PUA – Privacy and Unforgeability for Aggregation

Iraklis Leontiadis, Kaoutar Elkhiyaoui, Refik Molva, Melek Önen

EURECOM, Sophia Antipolis, France
{firstname.lastname}@eurecom.fr
http://www.eurecom.fr/rs/applied_crypto_group

Abstract. Existing work on data collection and analysis for aggregation is mainly focused on confidentiality issues. That is, the untrusted Aggregator learns only the aggregation result without divulging individual data inputs. In this paper we extend the existing models with stronger security requirements. Apart from the privacy requirements with respect to the individual inputs we ask for *unforgeability* for the aggregate result. We first define the new security requirements of the model. We also instantiate a protocol for private and unforgeable aggregation for a *non-interactive multi-party* environment. I.e, multiple unsynchronized users owing to personal sensitive information without interacting with each other contribute their values in a secure way: The Aggregator learns the result of a function without learning individual values and moreover it constructs a proof that is forwarded to a verifier that will let the latter be convinced for the correctness of the computation. The verifier is restricted to not communicate with the users. Our protocol is provably secure in the random oracle model.

1 Introduction

With the advent of the *Big Data era*, research on privacy preserving data collection and analysis is culminating. Users continuously produce data that can be considered as valuable whenever they are aggregated. We therefore consider a scenario whereby an Aggregator collects individual data from multiple users who do not interact with each other and executes a function which outputs an aggregate value. This result is further forwarded to the Data Analyzer who can finally extract useful information about the entire population. Various motivating examples under the predefined model exist in the real-world:

- The analysis of different user profiles and the derivation of statistics can help recommendation engines provide targeted advertisements. In such scenarios a service provider would collect data from each individual user (i.e: on-line purchases), thus acting as an Aggregator, and compute an on-demand aggregate value upon receiving a request from the advertisement company. The latter will further infer some statistics acting as a Data Analyzer, in order to send the appropriate advertisements to each category of users.
- Data aggregation is a promising tool in the field of healthcare research. Different types of data sensed by body sensors (eg. blood pressure) are collected in large scale in data enclaves who can be considered as Aggregators. Health scientists who act as Data Analyzers are interested in inferring some statistical information from

these data without having access to each individual data (for privacy and performance reasons). An aggregate value computed over a large population would give very useful information for deriving statistical models for evaluating therapeutic performance or for learning the likelihood of upcoming patients' diseases.

Unfortunately, existing solutions only focus on the problem of data confidentiality and consider the Aggregator as being *honest-but-curious*: the aggregator is curious in discovering the content of each individual data but performs the aggregation operation correctly. In this paper we consider a more powerful security model and assume the existence of an untrusted Aggregator: The Aggregator may provide a bogus aggregate value to the data analyzer. In order to protect against such a malicious behavior, we propose that along with the aggregate value, the Aggregator provides a proof on the correctness of the computation of the aggregate result.

The underlying idea of our solution is that each user encrypts its data according to Shi *et al.* [16] scheme, and sends it to the untrusted Aggregator, using their secret encryption key. They also homomorphically tag their data by using two layers of randomness with two different keys and similarly the tags are forwarded to the Aggregator. The latter computes the sum by applying operations on the ciphertexts and it also computes a proof for the correctness of the result from the tags. The Aggregator finally sends the result and the proof to the Data Analyzer who verifies the correctness of the computation. We also require the Data Analyzer not to be able to communicate with each user and the result to be publicly verifiable. Moreover, similarly to the existing solutions, the proposed protocol assures obliviousness against the Aggregator and the data analyzer in the multi-user setting; meaning that neither the data analyzer nor the aggregator learns individual data inputs.

To the best of our knowledge we are the first who define a model for *Privacy and Unforgeability for Aggregation* ($\mathbf{PUA}$). We also instantiate a $\mathbf{PUA}$ scheme which mainly pursues in three objectives:

- Multi-user setting where multiple users produce personal sensitive data without interacting with each other.
- Public verifiability of the aggregate value without holding/receiving the original data input.
- Privacy of individual data for all participants.

2 Problem Statement

We are envisioning a scenario whereby a set of users $\mathbb{U} = \{\mathcal{U}_i\}_{i=1}^{n}$ are producing sensitive data inputs $x_{i,t}$ at each time interval t. These individual data are first encrypted into ciphertexts $c_{i,t}$ and further forwarded to an untrusted aggregator $\mathcal{A}$. Aggregator $\mathcal{A}$ aggregates all the received ciphertexts, decrypts the aggregate and forwards the resulting plaintext to a data analyzer $\mathcal{DA}$ together with a cryptographic proof that assures the correctness of the aggregation operation, which in this paper corresponds to the *sum* of the users' individual data. An important criterion that we aim to fulfill in this paper is to ensure that data analyzer $\mathcal{DA}$ verifies the corretness of the aggregator's output without compromising users' privacy. Namely, at the end of the verification operation, both aggregator $\mathcal{A}$ and data analyzer learn nothing but the value of the aggregation. While homomorphic signatures proposed in [6, 12] seem to answer to the verifiability requirement, authors in those papers only consider scenarios where a single user generates data.

In the aim of assuring both individual user's privacy and unforgeable aggregation, we first come up with a generic model for privacy preserving and unforgeable aggregation that identifies the algorithms necessary to implement such functionalities and defines the corresponding privacy and security models. Furthermore, we propose a concrete solution which combines an already existing privacy preserving aggregation scheme [16] with an additively homomorphic tag designed for bilinear groups.

Notably, a scheme that allows a malicious aggregator to compute the sum of users' data in privacy preserving manner and to produce a proof of correct aggregation will start by first running a setup phase. During setup, each user receives a secret key that will be used to encrypt the user's private input and to generate the corresponding authentication tag; the aggregator $\mathcal{A}$ and the data analyzer $\mathcal{DA}$ on the other hand, are provided with a secret decryption key and a public verification key respectively. After the key distribution, each user sends its data encrypted and authenticated to aggregator $\mathcal{A}$, while making sure that the computed ciphertext and the matching authentication tag leak no information about its private input. On receiving users' data, aggregator $\mathcal{A}$ first aggregates the received ciphertexts and decrypts the sum using its decryption key, then uses the received authentication tags to produce a proof that demonstrates the correctness of the decrypted sum. Finally, data analyzer $\mathcal{DA}$ verifies the corretness of the aggregation thanks to the public verification key.

2.1 PUA Model

A $\mathbf{PUA}$ scheme consists of the following algorithms:

- **Setup**$(1^\kappa) \rightarrow (\mathcal{P}, \mathsf{sk_A}, \{\mathsf{SK}_i\}_{\mathcal{U}_i \in \mathbb{U}}, \mathsf{VK})$: It is a randomized algorithm run by a trusted dealer which on input of a security parameter κ outputs the public parameters $\mathcal{P}$ that will be used by subsequent algorithms, the aggregator $\mathcal{A}$'s secret key $\mathsf{sk_A}$, the secret keys $\mathsf{SK_i}$ of users $\mathcal{U}_i$ and the public verification key VK.
- **EncTag**$(t, \mathsf{SK}_i, x_{i,t}) \rightarrow (c_{i,t}, \sigma_{i,t})$: It is a randomized algorithm which on inputs of time interval t, secret key SK_i of user $\mathcal{U}_i$ and data $x_{i,t}$, encrypts $x_{i,t}$ to get a ciphertext $c_{i,t}$ and computes a tag $\sigma_{i,t}$ authenticating $x_{i,t}$.
- **Aggregate**$(\mathsf{sk_A}, \{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}) \rightarrow (\mathsf{sum_t}, \sigma_\mathsf{t})$: It is a deterministic algorithm run by the aggregator $\mathcal{A}$. It takes as inputs aggregator $\mathcal{A}$'s secret key $\mathsf{sk_A}$, ciphertexts $\{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$ and authentication tags $\{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$, and outputs in cleartext the sum $\mathsf{sum_t}$ of the values $\{x_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$ and computes a proof σ_t assessing the correctness of $\mathsf{sum_t}$ using the authentication tags $\{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$.
- **Verify**$(\mathsf{VK}, \mathsf{sum_t}, \sigma_\mathsf{t}) \rightarrow \{0, 1\}$: It is a deterministic algorithm that is executed by the data analyzer $\mathcal{DA}$. It outputs 1 if data analyzer $\mathcal{DA}$ is convinced that the sum $\mathsf{sum_t} = \sum_{\mathcal{U}_i \in \mathbb{U}} \{x_{i,t}\}$; and 0 otherwise.

2.2 Security Model

In this paper, we only focus on the adversarial behavior of aggregator $\mathcal{A}$. The rationale behind this is that aggregator $\mathcal{A}$ is the only party in the protocol that sees all the messages exchanged during the protocol execution: Namely, aggregator $\mathcal{A}$ has access to users' ciphertexts and it is the party that interacts directly with the data analyzer. It follows that by ensuring security properties against the aggregator, one by the same token ensures these security properties against both data analyzer $\mathcal{DA}$ and external parties.

In accordance with previous work [13, 16], we formalize the property of *aggregator obliviousness*. Aggregator obliviousness ensures that at the end of a protocol execution, aggregator $\mathcal{A}$ only learns the sum of users' inputs $x_{i,t}$ and nothing else. Also, we enhance the security definitions of data aggregation with the notion of *aggregate unforgeability*. As the name implies, aggregate unforgeability guarantees that aggregator $\mathcal{A}$ cannot forge a valid proof σ_{t} for a sum $\mathsf{sum_t}$ that was not computed correctly from users' inputs (i.e. cannot generate a proof for $\mathsf{sum_t} \neq \sum x_{i,t}$).

Aggregator Obliviousness Aggregator obliviousness ensures that when users $\mathcal{U}_i$ provide aggregator $\mathcal{A}$ with ciphertexts $c_{i,t}$ and authentication tags $\sigma_{i,t}$, they do not reveal any information about their individual inputs $x_{i,t}$, other than the sum value $\sum x_{i,t}$. We extend the existing definition of *Aggregator Obliviousness* (cf. [13, 14, 16]) so as to capture the fact that aggregator $\mathcal{A}$ not only has access to ciphertexts $c_{i,t}$, but also has access to the authentication tags $\sigma_{i,t}$ that enable aggregator $\mathcal{A}$ to generate proofs of correct aggregation.

