# Cryptonite: A Framework for Flexible Time-Series Secure Aggregation with Online Fault Tolerance

Ryan Karl
University of Notre Dame
rkarl@nd.edu

Jonathan Takeshita
University of Notre Dame
jtakeshi@nd.edu

Taeho Jung
University of Notre Dame
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 party to enable an aggregator to achieve offline fault tolerance, but in the real world this may not be practical. We develop a new framework that supports PSA in a way that is robust to online user faults, while still supporting a strong guarantee on each individual's privacy. We first must define a new level of security in the presence of online faults and malicious adversaries because the existing definition does not account for online faults. After this we describe a general framework that allows existing work to reach this new level of security. Furthermore, we develop the first protocol that provably reaches this level of security by leveraging trusted hardware. After we develop a methodology to outsource computationally intensive work to higher performance devices, while still allowing for strong privacy, we reach new levels of scalability and communication efficiency over existing work seeking to support offline fault tolerance, and achieve differential privacy.

## Keywords

Secure aggregation, Time-series aggregation, Fault tolerance

## 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.[15]. Given the great abundance of data in modern times, the publication, sharing, and analysis of user-generated data has become common, and 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 legal agreements such as HIPPA, GDPR, etc.. Current best practices generally advocate for the sanitization of the sensitive information in the publication [21] (which may bring distortion to the data which leads to accuracy loss), or regulations to allow for the restriction of the usage of the published data [28] (which solely relies on legal or voluntary agreements, but does not provide any technical safeguards).

With such approaches, even if data recipients do not misbehave, they can still be vulnerable to third party attacks, and the utility of the data may be limited. A number of previous works focus on anonymizing the data [33] to protect data owners' identities, but this approach cannot provably guarantee individual privacy. In this paper, we focus on protecting the confidentiality of the data itself. Our secure aggregation scheme 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. It is well known that existing work has proposed to support such privacy preserving computation (secure multi-party computation, functional encryption, perturbation, etc.), but existing solutions fail to achieve one or more goals: 1) allowing for dynamically updating user groups between each aggregation, 2) low communication and computation overhead, 3) allowing for zero accuracy loss in secure aggregation, 4) tolerance against online faults during the aggregation without placing trust on the aggregators. Within the context of many applications (A/B Testing, Crowd Sensing, Smart Metering, Federated Learning, etc.), all four of these requirements must be satisfied, because 1) user groups change frequently in real-world applications (e.g., marketing analysts often need to run tests over multiple varying user groups), 2) the analysis may need to be run over millions or even billions of users efficiently to discover trends within big data, 3) any amount of noise or accuracy loss in the secure aggregation (except what may be needed for differential privacy) can greatly harm the predictive power of market research, and 4) failure of one or more participants during the aggregation (i.e., online faults), which will occur in large-scale studies, should not prevent analysts from conducting the analysis. In this paper, we present a novel framework, which allows a third-party to obliviously compute an aggregate analysis on private records provided by a group of users. While there are many existing works that build ad-hoc solutions for this purpose that generally focus on providing only one or a few of the goals above, such as privacy, efficiency, or practical benefits such as permitting a user to drop in and out, our framework generalizes data aggregation without sacrificing speed or security, and more importantly, it introduces tolerance against online faults to existing secure aggregation primitives without trusting the aggregator.

This is a challenging problem to solve in the context of non-interactive protocols with just one round of online communication, as correct function output must be guaranteed with only one round without sending additional messages to support recovery. As a result, these protocols are by their nature vulnerable to the residual function attack in the standard model, because a party that does not receive all of the users' encrypted inputs must be able to simulate acquiring such inputs, in order to complete the calculation. They must not have the opportunity to repeatedly evaluate the function locally, while varying some inputs and fixing the inputs of others, to deduce the values entered by the participants. Existing works that support offline fault tolerance simply allow the aggregator to evaluate aggregation multiple times, and this presents a privacy risk if the data is not sufficiently protected with privacy preserving noise.

We meet all of the above requirements, as our overall goal is to achieve a protocol that supports theoretic guarantees, while still being useful to market analysts. Our framework was designed with a special emphasis on deployability in the real world, and as such we are the first to design a system that allows for online fault tolerance but still supports the full privacy of users' data without relying on a trusted aggregator or trusted entity to allow for recovery. In this context, online fault tolerance refers to the ability to recover from faults dynamically "on the fly", whereas offline fault tolerance requires that additional messages must be sent to support recovery. Existing work allows an aggregator to recover some partial data from the function they were to compute, but does so by relying on sending a message to a trusted entity or aggregator to provide sensitive information that could harm an individual user's privacy if released publicly [9, 34]. Our scheme does not rely on any trusted entities or aggregators to support online fault tolerance, thus cutting down on communication, while also supporting partial aggregation among the surviving participants (and thus achieving the online fault tolerance), to maximize utility for the aggregator. Our scheme can support secure data aggregation over millions or even billions of devices without suffering from the high overhead traditionally associated with computing cryptographic operations over large numbers of devices by relying on a Trusted Execution Environment (TEE) [15]. Our experiments show the computational performance of our framework is comparable to existing works, and can remove up to two communication rounds over existing works [10, 14] seeking to support offline fault tolerance, while providing semantic security in the presence of stronger adversaries. Our contributions for this paper are as follows:

(1) We identify the trust issues of aggregators when online fault tolerance needs to be achieved in the secure aggregation without extra interactions, and define a new, stronger level of privacy in the presence of online faults and malicious aggregators – *fault-tolerable aggregator obliviousness*.
(2) We describe a general framework that allows existing work to reach this stronger level of security, by leveraging trusted hardware in the context of secure aggregation.
(3) We develop the first protocol that provably reaches this level of privacy using trusted hardware. Later we minimize the performance impact from trusted hardware, which is necessary for this level of security, by developing a methodology to outsource computationally intensive work to high performance hardware, while still allowing for strong privacy guarantees.
(4) We demonstrate new levels of scalability and communication efficiency over existing work that supports offline fault tolerance. Our code is public and available at: https://anonymous.4open.science/r/053b49b8-8e83-4a39-ba78-8a0453feca2e/.

Note that this scheme can be easily extended to support a wider variety of functions, such as min, max, average, etc. [14, 36].

## 2 Related Work

**Early Work:** The first work on aggregating encrypted data was done in the early 2000s [29] and further refined later [8]. The authors propose to use homomorphic encryption schemes to enable arithmetic operations over ciphertexts that need to be transmitted in a multi-hop manner. Although interesting from an application standpoint, it has restricted usability, due to its weak security level. Our own work is most similar to the problem of private stream aggregation, introduced by Shi et al. and Rastogi et al. [20, 21].

