Projet Scientifique Artistique et Technique : Analyse d'impact d'un protocole de federated learning sécurisé Publication

BAILLY Grégoire
CHATELIN Corentin
GIRAUD Corentin
TOULIER-ANCIAN Lucas
VIRY Baptiste

27 janvier 2020

Table des matières

Résumé

Les personnes victimes d'un AVC sont considérées comme étant à risque dans la suite de leur vie quotidienne, et peuvent être sujettes à de nombreux problèmes graves arrivant parfois sans signes avant-coureurs. Il est donc important de permettre à ces personnes d'accéder à un suivi strict et constant de leur quotidien, afin d'obtenir une réaction rapide des services de santé en cas de détection de problèmes. Comment donc permettre un suivi "strict et constant" tout en se conformant aux contraintes qu'impliquent des données de santé ? Nos téléphones intelligents, tous équipés de capteurs gyroscopiques, ainsi que d'accéléromètres, offrent une solution intéressante. L'objet de cette étude est de parvenir à mettre en place un prototype d'application basée sur l'apprentissage fédéré sur mobile avec l'aide de modèles de reconnaissances d'activité humaine. Ainsi, chaque téléphone intelligent, simulé par un "client" entraînera/raffinera un modèle sur les données de l'utilisateur, une portion aléatoire du jeu de donnée complet.Seul le modèle entraîné sera envoyé au serveur central évitant ainsi l'envoi de données sensibles à un tiers. Le serveur agrégera ensuite les contributions de chacun des utilisateurs sans connaître la contribution de ces derniers dans la mise à jour du modèle. Cette agrégation particulière repose sur un protocole sécurisé et l'impact de sont utilisation sera étudiée.

1 État de l'art

1.1 Introduction

L'intelligence artificielle est l'un des domaines générant le plus de recherche en informatique aujourd'hui. Un moyen, assez utilisé aujourd'hui, de travailler dans ce domaine, est d'utiliser des réseaux de neurones, afin de faire de l'apprentissage statistique. Ceux-ci sont composés de plusieurs "couches". La première représente celle des intrants, les variables du système. On peut ensuite avoir plusieurs couches "intérieures" au réseau de neurones, dans laquelle chaque neurone est lié à chacun de ceux présents dans la couche précédente, ainsi que la suivante. Ces liaisons sont pondérées par des "poids" et des "biais", qui suffisent à eux-seuls à représenter le réseau de neurones dans un état à un instant donné. Pour entraîner un réseau de neurones, à l'origine on va utiliser un dataset, un jeu de données très volumineux, que l'on va diviser en deux. La première moitié sera labellisée pour dire au réseau de neurones directement quelle valeur il doit trouver, pour qu'il ait une base de prédiction, alors que la deuxième servira à le tester : on lui enverra chacun des éléments de la deuxième moitié, et, pour chacune de ses prédictions, l'écart des poids et de biais à la fonction de pertes sera corrigé par un mécanisme de rétro-propagation dans le réseau de neurones. Cela lui permettra d'affiner ses prédictions. Les poids et les biais seront les symboles de cet affinement des prédictions. Lorsqu'il est nécessaire de les exporter, s'agissant de valeur numériques, la manière la plus simple est de les transformer en un seul grand vecteur, facile à traiter. L'un des domaines d'applications possibles d'un réseau de neurones est la reconnaissances d'activité, via des données d'accélération et gyroscopiques de téléphone. Également, afin de pouvoir entraîner un réseau de neurones à l'avance avec des données existantes, on peut utiliser des datasets open source, comme par exemple MotionSense : qui inclut des données accélérométriques et gyroscopiques en fonction du temps à partir d'un iPhone 6S pour 24 personnes. [1] [2].

1.2 L'apprentissage fédéré

L'apprentissage fédéré (*federated learning* en anglais) consiste à entraîner un algorithme sur la machine des utilisateurs d'une application et à partager les apprentissages réalisés sur la machine de chaque utilisateur.

1.2.1 Processus traditionnel dans une architecture client/serveur

Dans une architecture client / serveur, la manière traditionnelle d'utiliser un modèle pour faire des prédictions est résumé dans le schéma ci-dessous et suit le processus suivant :

1. Le client fait une requête au serveur pour demander des prédictions sur des données qu'il joint à la requête.
2. Le serveur utilise son modèle local pour faire des prédictions sur les données du client qu'il stocke.
3. Le serveur renvoie les prédictions au client qui les affiche à l'utilisateur.
4. Ce dernier peut alors juger de la pertinence des résultats et corriger les prédictions si nécessaire. Il renvoie alors la correction des prédictions au serveur.
5. Le serveur stocke de nouveau la correction des prédictions qu'il utilisera par la suite pour entraîner de nouveau le modèle.

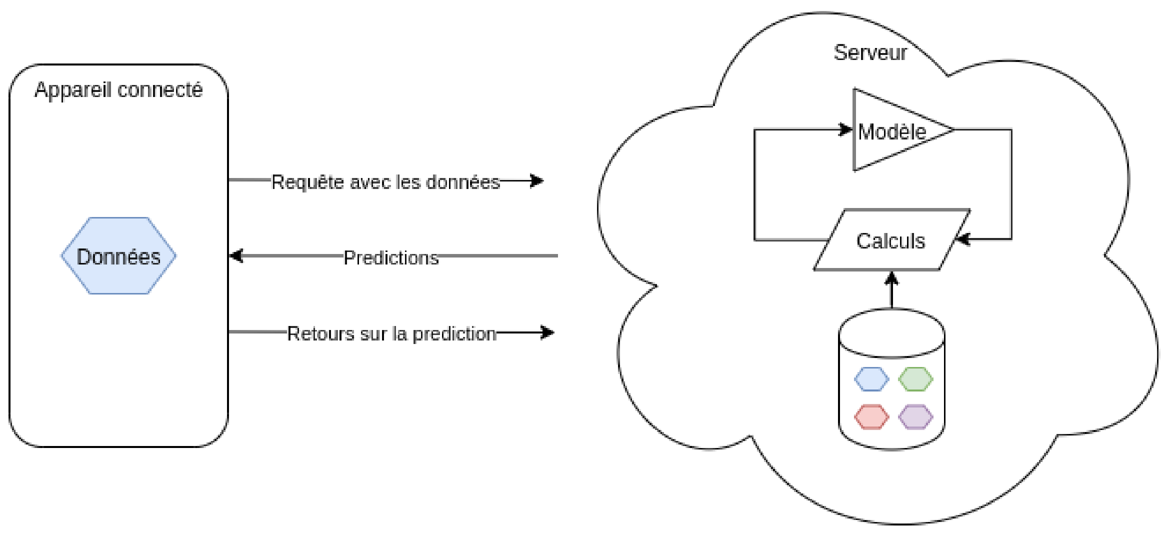


FIGURE 1 – Mode centralisé

Néanmoins, les inconvénients de ce processus sont nombreux. En effet, dans le contexte des applications mobiles, l'expérience utilisateur peut être détériorée par des latences réseau,ou pire, des ruptures de connexion. De plus, l'application ne peut proposer de mode hors ligne. Les données envoyées au serveur sont aussi limitées par la faible bande passante et rendent impossible la prédiction sur de gros jeux de données. Enfin, la protection des données sensibles est inexistante puisque le serveur en conserve une copie.

1.2.2 Processus d'apprentissage fédéré dans une architecture client/serveur

L'apprentissage fédéré est utilisé pour résoudre les inconvénients listés ci-dessus. C'est un domaine de recherche actif et récent. Il fonctionne suivant le processus détaillé ci-dessous :

1. Le serveur distribue un modèle initial aux clients.
2. Chaque client utilise ses propres données pour produire un nouveau modèle localement entraîné.
3. Ce modèle va être converti en une représentation structurelle (liste de poids et de biais) qui sera envoyée au serveur.
4. Ce dernier sera responsable de l'agrégation des modèles des différents clients.

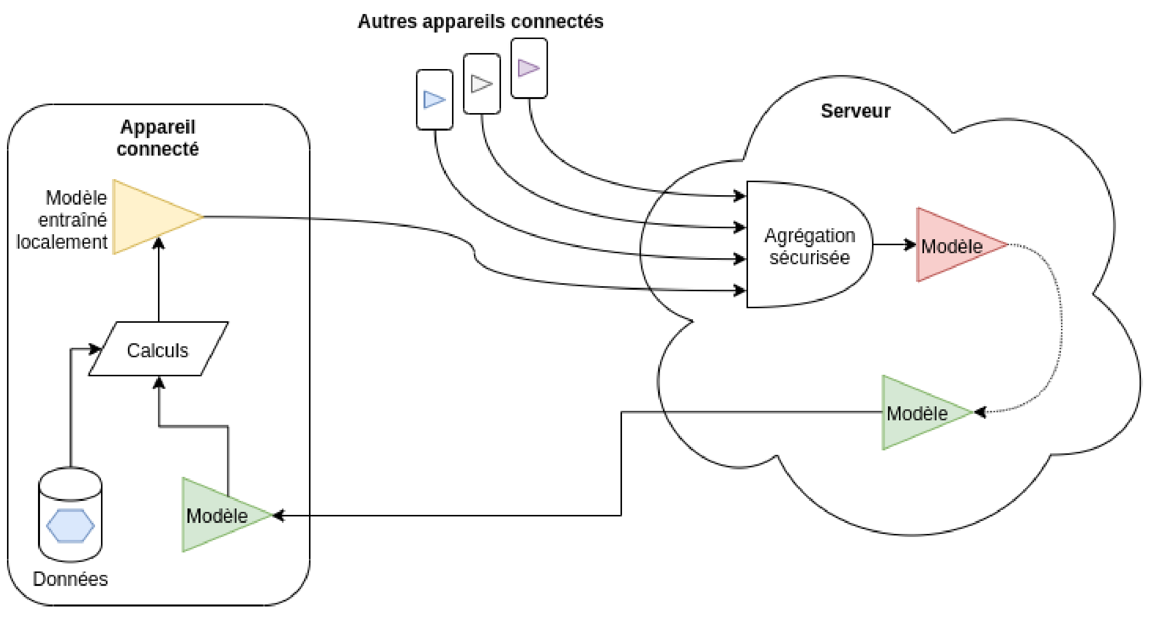


FIGURE 2 – Mode décentralisé

Il existe plusieurs manières d'agréger les modèles de client. L'approche standard est triviale. Considérons que la représentation structurelle du réseau de neurones pour un utilisateur u appartenant à l'ensemble des utilisateurs U soit un vecteur x_u de m dimensions. Il suffit que le serveur calcule une moyenne des x_u. La représentation structurelle x du modèle mis à jour sera alors :

$$x = \frac{\sum_{u \in U} x_u}{|U|}$$

Malheureusement, dans certaines situations, il est possible de déterminer les activités qu'un utilisateur a exercé en analysant sa contribution dans la mise à jour du modèle. En effet la variation de la structure du modèle est représentative de l'entraînement réalisé sur le téléphone de l'utilisateur. C'est pourquoi, il peut être possible pour un serveur malicieux de remonter jusqu'aux données de l'utilisateur à partir seulement de sa contribution. Cependant, dans le contexte de l'apprentissage fédéré, le serveur n'a pas besoin de connaître la contribution individuelle de chacun des utilisateurs mais seulement la moyenne des contributions. Nous sommes donc confrontés à un problème de calcul multipartie sécurisé.

1.3 Les moyens techniques

Dans le cadre de notre étude, nous utilisons un réseau de neurones afin de prédire les activités physiques des utilisateurs. Ce réseau a été spécialement pensé et créé par un collaborateur du projet initial : Monsieur Théo Jourdan. Les données utilisées par notre modèle durant cette expérience proviennent du dataset MotionSense [1] [2]. Une fois toutes les problématiques liées au modèle et aux données réglées, il nous a fallu nous concentrer sur l'entraînement et la classification. Nous avons donc comparé les différentes solutions de machine learning existantes pour trouver celle qui serait la plus cohérente avec notre projet.

Durant l'implémentation de notre preuve de concept, nous nous sommes confrontés à des problèmes de compatibilité entre les différents frameworks. En effet le modèle a été initialement entraîné avec la solution de Google Tensorflow puis devait être converti en un modèle TensorFlowLite exécutable sur téléphone. Cependant, cette dernière librairie ne permet pas l'entraînement directement sur mobile, mais seulement la prédiction.

La solution pour résoudre ce problème était donc d'adapter à la main l'API Java de TensorFlow à un environnement mobile (Android). Cependant, il est actuellement impossible de réaliser ce passage d'une architecture à l'autre sans recompiler totalement l'API Java TensorFlow et l'application Android. Il faut de ce fait comprendre comment fonctionnent précisément ces technologies et leur code source afin de pouvoir espérer les compiler pour une certaine architecture compatible. Pour pouvoir nous concentrer sur le sujet principal du projet, nous avons décidé de réaliser tout un protocole de federated learning en simulant les téléphones Android par un script python. Ces clients communiquent avec le serveur comme le ferait une application basique sur mobile mais permettent cependant d'entraîner un modèle avec TensorFlow. Cette abstraction nous a permis de plus nous focaliser sur le coeur du projet sans perdre des dizaines d'heures de travail sur la compréhension du noyau Android.

L'API TensorFlow repose sur la librairie Keras écrite principalement en Python. Pour cette raison, nous avons décidé de développer le serveur et les clients en Python.

1.4 Focus sur un protocole sécurisé d'agrégation

Dans cette étude, nous avons choisi d'implémenter le protocole décrit en détails dans le document *Practical Secure Aggregation for Privacy-Preserving Machine Learning* [3]. L'essentiel de notre travail, ici, a donc été de comprendre les processus mathématiques sous-jacents à ce protocole et reproduire son fonctionnement le plus fidèlement possible à l'aide du langage de programmation Python.

L'idée générale de ce protocole est que les clients masquent le vecteur x_u en un vecteur y_u en ajoutant ou soustrayant des constantes. Ces constantes sont définies entre un utilisateur u_1 et un utilisateur u_2 tel que $y_{u_1} + y_{u_2} = x_{u_1} + x_{u_2}$. Cela signifie que les masques des utilisateurs u_1 et u_2 s'annulent lors de la somme.

1.4.1 Masquage du vecteur secret x_u

Pour obtenir un masque commun entre un utilisateur u_1 et un utilisateur u_2, l'idée est de générer une clé partagée entre eux utilisée comme graine d'un générateur de nombre pseudo-aléatoire.

ECDH : protocole d'échange de clés Diffie-Hellman basé sur les courbes elliptiques Afin que deux utilisateurs puissent se mettre d'accord sur un masque commun, chaque utilisateur u génère une paire de clés asymétriques C_u tel que C_u^{SK} représente la clé privée (Secrète) et C_u^{PK} la clé Publique. Il partage par la suite C_u^{PK} avec les autres utilisateurs. Ces derniers génèrent une clé partagée entre chacun des utilisateurs et eux même en utilisant le protocole d'échange de clés Diffie-Hellman basé sur les courbes elliptiques [4].

AES Mode CTR : un générateur de nombre pseudo-aléatoire Une fois la clé partagée générée, chaque utilisateur l'utilise pour créer un masque de longueur souhaitée en utilisant le chiffrement symétrique AES en mode CTR (counter). Ce mode permet à AES de fonctionner comme un générateur de nombre pseudo-aléatoire. C'est à dire qu'à partir d'une clé et d'une graine, l'algorithme générera toujours la même suite de bits, et ce de manière infinie. En quelque sorte, on augmente la taille de la clé partagée afin d'obtenir un masque de la taille du vecteur secret x_u.

En conclusion, chaque utilisateur u calcule le vecteur masqué y_u de la manière suivante :

$$y_u = x_u + \sum_{v \in U : u < v} p_{u,v} - \sum_{v \in U : u > v} p_{v,u}$$

Ainsi, lorsque le serveur calcule la somme totale $\sum_{u \in U} y_u$, les masques s'annulent automatiquement sans que le serveur ne connaisse la contribution de chacun des utilisateurs x_u.

Un problème subsiste : comment assurer le calcul juste de la somme si un utilisateur ne communique plus au cours du protocole (par exemple, après avoir partagé sa clé publique C_u^{PK}) ?

1.4.2 Shamir : le partage de secret entre plusieurs personnes

Afin que le serveur puisse calculer le masque p_u, v en cas d'abandon de l'utilisateur u, il doit connaître sa clé privée C_u^{SK}. Pour ce faire, l'utilisateur u va non seulement envoyer sa clé public C_u^{PK}, mais aussi une partie de sa clé privée C_u^{SK}. L'algorithme de Shamir [5] est utilisé pour diviser la clé privée en $|U|$ parties telles que t parties permettent de retrouver la clé privée (avec $t < |U|$). Il envoie une partie à chaque utilisateur sous forme chiffrée avec un algorithme de chiffrement symétrique en utilisant la clé partagée. Par conséquent, si un utilisateur abandonne, le serveur n'a qu'à demander aux utilisateurs restants les parties de la clé privée C_u^{SK} et ainsi calculer les masques partagés entre l'utilisateur u et tous les autres utilisateurs v. On voit alors l'importance qu'au moins t utilisateurs déroulent le protocole jusqu'à la fin.

Un problème subsiste : si le serveur peut retrouver la clé C_u^{SK}, alors il peut remonter au vecteur secret x_u. Comment garantir qu'il reste secret ?

1.4.3 Le double masque

En effet, un serveur corrompu peut très bien simuler le fait qu'un utilisateur u ait perdu sa connexion pour retrouver C_u^{SK} auprès des utilisateurs restants. L'idée est alors de générer un deuxième masque p_u à partir d'une graine aléatoire b_u en utilisant AES en mode CTR (voir ci dessus). Cette graine b_u est alors divisée en t parties et envoyée à chaque utilisateur sous forme chiffrée de la même sorte que la clé privée C_u^{SK}. Le vecteur y_u doublement masqué est alors :

$$y_u = x_u + p_u + \sum_{v \in U : u < v} p_{u,v} - \sum_{v \in U : u > v} p_{v,u}$$

Pendant la dernière étape du protocole, le serveur a alors le choix entre demander une partie de la clé privée C_u^{SK} **OU** une partie de b_u pour un utilisateur u. Ainsi, une fois que le serveur a obtenu les morceaux de clefs secrètes pour chaque utilisateur déconnecté et les parties de b_u, il peut supprimer les deux masques pour révéler la somme totale des contributions des utilisateurs restants.

2 Expérimentations

Notre processus d'expérimentation est découpé en 3 parties, une pour chaque type d'apprentissage différent. Chacune de ces parties consistera à mesurer plusieurs variables :

1. La qualité de prédiction du modèle entraîné
2. Le temps d'exécution total (transmission + apprentissage)
3. Les variations de temps d'exécution en fonction du nombre de clients connectés
4. L'évolution de la qualité des prédictions au fur et à mesure du nombre d'exécutions du protocole.

