

# PUA – Privacy and Unforgeability for Aggregation

Iraklis Leontiadis, Kaoutar Elkhiyaoui, Refik Molva, Melek Önen

EURECOM, Sophia Antipolis, France  
{firstname.lastname}@eurecom.fr

**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.* [17] 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 (PUA)*. We also instantiate a 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 correctness 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, 13] 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 [17] 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 correctness of the aggregation thanks to the public verification key.

## 2.1 PUA Model

A PUA scheme consists of the following algorithms:

- **Setup**( $1^\kappa$ )  $\rightarrow$  ( $\mathcal{P}, sk_A, \{SK_i\}_{\mathcal{U}_i \in \mathbb{U}}, VK$ ): It is a randomized algorithm run by a trusted dealer which on input of a security parameter  $\kappa$  outputs the public parameters  $\mathcal{P}$  that will be used by subsequent algorithms, the aggregator  $\mathcal{A}$ ’s secret key  $sk_A$ , the secret keys  $SK_i$  of users  $\mathcal{U}_i$  and the public verification key  $VK$ .
- **EncTag**( $t, SK_i, x_{i,t}$ )  $\rightarrow$  ( $c_{i,t}, \sigma_{i,t}$ ): 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**( $sk_A, \{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$ )  $\rightarrow$  ( $sum_t, \sigma_t$ ): It is a deterministic algorithm run by the aggregator  $\mathcal{A}$ . It takes as inputs aggregator  $\mathcal{A}$ ’s secret key  $sk_A$ , 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 plaintext the sum  $sum_t$  of the values  $\{x_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$  and computes a proof  $\sigma_t$  assessing the correctness of  $sum_t$  using the authentication tags  $\{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$ .
- **Verify**( $VK, sum_t, \sigma_t$ )  $\rightarrow$   $\{0, 1\}$ : 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  $sum_t = \sum_{\mathcal{U}_i \in \mathbb{U}} \{x_{i,t}\}$ ; 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 [14, 17], 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  $\sigma_t$  for a sum  $\text{sum}_t$  that was not computed correctly from users' inputs (i.e. cannot generate a proof for  $\text{sum}_t \neq \sum x_{i,t}$ ).

**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. [14, 15, 17]) 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 [14, 17], we formalize aggregator obliviousness using an indistinguishability-based game in which aggregator  $\mathcal{A}$  accesses the following oracles:

- $\mathcal{O}_{\text{Setup}}$ : 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  $\text{sk}_A$  and public verification key  $\text{VK}$  to  $\mathcal{A}$ .
- $\mathcal{O}_{\text{Corrupt}}$ : When queried by aggregator  $\mathcal{A}$  with a user  $\mathcal{U}_i$ 's identifier  $\text{uid}_i$ , this oracle provides aggregator  $\mathcal{A}$  with  $\mathcal{U}_i$ 's secret key denoted  $\text{SK}_i$ .
- $\mathcal{O}_{\text{EncTag}}$ : When queried with time  $t$ , user  $\mathcal{U}_i$ 's identifier  $\text{uid}_i$  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  $\text{SK}_i$ .
- $\mathcal{O}_{\text{AO}}$ : When called with a subset of users  $\mathbb{S} \subset \mathbb{U}$  and with two time-series  $(\mathcal{U}_i, t, x_{i,t}^0)_{\mathcal{U}_i \in \mathbb{S}}$  and  $(\mathcal{U}_i, t, x_{i,t}^1)_{\mathcal{U}_i \in \mathbb{S}}$  such that  $\sum x_{i,t}^0 = \sum x_{i,t}^1$ , this oracle flips a random coin  $b \in \{0, 1\}$  and returns an encryption of the time-series  $(\mathcal{U}_i, t, x_{i,t}^b)_{\mathcal{U}_i \in \mathbb{S}}$  (that is the tuple of ciphertexts  $\{c_{i,t}^b\}_{\mathcal{U}_i \in \mathbb{S}}$ ) and the corresponding authentication tags  $\{\sigma_{i,t}^b\}_{\mathcal{U}_i \in \mathbb{S}}$ .

