PUDA – Privacy and Unforgeability for Data Aggregation

Iraklis Leontiadis, Kaoutar Elkhiyaoui, Melek Önen, Refik Molva

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

Abstract. Existing work on secure data collection and aggregation is mainly focused on confidentiality issues. That is, ensuring that the untrusted Aggregator learns only the aggregation result without divulging individual data inputs. In this paper however we consider a malicious Aggregator which is not only interested in compromising users' privacy but also is interested in providing bogus aggregate values. More concretely, we extend existing security models with the requirement of *aggregate unforgeability*. Moreover, we instantiate an efficient protocol for private and unforgeable data aggregation that allows the Aggregator to compute the sum of users' inputs without learning individual values and constructs a proof of correct computation that can be verified by any third party. The proposed protocol is provably secure and its communication and computation overhead is minimal.

1 Introduction

With the advent of *Big Data*, research on privacy preserving data collection and analysis is culminating as users continuously produce data which once aggregated becomes very valuable. Often scenarios regarding data analysis involve an Aggregator which collects individual data from multiple (independent) users to compute useful statistics, these statistics are generally forwarded to Data Analyzers whose role is to extract insightful information about the entire user population. Various motivating examples for the aforementioned generic scenario 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 on a large scale by Aggregators. Health scientists who act as Data Analyzers infer statistical information from these data without accessing the individual inputs (for privacy reasons). An aggregate value computed over a large population would give very useful information for deriving statistical models, evaluating therapeutic performance or learning the likelihood of upcoming patients' diseases.

Unfortunately, existing solutions only focus on the problem of data confidentiality and consider the Aggregator to be *honest-but-curious*: the Aggregator wants to discover the content of each individual data, but performs the aggregation operation correctly. In this paper we consider a more powerful security model by assuming a malicious 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 of the correctness of the computation of the aggregate result to the Data Analyzer. For efficiency reasons, we require that the Data Analyzer verifies the correctness of the computation without communicating with users in the system.

The underlying idea of our solution is that each user encrypts its data according to Shi *et al.* [17] scheme using its own secret encryption key, and sends the resulting ciphertext to the untrusted Aggregator. Users, also homomorphically tag their data using two layers of randomness with two different keys and forward the tags to the Aggregator. The latter computes the sum by applying operations on the ciphertexts and derives a proof of computation correctness from the tags. The Aggregator finally sends the result and the proof to the Data Analyzer. In addition to ensuring obliviousness against the Aggregator and the Data Analyzer (i.e. neither the Data Analyzer nor the Aggregator learns individual data inputs), the proposed protocol assures *public verifiablity*: any third party can verify the correctness of the aggregate value.

To the best of our knowledge we are the first to define a model for *Privacy and Unforgeability for Data Aggregation* ($\mathbf{PUDA}$). We also instantiate a $\mathbf{PUDA}$ scheme that supports:

- A multi-user setting where multiple users produce personal sensitive data without interacting with each other.
- Privacy of users' individual data.
- Public verifiability of the aggregate value.

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 $\mathcal{DA}$ learn nothing, but the value of the aggregation. While homomorphic signatures proposed in [4, 10] seem to answer 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 $\mathbf{PUDA}$ Model

