and efficiency in the case of a secure census.

As servers cannot anticipate which proofs will be verified, they must still compute correctly, if they do not want to be caught cheating.

The second tradeoff is for DPs to have all the filtering-attribute values stored in clear text. This enables servers to locally aggregate DPs' responses for each combination of filtering attributes, thus reducing the number of responses to be processed from the number of records sent by the DPs to the number of different filtering attribute combinations. The latter usually being considerably smaller than the former.

In Figure 9, we can verify that each of these tradeoffs significantly enhances performance. Producing and verifying 50% of the proofs at each server reduces the execution time by 45%, whereas having all DPs send clear text filtering attributes reduces it by 99%.

Considering the results shown in Figure 9, it becomes obvious that, in order to control the computation and storage overhead, the categorization of attributes as either sensitive or non-sensitive is key in deciding what needs to be encrypted. We suggest the following guidelines for an efficient and privacy-preserving solution.

Non-sensitive attributes (e.g., age, gender or ethnicity). Stored in clear and protected by privacy-protection techniques yielding, for example, k-anonymity [43].

Sensitive attributes (e.g., salary). Store encrypted under the CA's public key K.

We provide an example database, Table 3, that respects these guidelines where $\mathrm{E}_K(x)$ refers to the ElGamal encryption of message x.

Finally, we argue that a response time of 24 minutes, for a secure distributed and privacy-preserving survey on 400,000 records, is acceptable. We recall that this response time does not include Steps 0, 1, 2 and 8, as explained in Section 7. In this specific case, if we assume that (1) DPs machines have the same settings as the servers described in Section 7, (2) data are encrypted beforehand and (3) DPs respond all at the same time, then the time to transmit these data (Step 2) is around 0.4 seconds. This time depends exclusively on the communication link between the servers and DPs and on the amount of data to be sent.

ID	Gen.	Age	Ethnicity	...	Salary
P1	F	40:50	Caucasian	...	$\mathrm{E}_K(100{,}032)$
P2	M	40:50	Caucasian	...	$\mathrm{E}_K(10{,}009)$
P3	M	30:40	Asian	...	$\mathrm{E}_K(10{,}080)$
P4	F	30:40	Asian	...	$\mathrm{E}_K(100{,}014)$

Table 3. Proposed database structure. In this specific example, gender, age and ethnicity are non-sensitive filtering attributes and are therefore left in clear. To reduce the risk of identity disclosure, the table values are generalized to satisfy k-anonymity with $k=2$ for the quasi-identifiers age, gender and ethnicity. Salary is a sensitive attribute and is therefore encrypted under the CA's public key K

In conclusion, if we assume that the filtering attributes, namely age, gender and ethnicity are stored in clear, protected by anonymization techniques and that the proofs of correctness are not executed, then UnLynx's response time is reduced to 0.4 seconds as shown in Figure 9.

9 Related Work

In the database research community, various architectures have been proposed for efficient data-sharing and distributed-data management among different institutions. For example, PeerDB [36] is a peer-to-peer (P2P) distributed data-sharing system that offers capabilities for data management, content-based search and a flexible configuration of network topology. Yang et al. [48] propose a hybrid P2P system for distributed data-sharing that combines the efficiency of structured P2P networks and the flexibility of unstructured ones. Yet, these solutions, as opposed to UnLynx, provide almost no security or privacy guarantees for dealing with sensitive data.

Similarly, in the security and privacy community, most of the existing systems, which use cryptographic or statistical techniques to enable sensitive data protection and sharing, strive to provide the same features as UnLynx; they either lack efficiency or provide lower security.

In general, privacy-preserving data-sharing systems are designed as either centralized or decentralized. Due to low complexity and communication overhead, centralized systems such as CryptDB [39] and Mona [30], are popular in cloud computing and are usually more efficient than their decentralized counterparts. However, centralized systems provide weakest-link security and some assume a trusted third party [10, 20, 30, 39]. In the presence of a strong and persistent adversary that targets a single point of failure, these methods are inappropriate for handling sensitive data.

