IN, LCA1: Decentralized Data Sharing System based on Secure Multi-Party Computation

Due on Autumn 2017

D.Froelicher, J.Troncoso-Pastoriza

Max Premi

January 29, 2018

Abstract

Current solutions for privacy-preserving data sharing among multiple parties either rely on legal agreements, secure multiparty computation, differential privacy or homomorphic encryption. Legal contracts usually ensure the accountability of the involved parties but do not protect the privacy of the data. In this report, we study two existing systems, namely Unlynx and Prio.
They are two privacy-preserving data sharing systems, each with its own way to encode/encrypt, decode/decrypt and aggregate data. While Unlynx [1] uses homomorphic encryption based on Elliptic Curve ElGamal and zero-knowledge proofs, Prio [2] uses Secret-sharing encoding and *secret-shared non-interactive proofs*(SNIP's) to validate the data, which are claimed to perform better than classic zero-knowledge proofs [20, 21] in term of computation time, by relying on a time/bandwidth trade-off.
In this project, we compare both Unlynx and Prio in order to find a solution allying the best of both protocols.
The protocols pros and cons are the following:
First, Unlynx's collective authority is almost static and it is needed to trust at least one server to ensure correctness and privacy. However, Unlynx also requires the data to be stored encrypted under a collective key at the data providers, and supports few aggregation functions, i,e, sum and count.
In the contrary, Prio extends classical private aggregation functions, with the addition of sets union/intersection or the support of least-square regression. It also does not use homomorphic encryption and leaves the data storage and protection to the DPs, as it can process input data decomposed in shares. Similarly to Unlynx, privacy is assured if at least one server is trusted. Yet, in order to ensure correctness, it is needed to trust all servers and the collective authority is non-static.

The goal of this project is to :

- First implement Prio in the Unlynx framework.
- Second to implement input validation [3] adapted to be non-interactive, to work in the threat model of UnLynx (Anytrust) and doable with Elliptic Curve ElGamal encryption used in Unlynx.
- Then modify the protocols' settings to run with the least significant difference in term of assumption and model.This will enable us to compare the protocols in a fair way. We compare in section 7, Prio against UnLynx and against UnLynx with Prio threat model.
- Finally, design a system that implements the best of both privacy-preserving data sharing protocols. The idea would be to combine Prio flexibility in terms of computations with the privacy and security ensured by Unlynx.

Contents

1 Introduction

Nowadays, tons of data are generated by individuals about what they do and are collected and used to compute statistics, by different parties. Even if these statistics are gathered with the goal of learning useful aggregate information about the users/population, for example, health or work statistics for a country [7], it might end in collecting and storing private data from data providers, which poses a serious privacy and security problem.
We can illustrate this example with the numerous problems of Cloud leaks/breaches that happened several times in the past years [5], or the divulgation of sensitive health data such as susceptibility to contract diseases, that can be used against one individual [17]. They might even be disclosed, sold for profit [11] or be used by agencies for targeting and mass surveillance goals. Indeed some countries do not have highly regulated data privacy laws, e.g., the U.S [18], meaning data collectors are less restricted on the use of these data.
The need to collect data and to share them in a privacy-preserving way has become crucial in this context, and a lot of research has been done on this topic.
The techniques are developed through research efforts at prominent research groups [1, 2], and some companies have already implemented some of these mechanisms in their commercial products [6].
Developed techniques for privacy-preserving data sharing among multiple parties either rely on legal agreements, secure multiparty computation (MPC), differential privacy or homomorphic encryption. Legal agreements is not a robust solution because they ensure the accountability of the involved parties but do not protect the privacy of the data. Homomorphic encryption is an effective technique to preserve privacy, but typically does not provide verifiable computing, and performance is also often a disadvantage as ciphertexts are larger than plaintexts.
Secure MPC is a technique to counter attacks coming from inside the system but can be difficult to scale two more than two or three parties depending on the techniques that are used and typically does not provide verifiable computing.
Differential privacy is a technique developed to ensure stronger privacy in queries, but it does not support all aggregation function, i.e, it works well for count but not for max.
Techniques usually rely on a combination of those techniques to overcome the singular weaknesses and create a better system. Already implemented systems, such as the one proposed by Ruichuan et al, [27] combine homomorphic encryption, differential privacy, and input validation but can only aggregate and require an honest-party which is a centralized single point of failure. Another example system, SCMCQL by Bater et al. [28] does all operations thanks to garbled circuit but is a two-party protocol which does not scale and the threat model, honest-but-curious, is not the ideal model to perform data sharing in a real-world context.
Even if fewer distributed systems have been deployed to the real world, it is a rising and interesting subject. We can also illustrate the growth of decentralized systems with the rise of Cryptocurrencies such as Bitcoin [19]. Prio and Unlynx are both distributed protocols following the same decentralized trust trend.
Unlynx ensures confidentiality, privacy, a threat model fitted for real-world application and can handle differential privacy. Prio is one of the few MPC system that ensures correctness, scalability and good performances.
In this paper, we present the implementation of Prio into Unlynx's framework, an implementation of an improved Unlynx that includes input validation and a theoretical comparison and discussion for future possible new systems that combine, if possible, the best part of each approach.

2 Contributions

In this paper, the following contributions are made:
- An implementation of Prio Secred Shared Non-Interactive Proofs (SNIPs) system in Unlynx's framework, represented as two new protocols, and a new service. It includes the input validation and the aggregation.

It also contains the different data types supported by Prio.

- The implementation of a novel proof for input range validation for Elliptic Curve ElGamal, using pairings on a specific Elliptic Curve. This allows the server to exclude faulty data sent by possible malicious clients. The range to check is $[0, u^l]$ with u, l taking arbitrary values.

- An evaluation of both protocols in terms of privacy and efficiency, with a comparison with the most similar settings. Then, we provide some guidelines on how to combine the advantages of both systems and a description of future work.

3 Background

This section introduces some fundamental concepts used throughout the rest of the report. *Collective Authority* that is the base of both system functionality. *ElGamal encryption* is used in Unlynx to ensure privacy, while *arithmetic circuits* are used by Prio to ensure correctness of data.
Beaver's triple are used in Prio's multi-party computation part, and *Affine-aggregatable functions* are used to enable different aggregation functions. Finally, Bilinear pairing over Elliptic Curve is used in Unlynx's input range validation.

3.1 Collective Authority

Nowadays, applications and systems rely on third-party authorities to provide security services. For example the creation of certificates to prove ownership of a public key. A collective authority is a set of m servers that are deployed in a decentralized and distributed way, to support a given number of protocols.
Each of them possesses a private-public key pair (k_i, K_i), where $K_i = k_i B$ with k_i a scalar and K_i a point in a given Elliptic Curve. This authority constructs a public key $K = \sum_{i=1}^{m} K_i$ which is the sum of all the server's public keys. To decrypt a message, each server i partially decrypts a message encrypted using k_i. Thus the collective authority key provides strongest link security, as no intermediate can decrypt the data without the contribution of all the servers.

3.2 ElGamal Encryption

All the involved scalars belong to a field $\mathbb{Z}_p$.
For Unlynx, data are encrypted using Elliptic Curve ElGamal, more precisely, P is a public key, x is a message mapped to a point and B is a base point on the curve γ. The encryption is the following, with r a random nonce:
$E_P(x) = (rB, x+rP)$. The additive homomorphic property states that $\alpha E_P(x_1)+\beta E_P(x_2) = E_P(\alpha x_1+\beta x_2)$
To decrypt, the owner of the private key p satisfying $P = pB$ multiplies rB and p to get rP and substracts it from $x + rP$ to recover x.

3.3 Arithmetic Circuits

An arithmetic circuit C over a finite field $\mathbb{F}$ takes as input a vector $x = (x^{(1)}, ..., x^{(L)}) \in \mathbb{F}^L$. It is represented as an acyclic graph with each vertex either being an *input*, *output* or a *gate*.
There are only two types of gates, addition and multiplication, all in finite field $\mathbb{F}$.
A circuit C is just a mapping $\mathbb{F}^L \to \mathbb{F}$, as evaluating is a walk through the circuit from inputs to outputs.