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

- $\mathsf{\mathbf{Setup}}(1^\kappa) \to (\mathcal{P}, \mathsf{SK_A}, \{\mathsf{SK}_i\}_{\mathcal{U}_i \in \mathbb{U}}, \mathsf{VK})$: It is a randomized algorithm which on input of a security parameter κ, this algorithm 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.
- $\mathsf{\mathbf{EncTag}}(t, \mathsf{SK}_i, x_{i,t}) \to (c_{i,t}, \sigma_{i,t})$: 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 $\sigma_{i,t}$ that authenticates $x_{i,t}$.
- $\mathsf{\mathbf{Aggregate}}(\mathsf{SK_A}, \{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}) \to (\mathsf{sum_t}, \sigma_\mathrm{t})$: It is a deterministic algorithm executed 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 the sum $\mathsf{sum_t}$ of the values $\{x_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}$ in cleartext and a proof σ_t of correctness for $\mathsf{sum_t}$.
- $\mathsf{\mathbf{Verify}}(\mathsf{VK}, t, \mathsf{sum_t}, \sigma_\mathrm{t}) \to \{0, 1\}$: 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 proof σ_t corresponds to the sum $\mathsf{sum_t} = \sum_{\mathcal{U}_i \in \mathbb{U}}\{x_{i,t}\}$, where $x_{i,t}$ are individual data inputs at time interval t of user $\mathcal{U}_i$; 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. 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 [11, 17], we formalize the property of *Aggregator obliviousness*, which ensures that at the end of a protocol execution, Aggregator $\mathcal{A}$ only learns the sum of users' inputs 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 $\sigma_{i,t}$, Aggregator $\mathcal{A}$ cannot reveal any information about individual inputs $x_{i,t}$, other than the sum value $\sum x_{i,t}$. We extend the existing definition of *Aggregator Obliviousness* (cf. [11, 13, 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 [11, 17], 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 $\mathcal{X}^0_{t^*} = (\mathcal{U}_i, t, x^0_{i,t})_{\mathcal{U}_i \in \mathbb{S}}$ and $\mathcal{X}^1_{t^*} = (\mathcal{U}_i, t, x^1_{i,t})_{\mathcal{U}_i \in \mathbb{S}}$ 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 $\sum x^0_{i,t^*} = \sum_{\mathcal{U}_i \in \mathbb{S}^*} x^1_{i,t^*}$. 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

```
$(\mathcal{P}, \mathsf{SK}_\mathsf{A}, \mathsf{VK}) \leftarrow \mathcal{O}_{\mathsf{Setup}}(1^\kappa)$;
// $\mathcal{A}$ 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). *Let* $\Pr[\mathcal{A}^{\mathbf{AO}}]$ *denote the probability that Aggregator* $\mathcal{A}$ *outputs* $b^* = b$. *Then an aggregation protocol is said to ensure Aggregator obliviousness if for any polynomially bounded Aggregator* $\mathcal{A}$ *the probability* $\Pr[\mathcal{A}^{\mathbf{AO}}] \leqslant \frac{1}{2} + \epsilon(\kappa)$*, 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 is also interested in tampering with the sum of users' inputs. That is, Aggregator $\mathcal{A}$ may sometimes have an incentive to feed Data Analyzer $\mathcal{DA}$ erroneous sums. 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, as long as users $\mathcal{U}_i$ in the system are honest (i.e. they always submit their correct input and do not collude with the Aggregator $\mathcal{A}$).

In compliance with previous work [7, 10] on homomorphic signatures, we formalize *aggregate unforgeability* via a game in which Aggregator $\mathcal{A}$ accesses 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 runs the algorithm Verify by 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 κ. Oracle $\mathcal{O}_{\mathsf{Setup}}$ accordingly returns public parameters $\mathcal{P}$, verification key VK and the secret key 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

Algorithm 3: Learning phase of the aggregate unforgeability 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
// $\mathcal{A}$ is allowed to call $\mathcal{O}_{\mathsf{EncTag}}$ for all users $\mathcal{U}_i$
$(c_{i,t}, \sigma_{i,t}) \leftarrow \mathcal{O}_{\mathsf{EncTag}}(t, \mathsf{uid}_i, x_{i,t})$;
```

Algorithm 4: Challenge phase of the aggregate unforgeability game

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

$\mathcal{A}$ 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 the *aggregate unforgeability* game (see Algorithm 4), Aggregator $\mathcal{A}$ outputs a tuple $(t^*, \mathsf{sum}_{t^*}, \sigma_{t^*})$. We say that Aggregator $\mathcal{A}$ wins the *aggregate unforgeability* game if one of the following statements holds:

1. $\mathsf{Verify}(\mathsf{VK}, t^*, \mathsf{sum}_{t^*}, \sigma_{t^*}) \rightarrow 1$ 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. $\mathsf{Verify}(\mathsf{VK}, t^*, \mathsf{sum}_{t^*}, \sigma_{t^*}) \rightarrow 1$ 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 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 aggregator $\mathcal{A}$*,* $\Pr[\mathcal{A}^{\mathbf{AU}}] \leqslant \epsilon(\kappa)$*, where* ϵ *is a negligible function in the security parameter* κ*.*