Similarly to the work of [13, 16], we formalize aggregator obliviousness using an indistinguishability-based game in which aggregator $\mathcal{A}$ accesses the following oracles:

- $\mathcal{O}_{\mathsf{Setup}}$: When called by aggregator $\mathcal{A}$, this oracle initializes the system parameters; it then gives the public parameters $\mathcal{P}$, the aggregator's secret key $\mathsf{sk_A}$ and public verification key VK to $\mathcal{A}$.
- $\mathcal{O}_{\mathsf{Corrupt}}$: When queried by aggregator $\mathcal{A}$ with a user $\mathcal{U}_i$' s identifier uid_i, this oracle provides aggregator $\mathcal{A}$ with $\mathcal{U}_i$'s secret key denoted $\mathsf{SK_i}$.
- $\mathcal{O}_{\mathsf{EncTag}}$: When queried with time t, user $\mathcal{U}_i$'s identifier uid_i and a data point $x_{i,t}$, this oracle outputs the ciphertext $c_{i,t}$ and the authentication tag $\sigma_{i,t}$ of $x_{i,t}$ computed using $\mathcal{U}_i$'s secret key SK_i.
- $\mathcal{O}_{\mathsf{AO}}$: When called with a subset of users $\mathbb{S} \subset \mathbb{U}$ and with two time-series $(\mathcal{U}_i, t, x^0_{i,t})_{\mathcal{U}_i \in \mathbb{S}}$ and $(\mathcal{U}_i, t, x^1_{i,t})_{\mathcal{U}_i \in \mathbb{S}}$ such that $\sum x^0_{i,t} = \sum x^1_{i,t}$, this oracle flips a random coin $b \in \{0,1\}$ and returns an encryption of the time-serie $(\mathcal{U}_i, t, x^b_{i,t})_{\mathcal{U}_i \in \mathbb{S}}$ (that is the tuple of ciphertexts $\{c^b_{i,t}\}_{\mathcal{U}_i \in \mathbb{S}}$) and the corresponding authentication tags $\{\sigma^b_{i,t}\}_{\mathcal{U}_i \in \mathbb{S}}$.

Aggregator $\mathcal{A}$ is accessing the aforementioned oracles during a learning phase (cf. Algorithm 1) and a challenge phase (cf. Algorithm 2). In the learning phase, $\mathcal{A}$ calls oracle $\mathcal{O}_{\mathsf{Setup}}$ which in turn returns the public parameters $\mathcal{P}$, the public verification key VK and the aggregator's secret key $\mathsf{sk_A}$. It also interacts with oracle $\mathcal{O}_{\mathsf{Corrupt}}$ to learn the secret keys SK_i of users $\mathcal{U}_i$, and oracle $\mathcal{O}_{\mathsf{EncTag}}$ to get a set of ciphertexts $c_{i,t}$ and authentication tags $\sigma_{i,t}$.

In the challenge phase, aggregator $\mathcal{A}$ chooses a subset $\mathbb{S}^*$ of users that were not corrupted in the learning phase, and a challenge time interval t^* for which it did not make an encryption query. Oracle $\mathcal{O}_{\mathsf{AO}}$ then receives two time-series $\mathcal{X}^0_{t^*} = (\mathcal{U}_i, t^*, x^0_{i,t^*})_{\mathcal{U}_i \in \mathbb{S}^*}$ and $\mathcal{X}^1_{t^*} = (\mathcal{U}_i, t^*, x^1_{i,t^*})_{\mathcal{U}_i \in \mathbb{S}^*}$ from $\mathcal{A}$, such that $\sum x^0_{i,t^*} = \sum x^1_{i,t^*}$. Then oracle $\mathcal{O}_{\mathsf{AO}}$ flips a random coin $b \xleftarrow{\$} \{0,1\}$ and returns to $\mathcal{A}$ the ciphertexts $\{c^b_{i,t^*}\}_{\mathcal{U}_i \in \mathbb{S}^*}$ and the matching authentication tags $\{\sigma^b_{i,t^*}\}_{\mathcal{U}_i \in \mathbb{S}^*}$.

At the end of the challenge phase, aggregator $\mathcal{A}$ outputs a guess b^* for the bit b.

We say that aggregator $\mathcal{A}$ succeeds in the aggregator obliviousness game, if its guess b^* equals b.

Algorithm 1: Learning phase of the obliviousness game

$(\mathcal{P}, \mathsf{SK_A}, \mathsf{VK}) \leftarrow \mathcal{O}_{\mathsf{Setup}}(1^\kappa)$;
// $\mathcal{A}$ executes the following a polynomial number of
// times
$\mathsf{SK}_i \leftarrow \mathcal{O}_{\mathsf{Corrupt}}(\mathsf{uid}_i)$;
// $\mathcal{A}$ is allowed to call $\mathcal{O}_{\mathsf{EncTag}}$ for all users $\mathcal{U}_\mathsf{i}$
$(c_{i,t}, \sigma_{i,t}) \leftarrow \mathcal{O}_{\mathsf{EncTag}}(t, \mathsf{uid}_i, x_{i,t})$;

Algorithm 2: Challenge phase of the obliviousness game

$\mathcal{A} \rightarrow t^*, \mathbb{S}^*$;
$\mathcal{A} \rightarrow \mathcal{X}^0_{t^*}, \mathcal{X}^1_{t^*}$;
$(c^b_{i,t^*}, \sigma^b_{i,t^*})_{\mathcal{U}_i \in \mathbb{S}^*} \leftarrow \mathcal{O}_{\mathsf{AO}}(\mathcal{X}^0_{t^*}, \mathcal{X}^1_{t^*})$;
$\mathcal{A} \rightarrow b^*$;

Definition 1 (Aggregator Obliviousness). *An aggregation protocol is said to ensure aggregator obliviousness if for any polynomially bounded adversary $\mathcal{A}$ the probability* $\Pr(b = b^*) \leqslant \frac{1}{2} + \epsilon(\kappa)$*, where ϵ is a negligible function and κ is the security parameter.*

Aggregate Unforgeability We augment the security requirements of data aggregation with the requirement of *Aggregate Unforgeability*. More precisely, we assume that aggregator $\mathcal{A}$ is not only interested in compromising the privacy of users participating in the data aggregation protocol, but also interested in tampering with the sum of users' inputs. That is, aggregator $\mathcal{A}$ may sometimes has an incentive to feed data analyzer $\mathcal{DA}$ with erroneous sums $\mathsf{sum_t}$). Along these lines, we define *Aggregate Unforgeability* as the security feature that ensures that aggregator $\mathcal{A}$ cannot convince data anlyzer $\mathcal{DA}$ to accept a bogus sum, and assume that users $\mathcal{U}_i$ in the system are honest[1]. (i.e. they always submit their correct input and do not collude with the aggregator $\mathcal{A}$).

[1] A scheme with dishonest users that will try produce tags on behalf of other users can be constructed with an existential unforgeable signature scheme Σ in which each user signs the tag with its secret signing key $\Sigma.\mathsf{ssk}_i$

Algorithm 3: Learning phase of the Aggregate Unforgeability game

$\mathcal{P}, \mathsf{VK} \leftarrow \mathcal{O}_{\mathsf{Setup}}(1^{\kappa})$;
// $\mathcal{A}$ executes the following a polynomial number of
//times
// $\mathcal{A}$ is allowed to call $\mathcal{O}_{\mathsf{Tag}}$ for all users $\mathcal{U}_{\mathsf{i}}$
$(c_{i,t}, \sigma_{i,t}) \leftarrow \mathcal{O}_{\mathsf{EncTag}}(t, \mathsf{uid}_i, x_{i,t})$;

Algorithm 4: Challenge phase of the Aggregate Unforgeability game

$(t^*, \mathsf{sum}_{t^*}, \sigma_{t^*}) \leftarrow \mathcal{A}$

In compliance with previous work [9, 12], we formalize *Aggregate Unforgeability* via a game in which aggregator $\mathcal{A}$ accesses as oracles $\mathcal{O}_{\mathsf{Setup}}$ and $\mathcal{O}_{\mathsf{EncTag}}$. Furthermore, given the property that anyone holding the public verification key VK can execute the algorithm Verify, we assume that aggregator $\mathcal{A}$ during the unforgeability game can run the algorithm Verify for itself.

As shown in Algorithm 3, Aggregator $\mathcal{A}$ enters the *Aggregate Unforgeability* game by querying the oracle $\mathcal{O}_{\mathsf{Setup}}$ with a security parameter. Oracle $\mathcal{O}_{\mathsf{Setup}}$ accordingly returns public parameters $\mathcal{P}$, verification key VK and the secret key $\mathsf{sk_A}$ of aggregator $\mathcal{A}$. Moreover, aggregator $\mathcal{A}$ calls oracle $\mathcal{O}_{\mathsf{EncTag}}$ with tuples $(t, \mathsf{uid}_i, x_{i,t})$ in order to receive the ciphertext $c_{i,t}$ encrypting $x_{i,t}$ and the matching authenticating tag $\sigma_{i,t}$, both computed using user $\mathcal{U}_i$'s secret key SK_i. Note that for each time interval t, aggregator $\mathcal{A}$ is allowed to query oracle $\mathcal{O}_{\mathsf{EncTag}}$ for user $\mathcal{U}_i$ only once. In other words, aggregator $\mathcal{A}$ cannot submit two distinct queries to oracle $\mathcal{O}_{\mathsf{EncTag}}$ with the same time interval t and the same user identifier uid_i. Without loss of generality, we suppose that for each time interval t, aggregator $\mathcal{A}$ invokes oracle $\mathcal{O}_{\mathsf{EncTag}}$ for all users $\mathcal{U}_i$ in the system.

At the end of *Aggregate Unforgeability* game (see Algorithm 4), aggregator $\mathcal{A}$ outputs a tuple $(t^*, \mathsf{sum}_{t^*}, \sigma_{t^*})$.

