VerSA: Verifiable Secure Aggregation for Cross-Device Federated Learning

Changhee Hahn, Hodong Kim, Minjae Kim, and Junbeom Hur

Abstract—In privacy-preserving cross-device federated learning, users train a global model on their local data and submit encrypted local models, while an untrusted central server aggregates the encrypted models to obtain an updated global model. Prior work has demonstrated how to verify the correctness of aggregation in such a setting. However, such verification relies on strong assumptions, such as a trusted setup among all users under unreliable network conditions, or it suffers from expensive cryptographic operations, such as bilinear pairing. In this paper, we scrutinize the verification mechanism of prior work and propose a model recovery attack, demonstrating that most local models can be leaked within a reasonable time (e.g., 98% of encrypted local models are recovered within 21 h). Then, we propose VerSA, a verifiable secure aggregation protocol for cross-device federated learning. VerSA does not require any trusted setup for verification between users while minimizing the verification cost by enabling both the central server and users to utilize only a lightweight pseudorandom generator to prove and verify the correctness of model aggregation. We experimentally confirm the efficiency of VerSA under diverse datasets, demonstrating that VerSA is orders of magnitude faster than verification in prior work.

Index Terms—Federated learning, distributed machine learning, security, privacy

1 Introduction

With the explosive growth in data, neural networks have demonstrated promise and usefulness in solving real-world problems in diverse domains, such as computer vision and speech recognition [1], [2], [3]. Due to the ever-increasing need to develop a robust and accurate deep learning model in unpredictable environments, building a joint dataset across organizations and individuals is incredibly important for quality training of the learning model and widespread deployment of its technologies. However, it raises daunting challenges regarding data privacy, especially when the data are shared across multiple users [4].

As an effective countermeasure to handle such a privacy concern, Google proposed *federated learning* (FL) [5], [6], in which a federation of users individually train a shared global model maintained by a central coordinating server. In FL, each user has a training dataset on the local device, and only computes an update to the current model. The central server aggregates the locally updated models from the users to obtain a global model. Although the training data never leave the users' local storage, many studies have shown that diverse privacy attacks are possible on the local model updates to reconstruct the private training data and infer the presence of a specific data item in the training dataset [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17]. Manifold counter approaches to resolving the problem have leveraged various security primitives, such as differential privacy [18], [19], [20], homomorphic encryption [21], [22], post-quantum secure gadgets [23], and multi-party computation [24], [25]. By properly applying these techniques, the central server can perform model aggregation without compromising confidentiality of the local model updates.

Alongside the model privacy, ensuring *correctness* of model aggregation is another important security requirement in FL. Indeed, the central server may have a financial incentive if it performs an exceedingly fast but somewhat incorrect computation for the sake of saving computational resources. One straightforward method to guarantee correctness is applying multiclient verifiable computation [26], [27], [28], which supports the verification of computations over a series of joint inputs from multiple users while preserving privacy of individual input. However, multiclient verifiable computation allows only one user to learn the computational result, which does not fit into FL because all users in FL require this result (i.e., the aggregated model).

Several verifiable FL schemes were recently proposed to address the problem [29], [30], [31], [32]. Among them, VerifyNet [30] was designed specifically for *cross-device* FL. Cross-device FL refers to a distributed machine learning model where numerous users equipped with potentially lightweight devices and unreliable network connections participate in the training phase [33]. In this setting, many pragmatic constraints are considered, such as user dropouts, the joining of never-before-seen users at any time, and the survival of only a fraction of users at the end of each training epoch. These constraints significantly complicate

• *Changhee Hahn is with the Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, Korea. E-mail: chahn@seoultech.ac.kr.*
• *Hodong Kim, Minjae Kim, and Junbeom Hur are with the Department of Computer Science and Engineering, Korea University, Seoul 02841, Korea. E-mail: {hdkim, mjkim}@isslab.korea.ac.kr, jbhur@korea.ac.kr.*

Manuscript received 5 April 2021; revised 1 November 2021; accepted 4 November 2021. Date of publication 9 November 2021; date of current version 16 January 2023. This work was supported in part by the National Research Foundation of Korea (NRF) through the Korea Government(MSIT) under Grant 2021R1F1A1061420, in part by the Institute of Information & communications Technology Planning & Evaluation (IITP) through the Ministry of Science and ICT under Grants IITP-2021-0-01810, IITP-2021-2020-0-01819, and in part by the National Research Foundation of Korea (NRF) through the Ministry of Education under Grant NRF-2021R1A6A1A13044830, South Korea.
(Corresponding author: Junbeom Hur.)
Digital Object Identifier no. 10.1109/TDSC.2021.3126323

some security assumptions, such as a private key sharing across all participating users. However, VerifyNet depends on reliable secret distribution among all users, where the shared secret is a key to achieving verifiability [30].

In this regard, we raise two questions. First, is the trusted setup adopted by prior work secure in the context of privacy-preserving cross-device FL? Second, is it possible to construct a verifiable FL scheme for privacy-preserving cross-device settings without relying on any trusted setup?

In search of answers, we first scrutinize the verification mechanism of VerifyNet and demonstrate that an attacker can uncover encrypted local model updates from a victim within a reasonable time. Specifically, this attack is realized by our twofold observations. First, the model updates are highly biased, forming bell-shaped distributions [34]. Second, users in VerifyNet share a predefined secret key for a keyed homomorphic hash [35], [36], not only to verify the correctness of model aggregation but also to encode the local model updates. Consequently, this creates a potential vulnerability allowing attackers to deliver brute-force attacks on a victim's hashes (images) and recover the corresponding local model updates (preimages). In our experiment on the MNIST training dataset run by VerifyNet, an attacker can recover about 980 out of 1,000 local model updates from their hashes within one day (when model updates have 16-bit precision) to 90 days (32-bit precision).

Then, we propose VERSA, a verifiable secure aggregation (SA) scheme for cross-device FL, which is designed to run on top of SA [37], a lightweight model aggregation protocol that preserves the privacy of individual model updates in the cross-device FL environment. Considering the unreliability and resource constraints of users in FL, VERSA has the following two design goals. First, verifiability should be achieved without relying on any trusted setup among all users, and second, the verification process should be computationally lightweight.

VERSA satisfies these requirements by employing a secret expansion through a pseudorandom generator (PRG). Specifically, users in SA collectively generate a set of pairwise shared secrets for local model encryption. In VERSA, the shared secret is used as a master key to derive three session keys through PRG. One session key is used to encode the local model update into a *model verification code*. This code is particularly useful for verifying model aggregation because an aggregation of individual codes encodes an aggregation of individual local model updates. Unfortunately, users cannot send their model verification codes in plaintext to the central server because they are vulnerable to the brute-force attacks described earlier. To address this problem, the other two session keys are used to protect the local model update itself and the corresponding model verification code via the encryption of SA. Under the security of SA, an attacker can only obtain an aggregated verification code and cannot recover any individual model verification code from it. Meanwhile, every user employs the aggregated verification code to verify the correctness of the corresponding aggregated model update.

VERSA achieves the verifiability of the model aggregation via the same lightweight cryptographic primitive as the PRG that SA utilizes. This property is extremely useful when a model update is a high-dimensional vector [38], [39]. To verify an aggregated model that consists of an n-dimensional vector, each user in VERSA performs n executions of integer multiplication and addition. In contrast, users in VerifyNet perform $4n$ pairing and n exponentiation operations in a cyclic group of prime order, both of which are computationally extensive group operations. To demonstrate our scheme's efficacy, we implement VERSA under three datasets (MNIST [40], SVHN [41], and CIFAR100 [42]). The SA [37] is a baseline privacy-preserving model aggregation protocol on which VERSA is built; thus, we demonstrate the cost of achieving verifiability over SA by comparing it with SA. As VerifyNet [30] was also built upon SA, we present the cost-effectiveness of our scheme compared with VerifyNet. According to the experimental results, the user-side cost in terms of time for verifying a 1,000-dimensional vector in VERSA is approximately 15 ms, whereas the cost is 20 min for VerifyNet.