3 Idea of our PUDA protocol

In a security model where the Aggregator is malicious, it is of utmost importance to design a solution in which the Aggregator cannot provide bogus results to the Data Analyzer without being detected. Such a solution will use a proof system that enables the Data Analyzer to verify the correctness of the computation. Yet verifiability should be achieved without sacrificing privacy. In order to achieve this goal, we propose a protocol that relies on the following techniques:

- 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 aggregated 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 $\overline{\mathsf{tk_i}}$ thereof to the key dealer. After collecting all encoded keys $\overline{\mathsf{tk_i}}$, the key dealer publishes the corresponding public verification key VK. This verification key is computed as a function of the encodings $\overline{\mathsf{tk_i}}$. Later, the key 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 key dealer provides the Aggregator with secret key SK_A computed as the sum of encryption keys ek_i and goes off-line.

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.* [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 $\mathsf{sum_t}$ of users' data and a matching proof σ_{t}, 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_{t} the validity of the result $\mathsf{sum_t}$.

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 $\mathcal{DA}$ does not keep any state with respect to users' transcripts be they ciphertexts or tags, but it only holds the public verification key, the sum $\mathsf{sum_t}$ and the proof σ_{t}.

4 PUDA Instantiation

Let $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ be three cyclic groups of large prime order p and g_1, g_2 be 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*: $e(g_1^a, g_2^b) = e(g_1, g_2)^{ab}$, for all $g_1, g_2 \in \mathbb{G}_1 \times \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$.
2. *Computability*: there exists an efficient algorithm that computes $e(g_1^a, g_2^b)$ for all $g_1, g_2 \in \mathbb{G}_1 \times \mathbb{G}_2$ and $a, b \in \mathbb{Z}_p$.
3. *Non-degeneracy*: $e(g_1, g_2) \neq 1$.

To encrypt users' data homomorphically, we employ the *discrete logarithm* based encryption scheme of Shi *et al.* [17]:

4.1 Shi-Chan-Rieffel-Chow-Song Scheme

- $\mathsf{Setup}(1^\kappa)$: Let $\mathbb{G}_1$ be a group of large prime order p. A trusted key dealer $\mathcal{KD}$ selects a hash function $H : \{0,1\}^* \to \mathbb{G}_1$. Furthermore, $\mathcal{KD}$ selects secret encryption keys $\mathsf{ek}_i \in \mathbb{Z}_p$ uniformly at random. $\mathcal{KD}$ distributes to each user $\mathcal{U}_i$ the secret key ek_i and sends the corresponding decryption key $\mathsf{SK}_A = -\sum_{i=1}^{n} \mathsf{ek}_i$ to the Aggregator.

- **Encrypt**$(\mathsf{ek}_i, x_{i,t})$: Each user $\mathcal{U}_i$ encrypts the value $x_{i,t}$ using its secret encryption key ek_i and outputs the ciphertext $c_{i,t} = H(t)^{\mathsf{ek}_i} g_1^{x_{i,t}} \in \mathbb{G}_1$.
- **Aggregate**$(\{c_{i,t}\}_{i=1}^n, \{\sigma_{i,t}\}_{i=1}^n, \mathsf{SK_A})$: Upon receiving all the ciphertexts $\{c_{i,t}\}_{i=1}^n$, the Aggregator computes: $V_t = (\prod_{i=1}^n c_{i,t}) H(t)^{\mathsf{SK}_A} = H(t)^{\sum_{i=1}^n \mathsf{ek}_i} g_1^{\sum_{i=1}^n x_{i,t}} H(t)^{-\sum_{i=1}^n \mathsf{ek}_i} = g_1^{\sum_{i=1}^n x_{i,t}} \in \mathbb{G}_1$. 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_t on the base g_1. The sum computation is correct as long as $\sum_{i=1}^n x_{i,t} < p$.

The above scheme is efficient as long as the plaintext values remain in a small range so as to the discrete logarithm computation during **Aggregate** algorithm is fast.