3.4 Beaver's MPC triples

A Beaver triple is defined as follows:
$(a, b, c) \in \mathbb{F}^3$, chosen at random with the constraint that $a \cdot b = c$.
As we use it in a multiparty computation context, each server i holds a share $([a]_i, [b]_i, [c]_i) \in \mathbb{F}^3$.
Using these shares, the goal is to multiply two numbers x and y without leaking anything about them. In Prio the goal is to multiply shares $[x]_i$ and $[y]_i$.
To do so, the following values are computed:

$$[d]_i = [x]_i - [a]_i \quad ; \quad [e]_i = [y]_i - [b]_i$$

Then from this shares, each server can compute d and e and compute this formula:

$$\sigma_i = de \cdot m^{-1} + d[b]_i + e[a]_i + [c]_i$$

The sum of these shares yields:

$$\begin{aligned} &\sum_i \sigma_i = \sum_i (de/m + d[b]_i + e[a]_i + [c]_i) \\ &= de + db + ea + c \\ &= (x-a)(y-b) + (x-a)b + (y-b)a + c \\ &= (x-a)y + (y-b)a + c \\ &= xy - ab + c \\ &= xy \end{aligned} \tag{1}$$

As $\sum_i \sigma_i = xy$, it implies that $\sigma_i = [xy]_i$

3.5 Affine-aggregatable encodings (AFEs) functions

Given the fact that each data provider i holds a value $x_i \in D$, and the servers have an aggregation function $f : D^n \to A$, whose range is a set of aggregates A, an AFE gives an efficient way to encode data such that it is possible to compute the value of the aggregation function $f(x_1, ..., x_n)$ given only the *sum of the encodings* $x_1, ..., x_n$. This technique enables the computation of min, max, set intersection and other functions by doing only aggregations at the servers.
It comprises three Algorithm (Encode, Valid, Decode) defined in a field $\mathbb{F}$ and two integers $k' \leq k$

- **Encode(x)**: maps an input $x \in D$ to its encoding in $\mathbb{F}^k$
- **Valid(y)**: returns true if and only if y is a valid encoding of some item in D
- **Decode(σ)**: $\sigma = \sum_{i=1}^{n} Trunc_{k'}(Encode(x_i)) \in \mathbb{F}^{k'}$ is the input (Trunc takes the $k' \leq k$ components of the encoding), and it outputs the aggregation result $f(x_1, ..., x_n)$.

3.6 Bilinear pairings over Elliptic Curve

Let G_1 and G_T be two additive groups of points of an elliptic curve G over a field $\mathbb{F}$ of order n and with identity O. Then the mapping $e : G_1 \times G_1 \to G_T$ satisfies the following conditions:
For all $R, S \in G_1$ and $x \in \mathbb{F}$, $e(R, S)(x) = e(Rx, S) = e(R, Sx)$
For B the base point, $e(B, B) \neq O$, and the mapping is efficiently computable.

4 Unlynx System

This section presents Unlynx [1] system in general. It goes through the model design, the assumptions made about the parties taking part in the general protocol, and a detailed pipeline of which protocols are executed to achieve the goal of the system. It also describes briefly describes the proofs done at each step.

4.1 System Model

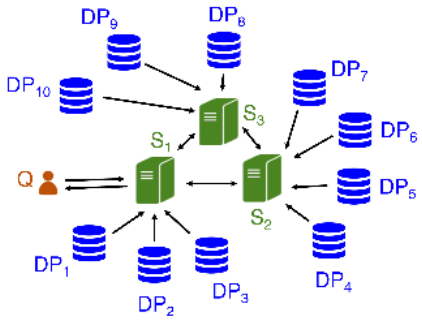


Figure 1: **Data providers (in blue),collective-authority servers (in green) and querier (in red). In this example m=3 and n=10**

Unlynx is a privacy-preserving data sharing system developed by LCA1 [8] in collaboration with DeDiS [9]. It consists of a collective authority (CA) formed by m servers $S_1, ..., S_m$, and n data providers $DP_1, ...DP_n$ containing sensitive data, encrypted using EC ElGamal by following the scheme described section 3. These DPs combined represent a distributed database that is used to answer queries made by a querier Q. The querier and DPs choose one server of the CA to communicate with and can change this choice at any given time.
Functionality: Unlynx should permit SQL queries of the form SELECT SUM(*)/COUNT(*) FROM DP,.... WHERE * AND/OR GROUP BY *, with any number of * clauses, but only equality ones.
Privacy and Robustness: Both are assured if at least one server is trusted, as we use the fact that it is allowed to publish ciphertexts and their aggregation to show that the computation at the servers is actually correct. It leaks nothing as all data are encrypted.
Also, the data are never decrypted and a key switch protocol is used to sequentially change the encryption key from the CA's public one to the querier's Q public one. This way, the privacy of data is ensured, as to break the protocol, all servers in the CA would have to collude, which is not possible with the actual threat model, described in the next section.

4.2 Threat Model

Collective authority servers It is assumed an Anytrust model [14]. It does not require any particular server to be trusted or to be honest-but-curious. Whenever one of the servers is not malicious, functionality, security, and privacy are guaranteed.
Data providers are assumed to be honest-but-curious. The system does not protect against malicious DPs sending false information, but a solution will be discussed in section 4.4.
Queriers are assumed to be malicious, and can collude between themselves or with a subset of the CA

servers.
It is also assumed that all network communication is encrypted and authenticated, by using a protocol like TLS for example.

4.3 Pipeline and proof

The protocol starts when a querier wants to retrieve some information about sensitive data. It sends the query to one of the servers of the CA. Upon receiving the request, the server broadcasts this query to the other servers in the collective authority.
From here the data are privately and securely processed by the CA, before sending back the result to the querier, encrypted under the public key of the querier. During all the steps of the protocol, the servers will never see the data in clear.
The pipeline comprises the following secure protocols: Encryption, Verifiable Shuffle, Distributed Deterministic Tag, Collective Aggregation, Distributed Results Obfuscation and finally Key Switch. At the end of this pipeline, the querier gets the data and can decrypt them to get the aggregate statistics he asked for, without any server seeing the data in clear, or knowing from which data provider the data are from.
The steps of the protocol are not detailed in this report, but some of them are used for comparison with Prio and are discussed in section 6.

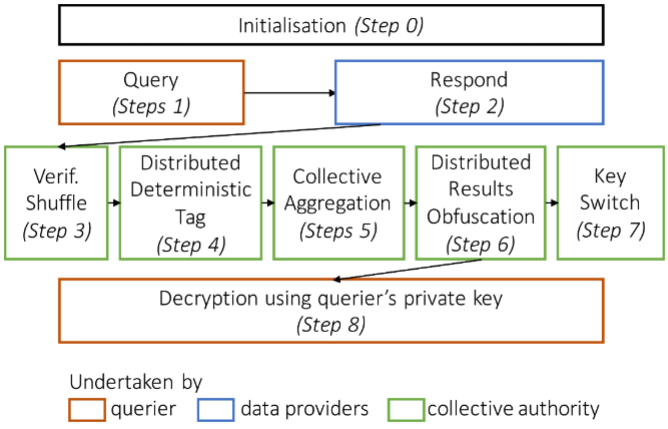


Figure 2: Unlynx query processing pipeline

Proofs: Zero-knowledge proofs are used to validate the correctness of inputs and computations without disclosing the involved secrets and private inputs.

To illustrate one of them, while doing the aggregation phase, the servers publish the ciphertexts and the result of their aggregation. As the data are encrypted using Elliptic curve ElGamal, it leaks nothing about them.
However, one of the problems in Unlynx is that it provides no input range proof for data coming from the data providers, meaning the DPs can send false, forged or incorrect data, e.g., very big or very small quantities, and as data are encrypted the servers have no way to directly check the actual value sent. This makes the whole result and computation invalid if we assume malicious DPs, and this is why in the basic threat model, data providers are assumed honest-but-curious. An input validation proof is implemented and described in the following section. It is an adaptation, completely novel, developed for the project, taken from the references [3], and adapted to be non-interactive and usable with Elliptic Curve ElGamal encryption.

4.4 Input range validation for Elliptic Curve ElGamal

A problem with data encryption is that we can not be sure if received data are correct. An example would be an aggregation where input values should be in the range $[0, 100)$. In the current system's settings, if a malicious data provider wants to send a value of 15000 to falsify information, it can do it. This section presents an interactive algorithm based on a classic ElGamal input range validation [3], and we discuss its novel non-interactive form in section 6.
We need to define some parameters before jumping into the algorithm. A scalar in the elliptic curve is defined over the field $\mathbb{Z}_p$. Also, we denote $e()$ the bilinear map that is described in section 1.
This is the algorithm that allows checking the validity of a secret σ in a range $[0, u^l]$. The algorithm can be adapted to check any range $[a, b]$.

Algorithm 1 Interactive Range Validation

In the algorithm, the terms *Prover* and *Verifier* are used. In our case it should be clear that the prover is the data provider and the verifier the CA containing all the servers.