Accordingly, we say that aggregator $\mathcal{A}$ wins the *Aggregate Unforgeability* game if the one of following statements holds:

1. $\mathsf{Verify}(\mathsf{sum}_{t^*}, \sigma_{t^*}) \rightarrow 1$ and aggregator $\mathcal{A}$ never made a query to oracle $\mathcal{O}_{\mathsf{EncTag}}$ that comprises time interval t^*. In the remainder of this paper, we denote this type of forgery **Type I Forgery**.
2. $\mathsf{Verify}(\mathsf{sum}_{t^*}, \sigma_{t^*}) \rightarrow 1$ and aggregator $\mathcal{A}$ has made a query to oracle $\mathcal{O}_{\mathsf{EncTag}}$ for time t^*, however the sum $\mathsf{sum}_{t^*} \neq \sum_{\mathcal{U}_i} x_{i,t^*}$. In what follows, we call this type of forgery **Type II Forgery**.

Definition 2 **(*Aggregate Unforgeability*).** *Let* $\Pr[\mathcal{A}^{\mathbf{AU}}]$ *denote the probability that aggregate* $\mathcal{A}$ *wins the* Aggregate Unforgeability *game, that is, the probability that aggregator* $\mathcal{A}$ *outputs a* **Type I Forgery** *or* **Type II Forgery** *that will be accepted by algorithm* Verify.

An aggregation protocol is said to ensure Aggregate Unforgeability *if for any polynomially bounded adversary* $\mathcal{A}$*,* $\Pr[\mathcal{A}^{\mathbf{AU}}] \leqslant \epsilon(\kappa)$*, where* ϵ *is a negligible function in the security parameter* κ*.*

3 Idea of our PUA protocol

In an extended model with an untrusted Aggregator it is of utmost importance to design a solution in which the untrusted aggregator cannot provide bogus results to the data analyzer. Such a solution will use a proof system that enable the data analyzer to verify the correctness of the computation. Yet verifiability should be achieved without sacrificing privacy. Towards this end, we propose a protocol that rely on the following techniques to allow privacy preserving aggregation that also supports verifiabilty:

- A *homomorphic encryption* algorithm that allows the aggregator to compute the sum without divulging individual data.
- A *homomorphic tag* that allows each user to authenticate the data input $x_{i,t}$, in such a way that the aggregator can use the collected tags to construct a proof that demonstrates to the data analyzer $\mathcal{DA}$ the correctness of the aggregate sum.

Concisely, a set of non-interacting users are connected to personal services and devices that produce personal data. Without any coordination, each user chooses a random tag key $\mathsf{tk_i}$ and sends an encoding thereof, $\bar{\mathsf{tk}}_\mathsf{i}$ to the dealer. The dealer after collecting all encoded keys $\bar{\mathsf{tk}}_\mathsf{i}$ by users, publishes the public verification key VK of this group of users. This verification key is computed as a function of the encodings $\bar{\mathsf{tk}}_\mathsf{i}$. Later, the dealer gives to each user in the system an encryption key $\mathsf{ek_i}$ that will be used to compute the user's ciphertexts. Accordingly, the secret key of each user SK_i is defined as the pair of tag key tk_i and encryption key ek_i. Finally, the dealer provides the aggregator with secret key SK_A computed as the sum of encryption keys ek_i and goes offline.

Now at each time interval t, each user employs its secret key $\mathsf{SK_i}$ to compute a ciphertext based on the encryption algorithm of Shi *et al.* [16] and a homomorphic tag on its sensitive data input. When the aggregator collects the ciphertexts and the tags from all users, it computes the sum $\mathsf{sum_t}$ of users' data and a proof σ of correct aggregation, and forwards the sum and the proof to the data analyzer. At the final step of the protocol, the data analyzer verifies with the verification key VK and proof σ the validity of the result $\mathsf{sum_t}$.

Thanks to the homomorphic encryption algorithm of Shi *et al.* [16] and the way in which we construct our homomorphic tags, we show that our protocol ensures aggregator obliviousness. Moreover, we show that the aggregator cannot forge bogus results. Finally, we note that the data analyzer does not keep any state with respect to users transcripts be it ciphertexts or tags, but only the public verification key, the sum $\mathsf{sum_t}$ and the proof σ_t.

4 PUA Instantiation

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be cyclic groups of safe prime order p and g_1, g_2 generators of $\mathbb{G}_1, \mathbb{G}_2$ accordingly. We say that e is a bilinear map, if the following properties are satisfied:

1. *bilinearity*: $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$, where $g_1, g_2 \in \mathbb{G}_1 \times \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$.
2. *Computability*: there exists an efficient algorithm that computes $e(g_1^a, g_2^b)$ where $g_1, g_2 \in \mathbb{G}_1 \times \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$.

3. *Non-degeneracy*: $e(g_1, g_2) \neq 1$.

For encryption and result computation (**Aggregate** algorithm) we employ the *discrete logarithm* based encryption scheme originated from Shi *et al.* scheme [16]:

4.1 Shi-Chan-Rieffel-Chow-Song Scheme

- **Setup**(1^κ): Let $\mathbb{G}_1$ a subgroup of $\mathbb{Z}_p$ of safe prime order p. A trusted key dealer $\mathcal{KD}$ selects a hash function $H : \{0,1\}^* \rightarrow \mathbb{G}_1$. Furthermore, $\mathcal{KD}$ selects uniformly at random, secret encryption keys $\mathsf{ek}_i \in \mathbb{Z}_p$. It distributes them to each user $\mathcal{U}_i$ and it also sends to the Aggregator the secret key $\mathsf{sk_A} = -\sum_{i=1}^{n} \mathsf{ek}_i$.
- **Encrypt**$(\mathsf{ek}_i, x_{i,t})$: Each user $\mathcal{U}_i$ encrypts the value $x_{i,t}$ by using its secret encryption key ek_i in order to compute the ciphertext $c_{i,t} = H(t)^{\mathsf{ek}_i} g_1^{x_{i,t}} \in \mathbb{G}_1$.
- **Aggregate**$(\{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \mathsf{sk_A})$: The Aggregator upon receiving all the ciphertexts $\{c_{i,t}\}_{i=1}^{n}$ computes: $V = (\prod_{i=1}^{n} c_{i,t}) \cdot H(t)^{\mathsf{sk}_A} = H(t)^{\sum_{i=1}^{n} \mathsf{ek}_i} g_1^{\sum_{i=1}^{n} x_{i,t}} \cdot H(t)^{-\sum_{i=1}^{n} \mathsf{ek}_i} = g_1^{\sum_{i=1}^{n} x_{i,t}} \in \mathbb{G}_1$. Finally $\mathcal{A}$ learns the sum $\mathsf{sum_t} = \sum_{i=1}^{n} x_{i,t} \in \mathbb{Z}_p$ by computing the discrete logarithm of V on the base g_1:$\mathsf{sum_t} = \log_{g_1} V = \sum_{i=1}^{n} x_{i,t}$. The result computation is correct as long as $\sum_{i=1}^{n} x_{i,t} < p$.

4.2 PUA scheme

In what follows we describe our $\mathbf{PUA}$ protocol:

- **Setup**(1^κ): $\mathcal{KD}$ outputs $(p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ for an efficient computable bilinear map $e : G_1 \times G_2 \rightarrow G_T$, where g_1 and g_2 are two random generators for the multiplicative groups $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively and p is a safe prime number that denotes the order of all the groups $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$. Moreover a secret key a is selected by the $\mathcal{KD}$. Each $\mathcal{U}_i$ selects and a random tag key $\mathsf{tk}_i \in \mathbb{Z}_p$ independently and forwards $g_2^{\mathsf{tk}_i}$ to $\mathcal{KD}$. $\mathcal{KD}$ also publishes the verification key $\mathsf{VK} = (\mathsf{vk}_1, \mathsf{vk}_2) = (g_2^{\sum_{i=1}^{n} \mathsf{tk}_i}, g_2^a)$ and distributes to each user $\mathcal{U}_i \in \mathbb{U}$ the secret key $g_1^a \in \mathbb{G}_1$ through a secure channel. Thus the secret keys of the scheme are $\mathsf{SK}_i = (\mathsf{ek}_i, \mathsf{tk}_i, g_1^a)$. After publishing the public parameters $\mathcal{P} = (H, p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ and the verification key VK, $\mathcal{KD}$ goes off-line and it does not further participate in any protocol phase.
- **EncTag**$(t, \mathsf{SK}_i = (\mathsf{ek}_i, \mathsf{tk}_i, g_1^a), x_{i,t})$: At each time interval t each user $\mathcal{U}_i$ encrypts the data value $x_{i,t}$ using its secret encryption key ek_i, with the encryption algorithm as shown in subsection 4.1, that results in a ciphertext $c_{i,t} = H(t)^{\mathsf{ek}_i} g_1^{x_{i,t}} \in \mathbb{G}_1$.
  $\mathcal{U}_i$ also constructs a tag in $\mathbb{G}_1$ with its secret tag key (tk_i, g_1^a):
  $$\sigma_{i,t} = H(t)^{\mathsf{tk}_i} (g_1^a)^{x_{i,t}} \in \mathbb{G}_1$$
  and sends $c_{i,t}, \sigma_{i,t}$ to $\mathcal{A}$.
- **Aggregate**$(\mathsf{sk_A}, \{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}})$: The Aggregator computes the sum $\mathsf{sum_t} = \sum_{i=1}^{n} x_{i,t}$ by using the Aggregate function as presented in subsection 4.1.

Moreover aggregates tags by computing:

$$\sigma_{\mathsf{t}} = \prod_{i=1}^{n} \sigma_{\mathsf{i,t}} = \prod_{i=1}^{n} H(t)^{\mathsf{tk}_i}(g_1^a)^{x_{i,t}} = H(t)^{\sum \mathsf{tk}_i}(g_1^a)^{\sum x_{i,t}}$$

It finally forwards to the data analyzer $\mathcal{DA}$ $\mathsf{sum}_{\mathsf{t}}$ and σ_{t}.