In order to avoid weakest-link security, some systems use a decentralized framework. For example, Duan et al. [21]

propose P4P (Peers for Privacy) for privacy-preserving data mining by employing a hybrid architecture that integrates the popular client-server paradigm and decentralizes the computation among a server and a number of peers. The framework assumes an adversary model with a number of constraints, such as a well-protected server and non-collusion between server and peers. Multi-server systems, which include those using public cloud servers, often make use of secret sharing [14, 26, 49] where a number of servers secret-share the data providers' data in order to guarantee confidentiality of the data, as long as a threshold number of the servers does not collude. A fundamental issue with data-secret sharing, however, is that data providers cannot store and manage their own data; instead, this is handled by the servers. Finally, storing sensitive data at a server might not be desirable, or even possible, especially if this server is physically in a different country or jurisdiction.

UnLynx does not present any of these limitations. Another family of decentralized frameworks ([11, 28, 33, 40, 42, 50]) is based on secure multiparty computation (SMPC) protocols that can theoretically perform any computation task without leaking any party's private data. Yet, several critical issues, with current SMPC solutions, render them impractical in real operational settings. For instance, state-of-the-art SMPC libraries cannot appropriately address computations that involve more than two participants (e.g., [11, 28, 33, 42]). The computation and memory costs of these solutions are already prohibitively high in a semi-honest adversary model, let alone when considering malicious behavior. UnLynx is designed to be inherently parallelizable to guarantee efficient sharing of sensitive data among any number of peers with strongest-link security and verifiable computations.

Finally, in contrast to the heavy cryptographic solutions, hardware-based solutions (e.g., [38, 41]) tend to become increasingly popular in the privacy and security research community, due to the recent technology advances in trusted hardware, such as Intel Software Guard eXtensions (SGX). Nevertheless, hardware-based solutions rely on the fact that the users *must* trust the hardware producers (e.g., Intel) who manage the master keys that are involved in some important protocols. Furthermore, even in the presence of trusted hardware, side-channel attacks based on memory and network access patterns have proven to be effective in many scenarios [29, 37, 47], which shows the immaturity of deploying such systems - at their current stage - to address critical challenges such as secure data sharing. Instead, UnLynx, is based on well-established cryptographic techniques that rely on a standard security model, and it provides a set of critical security features that none of previous contributions have achieved, such as proof of computation and decentralized trust. Hence, its strong security guarantees, coupled with its ability to efficiently support thousands of data providers, make UnLynx ready for immediate deployment in real operational environments.

In UnLynx, we propose an efficient protocol that provides differential privacy guarantees for queries executed on distributed databases. This solution, as most other existing solutions, relies on the addition of some noise that is derived from a probability distribution such that the final result respects differential privacy. Anandan et al. [9], Narayan et al. [32] and Mohammed et al. [31] propose to use secure two-party computation in order to jointly generate a probability distribution and obliviously derive a noise value from it. These solutions are efficient, but limited to only two parties. Chen et al. [17] propose a solution with multiple parties that work together with a trusted proxy in order to add a distributively created noise. Dwork et al. [22] remove the need for a trusted party by using verifiable secret sharing of noise values that are then combined in order to generate the noise. Nevertheless, secret sharing imposes that at least two-thirds of the parties are honest, whereas our protocol is secure in an Anytrust Model [46].

10 Conclusions and Future Work

UnLynx is a modular decentralized system for privacy-preserving data sharing among multiple data providers. Specifically, we enable a querier to obtain aggregate statistics for different grouping criteria on a set of different databases. We achieve this through protocols that enable a number of independent servers to compute on distributed data sets and that provide proofs of correctness of their work. We further build upon advances in several areas, such as zero-knowledge proofs and verifiable shuffling, and we bring them all together into UnLynx, providing security and privacy guarantees against malicious behavior. Additionally, by introducing a new protocol for distributed obfuscation of results, UnLynx ensures (ϵ, δ)-differential privacy for individuals sharing their data. The performance evaluation of our prototype shows that it is efficient and almost linearly scalable with the amount of data to be processed. We provide a realistic use case of a secure distributed survey.