Le but de notre cas d'étude est d'analyser les performances de 3 possibilités d'apprentissage :

1. 1er cas : Entraînement entièrement fait sur serveur, avec envoi des données récoltées par les clients mobiles
2. 2ème cas : Entraînement sur mobile et agrégation des poids et biais de tous les clients sur le serveur de manière non sécurisée
3. 3ème cas : Entraînement sur mobile et envoi des poids et des biais de tous les terminaux sur le serveur de manière sécurisée

Note : L'intégralité de notre code source est disponible de manière publique :
https://github.com/kistora/federated-learning-secure-aggregation

2.1 Mode 1

Le premier cas demande un envoi d'une quantité considérable de données, et donc une forte consommation de bande passante, mais permet de faire tout l'apprentissage sur le serveur centralisé, plus puissant. Il s'affranchit des problématiques, encore actuelles, liées à un apprentissage directement sur mobile. Toutefois, il implique d'envoyer à un serveur centralisé des données brutes, qui peuvent parfois avoir une valeur confidentielle aux yeux de l'utilisateur, comme des données de santé, etc...

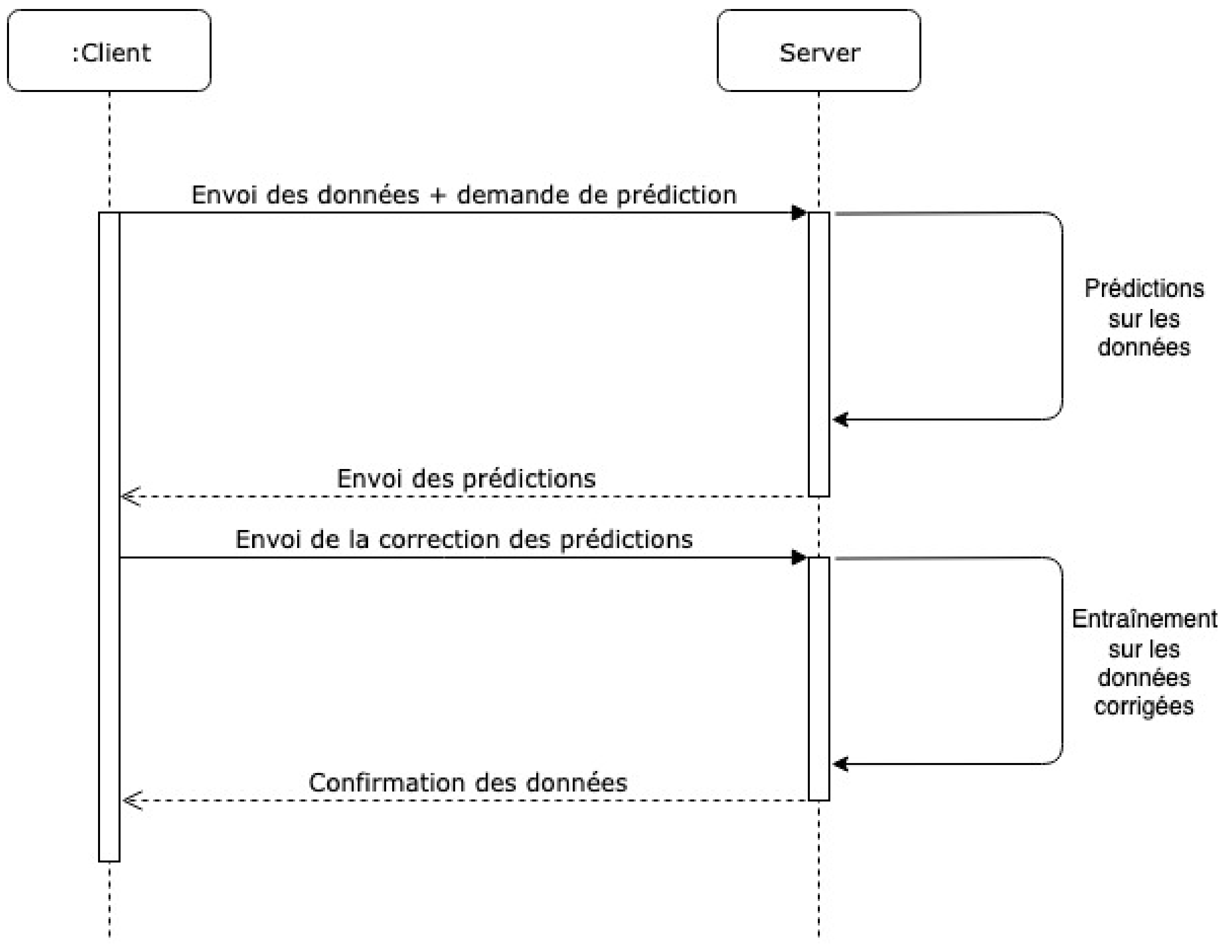

:Client
Server
Envoi des données + demande de prédiction
Prédictions sur les données
Envoi des prédictions
Envoi de la correction des prédictions
Entraînement sur les données corrigées
Confirmation des données

FIGURE 3 – Diagramme de séquence du Mode 1

2.2 Mode 2

Le deuxième cas permet de faire l'entraînement directement sur le dispositif mobile, mais demande donc une puissance de calcul très élevée par rapport à celle d'un téléphone, qui n'est pas forcément fait pour ces applications. La bande passante nécessaire est considérablement réduite, car on n'envoie que des vecteurs de poids et de biais, et la communication est relativement rapide, mais l'agrégation n'est toujours pas sécurisée. De plus, l'envoi des données au niveau du serveur est également nécessaire, et diminue quelque peu la sensibilité à chaque utilisateur du réseau raffiné par le serveur.

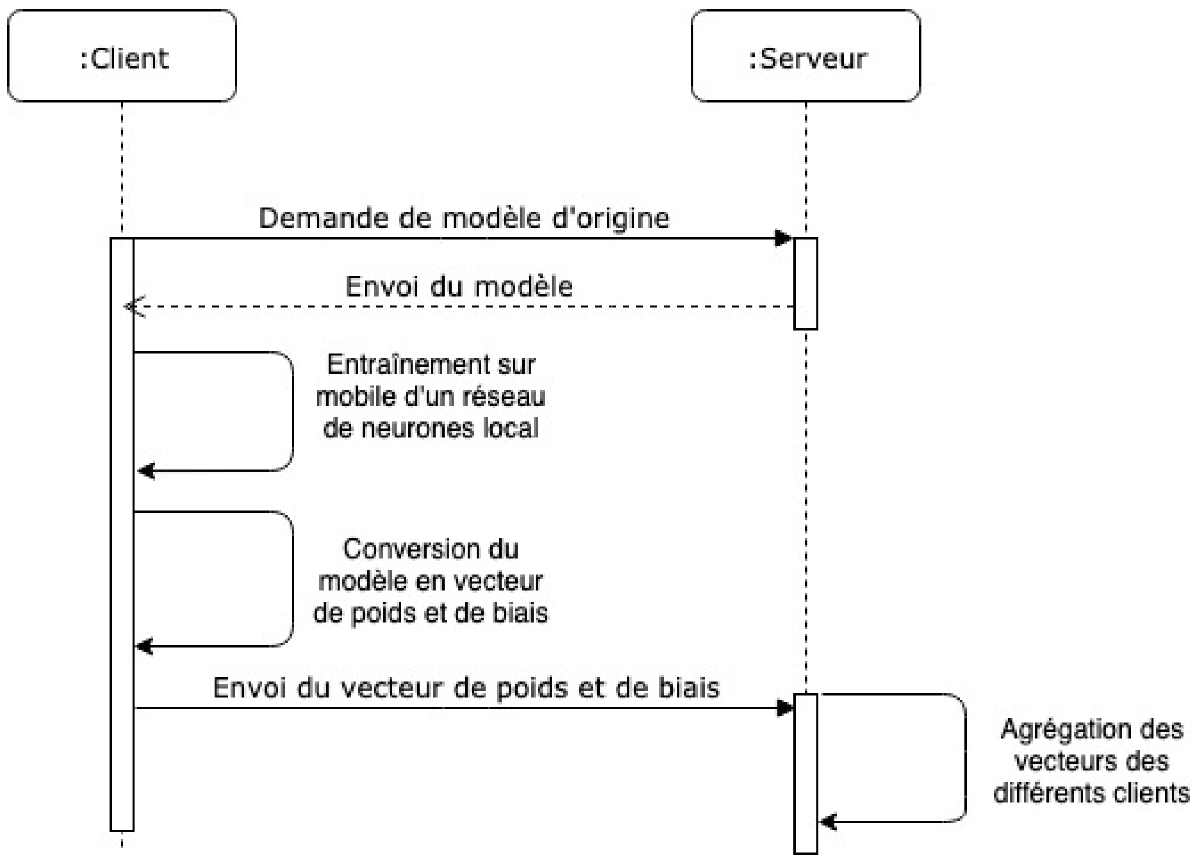

:Client
:Serveur
Demande de modèle d'origine
Envoi du modèle
Entraînement sur mobile d'un réseau de neurones local
Conversion du modèle en vecteur de poids et de biais
Envoi du vecteur de poids et de biais
Agrégation des vecteurs des différents clients

FIGURE 4 – Diagramme de séquence du Mode 2

2.3 Mode 3

Le troisième cas est similaire au deuxième, sauf qu'on utilise un protocole sécurisé pour envoyer les données au serveur. Le but étant de garantir que celui-ci ne sera pas capable de remonter aux données d'un utilisateur à partir de sa contribution de poids et de biais envoyés, dans le cas où le serveur soit compromis. Il présente les mêmes avantages et les mêmes inconvénients, à ceci près que le protocole d'envoi est ralenti par la sécurisation et les vérifications nécessaires. Les connexions mobiles étant très volatile et sujettes à des déconnexions beaucoup plus fréquentes qu'une connexion client-serveur classique, il semble essentiel d'implémenter un protocole permettant de gérer les cas de déconnexion, tout en empêchant le serveur de remonter aux contributions de chaque terminal mobile. Nous allons donc étudier les performances en temps d'exécution et en précision d'agrégation, d'un protocole de partage de secrets sécurisé décrit dans *Practical Secure Aggregation for Privacy-Preserving Machine Learning* [3]. Le but étant ici de déterminer les possibilités d'utilisations éventuelles dans un cadre d'utilisation du protocole pour un envoi de données depuis de multiples terminaux mobiles.

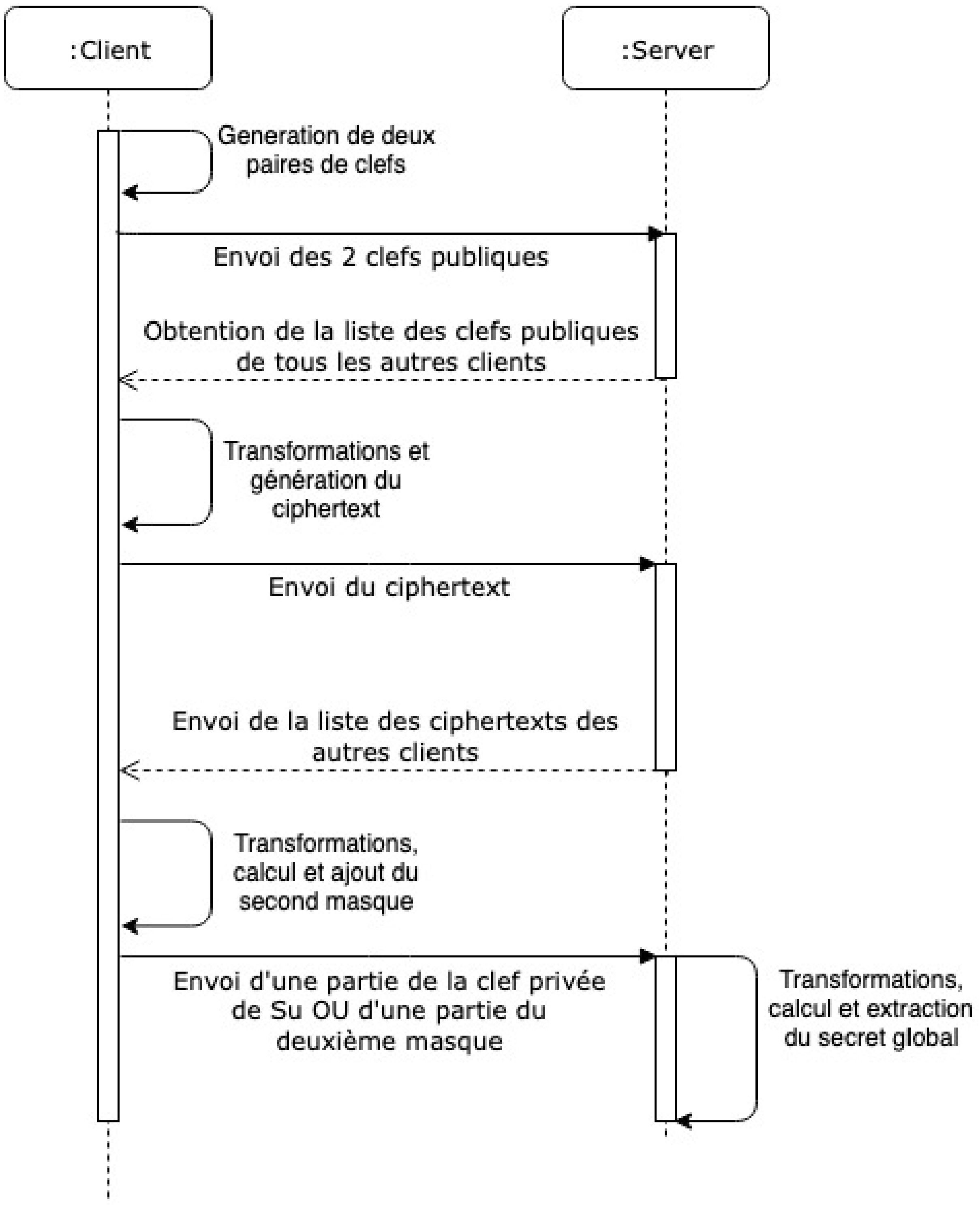

:Client
:Server
Generation de deux paires de clefs
Envoi des 2 clefs publiques
Obtention de la liste des clefs publiques de tous les autres clients
Transformations et génération du ciphertext
Envoi du ciphertext
Envoi de la liste des ciphertexts des autres clients
Transformations, calcul et ajout du second masque
Envoi d'une partie de la clef privée de Su OU d'une partie du deuxième masque
Transformations, calcul et extraction du secret global

Figure 5 – Diagramme de séquence du Mode 3

3 Résultats

Dans un premier temps, nous mesurons les variables de temps d'exécution et de précision du modèle afin de comparer les 3 modes que nous avons implémentés. Afin de garantir des valeurs cohérentes selon les modes, nos mesures sont effectuées sur le même ordinateur. Pour ces différents tests, nous fixons le nombre de clients à 5. Dans un second temps, nous mesurerons l'impact du nombre de clients sur le temps d'exécution.

3.1 Les temps d'exécution

Le temps d'exécution est calculé entre la réception de la 1ère requête client et la publication du résultat du test de précision du modèle.

Pour le mode 1, le temps d'exécution comporte la réception de toutes les données clients sur lesquelles entraîner le modèle, l'entraînement du modèle puis son évaluation sur le jeu de données de test.

Pour le mode 2, le temps d'exécution comporte l'entraînement du modèle par chaque client en parallèle, la réception de tous les modèles clients entraînés, l'agrégation des modèles puis l'évaluation du modèle obtenu sur le jeu de données de test.

Pour le mode 3, le temps d'exécution comporte la réception de tous les modèles clients via le protocole sécurisé, l'agrégation des modèles et l'évaluation du modèle obtenu. L'entraînement du modèle est réalisé indépendamment par chaque appareil après réception des paramètres d'initialisation lors du protocole sécurisé.

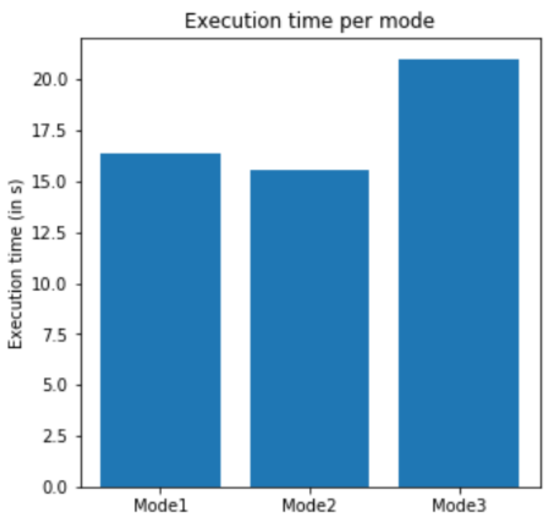


FIGURE 6 – Graphique du temps d'exécution moyen, en seconde, pour chaque mode.

La figure 6 montre le temps d'exécution moyen selon le mode choisi. Le temps moyen correspond à la moyenne de 10 essais réalisés sur un même mode. Le mode 1 nous sert de mode témoin : apprentissage non fédéré. Nous observons un temps moyen d'exécution plus court pour le mode 2. Cela s'explique par l'entraînement du modèle sur une jeu de données plus petit du côté client. Chaque appareil entraîne le modèle sur une partie du jeu de données : l'entraînement est donc parallélisé. Un autre point pouvant expliquer cet écart de temps est la taille des informations à transmettre au serveur. En effet, la représentation structurelle du modèle (vecteur de poids et de biais) est nettement plus petite que les données brutes. Le mode 3 est en moyenne plus long car le protocole sécurisé d'agrégation demande plusieurs échanges et l'utilisation de clés de chiffrement asymétriques. Néanmoins l'écart représente seulement 25% par rapport au mode 2 (avec 5 clients). Notre jeu de données est assez basique. Un jeu de données plus complexe et plus grand pourrait faire augmenter de façon significative le temps nécessaire à l'envoi des données au serveur, alors que la représentation structurelle du modèle ne varie pas en taille. Cela pourrait grandement faire varier l'écart entre le mode 1 et les modes 2 et 3, rendant l'écart entre le mode 2 et le mode 3 dérisoire face à la protection de la vie privée.

3.2 La précision

La précision du modèle est calculée lors de l'évaluation du modèle sur un jeu de données de test, indépendant du jeu de données d'entraînement. Le processus d'évaluation s'effectue par le serveur après entraînement du modèle (mode 1) ou après agrégation des modèles des clients (modes 2 et 3).

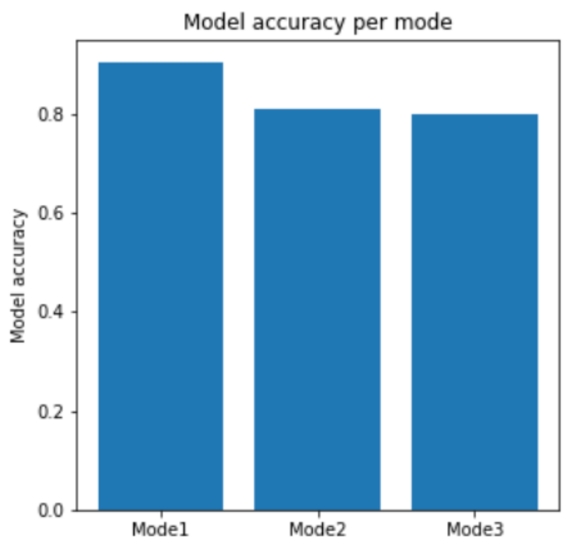


Figure 7 – Graphique de la précision, entre 0 et 1 pour chaque mode.