– **Verify**($\mathsf{VK} = (\mathsf{vk}_1 = g_2^{\sum \mathsf{tk}_i}, \mathsf{vk}_2 = g_2^a), \mathsf{sum}_{\mathsf{t}}, \sigma_{\mathsf{t}}$): During the verification phase the $\mathcal{DA}$ verifies the correctness of the computation by checking:

$$e(\sigma_{\mathsf{t}}, g_2) \stackrel{?}{=} e(H(t), \mathsf{vk}_1)e(g_1^{\mathsf{sum}_{\mathsf{t}}}, \mathsf{vk}_2)$$

Verification correctness follows from bilinear pairing properties:

$$e(\sigma_{\mathsf{t}}, g_2) = e(\prod_{i=1}^{n} \sigma_{i,t}, g_2) = e(\prod_{i=1}^{n} H(t)^{\mathsf{tk}_i} g_1^{ax_{i,t}}, g_2) =$$
$$e(H(t)^{\sum_{i=1}^{n} \mathsf{tk}_i} g_1^{a\sum_{i=1}^{n} x_{i,t}}, g_2) = e(H(t)^{\sum_{i=1}^{n} \mathsf{tk}_i}, g_2)e(g_1^{a\sum_{i=1}^{n} x_{i,t}}, g_2) =$$
$$e(H(t), g_2)^{\sum_{i=1}^{n} \mathsf{tk}_i} e(g_1, g_2)^{a\sum_{i=1}^{n} x_{i,t}} = e(H(t), g_2^{\sum_{i=1}^{n} \mathsf{tk}_i})e(g_1^{\sum_{i=1}^{n} x_{i,t}}, g_2^a) =$$
$$e(H(t), g_2^{\sum_{i=1}^{n} \mathsf{tk}_i})e(g_1^{\mathsf{sum}_{\mathsf{t}}}, g_2^a) = e(H(t), \mathsf{vk}_1)e(g_1^{\mathsf{sum}_{\mathsf{t}}}, \mathsf{vk}_2)$$

5 Analysis

5.1 Obliviousness

Theorem 1. *The proposed solution achieves obliviousness in the random oracle model under the decisional Diffie-Hellman (DDH) assumption in* $\mathbb{G}_1$.

Due to space limitations the proof of Theorem 1 can be found in Appendix A section.

5.2 *Aggregate Unforgeability*

We first introduce a new assumption that is used during the security analysis of our **PUA** instantiation. Our new assumption named hereafter as LEMO is a variant of the LRSW assumption which is proven secure in the generic model [17] and it used for the construction of the CL signatures [7]. We omit evidence of its security for a full extended version of the current paper due to space limitations. Intuitively we follow the generic model [17] that has been used for other constructions also [5]. The new LEMO assumption extends the LRSW assumption by incorporating another public key that is used to annihilate the randomness in the second part of the authentication tag c. Moreover we introduce the time interval t that is used in order to control the randomness. I.e: the randomness a that is employed by the oracle $\mathcal{O}_{\mathsf{LEMO}}$ is reused for a message m and time interval t for which the oracle has been already queried for (m', t'), where $t = t', m \neq t'$. The purpose of this is to allow the homomorphic evaluation of the sum in the exponent.

Assumption 1 *(LEMO Assumption) Let $\mathcal{G}$ be an algorithm that on input the security parameter κ outputs the parameters of a bilinear group as $\mathtt{G}$=$(\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$. For a set of $\{\mathcal{U}\}_{i=1}^{n}$ let $X = g_2^{\delta}, Y = g_2^{\sum_{i=1}^{n}\gamma_i} \in \mathbb{G}_2^2$ for $\delta, \gamma_i \in \mathbb{Z}_p$. Consider an oracle $\mathcal{O}_{\mathsf{LEMO}}$ that on input a tuple (i, t, x) responds with $(\alpha, \beta_t, \beta_t^{\gamma_i}\alpha^{x\delta})$ for a uniformly at random element $\alpha, \beta_t \in \mathbb{Z}_p$ iff for a t' that has already been queried with an i that was already part of a previous query $m = m'$ and the total queries for a t are less than n.*

Then for all probabilistic polynomial time adversaries $\mathcal{A}$ the probability:

$$\Pr[\mathtt{G} \leftarrow \mathcal{G}(1^{\kappa}); X = g_2^{\delta}, Y = g_2^{\sum_{i=1}^{n}\gamma_i} \mid \delta, \gamma_i \in \mathbb{Z}_p; (a, b, c) \leftarrow \mathcal{A}^{\mathcal{O}(i,t,x)}$$
$$\mid (t' = t \wedge x'_{u,t} = x_{u,t}) \wedge (t' = t \wedge |x_{u,t}| \leq n) \wedge a = \alpha \wedge b = \beta_t \wedge c = \beta_t^{\gamma_i}\alpha^{x\delta}] \leq \epsilon_2(\kappa)$$

We first show in our analysis that a **Type I Forgery** implies a break of the BLS signatures and next that a **Type II Forgery** implies a break of the LEMO assumption.

Theorem 2. *Our scheme achieves* Aggregate Unforgeability *for a* **Type I Forgery** *under* BCDH *assumption in the random oracle model.*

Theorem 3. *Our scheme guarantees* Aggregate Unforgeability *for a* **Type II Forgery** *under the* LEMO *assumption in the random oracle model.*

Due to space limitations the proof of Theorem 2 and 3 is deferred in Appendix B.

5.3 Performance Evaluation

In this section we analyze the extra overhead that is occured for the aggregate unforgeability property of our **PUA** instantiation scheme with respect to Shi *et al.* scheme [16] which is used for encryption. First we consider a theoretical evaluation with respect to the mathematical operations a participant of the protocol be it user, Aggregator or Data Analyzer has to perform with respect to the verifiability transcripts. That is, the computation of the tag by each user, the proof by the Aggregator and the verification of the proof by the Data Analyzer. We also present an experimental evaluation that shows the practicality of out scheme.

To allow the Data analyzer to verify the correctness of computations performed by an untrusted Aggregator each user selects uniformly and at random a secret key $\mathsf{tk}_i \in \mathbb{Z}_p$. The key dealer distributes to each user $g_1^a \in \mathbb{G}_1$ and publishes $g_2^a \in \mathbb{G}_2$, which calls for two exponentiations: one in $\mathbb{G}_1$ and one in $\mathbb{G}_2$. At each time interval t each user computes $\sigma_{i,t} = H(t)^{\mathsf{tk}_i}(g_1^a)^{x_{i,t}} \in \mathbb{G}_1$, which entails one hash evaluation , two exponentiations and one multiplication in $\mathbb{G}_1$. For the computation of the σ_{t} the Aggregator is involved in $n-1$ multiplications in $\mathbb{G}_1$: $\prod_{i=1}^{n}\sigma_{\mathsf{i,t}}$. Finally the data analyzer verifies by checking the equality: $e(\sigma_{\mathsf{t}}, g_2) \stackrel{?}{=} e(H(t), \mathsf{vk}_1)e(g_1^{\mathsf{sum_t}}, \mathsf{vk}_2)$, which asks for three pairing evaluations, one hash in $\mathbb{G}_1$, one exponentiation in $\mathbb{G}_1$ and one multiplication in $\mathbb{G}_T$ (see table 1). The efficiency of $\mathbf{PUA}$ stems from the constant time verification with respect to the size of the users. This is of crucial importance since the Data Analyzer may not own computational power. In contrast the Aggregator's proof is linear on the number users n, but since the Aggregator is modeled as a powerful machine this does not entails efficiency barriers.

Participant	Computation	Com.
User	$\mathbf{2}$ EXP $+\mathbf{1}$ MUL $+\mathbf{1}$HASH	$\mathbf{2} \cdot \mathbf{l}$
Aggregator	$(\mathbf{n}-\mathbf{1})$ MUL	$\mathbf{2} \cdot \mathbf{l}$
Data Analyzer	$\mathbf{3}$ PAIR $+\mathbf{1}$ EXP $+\mathbf{1}$ MUL $+\mathbf{1}$HASH	-

Table 1: Performance of tag computation, proof construction and verification operations. l denotes the bit-size of the prime number p.

Operation \ Pairings	MNT159	MNT201	MNT224
Tag	$\mathbf{1.2}$ ms	$\mathbf{1.8}$ ms	$\mathbf{2.2}$ ms
Verify	$\mathbf{28.3}$ ms	$\mathbf{42.7}$ ms	$\mathbf{53.5}$ ms

Table 2: Computational cost of $\mathbf{PUA}$ operations with respect to different pairings.

Operation \ Curve	MNT159	MNT201	MNT224
HASH in $\mathbb{G}_1$	$\mathbf{0.139}$ ms	$\mathbf{0.346}$ ms	$\mathbf{0.296}$ ms
HASH in $\mathbb{G}_2$	$\mathbf{25.667}$ ms	$\mathbf{41.628}$ ms	$\mathbf{48.305}$ ms
MUL in $\mathbb{G}_1$	$\mathbf{0.004}$ ms	$\mathbf{0.0006}$ ms	$\mathbf{0.006}$ ms
MUL in $\mathbb{G}_2$	$\mathbf{0.040}$ ms	$\mathbf{0.051}$ ms	$\mathbf{0.054}$ ms
MUL in $\mathbb{G}_T$	$\mathbf{0.012}$ ms	$\mathbf{0.015}$ ms	$\mathbf{0.016}$ ms
EXP in $\mathbb{G}_1$	$\mathbf{0.072}$ ms	$\mathbf{0.092}$ ms	$\mathbf{0.099}$ ms
EXP in $\mathbb{G}_2$	$\mathbf{0.615}$ ms	$\mathbf{0.757}$ ms	$\mathbf{0.784}$ ms
PAIR	$\mathbf{7.077}$ ms	$\mathbf{10.674}$ ms	$\mathbf{13.105}$ ms

Table 3: Average computation overhead of the underlying mathematical group operations for different type of curves.

We implemented the verification functionalities of $\mathbf{PUA}$ with the `Charm` cryptographic framework [1, 2]. For pairing computations it inherits the `PBC` [15] library which is also written in `C`. All of our benchmarks are executed on Intel Core i5 CPU M 560 @ 2.67GHz $\times$ 4 with 8GB of memory, running Ubuntu 12.04 32bit. `Charm` uses 3 types of asymmetric pairings: `MNT159`, `MNT201`, `MNT224`. We run our benchmarks with these three different types of asymmetric pairings. The timings for all the underlying mathematical group operations are summarized in table 3. There is a vast difference on the computation time of operations between $\mathbb{G}_1$ and $\mathbb{G}_2$ for all the different curves. The reason is the fact that the bit-length of elements in $\mathbb{G}_2$ is much larger than in $\mathbb{G}_1$.