1.1 Contribution

Our work makes the following contributions:

- We propose a model recovery attack on VerifyNet [30] and demonstrate that an attacker can recover a victim's model updates within a reasonable time.
- We propose VERSA, a verifiable and privacy-preserving model aggregation scheme. VERSA achieves verifiability of model aggregation via a lightweight primitive like pseudorandom generators, best supporting cross-device FL.
- We experimentally confirm the model accuracy of VERSA under three datasets (MNIST, SVHN, and CIFA100), revealing that VERSA achieves both privacy and verifiability of model updates without degrading accuracy.
- We conduct a comparative performance analysis of VERSA over SA and VerifyNet. The evaluation results demonstrate that the additional cost to run VERSA on top of SA is remarkably small and orders of magnitude faster than VerifyNet.

1.2 Organization

The rest of this paper is structured as follows. We introduce the background of FL, SA, and verifiability in the FL context in Section 2. Then, we describe the system and threat model of VERSA in Section 3. In Section 4, we explain the model recovery attack and its result. In Section 5, we present the construction of VERSA in detail, followed by the security and performance analyses in Sections 6 and 7, respectively. We provide related work in Section 8, and conclude the paper in Section 9.

2 BACKGROUND

In this section, we introduce the systematic overview of FL and SA, and their security issues. Then, we describe verifiability in the context of FL.

2.1 Federated Learning

Federated learning (FL) is a decentralized machine learning technique where a centralized model is trained by aggregating locally computed models, while the training data are distributed to each user and never shared with others [5]. A central server orchestrates the training process by repeating a series of rounds, each consisting of the following processes:

- *User selection*: The server selects a set of users and provides the current model parameters to them. These users may survive throughout the round or drop out before the current round is completed.
- *Local training*: Each user locally computes an update to the model. The update typically involves a variant of a mini-batch stochastic gradient descent rule [39], which returns a gradient as a local training result.
- *Aggregation*: The server collects a set of gradients sent from the surviving users and computes a model aggregation.
- *Global model update*: The server updates the current global model using the aggregated gradient. This new global model is published for the subsequent round.

Although the training data of users never leave each user's local storage, the locally computed model parameters can be exploited to infer the training data [7], [10]. Therefore, it is highly desirable to keep the local model parameters confidential while correctly enabling their aggregation.

FL is widely applicable to two distinctive scenarios: the cross-device and cross-silo settings. In the cross-device setting, numerous resource-constrained users participate in the learning, and the network condition is highly unreliable [43], [44]. In contrast, the cross-silo setting encompasses more stable environments where a small number of organizations equipped with rich computing resources participate as users, and the network channel is reliable [45], [46]. We focus on the cross-device setting in this paper.

2.2 Secure Aggregation

Secure aggregation (SA) [37] can effectively deal with the privacy concern of cross-device FL. SA enables a set of users, who are unreliable and do not have direct communication channels to collectively update global models via a central server while hiding individual gradients. Precisely, at the user selection phase, the server broadcasts a list of selected users to request their participation in the current round. The server and users conduct the following processes sequentially:

- *Advertising and sharing key*: This corresponds to the *user selection* phase in FL. Each selected user is notified of the list of the other participating users and their public keys. The user derives a noise from the secret key. This secret key is further split into n shares via a t-out-of-n secret sharing algorithm [47], where n and t refer to the number of participating users and quota for reconstructing the secret key, respectively. Each share is encrypted using public keys and sent to the corresponding users individually.
- *Masking*: This corresponds to the *local training* phase in FL. Each user trains locally and masks the gradient using the noise he computed in the previous phase.
- *Unmasking*: This corresponds to the *aggregation* phase in FL. The server collects a set of masked gradients sent from the users in the previous phase and computes a model aggregation by aggregating all masked gradients. If all masked gradients are submitted successfully without any dropouts, then their aggregation leads to the correct aggregation of all gradients. Otherwise, the aggregated gradient contains noises, where each noise is made by each dropped-out user. The server handles this problem by making t surviving users reconstruct the secret keys of dropped-out users. Then, the server calculates the corresponding noise from the reconstructed secret keys, removing the noise from the noisy aggregation.

Due to the usage of t-out-of-n secret sharing [47], the privacy of the individual gradient of each user is preserved as long as the maximum number of users the server may corrupt does not exceed t. In addition, SA uses a lightweight cryptographic primitive (e.g., a PRG) to preserve privacy. Thus, SA is applicable to cross-device FL where user communication and computational resources are limited. A technical overview of SA is given in Section 5.2.

2.3 Verifiable Federated Learning

Verifiabililty is one of the important security requirements in FL [33]. In FL, achieving verifiability is considered in two different scenarios. The first scenario is that the server plays the role of a prover, attesting to its execution of faithfully aggregating the local model parameters. Users in this scenario are individual verifiers. In the second scenario, the users play the role of provers attesting that they are not deviating from the protocol (e.g., individual local model parameters are correctly generated), whereas the server is the verifier. In this paper, we focus on the first scenario.

Xu *et al.* [30] recently showed how to achieve verifiability on top of SA. Their scheme has limitations in terms of performance and security. The performance issue arises from using a homomorphic hash [36], which is computationally costly as it heavily relies on the bilinear pairing [48]. The usage of the bilinear pairing is a dominant performance bottleneck when the gradients are high-dimensional vectors because the pairing operations must run on each entry in a vector.

The security concern is significantly more alarming than the performance issue because it can lead to privacy violations. More precisely, we conduct an in-depth analysis of the scheme by Xu *et al.* [30] and find that collusion between the server and user can enable brute-force attacks on the *hashed* values submitted by a victim such that most entries of the input vectors can be recovered within a reasonable amount of time. In Section 4, we describe how the homomorphic hash is used to verify the model aggregation and demonstrate that their scheme is vulnerable to our model recovery attack.

3 System and Threat Model

In this section, we describe the system and threat model of VerSA. Fig. 1 depicts the system model of VerSA compared