**Dynamic User Groups:** More recently, researchers have built on their work [4, 13, 27, 31, 34], as there has been an interest in constructing systems that allow for dynamic user groups and/or offline fault tolerance. One of the first works explicitly interested in supporting offline fault tolerance [9] used a novel approach based on a binary interval tree technique to reduce the communication cost for joins and leaves. However, their scheme has a high aggregation error, which leads to the poor utility of the aggregate. Later work [34] leveraged a ring based construction to improve efficiency, but had to rely on interacting with a trusted entity to recover from faults, which can lead to high communication overhead. Similar work explored outsourcing expensive computations to the cloud [14], within the context of secure health analytics, to support a wider variety of functions instead of just sum, such as min, average, etc.. However, they still rely on trusted entities to achieve offline fault tolerance.

**Improved Fault Tolerance:** Following this, a technique [24] for buffering future ciphertexts was used to reduce communication overhead, which was later made more efficient and scalable [5, 12]. A security-enhanced data aggregation scheme [10] with offline 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. Other work [2, 3] explored using Elliptic curve cryptography to improve the overhead of communication and computation, while still supporting offline fault tolerance, but this requires that some trusted, independent entities be communicated with each round to recover from faults. Note that there has been some orthogonal work investigating the online fault tolerance of users, where privacy is not the main focus of the research effort [1, 16]. All of these schemes fail to achieve one or more of our four goals stated above. We summarize our findings in work supporting offline/online fault tolerance in Table 1.

## 3 Background

Our goals in designing this protocol are to 1) enable computations at aggregate levels while still protecting any individual level data, 2) maximizing user's trust in the protocol by requiring that any servers used to facilitate the aggregation not be trusted by the users. Any system seeking to support such goals should provide a formal privacy analysis to demonstrate that the mechanism achieves the above privacy goals. For our adversary model, we assume that the users and the aggregator are untrusted and may behave in a fully malicious manner. Also, the aggregator and/or any subset of users may collude with eachother to attempt to infer information about other users' private data. Note that malicious participants can lie about their values to pollute the final output. Although these attacks are outside the scope of the paper, it should be said that one possible

**Table 1: Table of Related Work; $n$: the total number of participants, $d$: a parameter smaller than 100 in most practical settings in the scheme, $k$: a security parameter, $B$: the size of the future ciphertext buffer, $w$: the number of participant failures, $m$: the correct message, $\hat{m}$: the noisy message**

| Scheme | Dynamic join/leave | Fault tolerance | Secure against residual function attack | Trusted entity not needed for recovery | Noninteractive recovery | Comm. cost for join and leave | Comm. cost per period | Aggregation error |
|---|---|---|---|---|---|---|---|---|
| RN [20] | No | - | Yes | - | No | – | $O(1)$ | $O(1)$ |
| SCRCS [21] | No | No | Yes | - | - | $O(n), O(n)$ | $O(1)$ | $O(1)$ |
| AC [13] | Yes | Partially | No | No | No | $O(n), O(1)$ | $O(1)$ | $O(1)$ |
| Binary [9] | Yes | No | No | No | Yes | $O(n), O(n)$ | $O(\log n)$ | $\tilde{O}\left((\log n)^{1.5} \cdot \sqrt{w+1}\right)$ |
| LC [34] | Yes | No | Yes | - | - | $O(d), O(d)$ | $O(1)$ | $O(1)$ |
| WMYR [24] | Yes | No | No | No | Yes | $O(kB), O(kB)$ | $O(1)$ | $O(\sqrt{w+1})$ |
| CMZ [12] | Yes | Yes | No | No | Yes | $O(B), O(B)$ | $O(1)$ | $O(\sqrt{w+1})$ |
| PDAFT [10] | Yes | Yes | No | No | No | $O(kB), O(kB)$ | $O((2n)d)$ | $O(1)$ |
| BL [5] | No | Yes | No | No | No | $O(n), O(n)$ | $O(kd)$ | $O(\sqrt{w+1})$ |
| Ni et al. [19] | No | Yes | No | No | Yes | $O(n), O(n)$ | $O(kd)$ | $O(1)$ |
| Han et al. [14] | Yes | Yes | No | No | No | $O(n), O(n)$ | $O(2k(p+1))$ | $O(1)$ |
| BL [3] | Yes | Yes | No | No | No | $O(kB), O(k)$ | $O(kd)$ | $O(\sqrt{\frac{1}{d}\sum_{t=1}^{d}(\hat{m}-m)^2})$ |
| DDPFT [2] | No | Yes | No | No | No | $O(n), O(n)$ | $O((2n)d)$ | $O(1)$ |
| Ours | Yes | Yes | Yes | Yes | Yes | $O(d), O(1)$ | $O(1)$ | $O(1)$ |

defense is for each user to use a non-interactive zero-knowledge proof to prove the encrypted data is within a valid range.

## 3.1 Applications

There are many useful applications of our work. For private A/B testing, to support better privacy guarantees for users, our system is designed so that when the reporting origin wants to learn aggregate, cross-site information, it receives encrypted reports from multiple browsers. After receiving the encrypted reports, the reporting origin can use the encrypted reports to learn the private aggregate results for the data. Consumers of this service's output might include ad-tech, analytics, and other companies with similar use cases. Our method could be used by new web platform APIs that wish to expose some amount of cross-site data in a privacy-safe way without using persistent tracking IDs like third party cookies.

In the context of crowd sensing, many common real-life applications in the big data industry can be posed as a problem of the third-party aggregation of time-series data, i.e. stream aggregation. Specifically, applications based on statistical analysis, especially in marketing and healthcare analysis, often involve the collection of user-generated data periodically for calculating aggregate functions such as average, standard deviation, sum, etc.. Protecting data privacy during aggregation is of paramount importance.

Within Deep Learning (DL) our work can be extended to calculate the average function, which has become a core component of federated learning, a distributed paradigm of DL, where a series of parameters need to be aggregated. Due to the data security and individual privacy concerns involved in the collection of consumer datasets, it is desirable to apply privacy-preserving techniques to allow third-party aggregators to only learn aggregation outcomes.

## 3.2 Private Aggregation

The field of Private Stream Aggregation 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).