Common Input: B Elliptic Curve base, P a public key, u, l , 2 integers and C commitment.
Prover Input: σ, the value to encode mapped to a point in the EC and r a scalar in the EC such that $C = \sigma + Pr$, with $\sigma \in [0, u^l)$

$\mathbf{P} \leftarrow \mathbf{V}$: verifiers pick $x_i \in \mathbb{Z}_p$ define $y_i \leftarrow Bx_i$
and send to the prover $A_{i,j} \leftarrow B(x_i + j)^{-1}\ \forall j \in \mathbb{Z}_u$ and $i \in \{0, ..., m\}$

$\mathbf{P} \rightarrow \mathbf{V}$ Then prover encodes the signature of the value to check in base u with randomly picked v_j.
So $\forall j \in \mathbb{Z}_l$ such that $\sigma = \sum_j \sigma_j u^j$, it picks $v_j \in \mathbb{Z}_p$ and sends $V_{i,j} = A_{i,\sigma_j} v_j$ back to server i

$\mathbf{P} \rightarrow \mathbf{V}$: prover picks 3 values $s_j, t_j, m_j \in \mathbb{Z}_p$, $\forall j \in \mathbb{Z}_l$ and sends to each server i:
$a_{i,j} \leftarrow e(V_{i,j}, B)(-s_j) + e(B, B)(t_j)$
$D \leftarrow \sum_j (u^j s_j + P m_j)$

$\mathbf{P} \leftarrow \mathbf{V}$ Verifier sends a random challenge $c \in \mathbb{Z}_p$

$\mathbf{P} \rightarrow \mathbf{V}$ Prover sends the following value for Verifiers to compute verification.
$\forall j \in \mathbb{Z}_l$, $Z_{\sigma_j} \leftarrow s_j - \sigma_j c$ and $Z_{v_j} \leftarrow t_j - v_j c$
$Z_r = m - rc$, where $m = \sum_j m_j$

Verifier i checks that $D = Cc + PZ_r + \sum_j (Bu^j Z_{\sigma_j})$
$a_{i,j} = e(V_{i,j}, y)c + e(V_{i,j}, B)(-Z_{\sigma_j}) + e(B, B)(Z_{v_j})$, $\forall j \in \mathbb{Z}_l$

This algorithm has been adapted from the paper on Set membership over ElGamal [3], used with classic ElGamal encryption. The prover needs to compute $5l$ point multiplications in the protocol.
The completness follows from inspection, while soundness follows from the unforgeability of the Boneh-Boyen signature [22].
In addition, a zero-knowledge proof must satisfy another property in addition to completeness and soundness. It is that if the statement is true, no verifier learns anything else than the fact that the statement is true. This can be showed by a *simulator*, that given only the statement to be proved and no access to the prover, it can produce a transcript that "looks like" an interaction between a prover and the cheating verifier.
The simulator *Sim* is constructed as follows for a verifier V:

1. *Sim* retrieves $y, \{A_i\}$ from V with $i \in \mathbb{Z}_u$.
2. *Sim* chooses $\sigma \in [0, u^l)$, $v_j \in \mathbb{Z}_p$ and sends $V_j \leftarrow A_{\sigma_j} v_j$, for each σ_j such that $\sigma = \sum_j \sigma_j u^j$ with $j \in \mathbb{Z}_l$
3. *Sim* chooses $s_j, t_j, m_j \in \mathbb{Z}_p$ for each V_j and sends $a_j \leftarrow e(V_j, B)(-s_j) + e(B, B)(t_j)$ for each j and $D \leftarrow \sum_j (u^j s_j + P m_j)$
4. *Sim* receives c from V.
5. Eventually *Sim* computes for each j, $Z_{\sigma_j} = s_j - \sigma_j c$, $Z_{v_j} = t_j - v_j c$ and also $Z_r = m - rc$, where m is the sum of m_j, and sends them to V.

Arbitrary Range :

To handle an arbitrary range $[a, b]$, the prover needs to show that $\sigma \in [a, a + u^l]$ AND $\sigma \in [b - u^l, b]$, this leads to the following formula:

$$\sigma \in [b - u^l, b] \iff \sigma - b + u^l \in [0, u^l)$$
$$\sigma \in [a, a + u^l] \iff \sigma - a \in [0, u^l)$$

The only needed modification in the algorithm is the verifier's check which is now:

$$D = Cc + B(-B + u^l) + P(Z_r) + \sum_j B(Z_{\sigma_j})$$
$$D = Cc + B(-A) + P(Z_r) + \sum_j B(Z_{\sigma_j})$$

In the original paper [3], it is also discussed set membership that can be more efficient if the range is really small, for example checking that people are in the range of age [18-25] for delivering a discount. In this case, the protocol is simpler as range validation is just a special case of set membership. The difference is that the prover sends back only one obfuscated value which is the value supposedly contained in the set.

5 Prio System

This section presents Prio system in general, the same way we did for Unlynx. It goes through the model design, the assumptions made about the parties taking part in the general protocol, and a detailed pipeline of the executed protocols to achieve the goal of the system. It also describes more precisely how the Input range validation works for Prio.

5.1 System Model

Prio [2] is also a privacy-preserving data sharing system developed at Stanford University.
It consists of a collective authority (CA) formed by a number m of servers $S_1, ...S_m$, and each data provider holds a private value x_i that is sensitive. Unlike Unlynx, Prio does not encrypt private value x_i that is why it is a more challenging aggregation in terms of privacy.
Prio is based on the splitting of each data x in m shares such that $\sum_{k=1}^{m} x_k = x$ in a defined finite field $\mathbb{F}$ i.e., modulo a prime p. This encoding helps to keep privacy, as getting $m - 1$ shares does not leak anything about x itself.
Communication is assumed to be done in secure channels as previously described for Unlynx.
Functionality: Prio should permit the collective authority to compute some aggregation function $f(x_1, ...x_n)$ over private values of data providers, in a way that leaks as little as possible about these, except what can

be inferred from the aggregation itself.
It is also possible to gather more complex statistics such as variance, standard deviation, frequency count or even sets intersections/unions. All the aforementioned functions are computed from an encoding called AFE (section Background), that helps computing function with only the **sum** of the encodings. So the collective authority always computes the sum no matter what function is asked.
Privacy and Robustness: Privacy is assured if at least one server is trusted, but robustness is satisfied if and only if all servers are trusted, as we cannot be assured that computation at the server is correct.

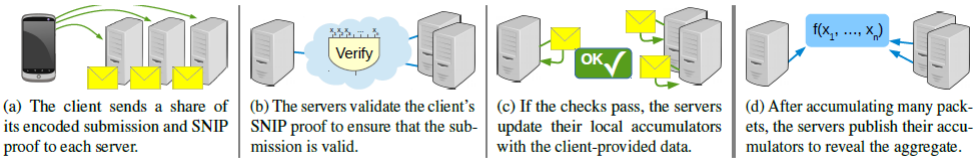


Figure 3: Prio's SNIP pipeline

5.2 Threat Model

Collective authority servers: In Prio, it is needed that all servers are not malicious and trusted so that security and privacy are guaranteed.
Robustness against malicious server seems desirable but that would impose an overhead in terms of cost privacy and performance degradation, which is not desirable.
Indeed if the sytem protects *robustness* in the presence of k faulty servers, it can protect *privacy* only against a coalition of at most $s-k-1$ malicious servers. This is because $s-k$ honest server must be able to produce a correct aggregattion function even if k faulty servers are offline. This is describe more formally in Prio original paper [2], but increasing robustness weakens privacy in these conditions.
Data providers are assumed to be malicious. The system protects itself against malicious DPs. All data that do not pass the SNIPs proof, will be discarded.

5.3 Pipeline and proof

Prio's pipeline is a little different than Unlynx, as first the proof is ran on the data when they arrived and if they pass the proof, they are stored and aggregated later with more data. The general pipeline is shown in Figure 3.
First, it is needed to define an arithmetic circuit, $Valid(.)$, for each data provider. When the data provider runs the circuit with its secret value x as input, and sends a share $[x]_i$ of the secret value to server i, such that $\sum_i [x]_i = x$ along with a share of the polynomial $[h]_i$ derived from the circuit. The output of $Valid(x)$ is 1.
This circuit is only defined in function of the number of bits of each share $[x]_i$ from the data provider, so it also sends a configuration file to the server such that it can reconstruct the circuit to verify the input. From this circuit, 3 polynomials are extracted f, g and h.

The data providers first upload their shares $[x_i]_1, ..., [x_i]_m$ of their respective private value and a share of polynomial h extracted from the arithmetic circuit.
The servers verify the SNIPs provided by the data providers to jointly confirm that $Valid(x_i) = 1$. If it fails, the server discards the submission.
Then each server saves in an accumulator the data they need to aggregate and they all together run the collective aggregation on the verified data.

When received inputs from all the involved DPs, they each publish their aggregation result to yield the final aggregation (which is the sum of all aggregation) result.

5.4 Prio range validation (SNIPs)

In this section, we describe into more details the secret-shared non-interactive proofs(SNIP) protocol.
Assumption: A **Valid** circuit has M multiplication gates, we work over a field $\mathbb{F}$ such that $2M << |\mathbb{F}|$

5.4.1 Data provider evaluation

First the data provider (DP) evaluates the circuit **Valid** on its input x. It constructs three polynomials f, g and h which encode respectively the two input wires and the ouput wire of each of the M multiplication gates in the **Valid**(x) circuit.
This step is done by polynomial interpolation to construct f, g and get $h = f \cdot g$.
So polynomials f, g have a degree at most $M-1$ while $h = f \cdot g$ have a degree at most $2M-2$.
Then the DP splits the polynomial h using additive secret sharing and sends the ith ($[h]_i$) share to server i.

5.4.2 Consistency checking at the server

At this time each server i holds the shares $[x]_i$ and $[h]_i$ sent by the data provider. From both these values, the servers can reproduce $[f]_i$ and $[g]_i$ without communicating with each other.
Indeed $[x]_i$ is a share of the input, and $[h]_i$ contains a share of each wire value coming out of a multiplication gate. Thus, it can derive all other values via affine operations on the wire.