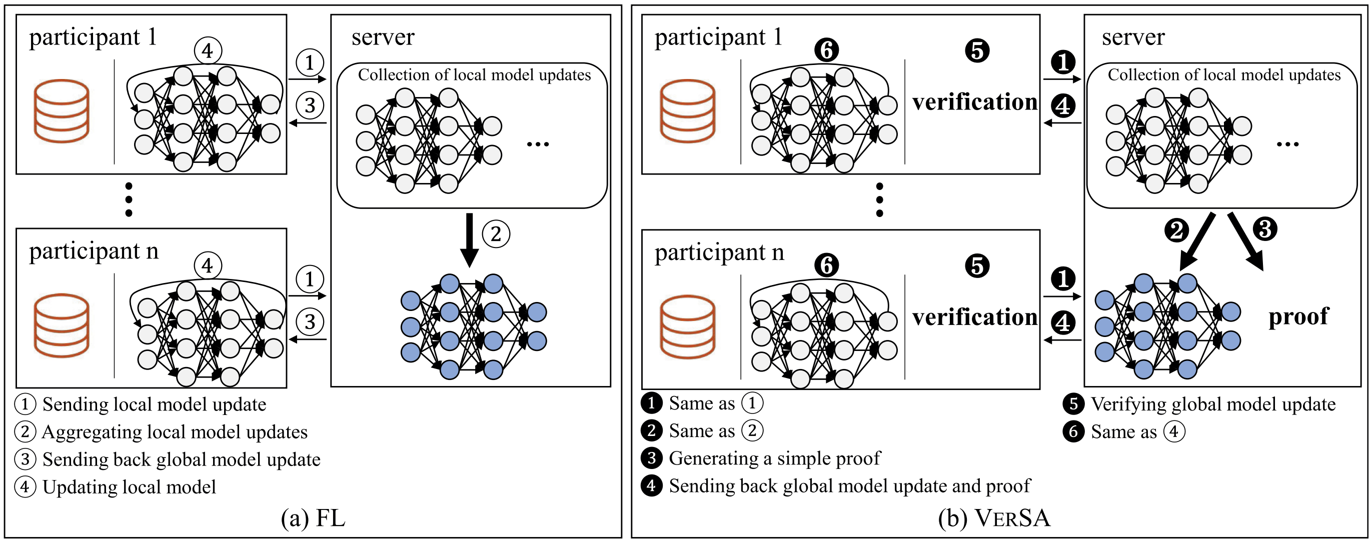

participant 1
④
①
③
server
Collection of local model updates
...
participant n
④
①
③
②
① Sending local model update
② Aggregating local model updates
③ Sending back global model update
④ Updating local model
(a) FL
participant 1
❻
❺
verification
❶
❹
server
Collection of local model updates
...
participant n
❻
❺
verification
❶
❹
❷
❸
proof
❶ Same as ①
❷ Same as ②
❸ Generating a simple proof
❹ Sending back global model update and proof
❺ Verifying global model update
❻ Same as ④
(b) VERSA

Fig. 1. System models of FL and VERSA.

with that of FL. In Fig. 1a, FL is run by n participants and a server. All participants send their local model updates to the server (①). The server aggregates local model updates (②) and sends the result back to each participant (③). Lastly, all participants apply the global model update to their model (④). By applying SA to FL, the local model updates are encrypted prior to be sent to the server (①), while enabling the server to retrieve the aggregated result in plaintext without decryption (②). In Fig. 1b, VERSA runs on top of the SA-enabled FL, where the server generates a proof attesting to the correctness of its execution (❸) and broadcasts the result (❹). Finally, each participant verifies the global model update (❺).

3.1 System Model

VERSA enables every surviving user to verify the correctness of the aggregated model and runs on top of SA, achieving verifiability using the same lightweight cryptographic primitive as SA. Each surviving user runs the following sequential processes:

– *Advertising and sharing key*: This is same as the *advertising and sharing key* phase in SA.
– *Masking*: This corresponds to the *masking* phase in SA. In addition to the gradient masking, the user derives a public evaluation key and private verification key, where the two keys are used in the subsequent phases by the server and user, respectively.
– *Unmasking and generating proof of model aggregation*: This corresponds to the *unmasking* phase in SA. In addition to the model aggregation, the server generates a proof attesting to the correctness of the aggregated gradient using the evaluation key. The server broadcasts the proof and aggregated gradient.
– *Verifying model aggregation*: The user verifies the correctness of the model aggregation using the verification key. The aggregated gradient is accepted if the verification succeeds; otherwise, it is rejected.

Technical details of VERSA are provided in Section 5.

3.2 Threat Model

In VERSA, we assume that all users agree upon a model aggregation orchestrated by the central server. The users consent to release the final result of the model aggregation to every user. These users have a common interest in soundness (i.e., retrieving correct global model updates from the untrusted central server) and privacy (i.e., hiding their local model updates from each other and the server).

The central server should be reliable because its role is important in FL, i.e., it orchestrates the training process. Nevertheless, the central server also potentially represents a single point of failure, and having a trusted server may not always be available or desirable in many collaborative learning scenarios [33].

In accordance with the aforementioned assumption, we suppose the server is malicious in terms of soundness. The server may deviate arbitrarily from the protocol by supplying incorrect messages or executing different computation than expected to return an incorrect result to all users. Such an assumption captures the adversarial behaviors in multiclient verifiable computation [26], where the goal of the server is to deceive the users into accepting a wrong result.

In terms of privacy, we assume that the server is malicious and and users are semi-honest. The server and some users may collude to obtain the best offensive capabilities as possible against the other victims' local model updates. The maximum number of users the server may corrupt does not exceed t which is the threshold of the t-out-of-n secret sharing algorithm [47]. Specifically, given a set of user group $\mathcal{U}$ and any subset $\mathcal{C} \subset \mathcal{U} \cup \{S\}$, where S is the server, it is required that $|\mathcal{C}/\{S\}| < t$.

Out-of-Scope Attacks. In this paper, we do not consider user-server collusion attacks to bypass the *verifying model aggregation* phase. This exception is due to the impossibility result [27], in which multiclient verifiable computation cannot achieve verifiability in the presence of the users colluding with the server. In addition, we consider neither membership inference attacks [13] that infer the training data from the model itself nor poisoning attacks [12] in which malicious users deliberately affect their training data to manipulate the global model. In the poisoning attack, these malicious attackers may submit distorted local model updates (e.g., outside the bounds of their expected range). Such bad inputs could be handled by employing noninteractive zero-knowledge [49] under a trusted setup among all users. However, it is costly in the cross-device FL setting because users may frequently drop out and do not have direct communication channels with each other.

4 Model Recovery Attack

We describe the model recovery attack on VerifyNet [30], a homomorphic hash-based verifiable SA, and demonstrate the feasibility of our attack. To better understand the attack procedure, we briefly explain the general methodology for achieving verifiability over aggregated gradients using homomorphic hashes.

4.1 Homomorphic Hash

A homomorphic hash H [35], [36] allows evaluating an arithmetic function f on the input from a set of hash values $H(m_1), \dots, H(m_t)$ such that an evaluation algorithm returns $H(f(m_1, \dots, m_t))$. More precisely, a family of one-way keyed homomorphic hash H consists of three algorithms, as follows:

- $k \leftarrow H.gen$: This is a private key k generation algorithm for a keyed hash function H_k.
- $H_k(m) \leftarrow H$: This is a hash computation algorithm that returns $H_k(m)$ on input m.
- $H_k(f(m_1, \dots, m_t)) \leftarrow H.eval$: This is a function evaluation algorithm that returns $H_k(f(m_1, \dots, m_t))$ on input a set of hash values $H_k(m_1), \dots, H_k(m_t)$.

The security of a one-way keyed hash function H guarantees that it is practically infeasible to invert from $H_k(m)$ to recover m.

Application to SA. An aggregated model update is the sum of all gradients in FL, each generated by individual users. As H is one-way and supports computations of arithmetic circuits, one can easily apply a homomorphic hash to SA to verify the correctness of the sum while preserving the privacy against the server by performing the following process sequentially:

- All users $u \in \mathcal{U}$ share the secret key k for a homomorphic hash H.
- Each user u computes $H_k(\mathbf{x}_u)$, a hash of the gradient $\mathbf{x}_u$. Each user u submits the pair $(\mathrm{SA.mask}(\mathbf{x}_u), H_k(\mathbf{x}_u))$ to the server, where $\mathrm{SA.mask}(\mathbf{x}_u)$ refers to the masked gradient of $\mathbf{x}_u$.
- The server aggregates all masked gradients $\{\mathrm{SA.mask}(\mathbf{x}_u)\}_{u \in \mathcal{U}}$, which returns $\mathbf{z}$. The server evaluates all hash values received from $u \in \mathcal{U}$, which returns $\hat{\mathbf{z}} = H_k(\sum_{u \in \mathcal{U}} \mathbf{x}_u)$. The server broadcasts a pair $(\mathbf{z}, \hat{\mathbf{z}})$.
- In this phase, users verify whether $\mathbf{z} \overset{?}{=} \sum_{u \in \mathcal{U}} \mathbf{x}_u$ with the assistance of $\hat{\mathbf{z}}$. To do this, each user computes a hash of $\mathbf{z}$. Then, the user accepts $\mathbf{z}$ as a correctly aggregated model update if and only if $H_k(\mathbf{z}) = \hat{\mathbf{z}}$ and aborts otherwise.