## 3.3 Fault Tolerance

Offline 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 successfully sent. There are primarily two existing paradigms to achieve this property. In the first, the aggregator communicates with a trusted entity (note that we are not referring to schemes that rely on one-time access to a trusted key dealer during setup, but refer to those that require communicating with a trusted entity each round to recover from a fault) to notify them of the fault, and the trusted entity provides the inputs to the aggregator to allow for the successful completion of the protocol. Since the trusted entity 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. In the second paradigm, users buffer their inputs that they send to the aggregator. Essentially, in this method users send a set of ciphertexts corresponding to several timestamps to the aggregator (generally the ciphertexts correspond to encryptions of zero). In this way, 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 method 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.). 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 "reset" [25, 26], 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. Note, we can trivially apply their techniques if necessary to our own work.

## 4 New Notion of Security

Current aggregation schemes generally follow the $\pi$ framework [21], shown in Figure 1. 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.*

While this definition is useful in schemes that do not consider online fault tolerance, from a practical standpoint, it becomes less useful once faults are introduced to a system. By definition, to recover from a fault, an aggregator must have some mechanism to generate synthetic input from any user to complete the calculation. Clearly in the event a user fails to respond, if the aggregator wants to calculate a partial sum, they must have some way to simulate receiving the user's response. This is because secure aggregation schemes must encode data in a way such that no partial information can be gained unless every participant's key is used to decrypt the partial aggregation. However, this directly violates the first criterion of aggregator obliviousness, since to satisfy online fault tolerance, an aggregator must then utilize this mechanism to recover the information needed to compute the partial aggregate.

It is important to note that schemes that incorporate differential privacy into their construction are only minimally impacted by this attack. Essentially, since the informal goal of differential privacy is to guarantee that an adversary viewing the output of a function cannot tell if a particular individual's information was used in the computation, if this attack is executed, the most an attacker can learn is the noisy plaintext value the user submitted that has already been masked with differentially private noise. Although the amount of privacy afforded to the user is heavily dependent on the sampling techniques used and the privacy budget allocated [11], we are more interested in the case where users do not apply differentially private noise to their inputs. Many applications cannot afford to allow users to add such noise, as the significant loss in data accuracy prevents the subsequent data analysis from having any utility to analysts. We are primarily interested in investigating how to design a system where users can still achieve a high level of security guarantees with their data, even if no differentially private noise is added. In such a scenario, existing schemes generally lose their security guarantees once the differentially private noise is removed from the scheme. This is because, after an adversary completes the residual function attack, they can learn the users' plaintext input in the clear.

Existing works try to avoid this issue by introducing a trusted entity that can assist the untrusted aggregator with completing the protocol. This is facilitated by allowing the aggregator to request the trusted entity provide the keys or encrypted ciphertexts the user was supposed to send (of 0) 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 entity (arguably, if such an entity exists it may be easier for users to send their plaintexts directly to this entity to speed up processing). More specifically, we are interested in supporting privacy in a scenario where there are no trusted entities (except during the one time setup phase with a trusted key dealer). In this setting, the two existing methods of achieving offline fault tolerance are ineffective, and are vulnerable to the residual function attack. In this attack, an aggregator can compute the same function over different inputs, and compute the difference between the final outputs, to infer individual values inputted by different users.

For instance, consider the first family of fault tolerant protocols, which allow the aggregator to ask a trusted entity to provide the information needed to recover the output. If such an entity is not trusted, the aggregator can request all of the private information from this entity and recover every party's individual input via the residual function attack. We also note that even if this entity 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. Again, if there is no trusted entity, 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 entity 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 formalized definition of aggregator obliviousness within the context of fault tolerant systems, that accounts for such scenarios. By extending the existing definitions [21, 32], we formally 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*

Protocol $\pi$

**Setup**$(1^\lambda)$ : Takes in a security parameter $\lambda$, 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 an additvely homomorphic cryptosystem, denoted $ENC(x_i)$ or $c_i$.
**AggrDec**$(param, sk_0, t, c_1, c_2, \ldots, c_n)$ : Takes in the public parameters $param$, a key $sk_0$, and ciphertexts $c_1, c_2, \ldots, c_n$ for the same time period $t$. For each $i \in N$ let $c_i = Enc\,(param, sk_i, t, x_i)$. Let $\mathbf{x} := (x_1, \ldots, x_n)$. The decryption algorithm outputs $f(\mathbf{x})$.

**Figure 1: Existing Framework**

*secure aggregation scheme $\beta$, with public parameters params, during timestep $t$, whose inputs and secret keys are denoted $x_i$ and $sk_i$ respectively, achieve aggregator obliviousness with online 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 types of queries:

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

•**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 $i^{\text{th}}$ participant, to the adversary.

•**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, where $J = Y$ for the duration of the game. 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 $a$ is 0 or 1. We say that the adversary wins the game if they correctly guess $a$ 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 $Q = \overline{K \cup M} := N \backslash (K \cup M)$, $J = Y$, 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 a user will only encrypt once may limit the utility of the previous definition.

## 5 New Constructions

### 5.1 Our Framework

To achieve the definition of privacy we put forth above, we design a new secure aggregation framework $\beta$ in Figure 2 that supports online fault tolerance. Note that this framework supports the same general functionality as the previous framework, but allows for 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 Achieving Aggregator Oblivious Security with Online Fault Tolerance

The greatest challenge we face when designing this mechanism is how to guarantee that the aggregator cannot collude with or dupe the entity it must interact with to acquire the synthetic data it needs to allow for fault recovery. Since both of these parties are untrusted in our adversary model, and must have tightly controlled interaction, a natural choice to support this functionality is leveraging trusted hardware, such as a Trusted Execution Environment (TEE). Specifically, we utilize the Intel SGX, perhaps the most widely deployed form of modern trusted hardware, which is rapidly becoming a major part of the digital security and privacy ecosystem. However, any TEE such as ARM TrustZone, AMD PSP, etc. can be used to implement our protocol. We note that there has been some controversy over the use of Intel SGX, and some notable side-channel attacks against it have been published [35]. Defending against these attacks is beyond the scope of this work, but Intel has made numerous statements regarding its commitment to patching such unintended weaknesses, making SGX a credible choice for deploying trusted hardware based protocols in the real world [35]. We also leverage elliptic curve based cryptography, in order to minimize the total number of bytes sent by each user to the aggregator to improve efficiency. We discuss the necessary background below:

Protocol $\beta$

**Setup**$(1^\lambda)$ : This is exactly the same as above.
**Enc**$(param, sk_i, t, x_i)$ : This is exactly the same as above.
**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 $f(\mathbf{x})$.