La figure 7 montre la précision moyenne en fonction du mode d'entraînement du modèle (sur 10 essais réalisés pour chaque mode). Nous observons une meilleure précision en moyenne pour le mode 1 par rapport aux modes 2 et 3. Cet écart de l'ordre de 10% s'explique par une agrégation des modèles pour les modes 2 et 3 qui semble réduire la qualité du modèle final. La comparaison entre les modes 2 et 3 confirme que le protocole sécurisé n'impacte pas la qualité du modèle final.

3.3 Évolution de la précision en fonction des itérations

La précision du modèle est dépendante du jeu de données d'entraînement, et du nombre d'itérations du protocole d'agrégation réalisés.

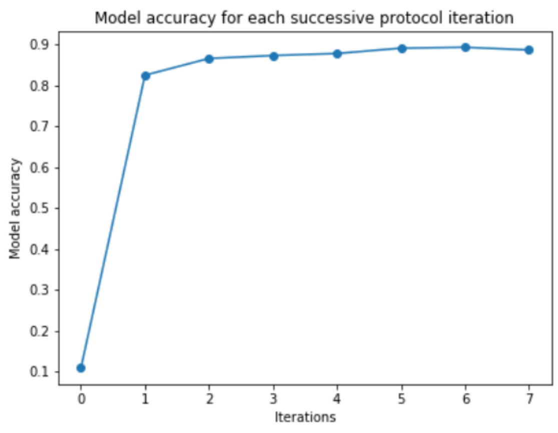


Figure 8 – Graphique de la précision, entre 0 et 1 pour le mode 3, après chaque itération, selon le mode 3.

La figure 8 montre la précision en fonction du nombre d'itérations du protocole, selon le mode 3. Nous observons qu'augmenter le nombre d'itérations implique l'augmentation de la précision.

Cette expérimentation simule l'entraînement fédératif itératif, avec la possibilité que les appareils clients varient selon les itérations. Cette variation implique une variation des jeux de données sur lesquels s'appuie le modèle pour s'entraîner et donc une précision plus grande.

On observe aussi que le mode 3 obtient pratiquement la même précision que le mode 1 en seulement X itérations. Ceci remet en cause notre remarque sur la perte de qualité d'entraînement engendrée par l'agrégation (sécurisée (mode 3) ou non (mode 2)).

Il reste toutefois important de souligner que le nombre d'itération n'est pas une garantie de réussite. En effet, celui-ci peut provoquer un phénomène appelé l'Overfitting, le surapprentissage, en français, qui correspond à une situation où le modèle est trop adapté au dataset d'apprentissage, ce qui le rend moins efficace pour les prédictions sur les données futures.

3.4 Impact du nombre de clients sur le temps d'exécution

Le nombre de clients et son impact sur le temps d'exécution est uniquement étudié sur le mode 3. Les clients travaillant en parallèles, seul le temps nécessaire pour recevoir les différentes informations (clés publiques, textes chiffrés,...) et le temps d'agrégation des données est impacté par le nombre de clients.

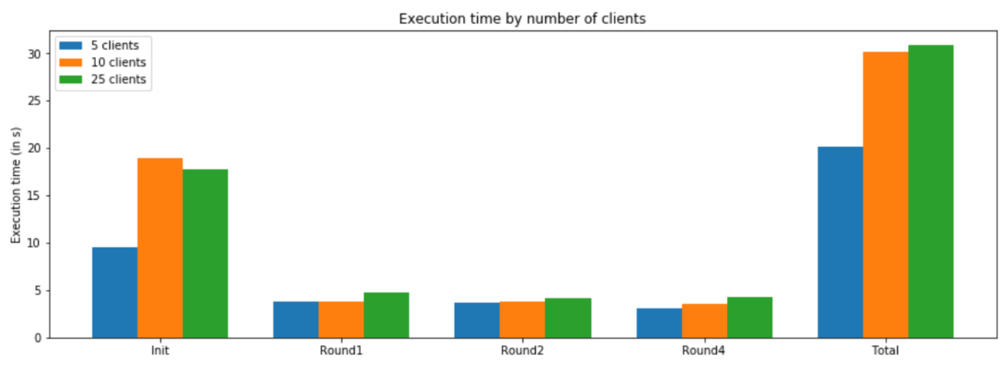


FIGURE 9 – Graphique du temps d'exécution moyen en fonction du nombre de clients, selon le mode 3.

La figure 9 montre l'évolution du temps d'exécution moyen en fonction du nombre de clients, selon le mode 3. Nous observons qu'augmenter le nombre de clients augmente le temps d'exécution. Cette variable est donc importante lors du dimensionnement du système. En particulier la valeur minimum du nombre de clients par itération peut jouer un rôle important pour limiter le temps d'exécution et améliorer la vitesse totale d'une itération.

Cette expérimentation impose la simulation d'un grand nombre de clients, chacun ayant à entraîner un modèle en parallèle. L'entraînement à lieu durant la phase d'initialisation du protocole, phase la plus gourmande par rapport au temps total. Une limitation matérielle apparaît rapidement pour mesurer l'impact de cette variable : il est difficile de simuler plus de 25 clients sur un même ordinateur. De plus, un système d'apprentissage fédératif reposant sur des appareils clients tel que les smartphones pourrait compter plusieurs milliers de clients par itération. Il est donc difficile de valider le passage à l'échelle et une expérimentation sur le terrain est nécessaire afin de préciser l'impact des variables "nombre de clients" et " minimum du nombre de clients par itération".

4 Conclusion

Nous vous avons présenté ici une implémentation du federated learning qui permet une communication sécurisée entre les utilisateurs et le serveur. Notre solution limite la fuite de données personnelles en ajoutant plusieurs couches de chiffrement et par ce fait respecte la vie privée de tous les clients connectés. Notre projet et notre démarche s'inscrivent donc dans cette tendance actuelle de protection des données et contribuent à la résolution de différents enjeux éthiques. Notre solution n'est sûrement pas optimisée au maximum mais représente une très bonne base à de nouveaux travaux sur ce sujet.

5 Pistes d'améliorations et travaux futurs

Ce projet de recherche ayant été réalisé sur une durée relativement courte, nous n'avons pas pu explorer toutes les idées, ni réaliser tous les tests auxquels nous avons pensé. Voilà donc une liste non exhaustive de quelques points laissés pour des travaux futurs.

5.1 Protocole

Du côté de notre protocole, même s'il est fonctionnel, il est envisageable de retravailler certains points. Les deux principaux que nous aimerions étudier sont :

— Ajouter une couche de sécurité avec pour objectif d'empêcher des clients malicieux de transmettre des données mal formées qui pourraient nuire au fonctionnement de l'agrégation ou à la qualité du modèle;

— Mener des recherches pour tenter de limiter les échanges de données entre clients tout en gardant un niveau de sécurité équivalent.

— Implémenter une version sécurisée pour prévenir des attaques "actives"(type MITM entre le client et le serveur).

De plus, il serait intéressant de réaliser différents tests de charge, pour vérifier comment notre simulation réagit à un plus grand nombre de client ($10^5..10^9$).

5.2 Autres

D'autres idées d'amélioration mériteraient d'être étudiées et seront peut-être le sujet d'un nouveau travail :

— Implémenter une version Android des clients de manière à tester notre protocole d'abord sous forme de simulation puis en conditions réelles;

— Travailler et tester l'agrégation entre le modèle spécialisé du client et le modèle général du serveur chez le client de manière à garder une certaine spécialisation.

6 Référence

Références

[1] Mohammad Malekzadeh et al. “Mobile Sensor Data Anonymization”. In : *Proceedings of the International Conference on Internet of Things Design and Implementation.* IoTDI ’19. Montreal, Quebec, Canada : ACM, 2019, p. 49-58. isbn : 978-1-4503-6283-2. doi : `10.1145/3302505.3310068`. url : `http://doi.acm.org/10.1145/3302505.3310068` (visité le 03/12/2019).

[2] Mohammad Malekzadeh et al. “Privacy and Utility Preserving Sensor-Data Transformations”. In : (14 nov. 2019). arXiv : `1911.05996 [cs, eess, stat]`. url : `http://arxiv.org/abs/1911.05996` (visité le 03/12/2019).

[3] Keith Bonawitz et al. “Practical Secure Aggregation for Privacy-Preserving Machine Learning”. In : *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security - CCS ’17.* The 2017 ACM SIGSAC Conference. Dallas, Texas, USA : ACM Press, 2017, p. 1175-1191. isbn : 978-1-4503-4946-8. doi : `10.1145/3133956.3133982`. url : `http://dl.acm.org/citation.cfm?doid=3133956.3133982` (visité le 28/11/2019).

[4] Elaine Barker et al. *Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography.* NIST Special Publication (SP) 800-56A Rev. 3. National Institute of Standards and Technology, 16 avr. 2018. doi : `https://doi.org/10.6028/NIST.SP.800-56Ar3`. url : `https://csrc.nist.gov/publications/detail/sp/800-56a/rev-3/final` (visité le 26/01/2020).

[5] Adi Shamir. “How to Share a Secret”. In : *Commun. ACM* 22.11 (nov. 1979), p. 612-613. issn : 0001-0782. doi : `10.1145/359168.359176`. url : `http://doi.acm.org/10.1145/359168.359176` (visité le 02/12/2019).

7 ANNEXES

7.1 ANNEXE 1 : Practical Secure Aggregation for Privacy-Preserving : protocol description

Practical Secure Aggregation for Privacy-Preserving Machine Learning

Keith Bonawitz
bonawitz@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

Vladimir Ivanov
vlivan@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

Ben Kreuter
benkreuter@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

Antonio Marcedone†
marcedone@cs.cornell.edu
Cornell Tech
2 West Loop Rd.
New York, NY 10044

H. Brendan McMahan
mcmahan@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

Sarvar Patel
sarvar@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

Daniel Ramage
dramage@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

Aaron Segal
asegal@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

Karn Seth
karn@google.com
Google
1600 Amphitheatre Parkway
Mountain View, California 94043

ABSTRACT

We design a novel, communication-efficient, failure-robust protocol for secure aggregation of high-dimensional data. Our protocol allows a server to compute the sum of large, user-held data vectors from mobile devices in a secure manner (i.e. without learning each user's individual contribution), and can be used, for example, in a federated learning setting, to aggregate user-provided model updates for a deep neural network. We prove the security of our protocol in the honest-but-curious and active adversary settings, and show that security is maintained even if an arbitrarily chosen subset of users drop out at any time. We evaluate the efficiency of our protocol and show, by complexity analysis and a concrete implementation, that its runtime and communication overhead remain low even on large data sets and client pools. For 16-bit input values, our protocol offers 1.73× communication expansion for 2^{10} users and 2^{20}-dimensional vectors, and 1.98× expansion for 2^{14} users and 2^{24}-dimensional vectors over sending data in the clear.

CCS CONCEPTS

• **Security and privacy → Privacy-preserving protocols**;

KEYWORDS

privacy-preserving protocols, secure aggregation, machine learning, federated learning

ACM ISBN 978-1-4503-4946-8/17/10…$15.00
https://doi.org/10.1145/3133956.3133982

1 INTRODUCTION

Machine learning models trained on sensitive real-world data promise improvements to everything from medical screening [46] to disease outbreak discovery [37]. And the widespread use of mobile devices means even richer—and more sensitive—data is becoming available [35].

However, large-scale collection of sensitive data entails risks. A particularly high-profile example of the consequences of mishandling sensitive data occurred in 1988, when the video rental history of a nominee for the US Supreme Court was published without his consent [4]. The law passed in response to that incident remains relevant today, limiting how online video streaming services can use their user data [42].

This work outlines an approach to advancing privacy-preserving machine learning by leveraging secure multiparty computation (MPC) to compute sums of model parameter updates from individual users' devices in a secure manner. The problem of computing a multiparty sum where no party reveals its update in the clear—even to the aggregator—is referred to as *Secure Aggregation*. As described in Section 2, the secure aggregation primitive can be used to privately combine the outputs of local machine learning on user devices, in order to update a global model. Training models in this way offers tangible benefits—a user's device can share an update knowing that the service provider will only see that update after it has been averaged with those of other users.

The secure aggregation problem has been a rich area of research: different approaches include works based on generic secure

† Research performed during an internship at Google.

multi-party computation protocols, works based on DC-nets, works based on partially- or fully-homomorphic threshold encryption, and works based on pairwise masking. We discuss these existing works in more detail in Section 9, and compare them to our approach.

We are particularly focused on the setting of mobile devices, where communication is extremely expensive, and dropouts are common. Given these constraints, we would like our protocol to incur no more than twice as much communication as sending the data vector to be aggregated in the clear, and would also like the protocol to be fully robust to users dropping at any point. We believe that previous works do not address this mixture of constraints, which is what motivates our work.

1.1 Our Results

We present a protocol for securely computing sums of vectors, which has a constant number of rounds, low communication overhead, robustness to failures, and which requires only one server with limited trust. In our design the server has two roles: it routes messages between the other parties, and it computes the final result. We present two variants of the protocol: one is more efficient and can be proven secure against honest but curious adversaries, in the plain model. The other guarantees privacy against active adversaries (including an actively adversarial server), but requires an extra round, and is proven secure in the random oracle model. In both cases, we can show formally that the server only learns users' inputs in aggregate, using a simulation-based proof as is standard for MPC protocols. Both variants we present are practical and we present benchmark results from our prototype implementation.

1.2 Organization

In Section 2 we describe the machine learning application that motivates this work. In Section 3 we review the cryptographic primitives we use in our protocol. We then proceed to give a high-level overview of our protocol design in Section 4, followed by a formal protocol description in Section 5. In Section 6 we prove security against honest-but-curious (passive) adversaries and include a high-level discussion of privacy against active adversaries.In Section 7, we give performance numbers based both on theoretical analysis as well as on a prototype implementation. Finally, we discuss some issues surrounding practical deployments and future work in Section 8 and conclude with a discussion of related work in Section 9.

2 SECURE AGGREGATION FOR FEDERATED LEARNING

Consider training a deep neural network to predict the next word that a user will type as she composes a text message. Such models are commonly used to to improve typing efficacy for a phone's on-screen keyboard [30]. A modeler may wish to train such a model on all text messages across a large population of users. However, text messages frequently contain sensitive information; users may be reluctant to upload a copy of them to the modeler's servers. Instead, we consider training such a model in a *Federated Learning* setting, wherein each user maintains a private database of her text messages securely on her own mobile device, and a shared global model is trained under the coordination of a central server based upon highly processed, minimally scoped, ephemeral updates from users [43, 50].

These updates are high-dimensional vectors based on information from the user's private database. Training a neural net is typically done by repeatedly iterating over these updates using a variant of a mini-batch stochastic gradient descent rule [15, 29]. (See Appendix B for details.)

Although each update is ephemeral and contains no more (and typically significantly less) information than the user's private database, a user might still be concerned about what information remains. In some circumstances, it is possible to learn invididual words that a user has typed by inspecting that user's most recent update. However, in the Federated Learning setting, the server does not need to access any *individual* user's update in order to perform stochastic gradient descent; it requires only the element-wise *weighted averages* of the update vectors, taken over a random subset of users. Using a Secure Aggregation protocol to compute these weighted averages[1] would ensure that the server may learn only that *one or more* users in this randomly selected subset wrote a given word, but not *which* users.

Federated Learning systems face several practical challenges. Mobile devices have only sporadic access to power and network connectivity, so the set of users participating in each update step is unpredictable and the system must be robust to users dropping out. Because the neural network may be parameterized by millions of values, updates may be large, representing a direct cost to users on metered network plans. Mobile devices also generally cannot establish direct communications channels with other mobile devices (relying on a server or service provider to mediate such communication) nor can they natively authenticate other mobile devices.

Thus, Federated Learning motivates a need for a Secure Aggregation protocol that:

1. operates on high-dimensional vectors
2. is highly communication efficient, even with a novel set of users on each instantiation
3. is robust to users dropping out
4. provides the strongest possible security under the constraints of a server-mediated, unauthenticated network model

3 CRYPTOGRAPHIC PRIMITIVES

In this section, we discuss the cryptographic primitives and assumptions needed for our construction.

3.1 Secret Sharing

We rely on Shamir's t-out-of-n Secret Sharing [48], which allows a user to split a secret s into n shares, such that any t shares can be used to reconstruct s, but any set of at most $t-1$ shares gives no information about s.

The scheme is parameterized over a finite field $\mathbb{F}$ of size at least $l > 2^k$ (where k is the security parameter of the scheme), e.g. $\mathbb{F} = \mathbb{Z}_p$ for some large public prime p. We note that such a large field size is needed because our scheme requires clients to secret

[1]Computing a secure weighted average given a secure sum operation is straightfoward; for detail, see Appendix B.

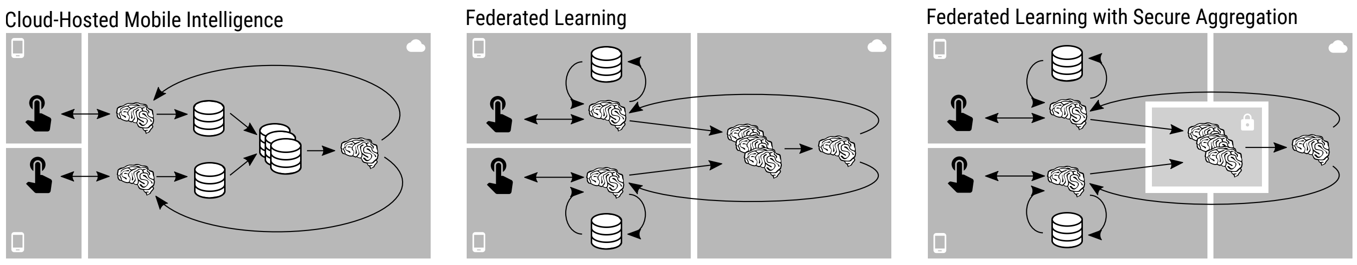

Cloud-Hosted Mobile Intelligence
Federated Learning
Federated Learning with Secure Aggregation

Figure 1: *Left:* In the cloud-centric approach to machine intelligence, user devices interact with cloud-hosted models, generating logs that can be used as training examples. The logs from many users are combined and used to improve the model, which is then used to serve future user requests. *Middle:* In Federated Learning, machine intelligence models are shipped to users' devices where they are both evaluated and trained locally. Summaries of improved models are shared with the server, where they are aggregated into a new model and deployed to user devices. *Right:* When Secure Aggregation is added to Federated Learning, the aggregation of model updates is logically performed by the virtual, incorruptible third party induced by the secure multiparty communication, so that the cloud provider learns only the aggregated model update.

share their secret keys (whose length must be proportional to the security parameter for the security proof to go through). We also assume that integers $1, \ldots, n$ (which will be used to denote users in the protocol) can be identified with distinct field elements in $\mathbb{F}$. Given these parameters, the scheme consists of two algorithms. The sharing algorithm $\mathbf{SS.share}(s, t, \mathcal{U}) \to \{(u, s_u)\}_{u \in \mathcal{U}}$ takes as input a secret s, a set $\mathcal{U}$ of n field elements representing user IDs, and a threshold $t \leq |\mathcal{U}|$; it produces a set of shares s_u, each of which is associated with a different $u \in \mathcal{U}$. The reconstruction algorithm $\mathbf{SS.recon}(\{(u, s_u)\}_{u \in \mathcal{V}}, t) \to s$ takes as input the threshold t and the shares corresponding to a subset $\mathcal{V} \subseteq \mathcal{U}$ such that $|\mathcal{V}| \geq t$, and outputs a field element s.