5.4.3 Polynomial identity test

Now each server has reconstructed shares $[\hat{f}]_i, [\hat{g}]_i$ from $[h]_i$ and $[x]_i$. It holds that $\hat{f} \cdot \hat{g} = h$ if and only if the servers collectively hold a set of wire values that, when summed is equal to the internal wire value of the **Valid**(x) circuit computation.
All execute the Schwartz-Zippel randomized polynomial identity test [23] to check if relation holds and no data have been corrupted or malicious DPs have tried to send wrong data.
The principle is that if $\hat{f} \cdot \hat{g} \neq h$, then the polinomial $\hat{f} \cdot \hat{g} - h$ is a non-zero polynomial of degree at most $2M-2$ in $\mathbb{F}$. It can have at most $2M-2$ zeros, so we choose a random $r \in \mathbb{F}$ and evaluate this polynomial, the servers will detect with probability at least $1 - \frac{2M-2}{|F|}$ that $\hat{f} \cdot \hat{g} \neq h$.
The servers can use a linear operation to get share $\sigma_i = [\hat{f}(r) \cdot \hat{g}(r) - h(r)]_i$. Then publish this σ_i to ensure that $\sum_i \sigma_i = 0 \in \mathbb{F}$. If it is not 0 the servers reject the client submission.

5.4.4 Multiplication of shares

Finally all servers need to multiply the shares $[\hat{f}(r)]_i$ and $[\hat{g}(r)]_i$ to get the share $[\hat{f}(r)]_i \cdot [\hat{g}(r)]_i$ without leaking anything to each other about the values of the two polynomials. It is here that the Beaver MPC triple enters the computation. Each server also received from a trusted dealer one-time-use shares $([a]_i, [b]_i, [c]_i) \in \mathbb{F}^3$ such that $a \cdot b = c \in \mathbb{F}$. We can from this shares, efficiently compute a multiparty multiplication of a pair of secret-shared values. This only requires each server to broadcast a single message.
The triple is generated by the data provider but it could be generated by the servers themselves by following an interactive secure protocol. There have been a lot of works in this topic during the last years [29]. This would avoid the dependency on the data provider to generate these values, which could lead to forging them to make the servers accept the test by designing α to cancel out the sum shown at the end of the paragraph. The servers are then able to compute if values are correct. Moreover, even if the data provider sends wrong values, the server still catches the cheating data provider with high probability. Indeed if $a \cdot b \neq c \in \mathbb{F}$ then

we can write $a \cdot b = (c + \alpha) \in \mathbb{F}$. We can then run the polynomial identity test with $\hat{f}(r) \cdot \hat{f}(r) - h(r) + \alpha = 0$. The servers will catch a wrong input with probability at least $1 - \frac{2M-2}{|F|}$.

5.4.5 Output verification

If all servers are honest, each will hold a set of shares of the **Valid** circuit's values. To confirm that **Valid(x)** is indeed 1 they only need to publish their share of the *output wire*. Then, all server sum up to confirm that it is indeed equal to 1, except with some small failure probability due to the polynomial identity test.

6 Implementation

This section details what was implemented more thoroughly. We include the description of the deployed code, but also of the theoretical work done to design some algorithms.

6.1 Prio implementation

First, we will address the implementation of Prio:

Prio code [15] was implemented in Go, with 3 dependencies in C (FLINT, GMP, and MPFR) to execute polynomial operations and designed by Corrigan-Gibbs et al.
This project contribution is mostly porting code to use the multiparty computation and SNIPs in the Unlynx framework. So most of the code is used directly from the repository [30], but all the communication between the servers and with data providers has been reworked to be compatible with the Tree structure used in Unlynx [16].
To follow the structure, the aggregation and verification protocols were split into two different protocols, even if they both work together to get the final result.
To be more precise, the data providers send shares $[x_i]$ and $[h_i]$, to the servers that do the SNIPs proof by evaluating the share on the *Valid()* circuit, do the MPC to fuse each validation of the shares , if it passes it aggregates the result of the SNIPs, else it will discard.
Let's start by describing the easiest protocol, the aggregation and then the validation.

Aggregation

The protocols are run at each server j, and each protocol has received a share $[x_i]_j$ from data provider i represented by an array *type big.Int* in Go. The protocol structure is a binary Tree. This share actually represents the encoding in AFE of the original value which was an integer.

When several data have been received, aggregation starts by the root node, that notifies all children that the aggregation will start, and wait for the children to send their local sum. This notification goes down the Tree until there are no more children, and at this point each leaf l aggregates locally the share by simply summing $\sum_i [x_i]_l$, and sends the result to their unique parents. On receiving the response from its children, the other nodes aggregate locally their own shares, and send to the parent. This is done until the root has received all the data. It then aggregates all the data and publishes them.
The only optimization made is the transfer of shares between servers. The structure transforms these shares into a byte array and uses the method of big int to set it directly back from byte array to a big integer. This is made because the transfer functions are more efficient with bytes.

However, Prio has a more interesting feature that helps aggregating different types of data and executing different aggregation functions. Indeed, one can represent the value to aggregate with an AFE, where the AFE

is an array of *big.Int*. This helps computing OR and AND, MIN and MAX and even set intersection/union given only the aggregation of the special AFE encoding for each for each of these functions.
To illustrate an AFE, let us see the OR. To do OR$(x_1, ...x_n)$ where $x_i \in \{0, 1\}$, x is encoded as follows in $\mathbb{F}_2^\lambda$ (for a λ-bit string):

$$Encode(x) = \begin{cases} \lambda \text{ zeros if x} = 0 \\ \text{Random element} \in \mathbb{F}_2^\lambda \text{ if x} = 1 \end{cases}$$

The Valid algorithm is always true as long as all encodings have the same size. To Decode, we output 0 if and only if the λ-bit string is composed of λ zeros. This AFE outputs the boolean private-OR of the values with probability $p = 1 - 2^{-\lambda}$, over the randomness of the encoding.
As seen previously, we can compute a function thanks to the sum of the AFE encoding. Hence, modifications only affect the input data and the way the final result is computed at the root (the Decode algorithm). No modification to the protocol has to be made in order to compute different statistics.

Verification

The verification protocol is the part where the Beaver's MPC triples are used. The data provider runs a function to create requests for the servers from 3 parameters: *Its secret data, the number of servers, and the index of the leader for the request.*
This will create a request for each server, by creating a **Valid** circuit, evaluating the secret on it and constructing the polynomial. Each request contains a share of the Beaver MPC triple $[a]_i, [b]_i$ and $[c]_i$, but more importantly, a share of the secret value $[x]_i$ and one of the polynomial created previously $[h]_i$.

The splitting is optimized by Prio, in a way that instead of picking the $s - 1$ shares randomly and setting the last one, it uses a pseudo-random generator (such as AES in counter mode). The data provider can then pick the keys and shares them instead of sharing the integer directly. Then the data provider sends to each server the request it is assigned to. From this point, before stepping into the proof, the protocol needs to initialize some parameters.

First, the server reconstructs the **Valid** circuit from the type and number of bits of the data provider data which are public. Two structures called *Checker* and *CheckerPrecomp* were already implemented in the Prio code and are re-used from the original implementation. They are initialized with the circuit computed previously and the leader index. The leader is the server that returns the proof result, and that coordinates the proving protocol. The protocol starts by assigning a request to the Checker structure and reconstructing the polynomial shares $[f]_i$ and $[g]_i$ for server i by traversing the circuit. The server then evaluates the expression $[d]_i$ and $[e]_i$ from the MPC protocol described in section 1. Then they all broadcast these shares, and reconstruct d and e. An optimization was also already done in Prio, which is the verification without interpolation. The point r to do the polynomial test is fixed beforehand, this way any server can interpolate and evaluate in one single step.
Then the goal is to check that the evaluation of $\hat{f}(r) \cdot \hat{g}(r) - \hat{h}(r) = 0$ at each server. These values are all sent to the root that checks that the sum is 0. In the same ways as before all communications are made in bytes, and channels are used to solve the problem of waiting for the result.
At the end, the protocol returns the original shares to aggregate, that is the output of the circuit evaluation on shares if the test is true, or empty shares if the protocol fails.

6.2 Unlynx system implementation

We present the addition made to Unlynx framework to verify that a cipher belongs to a correct defined range.

From this point on, Unlynx's DPs do not store encrypted data, but locally aggregate the result of the query and then encrypt the locally computed aggregation before sending it to the CA.

Input Range

Now we are going to look at the input range validation implementation for EC ElGamal:

First and foremost, pairing over elliptic curve is not supported by all the curves, so we used a pairing already implemented by DeDiS in the paper_dfinity repository [24]. This is based on a Barreto-Naehrig curve implementation produced by dfinity [25], that uses some libraries in C++ depicted in the GitHub description. We then use this curve for the whole Unlynx protocol. The protocol as proposed here with this configuration differs from the original one, so it should be subjected to a security analysis under a simulator argument to assert that it preserves all the security features of the original protocol. This is left as future work.