**Figure 2: Our Framework**

*5.2.1 Trusted Hardware* Note that our framework can work with any form of 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.

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. Only the code within the enclave can access its data, and external accesses are always denied. When it returns, enclave data stays in the protected memory. In some operating systems, enclaves must be less than 128 MB, which presents a constraint on the size of SGX dependent programs. The enclave is decrypted "on the fly" only within the CPU itself, and only for code and data running from within the enclave itself. This is enabled by an autonomous hardware unit called the Memory Encryption Engine (MEE) that protects the confidentiality and integrity of the CPU-DRAM traffic over a specified memory range.

Before performing computation on a remote platform, a user can verify the authenticity of the trusted environment. Via the attestation mechanism, a third party can establish that software is running on an Intel SGX enabled device and within an enclave. To preserve secrets after an enclave is destroyed, both as part of its normal operation and unplanned termination, Intel SGX provides data sealing and unsealing functions to protect data outside the boundary of an enclave. To handle this, sealing and unsealing functions retrieve unique keys based on the physical enclave platform. The enclave uses this key to encrypt data to the platform ("sealing"), or to decrypt data already on the platform ("unsealing").

*5.2.2 Elliptic Curves* Note that our framework can work with any form of additively homomorphic cryptography, but we chose elliptic curve cryptography for our concrete instantiation. Elliptic Curve Cryptography (ECC) provides the same level of security as RSA, Pallier, 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. More formally:

Definition 3. *The elliptic curve over $\mathbb{Z}_p, p > 3$, is the set of all pairs $(x, y) \in \mathbb{Z}_p$ which fulfill $y^2 \equiv x^3 + a \cdot x + b \bmod p$ together with an imaginary point of infinity $\mathbb{O}$, where $a, b \in \mathbb{Z}_p$ and the condition $4 \cdot a^3 + 27 \cdot b^2 \neq 0 \bmod p$ holds.*

There are two important operations associated with points on an elliptic curve: **(1) Point Addition** $A + B$, and **(2) Scalar Multiplication** $s \times A$, where $s$ is a scalar. For **(1)**, this is the case where we compute $C = A + B$ and $A \neq B$, for **(2)**, this is the case where we compute the addition group operation ($C = A + A + ... + A$) a total of $s$ times where $A = A$. We denote the group operation addition with the symbol +. Here, addition means that given two points and their coordinates, $A = (x_1, y_1)$ and $B = (x_2, y_2)$, we have to compute the coordinates of a third point $R$ such that: $A + B = C$ where $(x_1, y_1) + (x_2, y_2) = (x_3, y_3)$. To compute these, we can use a standard formula [30]. Note that it is known that:

Theorem 1. *The points on an elliptic curve together with $\mathbb{O}$ have cyclic subgroups. Under certain conditions all points on an elliptic curve form a cyclic group.*

To utilize elliptic curves in cryptography, we rely on the hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP):

Definition 4. *Given is an elliptic curve E. We consider an element P and another element T. The DL problem is finding the integer d, where* $1 \leq d \leq E$*, such that:*

$$\underbrace{P + P + \cdots + P}_{d \text{ times}} = dP = T$$

Intuitively, cryptosystems are based on the idea that $d$ is large and kept secret and attackers cannot compute it easily. Under the same condition, the Decisional Diffie-Helman assumption states that the ECDLP problem in $Z_N^*$ is believed to be intractable if $d$ is unknown. That is, for any probabilistic polynomial time adversary *PPTA*, denoted $\mathcal{A}$, the advantage of the adversary in solving the ECDLP problem, denoted $adv_{\mathcal{A}}^{ECDLP}$, is a negligible function of the security pararmeter. In this paper, we rely on the above assumption to define and prove the security. The original Elliptic Curve (EC) encryption scheme, is not additively homomorphic. However, the elliptic curve group is an additive group, which can be used to build an additive homomorphic scheme (EC-ElGamal). Algorithms 1 and 2 show the methods for EC encryption and decryption, respectively, over the finite field $GF(p)$ [22]. Note a finite field $GF(p)$ is a set of finite numbers on which the operations of multiplication, addition, subtraction and division are defined and satisfy certain basic rules such as associativity, commutativity, the existence of additive and multiplicative identities and inverses, etc.. The order of the curve $E$ is denoted $ord(E)$ and $g$ is the generator point of the curve. The secret key is defined as integer number $x \in GF(p)$, while the public key is determined as $Y = xG$. The function $map()$ [22] is a deterministic mapping function used to map values $m_i \in GF(p)$ to curve points $M_i \in E$ such that the following holds: $\text{map}(m_1 + m_2 + \ldots) = \underbrace{\text{map}(m_1)}_{M_1} + \underbrace{\text{map}(m_2)}_{\ldots} + \ldots$.

Note because addition over an elliptic curve is only possible with points on that curve, integers must be mapped to corresponding points. To accomplish this, each integer $m$ is mapped to a curve point $M$ where $M = mG$. The reverse mapping function rmap() extracts $m$ from a given point $mG$ [22]. The mapping function $\text{map} : m \rightarrow mG$ with $m \in GF(p)$ supports additive homomorphism since $M_1 + M_2 + \ldots = \text{map}(m_1 + m_2 + \ldots) = (m_1 + m_2 + \ldots) \times G = m_1 \times G + m_2 \times G + \ldots$ holds, where $m_1, m_2, \cdots \in GF(p)$. Specifically, we leverage the BGN variant devised by Boneh et al. [7] which was the first to allow both additions and multiplications with a constant-size ciphertext. Note that while the additive property is the same as for the ElGamal variant described above, one multiplication is permitted (i.e. the system is "somewhat homomorphic").

**Algorithm 1** EC-ElGamal Encryption

**Require**: public key $Y$ plaintext $m$
1: choose random k ∈ [1,ord(E) - 1]
2: M := map(m)
3: R := kG
4: S := M + kY
5: **Return:** ciphertext $(R, S)$

**Algorithm 2** EC-ElGamal Decryption

**Require**: secret key x, ciphertext (R, S)
1: M := -xR + S
2: m := rmap(M)
3: **Return:** m

## 5.3 Basic Protocol

To formally define the correctness and the security of our framework, we first present a precise definition of Cryptonite. We first present our basic approach, and we later overcome performance limitations in our optimized version, which is presented in the following section. Note that when the context is clear, we sometimes use standard addition and multiplication operators, as done in previous PSA papers [9, 21], instead of referencing the BGN scheme, 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 $H : \mathbb{Z} \rightarrow \mathbb{G}$ denote a hash function modeled as a random oracle. We assume the aggregator is equipped with an Intel SGX.