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  $\mathcal{O}_{\text{Setup}}$  which in turn returns the public parameters  $\mathcal{P}$ , the public verification key  $\text{VK}$  and the aggregator's secret key  $\text{sk}_A$ . It also interacts with oracle  $\mathcal{O}_{\text{Corrupt}}$  to learn the secret keys  $\text{SK}_i$  of users  $\mathcal{U}_i$ , and oracle  $\mathcal{O}_{\text{EncTag}}$  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  $\mathcal{O}_{\text{AO}}$  then receives two time-series  $\mathcal{X}_{t^*}^0 = (\mathcal{U}_i, t^*, x_{i,t^*}^0)_{\mathcal{U}_i \in \mathbb{S}^*}$  and  $\mathcal{X}_{t^*}^1 = (\mathcal{U}_i, t^*, x_{i,t^*}^1)_{\mathcal{U}_i \in \mathbb{S}^*}$  from  $\mathcal{A}$ , such that  $\sum x_{i,t^*}^0 = \sum x_{i,t^*}^1$ . Then oracle  $\mathcal{O}_{\text{AO}}$  flips a random coin  $b \xleftarrow{\$} \{0, 1\}$  and returns to  $\mathcal{A}$  the ciphertexts  $\{c_{i,t^*}^b\}_{\mathcal{U}_i \in \mathbb{S}^*}$  and the matching authentication tags  $\{\sigma_{i,t^*}^b\}_{\mathcal{U}_i \in \mathbb{S}^*}$ .

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

---

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

---



---

**Algorithm 2:** Challenge phase of the obliviousness game

---

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

---

**Definition 1 (Aggregator Obliviousness).** An aggregation protocol is said to ensure aggregator obliviousness if for any polynomially bounded adversary  $\mathcal{A}$  the probability  $\Pr(b = b^*) \leq \frac{1}{2} + \epsilon(\kappa)$ , where  $\epsilon$  is a negligible function and  $\kappa$  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  $\text{sum}_t$ . Along these lines, we define *Aggregate Unforgeability* as the security feature that ensures that aggregator  $\mathcal{A}$  cannot convince data analyzer  $\mathcal{DA}$  to accept a bogus sum, and assume that users  $\mathcal{U}_i$  in the system are honest<sup>1</sup>. (i.e. they always submit their correct input and do not collude with the aggregator  $\mathcal{A}$ ).

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

**Algorithm 3:** Learning phase of the Aggregate Unforgeability game

---

```

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

```

---

**Algorithm 4:** Challenge phase of the Aggregate Unforgeability game

---

```

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

```

---

In compliance with previous work [10, 13], we formalize *Aggregate Unforgeability* via a game in which aggregator  $\mathcal{A}$  accesses as oracles  $\mathcal{O}_{\text{Setup}}$  and  $\mathcal{O}_{\text{EncTag}}$ . Furthermore, given the property that anyone holding the public verification key  $\text{VK}$  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  $\mathcal{O}_{\text{Setup}}$  with a security parameter. Oracle  $\mathcal{O}_{\text{Setup}}$  accordingly returns public parameters  $\mathcal{P}$ , verification key  $\text{VK}$  and the secret key  $\text{sk}_\mathcal{A}$  of aggregator  $\mathcal{A}$ . Moreover, aggregator  $\mathcal{A}$  calls oracle  $\mathcal{O}_{\text{EncTag}}$  with tuples  $(t, \text{uid}_i, x_{i,t})$  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  $\text{SK}_i$ . Note that for each time interval  $t$ , aggregator  $\mathcal{A}$  is allowed to query oracle  $\mathcal{O}_{\text{EncTag}}$  for user  $\mathcal{U}_i$  only once. In other words, aggregator  $\mathcal{A}$  cannot submit two distinct queries to oracle  $\mathcal{O}_{\text{EncTag}}$  with the same time interval  $t$  and the same user identifier  $\text{uid}_i$ . Without loss of generality, we suppose that for each time interval  $t$ , aggregator  $\mathcal{A}$  invokes oracle  $\mathcal{O}_{\text{EncTag}}$  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  $(t^*, \text{sum}_{t^*}, \sigma_{t^*})$ .