Correctness requires that $\forall s \in \mathbb{F}, \forall t, n$ with $1 \leq t \leq n$, $\forall \mathcal{U} \subseteq \mathbb{F}$ where $|\mathcal{U}| = n$, if $\{(u, s_u)\}_{u \in \mathcal{U}} \leftarrow \mathbf{SS.share}(s, t, \mathcal{U})$, $\mathcal{V} \subseteq \mathcal{U}$ and $|\mathcal{V}| \geq t$, then $\mathbf{SS.recon}(\{(u, s_u)\}_{u \in \mathcal{V}}, t) = s$. Security requires that $\forall s, s' \in \mathbb{F}$ and any $\mathcal{V} \subseteq \mathcal{U}$ such that $|\mathcal{V}| < t$:

$$\{\{(u, s_u)\}_{u \in \mathcal{U}} \leftarrow \mathbf{SS.share}(s, t, \mathcal{U}) : \{(u, s_u)\}_{u \in \mathcal{V}}\} \equiv$$
$$\{\{(u, s_u)\}_{u \in \mathcal{U}} \leftarrow \mathbf{SS.share}(s', t, \mathcal{U}) : \{(u, s_u)\}_{u \in \mathcal{V}}\}$$

where "≡" denotes that the two distributions are identical.

3.2 Key Agreement

Key Agreement consists of a tuple of algorithms ($\mathbf{KA.param}, \mathbf{KA.gen}, \mathbf{KA.agree}$). The algorithm $\mathbf{KA.param}(k) \to pp$ produces some public parameters (over which our scheme will be parameterized). $\mathbf{KA.gen}(pp) \to (s_u^{SK}, s_u^{PK})$ allows any party u to generate a private-public key pair. $\mathbf{KA.agree}(s_u^{SK}, s_v^{PK}) \to s_{u,v}$ allows any user u to combine their private key s_u^{SK} with the public key s_v^{PK} for any v (generated using the same pp), to obtain a private shared key $s_{u,v}$ between u and v.

The specific Key Agreement scheme we will use is Diffie-Hellman key agreement [19], composed with a hash function. More specifically, $\mathbf{KA.param}(k) \to (\mathbb{G}', g, q, H)$ samples group $\mathbb{G}'$ of prime order q, along with a generator g, and a hash function H[2]; $\mathbf{KA.gen}(\mathbb{G}', g, q, H) \to (x, g^x)$ samples a random $x \leftarrow \mathbb{Z}_q$ as the secret key s_u^{SK}, and g^x as the public key s_u^{PK}; and $\mathbf{KA.agree}(x_u, g^{x_v}) \to s_{u,v}$ outputs $s_{u,v} = H((g^{x_v})^{x_u})$.

[2]In practice, one can use SHA-256.

Correctness requires that, for any key pairs generated by users u and v (using $\mathbf{KA.gen}$ and the same parameters pp), $\mathbf{KA.agree}(s_u^{SK}, s_v^{PK}) = \mathbf{KA.agree}(s_v^{SK}, s_u^{PK})$. For security, in the honest but curious model, we want that for any adversary who is given two honestly generated public keys s_u^{PK} and s_v^{PK} (but neither of the corresponding secret keys s_u^{SK} or s_v^{SK}), the shared secret $s_{u,v}$ computed from those keys is indistinguishable from a uniformly random string. This exactly mirrors the Decisional Diffie-Hellman (DDH) assumption, which we recall below:

Definition 3.1 (Decisional Diffie-Hellman assumption). Let $\mathcal{G}(k) \to (\mathbb{G}', g, q, H)$ be an efficient algorithm which samples a group $\mathbb{G}'$ of order q with generator g, as well as a function $H : \{0,1\}^* \to \{0,1\}^k$. Consider the following probabilistic experiment, parameterized by a PPT adversary M, a bit b and a security parameter k.

$\mathbf{DDH\text{-}Exp}^b_{\mathcal{G},M}(k)$:

(1) $(\mathbb{G}', g, q, H) \leftarrow \mathcal{G}(k)$
(2) $a \leftarrow \mathbb{Z}_q; A \leftarrow g^a$
(3) $b \leftarrow \mathbb{Z}_q; B \leftarrow g^b$
(4) if $b = 1$, $s \leftarrow H(g^{ab})$, else $s \xleftarrow{\$} \{0,1\}^k$
(5) $M(\mathbb{G}', g, q, H, A, B, s) \to b'$
(6) Output 1 if $b = b'$, 0 o/w.

The advantage of the adversary is defined as

$$Adv^{DDH}_{\mathcal{G},M}(k) := |\Pr[\,\mathbf{DDH\text{-}Exp}^1_{\mathcal{G},M}(k) = 1] - \Pr[\mathbf{DDH\text{-}Exp}^0_{\mathcal{G},M}(k) = 1]|$$

We say that the Decisional Diffie-Hellman assumption holds for $\mathcal{G}$ if for all PPT adversaries M, there exists a negligible function ϵ such that $Adv^{DDH}_{\mathcal{G},M}(k) \leq \epsilon(k)$.

Note that, traditionally, the Diffie-Hellman assumption does not directly involve a hash function H (i.e. line step 4 is substituted with "if $b = 1$, $s \leftarrow g^{ab}$, else $s \xleftarrow{\$} \mathbb{G}'$"), and therefore to get from a random element of the group $\mathbb{G}'$ to a uniformly random string (which is necessary to be used as the seed for a **PRG**, or to sample secret keys for other primitives), one has to compose g^{ab} with a secure randomness extractor (which composes well with this specific key agreement operation). For simplicity, we choose to incorporate such an extractor function H in the assumption.

In order to prove security against active adversaries (Theorem 6.5), we need a somewhat stronger security guarantee for Key Agreement, namely that an adversary who is given two honestly generated public keys s_u^{PK} and s_v^{PK}, and also the ability to learn $\mathbf{KA.agree}(s_u^{SK}, s^{PK})$ and $\mathbf{KA.agree}(s_v^{SK}, s^{PK})$ for any s^{PK}s of its choice (but different from s_u^{PK} and s_v^{PK}), still cannot distinguish $s_{u,v}$ from a random string. In order to get this stronger property, we need to rely on a slight variant of the Oracle Diffie-Hellman assumption (ODH) [2], which we call Two Oracle Diffie-Hellman assumption (2ODH):

Definition 3.2 (Two Oracle Diffie-Hellman assumption (2ODH)). Let $\mathcal{G}(k) \to (\mathbb{G}', g, q, H)$ be an efficient algorithm which samples a group $\mathbb{G}'$ of order q with generator g, as well as a function $H : \{0,1\}^* \to \{0,1\}^k$. Consider the following probabilistic experiment, parameterized by a PPT adversary M, a bit b and a security parameter k.

$\mathbf{2ODH\text{-}Exp}^{b}_{\mathcal{G},M}(k)$:

(1) $(\mathbb{G}', g, q, H) \leftarrow \mathcal{G}(k)$
(2) $a \leftarrow \mathbb{Z}_q; A \leftarrow g^a$
(3) $b \leftarrow \mathbb{Z}_q; B \leftarrow g^b$
(4) if $b = 1$, $s \leftarrow H(g^{ab})$, else $s \xleftarrow{\$} \{0,1\}^k$
(5) $M^{O_a(\cdot), O_b(\cdot)}(\mathbb{G}', g, q, H, A, B, s) \to b'$
(6) Output 1 if $b = b'$, 0 o/w.

where $O_a(X)$ returns $H(X^a)$ on any $X \neq B$ (and an error on input B) and similarly $O_b(X)$ returns $H(X^b)$ on any $X \neq A$. The advantage of the adversary is defined as

$$Adv^{2ODH}_{\mathcal{G},M}(k) := |\Pr[\,\mathbf{2ODH\text{-}Exp}^{1}_{\mathcal{G},M}(k) = 1] - \Pr[\mathbf{2ODH\text{-}Exp}^{0}_{\mathcal{G},M}(k) = 1]|$$

We say that the Two Oracle Diffie-Hellman assumption holds for $\mathcal{G}$ if for all PPT adversaries M, there exists a negligible function ϵ such that $Adv^{2ODH}_{\mathcal{G},M}(k) \leq \epsilon(k)$.

This assumption can be directly used to prove the security property we need for Key Agreement: the two oracles $O_a(\cdot), O_b(\cdot)$ formalize the ability of the adversary M to learn $\mathbf{KA.agree}(s_u^{SK}, s^{PK})$ and $\mathbf{KA.agree}(s_v^{SK}, s^{PK})$ for different s^{PK}, and the negligible advantage of M in the above game corresponds to an inability to distinguish between $s = s_{u,v} \leftarrow H(g^{ab})$, and $s \xleftarrow{\$} \{0,1\}^k$.

3.3 Authenticated Encryption

(Symmetric) Authenticated Encryption combines confidentiality and integrity guarantees for messages exchanged between two parties. It consists of a key generation algorithm that outputs a private key[3], an encryption algorithm **AE.enc** that takes as input a key and a message and outputs a ciphertext, and a decryption algorithm **AE.dec** that takes as input a ciphertext and a key and outputs either the original plaintext, or a special error symbol $\perp$. For correctness, we require that for all keys $c \in \{0,1\}^k$ and all messages x, $\mathbf{AE.dec}(c, \mathbf{AE.enc}(c, x)) = x$. For security, we require indistinguishability under a chosen plaintext attack (IND-CPA) and ciphertext integrity (IND-CTXT) as defined in [7]. Informally, the guarantee is that for any adversary M that is given encryptions of messages of its choice under a randomly sampled key c (where c is unknown to M), M cannot distinguish between fresh encryptions under c of two different messages, nor can M create new valid ciphertexts (different from the ones it received) with respect to c with better than negligible advantage.

[3]Without loss of generality, we make the simplifying assumption that the key generation algorithm samples keys as uniformly random strings.

3.4 Pseudorandom Generator

We require a secure Pseudorandom Generator [9, 54] **PRG** that takes in a uniformly random seed of some fixed length, and whose output space is $[0, R)^m$ (i.e. the input space for the protocol). Security for a Pseudorandom Generator guarantees that its output on a uniformly random seed is computationally indistinguishable from a uniformly sampled element of the output space, as long as the seed is hidden from the distinguisher.

3.5 Signature Scheme

The protocol relies on a standard UF-CMA secure signature scheme (**SIG.gen**, **SIG.sign**, **SIG.ver**). The key generation algorithm $\mathbf{SIG.gen}(k) \to (d^{PK}, d^{SK})$ takes as input the security parameter and outputs a secret key d^{SK} and a public key d^{PK}; the signing algorithm $\mathbf{SIG.sign}(d^{SK}, m) \to \sigma$ takes as input the secret key and a message and outputs a signature σ; the verification algorithm $\mathbf{SIG.ver}(d^{PK}, m, \sigma) \to \{0,1\}$ takes as input a public key, a message and a signature, and returns a bit indicating whether the signature should be considered valid. For correctness, we require that $\forall m$,

$$\Pr[(d^{PK}, d^{SK}) \leftarrow \mathbf{SIG.gen}(k), \sigma \leftarrow \mathbf{SIG.sign}(d^{SK}, m) : \mathbf{SIG.ver}(d^{PK}, m, \sigma) = 1] = 1$$

Security demands that no PPT adversary, given a fresh honestly generated public key and access to an oracle producing signatures on arbitrary messages, should be able to produce a valid signature on a message on which the oracle was queried on with more than negligible probability.

3.6 Public Key Infrastructure

To prevent the server from simulating an arbitrary number of clients (in the active-adversary model), we require the support of a public key infrastructure that allows clients to register identities, and sign messages using their identity, such that other clients can verify this signature, but cannot impersonate them. In this model, each party u will register $\left(u, d_u^{PK}\right)$ to a public bulletin board during the setup phase. The bulletin board will only allow parties to register keys for themselves, so it will not be possible for the attacking parties to impersonate honest parties.

4 TECHNICAL INTUITION

We note that our protocol is quite similar to the work of Ács and Castelluccia [3], and we give a detailed comparison between our approaches in Section 9. As in their protocol, we divide the parties into two classes: a single server S that *aggregates* inputs from n client parties $\mathcal{U}$. Each user[4] $u \in \mathcal{U}$ holds a private vector $\boldsymbol{x}_u$ of dimension m; for simplicity we assume that the elements of $\boldsymbol{x}_u$ and $\sum_{u \in \mathcal{U}} \boldsymbol{x}_u$ are in $\mathbb{Z}_R$ for some R. The goal of the protocol is to

[4]We use the terms user and client interchangeably.

compute $\sum_{u\in\mathcal{U}} \boldsymbol{x}_u$ in a secure fashion: at a high level, we guarantee that the server only learns a sum of the clients' inputs containing contributions from at least a large fraction of the users and that the users learn nothing.

4.0.1 *Masking with One-Time Pads.*

The first observation is that $\sum_{u\in\mathcal{U}} \boldsymbol{x}_u$ can be computed with perfect secrecy if $\boldsymbol{x}_u$ is masked in a particular way. Assume a total order on users, and suppose each pair of users (u, v), $u < v$ agree on some random vector $\boldsymbol{s}_{u,v}$. If u adds this to $\boldsymbol{x}_u$ and v subtracts it from $\boldsymbol{x}_v$, then the mask will be canceled when their vectors are added, but their actual inputs will not be revealed. In other words, each user u computes:

$$\boldsymbol{y}_u = \boldsymbol{x}_u + \sum_{v\in\mathcal{U}:u<v} \boldsymbol{s}_{u,v} - \sum_{v\in\mathcal{U}:u>v} \boldsymbol{s}_{v,u} \pmod{R}$$

and sends $\boldsymbol{y}_u$ to the server, and the server computes:

$$\begin{aligned} \boldsymbol{z} &= \sum_{u\in\mathcal{U}} \boldsymbol{y}_u \\ &= \sum_{u\in\mathcal{U}} \left(\boldsymbol{x}_u + \sum_{v\in\mathcal{U}:u<v} \boldsymbol{s}_{u,v} - \sum_{v\in\mathcal{U}:u>v} \boldsymbol{s}_{v,u} \right) \\ &= \sum_{u\in\mathcal{U}} \boldsymbol{x}_u \pmod{R} \end{aligned}$$

There are two shortcomings to this approach. The first is that the users must exchange the random vectors $\boldsymbol{s}_{u,v}$, which, if done naively, would require quadratic communication overhead ($|\mathcal{U}|\times|\boldsymbol{x}|$). The second is that there is no tolerance for a party failing to complete the protocol: if a user u drops out after exchanging vectors with other users, but before submitting $\boldsymbol{y}_u$ to the server, the vector masks associated with u would not be canceled in the sum $\boldsymbol{z}$.

4.0.2 *Efficient Communication and Handling Dropped Users.*

We notice that we can reduce the communication by having the parties agree on common seeds for a pseudorandom generator (PRG) rather than on the entire mask $\boldsymbol{s}_{u,v}$. These shared seeds will be computed by having the parties broadcast Diffie-Hellman public keys and engaging in a key agreement.

One approach to handling dropped-out users would be to notify the surviving users of the drop-out, and to have them each reply with the common seed they computed with the dropped user. This approach still has a problem: additional users may drop out in the recovery phase before replying with the seeds, which would thus require an additional recovery phase for the newly dropped users' seeds to be reported, and so on, leading the number of rounds up to at most the number of users.

We resolve this problem by using a threshold secret sharing scheme and having each user send shares of their Diffie-Hellman secret to all other users. This allows pairwise seeds to be recovered even if additional parties drop out during the recovery, as long as some minimum number of parties (equal to the threshold) remain alive and respond with the shares of the dropped users' keys.

This approach solves the problem of unbounded recovery rounds, but still has an issue: there is a possibility that a user's data might accidentally be leaked to the server. Consider a scenario where a user u is too slow in sending her $\boldsymbol{y}_u$ to the server. The server assumes that the user has dropped, and asks all other users to reveal their shares of u's secret key, in order to remove u's uncancelled masks from $\boldsymbol{z}$. However, just after receiving these shares and computing each of the $\boldsymbol{s}_{u,v}$ values, the server may receive the delayed $\boldsymbol{y}_u$ from u. The server is now able to remove all the masks from $\boldsymbol{y}_u$, and learn $\boldsymbol{x}_u$ in the clear, breaking security for u. Moreover, an adversarial server in the active model can similarly learn $\boldsymbol{x}_u$ simply by lying about whether user u has dropped out.

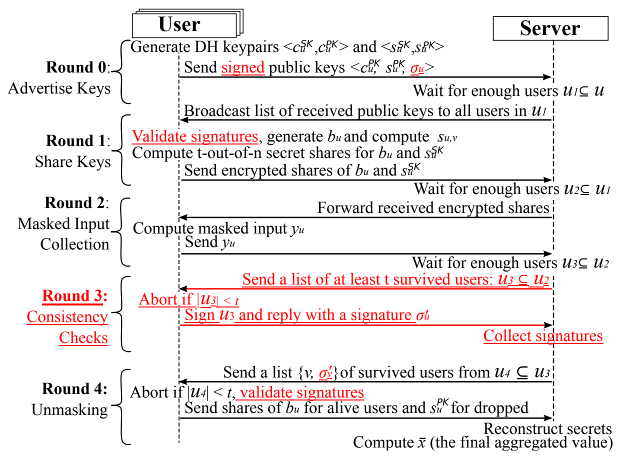

User
Server
Round 0: Advertise Keys
Generate DH keypairs $<c_u^{SK}, c_u^{PK}>$ and $<s_u^{SK}, s_u^{PK}>$
Send signed public keys $<c_u^{PK}, s_u^{PK}, \sigma_u>$
Wait for enough users $\mathcal{U}_1 \subseteq \mathcal{U}$
Broadcast list of received public keys to all users in $\mathcal{U}_1$
Round 1: Share Keys
Validate signatures, generate b_u and compute $s_{u,v}$
Compute t-out-of-n secret shares for b_u and s_u^{SK}
Send encrypted shares of b_u and s_u^{SK}
Wait for enough users $\mathcal{U}_2 \subseteq \mathcal{U}_1$
Round 2: Masked Input Collection
Forward received encrypted shares
Compute masked input y_u
Send y_u
Wait for enough users $\mathcal{U}_3 \subseteq \mathcal{U}_2$
Round 3: Consistency Checks
Send a list of at least t survived users: $\mathcal{U}_3 \subseteq \mathcal{U}_2$
Abort if $|\mathcal{U}_3| < t$
Sign $\mathcal{U}_3$ and reply with a signature σ'_u
Collect signatures
Round 4: Unmasking
Send a list $\{v, \sigma'_v\}$ of survived users from $\mathcal{U}_4 \subseteq \mathcal{U}_3$
Abort if $|\mathcal{U}_4| < t$, validate signatures
Send shares of b_u for alive users and s_u^{PK} for dropped
Reconstruct secrets
Compute $\bar{x}$ (the final aggregated value)

Figure 2: High-level view of our protocol. Red, underlined parts are required to guarantee security in the active-adversary model (and not necessary in the honest-but-curious one).

4.0.3 *Double-Masking to Protect Security.*

To resolve this new security problem, we introduce a double-masking structure that protects $\boldsymbol{x}_u$ even when the server can reconstruct u's masks.