Then, the data provider needs to prove to a single server that a secret σ lies in a given range. The Range proof validation protocol assumes that the verifiers (in our scenario, the CA servers) sign each element of the base u with a key that has to remain secret. For the protocol to be sound, it must happen that if the statement is false no cheating prover can convince the honest verifier that it is true, except with a small probability. Hence, we do not want the provers the prover to be able to forge signatures. With Boneh-Boyen signature, as long as the key remains secret, this property holds.
In order to conform to Anytrust model, the computation of the key cannot be performed with a sequential protocol, as the servers would all have the same private key and if one divulgates it, the soundness of the protocol would not hold.
The first option is the computation of the signatures in a sequential way (in the same way as the key switch protocol does), with each server having their own private key, and contributing to the final signature. Every step should be verified with zero-knowledge proof, and at the end, the final server broadcasts the signatures computed, and the public key constructed to verify those. This way, no server can leak the private key because none of the servers have the full key, but all servers have the same signatures.
An alternative option is that each server has its own private key and computes the signatures. All servers will verify an input from a data provider and the protocol passes if and only if all server found that the secret σ indeed lies in the given range. This ensures that at least one proof has been done on a secret key that remained secret.
A third alternative would be that the signatures are collectively constructed; this would require a protocol for the collective computation of an inverse. Because of the uncertainty of this approach, the second approach was chosen.

Algorithm 2 Non-Interactive Range Validation

Common Input: B the base point in the EC, P the public key used to encode data, u, l 2 integers and commitment C.
Prover Input: σ the secret integer mapped to a point and r a scalar in the EC such that $C = \sigma + Pr$, $\sigma \in [0, u^l)$

Initialization Phase: Each server i in the collective authority compute the following values :
Pick a random $x_i \in \mathbb{Z}_p$
$y_i \leftarrow Bx_i$
$A_{i,j} \leftarrow B(x_i + j)^{-1}$, $\forall j \in \mathbb{Z}_u$ and $i \in \{1, ..., m\}$

Servers make their signature public as well as the key y_i. When a query is issued by a querier Q, we now assume that the range are contained in the query and broadcasted by the server as usual to the data provider.

Online Phase: Data provider encodes the signature of the value to check in base u with randomly picked v_j.
So $\forall j \in \mathbb{Z}_l$ such that $\sigma = \sum_j \sigma_j u^j$, it picks $v_j \in \mathbb{Z}_p$ and compute $V_{i,j} = A_{i,\sigma_j} v_j$
It also picks 3 values $s_j, t_j, m_j \in \mathbb{Z}_p$, $\forall j \in \mathbb{Z}_l$ and sends:
First : a value $c = H(B, C, y_i)$, where $H()$ is a cryptographic hash function.
Then: $Z_r = m - rc$ and $D \leftarrow \sum_j (u^j s_j + P m_j)$
And eventually $\forall j \in \mathbb{Z}_l$
$a_{i,j} \leftarrow e(V_{i,j}, B)(-s_j) + e(B, B)(t_j)$
$Z_{\sigma_j} \leftarrow s_j - \sigma_j c$ and $Z_{v_j} \leftarrow t_j - v_j c$
To be more precise the data provider sends the following values to ALL servers: $c, Z_r, Z_{v_j}, Z_{\sigma_j}$ with C the encrypted value public, and $D, a_{i,j}$ value published to check proof.

Server i checks that $D = Cc + PZ_r + \sum_j (u^j Z_{\sigma_j})$
$a_{i,j} = e(V_{i,j}, y)c + e(V_{i,j}, B)(-Z_{\sigma_j}) + e(B, B)(Z_{v_j})$, $\forall j \in \mathbb{Z}_l$, and publish the result.
Then the server responsible for the data provider keeps the value if all the published value match the one computed by the data provider.

At the end of the protocol, all values computed are made public so that anyone can verify that the servers have computed the verification correctly and act accordingly. This means that servers and data providers act as verifiers/provers. When a prover publishes its proof, it can be universally verified by any entity.

The completness follows from inspection. The soundness is still based on the Boneh signature.
The zero-knowledge was satisfied in the interactive version of this protocol. As we use the Fiat-Shamir heuristic [26] to transform the protocol into a non-interactive one, the zero-knowledge property is still assumed to hold.

Let us now look at the communication complexity. The initialization phase, which includes signature and public key sending, is not counted as a part of the protocol, as it is executed once in the beginning and reused afterwards.
The prover sends l blinded values V_j to each verifier, as well as l Z_{σ_j} and Z_{v_j} but also each a_j computed from the blinded signatures, a challenge c, a value Z_r, and a value D.
Taking into account the mapping $e : G_1 \times G_1 \rightarrow G_T$, it requires $l \cdot |G_1|$ communication to send V_j, $l \cdot |Z_p|$ bits for both Z_{v_j} and Z_{σ_j}. a_j needs $l \cdot |G_T|$ bits, and the unique items c, Z_r and D require respectively $|Z_p|, |Z_p|$ and $|G_1|$ bits.

The total communication required for the protocol in bits is:
$Com(P, V) = l \cdot (|G_1| + 2 \cdot |Z_p| + |G_T|) + |G_1| + 2 \cdot |Z_p|$

Optimizations

The input range implementation can be further optimized and we discuss the possible improvements in this sub-section as well as what was implemented in the actual framework.
First of all, the service implemented consists of 3 steps: Range verification, Aggregation, and Key switch. This is the best approximation of service to compare with the Prio verification and aggregation, with the least differences.
This service is in no case optimized, it serves as a comparison, so a lot of values are hardcoded, such as the number of data providers submissions to wait for before aggregation, or the keys for encryption, which are chosen beforehand randomly.

One thing is that each server received a different structure containing all parameters to validate including $C, c, \{V\}, \{Z_v\}, \{Z_\sigma\}, D$ and a. We could improve the protocol to compute only once the D and Z_r, and pick for a single value the same m_j, t_j and s_j at prover.
This would lead to a small reduction in computation time at the prover side, and some bandwidth reduction for servers.

7 Performance evaluation

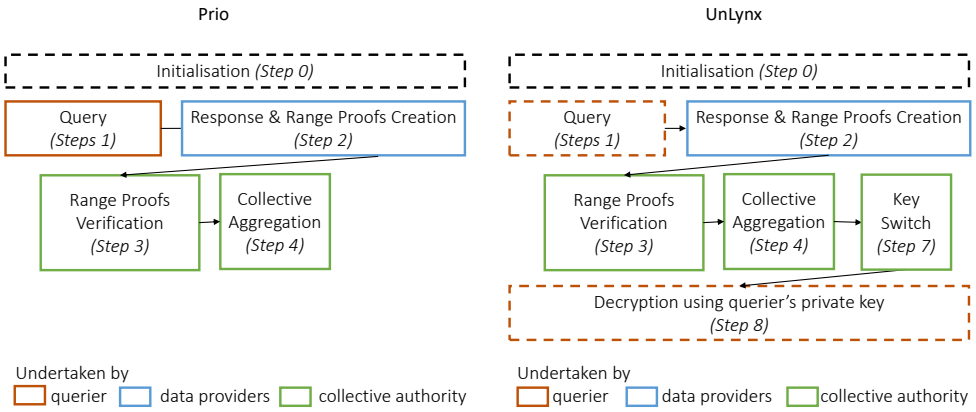


Figure 4: Pipeline for Unlynx and Prio. Dotted steps are not measured.

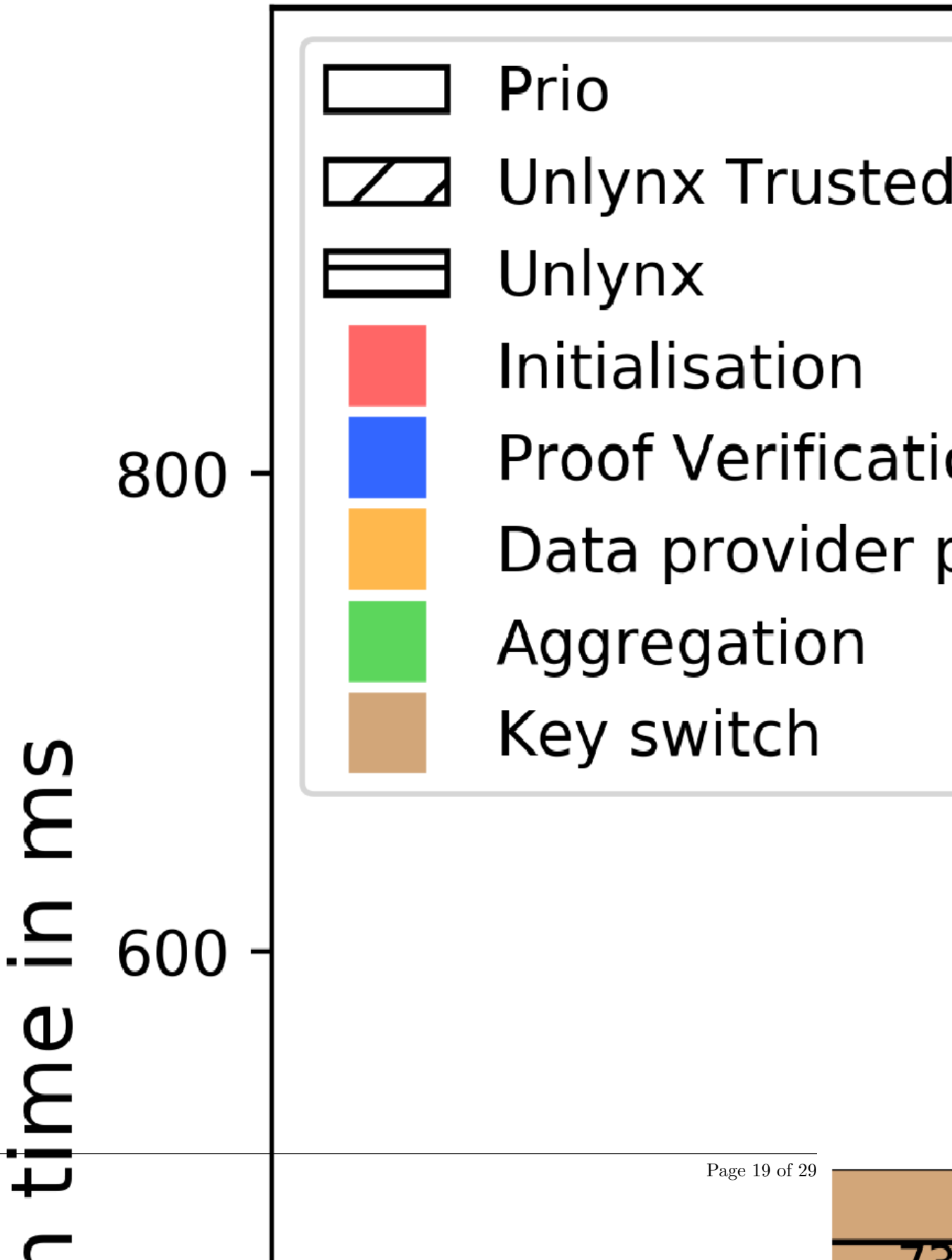
This section presents the time required for the 3 different systems described in the next lines to execute a request. Figure 4 presents the protocol for each system (both versions of Unlynx have the same pipeline).
All tests were run on VMWare, Ubuntu64 bits v17.10, 6.1 GB RAM, 2 cores Intel I5-4590 @3.3Ghz.
For Prio, 64 bits inputs are used, while we use a range of $[0, 16^{16})$ for Unlynx.
We evaluate the computation time in function of the number of servers, with 10 data providers (each having one data point), and the scaling with the number of data providers with 5 fixed servers.
The 3 systems are defined as follows:

- Prio, with the threat model presented in section 2.
- *Trusted Unlynx*, that represents the Unlynx system with the exact same threat model as Prio, i.e, we need all the servers to be trusted to ensure correctness. The range validation is slightly modified, as the prover (DP) only needs to send the proof to the server responsible for the aggregation of its data. This drastically reduces the communication and the computation time because the server can verify in parallel the data coming from the DPs. In this system, DPs only compute one proof from one set of signatures and send it to a single server, which also reduces the computation time at the data provider. The data are assumed to be evenly distributed to the server in this context.
- Unlynx under its original threat model.

Prio already assumes that each data provider send a single value, but both Unlynx systems are modified so that each data provider aggregate locally their querry result, before sending it encrypted to the server.

7.1 Scaling in number of data providers

We first present Figure 5 showing the execution time of the three protocols for increasing number of data providers (from 3 to 10).
The number of servers is fixed to 5.


Prio
Unlynx Trusted
Unlynx
Initialisation
Proof Verificati
Data provider p
Aggregation
Key switch
800
600
n time in ms

The comparison of computation time will be done in the same order as the protocol runs it.
First, an interesting computation time evolution to look at is the *input validation*. In Prio, it grows linearly with a value of 75 ms at 3 servers and 297 ms at 10 servers. For the original Unlynx, it also grows linearly with the number of data providers, but starts at 187ms for 3 servers and hits 625 ms at 10 data providers. Trusted Unlynx outperforms both those systems with a linear growth in function of the ratio $\frac{\#DP}{\#Servers}$ (if the data are evenly distributed across all DPs). This leads to a time of 62 ms for 3 and 5 servers, and 125 ms for 10 servers. First, *Initialization phases* for Prio or Unlynx are negligible compared to the other steps of the protocol with an execution time inferior to 5 ms.
The *Aggregation* protocol is constant for all protocols with an approximate time of 10 ms.
The difference of threat model in the Unlynx protocol is accentuated with the DPs proof computation time. In the trusted one it is only 46 ms while in the non trusted, it is increased by a factor 5 (234 ms), which is the number of servers.
Indeed in the second case the data providers need to compute the proof for each server. At last, the *Key Switch* protocol is constant with 73 ms no matter how many data providers there are.

Overall, when the number of data providers grows, trusted Unlynx outperforms Prio. The original Unlynx is way behind with an execution time almost 4 times higher to Prio.
We support our analysis with Figure 6, that scales the number of data providers to 500. The scale of execution times is logarithmic, however, trusted Unlynx is the best protocol in term of execution time ahead of Prio, and both of them are way better than Unlynx.

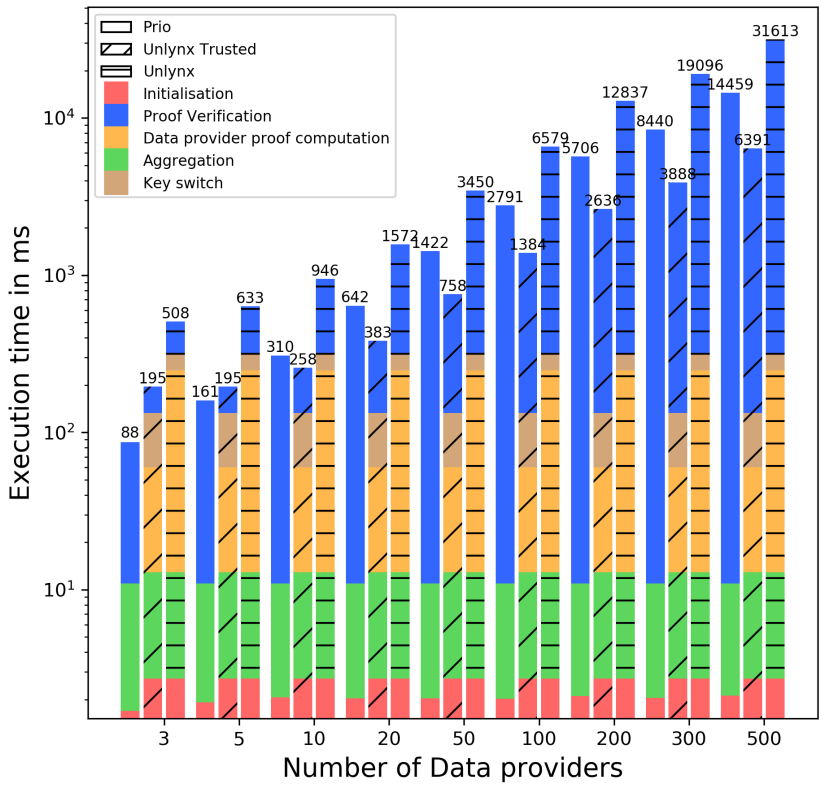
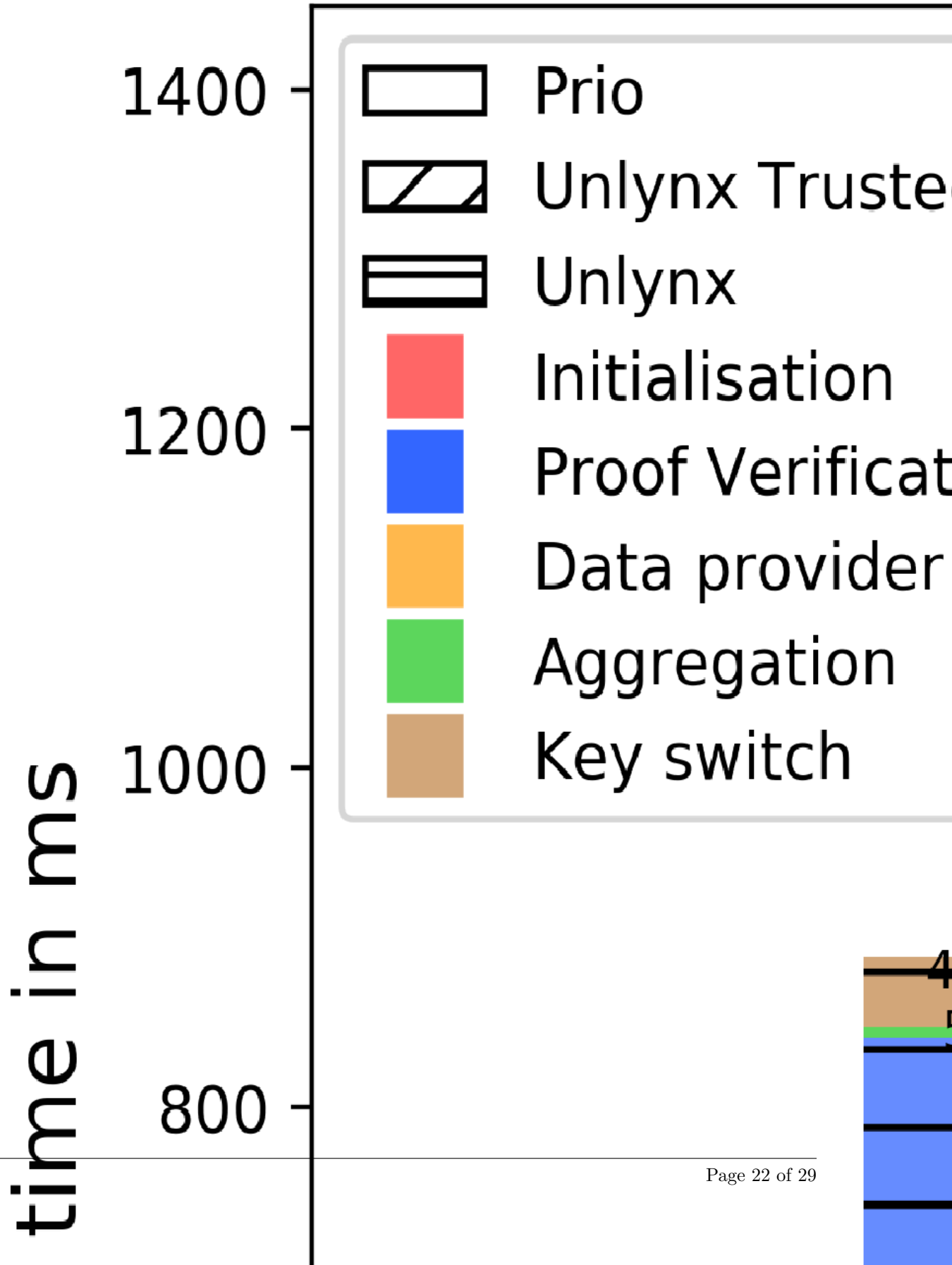


