Cryptonite: A Framework for Flexible Time-Series Secure Aggregation with Non-interactive Fault Recovery

Ryan Karl, Jonathan Takeshita, and Taeho Jung

University of Notre Dame, Notre Dame IN 46556, USA
{rkarl,jtakeshi,tjung}@nd.edu

Abstract. Private stream aggregation (PSA) allows an untrusted data aggregator to compute statistics over a set of multiple participants' data while ensuring the data remains private. Existing works rely on a trusted third party to enable an aggregator to achieve fault tolerance, that requires *interactive recovery*, but in the real world this may not be practical or secure. We develop a new formal framework for PSA that accounts for user faults, and can support *non-interactive recovery*, while still supporting strong individual privacy guarantees. We first must define a new level of security in the presence of faults and malicious adversaries because the existing definitions do not account for faults and the security implications of the recovery. After this we develop the first protocol that provably reaches this level of security, i.e., individual inputs are private even after the aggregator's recovery, and reach new levels of scalability and communication efficiency over existing work seeking to support fault tolerance. The techniques we develop are general, and can be used to augment any PSA scheme to support non-interactive fault recovery.

Keywords: Fault Tolerance · Trusted Hardware · Secure Aggregation

1 Introduction

Third-party analysis on private records is becoming more important due to widespread data collection for various analysis purposes in business, government, academia, etc.. This can be observed in many real life applications, such as the Smart Grid, Social Network Services, Location Based Services, etc. [14]. Given the great abundance of user-generated data and the collection of it in modern times, data analysis frameworks must be capable of processing queries over millions and sometimes billions of devices with little to no latency. While existing service providers support this over unencrypted data, data in its plaintext form often contains private information about individuals, and the publication of such data may violate privacy laws such as HIPPA, FERPA, GDPR, etc..

Within the context of many applications that process large amounts of data, it is paramount that fresh results be available to consumers, despite the presence of frequent system faults [20]. For example, web companies such as Facebook and LinkedIn execute daily data mining queries to analyze their latest web logs,

and online marketplace providers such as eBay and BetFair run fraud detection algorithms on real-time consumer trading activity [22]. Similarly, various types of failures are common in systems with user interactions, and the fault recovery must not affect performance adversely. Critically, due to the number of users participating in such protocols, the per-machine resource overhead of any fault tolerance mechanism should be low. Thus, such systems must be able to recover from failures without significantly impacting output accuracy, computation time expectations, or requiring interaction with unreliable/untrusted parties.

It is well known that existing work has proposed to support privacy preserving computation (secure multi-party computation (MPC), functional encryption (FE), perturbation, etc.) over multiple users' data. Of the existing techniques, Private Stream Aggregation (PSA) is very promising. PSA allows a third-party aggregator to receive encrypted values from multiple parties and compute an aggregate function without learning anything else, except what is learnable from the aggregate value. PSA is generally superior to other types of secure computation paradigms (e.g., MPC, FE) in large-scale applications involving time-series data because of its extremely low overhead and the ease of key management [13, 23]. Notably, PSA is non-interactive (i.e., users send their time-series data in a "stream" and only one message is sent per time interval) and asynchronous (i.e., users can leave after submitting their inputs), making it more efficient in communication than most existing alternative techniques [26]. However, existing solutions fail to achieve tolerance against faults during the aggregation without placing trust in the aggregators. We distinguish between *non-interactive* fault tolerance, which is the ability to recover from faults dynamically and "on the fly" without requiring extra messages be sent from/to faulted users or some trusted party, and *interactive* fault tolerance, which requires additional messages be exchanged to support recovery.

In this paper, we present a novel framework, Cryptonite, that allows any PSA scheme to gain non-interactive fault tolerance without significant additional overhead. There are many existing works that build ad-hoc solutions for this purpose that generally focus on providing one or a few of the following goals: privacy, efficiency, practical benefits such as permitting a user to drop in and out, or some type of interactive fault recovery mechanism. In contrast, our framework generalizes data aggregation, while still achieving traditional levels of performance and security, but more importantly, it introduces non-interactive recovery against faults to existing secure aggregation primitives without requiring users to trust the aggregator or requiring extra interaction.

This is a challenging problem to solve efficiently and securely, as most existing solutions require communicating with a trusted third party key dealer, which requires sending additional messages (generally two) during the protocol, greatly increasing the total overhead. A better solution would be *non-interactive* and would allow the aggregator to recover from a fault locally without sending additional messages, or requiring additional computation on the user end. However, a non-interactive protocol would need to guarantee correct function output with only one communication round. As a result, such a protocol would be by its

nature vulnerable to the residual function attack [15] in the standard model. In this attack, an adversary can repeatedly evaluate the function locally, while varying some inputs and fixing the inputs of others, to deduce the values entered by the participants. This vulnerability occurs because an aggregator that does not receive all of the users' encrypted inputs must be able to simulate acquiring such inputs, in order to complete the calculation. Existing work allows an aggregator to recover some partial data from the function they were to compute, but does so by sending a message to a trusted third party [8, 11] or aggregator to provide sensitive information that could harm an individual user's privacy if released publicly [6, 16]. Such existing work supports interactive fault tolerance simply by allowing the aggregator to evaluate an aggregation multiple times, which is essentially the residual function attack. This technique is insecure, and presents a serious privacy risk even if the data is protected with privacy preserving (e.g., differentially private) noise. We need a new, more rigorous notion of privacy that accounts for fault tolerance without sacrificing traditional security expectations.

In contrast, our scheme does not rely on any interaction with a third party, thus cutting down on communication, while also supporting partial aggregation among the surviving participants (thus achieving non-interactive fault tolerance), to maximize utility for the aggregator. Our simulations show that the fault recovery mechanism introduces negligible extra overhead to a PSA scheme when no faults occur. More importantly, when faults occur, our framework allows the PSA to recover from faults much more efficiently than other fault recovery mechanisms for PSA. We achieve all of this while providing security in the presence of stronger adversaries, and our scheme can be easily extended to support a wider variety of functions, such as max, average, etc. [11, 24]. Our goals in designing this framework are to 1) devise a system that is able to recover from failures without significantly impacting processing result accuracy or computation/communication time expectations. 2) maximize user's trust in the protocol by requiring that any servers used to facilitate the aggregation not be trusted by the users, and 3) enable computations at aggregate levels while still protecting any individual level data. Any system seeking to support such goals should provide a formal privacy analysis to demonstrate that the mechanism achieves the above privacy goals. Our contributions are as follows:

1. We identify the trust issues of aggregators when fault tolerance needs to be achieved during secure aggregation without extra interactions, and define a new, stronger level of privacy in the presence of faults and malicious aggregators – *fault-tolerable aggregator obliviousness*.
2. We develop a new formal framework for PSA that accounts for user faults, and develop general techniques that can be used to augment any PSA scheme to support *non-interactive fault recovery*.
3. We develop the first protocol that provably reaches this level of privacy using a Trusted Execution Environment (TEE). Rather than compute everything in the TEE, we minimize the performance impact from the TEE by outsourcing computationally intensive work to an untrusted domain for efficiency, while still allowing for strong privacy guarantees.

4. We demonstrate new levels of scalability and communication efficiency over existing work that supports interactive fault tolerance. Our anonymized code is public and available at: `https://anonymous.4open.science/r/053b49b8-8e83-4a39-ba78-8a0453feca2e/`.