For future work, we intend to explore different cryptosystems (e.g., lattice-based homomorphic encryption) suitable for decentralized trust, which would enable more efficient computations, more flexible and sophisticated queries, and lower storage overhead.

Acknowledgments

We would like to thank all of those who reviewed the manuscript or somehow participated in the development of this solution, in particular: Juan Troncoso-Pastoriza and the DeDiS team from EPFL.

References

[1] Bluekrypt, cryptographic key length recommendation. https://www.keylength.com/fr/4/#Biblio4.

[2] DeDiS Research Lab at EPFL, advanced crypto library for the Go language. https://github.com/DeDiS/crypto.

[3] Dyadic security. https://www.dyadicsec.com/.

[4] General Data Protection Regulation. http://ec.europa.eu/justice/data-protection/international-transfers/index_en.htm.

[5] The Go Programming Language. https://golang.org.

[6] Mininet, An Instant Virtual Network. http://mininet.org.

[7] Unlynx experimental implementation. https://github.com/lca1/unlynx.

[8] What is the Future of Data Sharing? http://www8.gsb.columbia.edu/globalbrands/sites/globalbrands/files/images/The_Future_of_Data_Sharing_Columbia-Aimia_October_2015.pdf.

[9] B. Anandan and C. Clifton. Laplace noise generation for two-party computational differential privacy. In *13th Annual Conference on Privacy, Security and Trust (PST)*, pages 54–61, 2015.

[10] Dixie B. Baker, Jane Kaye, and Sharon F. Terry. Privacy, Fairness, and Respect for Individuals. *eGEMS (Generating Evidence & Methods to Improve Patient Outcomes)*, 4(2), 2016.

[11] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway. Efficient Garbling from a Fixed-Key Blockcipher. In *2013 IEEE Symposium on Security and Privacy (SP)*, pages 478–492, May 2013.

[12] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering 2*, pages 77–89, 2012.

[13] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, and Riivo Talviste. Students and taxes: a privacy-preserving study using secure computation. In *Proceedings on Privacy Enhancing Technologies 2016*, 2016.

[14] Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*, pages 192–206. Springer, 2008.

[15] Jan Camenisch, Rafik Chaabouni, and Abhi Shelat. Efficient protocols for set membership and range proofs. In *ASIACRYPT 2008*, pages 234–252, 2008.

[16] Jan Camenisch and Markus Stadler. Proof systems for general statements about discrete logarithms. *Technical Report*, (260), 1997.

[17] R. Chen, A. Reznichenko, P. Francis, and J. Gehrke. Statistical queries over distributed private user data. In *NSDI. Vol. 12*, 2012.

[18] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science*, pages 383–395. IEEE, 1985.

[19] Tulio de Souza, Joss Wright, Piers O'Hanlon, and Ian Brown. Set difference attacks in wireless sensor networks. *International Conference on Security and Privacy in Communication Systems*, 2012.

[20] Xin Dong, Jiadi Yu, Yuan Luo, Yingying Chen, Guangtao Xue, and Minglu Li. Achieving an effective, scalable and privacy-preserving data sharing service in cloud computing. *Computers & security*, 42:151–164, 2014.

[21] Yitao Duan, John Canny, and Justin Zhan. Efficient privacy-preserving association rule mining: P4P style. In *Symposium on Computational Intelligence and Data Mining*, pages 654–660. IEEE, 2007.

[22] C. Dwork, K. Kenthapadi, F. McSherry, I. Mironov, and M. Naor. Our data, ourselves: Privacy via distributed noise generation. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 486–503. Springer Berlin Heidelberg, 2006.

[23] Cynthia Dwork. Differential privacy. Venice, Italy, July 2006. Springer Verlag.

[24] Cynthia Dwork. A firm foundation for private data analysis. In *Communications of the ACM, 54(1)*, pages 86–95, 2011.

[25] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference*, pages 265–284. Springer, 2006.

[26] Benjamin Fabian, Tatiana Ermakova, and Philipp Junghanns. Collaborative and secure sharing of healthcare data in multi-clouds. *Information Systems*, 48:132–150, March 2015.