Figure 6: Comparison of the 3 systems in function of the number of data providers

7.2 Scaling in number of servers

We now present the execution time of the three protocols for increasing number of servers (from 3 to 10), shown in Figure 7.
The number of data providers is fixed to 10.


1400
1200
1000
800
time in ms
Prio
Unlynx Truste
Unlynx
Initialisation
Proof Verificat
Data provider
Aggregation
Key switch

First the execution times that do not change (or barely change) in function of the number of servers are *the Data provider proof computation* for trusted Unlynx system. It is around 50 ms, while it grows linearly in the number of servers in Unlynx amounting to 225 ms and 470 ms with 5 and 10 servers respectively.
The *aggregation* grows with the number of servers going from 10 ms at 5 servers to 45 ms at 10 servers for Unlynx. Prio starts lower at 8 ms and hits 29 ms. So Prio aggregation is more efficiently scalable with the number of servers.
Key Switch also increases more than linearly when the number of servers grows large, going from 69 ms to 233 ms when comparing 5 and 10 servers.
The most interesting result is the *Proof Verification*. In Prio, it starts high (297 ms) and increases when the number of servers go larger (780 ms) than 5.
In trusted Unlynx, it also starts high with 275 ms but decreases with the ratio $\frac{\#DP}{\#Server}$. So when there is a number of servers equal or larger than the number of data providers, the time to verify is around 62 ms. Unlynx's verification phase decreases slightly with the number of servers, and goes under Prio's verification time at 10 servers (629 ms vs 780 ms).

Overall, trusted Unlynx is, again, the protocol that outperforms the other two. But there is a difference between Prio an Unlynx compared to the scaling in the number of data providers. Indeed, when looking at Figure 8, we notice that Unlynx performs better than Prio when the number of servers grows bigger than 20. This is likely because of Prio's broadcast step in the verification. This step becomes bottleneck in terms of execution time, as the servers need to send a lot of messages to all other servers and wait for all to received for synchronizing.

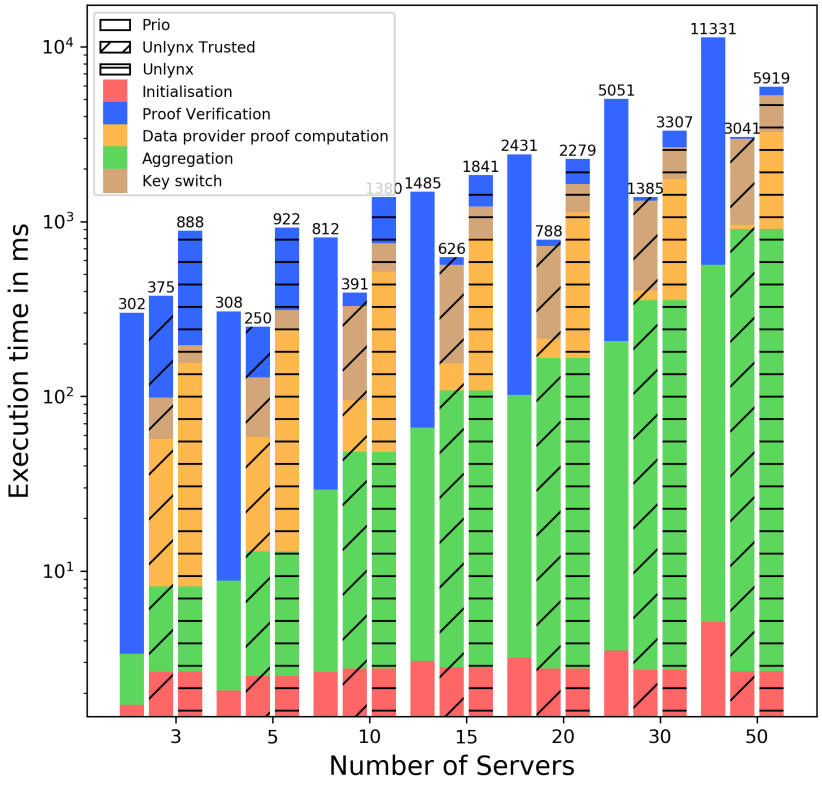


Figure 8: Comparison of the 3 systems in function of the number of servers

8 Systems' Comparison

As explained previously, the service implemented for Unlynx range proof consists of 3 steps: Range verification, Aggregation, and Key switch. This is the best break down of the services to compare with the Prio verification and aggregation, with the least significant differences.
The key switch used is not the optimal one as it does not use the byte structure to communicate, due to a problem on the new Elliptic curve used. This problem is linked to the implementation of the EC, with some missing functions.

The threat models are almost the same (the small difference is still in the collective authority, where no server or data provider needs to be trusted for Unlynx to work) and we study the time it takes to perform an aggregation.
To be more precise, in Prio we measure the time, for different clients, it takes to verify all submissions, aggregate and publish the computed aggregate function. While in Unlynx, the measured time is the sum of the verification, the aggregation of cipher and the key switch, as the servers need to re-encrypt the aggregate data to make it recoverable by the querier.

For Prio, Table 1 reports time ans size measurements for one client input and different number of servers:

Table 1: Prio measurements

# Servers	3	5	10
Data provider initialization time (creation of request for Prio protocol) (6bits/32bits/64bits)	0.401 ms/ 1.83 ms / 3.45 ms	0.563 ms / 2.11 ms /3.329 ms	0.863ms / 2.67 ms / 4.3 ms
Size of data sent in bytes by Data provider in function of circuit size (136 bytes / 328 bytes / 584 bytes)	136*3 / 328*3 / 584 *3	136*5 / 328*5 / 584*5	1360 / 3280 / 5840
Size of messages sent in broadcast phase of the protocol by 1 server /Total size of all message exchanged in broadcast (in bytes)	32 / 96	64/320	144/1440
Size of message sent by 1 server to root for final phase / Total size of messages sent to root for final phase (in bytes)	8 / 16	8 / 32	8 / 72
Total time to execute the verification at the servers in function of circuit size (6bits/32bits/64bits)	8.07 ms/ 11.95 ms/ 19.57 ms	20.19ms / 35.95ms/ 46.80ms	67.85ms/85.72ms/115.1ms
Aggregation time for all server in function of data size in bits (6/32/64)	2.32ms / 2.76ms / 3.1ms	10.47ms / 11.06ms / 11.78ms	29.37ms / 35.32ms / 37.93ms

We can state that the execution time grows linearly with the circuit size, and it is mentionned in Prio's paper [2] that it is possible to do it with a sub-linear factor.
Moreover, the quantity of sent data is pretty low for small size input data, and grows sub linearly with the size, in bits, of the input data.
The size of data in the broadcast phase grows quadratically with the number of servers, which is problematic if the number of servers grows. However, the number of servers in the collective authority should not grow too large in practical scenarios, in which case this is still acceptable.

Table 2 reports the same measurements for Unlynx with the same threat model as the orginal work [1]:

Table 2: Unlynx range proof measurement

# Server	3	5	10
Time for initialization of Signature at 1 Server in function of base u (2/4/8)	0.93ms/1.20ms/1.97ms	0.93ms/1.20ms/1.97ms	0.93ms/1.20ms/1.97ms
Size of Signatures in function of base u at 1 Server (2/4/8)	64 bytes /112 bytes /208 bytes	64 bytes /112 bytes /208 bytes	64 bytes /112 bytes /208 bytes
Time at Data provider to compute proof value from signature in function of exponent l (6/10/20)	19.41*3 ms/31.75*3 ms/59.27*3 ms	97.05ms/158.75ms/296.35ms	194.1ms/317.5ms/592.7ms
Total size of message send by Data provider to all server in function of l (6/10/20)	1520 bytes/ 2480 bytes/ 4880 bytes	2480 bytes/ 4080 bytes/ 8080 bytes	4880 bytes/ 8080 bytes/ 16080 bytes
Computation time at 1 server for proof verification in function of l (6/10/20)	25.02ms/39.16ms/81.18ms	25.02ms/39.16ms/81.18ms	25.02ms/39.16ms/81.18ms
Aggregation time	4.67ms	8.43ms	31.38ms
Key switch time	19.34ms	37.97ms	97.28ms

The initialization time is negligible with a really low execution time compared to the other steps of the protocol.
The set of signatures' size is not really large, as the range should be chosen wisely, e.g, pick 16^{16} instead of 2^{64} to optimize global traffic. A major problem is the computation time for proof at DPs. It grows linearly in the number of servers from 60 ms to 600 ms for 10 servers. This is due to the fact that each server will send its validation to the data provider responsible for the data. It also means that the traffic for proof sending to the servers grows the same way, achieving more than 10000 bytes for 10 servers.
Aggregating ciphers in Unlynx takes a little longer. Both Aggregation and Key Switch are the exact same protocol as in the original work [1], and scale almost linearly.