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

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

**Definition 2 (Aggregate Unforgeability).** Let  $\text{Pr}[\mathcal{A}^{\text{AU}}]$  denote the probability that aggregator  $\mathcal{A}$  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  $\mathcal{A}$ ,  $\text{Pr}[\mathcal{A}^{\text{AU}}] \leq \epsilon(\kappa)$ , where  $\epsilon$  is a negligible function in the security parameter  $\kappa$ .

### 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 verifiability:

- 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  $tk_i$  and sends an encoding thereof,  $\bar{tk}_i$  to the dealer. The dealer after collecting all encoded keys  $\bar{tk}_i$  by users, publishes the public verification key  $VK$  of this group of users. This verification key is computed as a function of the encodings  $\bar{tk}_i$ . Later, the dealer gives to each user in the system an encryption key  $ek_i$  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  $SK_i$  to compute a ciphertext based on the encryption algorithm of Shi *et al.* [17] 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  $sum_t$  of users' data and a proof  $\sigma$  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  $\sigma$  the validity of the result  $sum_t$ . Although the modification seems straightforward the unforgeability proof for **Type II Forgery** turns out to be challenging.

Thanks to the homomorphic encryption algorithm of Shi *et al.* [17] 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  $sum_t$  and the proof  $\sigma_t$ .

### 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 [17]:

#### 4.1 Shi-Chan-Rieffel-Chow-Song Scheme

- **Setup**( $1^\kappa$ ): 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  $ek_i \in \mathbb{Z}_p$ . It distributes them to each user  $\mathcal{U}_i$  and it also sends to the Aggregator the secret key  $sk_A = -\sum_{i=1}^n ek_i$ .
- **Encrypt**( $ek_i, x_{i,t}$ ): 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  $c_{i,t} = H(t)^{ek_i} g_1^{x_{i,t}} \in \mathbb{G}_1$ .
- **Aggregate**( $\{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, sk_A$ ): The Aggregator upon receiving all the ciphertexts  $\{c_{i,t}\}_{i=1}^n$  computes:  $V = (\prod_{i=1}^n c_{i,t}) \cdot H(t)^{sk_A} = H(t)^{\sum_{i=1}^n ek_i} g_1^{\sum_{i=1}^n x_{i,t}} \cdot H(t)^{-\sum_{i=1}^n ek_i} = g_1^{\sum_{i=1}^n x_{i,t}} \in \mathbb{G}_1$ . Finally  $\mathcal{A}$  learns the sum  $sum_t = \sum_{i=1}^n x_{i,t} \in \mathbb{Z}_p$  by computing the discrete logarithm of  $V$  on the base  $g_1$ :  $sum_t = \log_{g_1} V = \sum_{i=1}^n x_{i,t}$ . 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 **PUA** protocol:

- **Setup**( $1^\kappa$ ):  $\mathcal{KD}$  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  $tk_i \in \mathbb{Z}_p$  independently and forwards  $g_2^{tk_i}$  to  $\mathcal{KD}$ .  $\mathcal{KD}$  also publishes the verification key  $VK = (vk_1, vk_2) = (g_2^{\sum_{i=1}^n tk_i}, g_2^a)$  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  $SK_i = (ek_i, tk_i, g_1^a)$ . 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**( $t, SK_i = (ek_i, tk_i, g_1^a), x_{i,t}$ ): 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  $c_{i,t} = H(t)^{ek_i} g_1^{x_{i,t}} \in \mathbb{G}_1$ .  $\mathcal{U}_i$  also constructs a tag in  $\mathbb{G}_1$  with its secret tag key  $(tk_i, g_1^a)$ :

$$\sigma_{i,t} = H(t)^{tk_i} (g_1^a)^{x_{i,t}} \in \mathbb{G}_1$$

and sends  $c_{i,t}, \sigma_{i,t}$  to  $\mathcal{A}$ .