2 Related Work

Recently there has been interest in constructing PSA systems that allow for dynamic user groups or interactive fault tolerance, that are similar to fault-tolerable deterministic threshold signatures [21]. Fault tolerance in this context is the property that in the event that a user or group of users do not send data to the aggregator, either due to a natural failure or a malicious act, the aggregator can still recover a partial sum over the remaining users' messages that were successfully sent. There are primarily two existing paradigms for this.

(1) Recovery via trusted parties: In the first [1, 2, 11, 16], the aggregator communicates with an independent third party to notify them of the fault, and the third party provides the inputs to the aggregator to allow for the successful completion of the protocol for each aggregation. Since the third party knows the secrets assigned to every node, if some nodes fail to submit data, the aggregator asks the dealer to submit synthetic data on behalf of those failed nodes. This method incurs a round trip communication overhead between the key dealer and the aggregator for each aggregation (i.e., interactive). Some researchers [16] used a ring based construction to improve efficiency, but had to interact with a third party to recover from faults, which can lead to high communication delay. Other work [1, 2] explored using elliptic curves to improve the overhead of communication and computation, while still supporting interactive fault tolerance, but this requires that some trusted, independent third parties be communicated with each round for fault recovery. Similar work explored outsourcing expensive computations to the cloud [11] to support a wider variety of functions instead of just sum, such as min, average, etc., but they also require interactions with trusted third parties.

(2) Recovery via input buffering: In the second paradigm [3,7,8], users buffer their inputs that they send to the aggregator. Essentially, in this method users send a set of ciphertexts corresponding to several timestamps/inputs to the aggregator. Thus, if a user fails to communicate in the future, the aggregator can utilize these ciphertexts to complete the aggregation and cancel out the noise needed to recover the partial sum. This increases the overall message size by a factor of how many rounds the user buffers their input (to buffer for 2 rounds, the size of the message is twice as large, etc.). One of the first works explicitly interested in supporting interactive fault tolerance [6] used a novel approach based on a binary interval tree technique to reduce the communication cost for joins and leaves, via input buffering. However, their scheme has a high aggregation error, which leads to the poor utility of the aggregate. Another technique [30] for buffering future ciphertexts was developed to reduce communication overhead, and was later made more efficient and scalable [3, 7]. A security-enhanced data aggregation scheme [8]

with interactive fault tolerance based on Paillier's encryption scheme has been proposed. Unfortunately, internal attacks are not considered in the above data aggregation schemes thereby allowing internal attackers to access the consumers' data. This was later improved [19] by leveraging lifted El-Gamal encryption to improve performance, and authentication methods were added for message integrity, although the vulnerability to internal attackers was left as an open problem. Later work [9] investigated using techniques to make key generation non-interactive. There has been some work that tries to solve this problem by allowing users to communicate with each other if a fault is detected to restart the protocol [28, 31], but we are interested in developing better approaches that do not require interaction among users, as this can lead to significant overhead and scalability issues.

Advantage of our work: The aforementioned schemes are either inefficient, fail to achieve non-interactive fault tolerance (i.e. extra messages must be sent to trusted parties), and/or are insecure against the residual function attack. In contrast, our scheme supports non-interactive fault tolerance, thus cutting down on communication, while also supporting partial aggregation among the surviving participants without introducing residual function attack vulnerabilities.

Orthogonal work: Defending against users that lie about their values to pollute the final output is outside the scope of the paper, but one possible defense is for each user to use a non-interactive zero-knowledge proof to prove the encrypted input is either in a valid range or an already-committed value.

Common misconceptions: Note that it is not possible to simply leverage historical data, or utilize machine learning techniques to estimate possible inputs of faulted users and use the inferred inputs to recover the final aggregation. This is because, to have provable security guarantees, the ciphertexts shared with the aggregator in the PSA are computationally indistinguishable from random numbers. Therefore, no inference approaches can gain meaningful information from the ciphertexts to predict and recover the missing inputs (e.g., due to faults).

3 Preliminaries: Private Stream Aggregation

The field of PSA seeks to solve the following problem. Suppose an aggregator wishes to calculate the sum of n users periodically. Let $x_i^{(t)}$(where $x_i^{(t)} \in \{0, 1, \ldots, \Delta\}$) denote the data of user i in aggregation period t (where $t = 1, 2, 3, \ldots$). Then, the sum for time period t is $\sum_{i=1}^{n} x_i^{(t)}$. In some scenarios, in each time period t, each user i adds noise $r_i^{(t)}$ to their data $x_i^{(t)}$, encrypts the noisy data $\hat{x}_i^{(t)} = x_i^{(t)} + r_i^{(t)}$ with their key $k_i^{(t)}$ and sends the ciphertext to the aggregator. The aggregator can then use their own key, $k_0^{(t)}$ to decrypt the noisy sum $\sum_{i=1}^{n} \left(x_i^{(t)} + r_i^{(t)} \right)$. In this scenario, $k_i^{(t)}$ and $k_0^{(t)}$ change in every time period. Note that we focus on the aggregation scheme over the same time period and omit the t to save space when the context is clear. We also do not add noise $r_i^{(t)}$ for simplicity of presentation. We assume that every user communicates with the aggregator via a wireless connection, but note that in our setup there is no need

for users to communicate with each other. We assume that time is synchronized among nodes. Generally speaking, for a private aggregation protocol to be secure, it must achieve three properties: 1) the aggregator cannot achieve any meaningful intermediate results (i.e. they learn the final noisy sum but nothing else), 2) the scheme is aggregator oblivious (a party without the aggregator learns nothing), and 3) the scheme achieves differential privacy. Note that requirement 3 is needed in some contexts where it is assumed the accurate sum may leak user privacy in presence of side information. Thus, the aggregator is only allowed to obtain a noisy sum (the accurate sum plus noise).

4 New Notion of Security

To achieve a meaningful level of security, current aggregation schemes strive to guarantee *aggregator obliviousness* which is informally defined as follows:

Definition 1 (Aggregator Obliviousness).
Assuming that each honest participant p_i only encrypts once in each time period, a secure aggregation scheme achieves aggregator obliviousness if: 1) the aggregator can only learn the final aggregate for each time period, 2) without knowing the aggregator key, no one can learn anything about the encrypted data, even if several users collude, and 3) if the aggregator colludes with a subset of the users, or if a subset of the encrypted data has been leaked, the aggregator learns no additional information about the honest participants' individual data, beyond what can be inferred by the final aggregation.

While this definition is useful in schemes that do not consider fault tolerance, it becomes less useful once faults occur and need to be recovered without interactions. To recover from a fault without interactions, an aggregator must be able to generate synthetic input from any user to complete the calculation. This is because PSA schemes must encode data in such a way that no partial information can be gained unless every participant's key is used in the final aggregation (for the sake of aggregator obliviousness). However, this actually violates the aggregator obliviousness, since to recover from faults without interactions, an aggregator must be able to calculate any partial sums, which would allow the aggregator to deduce everyone's input by subtracting the partial sums (i.e., residual function attack). Introducing differential privacy is not sufficient as the noise must be significantly larger than that in the PSA schemes with computational differential privacy ($O(n)$ where n is the number of users rather than $O(1)$ in existing schemes [4, 23]) to prevent such residual function attack. Many applications cannot afford to operate over results with excessive noise, as the significant loss in data accuracy prevents the subsequent data analysis from having any utility to analysts [10]. Therefore, we are primarily interested in investigating how to design a system where the residual function attack is not possible even without differentially private noise being introduced to the input.

Note that introducing computational differential privacy [4, 23] on top of such a system is trivial. Users can locally add calibrated noise to their inputs before

encryption for the sake of computational differential privacy. This is independent from the rest of the PSA and our framework, therefore we omit the description due to the space limit.