**Setup**$(1^\lambda)$ : 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 runs BGN.Gen() 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 Shamir 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 $c \leftarrow ENC(x_i) + ENC(H(t) \cdot sk_i)$. 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 $c \leftarrow ENC(0) + ENC(H(t) \cdot sk_j)$ 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 $N = U \cup J$) $V \leftarrow ENC(H(t) \cdot sk_0) + \sum_{i=1}^{n} c_i$. To decrypt the sum, we can leverage Pollard's lambda method 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 [9]. Note that this construction is secure under Definition 2, and we can prove this via a security game, using proof techniques from existing work [21]:

Theorem 2. *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 online 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 $p$.

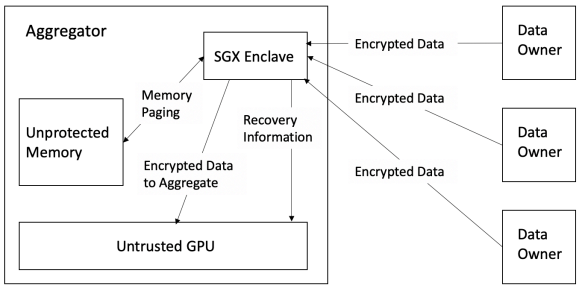



**Figure 3: System Diagram**

**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 $\alpha_i$. The challenger returns $\alpha_i$ 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 $a$. If $a = 0$, the challenger returns to the adversary $\{h^{\alpha_q} \mid q \in Q \backslash Y\}, \{h^{\alpha_y} \mid y \in Y\}$. If $a = 1$, 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 $a = 0$ or $a = 1$. The adversary wins the game if they have not asked for any $\alpha_q$ for $q \in Q$, $Y = J$, and if they successfully guess $a$.

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

Proof. 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: $\prod_{1 \le q \le d} R_q = \prod_{1 \le q \le d} h^{\beta_q}$. 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 $d^{\text{th}}$ 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 $\alpha_e$ and $\alpha_b$, however, it can compute the values of $g^{\alpha_b} = g^x$ and $g^{\alpha_e} = \left(\prod_{q \neq e} g^{\alpha_q}\right)^{-1} = (g^x)^{-1}$. $\prod_{q \neq e, q \neq b} g^{\alpha_q} \cdot \mathcal{B}$ 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 $\alpha_e$ and $\alpha_b$, $\mathcal{B}$ returns the corresponding $\alpha_q$ value to $\mathcal{A}$, but if $\mathcal{A}$ asks for $\alpha_e$ or $\alpha_b$, 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 $d^{\text{th}}$ 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 $C$ the tuple $g, h, g^{\alpha_0}, g^{\alpha_1}, \ldots, g^{\alpha_n}$. $\mathcal{B}$ sets $\alpha_0$ 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 $b^{\text{th}}$ 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, $\mathcal{B}$ simply 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 $b^{\text{th}}$ 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 $b^{\text{th}}$ 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 $g^{\alpha_q}$, $\mathcal{B}$ 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 $C$, and forwards the answer $\alpha_q$ to $\mathcal{A}$.

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

**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 $a' = 1$. Otherwise, $\mathcal{B}$ guesses that $a' = 0$

If the challenger $C$ 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. □

*5.3.1 Dynamic User Groups* Similar to other protocols, by supporting fault tolerance, we inherit the ability to easily add and remove users from the scheme, to support a dynamic user group. If a user drops out of the protocol, we can treat them as permanently faulted for every time period. If we want to add a user, since we already have all of the keys stored in the Intel SGX, we can compute a new polynomial of higher degree that fits the curve by leveraging a polynomial interpolation algorithm. With Newton's method, we can precompute most of the computation recursively so that we can later add users in $O(logN)$ time where $N$ is the number of users.

## 5.4 A More Efficient Protocol

Although the above protocol achieves security according to definition 2, it would be even better if we could further improve performance by outsourcing some computation to the untrusted aggregator. It is well known that most TEEs, such as Intel SGX enclaves, cannot access many OS features (e.g., sophisticated multi-threading, disk and driver IO), and 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]. Our goal is to find a secure method to outsource expensive computations instead of running the entire process in the enclave, without compromising security. We can accomplish this by following the same set up procedure as before, but instead having users send two messages simultaneously. They can send one short message to the Intel SGX to indicate they are participating in the protocol, and also send their ciphertext to the untrusted aggregator. Intuitively, the aggregator can simultaneously begin the partial summation of the ciphertexts of the users that did not fault, while the SGX iterates over the table of all users to determine which users faulted and computes their synthetic ciphertexts which are sent to the aggregator. In this way, the somewhat expensive aggregation step can be done on more powerful, albeit untrusted hardware. 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 directly to the SGX first. Then the SGX can output the users' ciphertexts who did not fault to the aggregator, along with the synthetic data used to overcome existing faults, which can be more efficiently computed outside the enclave. We present a more formal description of this protocol below:

Let $\mathbb{G}$ denote a cyclic group of prime order $p$ for which the Decisional Diffie-Hellman problem is hard. Let $H : \mathbb{Z} \rightarrow \mathbb{G}$ denote a hash function modeled as a random oracle. We assume the aggregator is equipped with an Intel SGX.

**Setup**$(1^\lambda)$: Each user first performs remote attestation with the aggregator's Intel SGX, to verify it will faithfully execute the protocol (this is a one time process). The Intel SGX runs BGN.Gen() 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 data aggregator obtains the key $sk_0 := s_0$ and participant $i$ obtains the secret key $sk_i := s_i$.

**Enc**$(param, sk_i, t, x)$: For participant $i$ to encrypt a value $x_i \in \mathbb{Z}_p$ for time step $t$, they compute the following ciphertext $c \leftarrow ENC(x_i) + ENC(H(t) \cdot sk_i)$. Note that the user sends its ciphertext to the untrusted aggregator 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 unique id 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. Then, since the Intel SGX has each user's secret key, it can compute $c \leftarrow ENC(0) + ENC(H(t) \cdot sk_j)$ for all users $j \in J$. Following this, all ciphertexts (i.e. those associated with $j \in J$ and $u \in U$) are output to the untrusted aggregator.