- **Aggregate**( $sk_A, \{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$ ): The Aggregator computes the sum  $sum_t = \sum_{i=1}^n x_{i,t}$  by using the Aggregate function as presented in subsection 4.1.



Moreover aggregates tags by computing:

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

It finally forwards to the data analyzer  $\mathcal{DA}$   $\text{sum}_t$  and  $\sigma_t$ .

- **Verify**( $\text{VK} = (\text{vk}_1 = g_2^{\sum \text{tk}_i}, \text{vk}_2 = g_2^a), \text{sum}_t, \sigma_t$ ): During the verification phase the  $\mathcal{DA}$  verifies the correctness of the computation by checking:

$$e(\sigma_t, g_2) \stackrel{?}{=} e(H(t), \text{vk}_1) e(g_1^{\text{sum}_t}, \text{vk}_2)$$

Verification correctness follows from bilinear pairing properties:

$$\begin{aligned} e(\sigma_t, g_2) &= e\left(\prod_{i=1}^n \sigma_{i,t}, g_2\right) = e\left(\prod_{i=1}^n H(t)^{\text{tk}_i} g_1^{a x_{i,t}}, g_2\right) = \\ &= e\left(H(t)^{\sum_{i=1}^n \text{tk}_i} g_1^{a \sum_{i=1}^n x_{i,t}}, g_2\right) = e\left(H(t)^{\sum_{i=1}^n \text{tk}_i}, g_2\right) e\left(g_1^{a \sum_{i=1}^n x_{i,t}}, g_2\right) = \\ &= e\left(H(t), g_2\right)^{\sum_{i=1}^n \text{tk}_i} e\left(g_1, g_2\right)^{a \sum_{i=1}^n x_{i,t}} = e\left(H(t), g_2^{\sum_{i=1}^n \text{tk}_i}\right) e\left(g_1^{\sum_{i=1}^n x_{i,t}}, g_2^a\right) = \\ &= e\left(H(t), g_2^{\sum_{i=1}^n \text{tk}_i}\right) e\left(g_1^{\text{sum}_t}, g_2^a\right) = e\left(H(t), \text{vk}_1\right) e\left(g_1^{\text{sum}_t}, \text{vk}_2\right) \end{aligned}$$

## 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 [18] and it used for the construction of the CL signatures [8]. Intuitively we follow the generic model [18] 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 a second index  $t$  that is used in order to control the randomness. I.e: the randomness  $\beta_t$  that is employed by the oracle  $\mathcal{O}_{\text{LEMO}}$  is reused for a message  $m$  queried with index  $t$  that has already been queried. The purpose of this is to allow the homomorphic evaluation of the sum in the exponent. At each query  $(i, t, x)$ , if the index  $\text{ind} = (i, t)$  already exists in the dictionary  $\text{I - Dic}$ :  $(i, t) \in \text{I - Dic}$  then  $\mathcal{O}_{\text{LEMO}}$  aborts. Otherwise when  $(i, t) \ni \text{I - Dic}$ , there are three cases for the new index:

$$(i, t) = \begin{cases} (i', t) & : \text{ctr}_t = \text{ctr}_t + 1, \text{I - Dic}[i', t], (\alpha, \beta_t, \beta_t^{\gamma_i} \alpha^{\delta x}) \\ (i', t') : \text{ctr}_t = 1, \beta_t \xleftarrow{\$} \mathbb{Z}_p, (\alpha, \beta_t, \beta_t^{\gamma_i} \alpha^{\delta x}), \text{I - Dic.append}((i', t') : \beta_t) \\ (i, t') & : \text{ctr}_t = 1, \beta_t \xleftarrow{\$} \mathbb{Z}_p, (\alpha, \beta_t, \beta_t^{\gamma_i} \alpha^{\delta x}), \text{I - Dic.append}((i, t') : \beta_t) \end{cases}$$

In the first case there is an index in  $\mathbf{I} - \mathbf{Dic}$  with  $i' \neq i$  and  $t = t$ . The oracle increases the counter  $\mathbf{ctr}_t$  by 1, fetches  $\beta_t$  from the dictionary and responds with  $(\alpha, \beta_t, \beta_t^{\gamma_i} \alpha^{\delta x})$ . In the other two cases the query includes a  $t$  that was not part of any query before. The oracle in both cases initializes the counter  $\mathbf{ctr}_t$  with 1, choses a random  $\beta_t$  and stores  $((i', t') : \beta_t)$  for the second case and  $((i, t') : \beta_t)$  for the third case to the dictionary.