4.2 PUDA Scheme

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

- **Setup**(1^κ): $\mathcal{KD}$ outputs the parameters $(p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T)$ for an efficient computable bilinear map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, 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 prime number that denotes the order of all the groups $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$. Moreover secret keys $a, \{\mathsf{tk_i}\}_{i=1}^n$ are selected by $\mathcal{KD}$. $\mathcal{KD}$ 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 $g_1^a \in \mathbb{G}_1$, the encryption key ek_i and the tag key $\mathsf{tk_i}$ 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 $x_{i,t}$ with its secret encryption key ek_i, using the encryption algorithm, described in section 4.1, which 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 $\sigma_{i,t}$ with its secret tag key (tk_i, g_1^a):

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

  Finally $\mathcal{U}_i$ sends $(c_{i,t}, \sigma_{i,t})$ to $\mathcal{A}$.
- **Aggregate**$(\mathsf{SK_A}, \{c_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}}, \{\sigma_{i,t}\}_{\mathcal{U}_i \in \mathbb{U}})$: Aggregator $\mathcal{A}$ computes the sum $\mathsf{sum_t} = \sum_{i=1}^n x_{i,t}$ by using the **Aggregate** algorithm presented in section 4.1. Moreover, $\mathcal{A}$ aggregates the corresponding tags as follows:

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

  $\mathcal{A}$ finally forwards $\mathsf{sum_t}$ and σ_{t} to data analyzer $\mathcal{DA}$.
- **Verify**$(\mathsf{VK}, t, \mathsf{sum_t}, \sigma_{\mathsf{t}})$: During the verification phase $\mathcal{DA}$ verifies the correctness of the computation with the verification key $\mathsf{VK} = (\mathsf{vk}_1, \mathsf{vk}_2) = (g_2^{\sum \mathsf{tk}_i}, g_2^a)$, by checking the following equality:

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

Verification correctness follows from bilinear pairing properties:

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

5 Analysis

5.1 Aggregator Obliviousness

Theorem 1. *The proposed solution achieves Aggregator 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 the full version [14].

5.2 Aggregate Unforgeability

We first introduce a new assumption that is used during the security analysis of our **PUDA** instantiation. Our new assumption named hereafter LEOM is a variant of the LRSW assumption [16] which is proven secure in the generic model [18] and used in the construction of the CL signatures [5].

The oracle $\mathcal{O}_{\mathsf{LEOM}}$ first chooses a and k_i, $1 \leqslant i \leqslant n$ in $\mathbb{Z}_p^*$. Then it publishes the tuple $(g_1, g_2^{\sum_{i=1}^{n} k_i}, g_2^{a})$. Thereafter, the adversary picks $h_t \in \mathbb{G}_1$ and makes queries $(h_t, i, x_{i,t})$ for $1 \leqslant i \leqslant n$ to the $\mathcal{O}_{\mathsf{LEOM}}$ oracle which in turn replies with $h_t^{k_i} g_1^{ax_{i,t}}$ for $1 \leqslant i \leqslant n$.

The adversary is allowed to query the oracle $\mathcal{O}_{\mathsf{LEOM}}$ for different h_t with the restriction that it cannot issue two queries for the same pair (h_t, i).

We say that the adversary breaks the LEOM assumption, if it outputs a tuple $(z, h_t, h_t^{\sum_{i=1}^{n} k_i} g_1^{az})$ for a previously queried t and $z \neq \sum_{i=1}^{n} x_{i,t}$.

Theorem 2. *(*LEOM *Assumption) Given the security parameter* κ, *the public parameters* $(p, e, \mathbb{G}_1, \mathbb{G}_2, g_1, g_2)$, *the public key* $(g_2^{a}, g_2^{\sum_{i=1}^{n} k_i})$ *and the oracle* $\mathcal{O}_{\mathsf{LEOM}}$, *we say that the* LEOM *assumption holds iff:*

For all probabilistic polynomial time adversaries $\mathcal{A}$, *the following holds:*

$$
\Pr[(z, h_t, \sigma_t) \leftarrow \mathcal{A}^{\mathcal{O}_{\mathsf{LEOM}}(.)} : z \neq \sum_{i=1}^{n} x_{i,t} \wedge \sigma_t = h_t^{\sum_{i=1}^{n} k_i} g_1^{az}] \leq \epsilon_2(\kappa)
$$

Where ϵ_2 *is a negligible function.*

Participant	Computation	Communication
User	$\mathbf{2}\,\mathsf{EXP} + \mathbf{1}\,\mathsf{MUL}$	$\mathbf{2}\cdot l$
Aggregator	$(\mathbf{n}-\mathbf{1})\,\mathsf{MUL}$	$\mathbf{2}\cdot l$
Data Analyzer	$\mathbf{3}\,\mathsf{PAIR} + \mathbf{1}\,\mathsf{EXP} + \mathbf{1}\,\mathsf{MUL} + \mathbf{1}\mathsf{H}ASH$	-

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

We show in our analysis that a **Type I Forgery** implies breaking the BCDH assumption and that a **Type II Forgery** implies breaking the LEOM assumption.