This process does not intend to enable the server to learn the individual gradient from $H(\mathbf{x}_u)$ as H is a one-way keyed hash function. VerifyNet [30] uses the above approach to allow users to check the correctness of aggregated results while preserving privacy of individual gradients.

4.2 Our Attack

Our model recovery attack on VerifyNet [30] leverages the following two features we observed. First, the distribution of model parameters (i.e., gradients) is highly biased, such that the entries of gradient generated by an SGD rule form bell-shaped distributions around zero [34]. Without this observation, we would have no choice but to launch a naïve brute-force attack over a very large range like $(-\infty, \infty)$, which is computationally infeasible. Second, to encode gradients and verify the aggregated gradient in VerifyNet, all users must share the same secret parameters used for a homomorphic hash. These observations lead us to hypothesize that VerifyNet is vulnerable to brute-force attacks launched by malicious users colluding with the server such that the encoded gradients can be recovered from a brute-force attack on a victim's homomorphic hash outputs. Such user-server collusion is the topmost security concern for preserving confidentiality of the individual gradients [30], [37]. Overall, these two observations are strong clues to balance recovery ratios and the time taken to recover the models, i.e., recovering as many model parameters as possible within a relatively short period.

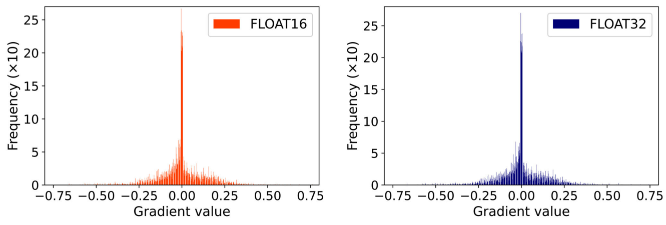
4.2.1 Attack Scenario

We simulated a cross-device FL setting to prove our hypothesis, where we run TensorFlow [50] on the MNIST training dataset [40] on each user's local device using the stochastic gradient descent rule. Then, we encoded the gradients using a homomorphic hash and sent the result to the server. By colluding with a malicious user, the server obtained all homomorphic hash parameters and ran a brute-force attack on the homomorphic hash outputs submitted by the victim.

4.2.2 Implementation Setup

To implement the homomorphic hash, we used two hash algorithms: the DRV hash [36] and the KFM hash [35]. The former was chosen because it is used in VerifyNet, whereas the latter was chosen to demonstrate that the feasibility of our attack does not depend on any specific homomorphic hash used to encode the gradients. For the DRV hash, we used MNT224, a Type-III curve for pairings implemented in the pairing-based cryptography library [51]. For the KFM hash, we used 256-bit, 512-bit, and 1,024-bit prime numbers, each instantiating three independent KFM hashes written in Java. Consequently, we used these three KFM hashes in order to measure the feasibility of our attack over various bit-length prime numbers.

Our attacks were run on two Ubuntu 18.04 LTS servers. First, the DRV hash attack was conducted on a system equipped with 32GB RAM and an AMD Ryzen 3950x 16-core processor, each running on a base clock of 3.5 GHz with 16-thread OpenMP setup. Next, to attack the KFM hashes, we used a system with an Intel Core i9-9900K processor with 3.6 GHz base clock and 64GB RAM. We used only a single core to launch our attack for each hashed gradient.



(a) 16-bit precision (b) 32-bit precision

Fig. 2. Gradient distributions.

4.2.3 Processing Float-Type Gradient

Because both DRV and KFM hash take integer values as input, the floating-point parameters obtained from training need to be scaled in the range $\{-\frac{p-1}{2}, \ldots, 0, \ldots, \frac{p-1}{2}\}$, where p is a large prime number used for the hashes. Following the float-to-integer conversion method [52], [53], we quantized a floating-point value v into an integer $\lfloor v \cdot \alpha \rfloor$ with a scaling factor α, where a larger value of α leads to low quantization errors. We set $\alpha = 2^{prec}$, where $prec$ refers to the bit-precision a neural net employs to represent the gradient values.

Recent studies have demonstrated that, instead of traditional single precision (32-bit), 16-bit precision can be sufficient to train a neural net without affecting the model accuracy in favor of energy efficiency and reduced precision representations [54], [55]. Thus, we used 32-bit precision and 16-bit precision for the MNIST training dataset to obtain two separate gradient sets. Our attack on the latter set measures the effect of reduced precision regarding the brute-force attacks over the homomorphic hash-based verifiable SA.

4.2.4 Attack Process

Each of the two gradient sets contains 750 gradients, where each gradient forms a vector of 10 entries. As the distribution is expected to be highly biased around zero, most of the parameters would reside within a short symmetric interval, e.g., $(-x, x)$. Fig. 2 plots the distributions of those entries, indicating that they form narrow bell-shaped curves centered on zero. The figure also indicates that most parameters would be recovered even if we set x as a small value, e.g., $x = 0.2$. Lastly, we obtained four hash values for each entry using DRV, KFM-256, KFM-512, and KFM-1024 hash functions.

Let $h(\lfloor v \cdot \alpha \rfloor)$ be the hashed counterpart of a float-type gradient entry v. Let $0 < r < 1$ be a float-type value such that there exists an integer n satisfying $\lfloor v \cdot \alpha \rfloor = r \cdot n$. While exhaustively increasing (or decreasing) i from zero, we check whether $h(\lfloor v \cdot \alpha \rfloor) = h(\lfloor r \cdot \alpha \rfloor \cdot i)$. This loop ends when $\lfloor r \cdot \alpha \rfloor \cdot i > \lfloor x \cdot \alpha \rfloor$ (or $\lfloor r \cdot \alpha \rfloor \cdot i < \lfloor -x \cdot \alpha \rfloor$), indicating that the loop is beyond the given range $(-x, x)$, where x is a real number.

The choice of (r, x) affects the attack performance. For example, the number of recovered gradients would increase when we set r and x as small and large as possible, respectively, at the cost of increased time to complete the attack. In this regard, we tried diverse pairs of (r, x) to empirically balance the number of recovered gradients and the required time. In our experiment, we set r as 10^{-7} and 10^{-9} for 16-bit and 32-bit precision, respectively, with $x \in \{0.2, 0.3, 0.4\}$.

TABLE 1
Summary of Attack Results

	16-bit		32-bit	
	# grad.	Hours	# grad.	Days
		$(-0.2, 0.2)$		
DRV	893	6.63	902	23.93
KFM-256	861	1.98	879	9.90
KFM-512	832	7.30	895	28.69
KFM-1024	866	16.81	894	73.99
		$(-0.3, 0.3)$		
DRV	936	10.98	952	36.59
KFM-256	940	2.51	969	9.56
KFM-512	956	7.37	958	31.17
KFM-1024	956	18.48	956	78.39
		$(-0.4, 0.4)$		
DRV	978	13.17	988	46.97
KFM-256	958	2.58	981	13.63
KFM-512	958	8.27	986	30.50
KFM-1024	976	20.44	988	87.03

# grad. refers to the number of gradient entries from 100 hashed gradients.

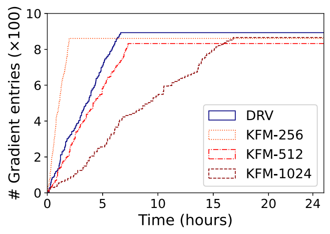
For each hashed gradient set, we chose 100 hashed gradients uniformly at random and launched our attack on each hashed gradient entry to measure the time taken to recover its counterpart (i.e., the float-type gradient entry). We aborted the attack on each hashed gradient entry when its counterpart was not found within the given range $(-x, x)$.