**Theorem 2.** (LEMO Assumption) *Let  $\mathcal{G}$  be an algorithm that on input the security parameter  $\kappa$  outputs the parameters of a bilinear group as  $\mathbf{G}=(\mathbb{G}_1, \mathbb{G}_2, g_1, g_2, p, e)$ . 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  $\mathcal{O}_{\text{LEMO}}$  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  $\alpha, \beta_t \in \mathbb{Z}_p$ .*

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

$$\Pr[\mathbf{G} \leftarrow \mathcal{G}(1^\kappa); \delta, \gamma_i \in \mathbb{Z}_p; (X = g_2^\delta, Y = g_2^{\sum_{i=1}^n \gamma_i}); (a, b, c) \leftarrow \mathcal{A}^{\mathcal{O}(i, t, x)} : \\ ((i, t) \ni \mathbf{I} - \mathbf{Dic}) \wedge (x \neq \sum_{i=1}^n x_i, t) \wedge (\mathbf{ctr}_t \leq n) \wedge a = \alpha \wedge b = \beta_t \wedge c = \beta_t^{\gamma_i} \alpha^{x\delta}] \leq \epsilon_2(\kappa)$$

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 3.** *Our scheme achieves Aggregate Unforgeability for a **Type I Forgery** under the co – CHD assumption in the random oracle model.*

**Theorem 4.** *Our scheme guarantees Aggregate Unforgeability for a **Type II Forgery** under the LEMO assumption in the random oracle model.*

Due to space limitations the security evidence of the LEMO assumption and proofs for Theorem 2, 3 are deferred in Appendix B and C.

### 5.3 Performance Evaluation

In this section we analyze the extra overhead that is occurred for the aggregate unforgeability property of our **PUA** instantiation scheme with respect to Shi *et al.* scheme [17] 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 our 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  $\text{tk}_i \in \mathbb{Z}_p$ . 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  $\sigma_{i,t} = H(t)^{\text{tk}_i} (g_1^a)^{x_{i,t}} \in \mathbb{G}_1$ , which entails one hash evaluation, two exponentiations and one multiplication in  $\mathbb{G}_1$ . For the computation of the  $\sigma_t$  the

Participant	Computation	Com.
User	2 EXP + 1 MUL + 1 HASH	2 · l
Aggregator	(n - 1) MUL	2 · l
Data Analyzer	3 PAIR + 1 EXP + 1 MUL + 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	1.2 ms	1.8 ms	2.2 ms
Verify	28.3 ms	42.7 ms	53.5 ms

Table 2: Computational cost of PUA operations with respect to different pairings.

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

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

Aggregator is involved in  $n - 1$  multiplications in  $\mathbb{G}_1 : \prod_{i=1}^n \sigma_{i,t}$ . Finally the data analyzer verifies by checking the equality:  $e(\sigma_t, g_2) \stackrel{?}{=} e(H(t), vk_1)e(g_1^{\text{sum}_t}, vk_2)$ , 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 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.

We implemented the verification functionalities of PUA with the Charm cryptographic framework [1, 2]. For pairing computations it inherits the PBC [16] 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 [9], 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  $f(r_1, \dots, r_n, d_1, \dots, d_n)$  takes computational time  $t$  proportional to  $n$ . Then the knowledge of a key  $K$  for a pseudorandom function  $F_K$  can evaluate  $f(r_1 = F_K(L_1), \dots, r_n = F_K(L_n), d_1, \dots, d_n)$  in time less than  $t$ . Contrary to our solution both solutions do not provide individual privacy. In the multi-user setting, *Choi et al.* [12] 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.* [11] 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. However the aggregate value is not allowed to be learnt in cleartext by the untrusted aggregator since the protocols are geared for cloud based scenarios.