As shown in table 2 the tag $\sigma_{i,t}$ computation implies a computation overhead at a scale of milliseconds with a gradual increase as the bit size of the underlying elliptic curve increases. The data analyzer is involved in pairing evaluations and computations at the target group independent of the size of the data-users.

6 Related Work

In [8], authors proposed a solution which is based on homomorphic message authenticators in order to verify the computation of generic functions on outsourced data. Each data input is authenticated with an authentication tag. A composition of the tags is being computed by the cloud in order to evaluate a program which takes as input a function f and a set of tags. Thanks to the homomorphic properties of the tags the user can verify the correctness of the program. The main drawback of the solution is that the user in order to verify the correctness of the computation has to be involved in computations that take exactly the same time as the computation of the function f. *Backes et al.* [3] proposed a generic solution for efficient verification of bounded degree polynomials in time less than the evaluation of f. The solution is based on *closed form efficient* pseudorandom function PRF. In a nutshell the idea of *closed form efficient* PRF which has been first introduced in [4] is the following: assume that the computation of a function $f(r_1, \ldots, r_n, d_1, \ldots, d_n)$ takes computational time t proportional to n. Then the knowledge of a key K for a pseudorandom function $\mathsf{F_K}$ can evaluates $f(r_1 = \mathsf{F_K}(\mathsf{L_1}), \ldots, \mathsf{r_n} = \mathsf{F_K}(\mathsf{L_n}), \mathsf{d_1}, \ldots, \mathsf{d_n})$ in time less than t. Contrary to our solution both solutions do not provide individual privacy. In the multi-user setting, Choi *et al.* [11] proposed a protocol in which multiple users are outsourcing their inputs to an untrusted server along with the definition of a functionality f. The server computes the result in a privacy preserving manner without learning the result and the computation is verified by a user that has contributed to the function input. The users are forced to operate in a *non-interactive* model, whereby they cannot communicate with each other. The underlying machinery entails a novel proxy based oblivious transfer protocol, which along with a fully homomorphic scheme and garbled circuits allows for verifiability and privacy.

Catalano *et al.* [10] employed a nifty technique to allow single users to verify computations on encrypted data. The idea is to re-randomize the ciphertext and sign it with a homomorphic signature. Computations then are performed on the randomized ciphertext and the original one. The aforementioned solution cannot be positioned with ours since the setting is different: The untrusted entity who performs the computations in $\mathbf{PUA}$ learns the result of the computations and also our solution is appropriately tailored for a multi-user scenario. Last but not least in [10] the aggregate value is not allowed to be learnt in cleartext by the untrusted aggregator since the protocols are geared for cloud based scenarios.

In [9] messages are modeled as the free terms of a polynomial. The signature is obtained by a first level encoding of the message. For additive function verification the signature can be obtained by multiplying two signatures. However the randomness produced at the first level of encoding does not allow for multiplicative function verification. The authors in [9] addressed the latter by creating a public version of the secret key and mapping the signature to another encoding. With the combination of the new public key and the graded encoding of the signature the randomness is cleared and a verifier can verify multiplicative functions over the messages. However the aforementioned scheme is tailored for single user scenario and is also inefficient due to its instantiation within the multi-linear maps framework since it is a generic construction not tailored for specific computations.

7 Concluding Remarks

In this paper we designed and analyzed a protocol for private and unforgeable aggregation. First we modeled its security and privacy requirements. In this setting a set of trustworthy users submits data coupled with unforgeable tags. The purpose of the protocol is to allow a data analyzer to verify the correctness of computation performed by a malicious Aggregator, without being able to discover the underlying data. The challenge of the verification in aggregation protocols that we tackled with the $\mathbf{PUA}$ protocol is the fact that the privacy from the authentication tags is guaranteed in a *non-interactive multi-user* setting. Our **PUA** instantiation allows for *public verifiability* in *constant* *time* and is provable secure under the DDH, BCDH and the new LEMO assumption in bilinear pairing groups in the random oracle model.

Bibliography

[1] J. A. Akinyele, M. Green, and A. D. Rubin. Charm: A tool for rapid cryptographic prototyping. http://www.charm-crypto.com/Main.html.

[2] J. A. Akinyele, M. Green, and A. D. Rubin. Charm: A framework for rapidly prototyping cryptosystems. *IACR Cryptology ePrint Archive*, 2011:617, 2011. http://eprint.iacr.org/2011/617.pdf.

[3] M. Backes, D. Fiore, and R. M. Reischuk. Verifiable delegation of computation on outsourced data. In *ACM Conference on Computer and Communications Security*, pages 863–874, 2013.

[4] S. Benabbas, R. Gennaro, and Y. Vahlis. Verifiable delegation of computation over large datasets. In *CRYPTO*, pages 111–131, 2011.

[5] D. Boneh and X. Boyen. Short signatures without random oracles and the sdh assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, 2008.

[6] D. Boneh, C. Gentry, B. Lynn, and H. Shacham. Aggregate and verifiably encrypted signatures from bilinear maps. In *EUROCRYPT*, pages 416–432, 2003.

[7] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International CryptologyConference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, pages 56–72, 2004.

[8] D. Catalano and D. Fiore. Practical homomorphic macs for arithmetic circuits. In *EUROCRYPT*, pages 336–352, 2013.

[9] D. Catalano, D. Fiore, and B. Warinschi. Homomorphic signatures with efficient verification for polynomial functions. In *Advances in Cryptology–CRYPTO 2014*, pages 371–389. Springer Berlin Heidelberg, 2014.

[10] D. Catalano, A. Marcedone, and O. Puglisi. Authenticating computation on groups: New homomorphic primitives and applications. In *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*, pages 193–212, 2014.

[11] S. G. Choi, J. Katz, R. Kumaresan, and C. Cid. Multi-client non-interactive verifiable computation. In *Proceedings of the 10th Theory of Cryptography Conference on Theory of Cryptography*, TCC'13, pages 499–518, Berlin, Heidelberg, 2013. Springer-Verlag.

[12] D. M. Freeman. Improved security for linearly homomorphic signatures: A generic framework. In *Public Key Cryptography - PKC 2012 - 15th International Conference on Practice and Theory in Public Key Cryptography, Darmstadt, Germany, May 21-23, 2012. Proceedings*, pages 697–714, 2012.

[13] M. Joye and B. Libert. A scalable scheme for privacy-preserving aggregation of time-series data. In *Financial Cryptography*, 2013.

[14] I. Leontiadis, K. Elkhiyaoui, and R. Molva. Private and dynamic time-series data aggregation with trust relaxation. In *Cryptology and Network Security - 13th*

International Conference, CANS 2014, Heraklion, Crete, Greece, October 22-24, 2014. Proceedings, pages 305–320, 2014.
[15] B. Lynn. The stanford pairing based crypto library. `http://crypto.stanford.edu/pbc`.
[16] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *NDSS*, 2011.
[17] V. Shoup. Lower bounds for discrete logarithms and related problems. In *Advances in Cryptology - EUROCRYPT '97, International Conference on the Theory and Application of Cryptographic Techniques, Konstanz, Germany, May 11-15, 1997, Proceeding*, pages 256–266, 1997.

A Obliviousness

Theorem 1. *The proposed solution achieves obliviousness in the random oracle model under the decisional Diffie-Hellman (DDH) assumption in* $\mathbb{G}_1$.

Proof. Assume there is an aggregator $\mathcal{A}$ which breaks the obliviousness of the **PUA** scheme with a non-negligible advantage ϵ. We build in what follows an adversary $\mathcal{B}$ who uses $\mathcal{A}$ as a subroutine to break the aggregator obliviousness of the private streaming aggregation (PSA) protocol presented in [16], which is guaranteed under DDH. Without loss of generality we call the oracles that the adversary $\mathcal{B}$ has access to from the PSA scheme as follows: $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{Setup}}$, $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{Corrupt}}$, $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{Encrypt}}$, and $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{AO}}$.

We consider in PSA as in **PUA** that there are n users $\mathcal{U}_i$ and each one of these users possesses a secret encryption key $\mathsf{ek_i}$. In the following, we show how an adversary $\mathcal{B}$ simulates the aggregator obliviousness game presented in Algorithms 1 and 2 to aggregator $\mathcal{A}$ and how therewith breaks the aggregator obliviousness of PSA.

Learning phase: In the learning phase, adversary $\mathcal{B}$ proceeds as following: Whenever $\mathcal{A}$ calls oracle $\mathcal{O}_{\mathsf{Setup}}$ with a security parameter κ, $\mathcal{B}$ queries oracle $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{Setup}}$ with the same security parameter. Oracle $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{Setup}}$ in turn outputs the public parameters that are composed of a hash function $H : \{0,1\}^* \to \mathbb{G}_1$, a generator g_1 of the group $\mathbb{G}_1$ of safe prime order p, and the aggregator's secret key $\mathsf{SK_A} = -\sum_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{ek_i}$. $\mathcal{B}$ then selects the parameters of a bilinear pairing $e : (g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$. $\mathcal{B}$ chooses uniformly at random $a, \{r_i\}_{\mathcal{U}_i \in \mathcal{U}}$ such and defines the verification key VK as follows:

$$\mathsf{VK} = (g_2^{a\mathsf{SK_A} + \sum_{i=1}^{n} \mathsf{r_i}}, g_2^a) \quad = (g_2^{a\sum_{i=1}^{n} \mathsf{ek_i} + \sum_{i=1}^{n} \mathsf{r_i}}, g_2^a) = (g_2^{\sum_{i=1}^{n} a\mathsf{ek}_i + r_i}, g_2^a)$$

This entails that tk_i is defined as: $a\mathsf{ek}_i + r_i$. Finally $\mathcal{B}$ forwards to $\mathcal{A}$ the public parameters:$\mathcal{P} = (H, p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$, the verification keys $\mathsf{VK} = (g_2^{\sum_{i=1}^{n} \mathsf{tk_i}}, g_2^a)$ and the secret key of the Aggregator $\mathsf{sk_A}$.