First, each user u samples an additional random seed $\boldsymbol{b}_u$ during the same round as the generation of the $\boldsymbol{s}_{u,v}$ values. During the secret sharing round, the user also generates and distributes shares of $\boldsymbol{b}_u$ to each of the other users. When generating $\boldsymbol{y}_u$, users also add this secondary mask:

$$\begin{aligned} \boldsymbol{y}_u = \boldsymbol{x}_u &+ \textbf{PRG}(\boldsymbol{b}_u) \\ &+ \sum_{v\in\mathcal{U}:u<v} \textbf{PRG}(\boldsymbol{s}_{u,v}) \\ &- \sum_{v\in\mathcal{U}:u>v} \textbf{PRG}(\boldsymbol{s}_{v,u}) \pmod{R} \end{aligned}$$

During the recovery round, the server must make an explicit choice with respect to each user u: from each surviving member v, the server can request *either* a share of the common secret $\boldsymbol{s}_{u,v}$ associated with u *or* a share of the $\boldsymbol{b}_u$ for u; an honest user v will never reveal both kinds of shares for the same user. After gathering at least t shares of $\boldsymbol{s}_{u,v}$ for all dropped users and t shares of $\boldsymbol{b}_u$ for all surviving users, the server can subtract off the remaining masks to reveal the sum.

4.0.4 *Putting it all Together.*

We summarize our protocol in Figure 2 and its asymptotic costs in Figure 3. The computational cost is quadratic for the users, and cubic for the server. As the size of the data vector gets large, the communication and storage overhead for

	User	Server[5]
computation	$O(n^2 + mn)$	$O(mn^2)$
communication	$O(n + m)$	$O(n^2 + mn)$
storage	$O(n + m)$	$O(n^2 + m)$

Figure 3: Cost summary for the protocol.

each of the clients and the server using our protocol approaches a multiplicative constant over sending the data in the clear.

5 A PRACTICAL SECURE AGGREGATION PROTOCOL

The protocol is run (in a synchronous network) between a server and a set of n users, and consists of four rounds. Each user u holds as input a vector $\boldsymbol{x}_u$ (of equal length m) consisting of elements from $\mathbb{Z}_R$ for some R. The server has no input, but can communicate with the users through secure (private and authenticated) channels. At any point, users can drop out of the protocol (in which case they stop sending messages completely), and the server will be able to produce a correct output as long as t of them survive until the last round. To simplify the notation we assume that each user u is assigned a unique "logical identity" (also denoted with u) in the form of an integer between 1 and n, so that no two honest users share the same index[6].

A complete description is provided in Figure 4. We stress that, in the figure, when we say that the server "collects messages from *at least* t users", we mean that the server receives the messages from all users that have not dropped out/aborted in that round (recall that we prove our results in the synchronous setting), and aborts if the number of messages received is less than t. In a practical implementation, the server would wait until a specified timeout (considering all users who did not respond in time to have dropped out), and abort itself if not enough messages are received before such timeout.

To prove security in the active adversary model, we also assume the existence of a Public Key Infrastructure, which for simplicity we abstract away by assuming all clients receive as input (from a trusted third party) public signing keys for all other clients.

Overall, the protocol is parameterized over a security parameter k, which can be adjusted to bound the success probability of any attacker. In all theorems, we implicitly assume that the number of clients n is polynomially bounded in the security parameter. Moreover, some of the primitives also require additional global parameters.

We note that Figure 4 presents both variants of the protocol: in the honest but curious case, since all parties are following the protocol honestly, we can avoid the use of signatures and the need for a PKI (which, most notably, allows us to avoid the **ConsistencyCheck** round entirely).

6 SECURITY ANALYSIS

In our security arguments, we will make use of the following technical lemma. It says that if users' values have uniformly random pairwise masks added to them, then the resulting values look uniformly random, conditioned on their sum being equal to the sum of the users' values. In other words, the pairwise masks hide all information about users' individual inputs, except for their sum.

Lemma 6.1. *Fix n, m, R, $\mathcal{U}$ with $|\mathcal{U}| = n$, and $\{\boldsymbol{x}_u\}_{u\in\mathcal{U}}$ where $\forall u \in \mathcal{U}, \boldsymbol{x}_u \in \mathbb{Z}_R^m$. Then,*

$$\begin{aligned}&\{\{\boldsymbol{p}_{u,v} \xleftarrow{\$} \mathbb{Z}_R^m\}_{u<v}, \quad \boldsymbol{p}_{u,v} := -\boldsymbol{p}_{v,u} \forall u > v \\ &\quad : \{\boldsymbol{x}_u + \sum_{v\in\mathcal{U}\setminus\{u\}} \boldsymbol{p}_{u,v} \pmod R\}_{u\in\mathcal{U}}\} \\ &\equiv \\ &\{\{\boldsymbol{w}_u \xleftarrow{\$} \mathbb{Z}_R^m\}_{u\in\mathcal{U}} \text{ s.t. } \sum_{u\in\mathcal{U}} \boldsymbol{w}_u = \sum_{u\in\mathcal{U}} \boldsymbol{x}_u \pmod R \\ &\quad : \{\boldsymbol{w}_u\}_{u\in\mathcal{U}}\}\end{aligned}$$

where "$\equiv$" denotes that the distributions are identical.

We omit the proof, noting that it can be proved easily by induction on n.

6.1 Honest but Curious Security

Here, we argue that our protocol is a secure multiparty computation in the honest but curious setting, regardless of how and when parties abort. In particular, we prove that when executing the protocol with threshold t, the joint view of the server and any set of less than t (honest) users does not leak any information about the other users' inputs, besides what can be inferred from the output of the computation. Before formally stating our result, we introduce some notation.

We will consider executions of our secure aggregation protocol where the underlying cryptographic primitives are instantiated with security parameter k, a server S interacts with a set $\mathcal{U}$ of n users (denoted with logical identities $1, \ldots, n$) and the threshold is set to t. In such executions, users might abort at any point during the execution, and we denote with $\mathcal{U}_i$ the subset of the users that correctly sent their message to the server at round $i - 1$, such that $\mathcal{U} \supseteq \mathcal{U}_1 \supseteq \mathcal{U}_2 \supseteq \mathcal{U}_3 \supseteq \mathcal{U}_4 \supseteq \mathcal{U}_5$. For example, users in $\mathcal{U}_2 \setminus \mathcal{U}_3$ are exactly those that abort before sending the message to the server in Round 2, but after sending the message of Round 1. If Round **ConsistencyCheck** has been omitted, define $\mathcal{U}_4 := \mathcal{U}_3$.

Denote the input of each user u with $\boldsymbol{x}_u$, and with $\boldsymbol{x}_{\mathcal{U}'} = \{\boldsymbol{x}_u\}_{u\in\mathcal{U}'}$ the inputs of any subset of users $\mathcal{U}' \subseteq \mathcal{U}$.

In such a protocol execution, the *view* of a party consists of its internal state (including its input and randomness) and all messages this party received from other parties (the messages sent by this party do not need to be part of the view because they can be determined using the other elements of its view). Moreover, if the party aborts, it stops receiving messages and the view is not extended past the last message received.

Given any subset $C \subseteq \mathcal{U} \cup \{S\}$ of the parties, let $\mathsf{REAL}_C^{\mathcal{U},t,k}(\boldsymbol{x}_{\mathcal{U}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5)$ be a random variable representing the combined views of all parties in C in the above protocol execution, where the randomness is over the internal randomness of all parties, and the randomness in the setup phase.

Our first theorem shows that the joint view of any subset of honest users (excluding the server) can be simulated given only the

[5]The server can reconstruct n secrets from aligned (t, n)-Shamir shares in $O(t^2 + nt)$ by caching Lagrange coefficients; see section 7.2 for details.
[6]These identities will be bound to the users' keys by a PKI. We rely on this in the active-adversary setting.

Secure Aggregation Protocol

- **Setup**:
  - All parties are given the security parameter k, the number of users n and a threshold value t, honestly generated $pp \leftarrow \textbf{KA.gen}(k)$, parameters m and R such that $\mathbb{Z}_R^m$ is the space from which inputs are sampled, and a field $\mathbb{F}$ to be used for secret sharing. All users also have a private authenticated channel with the server.
  - All users u receive their signing key d_u^{SK} from the trusted third party, together with verification keys d_v^{PK} bound to each user identity v.
- **Round** 0 **(AdvertiseKeys)**:
  User u:
  - Generate key pairs $(c_u^{PK}, c_u^{SK}) \leftarrow \textbf{KA.gen}(pp)$, $(s_u^{PK}, s_u^{SK}) \leftarrow \textbf{KA.gen}(pp)$, and generate $\sigma_u \leftarrow \textbf{SIG.sign}(d_u^{SK}, c_u^{PK}||s_u^{PK})$.
  - Send $(c_u^{PK}||s_u^{PK}||\sigma_u)$ to the server (through the private authenticated channel) and move to next round.
  Server:
  - Collect at least t messages from individual users in the previous round (denote with $\mathcal{U}_1$ this set of users). Otherwise, abort.
  - Broadcast to all users in $\mathcal{U}_1$ the list $\{(v, c_v^{PK}, s_v^{PK}, \sigma_v)\}_{v\in\mathcal{U}_1}$ and move to next round.
- **Round** 1 **(ShareKeys)**:
  User u:
  - Receive the list $\{(v, c_v^{PK}, s_v^{PK}, \sigma_v)\}_{v\in\mathcal{U}_1}$ broadcasted by the server. Assert that $|\mathcal{U}_1| \geq t$, that all the public key pairs are different, and that $\forall v \in \mathcal{U}_1, \textbf{SIG.ver}(d_v^{PK}, c_v^{PK}||s_v^{PK}, \sigma_u) = 1$.
  - Sample a random element $b_u \leftarrow \mathbb{F}$ (to be used as a seed for a **PRG**).
  - Generate t-out-of-$|\mathcal{U}_1|$ shares of s_u^{SK}: $\{(v, s_{u,v}^{SK})\}_{v\in\mathcal{U}_1} \leftarrow \textbf{SS.share}(s_u^{SK}, t, \mathcal{U}_1)$
  - Generate t-out-of-$|\mathcal{U}_1|$ shares of b_u: $\{(v, b_{u,v})\}_{v\in\mathcal{U}_1} \leftarrow \textbf{SS.share}(b_u, t, \mathcal{U}_1)$
  - For each other user $v \in \mathcal{U}_1 \setminus \{u\}$, compute $e_{u,v} \leftarrow \textbf{AE.enc}(\textbf{KA.agree}(c_u^{SK}, c_v^{PK}), u||v||s_{u,v}^{SK}||b_{u,v})$
  - If any of the above operations (assertion, signature verification, key agreement, encryption) fails, abort.
  - Send all the ciphertexts $e_{u,v}$ to the server (each implicitly containing addressing information u, v as metadata).
  - Store all messages received and values generated in this round, and move to the next round.
  Server:
  - Collect lists of ciphertexts from at least t users (denote with $\mathcal{U}_2 \subseteq \mathcal{U}_1$ this set of users).
  - Sends to each user $u \in \mathcal{U}_2$ all ciphertexts encrypted for it: $\{e_{u,v}\}_{v\in\mathcal{U}_2}$ and move to the next round.
- **Round** 2 **(MaskedInputCollection)**:
  User u:
  - Receive (and store) from the server the list of ciphertexts $\{e_{u,v}\}_{v\in\mathcal{U}_2}$ (and infer the set $\mathcal{U}_2$). If the list is of size $< t$, abort.
  - For each other user $v \in \mathcal{U}_2 \setminus \{u\}$, compute $s_{u,v} \leftarrow \textbf{KA.agree}(s_u^{SK}, s_v^{PK})$ and expand this value using a **PRG** into a random vector $\boldsymbol{p}_{u,v} = \Delta_{u,v} \cdot \textbf{PRG}(s_{u,v})$, where $\Delta_{u,v} = 1$ when $u > v$, and $\Delta_{u,v} = -1$ when $u < v$ (note that $\boldsymbol{p}_{u,v} + \boldsymbol{p}_{v,u} = 0 \;\forall u \neq v$). Additionally, define $\boldsymbol{p}_{u,u} = 0$.
  - Compute the user's own private mask vector $\boldsymbol{p}_u = \textbf{PRG}(b_u)$. Then, Compute the masked input vector $\boldsymbol{y}_u \leftarrow \boldsymbol{x}_u + \boldsymbol{p}_u + \sum_{v\in\mathcal{U}_2} \boldsymbol{p}_{u,v} \pmod R$
  - If any of the above operations (key agreement, PRG) fails, abort. Otherwise, Send $\boldsymbol{y}_u$ to the server and move to the next round.
  Server:
  - Collect $\boldsymbol{y}_u$ from at least t users (denote with $\mathcal{U}_3 \subseteq \mathcal{U}_2$ this set of users). Send to each user in $\mathcal{U}_3$ the list $\mathcal{U}_3$.
- **Round 3 (ConsistencyCheck)**:
  User u:
  - Receive from the server a list $\mathcal{U}_3 \subseteq \mathcal{U}_2$ consisting of at least t users (including itself). If $\mathcal{U}_3$ is smaller than t, abort.
  - Send to the server $\sigma'_u \leftarrow \textbf{SIG.sign}(d_u^{SK}, \mathcal{U}_3)$.
  Server:
  - Collect σ'_u from at least t users (denote with $\mathcal{U}_4 \subseteq \mathcal{U}_3$ this set of users). Send to each user in $\mathcal{U}_4$ the set $\{v, \sigma'_v\}_{v\in\mathcal{U}_4}$.
- **Round** 4 **(Unmasking)**:
  User u:
  - Receive from the server a list $\{v, \sigma'_v\}_{v\in\mathcal{U}_4}$. Verify that $\mathcal{U}_4 \subseteq \mathcal{U}_3$, that $|\mathcal{U}_4| \geq t$ and that $\textbf{SIG.ver}(d^{PK}, \mathcal{U}_3, \sigma'_v) = 1$ for all $v \in \mathcal{U}_4$ (otherwise abort).
  - For each other user v in $\mathcal{U}_2 \setminus \{u\}$, decrypt the ciphertext $v'||u'||s_{v,u}^{SK}||b_{v,u} \leftarrow \textbf{AE.dec}(\textbf{KA.agree}(c_u^{SK}, c_v^{PK}), e_{v,u})$ received in the **MaskedInputCollection** round and assert that $u = u' \land v = v'$.
  - If any of the decryption operations fail (in particular, the ciphertext does not correctly authenticate), abort.
  - Send a list of shares to the server, which consists of $s_{v,u}^{SK}$ for users $v \in \mathcal{U}_2 \setminus \mathcal{U}_3$ and $b_{v,u}$ for users in $v \in \mathcal{U}_3$.
  Server (generating the output):
  - Collect responses from at least t users (denote with $\mathcal{U}_5$ this set of users).
  - For each user in $u \in \mathcal{U}_2 \setminus \mathcal{U}_3$, reconstruct $s_u^{SK} \leftarrow \textbf{SS.recon}(\{s_{u,v}^{SK}\}_{v\in\mathcal{U}_5}, t)$ and use it (together with the public keys received in the **AdvertiseKeys** round) to recompute $\boldsymbol{p}_{v,u}$ for all $v \in \mathcal{U}_3$ using the PRG.
  - For each user $u \in \mathcal{U}_3$, reconstruct $b_u \leftarrow \textbf{SS.recon}(\{b_{u,v}\}_{v\in\mathcal{U}_5}, t)$ and then recompute $\boldsymbol{p}_u$ using the PRG.
  - Compute and output $\boldsymbol{z} = \sum_{u\in\mathcal{U}_3} \boldsymbol{x}_u$ as $\sum_{u\in\mathcal{U}_3} \boldsymbol{x}_u = \sum_{u\in\mathcal{U}_3} \boldsymbol{y}_u - \sum_{u\in\mathcal{U}_3} \boldsymbol{p}_u + \sum_{u\in\mathcal{U}_3, v\in\mathcal{U}_2\setminus\mathcal{U}_3} \boldsymbol{p}_{v,u}$

Figure 4: Detailed description of the Secure Aggregation protocol. Red, underlined parts are required to guarantee security in the active-adversary model (and not necessary in the honest-but-curious one).

knowledge of the inputs of those users. Intuitively, this means that those users learn "nothing more" than their own inputs.

Theorem 6.2 (Honest But Curious Security, against clients only). *There exists a PPT simulator* SIM *such that for all* $k, t, \mathcal{U}$ *with* $t \leq |\mathcal{U}|, \boldsymbol{x}_{\mathcal{U}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5$ *and* C *such that* $C \subseteq \mathcal{U}$, $\mathcal{U} \supseteq \mathcal{U}_1 \supseteq \mathcal{U}_2 \supseteq \mathcal{U}_3 \supseteq \mathcal{U}_4 \supseteq \mathcal{U}_5$, *the output of* SIM *is perfectly indistinguishable from the output of* $\mathsf{REAL}_C^{\mathcal{U},t,k}$*:*

$$\mathsf{REAL}_C^{\mathcal{U},t,k}(\boldsymbol{x}_{\mathcal{U}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5)$$
$$\equiv$$
$$\mathsf{SIM}_C^{\mathcal{U},t,k}(\boldsymbol{x}_C, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5)$$

Proof. Note that, since the view of the server is omitted, the joint view of the parties in C does not depend (in an information theoretic sense) on the inputs of the parties not in C. The simulator can therefore produce a perfect simulation by running the honest but curious users on their true inputs, and all other users on a dummy input (for example, a vector of 0s), and outputting the simulated view of the users in C. In more detail, the only value sent by the honest parties which depend on their input is $\boldsymbol{y}_u$ (sent to the server in round **MaskedInputCollection**). One can easily note that the response sent by the server to the users in round **MaskedInputCollection** just contains a list of user identities which depends on which users responded on the previous round, but not on the specific $\boldsymbol{y}_u$ values of the responses. This means that the simulator can use dummy values for the inputs of all honest parties not in C, and the joint view of users in C will be identical to that in $\mathsf{REAL}^{\mathcal{U},t,k}$. □

In our next theorem, we consider security against an honest-but-curious server, who can additionally combine knowledge with some honest-but-curious clients. We show that any such group of honest-but-curious parties can be simulated given the inputs of the clients in that group, and only the *sum* of the values of the remaining clients. Intuitively, this means that those clients and the server learn "nothing more" than their own inputs, and the sum of the inputs of the other clients. Additionally, if too many clients abort before Round **Unmasking**, then we show that we can simulate the view of the honest-but-curious parties given *no information* about the remaining clients' values. Thus, in this case, the honest-but-curious parties learn *nothing* about the remaining clients' values.

Importantly, the view to be simulated must contain fewer than t honest-but-curious clients, or else we cannot guarantee security.