## 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 **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, co – CHD 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] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. In *Proceedings of the 7th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology, ASIACRYPT '01*, pages 514–532, London, UK, UK, 2001. Springer-Verlag.
- [8] J. Camenisch and A. Lysyanskaya. Signature schemes and anonymous credentials from bilinear maps. In *Advances in Cryptology - CRYPTO 2004, 24th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 2004, Proceedings*, pages 56–72, 2004.
- [9] D. Catalano and D. Fiore. Practical homomorphic macs for arithmetic circuits. In *EUROCRYPT*, pages 336–352, 2013.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] M. Joye and B. Libert. A scalable scheme for privacy-preserving aggregation of time-series data. In *Financial Cryptography*, 2013.
- [15] 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.
- [16] B. Lynn. The stanford pairing based crypto library. <http://crypto.stanford.edu/pbc>.

- [17] E. Shi, T.-H. H. Chan, E. G. Rieffel, R. Chow, and D. Song. Privacy-preserving aggregation of time-series data. In *NDSS*, 2011.
- [18] 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  $\epsilon$ . 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 [17], 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:  $\mathcal{O}_{\text{Setup}}^{\text{PSA}}$ ,  $\mathcal{O}_{\text{Corrupt}}^{\text{PSA}}$ ,  $\mathcal{O}_{\text{Encrypt}}^{\text{PSA}}$ , and  $\mathcal{O}_{\text{AO}}^{\text{PSA}}$ .

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  $\text{ek}_i$ . 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  $\mathcal{O}_{\text{Setup}}$  with a security parameter  $\kappa$ ,  $\mathcal{B}$  queries oracle  $\mathcal{O}_{\text{Setup}}^{\text{PSA}}$  with the same security parameter. Oracle  $\mathcal{O}_{\text{Setup}}^{\text{PSA}}$  in turn outputs the public parameters that are composed of a hash function  $H : \{0, 1\}^* \rightarrow \mathbb{G}_1$ , a generator  $g_1$  of the group  $\mathbb{G}_1$  of safe prime order  $p$ , and the aggregator's secret key  $\text{SK}_{\mathcal{A}} = -\sum_{i=1}^n \text{ek}_i$ .  $\mathcal{B}$  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  $\text{VK}$  as follows:

$$\text{VK} = (g_2^{a\text{SK}_{\mathcal{A}} + \sum_{i=1}^n r_i}, g_2^a) = (g_2^{a \sum_{i=1}^n \text{ek}_i + \sum_{i=1}^n r_i}, g_2^a) = (g_2^{\sum_{i=1}^n a\text{ek}_i + r_i}, g_2^a)$$

This entails that  $\text{tk}_i$  is defined as:  $a\text{ek}_i + r_i$ . 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  $\text{VK} = (g_2^{\sum_{i=1}^n \text{tk}_i}, g_2^a)$  and the secret key of the Aggregator  $\text{sk}_{\mathcal{A}}$ .

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

Whenever  $\mathcal{A}$  calls oracle  $\mathcal{O}_{\text{EncTag}}$  with query  $(t, \text{uid}_i, x_{i,t})$ ,  $\mathcal{B}$  forwards the query to the  $\mathcal{O}_{\text{Encrypt}}^{\text{PSA}}$  oracle which returns the appropriate ciphertext  $c_{i,t} = H(t)^{\text{ek}_i} g_1^{x_{i,t}}$ .  $\mathcal{B}$  computes then the tag associated with ciphertext  $c_{i,t}$  as  $\sigma_{i,t} = (c_{i,t})^a H(t)^{r_i} = H(t)^{a\text{ek}_i + r_i} g_1^{ax_{i,t}} = H(t)^{\text{tk}_i} g_1^{ax_{i,t}}$  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  $\mathcal{O}_{\text{EncTag}}$ .  $\mathcal{A}$  then submits two time-series  $\mathcal{X}_0^* = (\mathcal{U}_i, t^*, x_{i,t^*}^0)_{\mathcal{U}_i \in \mathbb{S}^*}$  and  $\mathcal{X}_1^* = (\mathcal{U}_i, t^*, x_{i,t^*}^1)_{\mathcal{U}_i \in \mathbb{S}^*}$  such that  $\sum x_{i,t^*}^0 = \sum x_{i,t^*}^1$  to  $\mathcal{O}_{\text{AO}}$  oracle.  $\mathcal{B}$  simulates this oracle as follows:

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

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

$$\sigma_{i,t^*}^b = (c_{i,t^*}^b)^a H(t^*)^{r_i} = (H(t^*)^{\text{ek}_i} g_1^{x_{i,t^*}^b})^a H(t^*)^{r_i} = H(t^*)^{a\text{ek}_i + r_i} g_1^{ax_{i,t^*}^b} = H(t^*)^{\text{tk}_i} g_1^{ax_{i,t^*}^b}$$

Note that  $\sigma_{i,t^*}^b$  corresponds to a correctly computed tag for input  $x_{i,t^*}^b$ . Finally,  $\mathcal{B}$  forwards to  $\mathcal{A}$   $\{(c_{i,t^*}^b, \sigma_{i,t^*}^b)\}_{\mathcal{U}_i \in \mathbb{S}^*}$ . 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:

$$\begin{aligned} e\left(\prod_{i \in \mathbb{S}^*} \sigma_{i,t^*}^b, g_2\right) &= e\left(\prod_{i=1}^n H(t^*)^{\text{tk}_i} g_1^{ax_{i,t^*}^b}, g_2\right) = \\ e(H(t^*), g_2^{a \sum_{i=1}^n \text{ek}_i + \sum_{i=1}^n r_i}) e(g_1^{\sum_{i=1}^n x_{i,t^*}^b}, g_2^a) &= e(H(t^*), \text{vk}_1) e(g_1^{\sum_{i=1}^n x_{i,t^*}^b}, \text{vk}_2) \end{aligned}$$

It follows that if aggregator  $\mathcal{A}$  is able to output a correct guess  $b^*$  for the bit  $b$  with a non-negligible advantage  $\epsilon$  (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  $\epsilon$  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 Security evidence for the LEMO Assumption

In this section we provide security evidence for the hardness of the new LEMO assumption by presenting 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 [18]. 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  $\xi$  that maps elements from to  $\mathbb{G} \rightarrow \Xi$  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  $\xi_c : \mathbb{G}_c \rightarrow \{0, 1\}^{\lceil \log_2 q \rceil}$ .

**Theorem 2.** *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  $\mathcal{O}_{\text{LEMO}}(i, t, x)$  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  $\epsilon_2$  that  $\mathcal{A}$  on input  $(p, \xi_1(1), \xi_2(1), \xi_1(a), \xi_1(b), \xi_1(c), \xi_2(\delta), \xi_2(\sum_{i=1}^n \gamma_i))$  to output a tuple  $(\xi_1(a), \xi_1(b), \xi_1(c_f = \xi_1(\beta_t \sum_{u=1}^n \gamma_u + \alpha \delta \sum_{u=1}^n x_{u,t})))$  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:

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

where  $F_{1,i}, F_{2,i} \in \mathbb{Z}_p[A, B, \{G_u\}_{u=1}^n, \Delta, X]$  are multivariate polynomial on the indeterminates  $A, B, \{G_u\}_{u=1}^n, \Delta, X$  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  $\tau$ , 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  $F_{1,1} = 1, F_{2,1} = 2, F_{2,2} = \sum_{u=1}^n G_u, F_{2,3} = \Delta$  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  $\mathcal{O}_{\text{LEMO}}$ . 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 then 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  $F_{c,\tau_c}$  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 chooses 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  $\mathbb{G}_T$  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  $(F_{T,\tau_T}, \xi_{T,\tau_T})$ .
- $\mathcal{O}_{\text{LEMO}}$ :  $\mathcal{B}$  increments a counter  $\tau_{\mathcal{O}}$  by 1 and sets  $\tau_{1+} = 3$ .  $\mathcal{A}$  inputs  $(u, t, x_{u,t})$ .  $\mathcal{B}$  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  $l \in [1, \dots, \tau_1]$ . 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:

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

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:

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

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

$$F_f = F_{T,k} - F_{T,l}F_{T,o} \quad (2)$$

for indexes  $k, l, o$ . Simplifying the equation, since it is equal to 0, then the second part consists of a polynomial with determinants  $(\Delta\Gamma)^2, (E\Gamma)^2, A\Delta^4 X^2, (A\Delta X E)^2$  and the first part with determinants  $(\Delta\Gamma, E\Gamma, A\Delta^2 X, A\Delta X E)$ . 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  $y_{0,k} = y_{0,l}y_{0,o}$ .

$\mathcal{B}$  assigns random values  $(\alpha, \beta, \gamma, \delta, x)$  for the indeterminants  $A, B, \Gamma, \Delta, X$  and in order for  $\mathcal{A}$  to win in the game, it 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  $\epsilon_2 \leq \frac{(q_G + 12)^2}{p}$

### C Aggregate Unforgeability

**Theorem 3.** *Our scheme achieves Aggregate Unforgeability for a **Type I Forgery** under co – CHD assumption as in BLS signatures [7] in the random oracle model.*

*Proof.* (Sketch) The tag  $\sigma_{i,t}$  can be simulated with the BLS signatures for a **Type I Forgery** in which  $\mathcal{A}$  never queried the  $\mathcal{O}_{\text{EncTag}}^A$  oracle. We only depict here the idea of the reductionist proof since it is straightforward. We construct an adversary  $\mathcal{B}$ , who emulates the PUA oracles by communicating with the BLS challenger. In a nutshell, when  $\mathcal{A}$  submits a valid forgery,  $\mathcal{B}$  can strip off the extra randomness of  $g_1^{\text{asum}_t'}$  and responds to the BLS challenger with  $\sigma_t/g_1^{\text{asum}_t'} = H(t)^{tk_i}$  which is a valid forgery for the message  $t$ .  $\mathcal{B}$  also chooses secret encryption keys  $\{\text{ek}_i\}_{i=1}^n$  and constructs the ciphertext as  $c_{i,t} = H(t)^{\text{ek}_i} g_1^{x_{i,t}}$ . It also sends to  $\mathcal{A}$  the secret decryption key  $\text{sk}_A = \sum_{i=1}^n \text{ek}_i$ . As such the probability of  $\mathcal{A}$  to output a **Type I Forgery** is  $\Pr[\mathcal{A}^{\text{AU1}}] \leq \epsilon_1$  for a negligible function  $\epsilon_1$ .

**Theorem 4.** *Our scheme guarantees Aggregate Unforgeability for a **Type II Forgery** under the LEMO assumption in the random oracle model.*

*Proof.* The  $\mathcal{O}_{\text{EncTag}}^A$  oracle behaves equivalently as the oracle in the LEMO assumption.  $\mathcal{B}$  chooses secret encryption keys  $\{\text{ek}_i\}_{i=1}^n$  and sends to  $\mathcal{A}$  the secret decryption key  $\text{sk}_A = \sum_{i=1}^n \text{ek}_i$ . For queries  $(i = \text{uid}, t, x)$  to the  $\mathcal{O}_{\text{EncTag}}^A$  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  $c_{i,t} = H(t)^{\text{ek}_i} g_1^{x_{i,t}}$ . For a random oracle query  $H(t)$  the simulator  $\mathcal{B}$  queries the  $\mathcal{O}_{\text{LEMO}}$  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  $\Pr[\mathcal{A}^{\text{AU2}}] \leq \epsilon_2$  for a negligible function  $\epsilon_2$  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  $\Pr[\mathcal{A}^{\text{AU}}]$ , are taken over the union of the success probabilities for the two type of forgeries. As such

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

where  $\epsilon_1$  and  $\epsilon_2$  are negligible functions.