Issues with Existing Techniques: Existing works try to avoid this issue by introducing a trusted, independent third party that can assist the untrusted aggregator with completing the protocol. This is facilitated by allowing the aggregator to request the third party provide the keys or ciphertexts the user was supposed to send to the aggergator so that they can complete the calculation and determine the partial sum. While there may be scenarios where this adversary model is acceptable, in the real world, it may be difficult or even impossible to find such a trusted third party (arguably, if such a third party exists it may be easier for users to send their plaintexts directly to them to speed up processing). More specifically, we are interested in supporting privacy in a scenario where there are no independent third parties involved in fault recovery. In this setting, the two existing methods of achieving fault tolerance are ineffective, as they are vulnerable to the residual function attack. An aggregator can compute the same function over different inputs, compute the difference between the final outputs, to infer individual values inputted by different users.

Consider the first family of fault tolerant protocols, which allow the aggregator to ask an independent third party to provide the information needed to recover the output. If such an third party is not trusted, the aggregator can request all of the private information from this third party and recover every party's individual input via the residual function attack. We also note that even if this third party is trusted, in existing work, it is unclear how to prevent the untrusted aggregator from lying about users faulting, even if they complete their part of the protocol, to recover the synthetic inputs they need to launch the residual function attack. The second family of fault tolerant protocols, where users buffer future inputs to the aggregator is similarly vulnerable. If there is no trusted third party, the aggregator can simply request the buffered inputs, even if a user does not fault, to execute the residual function attack. Similarly, even if the third party that stores the buffer is trusted, the security guarantee is somewhat unclear, as the aggregator can lie about the fault status of users to recover the synthetic input needed to execute the residual function attack. Clearly, we need a new definition of aggregator obliviousness within the context of fault tolerant systems, that accounts for such scenarios. By extending the existing definitions [12, 23], we define the *fault-tolerable aggregator obliviousness* as follows:

Definition 2 (Fault-Tolerable Aggregator Obliviousness).

Define a set of users $i \in N$, where $0 \leq i \leq |N|$, where the subset of users that fault is denoted U and the set of users that do not fault is denoted J, were $N = U \cup J$. A set of users N participating in a secure aggregation scheme β, with public parameters $params$, during timestep t, whose inputs and secret keys are denoted x_i and sk_i respectively, achieve aggregator obliviousness with fault tolerance if no probabilistic polynomial-time adversary has more than negligible advantage in winning the below security game:

Setup : Challenger runs a Setup algorithm, and returns the public parameters *params* to the adversary.

Queries: The adversary makes the following three types of queries:

1. Encrypt: The adversary may specify (i, t, x) and ask for the ciphertext. Challenger returns the ciphertext affiliated with $\text{Enc}(sk_i, t, x_i)$ to the adversary.

2. Compromise: The adversary specifies an integer $i \in \{0, \ldots, |N|\}$ If $i = 0$, the challenger returns the aggregator key sk_0 to the adversary. If $i \neq 0$, the challenger returns sk_i the secret key for the i^{th} participant, to the adversary.

3. Challenge: This query can be made only once throughout the game. The adversary specifies a set of participants Q and a time t^* Any $q \in Q$ must not have been compromised at the end of the game. The adversary also specifies a subset of Q denoted Y of users they *claim* faulted (i.e. a user in Y may not have actually faulted). For each user $q \in Q$ the adversary chooses four plaintexts $(x_q), (x'_q), (x_y), (x'_y)$. The challenger flips a random bit b. If $b = 0$, the challenger computes $\forall q \in Q \backslash Y : \text{Enc}\left(sk_q, t^*, x_q\right)$, $\forall y \in Y : \text{Enc}\left(sk_y, t^*, x_y\right)$ and returns the ciphertexts to the adversary. If $b = 1$, the challenger computes and returns the ciphertexts $\forall q \in Q \backslash Y : \text{Enc}\left(sk_q, t^*, x'_q\right)$, $\forall y \in Y : \text{Enc}\left(sk_y, t^*, x'_y\right)$ instead.

Guess: The adversary outputs a guess of whether b is 0 or 1. We say that the adversary wins the game if they correctly guess b and the following condition holds. Let $K \subseteq N$ denote the set of compromised participants at the end of the game. Let $M \subseteq N$ denote the set of participants for whom an Encrypt query has been made on time t^* by the end of the game. Let $Q \subseteq N$ denote the set of (uncompromised) participants specified in the Challenge phase. If $Q = \overline{K \cup M} := N \backslash (K \cup M)$, $J \cup Y \neq \emptyset$, and the adversary has compromised the aggregator key, the following condition must be met: $\sum_{q \in Q} x_q + \sum_{y \in Y} x_y = \sum_{q \in Q} x'_q + \sum_{y \in Y} x'_y$.

Essentially we say that a secure aggregation scheme achieves fault-tolerable aggregator obliviousness if: 1) the aggregator can only learn one sum for each time period, even if a subset of users fault, 2) without knowing the aggregator key, no one can learn anything about the encrypted data, even if several users collude, and 3) if the aggregator colludes with a subset of the users, or if a subset of the encrypted data has been leaked, the aggregator learns no additional information about the honest participants' individual data. This better captures the requirements needed to protect against the residual function attack, since at least two separate function evaluations must be completed by an adversary for the attack to be successful. In the previous definition, multiple sums could still be calculated by an attacker, while still fulfilling the requirements of the definition. Also, to be fault tolerant, multiple ciphertexts associated with one user need to be available to the aggregator, so making an assumption that only one ciphertext is associated with each user may limit the utility of the previous definition, as if a user faults, another ciphertext associated with the user, but generated independently from the user may be needed for recovery.

Framework β

Setup(1^λ) : Takes in a security parameter λ, and outputs public parameters $param$, a private key sk_i for each participant, as well as a aggregator key sk_0 needed for decryption of aggregate statistics in each time period. Each participant i obtains the private key sk_i, and the data aggregator obtains the key sk_0 at the end of this algorithm.

Enc$(param, sk_i, t, x_i)$: During time step t, each participant calls the Enc algorithm to encode its data x_i via sk_i. The result is an encryption of x_i using the additive key homomorphism from the chosen PSA, denoted $ENC(x_i)$ or c_i.

FaultRecover(J, U, t): The fault recovery algorithm takes in the set of all the IDs of all the users that the aggregator reports as having faulted, denoted J, during time period t, along with the IDs of all of the users that successfully sent their encrypted data U. The algorithm then verifies that the two sets of users are disjoint. If the sets are not disjoint the algorithm outputs nothing and the protocol aborts. If the two sets are disjoint, the algorithm outputs for all $j \in J$ the ciphertexts corresponding to an encryption of 0 as c_j. This algorithm can only be called once for each time period.

AggrDec $(param, sk_0, t, c_u \forall u \in U, c_j \forall j \in J)$ Takes in the public parameters $param$, a key sk_0, the ciphertexts for all users in the set of users that did not fault $u \in U$ as c_u, and the ciphertexts for all users in the set of users that did fault $j \in J$ as c_j, for the same time period t. For each $i \in N$ where N is the union of U and J let $c_i = \text{Enc}\,(sk_i, t, x_i)$. Let $\mathbf{x} := (x_1, \ldots, x_n)$. The decryption algorithm outputs $f(\mathbf{x})$.