[27] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[28] Chang Liu, Xiao Shaun Wang, K. Nayak, Yan Huang, and E. Shi. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 359–376, May 2015.

[29] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*, pages 605–622, May 2015.

[30] Xuefeng Liu, Yuqing Zhang, Boyang Wang, and Jingbo Yan. Mona: secure multi-owner data sharing for dynamic groups in the cloud. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1182–1191, 2013.

[31] N. Mohammed, D. Alhadidi, BCM. Fung, and M. Debbabi. Secure two-party differentially private data release for vertically partitioned data. In *IEEE Trans Dependable Secur Comput 11*, pages 59–71, 2014.

[32] A. Narayan and A. Haeberlen. Djoin: Differentially private join queries over distributed databases. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, OSDI'12*, pages 149–162, 2012.

[33] K. Nayak, X. S. Wang, S. Ioannidis, U. Weinsberg, N. Taft, and E. Shi. GraphSC: Parallel Secure Computation Made Easy. In *2015 IEEE Symposium on Security and Privacy (SP)*, pages 377–394, May 2015.

[34] C Andrew Neff. Verifiable mixing (shuffling) of ElGamal pairs (2004).

[35] C Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *Proceedings ACM-CCS 2001*, pages 116–125, 2001.

[36] Wee Siong Ng, Beng Chin Ooi, Kian-Lee Tan, and Aoying Zhou. PeerDB: A P2P-based system for distributed data sharing. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 633–644. IEEE, 2003.

[37] Olga Ohrimenko, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Markulf Kohlweiss, and Divya Sharma. Observing and Preventing Leakage in MapReduce. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1570–1581, 2015.

[38] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious multi-party machine learning on trusted processors. In *25th USENIX Security Symposium (USENIX Security 16)*, 2016.

[39] Raluca Ada Popa, Catherine Redfield, Nickolai Zeldovich, and Hari Balakrishnan. CryptDB: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 85–100. ACM, 2011.

[40] A. Rastogi, M. A. Hammer, and M. Hicks. Wysteria: A Programming Language for Generic, Mixed-Mode Multiparty Computations. In *2014 IEEE Symposium on Security and Privacy*, pages 655–670, May 2014.

[41] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54, May 2015.

[42] E. M. Songhori, S. U. Hussain, A. R. Sadeghi, T. Schneider, and F. Koushanfar. TinyGarble: Highly Compressed and Scalable Sequential Garbled Circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428, May 2015.

[43] L Sweeney. k-anonymity: A Model for Protecting Privacy. *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, 10(5):557–570, 2002.

[44] Ewa Syta, Iulia Tamas, Dylan Visher, David Isaac Wolinsky, Philipp Jovanovic, Linus Gasser, Nicolas Gailly, Ismail Khoffi, and Bryan Ford. Keeping Authorities" Honest or Bust" with Decentralized Witness Cosigning. *arXiv preprint arXiv:1503.08768*, 2015.

[45] U.S. Department of Health and Human Services . Breach portal: Notice to the secretary of hhs breach of unsecured protected health information. https://ocrportal.hhs.gov/ocr/breach/breach_report.jsf. Last Accessed: June 18, 2017.

[46] David I Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Scalable anonymous group communication in the anytrust model. In *5th European Workshop on System Security*, 2012.

[47] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[48] Min Yang and Yuanyuan Yang. An efficient hybrid peer-to-peer system for distributed data sharing. *IEEE Transactions on computers*, 59(9):1158–1171, 2010.

[49] Mahdi Zamani, Mahnush Movahedi, and Jared Saia. Millions of millionaires: Multiparty computation in large networks. *IACR Cryptology ePrint Archive*, 2014:149, 2014.

[50] Ning Zhang, Ming Li, and Wenjing Lou. Distributed data mining with differential privacy. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5. IEEE, 2011.

A Zero-Knowledge Proofs

A.1 Distributed Deterministic-Tag

We recall the distributed deterministic-tag sub-protocol below.