Whenever $\mathcal{A}$ calls oracle $\mathcal{O}_{\mathsf{Corrupt}}$ with a user's identifier uid_i, $\mathcal{B}$ relays the query uid_i to $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{Corrupt}}$ of the PSA scheme which in turns outputs the secret encryption key $\mathsf{ek_i}$ of user $\mathcal{U}_i$. $\mathcal{B}$ then returns secret key $\mathsf{SK}_i = (\mathsf{ek}_i, \mathsf{tk}_i) = (\mathsf{ek}_i, a\mathsf{ek}_i + r_i)$.

Whenever $\mathcal{A}$ calls oracle $\mathcal{O}_{\mathsf{EncTag}}$ with query $(t, \mathsf{uid}_i, x_{i,t})$, $\mathcal{B}$ forwards the query to the $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{Encrypt}}$ oracle which returns the appropriate ciphertext $c_{i,t} = H(t)^{\mathsf{ek_i}} g_1^{x_{i,t}}$. $\mathcal{B}$ computes then the tag associated with ciphertext $c_{i,t}$ as $\sigma_{i,t} = (c_{i,t})^a H(t)^{r_i} = H(t)^{a\mathsf{ek_i}+\mathsf{r_i}} g_1^{ax_{i,t}} = H(t)^{\mathsf{tk}_i} g_1^{ax_{i,t}}$ and transmits to $\mathcal{A}$ ciphertext $c_{i,t}$ and tag $\sigma_{i,t}$.

Challenge phase: In the challenge phase $\mathcal{A}$ chooses a set of users $\mathbb{S}^*$ that have not been corrupted during the learning phase and a time interval t^* for which $\mathcal{A}$ did not make a query to oracle $\mathcal{O}_{\mathsf{EncTag}}$. $\mathcal{A}$ then submits two time-series $\mathcal{X}_0^* = (\mathcal{U}_i, t^*, x^0_{i,t^*})_{\mathcal{U}_i \in \mathbb{S}^*}$ and $\mathcal{X}_1^* = (\mathcal{U}_i, t^*, x^1_{i,t^*})_{\mathcal{U}_i \in \mathbb{S}^*}$ such that $\sum x^0_{i,t^*} = \sum x^1_{i,t^*}$ to $\mathcal{O}_{\mathsf{AO}}$ oracle. $\mathcal{B}$ simulates this oracle as follows:

It forwards the series $\mathcal{X}_0^*$ and $\mathcal{X}_1^*$ to $\mathcal{O}^{\mathsf{PSA}}_{\mathsf{AO}}$ which chooses uniformly at random a bit $b \overset{\$}{\leftarrow} \{0,1\}$ and returns to $\mathcal{B}$ the ciphertexts $\{c^b_{i,t^*}\}_{\mathcal{U}_i \in \mathbb{S}^*}$ encrypting time-serie $\mathcal{X}_b^*$.

Next, $\mathcal{B}$ constructs for all $\mathcal{U}_i$ in $\mathbb{S}^*$ the tag σ^b_{i,t^*} corresponding to ciphertext c^b_{i,t^*} by computing:

$$\sigma^b_{i,t^*} = (c^b_{i,t})^a H(t^*)^{r_i} = (H(t^*)^{\mathsf{ek}_i} g_1^{x^b_{i,t^*}})^a H(t^*)^{r_i} = H(t^*)^{a\mathsf{ek}_i + r_i} g_1^{ax^b_{i,t^*}} = H(t^*)^{\mathsf{tk}_i} g_1^{ax^b_{i,t^*}}$$

Note that σ^b_{i,t^*} corresponds to a correctly computed tag for input x^b_{i,t^*}. Finally, $\mathcal{B}$ forwards to $\mathcal{A}$ $\{(c^b_{i,t^*}, \sigma^b_{i,t^*}\}_{\mathcal{U}_i \in \mathbb{S}^*}$. At this point, the simulated view of aggregator $\mathcal{A}$ is computationally indistinguishable to its view in an actual *aggregator obliviousness* game as defined in Algorithms 1 and 2. This leads to correct verification of the sum computed by $\mathcal{A}$, more precisely:

$$e(\prod_{i \in \mathbb{S}^*} \sigma^b_{i,t^*}, g_2) = e(\prod_{i=1}^{n} H(t^*)^{\mathsf{tk}_i} g_1^{ax^b_{i,t^*}}, g_2) =$$

$$e(H(t^*), g_2^{a\sum_{i=1}^n \mathsf{ek}_i + \sum_{i=1}^n r_i}) e(g_1^{\sum_{i=1}^n x^b_{i,t^*}}, g_2^a) = e(H(t^*), \mathsf{vk}_1) e(g_1^{\sum_{i=1}^n x^b_{i,t^*}}, \mathsf{vk}_2)$$

It follows that if aggregator $\mathcal{A}$ is able to output a correct guess b^* for the bit b with a non-negligible advantage ϵ (i.e. is able to break the aggregator obliviousness of our scheme), then $\mathcal{B}$ will break the aggregator obliviousness of the PSA scheme with same non-negligible advantage ϵ by outputting the guess b^*.

As such PSA scheme ensures aggregator obliviousness under the DDH assumption in $\mathbb{G}_1$, we can conclude that our scheme also ensures aggregator obliviousness as long as DDH holds in $\mathbb{G}_1$.

B *Aggregate Unforgeability*

Theorem 2. *Our scheme achieves* Aggregate Unforgeability *for a* **Type I Forgery** *under* BCDH *assumption in the random oracle model.*

Proof. We show how to build an attacker $\mathcal{B}$ that solves BCDH in $(\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$. Let g_1 and g_2 be two generators for $\mathbb{G}_1$ and $\mathbb{G}_2$ respectively. $\mathcal{B}$ receives the challenge $(g_1, g_2, g_1^a, g_1^b, g_1^c, g_2^a, g_2^b)$ from the BCDH oracle $\mathcal{O}_{\mathsf{BCDH}}$ and is asked to output

$e(g_1, g_2)^{abc} \in \mathbb{G}_T$. $\mathcal{B}$ simulates the interaction with $\mathcal{A}$ in the two phases (**Setup, Learning**) as follows:
Setup:

- To simulate the $\mathcal{O}^{\mathcal{A}}_{\mathsf{Setup}}$ oracle $\mathcal{B}$ selects uniformly at random $2n$ keys $\{\mathsf{k}_i\}_{i=1}^n$, $\{\mathsf{y}_i\}_{i=1}^n \in \mathbb{Z}_p$ and outputs the public parameters $\mathcal{P} = (\kappa, p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2)$ the verification key $\mathsf{VK} = (\mathsf{vk}_1, \mathsf{vk}_2) = (g_2^{b\sum_{i=1}^n \mathsf{k}_i}, g_2^a)$ and the secret key of the Aggregator $\mathsf{sk_A} = -\sum_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{y_i}$.

Learning phase

- $\mathcal{A}$ is allowed to query the random oracle H for any time interval . $\mathcal{B}$ constructs a $\mathtt{H}-\mathtt{list}$ and responds to $\mathcal{A}$ query as follows:
  1. If query (t) already appears in a tuple H-tuple$\langle t : r_t, \mathsf{coin}(t), H(t)\rangle$ of the $\mathtt{H}-\mathtt{list}$ it responds to $\mathcal{A}$ with $H(t)$.
  2. Otherwise it selects a random number $r_t \in \mathbb{Z}_p$ and flips a random $\mathsf{coin} \xleftarrow{\$} \{0, 1\}$. With probability p, $\mathsf{coin(t)} = 0$ and $\mathcal{B}$ answers with $H(t) = g_1^{r_t}$. Otherwise if $\mathsf{coin(t)} = 1$ then $\mathcal{B}$ responds with $H(t) = g_1^{cr_t}$ and updates the $\mathtt{H}-\mathtt{list}$ with the new tuple H-tuple$\langle t : r_t, \mathsf{coin}(t), H(t)\rangle$.
- Whenever $\mathcal{A}$ submits a query $(t, \mathsf{uid_i}, \mathsf{x_{i,t}})$ to the $\mathcal{O}^{\mathcal{A}}_{\mathsf{EncTag}}$, $\mathcal{B}$ constructs a $\mathtt{T}-\mathtt{list}$ and responds as follows:
  1. If a tuple with equivalent uid_i with this of the query exists in the $\mathtt{T}-\mathtt{list}$ then $\mathcal{B}$ aborts. Otherwise:
  2. If at time interval t $\mathcal{A}$ has never queried before the $\mathcal{O}^{\mathcal{A}}_{\mathsf{EncTag}}$ oracle then:
     (a) $\mathcal{B}$ initializes a counter $\mathtt{cnt}_t = 1$ and a variable $\Sigma_t = 0$.
     (b) $\mathcal{B}$ calls the simulated random oracle, receives the result for H(t) and appends the tuple H-tuple$\langle t : r_t, \mathsf{coin}(t), H(t)\rangle$ to the $\mathtt{H}-\mathtt{list}$.
     (c) If $\mathsf{coin}(t) = 1$ then $\mathcal{B}$ stops the simulation.
     (d) Otherwise it chooses the secret tag key k_i where $i = \mathsf{uid_i}$ to be used as secret tag key from the the set of $\{\mathsf{k}_i\}$ keys, chosen by $\mathcal{B}$ in the Setup phase.
     (e) $\mathcal{B}$ sends to $\mathcal{A}$ the tag $\sigma_{i,t} = g_1^{r_t b k_i} g_1^{a x_{i,t}} = H(t)^{b k_i} g_1^{a x_{i,t}}$, which is a valid tag for the value $x_{i,t}$. Notice that $\mathcal{B}$ can correctly compute the tag without knowing a and b from the BCDH problem parameters g_1^a, g_1^b.
     (f) $\mathcal{B}$ chooses also a secret encryption key $\mathsf{y}_i \in \{\mathsf{y}_i\}_{i=1}^n \in \mathbb{Z}_p$ and computes the ciphertext as $c_{i,t} = H(t)^{\mathsf{y}_i} g_1^{x_{i,t}}$. The simulation is correct since $\mathcal{A}$ can check that the sum $\sum_{i=1}^n x_{i,t}$ corresponds to the ciphertexts given by $\mathcal{B}$ with its decryption key $\mathsf{sk_A} = -\sum_{\mathsf{i=1}}^{\mathsf{n}} \mathsf{y_i}$, considering the attacker has made distinct encryption queries for all the n users in the scheme at a time interval t.
     (g) $\mathcal{B}$ sets $\Sigma_t = \Sigma_t + x_{i,t}$ and updates the $\mathtt{T}-\mathtt{list}$ with the tuple: $\langle t, \mathsf{uid}_i, x_{i,t}, \sigma_{i,t}\rangle$
  3. If $\mathtt{cnt_t} = n$ $\mathcal{B}$ aborts.
  4. Otherwise $(0 < \mathtt{cnt_t} < n)$, $\mathcal{B}$ looks to the $\mathtt{H}-\mathtt{list}$ list for the tuple indexed by t in order to get $\langle t : r_t, \mathsf{coin}(t), H(t)\rangle$. If the tuple does not exist then $\mathcal{B}$ makes the call to the random oracle and if $\mathsf{coin}(t) = 1$ then $\mathcal{B}$ aborts.