Fig. 1. Our Framework

5 Cryptonite: A Novel Framework for Any PSA Scheme

5.1 The Framework Definition

To achieve the above notion of privacy, we design a new secure aggregation framework β in Figure 1, that addresses fault tolerance. At a high level, our framework follows the same general procedure used by existing PSA schemes based on additive key homomorphism to distribute private keys to each participant during **Setup**. Following this, each user leverages their private key to encrypt their private data during **Enc**. After the aggregator receives all the users' ciphertexts, the aggregator can optionally invoke a fault recovery mechanism, **FaultRecover**, for a subset of users they claim faulted. This mechanism will verify that the aggregator's claim is accurate, and they did not claim a user faulted when they in fact received their ciphertext. If it is found the aggregator made a false claim the protocol aborts. After this, aggregator can recover the final aggregation result of the data it successfully received from the users with **AggrDec**. We formalize the fault recovery mechanism so that we can better enforce that protocols will not be vulnerable to the residual function attack. This framework supports the same general functionality as the previous framework, but allows the aggregator to recover the needed information regarding users who fault to complete the protocol in a privacy preserving manner as described in Definition 2.

5.2 Framework Instantiation

To formally investigate the correctness and the security of our framework, we instantiate a precise protocol, θ, using Cryptonite. We first present our basic approach, and we later overcome performance limitations in our optimized version, which is presented in the following section. The greatest challenge we face when designing this protocol is how to guarantee that the aggregator cannot act maliciously and acquire the synthetic data it needs to execute a residual function attack. Since any actions taken by an aggregator must be tightly controlled to support non-interactive fault recovery, and previous work has shown achieving specific security guarantees in certain non-interactive protocols is impossible in the standard model without additional hardware assumptions [15], a natural choice to support this functionality is to leverage trusted hardware, such as a Trusted Execution Environment (TEE), combined with PSA based on additive key homomorphism. We summarize the requisite background below.

Trusted Hardware: Note that our framework instantiation can work with any TEE in the domain of trusted hardware, but we chose the Intel SGX for our concrete instantiation. Trusted hardware is a broad term used to describe any hardware that can be certified to perform according to a specific set of requirements, often in an adversarial scenario. One of the most prevalent in modern computing is Intel SGX. Intel SGX is a set of new CPU instructions that can be used by applications to set aside private regions of code and data. It allows developers to (among other things) protect sensitive data from unauthorized access or modification by malicious software running at superior privilege levels. To allow this, the CPU protects an isolated region of memory called Processor Reserved Memory (PRM) against other non-enclave memory accesses, including the kernel, hypervisor, etc.. Sensitive code and data is encrypted and stored as 4KB pages in the Enclave Page Cache (EPC), a region inside the PRM. Even though EPC pages are allocated and mapped to frames by the OS kernel, page-level encryption guarantees confidentiality and integrity. In addition, to provide access protection to the EPC pages, the CPU maintains an Enclave Page Cache Map (EPCM) that stores security attributes and metadata associated with EPC pages. This allows for strong privacy and integrity guarantees if applications can be written in a two part model, where applications must be split into a secure part and a non-secure part. The application can then launch an enclave, that is placed in protected memory, which allows user-level code to define private segments of memory, whose contents are protected and unable to be either read or saved by any process outside the enclave. Enclave entry points are defined during compilation. The secure execution environment is part of the host process, and the application contains its own code, data, and the enclave, but the enclave contains its own code and data too. An enclave can access its application's memory, but not vice versa, due to a combination of software and hardware cryptographic primitives.

Elliptic Curves: Note that our framework instantiation can work with any PSA that is based on additive key homomorphism [25], but we chose elliptic

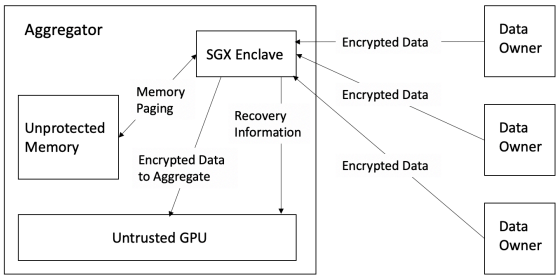


Fig. 2. System Diagram

curve cryptography (ECC) for our concrete instantiation. ECC provides the same level of security as RSA, Paillier, or discrete logarithm systems over Z_p with considerably shorter operands (approximately 160–256 bit vs. 1024–3072 bit), which results in shorter ciphertexts and signatures. As a result, in many cases, ECC has performance advantages over other public-key algorithms [5].

Protocol θ: Note that [25] uses a key-homomorphic weak PRF to construct PSA, and uses the seminal PSA of Shi et al. [23] as an example. Thus we choose to instantiate our framework with theirs, so that our framework can be adapted to turn any PSA that is based on additive key homomorphism into a fault-tolerable version. When the context is clear, we sometimes use standard addition and multiplication operators, as done in previous PSA papers [6, 23], when operating over ciphertexts, for simplicity of presentation. Let $\mathbb{G}$ denote a cyclic group of prime order p for which Decisional Diffie-Hellman is hard. Let $H : \mathbb{Z} \to \mathbb{G}$ denote a hash function modeled as a random oracle. We assume the aggregator is equipped with an Intel SGX, and model our system design in Figure 2.

Setup(1^λ) : Each user first performs attestation with the aggregator's Intel SGX, to verify it will faithfully execute the protocol (this is a one time process). The Intel SGX performs key generation, and chooses a random generator $g \in \mathbb{G}$, and $n+1$ random secrets $s_0, s_1, \ldots, s_n \in \mathbb{Z}_p$ such that $s_0 + s_1 + s_2 + \ldots + s_n = 0$. The public parameters $param := g$. The aggregator obtains the key $sk_0 := s_0$ and participant i obtains the secret key $sk_i := s_i$. For practical purposes, we can use secret shares that sum to zero as secret keys.

Enc$(param, sk_i, t, x_i)$: For participant i to encrypt a value $x \in \mathbb{Z}_p$ for time step t, they compute the following ciphertext $c \longleftarrow g^{x_i} \cdot H(t)^{sk_i}$, where $H(t)$ denotes the hash of t that maps t to an elliptic curve. Note, after this the user sends its ciphertext and unique id to the aggregator's SGX.

FaultRecover(c_j, c_u, t): Here, after the time period has ended, within the Intel SGX, we check each ciphertext that was received against a hash table of all users who participated in the setup process, and record which users failed to respond within the time window. Note this process cannot be tampered with from outside the enclave. Then, since the Intel SGX has each user's secret key, it can compute $c \longleftarrow g^0 \cdot H(t)^{sk_j}$ for all users $j \in J$. Notice that a nice property of this setup is that if a user is late and sends a ciphertext associated with time period t after that time period has passed, the Intel SGX can simply discard it and there is no danger of it being leaked to the aggregator.

AggrDec$(param, sk_0, t, c_j, c_u)$: Compute within the enclave (note $N = U \cup J$) $V \leftarrow H(t)^{\text{sk}_0} \prod_{i=1}^{n} c_i$. To decrypt the sum, we can leverage Pollard's lambda method, as done in previous works [23], to compute the discrete log of V base g. This method requires decryption time roughly square root in the plaintext space, although in general solving the discrete log is highly parallelizable and can be done efficiently in practice as long as the plaintext is small [6].