Theorem 3. *Our scheme achieves aggregate unforgeability against a* **Type I Forgery** *under* BCDH *assumption in the random oracle model.*

Theorem 4. *Our scheme guarantees aggregate unforgeability against a* **Type II Forgery** *under the* LEOM *assumption in the random oracle model.*

Due to space limitations, the security evidence of the LEOM assumption and proofs for Theorem 3, 4 are deferred to Appendix A and B.

5.3 Performance Evaluation

In this section we analyze the extra overhead of ensuring the *aggregate unforgeability* property in our **PUDA** instantiation scheme. 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 to ensure public verifiablity. 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, the key dealer distributes to each user $g_1^a, \mathsf{tk}_i \in \mathbb{G}_1$ and publishes $g_2^a, g_2^{\sum_{i=1}^{n} \mathsf{tk}_i} \in \mathbb{G}_2$, which calls for one exponentiation in $\mathbb{G}_1$ and $1+n$ 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 two exponentiations and one multiplication in $\mathbb{G}_1$. To compute the proof σ_t, the Aggregator carries out $n-1$ multiplications in $\mathbb{G}_1$. Finally the data analyzer verifies the validity of the aggregate sum by checking the equality: $e(\sigma_\mathsf{t}, g_2) \overset{?}{=} e(H(t), \mathsf{vk}_1)e(g_1^{\mathsf{sum}_\mathsf{t}}, \mathsf{vk}_2)$, 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 **PUDA** stems from the constant time verification with respect to the number of the users. This is of crucial importance since the Data Analyzer may not be computationally powerful.

We implemented the verification functionalities of **PUDA** 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 × 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

Pairings Operation	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 **PUDA** operations with respect to different pairings.

Curve Op.	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.

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 computation of tags $\sigma_{i,t}$ 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 [12] the authors presented a solution for verifiable aggregation in case of untrustworthy users. The solutions entails signatures on commitments of the secret values with non-interactive zero knowledge proofs, which are verified by the Aggregator. Hung-Min *et al.* [19] employed aggregate signatures in order to verify the integrity of the data, without addressing confidentiality issues for a malicious Aggregator. In [6], 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 computed by the cloud in order to verify the correctness of the output of a program P. 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. Contrary to our solution both solutions do not provide individual privacy and they are not designed for a multi-user scenario.

Catalano *et al.* [8] 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.

In the multi-user setting, Choi *et al.* [9] 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. However, the need of fully homomorphic encryption and garbled circuits renders the solution impractical for a real world scenario.

7 Concluding Remarks

In this paper, we designed and analyzed a protocol for privacy preserving and unforgeable data aggregation. The purpose of the protocol is to allow a data analyzer to verify the correctness of computation performed by a malicious Aggregator, without revealing the underlying data to either the Aggregator or the data analyzer. In addition to being provably secure and privacy preserving, the proposed protocol enables *public verifiability* in *constant time*.

Acknowledgments

We thank the anonymous reviewers for their suggestions for improving this paper. The research leading to these results was partially funded by the FP7-USERCENTRICNETWORKING european ICT project under the grant number 611001.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] K. Kursawe, G. Danezis, and M. Kohlweiss. Privacy-friendly aggregation for the smart-grid. In *PETS*, pages 175–191, 2011.

[13] 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.

[14] I. Leontiadis, K. Elkhyaoui, M. Önen, and R. Molva. Private and unforgeable data aggregation. *IACR Cryptology ePrint Archive*, 2015. `http://eprint.iacr.org/2015/562.pdf`.

[15] B. Lynn. The stanford pairing based crypto library. `http://crypto.stanford.edu/pbc`.