Theorem 6.3 (Honest But Curious Security, with curious server). *There exists a PPT simulator* SIM *such that for all* $t, \mathcal{U}, \boldsymbol{x}_{\mathcal{U}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4$, *and* C *such that* $C \subseteq \mathcal{U} \cup \{S\}$, $|C \setminus \{S\}| < t$, $\mathcal{U} \supseteq \mathcal{U}_1 \supseteq \mathcal{U}_2 \supseteq \mathcal{U}_3 \supseteq \mathcal{U}_4 \supseteq \mathcal{U}_5$, *the output of* SIM *is computationally indistinguishable from the output of* $\mathsf{REAL}_C^{\mathcal{U},t,k}$*:*

$$\mathsf{REAL}_C^{\mathcal{U},t,k}(\boldsymbol{x}_{\mathcal{U}}, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5)$$
$$\approx_c \mathsf{SIM}_C^{\mathcal{U},t,k}(\boldsymbol{x}_C, z, \mathcal{U}_1, \mathcal{U}_2, \mathcal{U}_3, \mathcal{U}_4, \mathcal{U}_5)$$

where

$$z = \begin{cases} \sum_{u \in \mathcal{U}_3 \setminus C} \boldsymbol{x}_u & \text{if } |\mathcal{U}_3| \geq t \\ \bot & \text{otherwise.} \end{cases}$$

Proof. We prove the theorem by a standard hybrid argument. We will define a simulator SIM through a series of (polynomially many) subsequent modifications to the random variable REAL, so that any two subsequent random variables are computationally indistinguishable.

Hyb_0 This random variable is distributed exactly as REAL, the joint view of the parties C in a real execution of the protocol.

Hyb_1 In this hybrid, we change the behavior of simulated honest parties in the set $\mathcal{U}_2 \setminus C$, so that instead of using **KA.agree**(c_u^{SK}, c_v^{PK}) to encrypt and decrypt messages to other users v in the same set, they use a uniformly random encryption key $c_{u,v}$ chosen by the simulator. The Decisional Diffie-Hellman assumption (as recalled in Definition 3.1) guarantees that this hybrid is indistinguishable from the previous one.

Hyb_2 In this hybrid, we substitute all ciphertexts encrypted by honest parties in the set $\mathcal{U}_2 \setminus C$ and sent to other honest parties with encryptions of 0 (padded to the appropriate length) instead of shares of s_u^{SK} and b_u. However, the honest clients in that set continue to respond with the correct shares of s_u^{SK} and b_u in Round **Unmasking**. Since only the contents of the ciphertexts have changed, IND-CPA security of the encryption scheme guarantees that this hybrid is indistinguishable from the previous one.

Hyb_3 Define:

$$\mathcal{U}^* = \begin{cases} \mathcal{U}_2 \setminus C & \text{if } z = \bot \\ \mathcal{U}_2 \setminus \mathcal{U}_3 \setminus C & \text{otherwise.} \end{cases}$$

This hybrid is distributed exactly as the previous one, but here we substitute all shares of b_u generated by parties $u \in \mathcal{U}^*$ and given to the corrupted parties in Round **ShareKeys** with shares of 0 (using a different sharing of 0 for every $u \in \mathcal{U}^*$). Note that, in this hybrid and the previous one, the adversary does not receive any additional shares of b_u for users u in the set $\mathcal{U}^*$ in Round **Unmasking**, either because the honest clients do not reveal shares of b_u for such u, or because all honest clients abort (when $|\mathcal{U}_3| < t$, which happens exactly when $z = \bot$). Thus, M_C's joint view contains only $|C| < t$ shares of each b_u. The properties of Shamir's secret sharing thus guarantee that the distribution of any $|C|$ shares of 0 is identical to the distribution of an equivalent number of shares of any given secret b_u, making this hybrid identically distributed to the previous one.

Hyb_4 In this hybrid, for all parties $u \in \mathcal{U}^*$, instead of computing $\boldsymbol{p}_u \leftarrow \mathbf{PRG}(b_u)$, we set it to be a uniformly random vector (of the appropriate size).

Note that, in the previous hybrid, since b_u is chosen uniformly at random and its shares given to the adversary are substituted with shares of 0, the output of the random variable does not depend on the seed of the **PRG** except through the **PRG**'s output. Therefore, the only change in this hybrid boils down to substituting the output of a **PRG** (on a randomly generated seed otherwise independent from the joint view of parties in C) with a uniformly random value. Therefore, leveraging the security of the **PRG**, we can argue that this hybrid is indistinguishable from the previous one.

Hyb$_5$ For all parties $u \in \mathcal{U}^*$, in Round **MaskedInputCollection**, instead of sending:

$$\boldsymbol{y}_u \leftarrow \boldsymbol{x}_u + \boldsymbol{p}_u + \sum_{v \in \mathcal{U}_2} \boldsymbol{p}_{u,v}$$

we send:

$$\boldsymbol{y}_u \leftarrow \boldsymbol{p}_u + \sum_{v \in \mathcal{U}_2} \boldsymbol{p}_{u,v}$$

Since $\boldsymbol{p}_u$ was changed in the previous hybrid to be uniformly random and independent of any other values, $\boldsymbol{x}_u + \boldsymbol{p}_u$ is also uniformly random, and so this hybrid and the previous hybrid are identically distributed. Further, this hybrid and all subsequent hybrids do not depend on the values $\boldsymbol{x}_u$ for $u \in \mathcal{U}^*$.

Note: If $z = \perp$, then we can ignore the further hybrids, and let SIM be as described in Hyb$_5$, since SIM can already simulate REAL without knowing $\boldsymbol{x}_u$ for any $u \notin C$. Therefore in the following hybrids we assume $z \neq \perp$.

Hyb$_6$ This random variable is distributed exactly as the previous one, but here we substitute all shares of s_u^{SK} generated by parties $u \in \mathcal{U}_3 \setminus C$ and given to the corrupted parties in Round **ShareKeys** with shares of 0 (using a different sharing of 0 for every $u \in \mathcal{U}_3 \setminus C$). Following an analogous argument to that for Hyb$_3$, the properties of Shamir's secret sharing guarantee that this hybrid is identically distributed to the previous one.

Hyb$_7$ We fix a specific user $u' \in \mathcal{U}_3 \setminus C$. For this user, and each other user $u \in \mathcal{U}_3 \setminus C$, in order to compute the value $\boldsymbol{y}_u$ sent to the server, we substitute the joint noise key (which would be computed by u' and u as $s_{u',u} = s_{u,u'} \leftarrow$ **KA.agree**$(s_{u'}^{SK}, s_u^{PK})$) with a uniformly random value (which will used by both parties as a **PRG** seed).

In more detail, for each user $u \in \mathcal{U}_3 \setminus C \setminus \{u'\}$, a value $s'_{u',u}$ is sampled uniformly at random and, instead of sending

$$\boldsymbol{y}_u \leftarrow \boldsymbol{x}_u + \boldsymbol{p}_u + \sum_{v \in \mathcal{U}_2} \boldsymbol{p}_{u,v}$$

SIM sends

$$\boldsymbol{y}'_u \leftarrow \boldsymbol{x}_u + \boldsymbol{p}_u + \sum_{v \in \mathcal{U}_2 \setminus \{u'\}} \boldsymbol{p}_{u,v} + \Delta_{u,u'} \cdot \mathbf{PRG}(s'_{u',u})$$

and accordingly

$$\boldsymbol{y}'_{u'} \leftarrow \boldsymbol{x}_{u'} + \boldsymbol{p}_{u'} + \sum_{v \in \mathcal{U}_2} \Delta_{u',v} \cdot \mathbf{PRG}(s'_{u',v})$$

where $\Delta_{u,v} = 1$ when $u > v$ and $\Delta_{u,v} = -1$ when $u < v$.

It is easy to see that the Decisional Diffie-Hellman Assumption (Definition 3.1) guarantees that this hybrid is indistinguishable from the previous one[7].

[7]It is important to note here that, in the previous hybrids, we removed all shares of s_u^{SK} for $u \in \mathcal{U}_3 \setminus C$ from the joint view of parties in C. Without doing so, we could not reduce to the security of DH Key Agreement.

Hyb$_8$ In this hybrid, for the same party u' chosen in the previous hybrid and all other parties $v \in \mathcal{U}_3 \setminus C$, instead of computing $\boldsymbol{p}_{u',v} \leftarrow \Delta_{u',v} \cdot \mathbf{PRG}(s'_{u',v})$, we compute it using fresh randomness $r_{u',v}$ (of the appropriate size) as $\boldsymbol{p}_{u',v} \leftarrow \Delta_{u',v} \cdot r_{u',v}$.

Note that, in the previous hybrid, since $s'_{u',v}$ is chosen uniformly at random (and independently from the Diffie-Hellman keys), the output of the random variable does not depend on the seed of the **PRG** except through the **PRG**'s output. Therefore, the only change in this hybrid boils down to substituting the output of a **PRG** (on an randomly generated seed otherwise independent from the joint view of parties in C) with a uniformly random value. Therefore, leveraging the security of the **PRG**, we can argue that this hybrid is indistinguishable from the previous one.

Hyb$_9$ In this hybrid, for all users $u \in \mathcal{U}_3 \setminus C$, in round **MaskedInputCollection** instead of sending:

$$\begin{aligned}\boldsymbol{y}_u \leftarrow \boldsymbol{x}_u + \boldsymbol{p}_u + \sum_{v \in \mathcal{U}_2} \boldsymbol{p}_{u,v} \\ = \boldsymbol{x}_u + \boldsymbol{p}_u + \sum_{v \in \mathcal{U}_3 \setminus C} \boldsymbol{p}_{u,v} + \sum_{v \in \mathcal{U}_2 \setminus \mathcal{U}_3 \setminus C} \boldsymbol{p}_{u,v}\end{aligned}$$

we send:

$$\boldsymbol{y}_u \leftarrow \boldsymbol{w}_u + \boldsymbol{p}_u + \sum_{v \in \mathcal{U}_2 \setminus \mathcal{U}_3 \setminus C} \boldsymbol{p}_{u,v}$$

Where $\{\boldsymbol{w}_u\}_{u \in \mathcal{U}_3 \setminus C}$ are uniformly random, subject to $\sum_{\mathcal{U}_3 \setminus C} \boldsymbol{w}_u = \sum_{\mathcal{U}_3 \setminus C} \boldsymbol{x}_u = \boldsymbol{z}$. Invoking Lemma 6.1 with $n = |\mathcal{U}_3 \setminus C|$, we have that this hybrid is identically distributed to the previous one. Moreover, note that to sample from the random variable described by this hybrid, knowledge of the individual $\boldsymbol{x}_u$ for $u \in \mathcal{U}_3 \setminus C$ is not needed, and their sum $\boldsymbol{z}$ is sufficient.

We can thus define a PPT simulator SIM that samples from the distribution described in the last hybrid. The argument above proves that the output of the simulator is computationally indistinguishable from the output of REAL, completing the proof. □

6.2 Privacy against Active Adversaries

In this section we discuss our argument for security against active adversaries (detailed proofs can be found the in full version of this paper). By active adversaries, we mean parties (clients or the server) that deviate from the protocol, sending incorrect and/or arbitrarily chosen messages to honest users, aborting, omitting messages, and sharing their entire view of the protocol with each other, and also with the server (if the server is also an active adversary).

We note that we only show *input privacy* for honest users: it is much harder to additionally guarantee *correctness* and *availability* for the protocol when some users are actively adversarial. Such users can distort the output of the protocol by setting their input values $\boldsymbol{x}_u$ to be out of range[8], by sending inconsistent Shamir shares to other users in Round **ShareKeys**, or by reporting incorrect shares

[8]Typically, each element of $\boldsymbol{x}_u$ is expected to be from a range $[0, R_U) \subset [0, R)$, such that the sum of all $\boldsymbol{x}_u$ is in $[0, R)$. However, an actively adversarial user could choose $\boldsymbol{x}_u$ outside the expected range, i.e. on $[R_U, R)$, allowing the adversarial user disproportionate impact on protocol's result, thus undermining correctness.

to the server in Round **Unmasking**. Making such deviations efficient to detect and possibly recover from is left to future work.

We note some key differences between the argument for honest-but-curious security, and the argument for privacy against active adversaries.

The first key difference is that, for the proof against active adversaries, we assume that there exists a public-key infrastructure (PKI), which guarantees to users that messages they receive came from other users (and not the server). Without this assumption, the server can perform a Sybil attack on the users in Round **ShareKeys**, by simulating for a specific user u all other users v in the protocol and thus receiving all u's key shares and recovering that users' input. Alternatively, we can require the server to *act honestly in its first message* (in Round **ShareKeys**). Specifically, the server must honestly forward the Diffie-Hellman public keys it receives to all other users, allowing them to set up pairwise private and authenticated channels amongst themselves.

However, if we assume a PKI, then we observe that the server's power in the remainder of the protocol is reduced to lying to users about which other users have dropped out: since all user-to-user messages (sent in round **ShareKeys**) are authenticated through an authenticated encryption scheme, the server cannot add, modify or substitute messages, but rather, can only fail to deliver them. Note, importantly, that the server can try to give a different view to each user of which other users have dropped out of the protocol. In the worst case, this could allow the server to learn a different set of shares from each user in Round **Unmasking**, allowing it to potentially reconstruct more secrets than it should be allowed to. The **ConsistencyCheck** round is included in the protocol to deal with this issue. The inclusion of the **ConsistencyCheck** round is the second key difference with the honest-but-curious proof.

The final key difference is that we need the proof to be in the random oracle (RO) model. To see why, notice that honestly acting users are essentially "commited" to their secrets and input by the end of the **MaskedInputCollection** round. However, the server can adaptively choose which users drop after the **MaskedInput-Collection** round. This causes problems for a simulation proof, because the simulator doesn't know honest users' real inputs, and must use dummy information in the earlier rounds, thus "committing" itself to wrong values that are potentially easily detectable. The random oracle adds a trapdoor for the simulator to equivocate, so that even if it commits to dummy values in early rounds, it can reprogram the random oracle to make the dummy values indistinguishable from honest users' values. A full proof of security appears in the full version of this paper.

6.3 Interpretation of Results

We summarize our system for the different security models we consider in Figure 5.

6.3.1 Security against only clients. In each of Theorems 6.2 and 6.4, we see that the joint view of any subset of clients, honest or adversarial, can be simulated given *no information* about the values of the remaining clients. This means, no matter how we set our t parameter, clients on their own learn nothing about other clients.

Theorem 6.4 (Privacy against actively adversarial users, with honest server). *There exists a PPT simulator* SIM *such that*

Threat model	Minimum threshold	Minimum inputs in sum
Client-only adversary	1	t
Server-only adversary	$\lfloor \frac{n}{2} \rfloor + 1$	t
Clients-Server collusion	$\lfloor \frac{2n}{3} \rfloor + 1$	$t - n_C$

Figure 5: Parameterization for different threat models. "Minimum threshold" denotes the minimum value of t required for security in the given threat model. "Minimum inputs in the sum" denotes a lower bound on the number of users' values that are included in the sum learned by the server. n denotes the total number of users, while n_C is the number of corrupt users.

for all PPT adversaries M_C, all $k, t, \mathcal{U}, \boldsymbol{x}_{\mathcal{U}\setminus C}, C \subseteq \mathcal{U}$, the output of SIM *is perfectly indistinguishable from the output of* $\mathsf{REAL}_C^{\mathcal{U},t,k}$:

$$\mathsf{REAL}_C^{\mathcal{U},t,k}(M_C, \boldsymbol{x}_{\mathcal{U}\setminus C}) \equiv \mathsf{SIM}_C^{\mathcal{U},t,k}(M_C)$$

Proof. See full version of this paper. □

6.3.2 Security against only the server. From Theorems 6.3 and 6.5, we see that if we set $n_C = 0$, that is, there are no clients who cheat or collaborate with the server, then setting $t \geq \lfloor \frac{n}{2} \rfloor + 1$ guarantees that the sum learned by the server contains the values of at least $t > \frac{n}{2}$ clients, and the protocol can deal with up to $\lceil \frac{n}{2} \rceil - 1$ dropouts.

Theorem 6.5 (Privacy against active adversaries, including the server). *There exists a PPT simulator* SIM *such that for all $k, t, \mathcal{U}, C \subseteq \mathcal{U} \cup \{S\}$ and $\boldsymbol{x}_{\mathcal{U}\setminus C}$, letting $n = |\mathcal{U}|$ and $n_C = |C \cap \mathcal{U}|$, if $2t > n + n_C$, then the output of* SIM *is computationally indistinguishable from the output of* $\mathsf{REAL}^{\mathcal{U},t,k}$:

$$\mathsf{REAL}_C^{\mathcal{U},t,k}(M_C, \boldsymbol{x}_{\mathcal{U}\setminus C}) \approx_c \mathsf{SIM}_C^{\mathcal{U},t,k,\mathsf{Ideal}^{\delta}_{\{\boldsymbol{x}_u\}_{u\in\mathcal{U}\setminus C}}}(M_C)$$

where $\delta = t - n_C$.

Proof. See full version of this paper. □

6.3.3 Security against a server colluding with clients. From Theorems 6.3 and 6.5, we see that we can allow a server (honest or adversarial) to collaborate with up to $n_C = \lceil \frac{n}{3} \rceil - 1$ users (honest or adversarial), if we set $t \geq \lfloor \frac{2n}{3} \rfloor + 1$, at the same time guaranteeing that the sum learned by the server contains the values of at least $\frac{n}{3}$ clients. Additionally, the protocol is robust to up to $\lceil \frac{n}{3} \rceil - 1$ users dropping out.

For all the results above, we reiterate that if we want security against servers that are allowed to actively deviate from the protocol (whether or not they collaborate with clients), we must use include the protocol features highlighted in Figure 4.

7 EVALUATION

We summarize the protocol's performance in Table 3. All calculations below assume a single server and n users, where each user holds a data vector of size m. We evaluate the honest-but-curious version of the protocol, and ignore the cost of the PKI, all signatures, and Round **ConsistencyCheck**. We note that including their cost does not change any of the asymptotics, and only slightly increases the computation and communication costs.

7.1 Performance Analysis of Client

Computation cost: $O(n^2 + mn)$**.** Each user u's computation cost can be broken up as (1) Performing the $2n$ key agreements, which take $O(n)$ time, (2) Creating t-out-of-n Shamir secret shares of s_u^{SK} and b_u, which is $O(n^2)$ and (3) Generating values $\boldsymbol{p}_u$ and $\boldsymbol{p}_{u,v}$ for every other user v for each entry in the input vector by stretching one **PRG** seed each, which takes $O(mn)$ time in total. Overall, each user's computation is $O(n^2 + mn)$.

Communication cost: $O(n + m)$**.** The communication costs of each user can be broken up into 4 parts: (1) Exchanging keys with each other user by sending 2 and receiving $2(n-1)$ public keys, (2) Sending $2(n-1)$ and receiving $2(n-1)$ encrypted secret shares, (3) Sending a masked data vector of size $m\lceil\log_2 R\rceil$ to the server, and (4) Sending the server n secret shares, for an overall communication cost of $2na_K + (5n-4)a_S + m\lceil\log_2 R\rceil$, where a_K and a_S are the number of bits in a key exchange public key and the number of bits in a secret share, respectively. Overall, the user's communication complexity is $O(n + m)$. Assuming inputs for each user are on the same range $[0, R_U - 1]$, we require $R = n(R_U - 1) + 1$ to avoid overflow. A user could transmit its raw data using $m\lceil\log_2 R_U\rceil$ bits. Taking $a_K = a_S = 256$ bits implies a communication expansion factor of $\frac{256(7n-4)+m\lceil\log_2 R\rceil}{m\lceil\log_2 R_U\rceil}$. For $R_U = 2^{16}$ (i.e. 16-bit input values), $m = 2^{20}$ elements, and $n = 2^{10}$ users, the expansion factor is $1.73\times$; for $n = 2^{14}$ users, it is $3.62\times$. For $m = 2^{24}$ elements and $n = 2^{14}$ users, the expansion factor is $1.98\times$.