Note that this construction is secure under Definition 2, and we can prove this via a security game, using proof techniques from existing work [23]. We include the full proof in Appendix A, and sketch it here for completeness. Essentially, assuming that the Decisional Diffie-Hellman problem is hard in the group $\mathbb{G}$ and that the hash function H is a random oracle, we can prove that the above construction satisfies aggregator oblivious security with fault tolerance, by showing via reduction to a series of hybrid games that the game described above is hard to win for our scheme. More specifically,to prove the theorem, we will modify the aggregator oblivious security game as such. In the **Encrypt** queries, if the adversary submits a request for some tuple (q, x, t^*) where t^* is the time step specified in the **Challenge** phase, the challenger treats this as a **Compromise** query, and simply returns the sk_q to the adversary. Given sk_q, the adversary can compute the requested ciphertext. The adversary has access to a the functionality, **FaultRecover**, that can only be called once (since this is enforced via trusted hardware), which takes in a set of users that have not been compromised ($j \in J$), and returns the set of ciphertexts that correspond to those users encrypting 0. This modification actually gives more power to the adversary. Note that this protocol is not vulnerable to the residual function attack, as the adversary cannot access multiple ciphertexts associated with a user for a given timestamp. Here, the individual ciphertexts are sent into the enclave, which can independently handle the computations needed for fault recovery in an isolated environment that cannot be spoofed or tampered with by an attacker (unlike in the previously discussed techniques that provide fault tolerance that requires additional communication rounds). Thus, the fault recovery process can be performed in a secure, non-interactive way, that removes the opportunity for an attacker to spoof the fault recovery to obtain an encryption of 0 for a user, even when the user participates and does not fault, such that the attacker can perform the residual function attack by utilizing both ciphertexts to deduce the user's plaintext input. Achieving differential privacy is not the primary focus of this paper, but we can easily adapt the methods of existing works if needed [6, 23].

A More Efficient Protocol The above protocol achieves security according to Definition 2, but it incurs additional computational overhead since the aggregation is done inside the TEE. It would be better if we could outsource the aggregation computation to the untrusted aggregator to improve performance and avoid the MEE's overhead. We can accomplish this by following the same **Setup** procedure as before, but instead having users send two messages simultaneously. They can send their ciphertext (i.e. the result of **Enc**) to the untrusted aggregator, and also send one separate message to the Intel SGX to indicate they are participating in the protocol. Intuitively, the aggregator can simultaneously begin the partial

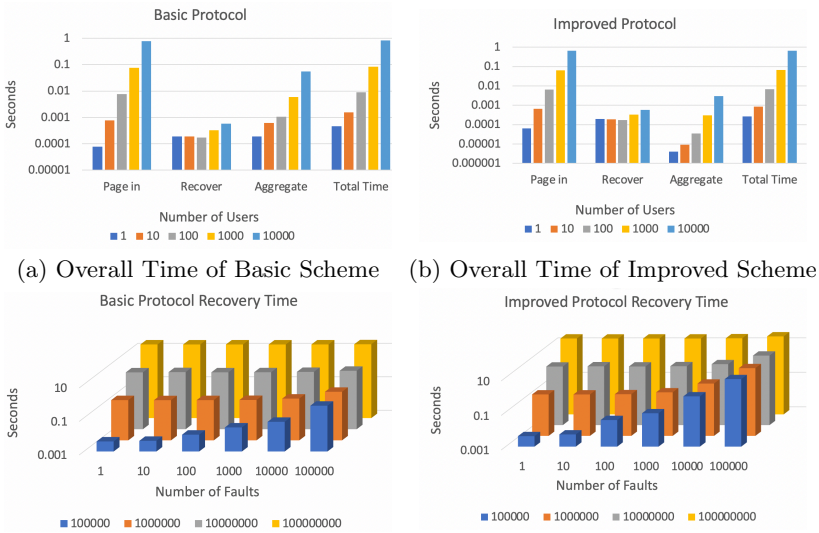
summation of the ciphertexts of the users that did not fault outside the TEE (by calling **AggrDec**), while inside the SGX, **FaultRecover** is run to determine which users faulted and computes their synthetic ciphertexts which are sent out of the TEE to the aggregator. In this way, the somewhat expensive aggregation step can be done on more powerful, albeit untrusted hardware (e.g., GPU, FPGA), that has better access to parallel computing resources, without compromising security. We note that this scheme is not secure if the adversary can disrupt communication between the users and the Intel SGX, but we can solve this by simply having all users send their ciphertexts signed with a digital signature directly to the SGX first, instead of just the separate message. Then the SGX can output the users' ciphertexts who did not fault to the untrusted space controlled by the aggregator, along with the synthetic data used to overcome existing, verified faults, which can be more efficiently aggregated outside the enclave.

Outsourcing to Parallel-friendly Processors It may seem more efficient to simply send plaintext data to an SGX enclave to be aggregated, but it is known that Intel SGX has difficulties exploiting multi-threading due to the lack of common synchronization primitive support often found on traditional operating systems [18] (threading can also introduce security vulnerabilities [29]). Also, TEEs have been shown to run common functionalities over an order of magnitude slower than what can be achieved on comparable untrusted hardware, due to the overhead of computing within the enclave [18], and performing a large number of context switches to send each user's data into the TEE can add serious overhead, especially in a big data setting. Overall performance can be improved if we minimize the number of context switches and outsource the aggregation step (i.e., **AggrDec** over inputs without faults) to processors with high parallel computing ability (e.g., many-core CPUs, GPUs, or FPGAs), because the additions of **AggrDec** are perfectly parallelizable.

PSA Schemes Requiring Trusted Parties In PSA schemes, the **Setup** is run only once and in a trusted manner [4, 12, 23]. This is typically accomplished through the use of an additional trusted third party key dealer or secure multiparty computation. However, with our framework, this can be replaced with the TEE, since the integrity of private key generation that is secure from eavesdropping will be guaranteed via remote attestation. Thus, our framework can remove the reliance on an external trusted third party in our PSA building block.

6 Experiments

To better understand the practical performance of our protocol we ran experiments using C++11 that simulated having thousands of users run our protocol, as is standard in the literature [6, 16]. For these tests, we used a workstation running Ubuntu 16.04 LTS equipped with a Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz (6 cores and 12 threads) with Intel SGX support. We did not leverage GPUs/FPGAs because we did not have access to computers equipped with both Intel SGX and GPUS/FPGAs. During tests we simulated the BGN [5] cryptosystem over



(a) Overall Time of Basic Scheme (b) Overall Time of Improved Scheme

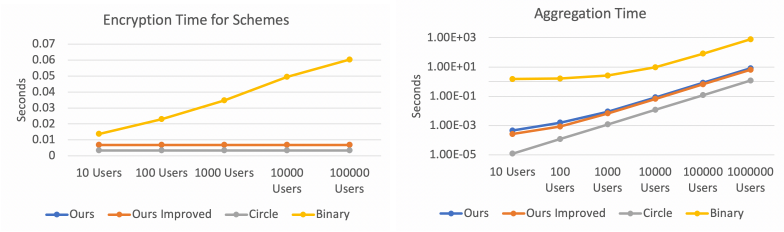
(c) Recovery Time of Basic Scheme (d) Recovery Time of Improved Scheme

Fig. 3. Experimental Results

Koblitz curve secp160k1, that offers 160 bits of security. We used time series data from the 3W dataset from the UCI machine learning data repository [27], and report the average time for 50 trials for each experiment.

Although there are space constraints associated with an Enclave, and a program that exceeds the allocated space incurs paging overhead, we found that in practice we could efficiently process aggregation over large numbers of users without major issues. Note the data footprint per user is roughly 100 bytes, and since in practice we can fit roughly 93 mb of data into an Enclave before triggering paging, we conservatively estimate that we can support about 900,000 users per Enclave, assuming we can fit the remaining program logic and metadata into roughly 3 mb. Since Intel plans to support Enclaves up to 1 terabyte in size in upcoming releases, we anticipate this being less of an issue in the future [17].