[16] A. Lysyanskaya, R. Rivest, A. Sahai, and S. Wolf. Pseudonym systems. In H. Heys and C. Adams, editors, *Selected Areas in Cryptography*, volume 1758 of *Lecture Notes in Computer Science*, pages 184–199. Springer Berlin Heidelberg, 2000.

[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.

[19] H.-M. Sun, Y.-H. Lin, Y.-C. Hsiao, and C.-M. Chen. An efficient and verifiable concealed data aggregation scheme in wireless sensor networks. In *Embedded Software and Systems, 2008. ICESS '08. International Conference on*, pages 19–26, July 2008.

A Security evidence for the LEOM Assumption

In this section we provide security evidence for the hardness of the new LEOM assumption by presenting bounds on the success probabilities of an adversary $\mathcal{A}$ which 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 ξ_c that maps elements in $\mathbb{G}_c \rightarrow \{0,1\}^{\lceil \log_2 p \rceil}$ to represent element in $\mathbb{G}_c, c \in [1, 2, T]$.

Theorem 5. *Suppose $\mathcal{A}$ is a polynomial probabilistic time adversary that breaks the* LEOM *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 $\mathcal{O}_{\mathsf{LEOM}}$ oracle, all counted together. Then the probability ϵ_2 that $\mathcal{A}$ breaks the* LEOM *assumption is bounded as follows:*

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

.

Due to space limitations we include the proof in the full version [14].

B Aggregate Unforgeability

Theorem 3. *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 adversary $\mathcal{B}$ 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 **Learning** phase as follows:
Setup:

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

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 π, $\mathsf{coin(t)} = 0$ 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 $(t, \mathsf{uid}_i, x_{i,t})$ to the $\mathcal{O}^{\mathcal{A}}_{\mathsf{EncTag}}$, $\mathcal{B}$ responds as follows:
   1. $\mathcal{B}$ 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 $\mathtt{H} - \mathtt{list}$.
   2. If $\mathsf{coin}(t) = 1$ then $\mathcal{B}$ stops the simulation.
   3. Otherwise it chooses the secret tag key tk_i where $i = \mathsf{uid_i}$ to be used as secret tag key from the set of $\{\mathsf{tk}_i\}$ keys, chosen by $\mathcal{B}$ in the **Setup** phase.
   4. $\mathcal{B}$ sends to $\mathcal{A}$ the tag $\sigma_{i,t} = g_1^{r_t b \mathsf{tk}_i} g_1^{a x_{i,t}} = H(t)^{b\mathsf{tk}_i} g_1^{a x_{i,t}}$, 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.
   5. $\mathcal{B}$ chooses also a secret encryption key $\mathsf{ek}_i \in \{\mathsf{ek}_i\}_{i=1}^n \in \mathbb{Z}_p$ and computes the ciphertext as $c_{i,t} = H(t)^{\mathsf{ek}_i} g_1^{x_{i,t}}$. 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 $\mathsf{SK}_A = -\sum_{i=1}^n \mathsf{ek_i}$, considering the adversary has made distinct encryption queries for all the n users in the scheme at a time interval t.

Now, when $\mathcal{B}$ receives the forgery $(\mathsf{sum_t}^*, \sigma_\mathsf{t}^*)$ at time interval $t \neq t^*$, 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 $\mathcal{A}$ outputs a valid forged σ_t^* at t^*, it is 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 $\mathcal{A}$ makes n queries for time interval t^* for distinct users to the $\mathcal{O}^{\mathcal{A}}_{\mathsf{EncTag}}$ oracle during the **Learning** phase. As such $\sigma_\mathsf{t}^* = g_1^{cr_t b \sum \mathsf{tk}_i} g_1^{a\mathsf{sum_t}^*}$. Finally $\mathcal{B}$ outputs:

$$e((\frac{\sigma_\mathsf{t}^*}{g_1^{a\mathsf{sum_t}^*}})^{\frac{1}{r_t \sum \mathsf{tk}_i}}, g_2^a) = e((\frac{g_1^{cr_t b \sum \mathsf{tk}_i} g_1^{a\mathsf{sum_t}^*}}{g_1^{a\mathsf{sum_t}^*}})^{\frac{1}{r_t \sum \mathsf{tk}_i}}, g_2^a) =$$
$$e((g_1^{cr_t b \sum \mathsf{tk}_i})^{\frac{1}{r_t \sum \mathsf{tk}_i}}, g_2^a) = e(g_1^{bc}, g_2^a) = e(g_1, g_2)^{abc}$$