We also want to compare Prio and Unlynx with the exact same threat model, and in this case, the DP sends its data to the server it is connected to, and this server is the only one that does the verification of the proof for this DP. This solution gives us the following table:

Table 3: Trusted Unlynx range proof measurement

# Server	3	5	10
Time for initialization of Signature at 1 Server in function of base u (2/4/8)	0.93ms/1.20ms/1.97ms	0.93ms/1.20ms/1.97ms	0.93ms/1.20ms/1.97ms
Size of Signatures in function of base u at 1 Server (2/4/8)	64 bytes /112 bytes /208 bytes	64 bytes /112 bytes /208 bytes	64 bytes /112 bytes /208 bytes
Time at Data provider to compute proof value from signature in function of exponent l (6/10/20)	19.41ms/31.75ms/59.27ms	19.41ms/31.75ms/59.27ms	19.41ms/31.75ms/59.27ms
Total size of message send by Data provider to its server in function of l (6/10/20)	560 bytes/ 880 bytes/ 1680 bytes	560 bytes/ 880 bytes/ 1680 bytes	560 bytes/ 880 bytes/ 1680 bytes
Computation time at 1 server for proof verification in function of l (6/10/20)	25.02ms/39.16ms/81.18ms	25.02ms/39.16ms/81.18ms	25.02ms/39.16ms/81.18ms
Aggregation time	4.67ms	8.43ms	31.38ms
Key switch time	19.34ms	37.97ms	97.28ms

This assumption in threath model allows the data provider to compute only one proof for one set of signatures, and the traffic and execution time does not scale with the number of servers (60 ms and 1680 bytes for higher exponent parameter).
As a result, the proof verification does not grow with the number of servers, as each server receive one proof to validate. This protocol is better than the implemented one, but sacrifices the assumption that only one server need not to be malicious, for one where the whole CA must be trusted.

As a general conclusion for this tables, Prio's and trusted Unlynx seem to be the two protocols more practical for a real application. Prio shows to be faster as data are not encrypted and there is no key switch protocol to perform, contrarily to Unlynx. However, trusted Unlynx can do its verification offline after the protocol, as everything is published and can be verified later by the original querier of the data.

9 Conclusions and Future Work

This project achieves an implementation of Prio in Unlynx's framework as well as a novel non-interactive input validation algorithm for Unlynx, based on Elliptic Curve pairing, that outperforms Prio, assuming the same threat model. We also state some guidelines to implement a system that would combine both systems strengths.

Both Unlynx and Prio are decentralized systems for privacy-preserving data sharing among multiple data providers. They each have their own advantages and disadvantages but can be combined to have a protocol with more modularity.
In light of the performance and scalability comparison, we can state that Unlynx input validation could be done offline and still achieve a correct execution time close to Prio's SNIP.
The input range validation introduced for Unlynx enables to handle malicious/faulty data providers. However, the best solution implies to trust all servers, which is not desirable. Yet, this can be randomized to achieve a trade-off: choose a random number of servers to validate the inputs from each of the data providers.
What could also be done is to adapt Prio SNIPs protocol so that it can handle ciphertexts instead of simple integers.
Moreover, it would be possibly better than Prio as servers could potentially publish the share of a cipher, the same way servers publish ciphers in Unlynx, to verify computations. It would also permit additional aggregation functions integration (the one that Prio supports) into Unlynx. This would allow keeping the Anytrust model in Unlynx. The technical challenge would be whether it is possible to use arithmetic circuits, MPC, and AFEs with ciphertexts. As ElGamal ciphertexts are additively homomorphic, AFE's and MPC should be doable. Points can also be serialized to an array of bytes and so potentially be taken as input of

an arithmetic circuit.
However, it would scale badly when the number of servers grows beyond 20 due to the broadcast phase of Prio, but for practical applications working with collective authorities, this number can be assumed to be small.
Another alternative would be to compute the signatures of bases, in the initialization phase of Algorithm 2, collectively without revealing the private key to the servers of the CA. This would either lead to research on sequential inverse addition or to new alternatives in terms of signatures functions.

In any case, the protocol for input range validation does not necessarily need to be run online, as verification can be computed after aggregating and publishing the data. The querier can then check the proof and decide to discard the query if there are too much incorrect data.

Eventually, the change of Elliptic Curve in Unlynx should be subjected to a security analysis to be sure that all original properties are still ensured.

References

[1] David Froelicher, Patricia Egger, Joo Sa Sousa, Jean Louis Raisaro, Zhicong Huang, Christian Mouchet, Bryan Ford and Jean-Pierre Hubaux.
UnLynx: A Decentralized System for Privacy-Conscious Data Sharing. *EPFL*

[2] Henry Corrigan-Gibbs and Dan Boneh.
Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. *Stanford University*

[3] Jan Camenisch, Rafik Chaabouni, and abhi shelat
Efficient Protocols for Set Membership and Range Proofs. *IBM Research, EPFL, U. of Virginia*

[4] Josh Keller, K.K Rebbeca and Nicole Pelroth
How many times has your personal information been exposed to hackers ?
`http://www.nytimes.com/interactives/2015/07/29technology/personaltech/what-parts-of-your-information-`

[5] Classified Pentagon data leaked on the public cloud, BBC news
`http://www.bbc.com/news/technology-42166004`

[6] Andy Greenberg, Apple's 'differential privacy' is about collecting your data-but not your data.
`https://www.wired.com/2016/06/apples-differential-privacy-collection-data/`

[7] Departement federal de l'economie, de la formation et de la recherche DEFR.
`https://www.amstat.ch`

[8] LCA1 laboratory, EPFL.
`http://lca.epfl.ch/`

[9] DeDis laboratory, EPFL.
`https://dedis.epfl.ch/`

[10] Stanley L. Warner
Randomized response: A survey technique for eliminating evasive bias.
Journal of the American Statistical Association 60,309 (1965),63-69

[11] Ben Smith
Uber executive suggest digging up dirt on journalits.
`http://www.buzzfeed.com/bensmith/uber-executive-suggests-digging-up-dirt-on-journalists`

[12] Dyadic security `https://www.dyadicsec.com/`

[13] Dan Bogdanov, Sven Laur, and Jan Wilemson.
Sharemind: A framework for fast privacy-preserving computations. In *European Symposium on Research in Computer Security*

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

[15] Prototype implementation of Prio in Go
`https://github.com/henrycg/prio`

[16] Decentralized privacy-preserving data sharing tool : Unlynx
`https://github.com/lca1/unlynx`

[17] The business of Data selling
https://www.theguardian.com/technology/2017/jan/10/medical-data-multibillion-dollar-business-report-warns

[18] Law on Data protection in the United-States
https://content.next.westlaw.com/Document/I02064fbd1cb611e38578f7ccc38dcbee/View/FullText.html?contextData=(sc.Default)& transitionType=Default& firstPage=true& bhcp=1

[19] Satoshi Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System.
https://bitcoin.org/bitcoin.pdf

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

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

[22] Dan Boneh and Xavier Boyen. Short signatures without random oracles. In christian Caching and Jan Camenisch, editors, EUROCRYPT, volume 3027 of *Lecture notes in Computer Science*, pages 56-73. Springer,2004.

[23] J. T.,Schwartz Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM* 27,4 (1980), 701-717.

[24] Paper on dfinity to do DKG with pairing. https://github.com/dedis/paper_17_dfinity

[25] dfinity Github repository https://github.com/dfinity/go-dfinity-crypto

[26] Fiat-Shamir heuristic to transform interactive proof of knowledge to non-interactive. https://en.wikipedia.org/wiki/Fiat%E2%80%93Shamir_heuristic

[27] Ruichuan Chen, Alexey Reznichenko, and Paul Francis, (2012, April). Towards Statistical Queries over Distributed Private User Data. In NSDI (Vol. 12, pp. 13-13).

[28] Johes Bater, Satyender Goel, Gregory Elliott, Abel Kho, Craig Eggen, and Jennie Rogers (2017). SMCQL: secure querying for federated databases. Proceedings of the VLDB Endowment, 10(6), 673-684.

[29] Francisco Martn-Fernndez,* Pino Caballero-Gil, and Cndido Caballero-Gil. Authentication Based on Non-Interactive Zero-Knowledge Proofs for the Internet of Things.

[30] Henry Corrigan-Gibbs, Prio prototype implementation
https://github.com/henrycg/prio