**AggrDec** ( $param, sk_0, t, c_j, c_u$): The untrusted aggregator computes the following (i.e. it combines the ciphertexts from the users with those from the SGX) $V \leftarrow H(t)^{\text{sk}_0} \sum_{i=1}^{n} c_i$.

To decrypt the sum, we can leverage Pollard's lambda method as was done before. The proof of security for this protocol is very similar to the earlier proof, and we omit it for the sake of space.

It should be said that although it may seem more efficient to simply send plaintext data to an SGX enclave to be aggregated, it is known that Intel SGX has difficulties exploiting multi-threading [37] due to the lack of common synchronization primitive support often found on traditional operating systems. Also, leveraging threading within the Intel SGX can introduce security vulnerabilities [38]. Overall performance can be improved if the aggregation step is outsourced to an untrusted space, which can leverage more robust forms of parallel computing, especially on higher performance hardware, such as GPUs, etc..

## 5.5 Differential Privacy

Achieving differential privacy is not the primary focus of this paper, but we can easily adapt the methods of existing works to achieve

this goal [9, 21]. We summarize these techniques here for completeness. The aggregator and the users may generate randomness for the protocol, and users can add this noise to their input. Therefore, we can define a distribution on the transcripts. Formally, we use the notation $\Lambda$ to denote a randomized protocol. In addition to the users' data $\mathbf{x}$, the protocol $\Lambda$ takes a security parameter $\lambda \in \mathbb{N}$. We use the notation $\Lambda(\lambda, \mathbf{x})$ to denote the distribution of the transcript when the security parameter is $\lambda$ and the input configuration is $\mathbf{x}$. Similar to existing work, we consider computational differential privacy (CDP), as it supports security guarantees against computationally-bounded adversaries. We now define CDP:

Definition 5. *(Computational Differential Privacy): Suppose users are compromised by a randomized process $C$, and we use $\Phi$ to denote the information obtained by the adversary from compromised users. Let $\epsilon, \delta > 0$. A randomized protocol $\Lambda$ preserves computational $(\epsilon, \delta)$-differential privacy if there exists a negligible function $\eta : \mathbb{N} \rightarrow \mathbb{R}^+$ such that for all $\lambda \in \mathbb{N}$, for all $i \in N$, for all vectors $\mathbf{x}$ and $\mathbf{y}$ in $\{0,1\}^n$ that differ only at position $i$, for all probabilistic polynomial-time Turing machines $\mathcal{A}$, for any output $b \in \{0,1\}$ :*

$\Pr_{C_i}[\mathcal{A}(\Lambda(\lambda, \mathbf{x}), \Phi) = b] \leq e^{\epsilon} \cdot \Pr_{C_i}[\mathcal{A}(\Lambda(\lambda, \mathbf{y}), \Phi) = b] + \delta + \eta(\lambda)$

*where the probability is taken over the randomness of $\mathcal{A}$, $\Lambda$, and $C_i$, which denotes the underlying compromise process conditioning on the event that user $i$ is uncompromised. A protocol $\Lambda$ preserves computational $\epsilon$-differential privacy if it preserves computational $(\epsilon, 0)$-differential privacy.*

The geometric distribution is commonly used to perturb the data and ensure differential privacy, because it allows us to work in the integer domain. The geometric distribution is useful when used in combination with cryptography, since most schemes operate over discrete mathematical structures, and are not designed to work with real numbers. We now define the symmetric geometric distribution.

Definition 6. *(Geometric Distribution) Let $\alpha > 1$. We denote by $Geom(\alpha)$ the symmetric geometric distribution that takes integer values such that the probability mass function at $k$ is $\frac{\alpha-1}{\alpha+1} \cdot \alpha^{-|k|}$.*

It is known that if we let $\epsilon > 0$, and denote $u$ and $v$ as two integers such that $|u - v| \leq \Delta$, and if we let $r$ be a random variable having distribution $\text{Geom}\left(\exp\left(\frac{\epsilon}{\Delta}\right)\right)$, then, for any integer $k$, $\Pr[u + r = k] \leq \exp(\epsilon) \cdot \Pr[v + r = k]$. In our setting, it we can simply consider $\text{Geom}(\alpha)$ with $\alpha = e^{\epsilon}$. Note that $\text{Geom}(\alpha)$ has variance $\frac{2\alpha}{(\alpha-1)^2}$, and since $\frac{\sqrt{\alpha}}{\alpha-1} \leq \frac{1}{\ln \alpha} = \frac{1}{\epsilon}$, the magnitude of the error added is $O\left(\frac{1}{\epsilon}\right)$. The following diluted geometric distribution is often used in the context of private aggregation [9, 21], because it helps minimize the distortion in accuracy caused by introducing DP noise:

Definition 7. *(Diluted Geometric Distribution): Let $0 < \Psi \leq 1, \alpha > 1$. A random variable has $\Psi$-diluted Geometric distribution $Geom^{\Psi}(\alpha)$ if with probability $\Psi$ it is sampled from $Geom(\alpha)$, and with probability $1 - \Psi$ is set to 0.*

We describe a simple scheme to achieve differential privacy, although optimizations exist [9, 17] that we can trivially adapt if tighter bounds on accuracy or other properties are required for an application. We have each user generate an independent noise from $\text{Geom}(e^{\epsilon})$, which is added to their input. Each user sends this noisy input to the aggregator, who computes the sum. As each user adds one copy of independent noise to their data, $N$ copies of noises accumulate in the sum. As some positive and negative noises may cancel out, the accumulated noise is $O(\frac{\sqrt{n}}{\epsilon})$ with high probability.

| Iteration | CPU Point Addition | SGX Point Addition | CPU Hash | SGX Hash | CPU Exponent | SGX Exponent |
|---|---|---|---|---|---|---|
| 1 | 0.000004 | 0.000191 | 0.000001 | 0.000144 | 0.000008 | 0.000176 |
| 10 | 0.000009 | 0.000609 | 0.000128 | 0.000162 | 0.000010 | 0.000183 |
| $10^2$ | 0.000035 | 0.001071 | 0.000151 | 0.000198 | 0.000025 | 0.000179 |
| $10^3$ | 0.000298 | 0.005947 | 0.000269 | 0.000183 | 0.000093 | 0.000234 |
| $10^4$ | 0.002923 | 0.054511 | 0.001247 | 0.002560 | 0.000850 | 0.000591 |
| $10^5$ | 0.007896 | 0.531627 | 0.005765 | 0.002960 | 0.004395 | 0.004114 |
| $10^6$ | 0.053851 | 5.265254 | 0.023503 | 0.068600 | 0.017307 | 0.038647 |
| $10^7$ | 0.507041 | 52.74683 | 0.202715 | 0.364000 | 0.142546 | 0.385640 |

**Table 2: Microbenchmarks Time in Seconds**

## 6 Experiments

To better understand the practical performance of our protocol we ran experiments using C that simulated having several million users run our protocol. For these tests, we used a workstation running Ubuntu 16.04 LTS equipped with a Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz. We used time series data from the 3W dataset from the UCI machine learning data repository [23], and report the average time for 50 trials for each experiment. Achieving accurate timings for operations within the enclave is difficult, because the SGX primitive `sgx_get_trusted_time` only supports second level precision, but many operations can be computed at millisecond precision. To measure microbenchmarks, we used the Unix `clock` primitive, which supports millisecond precision, and timed the overall time to compute ECALLs within the untrusted component of the program. We compare these times to computing the same operations on standard CPUs outside of the enclave in Table 2.