Let $\mathcal{A}^{\mathbf{AU1}}$ be the event when $\mathcal{A}$ successfully forges a **Type I forgery** σ_t for our **PUDA** protocol that happens with some non-negligible probability ϵ'. $\mathtt{event_0}$ is the event when $\mathsf{coin} = 0$ in the learning phase and $\mathtt{event_1}$ is the event when $\mathsf{coin} = 1$ in the challenge phase. Then $\Pr[\mathcal{B}^{\mathsf{BCDH}}] = \Pr[\mathtt{event_0}] \Pr[\mathtt{event_1}] \Pr[\mathcal{A}^{\mathbf{AU2}}] = \pi(1-\pi)^{\mathtt{q_H}-1}\epsilon'$, for $\mathtt{q_H}$ random oracle queries with the probability $\Pr[\mathsf{coin}(t) = 0] = \pi$. As such we ended up in a contradiction assuming the hardness of the BCDH assumption and finally $\Pr[\mathcal{A}^{\mathbf{AU1}}] \leq \epsilon_1$, where ϵ_1 is a negligible function.

Theorem 4. *Our scheme guarantees aggregate unforgeability against a* **Type II Forgery** *under the* LEOM *assumption.*

Proof. (Sketch) Here we show how an adversary $\mathcal{B}$ breaks the LEOM assumption by using an Aggregator $\mathcal{A}$ that provides a **Type II Forgery** with a non-negligible probability. Notably, adversary $\mathcal{B}$ simulates oracle $\mathcal{O}_{\mathsf{Setup}}$ as follows: It first picks secret encryptions keys $\{\mathsf{ek}_i\}_{i=1}^{n}$ and sets the corresponding decryption key $\mathsf{SK}_A = -\sum_{i=1}^{n}\mathsf{ek}_i$. Then, it forwards to $\mathcal{A}$ the public parameters $\mathcal{P} = (p, g_1, g_2, \mathbb{G}_1, \mathbb{G}_2)$, the public key $(\mathsf{vk}_1, \mathsf{vk}_2) = (g_2^{\sum_{i=1}^{n} k_i}, g_2^{a})$ of the $\mathcal{O}_{\mathsf{LEOM}}$ oracle and the secret key $\mathsf{SK}_A = -\sum_{i=1}^{n}\mathsf{ek}_i$.

Afterwards, when adversary $\mathcal{B}$ receives a query $(t, \mathsf{uid}_i, x_{i,t})$ for oracle $\mathcal{O}_{\mathsf{EncTag}}$, adversary $\mathcal{B}$ calls oracle $\mathcal{O}_{\mathsf{LEOM}}$ with the pair $(h_t = H(t), i, x_{i,t})$. Oracle $\mathcal{O}_{\mathsf{LEOM}}$ accordingly returns $h_t^{k_i} g_1^{a x_{i,t}}$ and adversary $\mathcal{B}$ outputs $\sigma_{i,t} = h_t^{k_i} g_1^{a x_{i,t}}$. Note that if we define the tag key tk_i of user $\mathcal{U}_i$ as k_i, then the tag $\sigma_{i,t} = h_t^{k_i} g_1^{a x_{i,t}}$ is computed correctly.

Eventually with a non-negligible advantage, Aggregator $\mathcal{A}$ outputs a **Type II Forgery** $(t^*, \mathsf{sum}_{t^*}, \sigma_{t^*})$ that verifies:

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

where t^* is previously queried by Aggregator $\mathcal{A}$ and $\mathsf{sum}_{t^*} \neq \sum_{i=1}^{n} x_{(i,t^*)}$.

It follows that $\mathcal{B}$ breaks the LEOM assumption with a non-negligible probability by outputting the tuple $(H(t^*), \mathsf{sum}_{t^*}, \sigma_{t^*})$. This leads to a contradiction under the LEOM assumption. We conclude that our scheme guarantees *aggregate unforgeability* for a **Type II Forgery** under the LEOM assumption.