5. If $\mathsf{coin}(t) = 0$ then $\mathcal{B}$ computes the tag identically as in 1(d)(e)(f)(g) steps: It chooses a key k_i where $i = \mathsf{uid_i}$ from the selected keys $\{\mathsf{k}_i\}$. It constructs the tag as $\sigma_{i,t} = g_1^{r_t b k_i} g_1^{a x_{i,t}} = H(t)^{bk_i} g_1^{a x_{i,t}}$, the ciphertext as $c_{i,t} = H(t)^{\mathsf{y}_i} g_1^{x_{i,t}}$. Finally $\mathcal{B}$ sets $\Sigma_t = \Sigma_t + x_{i,t}$, updates the $\mathtt{T-list}$ with the tuple: $\langle t, \mathsf{uid}_i, x_{i,t}, \sigma_{i,t}\rangle$ and increases the counter $\mathtt{cnt}_t = \mathtt{cnt}_t + 1$.

Now, when $\mathcal{B}$ receives the forgery $(\mathsf{sum_t}', \sigma_\mathsf{t}')$ at time interval $t = t'$, it continues if $\mathsf{sum_t}' \neq \Sigma_t$. $\mathcal{B}$ first queries the H-tuple for time t' in order to fetch the appropriate tuple.

- If $\mathsf{coin}(t') = 0$ then $\mathcal{B}$ aborts.
- If $\mathsf{coin}(t') = 1$ then since $\mathcal{A}_{U_1}$ output a valid forged σ_t' at t', it should be true that the following equation should hold:

$$e(\sigma_\mathsf{t}', g_2) = e(H(t'), \mathsf{vk}_1) e(g_1^{\mathsf{sum_t}'}, \mathsf{vk}_2)$$

which is true when $\mathcal{A}_{U_1}$ makes n queries for time interval t' for distinct users to the $\mathcal{O}^{\mathcal{A}}_{\mathsf{Tag}}$ oracle during the Learning phase. As such $\sigma_\mathsf{t}' = g_1^{c r_t b \sum \mathsf{k}_i} g_1^{a \mathsf{sum_t}'}$
Finally $\mathcal{B}$ outputs:

$$\begin{aligned} e\left(\left(\frac{\sigma_\mathsf{t}'}{g_1^{a\mathsf{sum_t}'}}\right)^{\frac{1}{r_t \sum \mathsf{k}_i}}, g_2^a\right) &= e\left(\left(\frac{g_1^{c r_t b \sum \mathsf{k}_i} g_1^{a \mathsf{sum_t}'}}{g_1^{a\mathsf{sum_t}'}}\right)^{\frac{1}{r_t \sum \mathsf{k}_i}}, g_2^a\right) \\ &= e\left((g_1^{c r_t b \sum \mathsf{k}_i})^{\frac{1}{r_t \sum \mathsf{k}_i}}, g_2^a\right) = e(g_1^{bc}, g_2^a) = e(g_1, g_2)^{abc} \end{aligned}$$

Let $\mathcal{A}^{\mathbf{AU1}}$ the event when $\mathcal{A}$ successfully forges a a type I forgery σ_t for our **PUA** protocol that happens with some non-negligible probability ϵ'. Then $\Pr[\mathcal{B}^{\mathsf{BCDH}}] = \Pr[\mathtt{event}_0]\Pr[\mathtt{event}_1]\Pr[\mathcal{A}^{\mathbf{AU2}}] = p(1-p)^{\mathtt{q_H}-1}\epsilon'$, for $\mathtt{q_H}$ random oracle queries with the probability $\Pr[\mathsf{coin}(t) = 0] = p$. As such we ended up in a contradiction assuming the hardness of the BCDH and finally $\Pr[\mathcal{A}^{\mathbf{AU1}}] = \epsilon_1$ for negligible a ϵ_1 function.

Theorem 3. *Our scheme guarantees* Aggregate Unforgeability *for a* **Type II Forgery** *under the* LEMO *assumption in the random oracle model.*

Proof. The $\mathcal{O}^{\mathcal{A}}_{\mathsf{EncTag}}$ oracle behaves equivalently as the oracle in the LEMO assumption. $\mathcal{B}$ chooses secret encryptions keys $\{\mathsf{ek}_i\}_{i=1}^n$ and sends to $\mathcal{A}$ the secret decryption key $\mathsf{sk}_A = \sum_{i=1}^n \mathsf{ek}_i$. For queries $(i = \mathsf{uid}, x, t)$ to the $\mathcal{O}^{\mathcal{A}}_{\mathsf{EncTag}}$ oracle the simulator $\mathcal{B}$ returns the responses $c = \beta_t^{\gamma_i}\alpha^{x\delta}$ from the $\mathcal{O}(i, t, x)$ oracle for a random message x as a tag and constructs the ciphertext as $c_{i,t} = H(t)^{\mathsf{ek}_i} g_1^{x_{i,t}}$. For a random oracle query H(t) the simulator $\mathcal{B}$ queries the $\mathcal{O}_{\mathsf{LEMO}}$ with input $(j \ni \mathcal{U}, t, x \xleftarrow{\$} \mathbb{Z}_p)$ which replies with $(a = \alpha \wedge b = \beta_t \wedge c = \beta_t^{\gamma_i}\alpha^{x\delta})$ and forwards to $\mathcal{A}$ $b = \beta_t$. Thus the probability of $\mathcal{A}$ to output a **Type II Forgery** is $\Pr[\mathcal{A}^{\mathbf{AU2}}] \leq \epsilon_2$ for a negligible function ϵ_2 and our scheme guarantees *Aggregate Unforgeability* for a **Type II Forgery** under the LEMO assumption in the random oracle model.

To conclude with the analysis the success probabilities for the *Aggregate Unforgeability* game $\Pr[\mathcal{A}^{\mathbf{AU}}]$, are taken over the union of the success probabilities for the two type of forgeries. As such

$$\Pr[\mathcal{A}^{\mathbf{AU}}] = \Pr[\mathcal{A}^{\mathbf{AU1}}] + \Pr[\mathcal{A}^{\mathbf{AU2}}] \leq \epsilon_1(\kappa) + \epsilon_2(\kappa)$$

where ϵ_1 and ϵ_2 are negligible functions.

C Security evidence for the LEMO Assumption

In this section we provide security evidence for the hardness of the new LEMO assumption by presenting lower bounds on the success probabilities of an adversary $\mathcal{A}$ who presumably breaks the assumption. We follow the theoretical *generic group model* (GGM) as presented in [17]. Namely under the GGM framework an adversary $\mathcal{A}$ has access to a black box that conceptualizes the underlying mathematical group $\mathbb{G}$ that the assumption takes place. $\mathcal{A}$ without knowing any details about the underlying group apart from its order p is asking for encodings of its choice and the black box replies through a random encoding function ξ that maps elements from to $\mathbb{G} \to \Xi$ as random bit strings of size $\lceil \log_2 q \rceil$. Since our construction operates on asymmetric bilinear pairing groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ we make use of three random encoding functions $\xi_c, c \in [1, 2, T]$ where $\xi_c : \mathbb{G}_c \to \{0, 1\}^{\lceil \log_2 q \rceil}$.

Theorem 4. *Suppose $\mathcal{A}$ is a polynomial probabilistic time adversary that solves the* LEMO *assumption, making at most q_G oracle queries for the underlying group operations on $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ and the $\mathcal{O}(i, t, m)$ oracle, all counted together. All the encodings $\xi_c, c \in [1, 2, T]$ and $\delta, \{\gamma_u\}_{u=1}^n \in \mathbb{Z}_p$ are chosen at random. Then the probability ϵ_2 that $\mathcal{A}$ on input $(p, \xi_1(g_1), \xi_2(g_2), \xi_2(a), \xi_2(b), \xi_2(c), \xi_2(\delta), \xi_2(\sum_{i=1}^n \gamma_i))$ to output $\xi_1(a, b, c = \xi_1(\beta_t \sum_{u=1}^n \gamma_u + \alpha\delta \sum_{u=1}^n x_{u,t}))$ for which neither exists a query for $u' = u, t' = t \wedge x_{u',t'} \neq x_{u,t}$ nor $\mathcal{A}$ has made more than n distinct queries for a fixed time interval t, is bounded as:*

$$\epsilon_2 \leq \frac{(q_G + 12)^2}{p}$$

Proof. We assume a polynomial time simulator $\mathcal{B}$ that interacts with adversary $\mathcal{A}$ and simulates the black box for the underlying groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$. $\mathcal{B}$ maintains 3 lists of tuples:

- $\mathcal{L}_1 = \{(F_{1,i}, \xi_{1,i}) : i = 1, \cdots, \tau_1\}$
- $\mathcal{L}_2 = \{(F_{2,i}, \xi_{2,i}) : i = 1, \cdots, \tau_2\}$
- $\mathcal{L}_T = \{(F_{T,i}, \xi_{T,i}) : i = 1, \cdots, \tau_T\}$

where $F_{1,i}, F_{2,i} \in \mathbb{Z}_p[A, B, \{\Gamma_u\}_{u=1}^n, \Delta, X]$ are multivariate polynomial on the indeterminants $A, B, \{\Gamma_u\}_{u=1}^n, \Delta, X$ of degree 1 and $F_{T,i}$a multivariate polynomial of degree at most 2. The random encodings $\xi_{c,i}, c \in [1, 2, T]$ of each list $\mathcal{L}_c, c \in [1, 2, T]$ are provided to the adversary $\mathcal{A}$ at each step τ, where $\tau = \tau_1 + \tau_2 + \tau_T + 4$. The lists are initialized at step $\tau = 0$ by setting $\tau_1 = 1, \tau_2 = 3, \tau_T = 0$ and assigning $F_{1,1} = g_1, F_{2,1} = g_2, F_{2,2} = g_2^{\sum_{u=1}^n \gamma_u}, F_{2,3} = g_2^\delta$ that corresponds to the generators $g_1.g_2$ and the public information $g_2^{\sum_{u=1}^n \gamma_u}, g_2^\delta$. $\mathcal{A}$ has access to the random encodings $\xi_{1,1}, \xi_{2,1}, \xi_{2,2}, \xi_{2,3}$ respectively.