This sub-protocol is made of two consecutive rounds. It starts with $E_K(x) = (C_1,\ C_2) = (rB,\ x + rK)$, the ciphertext tuple corresponding to an ElGamal encryption of message x under the CA's public key K.

In the first round, each server sequentially generates a fresh secret s_i and adds the value derived from its secret s_iB to C_2. This eliminates the possibility to have a deterministic tag of 0 as an output of the protocol when the input message is zero. After this first round, the encrypted message is $(C_1, C_2) = (rB, x + rK + \sum_{i=1}^{m} s_iB)$. Let $(\tilde{C}_{1,0},\ \tilde{C}_{2,0}) = (C_1,\ C_2)$ be a ciphertext resulting from the first round.

In the second round, each server partially and sequentially modifies this ciphertext. More specifically, when server S_i receives the modified ciphertext $(\tilde{C}_{1,i-1},\ \tilde{C}_{2,i-1})$ from server S_{i-1}, it computes $(\tilde{C}_{1,i},\ \tilde{C}_{2,i})$ as

$$\tilde{C}_{1,i} = s_i\tilde{C}_{1,i-1} \tag{13}$$

and

$$\tilde{C}_{2,i} = s_i\left(\tilde{C}_{2,i-1} - \tilde{C}_{1,i-1}k_i\right) \tag{14}$$

Once all of these computations are done, we discard the first component $\tilde{C}_{1,m}$ and obtain

$$\tilde{C}_{2,m} = sx + \sum_{i=1}^{m} s_i sB \tag{15}$$

where $s = \prod_{i=1}^{m} s_i$ is a short-term collective secret corresponding to the product of each server's fresh secret. $\tilde{C}_{2,m}$ is the deterministic tag collectively computed from the original ciphertext (C_1, C_2).

Each time a server does the computations in the first round and in Equations (13) and (14), it must also compute a zero-knowledge proof to prove that the computations have been done correctly. In this case, when adding a secret value and when computing $(\tilde{C}_{1,i},\ \tilde{C}_{2,i})$, server S_i is the prover and anybody can act as a verifier. In the first round, the prover proves that he knows s_i the discrete logarithm of s_iB. Coming back to Equation (1) in Section 2.3, it is easy to see that for Equation (14), $y_1 = s_i,\ y_2 = k_is_i$ are the discrete logarithms of s_iB and $k_is_iB = s_iK_i$, respectively. The points s_iK_i, s_iB, $A = \tilde{C}_{2,i}$, $A_1 = \tilde{C}_{2,i-1}$ and $A_2 = -\tilde{C}_{1,i-1}$ on $\mathcal{G}$ are public and are part of the proof. The publication of s_iB also guarantees that server S_i has used the same secret s_i for all data during a given query. This means that for each query, server S_i will pick a value s_i that will be used throughout the query and will be different for the next query.

For Equation (13), $y_1 = s_i$ is the discrete logarithm of $s_i\tilde{C}_{1,i-1}$. The points $\tilde{C}_{1,i}$, s_iB, $A = \tilde{C}_{1,i}$ and $A_1 = \tilde{C}_{1,i-1}$on $\mathcal{G}$ are public and are part of the proof.

A.2 Key Switch

We recall the key switch sub-protocol below.

We start with $\mathrm{E}_K(x) = (C_1,\ C_2) = (rB,\ x + rK)$, a ciphertext tuple corresponding to the ElGamal encryption of message x under the CA's public key K. Let $(\tilde{C}_{1,0},\ \tilde{C}_{2,0}) = (0,\ C_{2,j})$ be the initial modified ciphertext tuple. Each server will partially and sequentially modify this element. Specifically, when server S_i receives $(\tilde{C}_{1,i-1},\ \tilde{C}_{2,i-1})$ from server S_{i-1}, it generates a fresh random nonce v_i and computes $(\tilde{C}_{1,i},\ \tilde{C}_{2,i})$ as

$$\tilde{C}_{1,i} = \tilde{C}_{1,i-1} + v_iB \tag{16}$$

and

$$\begin{aligned}\tilde{C}_{2,i} &= \tilde{C}_{2,i-1} - (r_jB)k_i + v_iU \\ &= \tilde{C}_{2,i-1} - r_jK_i + v_iU.\end{aligned} \tag{17}$$

where $v=v_1+...+v_m$.