4.2.5 Attack Result

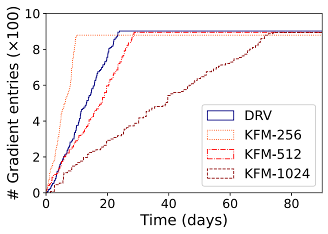
Table 1 summarizes the attack results measured over three months, while Fig. 3 presents in detail how fast we recovered gradient entries within the given time period. The attack results are evaluated regarding accuracy and efficiency. Accuracy refers to the ratio of the number of recovered gradient entries over total number of gradient entries. Efficiency refers to the time taken to complete the attack.

First, we explain the attack results in terms of accuracy. Within a small range $(-0.2, 0.2)$, our attack has an accuracy of 0.875 on average in both 16-bit and 32-bit precision. The accuracy increases as we expand the range, as shown in Table 1. For example, it reaches an accuracy of 0.936 and 0.978 when we launch the attack using DRV for 16-bit precision within the ranges $(-0.3, 0.3)$ and $(-0.4, 0.4)$, respectively. Lastly, our attack demonstrates that one cannot witness any significant accuracy difference between different homomorphic hashes. This result is alarming because, regardless of the type of underlying homomorphic hash, our attack can recover gradients with high accuracy.

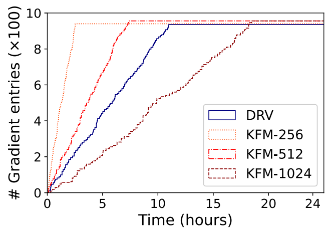
Next, we discuss our attack results in terms of efficiency. As the table and figures depict, one of the significant factors that affect efficiency is the bit precision. For example, in 16-bit precision, all of our attacks are complete within a day, e.g., the most time-consuming attack is on KFM-1024, which takes 20.44 hours. However, as we used 32-bit precision, efficiency degrades to a minimum of 9.56 to a maximum of 87.03 days. Such time is still considered extremely short. Moreover, utilizing multi-core capabilities could drastically



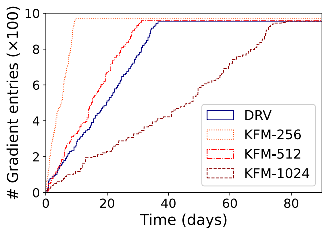
(a) 16-bit, $(-0.2, 0.2)$



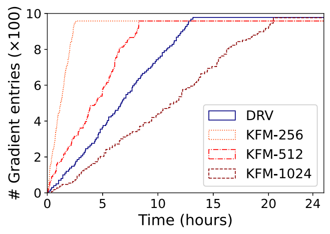
(b) 32-bit, $(-0.2, 0.2)$



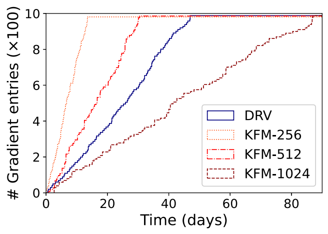
(c) 16-bit, $(-0.3, 0.3)$



(d) 32-bit, $(-0.3, 0.3)$



(e) 16-bit, $(-0.4, 0.4)$



(f) 32-bit, $(-0.4, 0.4)$

Fig. 3. Results of our attacks. In each plot, we chose, from over a set of 750 hashed gradients, 10^2 hashed gradients (i.e., 10^3 gradient entries in total) at random. For each hashed gradient, we launched our attack and recorded how many gradient entries were recovered from their hashed counterparts over time within a given range $(-x, x)$, where $x \in \{0.2, 0.3, 0.4\}$.

reduce the time. Recall that we used only a single core to launch the attack. Eventually, the attack confirms that the gradients encoded into homomorphic hashes can be recovered within a short period with high accuracy.

5 VerSA Construction

VerSA is designed to support the verification capability over the SA protocol from Bonawitz *et al.* [37] using the same lightweight cryptographic gadgets that SA relies on. Accordingly, we first recapitulate the cryptographic primitives of SA (and, accordingly, VerSA). We provide a technical overview of SA and describe the core technique of VerSA for achieving verifiable computation on top of SA. Lastly, we provide details on the design and implementation of VerSA.

5.1 Cryptographic Gadget

5.1.1 Key Agreement

The key agreement (*KA*) protocols [56] allow two different users to generate a shared secret key over a public channel. The *KA* consists of the following algorithms:

- $pp \leftarrow \mathbf{KA.param}(\lambda)$: This algorithm takes a security parameter λ as input and returns a public parameter pp.
- $(pk_u, sk_u) \leftarrow \mathbf{KA.gen}(pp)$: This algorithm takes pp as input and returns a public and secret key pair (pk_u, sk_u) for user u.
- $s_{u,v} \leftarrow \mathbf{KA.agree}(sk_u, pk_v)$: This algorithm takes sk_u of user u and pk_v of user v as input and returns a shared key $s_{u,v}$.

5.1.2 Secret Sharing

Shamir's t-out-of-n secret sharing (*SS*) protocol [47] allows a user to split a secret into n shares such that at least $t \leq n$ shares are required to recover this secret. Shares less than t leak no information about the secret. The *SS* consists of the following algorithms:

- $pp \leftarrow \mathbf{SS.share}(s_u, t, \mathcal{U})$: This algorithm takes a secret value s_u of u, a threshold t, and a set of user group $\mathcal{U}$ as input, where $t \leq |\mathcal{U}|$. It returns $\{s_{u,v}\}_{v \in \mathcal{U}}$.
- $(pk_u, sk_u) \leftarrow \mathbf{SS.recon}(\{s_{u,v}\}_{v \in \mathcal{V}}, t)$: This algorithm takes a set of shares $\{s_{u,v}\}_{v \in \mathcal{V}}$ as input, where $\mathcal{V} \subset \mathcal{U}$. The protocol aborts if $t > |\mathcal{V}|$; it reconstructs s_u otherwise.

5.1.3 Authenticated Encryption

Authenticated encryption (*AE*) [57] guarantees the confidentiality and integrity of messages. In this paper, we use symmetric *AE* (i.e., the encryption and decryption key is identical). The *AE* consists of the following algorithms:

- $k \leftarrow \mathbf{AE.gen}(\mathcal{K})$: This algorithm takes a keyspace $\mathcal{K}$ as input from which it samples a secret key k uniformly at random.
- $ct \leftarrow \mathbf{AE.enc}(k, m)$: This algorithm takes k and a message m as input and returns a ciphertext ct.
- $m \leftarrow \mathbf{AE.dec}(k, ct)$: This algorithm takes k and ct as input and returns m.

5.1.4 Pseudorandom Generator

A pseudorandom generator (*PRG*) [58] maps an input seed to a pseudorandom output sequence and guarantees that the output distribution on a uniformly chosen seed is computationally indistinguishable from a uniform distribution. In this paper, we use $\mathbf{PRG} : \{0,1\}^* \rightarrow \mathbb{Z}_R^n$, which expands an input value into n-dimensional output vector, where R is a large integer.

5.1.5 Public Key Infrastructure

The public key infrastructure [59] binds public keys with the respective identities of users, underpinning the confidence in trust that users have. A user with identity u has a signing and public key pair (sk_u, pk_u) issued by a trusted third party. In VerSA, we use the public key infrastructure when users advertise their keys to join the SA protocol.

5.1.6 Digital Signature

With the public key infrastructure, a digital signature (*DS*) provides an authentication mechanism that securely associates users with the messages of their choice. The *DS* consists of the following algorithms:

- $(sk, pk) \leftarrow \mathbf{DS}.\mathbf{gen}(\lambda)$: This algorithm takes a security parameter λ as input and returns a signing and public key pair (sk, pk).
- $\sigma \leftarrow \mathbf{DS}.\mathbf{sign}(sk, m)$: This algorithm takes sk and a message m as input and returns a signature σ.
- $\{0, 1\} \leftarrow \mathbf{DS}.\mathbf{vrfy}(pk, m, \sigma)$: This algorithm takes pk, m, and ct as input and returns a verification result of 0 and 1, indicating verification failure and success, respectively.