Basic Scheme: The results for our basic scheme, assuming no users fault, are shown in Figure 3a. It is interesting to note that in all cases the overall time is dominated by the overhead of paging into and out of the enclave, and other important operations, such as performing the aggregation, only minimally contribute to the overall runtime. This makes sense, as it has been documented that these operations are comparatively expensive, due to the expensive cryptographic operations involved and the time needed to marshal the data. However, our results show that the overall time scales well in the presence of a large number of users. For instance our protocol takes only a few seconds to finish when there are 100,000 users, assuming the setup step is precomputed. We report the additional time needed to recover from faults in Figure 3c. We notice that since the dummy ciphertexts can be precomputed, the amount of time needed to recover is dominated by the time needed to traverse the hash table to determine which users faulted. As a



(a) Encryption Comparison (b) Aggregation Comparison

Fig. 4. Experimental Comparison Results

result, the more users that are involved in the protocol, the longer this process takes. However, we note that even in the worst case, when many thousands of users fault, the additional recovery time is under 30 seconds, which is practical in our applications. Unlike existing work that requires additional communication to support fault recovery, since we leverage a co-located TEE, we can remove the time needed for two communication rounds over existing works [8, 11], while still supporting strong privacy guarantees, to improve communication complexity.

Improved Scheme: Since the amount of time needed to page into the enclave leads to significant overhead, we designed an improved protocol to try and minimize the performance impact by safely outsourcing more computations to the untrusted adversary. We report our results, assuming no users fault, in Figure 3b. It is interesting to note that because we reduce the amount of enclave computation, we are able to improve our overall performance by approximately 26% in most cases. This makes sense, as we are able to reduce the amount of expensive enclave operations. We report the recovery time in Figure 3d. We note that the amount of time needed to recover is comparatively more expensive than in the basic scheme, as we need to marshal out of the enclave the dummy ciphertexts needed to recover from faults to the untrusted aggregator. As a result, this can sometimes increase the overall runtime by several seconds in the worst case practical scenario when many users fault. This is tolerable for our applications, but it does illustrate a tradeoff that may inform which scheme should be used on a case by case basis.

Comparison to Existing Work: We experimentally evaluated our work when compared to baseline techniques, and ran simulations to compare our scheme to two state of the art secure aggregation schemes: 1) the Binary scheme [6] which has users buffer their inputs that they send to the aggregator, and 2) the Ring scheme [16], which has the aggregator communicate with a trusted party, to support fault tolerance. Our technique outperforms these schemes in scenarios where faults occur, often by several orders of magnitude. We compare times reported in Fig. 4a,b.

We compare the encryption time and the aggregation time of the respective protocols, assuming no users fault, and vary the number of users. Note that the computational complexity of both of our schemes and the Ring Scheme is much less than that of the Binary scheme. This makes sense, as the Binary scheme requires that users compute $\lfloor (log_2(n) \rfloor$ encryptions per round where n

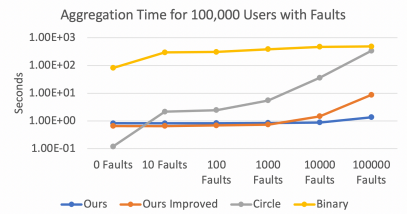


Fig. 5. Aggregation with Faults Comparison

is the number of users, in order to support fault tolerance via their binary tree mechanism, which negatively impacts the run time. In contrast, our schemes and the Ring scheme only require one encryption per round, and thus support more efficient encryption. Note that the Circle scheme is slightly faster than our scheme, as they leverage a more efficient cryptographic primitive, the HMAC. The HMAC also contributes to the improved performance of the Ring scheme over our schemes and the Binary technique during aggregation. Thus we conclude that our encryption scheme scales well in the presence of large numbers of users, but is roughly 2-6x slower that the state of the art Ring scheme if no users fault.

We also compared the aggregation time of the respective protocols when there are 100,000 participants, and varied the number of user faults. We report results in Figure 5. Note our schemes have the fastest overall run time when faults are introduced, sometimes by several orders of magnitude. This makes sense, as to recover from faults, we can efficiently interact with the on board TEE. In contrast the Ring Scheme incurs the roundtrip time of communicating with a trusted key dealer to collect the cryptographic keys needed to recover from the faults, and the Binary scheme must traverse the binary tree of ciphertexts it constructed to gather the ciphertexts it needs to cancel the appropriate randomness and recover the noisy plaintext. Unlike both of these schemes, we can recover from faults without either buffering ciphertexts, which causes increased communication overhead, or requiring additional rounds of communication, while supporting a stronger level of security overall, that does not require that we communicate with a trusted third party to recover from a fault.

7 Conclusion

We defined a new level of security for Private Stream Aggregation in the presence of faults and malicious adversaries. After describing a new framework for PSA that accounts for fault tolerance, we developed the first protocol that provably reaches this security level. Our simulations demonstrated our work reaches high levels of scalability and communication efficiency over existing work while supporting a higher level of security and better fault tolerance. Our techniques are general, and can extend any PSA scheme to support non-interactive fault recovery.

References

1. Bao, H., Lu, R.: Ddpft: Secure data aggregation scheme with differential privacy and fault tolerance. In: IEEE ICC 2015. pp. 7240–7245. IEEE (2015)

2. Bao, H., Lu, R.: A new differentially private data aggregation with fault tolerance for smart grid communications. IEEE IoT-J **2**(3), 248–258 (2015)
3. Bao, H., Lu, R.: A lightweight data aggregation scheme achieving privacy preservation and data integrity with differential privacy and fault tolerance. Peer-to-Peer Networking and Applications **10**(1), 106–121 (2017)
4. Becker, D., Guajardo, J., Zimmermann, K.H.: Revisiting private stream aggregation: Lattice-based psa. In: NDSS (2018)
5. Boneh, D., Goh, E.J., Nissim, K.: Evaluating 2-dnf formulas on ciphertexts. In: Theory of cryptography conference. pp. 325–341. Springer (2005)
6. Chan, T.H.H., Shi, E., Song, D.: Privacy-preserving stream aggregation with fault tolerance. In: FC. pp. 200–214. Springer (2012)
7. Chen, J., Ma, H., Zhao, D.: Private data aggregation with integrity assurance and fault tolerance for mobile crowd-sensing. Wireless Networks **23**(1), 131–144 (2017)
8. Chen, L., Lu, R., Cao, Z.: Pdaft: A privacy-preserving data aggregation scheme with fault tolerance for smart grid communications. Peer-to-Peer networking and applications **8**(6), 1122–1132 (2015)
9. Chotard, J., Sans, E.D., Gay, R., Phan, D.H., Pointcheval, D.: Decentralized multi-client functional encryption for inner product. In: Asiacrypt. pp. 703–732. Springer (2018)
10. Gillin, D.: The federal trade commission and internet privacy. Marketing Research **12**(3), 39 (2000)
11. Han, S., Zhao, S., Li, Q., Ju, C.H., Zhou, W.: Ppm-hda: privacy-preserving and multifunctional health data aggregation with fault tolerance. IEEE Transactions on Information Forensics and Security **11**(9), 1940–1955 (2015)
12. Joye, M., Libert, B.: A scalable scheme for privacy-preserving aggregation of time-series data. In: FC. pp. 111–125. Springer (2013)
13. Jung, T., Mao, X., Li, X., Tang, S., Gong, W., Zhang, L.: Privacy-preserving data aggregation without secure channel: multivariate polynomial evaluation. In: IEEE INFOCOM (2013)
14. Jung, T., Han, J., Li, X.Y.: Pda: Semantically secure time-series data analytics with dynamic subgroups. TDSC **PP**(99), 1–1 (2016)
15. Karl, R., Burchfield, T., Takeshita, J., Jung, T.: Non-interactive mpc with trusted hardware secure against residual function attacks. In: SecureComm. pp. 425–439. Springer (2019)
16. Li, Q., Cao, G.: Efficient privacy-preserving stream aggregation in mobile sensing with low aggregation error. In: PETS. pp. 60–81. Springer (2013)
17. Martin, D.: Intel xeon ice lake cpus to get sgx with expanded security features (2020)
18. Mofrad, S., Zhang, F., Lu, S., Shi, W.: A comparison study of intel sgx and amd memory encryption technology. In: ACM HASP. pp. 1–8 (2018)
19. Ni, J., Zhang, K., Alharbi, K., Lin, X., Zhang, N., Shen, X.S.: Differentially private smart metering with fault tolerance and range-based filtering. IEEE Transactions on Smart Grid **8**(5), 2483–2493 (2017)
20. Parikh, N., Sundaresan, N.: Scalable and near real-time burst detection from ecommerce queries. In: ACM SIGKDD. p. 972–980. KDD '08, ACM (2008)
21. Rabin, T.: A simplified approach to threshold and proactive rsa. In: Annual International Cryptology Conference. pp. 89–104. Springer (1998)
22. Russell, M.A.: Mining the social web. " O'Reilly Media, Inc." (2011)
23. Shi, E., Chan, T.H., Rieffel, E., Chow, R., Song, D.: Privacy-preserving aggregation of time-series data. In: Proc. NDSS. vol. 2, pp. 1–17. Citeseer (2011)