Each time a server does the computations in Equations (16) and (17), it must also compute a zero-knowledge proof to prove that the computations have been done correctly. Again, at each step i, server S_i is the prover and anybody can be the verifier. Coming back to Equation (1) in Section 2.3, it is easy to see that for Equation (17), $y_1=k_i$, $y_2=v_i$ are the discrete logarithms of $k_iB=K_i$ and v_iB, respectively. All points K_i, v_iB, $A=\tilde{C}_{2,i}-\tilde{C}_{2,i-1}$, $A_1=-r_jB$ and $A_2=U$ are made public and do not leak any information about underlying secrets.

For Equation (16), $y_1=v_i$ is the discrete logarithm of v_iB. All points v_iB, $A=\tilde{C}_{1,i}-\tilde{C}_{1,i-1}$ and $A_1=U$ are made public and do not leak any information about underlying secrets.

A.3 Dynamic Collective Authority

We recall the sub-protocol allowing to add/remove a server from the collective authority.

Let S_m be the server that needs to be added. Any data provider that stored data encrypted using the CA's previous public key K_{prev} must execute the corresponding sub-protocol in order to have its data encrypted under the CA's new public key K_{new}. When adding a new server S_m to the collective authority $S_1,...,S_{m-1}$, $K_{\text{prev}}=K_1+...+K_{m-1}$ and $K_{\text{new}}=K_1+...+K_m$. Starting from a message x encrypted under K_{prev}, $(C_1, C_2)=(rB, x+rK_{\text{prev}})$, server S_m multiplies C_1 by its private key k_m

$$C_1k_m=(rB)k_m=rK_m \tag{18}$$

and adds the result to C_2

$$\tilde{C}_2=C_2+rK_m=x+rK_{\text{prev}}+rK_m=x+rK_{\text{new}}. \tag{19}$$

Component $\tilde{C}_1$ remains the same, i.e.,

$$\tilde{C}_1=C_1.$$

Coming back to Equation (1) in Section 2.3, we see that for Equation (19), $y_1=k_m$ is the discrete logarithm of $k_mB=K_m$. All points K_m, $A=\tilde{C}_2-C_2$ and $A_1=C_1$ are made public and do not leak any information about underlying secrets.

Now, assume a collective authority of m servers $S_1,...,S_m$ and let S_m be the server that needs to be removed. In this case, $K_{\text{prev}}=K_1+...+K_m$ and $K_{\text{new}}=K_1+...+K_{m-1}$. In order to update the encryption of message x to the CA's new public key K_{new}, server S_m must compute $\tilde{C}_2$ as

$$\tilde{C}_2=C_2-rK_m=x+rK_{\text{prev}}-rK_m=x+rK_{\text{new}}. \tag{20}$$

Again, using Equation (1) in Section 2.3, we see that for Equation (20), $y_1=k_m$ is the discrete logarithm of $k_mB=K_m$. All points K_m, $A=\tilde{C}_2-C_2$ and $A_1=-C_1$ are made public and do not leak any information about underlying secrets. For Equation (18), $y_1=k_m$ is the discrete logarithm of k_mrB and the point k_mrB is public and do not leak any information about underlying secrets.

B SELECT∗ query

UnLynx's design enables the system to respond to queries of the form 'SELECT ∗'. The system would handle this query by executing all the steps of the Decentralized Data Sharing protocol 4.1 except Steps 5 and 6 that would be skipped.

While this can be useful for a data provider wishing to retrieve/decrypt parts/all of his database, UnLynx should not allow these queries on a distributed database held by multiple data providers. In fact, it is not possible to answer this request while preserving the privacy and data confidentiality of data providers because differential privacy cannot be ensured on non-aggregated data. Moreover, an external querier would have access to DP's raw data. Hence, our system should only allow this operation for a data provider querying his own database.