The SA can be either *DS*-enabled or *DS*-disabled. The former and latter are used to guarantee security in the active adversary and honest-but-curious adversary model, respectively. In this paper, we build VERSA on top of the *DS*-disabled SA for ease of presentation. Nevertheless, one can easily extend VERSA using the *DS*-enabled SA.

5.2 Technical Overview of SA

The SA uses *KA* to allow every pair of two users u, v, whose secret keys are sk_u and sk_v, to generate a shared secret value jointly. We denote this secret value as $s_{u,v}$ if it is held by user u or $s_{v,u}$ if it is held by user v. Note that $s_{u,v} = s_{v,u}$. User u uses $s_{u,v}$ as a seed for *PRG* to derive a random vector $\mathbf{p}_{u,v} = \mathbf{PRG}(s_{u,v})$ to encrypt the gradient. More precisely, given a set $\mathcal{U}$ of n users with logical identities $[1, \ldots, n]$, user u masks gradient $\mathbf{x}_u$ as follows:

$$\mathbf{y}_u = \mathbf{x}_u + \sum_{v \in \mathcal{U}, u < v} \mathbf{PRG}(s_{u,v}) - \sum_{v \in \mathcal{U}, u > v} \mathbf{PRG}(s_{u,v}), \tag{1}$$

where $\mathbf{y}_u \pmod{R} \in \mathbb{Z}_R^n$.

Consider an aggregation of two masked gradients $\mathbf{y}_u$ and $\mathbf{y}_v$, where $u < v$. Then, the two random vectors $\mathbf{PRG}(s_{u,v})$ and $\mathbf{PRG}(s_{v,u})$, each generated by u and v respectively, cancel each other out. Consequently, if all users submit $\mathbf{y}_{u \in \mathcal{U}}$ successfully, then the server obtains the aggregated gradient $\sum_{u \in \mathcal{U}} \mathbf{x}_u = \sum_{u \in \mathcal{U}} \mathbf{y}_u$.

The SA is further elaborated by addressing user dropouts and late response problems.

5.2.1 User Dropouts

Consider when user u drops out before sending his masked gradient. Then, all random vectors $\mathbf{PRG}(s_{v,u})$, where $v \in \mathcal{U}$ remain uncancelled in $\sum_{u \in \mathcal{U}} \mathbf{y}_u$. The SA addresses this concern as follows. Before submitting $\mathbf{y}_u$, user u splits the secret key sk_u into n shares using **SS**. These shares are distributed to all users $v \in \mathcal{U}$ such that user v is given exactly one share. Then, the server asks the surviving users to submit the shares of the dropped-out user's secret key sk_u, so the server recovers the key, computes $\mathbf{PRG}(s_{u,v})$, and removes all instances of $\mathbf{PRG}(s_{v,u})$s from $\sum_{u \in \mathcal{U}} \mathbf{y}_u$.

5.2.2 Late Response

Consider when the users fail to communicate with the server promptly. Specifically, user u may transfer the gradient late, so the server collects the shares of sk_u. This clearly raises a security issue because the server can recover sk_u from the shares and leak $\mathbf{x}_u$ from $\mathbf{y}_u$, which arrives late at the server, by deriving and removing all $\{s_{u,v}\}_{v \in \mathcal{U}}$ used for masking $\mathbf{x}_u$. The SA solves this problem by allowing u to mask $\mathbf{x}_u$ twice: u selects a random seed b_u and computes the following:

$$\mathbf{y}_u = \mathbf{y}_u + \mathbf{PRG}(b_u).$$

The server should remove $\mathbf{PRG}(b_u)$ to obtain the aggregated gradient. To this end, every user u splits b_u and sends the shares to all users $v \in \mathcal{U}$ beforehand. Then, during the unmasking phase, the server must make an explicit choice concerning each user. For all dropped-out users $u \in \mathcal{U}$, the server asks the surviving users to submit the shares of sk_u. For all surviving users $u \in \mathcal{U}$, the server asks them to submit the shares of b_u. Consequently, the server can recover the aggregated gradient in plaintext by removing all $\mathbf{PRG}(s_{v,u})$s and $\mathbf{PRG}(b_u)$s from $\sum_{u \in \mathcal{U}} \mathbf{y}_u$.

5.3 Proposed Scheme

A complete construction of VERSA is provided in Fig. 4. It is described following most terminologies used in SA [37] for consistency. Hereafter, we ignore user dropouts and late response for simplicity. That is, we assume that all pairs $(\mathbf{y}_u, \bar{\mathbf{y}}_u)_{u \in \mathcal{U}}$ are derived from Eqn. (1) and arrive at the server in time. We believe that this assumption helps readers quickly grasp the key ideas behind VERSA.

Intuitively, VERSA achieves the verifiability of an aggregated gradient employing double aggregation. The first aggregation is for computing the aggregated gradient itself, whereas the second one is for proving the correctness of the first aggregation. In VERSA, each user u submits $\mathbf{y}_u$ and $\bar{\mathbf{y}}_u$. Specifically, $\bar{\mathbf{y}}_u$ encrypts a model verification code $F(\mathbf{x}_u) = \mathbf{a} \circ \mathbf{x}_u + \mathbf{b}$, where operation $\circ$ is the Hadamard product. The two vectors $(\mathbf{a}, \mathbf{b})$ are secret vectors hidden from server view, and all users $u \in \mathcal{U}$ can compute the same pair of vectors $(\mathbf{a}, \mathbf{b})$. Below we explain (i) how the model verification code is used for verifiable computation and (ii) how to share $(\mathbf{a}, \mathbf{b})$ *without* requiring additional communications between users.

The server performs aggregation twice such that it obtains $\mathbf{z} = \sum_{u \in \mathcal{U}} \mathbf{y}_u$ and $\bar{\mathbf{z}} = \sum_{u \in \mathcal{U}} \bar{\mathbf{y}}_u$, where $\bar{\mathbf{z}} = \mathbf{a} \circ \sum_{u \in \mathcal{U}} \mathbf{x}_u + |\mathcal{U}| \cdot \mathbf{b}$. Eventually, the users verify $\mathbf{z}$ by checking if the following condition holds:

$$\bar{\mathbf{z}} \overset{?}{=} \mathbf{a} \circ \mathbf{z} + |\mathcal{U}| \cdot \mathbf{b}.$$

Intuitively, the verifiability of $\mathbf{z}$ is preserved for the following reason: $\bar{\mathbf{z}}$ encapsulates $\mathbf{z}$ using two vectors $(\mathbf{a}, \mathbf{b})$, where these vectors are hidden from the server. Thus, the server's probability of forging $\sum_{u \in \mathcal{U}} \mathbf{x}_u$ (i.e., the aggregated result obtained from $\mathbf{z}$) is reduced to the probability of recovering $(\mathbf{a}, \mathbf{b})$, which is infeasible due to the one-wayness property of the **PRG**. Moreover, the server cannot recover $F(\mathbf{x}_u)$ from $\bar{\mathbf{y}}_u$ as long as the privacy guarantee of SA is preserved. That is, we mask $F(\mathbf{x}_u)$ using SA's masking method. Thus, even if $(\mathbf{a}, \mathbf{b})$ is revealed to the server due to user-server collusion, the server probability of recovering $\mathbf{x}_u$ from $\bar{\mathbf{y}}_u$ is reduced to the probability of breaking SA.

This simplified description ignores a technical hurdle. Specifically, all surviving users must have the pair of vectors $(\mathbf{a}, \mathbf{b})$ in advance. One straightforward approach for realizing the assumption is establishing secure channels between them and ensuring they share such a pair via secure channels. However, this method is unsuitable

Setup:
- All parties agree on the usage of the **KA** protocol, **SS** protocol, **AE** algorithm, and **PRG**. They share the security parameter λ, number of users n, threshold t, and public parameter $pp \leftarrow \textbf{KA.param}(\lambda)$.