24. Shi, J., Zhang, R., Liu, Y., Zhang, Y.: Prisense: privacy-preserving data aggregation in people-centric urban sensing systems. In: INFOCOM. pp. 1–9. IEEE (2010)
25. Valovich, F.: Aggregation of time-series data under differential privacy. In: LATINCRYPT. pp. 249–270. Springer (2017)
26. Valovich, F., Aldà, F.: Computational differential privacy from lattice-based cryptography. In: NutMiC. pp. 121–141. Springer (2017)
27. Vargas, R.E.V., Munaro, C.J., Ciarelli, P.M., Medeiros, A.G., do Amaral, B.G., Barrionuevo, D.C., de Araújo, J.C.D., Ribeiro, J.L., Magalhães, L.P.: A realistic and public dataset with rare undesirable real events in oil wells. Journal of Petroleum Science and Engineering **181**, 106223 (2019)
28. Wang, X., Liu, Y., Choo, K.: Fault tolerant, multi-subset aggregation scheme for smart grid. IEEE Transactions on Industrial Informatics (2020)
29. Weichbrodt, N., Kurmus, A., Pietzuch, P., Kapitza, R.: Asyncshock: Exploiting synchronisation bugs in sgx enclaves. In: ESORICS. pp. 440–457. Springer (2016)
30. Won, J., Ma, C.Y., Yau, D.K., Rao, N.S.: Proactive fault-tolerant aggregation protocol for private smart metering. In: INFOCOM. pp. 2804–2812. IEEE (2014)
31. Xue, K., Yang, Q., Li, S., Wei, D.S., Peng, M., Memon, I., Hong, P.: Ppso: A privacy-preserving service outsourcing scheme for real-time pricing demand response in smart grid. IEEE Internet of Things Journal **6**(2), 2486–2496 (2018)

A Proof of Fault Tolerable Aggregator Obliviousness

Theorem 1. *Assuming that the Decisional Diffie-Hellman problem is hard in the group G and that the hash function H is a random oracle, then the above construction satisfies aggregator oblivious security with fault tolerance, as described in Definition 2.*

Proof. First, we prove that the following intermediate game is difficult to win, given that Decisional Diffie-Hellman is hard. Let $\mathbb{G}$ be a group of prime order p.

Setup: The challenger picks random generators $g, h \in \mathbb{G}$, and random $\alpha_0, \alpha_1, \ldots, \alpha_n \in \mathbb{Z}_p$ such that $\sum_{i=0}^{n} \alpha_i = 0$. The challenger gives the adversary: $g, h, g^{\alpha_0}, g^{\alpha_2}, \ldots, g^{\alpha_n}$.

Queries: The adversary can compromise users adaptively and ask for the value of α_i. The challenger returns α_i to the adversary when queried.

Challenge: The adversary selects an uncompromised set $Q \subseteq \{0, \ldots, N\}$, and specifies a subset of Q denoted Y of users they claim faulted, where $J = Y$ for the duration of the game. The challenger flips a random bit b. If $b = 0$, the challenger returns to the adversary $\{h^{\alpha_q} \mid q \in Q \backslash Y\}, \{h^{\alpha_y} \mid y \in Y\}$. If $b = 1$, the challenger picks $|Q|/|Y|$ random elements h'_q, for $q \in Q/Y$ and $|Y|$ random elements h'_y, for $y \in Y$ from the group $\mathbb{G}$, such that $\sum_{q \in Q} h'_q + \sum_{y \in Y} h'_y = \sum_{q \in Q} h^{\alpha_q} + \sum_{y \in Y} h^{\alpha_y}$. The challenger returns h'_q, for $q \in Q/Y$ and h'_y, for $y \in Y$ to the adversary. The adversary can make additional compromise queries, as described in the above step as they see fit.

Guess: The adversary guesses either $b = 0$ or 1. The adversary wins if they have not asked for any α_q for $q \in Q$, $Y = J$, and if they successfully guess b.

Lemma 1. *The above game is difficult for computationally bounded adversaries assuming Decisional Diffie Hellman is hard for group $\mathbb{G}$.*

We define the following sequence of hybrid games, and assume that the set Q specified by the adversary in the challenge stage is $Q = \{q_1, q_2, \ldots, q_m\}$. For simplicity, we write $(\beta_1, \ldots, \beta_m) := (\alpha_{q_1}, \ldots, \alpha_{q_m})$, and include Y within Q to save space. In $Game_d$, the challenger sends the following to the adversary: $R_1, R_2, \ldots, R_d, h^{\beta_{d+1}}, ..., h^{\beta_m}$. Here, each $R_q (q \in [d])$ means an independent fresh random number, and the following condition holds: $\prod_{1 \leq q \leq d} R_q = \prod_{1 \leq q \leq d} h^{\beta_q}$. Clearly $Game_1$ is equivalent to the case when $b = 0$, and $Game_{m-1}$ is equivalent to the case when $b = 1$. With the hybrid argument we can show that games $Game_{d-1}$ and $Game_d$ are computationally indistinguishable. To demonstrate this, we show that if, for some d, there exists a polynomial-time adversary $\mathcal{A}$ who can distinguish between $Game_{d-1}$ and $Game_d$, we can then construct an algorithm $\mathcal{B}$ which can solve the DDH problem.