Storage cost: $O(n + m)$. The user must store the keys and secret-shares sent by each other user, which are $O(n)$ in total, and the data vector (which it can mask in-place), which has size $O(m)$.

7.2 Performance Analysis of Server

Computation cost: $O(mn^2)$**.** The server's computation cost can be broken down as (1) Reconstructing n t-out-of-n Shamir secrets (one for each user), which takes total time $O(n^2)$, and (2) generating and removing the appropriate $\boldsymbol{p}_{u,v}$ and $\boldsymbol{p}_u$ values from the sum of the $\boldsymbol{y}_u$ values received, which takes time $O(mn^2)$ in the worst case.

We note that reconstructing n secrets in the Shamir scheme takes $O(n^3)$ time in the general case: each secret reconstruction $\textbf{SS.recon}(\{(u, s_u)\}_{u\in\mathcal{U}'}, t) \rightarrow s$ amounts to interpolating a polynomial L over the points encoded by the shares and then evaluating at 0, which can be accomplished via Lagrange polynomials:

$$s = L(0) = \sum_{u\in\mathcal{U}'} s_u \prod_{v\in\mathcal{U}'\setminus\{u\}} \frac{v}{v-u} \pmod{p}$$

Each reconstruction requires $O(n^2)$ computation and we must perform n reconstructions, implying $O(n^3)$ total time. However, in our setting, we can perform all of the reconstructions in $O(n^2)$ time by observing that all of our secrets will be reconstructed from identically-indexed sets of secret shares – that is, $\mathcal{U}'$ is fixed across all secrets, because in round **Unmasking**, each user that is still alive sends a share of *every* secret that needs to be reconstructed. Therefore, we can precompute the Lagrange basis polynomials

$$\ell_u = \prod_{v\in\mathcal{U}'\setminus\{u\}} \frac{v}{v-u} \pmod{p}$$

in $O(n^2)$ time and $O(n)$ space, then reconstruct each of n secrets in $O(n)$ time as $L(0) = \sum_{u\in\mathcal{U}'} s_u \ell_u \pmod{p}$ resulting in a total computational cost of $O(n^2)$ to reconstruct all the secrets.

We also note that the $O(mn^2)$ term can be broken into $O(m(n-d) + md(n-d))$, where d is the number of users that dropped from the protocol. In practice, d may be significantly smaller than n, which would also reduce the server's computation cost.

Communication cost: $O(n^2 + mn)$**.** The server's communication cost is dominated by its mediation of all pairwise communications between users, which is $O(n^2)$, and also for receiving masked data vectors from each user, which is $O(mn)$ in total.

Storage cost: $O(n^2 + m)$**.** The server must store t shares for each user, which is $O(n^2)$ in total, along with an m-element buffer in which to maintain a running sum of y_u as they arrive.

7.3 Prototype Performance

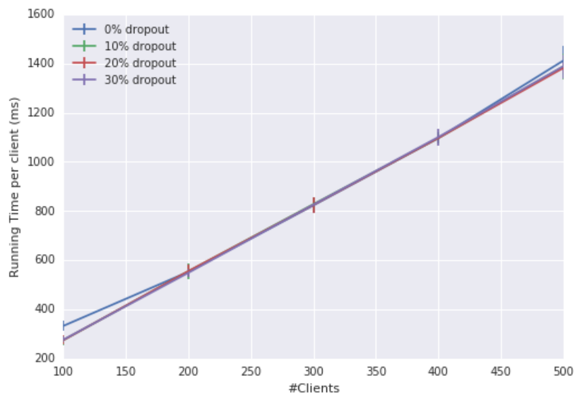
In order to measure performance, we implemented a prototype in Java, with the following cryptographic primitives:

- For Key Agreement, we used Elliptic-Curve Diffie-Hellman over the NIST P-256 curve, composed with a SHA-256 hash.
- For Secret Sharing, we used standard t-out-of-n Shamir Sharing.
- For Authenticated Encryption, we used AES-GCM with 128-bit keys.
- For the Pseudorandom Number Generator, we used AES in counter mode.

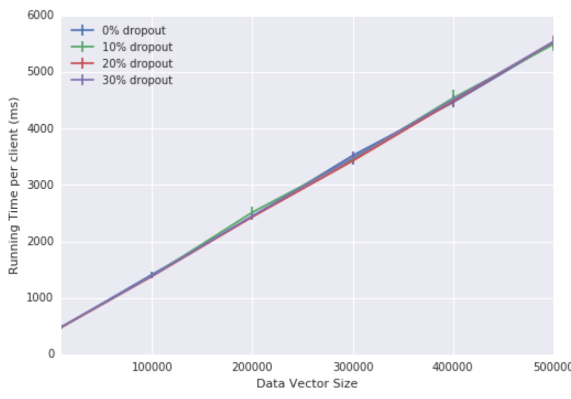
We assume an honest-but-curious setting, and thus omitted the portions of Figure 4 special to active clients from our simulations. We note that these omissions would not change the overall shape of our results in practice, since, as we discuss below, the bulk of the costs involve masking, storing and sending the large data vector.

Additionally, we assume that when clients drop out of the protocol, that they drop after sending their shares to all other clients, but before sending their masked input to the server. This is essentially the "worst case" dropout, since all other clients have already incorporated the dropped clients' masks, and the server must perform an expensive recovery computation to remove them. We also assumed that client's data vectors had entries such that at most 3 bytes are required to store the sum of up to all clients' values without overflow.

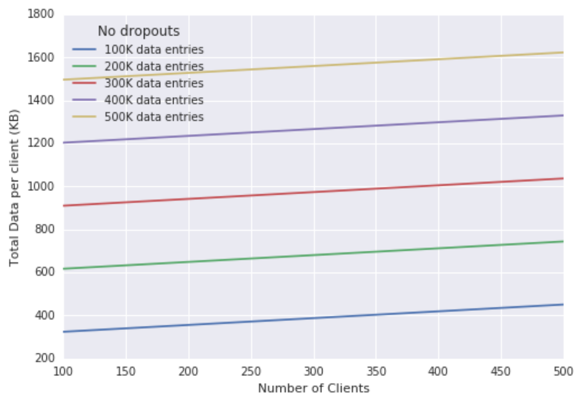
We ran single-threaded simulations on a Linux workstation with an Intel Xeon CPU E5-1650 v3 (3.50 GHz), with 32 GB of RAM. Wall-clock running times and communication costs for clients are plotted in Figure 6. Wall clock running times for the server are plotted in Figure 7, with different lines representing different percentages of clients dropping out. Figure 8 shows wall-clock times per round for both the client and the server. We omit data transfer plots for the server, as they are essentially identical to those for the client, except higher by a factor of n. This is because the incoming data of the server is exactly the total outgoing data of all clients, and vice versa. We also do not plot bandwidth numbers for different



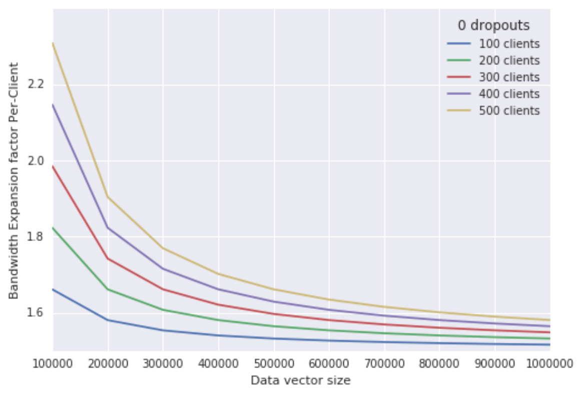
(a) Wall-clock running time per client, as the number of clients increases. The data vector size is fixed to 100K entries.



(b) Wall-clock running time per client, as the size of the data vector increases. The number of clients is fixed to 500.



(c) Total data transfer per client, as the number of clients increases. Different lines show different data vector sizes. Assumes no dropouts.



(d) Total data expansion factor per client, as compared to sending the raw data vector to the server. Different lines represent different values of n. Assumes no dropouts.

Figure 6: Client Running Time and Data Transfer Costs. All wall-clock running times are for a single-threaded client implemented in Java, and ignore communication latency. Plotted points represent averages over 10 end-to-end iterations, and error bars represent 95% confidence intervals. (Error bars are omitted where measured standard deviation was less than 1%).

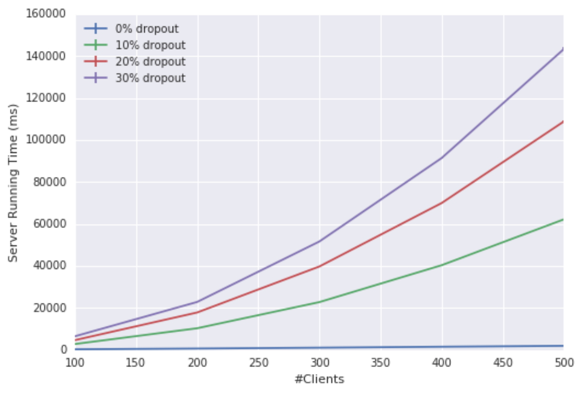
numbers of dropouts, as the number of dropouts does not have a significant impact on this metric.

In our simulations, for both the client and the server, almost all of the computation cost comes from expanding the various PRG seeds to mask the data vector. Compared to this, the computational costs of key agreement, secret sharing and reconstruction, and encrypting and decrypting messages between clients, are essentially negligible, especially for large choices of n and data vector size. This suggests that using an optimized PRG implementation would yield a significant running-time improvement over our prototype.

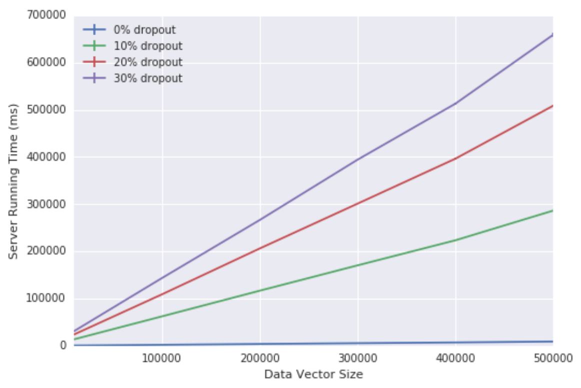
As seen in Figures 6a and 6b, the running time of each client increases linearly with both the total number of clients and the number of data vector entries, but does not change significantly when more clients drop out. In Figure 6c, the communication expansion factor for each client increases as the total number of clients increases, but this increase is relatively small compared to the impact of increasing the size of the data vector. This is also reflected in Figure 6d, where the communication expansion factor for each client increases as the total number of clients increases, but falls quickly as the size of the data vector increases. This shows that the cost of messages between clients amortizes well as the size of the data vector increases.

In the case of the server, Figures 7a and 7b show that the running time of the server increases significantly with the fraction of dropouts. This is because, for each dropped client u, the server must remove that client's pairwise masks $\boldsymbol{p}_{u,v}$ from each other surviving client v, which requires $(n - d)$ PRG expansions, where d is the number of dropped users. In contrast, each undropped user entails only a single PRG expansion, to remove its self-mask. The high cost of dealing with dropped users is also reflected in the server running times in Figure 8.

In Figure 9, we show the results of running the protocol over a Wide Area Network (WAN). The server and clients were run on geographically seperated datacenters, with contention for CPU and network. We give the standard deviations of the running times, which reflects this contention, and occasional machine failures (<1% of clients per execution). We observe that the clients have a somewhat shorter runtime than the server: this is because the



(a) Wall-clock running time for the server, as the number of clients increases. The data vector size is fixed to 100K entries.



(b) Wall-clock running time for the server, as the size of the data vector increases. The number of clients is fixed to 500.

Figure 7: Server Running Time and Data Transfer Costs. All wall-clock running times are for a single-threaded server implemented in Java, and ignore communication latency. Plotted points represent averages over 10 end-to-end iterations. Error bars are omitted where measured standard deviations are less than 1%.

	Num. Clients	Dropouts	AdvertiseKeys	ShareKeys	MaskedInputColl.	Unmasking	Total
Client	500	0%	1 ms	154 ms	694 ms	1 ms	849 ms
Server	500	0%	1 ms	26 ms	723 ms	1268 ms	2018 ms
Server	500	10%	1 ms	29 ms	623 ms	61586 ms	62239 ms
Server	500	30%	1 ms	28 ms	514 ms	142847 ms	143389 ms
Client	1000	0%	1 ms	336 ms	1357 ms	5 ms	1699 ms
Server	1000	0%	6 ms	148 ms	1481 ms	3253 ms	4887 ms
Server	1000	10%	6 ms	143 ms	1406 ms	179320 ms	180875 ms
Server	1000	30%	8 ms	143 ms	1169 ms	412446 ms	413767 ms

Figure 8: CPU wall clock times per round. All wall-clock running times are for a single-threaded servers and clients implemented in Java, and ignore communication latency. Each entry represents the average over 10 iterations. The data vector size is fixed to 100K entries with 24 bit entries.

Num. Clients	Total Runtime Per-Client	StdDev	Server Total Runtime	StdDev	Total Communication Per Client
500	13159 ms	6443 ms	14670 ms	6574 ms	0.95 MB
1000	23497 ms	6271 ms	27855 ms	6874 ms	1.15 MB

Figure 9: End-to-End running time for the protocol, executed over a wide-area-network. All running times are for a single-threaded servers and clients running in geographically separated datacenters, and include computation time, network latency, and time spent waiting for other participants. Each entry represents the average over 15 iterations, with iterations more than 3 standard deviations from the mean discarded. The data vector size is fixed to 100K entries with 62 bits per entry, and there are no induced dropouts (beyond <1% that occurred naturally).

server has to run the additional (expensive) unmasking step after all clients have completed.

8 DISCUSSION AND FUTURE WORK

Identifying and Recovering from Abuse The security proof in Theorem 6.5 guarantees that when users' inputs are learned by the server, they are always in aggregate with the values of other users. However, we do not protect against actively adversarial clients that try to prevent the server from learning *any* sum at all. For example, an attacker-controlled client could send malformed messages to other clients, causing enough of them to abort that the protocol fails before the server can compute its output. Ideally, we would like such abuse by corrupt clients to be efficiently identifiable, and the protocol to gracefully recover from it. However, the problem of assigning blame for abuse is subtle, and often adds several rounds to protocols. We leave this problem to future work.

Enforcing Well-formed Inputs Our protocol also does not verify that users' inputs are well-formed or within any particular bounds, so actively adversarial users could send arbitrary values of their choice, that can cause the output learned by the server to also be ill-formed. For our specific machine learning application, we will be able to detect obviously malformed outputs and can simply run the protocol again with a different set of clients. However, an adversarial client may be able to supply "slightly" malformed input values that are hard to detect, such as double its real values.

A possible solution is to use zero-knowledge proofs that the client inputs are in the correct range. Unfortunately, even using the best-known garbled circuit techniques [34], even one such proof would be more costly than the entire protocol. We leave the problem of guaranteeing well-formed inputs from the clients to future work.

Reducing Communication Further In the protocol we describe, all clients exchange pairwise masks with all other clients. However, it may be sufficient to have the clients exchange masks with only a subset of other clients, as long as these subsets of clients do not form disjoint clusters. In fact, previous works (notably, Ács et al. [3]) use this approach already. However, in our setting, this requires extra care because the server facilitates the communication among clients, and an actively adversarial server can choose dropouts based on its knowledge of which pairs of clients exchanged masks with each other. We leave this improvement to future work.

9 RELATED WORK

As noted in Section 2, we emphasize that our focus is on mobile devices, where bandwidth is expensive, and dropouts are common, and in our setting there is a single service provider. Consequently, our main goal is to minimize communication while guaranteeing robustness to dropouts. Computational cost is an important, but secondary, concern. These constraints will motivate our discussion of, and comparison with, existing works.

Works based on Multiple non-Colluding Servers: To overcome the constraints of client devices, some previous work has suggested that clients distribute their trust across multiple non-colluding servers, and this has been deployed in real-world applications [10]. The recently presented Prio system of Gibbs and Boneh [16] is, from the perspective of the client devices, non-interactive, and the computation among the servers is very lightweight. Prio also allows client inputs to be validated, something our current system cannot do, by relying on multiple servers.

Araki et al. recently presented a generic three-party computation protocol that achieves very high throughput [5]. This protocol could also be used in a setting where non-colluding servers are available, with the clients sending shares to each server that will be combined online.

Works based on Generic Secure Multiparty Computation: As noted in Section 1, there is a long line of work showing how multiple parties can securely compute any function using generic secure MPC [8, 18, 27, 40, 41]. These works generally fall into two categories: those based on Yao's garbled circuits, and those based on homomorphic encryption or secret sharing. The protocols based on Yao's garbled circuits are better suited to 2- or 3-party secure computation and do not directly extend to hundreds of users.

MPC protocols based on secret sharing, however, can extend to hundreds of users. In addition, these protocols have become relatively computationally efficient, and can be made robust against dropouts. Boyle et al. studied generic MPC at such scale, relying on a particular ORAM construction to help localize the computation and avoid broadcasts [11]. Some works, notably [12], optimize these generic techniques for the specific task of secure summation, and have publicly available implementations.

However, the weakness of generic MPC protocols based on secret-sharing is communication cost. In all such protocols, each user sends a secret-share of its entire data vector to some subset of the other users. To guarantee robustness, this subset of users must be relatively large: robustness is essentially proportional to the size of the subset. Additionally, each secret share is as long as the size of the entire data vector. In our setting the constraints on total communication make these approaches unworkable.

Works based on Dining Cryptographers Networks: Dining cryptographers networks, or DC-nets, are a type of communication network which provide anonymity by using pairwise blinding of inputs [14, 28], similarly to our secure aggregation protocol. The basic version of DC-nets, in which a single participant at a time sends an anonymous message, can be viewed as the restricted case of secure aggregation in which all users except for one have an input of 0.

Recent research has examined increasing the efficiency of DC-nets protocols and allowing them to operate in the presence of active adversaries [17]. But previous DC-nets constructions share the flaw that, if even one user aborts the protocol before sending its message, the protocol must be restarted from scratch, which can be very expensive [36].

Works based on Pairwise Additive Masking: Pairwise blinding using additive stream ciphers has been explored in previous work [3, 24, 31, 33], presenting different approaches to dealing with client failures.

The work of Ács and Castelluccia [3], and the modification suggested by [31], are the most closely related to our scheme, and have an explicit recovery round to deal with failures. Their protocols operate very similarly to ours: pairs of clients use Diffie-Hellman key exchange to agree on pairwise masks, and send the server their data vectors, summed with each of their pairwise masks and also a "self-mask". In the recovery step, the server tells the remaining clients which other clients dropped out, and each remaining client responds with the sum of their (uncancelled) pairwise masks with the dropped users, added to their "self-mask". The server subtracts these "recovery" values from the masked vectors received earlier, and correctly learns the sum of the undropped users' data.

However, their recovery phase is brittle: if additional users drop out during the recovery phase, the protocol cannot continue. Simply repeating the recovery round is not sufficient, since this has the potential to leak the "self-masks" of the surviving users, which in turn can leak their data vectors. Moreover, since the *entire* sum of the masks is sent, this round requires almost as much communication as the rest of the protocol, making further client failures during this step likely.