Advertising Key:
User u:
- Generate two key pairs $(pk_u^0, sk_u^0) \leftarrow \textbf{KA.gen}(pp)$, $(pk_u^1, sk_u^1) \leftarrow \textbf{KA.gen}(pp)$.
- Send the public key pair (pk_u^0, pk_u^1) to the server.

Server:
- Collect at least $t \leq |\mathcal{U}_1|$ key pairs sent by each user, where $\mathcal{U}_1$ denotes the set of users in this round.
- Broadcast $\{(pk_v^0, pk_v^1)\}_{v \in \mathcal{U}_1}$ to all users in $\mathcal{U}_1$.

Sharing Key:
User u:
- Receive $\{(pk_v^0, pk_v^1)\}_{v \in \mathcal{U}_1}$ and ensure that $t \leq |\mathcal{U}_1|$.
- Split sk_u^1 into t-out-of-$\mathcal{U}_1$ shares $\{sk_{u,v}^1\}_{v \in \mathcal{U}_1} \leftarrow \textbf{SS.share}(sk_u^1, t, \mathcal{U}_1)$.
- Sample $b_u \in_R \mathbb{R}$ and split b_u into t-out-of-$\mathcal{U}_1$ shares $\{b_{u,v}\}_{v \in \mathcal{U}_1} \leftarrow \textbf{SS.share}(b_u, t, \mathcal{U}_1)$.
- For each user $v \in \mathcal{U}_1$, compute $ct_{u,v} \leftarrow \textbf{AE.enc}(\textbf{KA.agree}(sk_u^0, pk_v^0), sk_{u,v}^1 || b_{u,v})$.
- Send all the ciphertexts $\{ct_{u,v}\}_{v \in \mathcal{U}_1}$ to the server.

Server:
- Collect at least $t \leq |\mathcal{U}_2|$ ciphertexts sent by each user, where $\mathcal{U}_2 \subseteq \mathcal{U}_1$ denotes the set of users in this round.
- Broadcast $\{ct_{v,u}\}_{v \in \mathcal{U}_2}$ (a set of ciphertexts generated by users $v \in \mathcal{U}_2$) to all users $u \in \mathcal{U}_2$.

Masking Input:
User u:
- Receive the set of ciphertexts $\{ct_{v,u}\}_{v \in \mathcal{U}_2}$ and ensure that $t \leq |\mathcal{U}_2|$.
- For each user $v \in \mathcal{U}_2$, compute $s_{u,v} \leftarrow \textbf{KA.agree}(sk_u^1, pk_v^1)$. Compute $\alpha \leftarrow \sum_{v \in \mathcal{U}_2} s_{u,v} \pmod R$.
- Expand $s_{u,v}$ into two random vectors $\mathbf{p}_{u,v,0} = \Delta_{u,v,i} \cdot \textbf{PRG}(s_{u,v}||0)$ and $\mathbf{p}_{u,v,1} = \Delta_{u,v,i} \cdot \textbf{PRG}(s_{u,v}||1)$, where, for $i \in \{0,1\}$, $\Delta_{u,v,i} = 1$ if $u > v$; $\Delta_{u,v} = -1$ if $u < v$. For all $u \neq v$, $\mathbf{p}_{u,v,i} + \mathbf{p}_{v,u,i} = \mathbf{0}$ holds.
- Expand b_u into two random vectors $\mathbf{p}_{u,0} = \textbf{PRG}(b_u||0)$ and $\mathbf{p}_{u,1} = \textbf{PRG}(b_u||1)$.
- Expand α into two random vectors $\mathbf{a} = \textbf{PRG}(\alpha||0)$ and $\mathbf{b} = \textbf{PRG}(\alpha||1)$. For input vector $\mathbf{v}$, define $F(\mathbf{v}) = \mathbf{a} \circ \mathbf{v} + \mathbf{b} \pmod R$.
- Given an input vector $\mathbf{x}_u$, compute and send to the server a pair of masked input vectors:

$$\mathbf{y}_u = \mathbf{x}_u + \mathbf{p}_{u,0} + \sum_{v \in \mathcal{U}_2} \mathbf{p}_{u,v,0} \pmod R \quad \text{and} \quad \bar{\mathbf{y}}_u = F(\mathbf{x}_u) + \mathbf{p}_{u,1} + \sum_{v \in \mathcal{U}_2} \mathbf{p}_{u,v,1} \pmod R.$$

Server:
- Collect at least $t \leq |\mathcal{U}_3|$ pairs sent by each user, where $\mathcal{U}_3 \subseteq \mathcal{U}_2$ denotes the set of users in this round.
- Broadcast the set $\mathcal{U}_3$ to all users $u \in \mathcal{U}_3$.

Unmasking Input and Returning Output:
User u:
- Receive the set $\mathcal{U}_3$ and ensure that $t \leq |\mathcal{U}_3|$.
- For each user $v \in \mathcal{U}_2$, decrypt the ciphertext to $sk_{v,u}^1 || b_{v,u} \leftarrow \textbf{AE.dec}(\textbf{KA.agree}(sk_u^0, pk_v^0), ct_{v,u})$.
- Send a set of shares $sk_{v,u}^1$ for users $v \in \mathcal{U}_2 \backslash \mathcal{U}_3$ and $b_{v,u}$ for users $v \in \mathcal{U}_3$.

Server:
- Collect responses from at least $t \leq |\mathcal{U}_4|$ users, where $\mathcal{U}_4 \subseteq \mathcal{U}_3$ denotes the set of users in this round.
- Reconstruct $sk_u^1 \leftarrow \textbf{SS.recon}(\{sk_{u,v}^1\}_{v \in \mathcal{U}_4}, t)$ to recover $(\mathbf{p}_{u,v,0}, \mathbf{p}_{u,v,1})$ for users $u \in \mathcal{U}_2 \backslash \mathcal{U}_3$.
- Reconstruct $b_u^1 \leftarrow \textbf{SS.recon}(\{b_{u,v}^1\}_{v \in \mathcal{U}_4}, t)$ to recover $(\mathbf{p}_{u,0}, \mathbf{p}_{u,1})$ for users $u \in \mathcal{U}_3$.
- Compute $\mathbf{z} = \sum_{u \in \mathcal{U}_3} \mathbf{x}_u$ and $\bar{\mathbf{z}} = \sum_{u \in \mathcal{U}_3} F(\mathbf{x}_u)$:

$$\mathbf{z} = \sum_{u \in \mathcal{U}_3} \mathbf{y}_u - \sum_{u \in \mathcal{U}_3} \mathbf{p}_{u,0} + \sum_{u \in \mathcal{U}_2 \backslash \mathcal{U}_3} \mathbf{p}_{u,v,0} \pmod R \quad \text{and} \quad \bar{\mathbf{z}} = \sum_{u \in \mathcal{U}_3} \bar{\mathbf{y}}_u - \sum_{u \in \mathcal{U}_3} \mathbf{p}_{u,1} + \sum_{u \in \mathcal{U}_2 \backslash \mathcal{U}_3} \mathbf{p}_{u,v,1} \pmod R.$$

- Broadcast $(\mathbf{z}, \bar{\mathbf{z}})$ to all users $u \in \mathcal{U}_4$.

Validating Output:
User u:
- Receive the pair $(\mathbf{z}, \bar{\mathbf{z}})$ and compute $\mathbf{z}' = \mathbf{a} \circ \mathbf{z} + |\mathcal{U}_3| \cdot \mathbf{b} \pmod R$.
- Accept $\mathbf{z}$ if $\mathbf{z}' = \bar{\mathbf{z}}$; abort otherwise.