In what follows we describe how $\mathcal{B}$ simulates the groups operations in $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ and the oracle responses to $\mathcal{O}_{\mathsf{LEMO}}$. We first assume that before $\mathcal{A}$ queries the oracle or asks for group operations it has already asked for the random encodings of the involved elements of the operations. Consequently when $\mathcal{A}$ asks for operations in $\mathbb{G}_c, c \in [1, 2, T]$ for some operands $\xi_c, c \in [1, 2, T]$, $\mathcal{B}$ checks if $\xi_c, c \in [1, 2, T]$ already exists in $\mathcal{L}_c, c \in [1, 2, T]$ and aborts if this happens.

- **Group operations:** $\mathcal{A}$ provides $\mathcal{B}$ two operands $\xi_{c,1}, \xi_{c,2}, c \in [1, 2, T]$ and a bit defining multiplication or division. $\mathcal{B}$ starts by incrementing $\tau_c+ = 1, c \in [1, 2, T]$. It the computes $F_{1,\tau_c} = F_{1,i} + F_{1,j}$, where $1 \leq i, j \leq \tau_c$ if the operation bit is for multiplication or $F_{c,\tau_c} = F_{1,i} - F_{1,j}$, where $1 \leq i, j \leq \tau_c$ if it is for division. If the new polynomial F_{c,τ_c} is equal with another polynomial $F_{c,l}$ for some $l \leq \tau_c, c \in [1, 2, T]$ in list $\mathcal{L}_c, c \in [1, 2, T]$ then $\mathcal{B}$ fetches the corresponding $\xi_{c,l}$ and forwards it to $\mathcal{A}$, otherwise it choses a fresh random $\xi_{c,\tau_c} \in \{0, 1\}^{\log_2 q}$ and gives it to $\mathcal{A}$. $\mathcal{B}$ finally appends to $\mathcal{L}_c, c \in [1, 2, T]$ the pair $(F_{c,\tau_c}, \xi_{c,\tau_c}), c \in [1, 2, T]$.
- **Pairing:** A pairing operation in $\mathcal{G}_T$ consists of two random encodings $\xi_{1,i}, \xi_{2,j}$ with $1 \leq i \leq \tau_1$ and $1 \leq j \leq \tau_2$. $\mathcal{B}$ first increments the counter $\tau_T+ = 1$. Afterwards it computes the pairing as the multiplication of the appropriate polynomials: $F_{T,\tau_T} = F_{1,\tau_1} \cdot F_{2,\tau_2}$. If the same polynomial already exists in $\mathcal{L}_T$: $F_{T,\tau_T} = F_{T,l}, 1 \leq l \leq \tau_T$ then $\mathcal{B}$ clones the random string $\xi_{T,l}$, otherwise it choses a fresh random $\xi_{T,\tau_T} \in \{0, 1\}^{\log_2 q}$ and gives it to $\mathcal{A}$. $\mathcal{B}$ finally appends to $\mathcal{L}_T$ the pair $(F_{T,\tau_T}, \xi_{T,\tau_T})$.
- $\mathcal{O}_{\mathsf{LEMO}}$**:** $\mathcal{B}$ increments a counter $\tau_{\mathcal{O}}$ by 1 and sets $\tau_1+ = 3$. $\mathcal{A}$ inputs $(u, t, x_{u,t})$. $\mathcal{B}$ keeps track of three lists: $\mathcal{U}$-list, $\mathcal{T}$-list and $\mathcal{M}$-list where $(u, t, x_{u,t})$ are respectively stored. $\mathcal{B}$ stops the simulation if:
  - $(t' = t \wedge x'_{u,t} \neq x_{u,t}) \vee$
  - $(t' = t \wedge |x_{u,t}| > n)$

  where $|x_{u,t}|$ denotes the cardinality of queries obtained at time interval t. If the conditions do not hold then $\mathcal{B}$ computes the polynomials $F_{1,\tau_{1-2}} = A_t, F_{1,\tau_{1-1}} = A_t(Y), F_{1,\tau_1} = (B\Gamma_u + A\Delta X)$ for the indeterminants $B, \Gamma_u, A, \Delta, X$. If any of the $F_{1,\tau_{1-2}}, F_{1,\tau_{1-1}}, F_{1,\tau_1}$ already exist in $\mathcal{L}_1$ then $\mathcal{B}$ clones the associated random encodings $\xi_{1,l}$ for some $l \in [1, \cdots, \tau_1]$. Otherwise it creates three random encodings $\xi_{1,\tau_{1-2}}, \xi_{1,\tau_{1-1}}, \xi_{1,\tau_1} \in \{0, 1\}^{\log_2 q}$ and forwards them to $\mathcal{A}$. It also stores the pairs $(F_{1,\tau_{1-2}}, \xi_{1,\tau_{1-2}}), (F_{1,\tau_{1-1}}, \xi_{1,\tau_{1-1}}), (F_{1,\tau_1}, \xi_{1,\tau_1})$ in $\mathcal{L}_1$ list.

Eventually $\mathcal{A}$ outputs a forgery $(m_f, \xi_{1,fa}, \xi_{1,fy}, \xi_{1,fxy})$.Let $F_{1,fa}, F_{1,fx}, F_{1,fxy}$ be the corresponding polynomials in $\mathcal{L}_1$ list. If $\mathcal{A}$'s forgery is valid then it must hold:

$$F_f = \prod c_t \cdot F_{2,1} - \beta_t \cdot F_{2,2} - F_{1,1} \sum x_u \cdot F_{2,3} = 0 \tag{1}$$

which corresponds to $e(\prod c_t, g_2) - e(\beta_t, g_2^{\sum_{u=1}^n \gamma_u}) e(a^{\sum_{u=1}^n m_u}, g_2^{\delta}) = 0 \in \mathbb{G}_T$. We show now that this does not happen always. Indeed w.l.g we have the following form for each polynomial in the three lists:

- $F_{1,i} = z_{0,i} + z_{1,i} h\Gamma_{u,i} + z_{2,i} A\Delta X$
- $F_{2,i} = w_{0,i} + w_{1,i}\Delta + w_{2,i}E$
- $F_{T,i} = y_{0,i} + \eta_{1,i}\Delta h\Gamma_{u,i} + \eta_{2,i} E h\Gamma_{u,i} + \rho_{1,i} A\Delta^2 X + \rho_{2,i} A\Delta X E$

Equation (1) following the aforementioned presentation of each polynomial can be rewritten as

$$F_f = F_{T,k} - F_{T,l} F_{T,o} \tag{2}$$

for indexes k, l, o. Simplifying the equation, since it is equal to 0, then the second part consists of a polynomial with determinants $(\Delta\Gamma)^2, (E\Gamma)^2, A\Delta^4X^2, (A\Delta XE)^2$ and the first part with determinants $(\Delta\Gamma, E\Gamma, A\Delta^2X, A\Delta XE)$. Since there are no common terms, then all are canceled out and we are left with $y_{0,k} = y_{0,l}y_{0,o}$. As such $F_f = 0$ only when $y_{0,k} = y_{0,l}y_{0,o}$ or two polynomials in the same list evaluate at the same value even if they are different.

$\mathcal{B}$ assigns random values $(\alpha, \beta, \gamma, \delta, x)$ for the indeterminants A, B, Γ, Δ, X and in order for $\mathcal{A}$ to win in the game $\mathcal{A}$ should find two identical polynomial in any of the lists $\mathcal{L}_1, \mathcal{L}_2, \mathcal{L}_T$ or $F_f = 0$. As such the success probability of $\mathcal{A}$ is bounded by the probability that one at least of the following equations holds:

1. $F_{1,i}(\alpha, \beta, \gamma, \delta, x) - F_{1,j}(\alpha, \beta, \gamma, \delta, x) = 0 : i \neq j$
2. $F_{2,i}(\alpha, \beta, \gamma, \delta, x) - F_{2,j}(\alpha, \beta, \gamma, \delta, x) = 0 : i \neq j$
3. $F_{T,i}(\alpha, \beta, \gamma, \delta, x) - F_{T,j}(\alpha, \beta, \gamma, \delta, x) = 0 : i \neq j$
4. $F_{f,i}(\alpha, \beta, \gamma, \delta, x) - F_{f,j}(\alpha, \beta, \gamma, \delta, x) = 0 : i \neq j$

$F_{1,i}, F_{2,i}$ are of degree 1 as such they vanishe with probability at most $\frac{1}{p}$ and $F_{T,i}$ of degree 2, which respectively vanishes with probability at most $\frac{2}{p}$ from the Schwartz-Zippel theorem. As such summing for all possible pairs i, j for each of the aforementioned polynomials the success probability of $\mathcal{A}$ is bounded by:

$$\epsilon_2 \leq \binom{\tau_1}{2}\frac{1}{p} + \binom{\tau_2}{2}\frac{1}{p} + \binom{\tau_T}{2}\frac{2}{p} + \frac{4}{p} \leq \frac{(\tau_1 + \tau_2 + \tau_T + 8)^2}{p}$$

As $\tau_1 + \tau_2 + \tau_T \leq q_G + 4$ then $\epsilon_2 \leq \frac{(q_G+12)^2}{p}$