Schemes based on (Threshold) Homomorphic Encryption Schemes based on threshold additively-homomorphic cryptosystems (e.g. the Paillier cryptosystem [38, 47]) can handle client dropouts, but are either computationally expensive or require additional trust assumptions. For example, Paillier-based schemes require an expensive-to-generate set of threshold decryption keys, that must either be generated and distributed by a trusted third party or generated online with an expensive protocol. Similarly the pairing-based scheme of Leontiadis et al. [39] calls for a trusted dealer to set up the keys.

The schemes of Shi et al. [49] and Chan et al. [13] use an approach similar to ours, but in the exponent in some group (the latter scheme extends the former to provide robustness against client dropouts). They also consider the need for differential privacy and give a rigorous analysis of distributed noise generation. Unfortunately, the size of the group elements is too large for our setting, and their schemes also call for a trusted dealer.

Halevi, Lindell and Pinkas [32] present a protocol that uses homomorphic encryption to securely compute the sum in just one round of interaction between the server and each of the clients (assuming a PKI is already in place). Their protocol has the advantage that all parties do not need to be online simultaneously for the protocol to execute. However, the protocol also requires the communication to be carried out sequentially between the clients and the server. More importantly for our setting, their protocol does not deal with clients dropping out: all clients included in the protocol must respond before the server can learn the decrypted sum.

10 CONCLUSION

We have presented a practical protocol for securely aggregating data while ensuring that clients' inputs are only learned by the server in aggregate. The overhead of our protocol is very low, and it can tolerate large numbers of failing devices, making it ideal for mobile applications. We require only one service provider, which simplifies deployment. Our protocol has immediate applications to real-world federated learning, and we expect to deploy a full application in the near future.

REFERENCES

[1] Martín Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 308–318.

[2] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. 2001. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In *Cryptographers' Track at the RSA Conference*. Springer, 143–158.

[3] Gergely Ács and Claude Castelluccia. 2011. I have a DREAM! (DiffeRentially privatE smArt Metering). In *International Workshop on Information Hiding*. Springer, 118–132.

[4] Stephen Advokat. 1987. Publication Of Bork's Video Rentals Raises Privacy Issue. *Chicago Tribune* (1987).

[5] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA, 805–817. https://doi.org/10.1145/2976749.2978331

[6] Michael Barbaro, Tom Zeller, and Saul Hansell. 2006. A face is exposed for AOL searcher no. 4417749. *New York Times* 9, 2008 (2006).

[7] Mihir Bellare and Chanathip Namprempre. 2000. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 531–545.

[8] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. 1988. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*. ACM, 1–10.

[9] Manuel Blum and Silvio Micali. 1984. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM journal on Computing* 13, 4 (1984), 850–864.

[10] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. 2009. Secure multiparty computation goes live. In *International Conference on Financial Cryptography and Data Security*. Springer, 325–343.

[11] Elette Boyle, Kai-Min Chung, and Rafael Pass. 2015. *Large-Scale Secure Computation: Multi-party Computation for (Parallel) RAM Programs*. Springer Berlin Heidelberg, Berlin, Heidelberg, 742–762. https://doi.org/10.1007/978-3-662-48000-7_36

[12] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. 2010. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. *Network* 1 (2010), 101101.

[13] T-H Hubert Chan, Elaine Shi, and Dawn Song. 2012. Privacy-preserving stream aggregation with fault tolerance. In *International Conference on Financial Cryptography and Data Security*. Springer, 200–214.

[14] David Chaum. 1988. The dining cryptographers problem: unconditional sender and recipient untraceability. *Journal of Cryptology* 1, 1 (1988), 65–75.

[15] Jianmin Chen, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting Distributed Synchronous SGD. In *ICLR Workshop Track*. https://arxiv.org/abs/1604.00981

[16] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, Robust, and Scalable Computation of Aggregate Statistics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 259–282. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs

[17] Henry Corrigan-Gibbs, David Isaac Wolinsky, and Bryan Ford. 2013. Proactively Accountable Anonymous Messaging in Verdict.. In *USENIX Security*. 147–162.

[18] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. 2012. Multiparty computation from somewhat homomorphic encryption. In *Advances in Cryptology–CRYPTO 2012*. Springer, 643–662.

[19] Whitfield Diffie and Martin Hellman. 1976. New directions in cryptography. *IEEE transactions on Information Theory* 22, 6 (1976), 644–654.

[20] John C Duchi, Michael I Jordan, and Martin J Wainwright. 2013. Local privacy and statistical minimax rates. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*. IEEE, 429–438.

[21] Cynthia Dwork. 2006. Differential Privacy, In 33rd International Colloquium on Automata, Languages and Programming, part II (ICALP 2006). 4052, 1–12. https://www.microsoft.com/en-us/research/publication/differential-privacy/

[22] Cynthia Dwork, Krishnaram Kenthapadi, Frank McSherry, Ilya Mironov, and Moni Naor. 2006. Our Data, Ourselves: Privacy Via Distributed Noise Generation.. In *Eurocrypt*, Vol. 4004. Springer, 486–503.

[23] Cynthia Dwork and Aaron Roth. 2014. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science* 9, 3–4 (2014), 211–407.

[24] Tariq Elahi, George Danezis, and Ian Goldberg. 2014. Privex: Private collection of traffic statistics for anonymous communication networks. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1068–1079.

[25] Úlfar Erlingsson, Vasyl Pihur, and Aleksandra Korolova. 2014. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*. ACM, 1054–1067.

[26] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. 2015. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1322–1333.

[27] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. ACM, 218–229.

[28] Philippe Golle and Ari Juels. 2004. Dining cryptographers revisited. In *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 456–473.

[29] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. Deep Learning. (2016). Book in preparation for MIT Press.

[30] Joshua Goodman, Gina Venolia, Keith Steury, and Chauncey Parker. 2002. Language modeling for soft keyboards. In *Proceedings of the 7th international conference on Intelligent user interfaces*. ACM, 194–195.

[31] Slawomir Goryczka and Li Xiong. 2015. A comprehensive comparison of multiparty secure additions with differential privacy. *IEEE Transactions on Dependable and Secure Computing* (2015).

[32] Shai Halevi, Yehuda Lindell, and Benny Pinkas. 2011. Secure computation on the web: Computing without simultaneous interaction. In *Annual Cryptology Conference*. Springer, 132–150.

[33] Rob Jansen and Aaron Johnson. 2016. Safely Measuring Tor. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1553–1567.

[34] Marek Jawurek, Florian Kerschbaum, and Claudio Orlandi. 2013. Zero-knowledge using garbled circuits: how to prove non-algebraic statements efficiently. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 955–966.

[35] Jakub Konečný, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492* (2016).

[36] Young Hyun Kwon. 2015. *Riffle: An efficient communication system with strong anonymity.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[37] Vasileios Lampos, Andrew C Miller, Steve Crossan, and Christian Stefansen. 2015. Advances in nowcasting influenza-like illness rates using search query logs. *Scientific reports* 5 (2015), 12760.

[38] Iraklis Leontiadis, Kaoutar Elkhiyaoui, and Refik Molva. 2014. *Private and Dynamic Time-Series Data Aggregation with Trust Relaxation.* Springer International Publishing, Cham, 305–320. https://doi.org/10.1007/978-3-319-12280-9_20

[39] Iraklis Leontiadis, Kaoutar Elkhiyaoui, Melek Önen, and Refik Molva. 2015. *PUDA – Privacy and Unforgeability for Data Aggregation.* Springer International Publishing, Cham, 3–18. https://doi.org/10.1007/978-3-319-26823-1_1

[40] Yehuda Lindell, Eli Oxman, and Benny Pinkas. 2011. The IPS Compiler: Optimizations, Variants and Concrete Efficiency. *Advances in Cryptology–CRYPTO 2011* (2011), 259–276.

[41] Yehuda Lindell, Benny Pinkas, Nigel P Smart, and Avishay Yanai. 2015. Efficient constant round multi-party computation combining BMR and SPDZ. In *Annual Cryptology Conference.* Springer, 319–338.

[42] Kathryn Elizabeth McCabe. 2012. Just You and Me and Netflix Makes Three: Implications for Allowing Frictionless Sharing of Personally Identifiable Information under the Video Privacy Protection Act. *J. Intell. Prop. L.* 20 (2012), 413.

[43] H Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, et al. 2016. Communication-Efficient Learning of Deep Networks from Decentralized Data. *arXiv preprint arXiv:1602.05629* (2016).

[44] Ilya Mironov, Omkant Pandey, Omer Reingold, and Salil Vadhan. 2009. Computational differential privacy. In *Advances in Cryptology-CRYPTO 2009.* Springer, 126–142.

[45] Arvind Narayanan and Vitaly Shmatikov. 2008. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (sp 2008).* IEEE, 111–125.

[46] John Paparrizos, Ryen W White, and Eric Horvitz. 2016. Screening for pancreatic adenocarcinoma using signals from web search logs: Feasibility study and results. *Journal of Oncology Practice* 12, 8 (2016), 737–744.

[47] Vibhor Rastogi and Suman Nath. 2010. Differentially private aggregation of distributed time-series with transformation and encryption. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* ACM, 735–746.

[48] Adi Shamir. 1979. How to share a secret. *Commun. ACM* 22, 11 (1979), 612–613.

[49] Elaine Shi, HTH Chan, Eleanor Rieffel, Richard Chow, and Dawn Song. 2011. Privacy-preserving aggregation of time-series data. In *Annual Network & Distributed System Security Symposium (NDSS).* Internet Society.

[50] Reza Shokri and Vitaly Shmatikov. 2015. Privacy-preserving deep learning. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security.* ACM, 1310–1321.

[51] Reza Shokri, Marco Stronati, and Vitaly Shmatikov. 2016. Membership Inference Attacks against Machine Learning Models. *arXiv preprint arXiv:1610.05820* (2016).

[52] Latanya Sweeney and Ji Su Yoo. 2015. De-anonymizing South Korean Resident Registration Numbers Shared in Prescription Data. *Technology Science* (2015).

[53] Martin J Wainwright, Michael I Jordan, and John C Duchi. 2012. Privacy aware learning. In *Advances in Neural Information Processing Systems.* 1430–1438.

[54] Andrew C Yao. 1982. Theory and application of trapdoor functions. In *Foundations of Computer Science, 1982. SFCS'08. 23rd Annual Symposium on.* IEEE, 80–91.

A DIFFERENTIAL PRIVACY AND SECURE AGGREGATION

While secure aggregation alone may suffice for some applications, for other applications stronger guarantees may be needed, as indicated by the failures of ad-hoc anonymization techniques [6, 45, 52], and by the demonstrated capability to extract information about individual training data from fully-trained models (which are essentially aggregates) [26, 50, 51].

In such cases, secure aggregation composes well with *differential privacy* [21]. This is particularly advantageous in the *local privacy* setting [20], which offers provable guarantees for the protection of individual training examples [1, 3] even when the data aggregator is not assumed to be trusted [25, 53]. For example, when computing averages, partial averages over subgroups of users may be computed and privacy-preserving noise may be incorporated [22, 31] before revealing the results to the data aggregator. Under some privatization schemes, for a fixed total number of users and for secure aggregation subgroups of size n, the same amount of (computational [44]) differential privacy may be offered to each user while reducing the standard deviation of the effective noise added to the estimated average across all users by a factor of $\sqrt{n}$ relative to providing local differential privacy without secure aggregation. Thus, secure aggregation over just 1024-user subgroups holds the promise of a 32× improvement in differentially private estimate precision. We anticipate that these utility gains will be crucial as methods for differentially private deep learning in the trusted-aggregator setting [1] are adapted to support untrusted aggregators, though a detailed study of the integration of differential privacy, secure aggregation, and deep learning is beyond the scope of the current work.

Suppose that each of U users has a vector $\boldsymbol{x_i}$ with an ℓ^2-norm bounded by $\frac{\Delta}{2}$, such that the ℓ^2-sensitivity of $\sum_i \boldsymbol{x_i}$ is bounded by Δ. For $\epsilon \in (0, 1)$, we can achieve (ϵ, δ)-differential privacy for the sum via the Gaussian mechanism [23], by adding zero-mean multivariate Gaussian noise drawn from $\mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$, where $\sigma = \frac{\Delta}{\epsilon}\sqrt{2\ln(\frac{1.25}{\delta})}$.

In the local privacy setting, users distrust the aggregator, and so before any user submits her value to the aggregator, she adds noise $\boldsymbol{z_i} \sim \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$, achieving (ϵ, δ)-differential privacy for her own data in isolation. Summing contributions at the server yields $\sum_{i=1}^{U} \boldsymbol{x_i} + \sum_{i=1}^{U} \boldsymbol{z_i}$. Observe that the mean of k normally distributed random variables $\boldsymbol{z_i} \sim \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$ is $\bar{\boldsymbol{z}} \sim \mathcal{N}(\boldsymbol{0}, \frac{\sigma^2}{k}\boldsymbol{I})$; it follows that the server can form an unbiased estimator of $\bar{\boldsymbol{x}}$ from the user contributions as

$$\hat{\boldsymbol{x}}_{\boldsymbol{LDP}} = \frac{1}{U}\left(\sum_{i=1}^{U} \boldsymbol{x_i} + \sum_{i=1}^{U} \boldsymbol{z_i}\right) \sim \mathcal{N}(\bar{\boldsymbol{x}}, \frac{\sigma^2}{U}\boldsymbol{I}).$$

Now consider a setting wherein a trusted third party is available that can aggregate and privatize batches of n user inputs; for simplicity, assume that U is a multiple of n. The users deliver raw inputs $\boldsymbol{x_i}$ to the third party, who produces $\frac{U}{n}$ batch-sums, each with (ϵ, δ)-differential privacy for users in the batch, by adding $\boldsymbol{z_j} \sim \mathcal{N}(\boldsymbol{0}, \sigma^2 \boldsymbol{I})$ noise to the batch-sum j before releasing it. Summing the released batch-sums at the server yields $\sum_{i=1}^{U} \boldsymbol{x_i} + \sum_{j=1}^{\frac{U}{n}} \boldsymbol{z_j}$. The server can once again form an unbiased estimator of $\bar{\boldsymbol{x}}$ as

$$\hat{\boldsymbol{x}}_{TTP} = \frac{1}{U}\left(\sum_{i=1}^{U} \boldsymbol{x}_i + \sum_{j=1}^{\frac{U}{n}} z_j\right) \sim \mathcal{N}(\bar{\boldsymbol{x}}, \frac{\sigma^2}{nU}\boldsymbol{I}).$$

Observe that the standard deviation of $\hat{\boldsymbol{x}}_{TTP}$ is a factor of $\frac{1}{\sqrt{n}}$ smaller than that of $\hat{\boldsymbol{x}}_{LDP}$. The secure aggregation protocol can be used in lieu of a trusted third party while retaining these gains by moving to a computational variant of differential privacy [44].

B NEURAL NETWORKS AND FEDERATED LEARNING UPDATES

A neural network represents a function $f(\boldsymbol{x}, \Theta) = \boldsymbol{y}$ mapping an input $\boldsymbol{x}$ to an output $\boldsymbol{y}$, where f is parameterized by a high-dimensional vector $\Theta \in \mathbb{R}^m$. For modeling text message composition, $\boldsymbol{x}$ might encode the words entered so far and $\boldsymbol{y}$ a probability distribution over the next word. A training example is an observed pair $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ and a training set is a collection $\mathcal{D} = \{\langle \boldsymbol{x}_i, \boldsymbol{y}_i \rangle; i = 1, \ldots, m\}$. A loss is defined on a training set $\mathcal{L}_f(\mathcal{D}, \Theta) = \frac{1}{|\mathcal{D}|}\sum_{\langle \boldsymbol{x}_i, \boldsymbol{y}_i \rangle \in \mathcal{D}} \mathcal{L}_f(\boldsymbol{x}_i, \boldsymbol{y}_i, \Theta)$, where $\mathcal{L}_f(\boldsymbol{x}, \boldsymbol{y}, \Theta) = \ell(\boldsymbol{y}, f(\boldsymbol{x}, \Theta))$ for a loss function ℓ, e.g., $\ell(\boldsymbol{y}, \hat{\boldsymbol{y}}) = ||\boldsymbol{y} - \hat{\boldsymbol{y}}||_2$.

Training a neural net consists of finding parameters Θ that achieve small $\mathcal{L}_f(\mathcal{D}, \Theta)$, typically by iterating a variant of a minibatch stochastic gradient descent rule [15, 29]:

$$\Theta^{t+1} \leftarrow \Theta^t - \eta \nabla \mathcal{L}_f(\mathcal{D}^t, \Theta^t)$$

where Θ^t are the parameters after iteration t, $\mathcal{D}^t \subseteq \mathcal{D}$ is a randomly selected subset of the training examples, and η is a learning rate parameter.

In the Federated Learning setting, each user $u \in \mathcal{U}$ holds a private set $\mathcal{D}_u$ of training examples with $\mathcal{D} = \bigcup_{u \in \mathcal{U}} \mathcal{D}_u$. To run stochastic gradient descent, for each update we select a random subset of users $\mathcal{U}^t \subseteq \mathcal{U}$ (in practice we might have say $|\mathcal{U}^t| = 10^4$ while $|\mathcal{U}| = 10^7$) and for each user $u \in \mathcal{U}^t$ we select a random subset of that user's data $\mathcal{D}_u^t \subseteq \mathcal{D}_u$. We then form a (virtual) minibatch $\mathcal{D}^t = \bigcup_{u \in \mathcal{U}^t} \mathcal{D}_u^t$.

The minibatch loss gradient $\nabla \mathcal{L}_f(\mathcal{D}^t, \Theta^t)$ can be rewritten as a weighted average across users:

$$\nabla \mathcal{L}_f(\mathcal{D}^t, \Theta^t) = \frac{1}{|\mathcal{D}^t|} \sum_{u \in \mathcal{U}^t} \boldsymbol{\delta}_u^t$$

where $\boldsymbol{\delta}_u^t = |\mathcal{D}_u^t| \nabla \mathcal{L}_f(\mathcal{D}_u^t, \Theta^t)$. A user can thus share just the concatenated vector $\left[|\mathcal{D}_u^t|\right] \| \boldsymbol{\delta}_u^t$ with the server, from which the server can compute the desired weighted average and a gradient descent step:

$$\Theta^{t+1} \leftarrow \Theta^t - \eta \frac{\sum_{u \in \mathcal{U}^t} \boldsymbol{\delta}_u^t}{\sum_{u \in \mathcal{U}^t} |\mathcal{D}_u^t|}$$

may be taken.

There is evidence that a trained neural network's parameters sometimes allow reconstruction of training examples [1, 26, 50, 51]; it is possible that the parameter updates be subject to similar attacks. For example, if the input x is a one-hot vocabulary-length vector encoding the most recently typed word, common neural network architectures will contain at least one parameter θ_w in Θ for each word w such that $\frac{\partial \mathcal{L}_f}{\partial \theta_w}$ is non-zero only when x encodes w. Thus, the set of recently typed words in $\mathcal{D}_u^t$ would be revealed by inspecting the non-zero entries of $\boldsymbol{\delta}_u^t$.