It is important to note that this comparison is somewhat unfair, as the times associated with the SGX also contain the time required to marshal data into the enclave, in addition to the time needed to perform the actual operation (due to the limitation of `sgx_get_trusted_time`). This explains why it appears at first that some SGX operations are at least two orders of magnitude slower than their CPU counterparts, but the performance difference is amortized over time as the total number of operations is increased. However, it should be noted that although the time to perform most simple operations in the SGX can amortized, when computing a large number of operations in a pipeline, to take roughly the same amount of time, there will always be overhead associated with computing in an enclave, due to the additional operations required to enforce strong isolation. Note, due to the cryptographic operations, all the computations in our scheme are in closed integer groups, and thus we need to use integers to represent real numbers. We can exploit the homomorphism to represent real numbers via integers using fixed point representation as done in earlier work [15].

### 6.1 Basic Scheme

The results for our basic scheme, assuming no users fault, are shown in Figures 4a,b. 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 4c. 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 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 [10, 14], while still supporting strong privacy guarantees, to improve communication complexity.

### 6.2 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 Figures 4d,e. 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 4f. 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.

It is well known that most PSA schemes are not implemented, so it is somewhat difficult to compare our practical performance to existing work. We do note that because we rely on elliptic curves for the underlying encryption, similar to [9, 21], our performance should roughly be about the same (on modern hardware with elliptic curves such as 25519 encryption time is approximately 0.6ms and decryption time is roughly 1.5s [21]). However, unlike their paper we are able to support a stronger level of security and recover from faults without either buffering ciphertexts, which causes increased communication overhead, or requiring additional rounds of communication overhead. It is worth noting that there is some existing work [6] that leverages lattice based encryption, which is known to be faster than elliptic curve based encryption. However, the authors note that while certain operations, such as decryption are roughly 150 times faster than similar elliptic curve based operations (note this comparison is against brute force decryption and not the optimized Pollard's rho method), their overall runtime is slower by a factor of roughly 6. There is another paper that claims faster encryption and decryption time [34], and this makes sense because the underlying cryptographic primitive they rely on is HMAC, which is known to be faster than elliptic curves. While they do briefly note possible fault recovery techniques, unlike their scheme, we can recover from faults without either buffering ciphertexts, which causes increased communication overhead, or requiring additional rounds of communication overhead, while supporting a stronger level of security overall. We emphasize that although our concrete instantiation may be slower than these works, it is simply a quickly-implemented instantiation to demonstrate the practical usefulness of our framework. Our instantiation can be easily replaced with faster cryptographic primitives (i.e. HMAC) to gain a similar performance improvement, and for future work, we are interested in adapting HMAC based techniques to improve our overall runtime.

## 7 Conclusion

We defined a new level of security for Private Stream Aggregation in the presence of faults and malicious adversaries. After describing a general framework that allows existing work to reach this level of security, we developed the first protocol that provably reaches this level of security. Furthermore, we developed a methodology to outsource computationally intensive work to high performance hardware while still allowing for a strong level of security. After implementing our protocol, we experimentally demonstrated out work reaches high levels of scalability and communication efficiency over existing work while supporting a higher level of security. We hope this work encourages other researchers to investigate combining trusted hardware and cryptography to build more secure systems.

## References


[1] HS Annapurna and M Siddappa. 2015. Secure data aggregation with fault tolerance for Wireless Sensor Networks. , 29–33 pages.

[2] Haiyong Bao and Rongxing Lu. 2015. Ddpft: Secure data aggregation scheme with differential privacy and fault tolerance. , 7240–7245 pages.

[3] Haiyong Bao and Rongxing Lu. 2015. A new differentially private data aggregation with fault tolerance for smart grid communications. *IEEE Internet of Things Journal* 2, 3 (2015), 248–258.

[4] Haiyong Bao and Rongxing Lu. 2015. "Privacy-Enhanced Data Aggregation Scheme Against Internal Attackers in Smart Grid". *IEEE Transactions on Industrial Informatics* 12 (01 2015), 1–1. https://doi.org/10.1109/TII.2015.2500882

[5] Haiyong Bao and Rongxing Lu. 2017. A lightweight data aggregation scheme achieving privacy preservation and data integrity with differential privacy and fault tolerance. , 106–121 pages.

[6] Daniela Becker, Jorge Guajardo, and Karl-Heinz Zimmermann. 2018. Revisiting Private Stream Aggregation: Lattice-Based PSA.