Suppose $\mathcal{B}$ obtains a DDH tuple $\left(g, g^x, g^l, T\right)$. $\mathcal{B}$'s task is to decide whether $T = g^{xl}$ or whether T is a random element from $\mathbb{G}$. Now $\mathcal{B}$ randomly guesses two indices e and b to be the d^{th} and the $(d+1)^{\text{th}}$ values of the set Q specified by the adversary in the challenge phase. The guess is correct with probability $\frac{1}{N^2}$, and in case the guess is wrong, the algorithm $\mathcal{B}$ aborts. Now $\mathcal{B}$ picks random exponents $\{\alpha_q\}_{q \neq e, q \neq b}$ and sets $\alpha_b = x$ and $\alpha_e = -\sum_{q \neq e} \alpha_q$. Notice that $\mathcal{B}$ does not know the values of α_e and α_b, however, it can compute the values of $g^{\alpha_b} = g^x$ and $g^{\alpha_e} = \left(\prod_{q \neq e} g^{\alpha_q}\right)^{-1} = (g^x)^{-1} \cdot \prod_{q \neq e, q \neq b} g^{\alpha_q} \cdot \mathcal{B}$ gives $\mathcal{A}$ the tuple $\left(g, h = g^l, g^{\alpha_1}, \ldots, g^{\alpha_n}\right)$. If $\mathcal{A}$ asks for any exponent except α_e and $\alpha_b, \mathcal{B}$ returns the corresponding α_q value to $\mathcal{A}$; if $\mathcal{A}$ asks for α_e or α_b, the algorithm $\mathcal{B}$ aborts.

In the challenge phase, $\mathcal{A}$ submits a set $Q = \{q_1, q_2, \ldots q_m\}$. If e and b are not the d^{th} and the $(d+1)^{\text{th}}$ values of the set Q, i.e., if $q_d \neq e$ or $q_{d+1} \neq b$, the algorithm $\mathcal{B}$ aborts. If $q_d = e$ and $q_{d+1} = b$, then $\mathcal{B}$ returns to $\mathcal{A}$: $R_1, R_2, \ldots, R_{d-1}$, $(\prod_{q \notin \{q_1, \ldots, q_{d+1}\}} (g^l)^{\alpha_q} \cdot \prod_{q=1}^{d-1} R_q \cdot T)^{-1}$, T, and $(g^l)^{\alpha_{q_{d+2}}, \ldots, (g^l)^{\alpha_{q_m}}}$. Clearly if $T = g^{xl}$, then the above game is equivalent to $Game_{d-1}$. Otherwise, if $T \in_R \mathbb{G}$, then the above game is equivalent to $Game_d$. Thus, if $\mathcal{A}$ has a non-negligible advantage in guessing whether it is playing $Game_{d-1}$ or $Game_d$ and $\mathcal{B}$ could solve the DDH problem with non-negligible advantage.

Now to prove the theorem, we will modify the aggregator oblivious security game. In the **Encrypt** queries, if the adversary submits a request for some tuple (q, x, t^*) where t^* is the time step specified in the **Challenge** phase, the challenger treats this as a **Compromise** query, and simply returns the sk_q to the adversary. Given sk_q, the adversary can compute the requested ciphertext. The adversary has access to a the functionality, **FaultRecover**, that can only be called once (since this is enforced via trusted hardware), which takes in a set of users that have not been compromised ($j \in J$), and returns the set of ciphertexts that correspond to those users encrypting 0. Note that this modification actually gives more power to the adversary. From now on, we will assume that the adversary does not make any **Encrypt** queries for the time t^*.

Let $K \subseteq N$ denote the set of compromised participants. Let $\bar{K} := N \backslash K$ denote the set of uncompromised participants. Since we assume the aggregator is untrusted, we are interested in the case where $Q = \bar{K}$ or the aggregator key

has been compromised. We must show that the adversary cannot distinguish whether the challenger returns a true encryption of the plaintext submitted in the challenge stage, or a random tuple with the same aggregation.

Given an adversary $\mathcal{A}$ who can break the PSA game with non-negligible probability, we construct an algorithm $\mathcal{B}$ that can solve the above intermediate problem with non-negligible probability. $\mathcal{B}$ obtains from the challenger $\mathcal{C}$ the tuple $g, h, g^{\alpha_0}, g^{\alpha_1}, \ldots, g^{\alpha_n}$. $\mathcal{B}$ sets α_0 to be the aggregator's key, and $\alpha_1, \ldots, \alpha_n$ to be the secret keys of participants 1 through n respectively. Note $param$ is g.

Let q_H denote the total number of oracle queries made by the adversary $\mathcal{A}$ and by the algorithm $\mathcal{B}$ itself. $\mathcal{B}$ guesses at random an index $b \in [q_H]$. Suppose the input to the b^{th} random oracle query is t^*. The algorithm $\mathcal{B}$ assumes that t^* will be the challenge time step. If the guess is found to be wrong later, $\mathcal{B}$ aborts.

Hash Function Simulation: The adversary submits a hash query for the integer t. $\mathcal{B}$ first checks the list $\mathcal{L}$ to see if t has appeared in any entry (t, z). If so, $\mathcal{B}$ returns g^z to the adversary. Otherwise, if this is not the b^{th} query, $\mathcal{B}$ picks a random exponent z and returns g^z to the adversary, and saves (t, z) to a list $\mathcal{L}$. For the b^{th} query, $\mathcal{B}$ returns h.

Then the following **Queries** can take place:

•**Encrypt**: The adversary $\mathcal{A}$ submits an **Encrypt** query for the tuple (q, x, t). In the modified version of the game, we ensure that $t \neq t^*$, as otherwise, we simply treat it as a **Compromise** query. $\mathcal{B}$ checks if a hash query has been made on t, and if not, $\mathcal{B}$ makes a hash oracle query on t. Thus, $\mathcal{B}$ learns the discrete log of $H(t)$. Now $H(t) = g^z$, so $\mathcal{B}$ knows z, and since $\mathcal{B}$ also knows g^{α_q}, $\mathcal{B}$ can compute the ciphertext $g^x \cdot (g^z)^{\alpha_q}$ as $g^x \cdot (g^{\alpha_q})^z$.

•**Compromise**: $\mathcal{B}$ forwards $\mathcal{A}$'s query to its own challenger $\mathcal{C}$, and forwards the answer α_q to $\mathcal{A}$.

•**FaultRecover**: $\mathcal{B}$ forwards $\mathcal{A}$'s query to its own challenger $\mathcal{C}$, and forwards the set of ciphertexts (i.e. $\forall j \in J$, $c \longleftarrow g^0 \cdot H(t)^{sk_j}$)) to $\mathcal{A}$.

Challenge: The adversary $\mathcal{A}$ submits a set $N = J \cup Q$ and a time t^*, as well as plaintexts $\{x_q \mid q \in N\}$. If t^* does not agree with the value submitted in the b^{th} hash query, then $\mathcal{B}$ aborts. $\mathcal{B}$ submits the set Q in a **Challenge** query to its own challenger, and it obtains a tuple $\{T_q\}_{q \in N}$. The challenger returns the following ciphertexts to the adversary: $\forall q \in Q : g^{x_q} \cdot T_q$ (i.e. $c \longleftarrow g^{x_q} \cdot H(t)^{sk_q} \cdot T_q$).

More queries: Same as the **Query** stage.

Guess: If the adversary $\mathcal{A}$ guesses that $\mathcal{B}$ has returned a random tuple then $\mathcal{B}$ guesses $b' = 1$. Otherwise, $\mathcal{B}$ guesses that $b' = 0$

If the challenger $\mathcal{C}$ returns $\mathcal{B}$ a faithful Diffie-Hellman tuple $\forall q \in Q : T_q = h^{\alpha_q}$, then the ciphertext returned to the adversary $\mathcal{A}$ is a true encryption of the plaintext submitted by the adversary. Otherwise, if the challenger returns to $\mathcal{B}$ a random tuple, then the ciphertext returned to $\mathcal{A}$ is random under the product constraint.