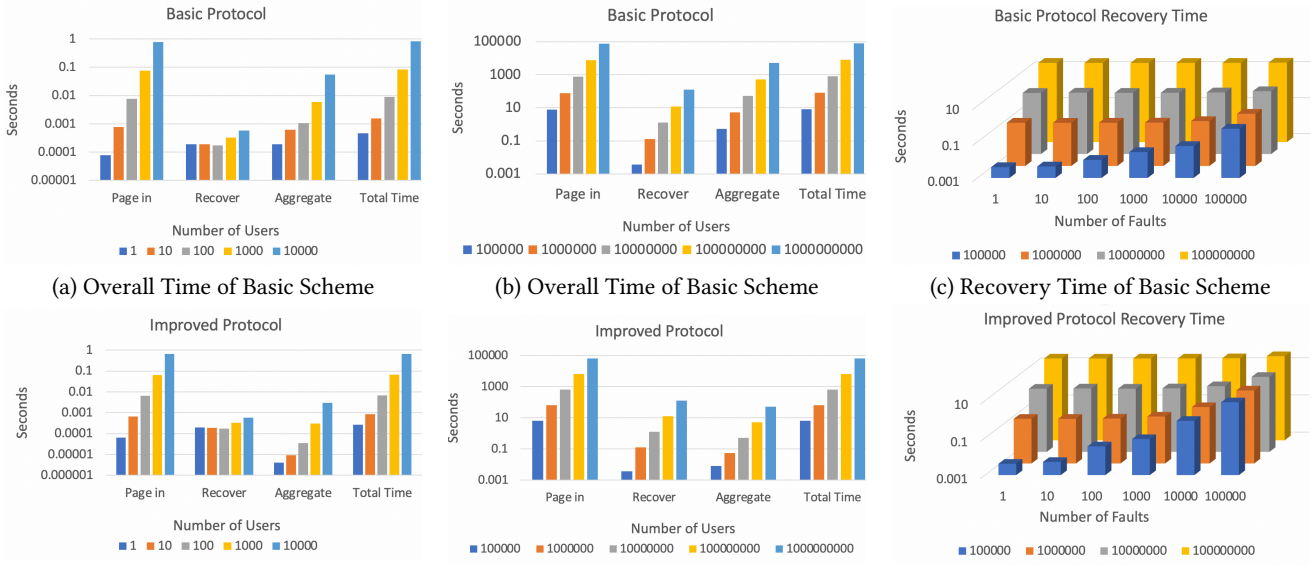
[7] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. 2005. Evaluating 2-DNF formulas on ciphertexts. , 325–341 pages.

[8] Claude Castelluccia, Einar Mykletun, and Gene Tsudik. 2005. Efficient aggregation of encrypted data in wireless sensor networks. In *International conference on mobile and ubiquitous systems: networking and services*. IEEE, 109–117.

[9] T-H Hubert Chan, Elaine Shi, and Dawn Song. 2012. Privacy-preserving stream aggregation with fault tolerance. , 200–214 pages.

[10] Le Chen, Rongxing Lu, and Zhenfu Cao. 2015. PDAFT: A privacy-preserving data aggregation scheme with fault tolerance for smart grid communications. *Peer-to-Peer networking and applications* 8, 6 (2015), 1122–1132.

[11] Cynthia Dwork. 2008. Differential privacy: A survey of results.




(a) Overall Time of Basic Scheme (b) Overall Time of Basic Scheme (c) Recovery Time of Basic Scheme

(d) Overall Time of Improved Scheme (e) Overall Time of Improved Scheme (f) Recovery Time of Improved Scheme

**Figure 4: Experimental Results**


[12] Chen et al. 2017. Private data aggregation with integrity assurance and fault tolerance for mobile crowd-sensing. *Wireless Networks* 23, 1 (2017), 131–144.

[13] Gergely et al. 2011. I have a dream!(differentially private smart metering). , 118–132 pages.

[14] Han et al. 2015. PPM-HDA: privacy-preserving and multifunctional health data aggregation with fault tolerance. *IEEE Transactions on Information Forensics and Security* 11, 9 (2015), 1940–1955.

[15] Jung et al. 2016. PDA: semantically secure time-series data analytics with dynamic user groups. *IEEE TDSC* 15, 2 (2016), 260–274.

[16] Li et al. 2012. Efficient authentication scheme for data aggregation in smart grid with fault tolerance and fault diagnosis.

[17] Liu et al. 2017. Achieving privacy protection using distributed load scheduling: A randomized approach. *IEEE Transactions on Smart Grid* 8, 5 (2017), 2460–2473.

[18] Mofrad et al. 2018. A comparison of Intel SGX and AMD memory encryption technology.

[19] Ni et al. 2017. Differentially private smart metering with fault tolerance and range-based filtering. *IEEE Transactions on Smart Grid* 8, 5 (2017), 2483–2493.

[20] Rastogi et al. 2010. Differentially private aggregation of distributed time-series with transformation and encryption. , 735–746 pages.

[21] Shi et al. [n.d.]. Privacy-preserving aggregation of time-series data. Citeseer.

[22] Ugus et al. 2009. Optimized implementation of elliptic curve based additive homomorphic encryption for wireless sensor networks.

[23] Vargas et al. 2019. A realistic and public dataset with rare undesirable real events in oil wells. *Journal of Petroleum Science and Engineering* 181 (2019), 106223. https://doi.org/10.1016/j.petrol.2019.106223

[24] Won et al. 2014. Proactive fault-tolerant aggregation protocol for privacy-assured smart metering. , 2804–2812 pages.

[25] Wang et al. 2020. Fault Tolerant Multi-subset Aggregation for Smart Grid.

[26] Xue et al. 2018. PPSO: A privacy-preserving service outsourcing scheme for real-time pricing demand response in smart grid. *IEEE Internet of Things Journal* 6, 2 (2018), 2486–2496.

[27] Chun-I Fan, Shi-Yuan Huang, and Yih-Loong Lai. 2013. Privacy-enhanced data aggregation scheme against internal attackers in smart grid. *IEEE Transactions on Industrial informatics* 10, 1 (2013), 666–675.

[28] Donna Gillin. 2000. The federal trade commission and Internet privacy. *Marketing Research* 12, 3 (2000), 39.

[29] Joao Girao, Dirk Westhoff, and Markus Schneider. 2005. CDA: Concealed data aggregation for reverse multicast traffic in wireless sensor networks. In *IEEE International Conference on Communications, ICC 2005.*, Vol. 5. IEEE, 3044–3049.

[30] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. 2006. Guide to elliptic curve cryptography.

[31] Xingze He, Xinwen Zhang, and C-C Jay Kuo. 2013. A distortion-based approach to privacy-preserving metering in smart grids. *IEEE Access* 1 (2013), 67–78.

[32] Marc Joye and Benoît Libert. 2013. A scalable scheme for privacy-preserving aggregation of time-series data. In *ICFCDS*. Springer, Berlin, Germany, 111–125.

[33] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. 2009. Closeness: A new privacy measure for data publishing. *IEEE Transactions on Knowledge and Data Engineering* 22, 7 (2009), 943–956.

[34] Qinghua Li and Guohong Cao. 2013. Efficient privacy-preserving stream aggregation in mobile sensing with low aggregation error. , 60–81 pages.

[35] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. 2020. A Survey of Published Attacks on Intel SGX.

[36] Jing Shi, Rui Zhang, Yunzhong Liu, and Yanchao Zhang. 2010. Prisense: privacy-preserving data aggregation in people-centric urban sensing systems.

[37] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware.

[38] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. , 440–457 pages.