Fig. 4. VerSA: A verifiable secure aggregation protocol.

for cross-device FL settings in practice. As an alternative approach, a multiparty computation protocol [60] can be applied to allow all users to compute the pair jointly; however, it requires communication rounds linear to the number of the participating entities. Although recent advances in multiparty computation require a constant number of communication rounds (see [61] for more details), its computation cost is still overwhelming due to heavy cryptographic algorithms, such as fully homomorphic encryption [62].

We address this issue using a secret expansion through the *PRG*. Specifically, recall that each user u in SA runs *KA* locally with all public keys of surviving users $v \in \mathcal{U}$ to compute a set of secret values $\{s_{u,v}\}_{v \in \mathcal{U}}$ to mask his gradient. Our approach is to allow every surviving user to derive another secret value from $\{s_{u,v}\}_{v \in \mathcal{U}}$ through the *PRG*. First, user u computes $\alpha \leftarrow \sum_{v\in\mathcal{U}} s_{u,v} \pmod R$. Next, u expands α into two vectors as follows:

$$\mathbf{a} = \mathbf{PRG}(\alpha||0), \quad \mathbf{b} = \mathbf{PRG}(\alpha||1).$$

Due to the one-wayness and pseudo-randomness of the *PRG*, the server can recover neither α nor $(\mathbf{a}, \mathbf{b})$ from $(\mathbf{z}, \bar{\mathbf{z}})$. In the meantime, every surviving user u can derive α from $\{s_{u,v}\}_{v\in\mathcal{U}}$ and generate $(\mathbf{a}, \mathbf{b})$, thus verifying $\mathbf{z}$.

5.3.1 Correctness

The correctness of the sum (i.e., $\sum_{u\in\mathcal{U}} \mathbf{x}_u$ and $\sum_{u\in\mathcal{U}} F(\mathbf{x}_u)$) in VERSA is reduced to the correctness of SA, and this reduction holds even when some users drop out. For ease of presentation, we assume that the server receives all pairs $(\mathbf{y}_u, \bar{\mathbf{y}}_u)_{u\in\mathcal{U}}$ and performs SA correctly. In the case of aggregating $\mathbf{y}_u$, the following condition holds.

$$\begin{aligned}\sum_{u\in\mathcal{U}} \mathbf{y}_u &= \sum_{u\in\mathcal{U}} \mathbf{x}_u + \sum_{u\in\mathcal{U}} \sum_{v\in\mathcal{U}, u<v} \mathbf{PRG}(s_{u,v}) \\ &\quad - \sum_{u\in\mathcal{U}} \sum_{v\in\mathcal{U}, u>v} \mathbf{PRG}(s_{u,v}) \\ &= \sum_{u\in\mathcal{U}} \mathbf{x}_u \pmod R,\end{aligned}$$

where all $\mathbf{PRG}(s_{u,v})$ and $\mathbf{PRG}(s_{v,u})$ generated by user u and v cancel each other out.

In the case of aggregating $\bar{\mathbf{y}}_u$, the following condition holds.

$$\begin{aligned}\sum_{u\in\mathcal{U}} \bar{\mathbf{y}}_u &= \sum_{u\in\mathcal{U}} F(\mathbf{x}_u) \\ &= \sum_{u\in\mathcal{U}} \mathbf{a} \circ \mathbf{x}_u + \sum_{u\in\mathcal{U}} \mathbf{b} \\ &= \mathbf{a} \circ \sum_{u\in\mathcal{U}} \mathbf{x}_u + |\mathcal{U}| \cdot \mathbf{b} \pmod R.\end{aligned}$$

5.3.2 Soundness

Intuitively, VERSA guarantees soundness when the server can convince users if and only if it returns a correct aggregated gradient. The server cannot generate a valid proof (i.e., $\bar{\mathbf{z}}$) without running VERSA correctly.

To formally capture the soundness property of VERSA, we use a game-based security model in which an adversary (who tries to break the scheme) interacts with a challenger (who runs the scheme). The adversary is the server, and the challenger is an entity representing the (surviving) users. Assuming that the *PRG* is secure, we demonstrate that VERSA is secure. To do this, we define a soundness game in which the adversary is given a set of pairs $\{(\mathbf{y}_u, \bar{\mathbf{y}}_u)\}_{u\in\mathcal{U}}$ from the challenger. The adversary returns $(\mathbf{z}, \bar{\mathbf{z}})$ attesting to the aggregation being done correctly. The adversary goal is to induce the challenger to accept a false pair $(\mathbf{z}^*, \bar{\mathbf{z}}^*)$. In this setting, we put a restriction on the challenger to verify $(\mathbf{z}^*, \bar{\mathbf{z}}^*)$ by running the *Validating Output* phase of VERSA, not by aggregating $\{(\mathbf{y}_u, \bar{\mathbf{y}}_u)\}_{u\in\mathcal{U}}$ from scratch. This restriction is reasonable because, in a cross-device FL setting, no entity accesses all pairs $\{(\mathbf{y}_u, \bar{\mathbf{y}}_u)\}_{u\in\mathcal{U}}$, except for the server (the adversary in the game). The soundness game is as follows:

Soundness Game:

- *Setup:* The challenger $\mathcal{C}$ runs the *Setup* to ensure users agree on the cryptographic gadgets and security parameter λ, number of users n, threshold t, and public parameter $pp \leftarrow \mathbf{KA.param}(\lambda)$. On behalf of the users, $\mathcal{C}$ runs the *Advertising Key* and *Sharing Key* with the server. Specifically, all users use $\mathcal{C}$ as a *proxy* that relays from or to users. In addition, $\mathcal{C}$ returns all public parameters and output values of the *Advertising Key* and *Sharing Key* to $\mathcal{A}$.
- *Query:* $\mathcal{A}$ adaptively makes the *Masking Input* queries. It sends a user set $\mathcal{U}^*$ of its choice to $\mathcal{C}$, where $t \le |\mathcal{U}^*|$. Moreover, $\mathcal{C}$ runs the *Masking Input* with all users in $\mathcal{U}^*$ to obtain a set of masked input vectors $\{(\mathbf{y}_u, \bar{\mathbf{y}}_u)\}_{u\in\mathcal{U}^*}$. Then, $\mathcal{C}$ returns this set to $\mathcal{A}$.

  Next, $\mathcal{A}$ continues to query $\mathcal{C}$ for the set of masked input vectors that correspond to the user set $\mathcal{U}^*$ of its choice. $\mathcal{A}$ and $\mathcal{C}$ record all sets of masked input vectors and the user sets in order.
- *Challenge:* $\mathcal{C}$ chooses a user set $\mathcal{U}^*$ and requests an aggregation of all masked input vectors corresponding to $\mathcal{U}^*$.
- *Forge:* $\mathcal{A}$ computes and returns a pair of vectors $(\mathbf{z}^*, \bar{\mathbf{z}}^*)$. If $\mathcal{C}$ accepts $\mathbf{z}^*$ after running the *Validating Output*, then $\mathcal{A}$ wins the game; otherwise, $\mathcal{A}$ loses the game.

We then have the following definition.

Definition 1. *VERSA is sound if the probability that any probabilistic polynomial time adversary wins the soundness game on an arbitrary set of users is negligible.*

6 SECURITY ANALYSIS

In this section, we analyze the security of VERSA regarding the gradient privacy and soundness. In terms of gradient privacy, any attempt to leak gradients in VERSA is reduced to breaking the underlying gradient encryption protocol, i.e., SA, because VERSA runs on top of SA to encrypt gradients. Because SA hides all information about users' individual gradients except their sum and because the user never decrypts the ciphertexts outside his local storage throughout the protocol, the privacy of the individual gradient is preserved in the presence of the server and any subset $\mathcal{U}' \subset \mathcal{U}$ of users.

In terms of soundness, to prove that the server can convince users if and only if it performs aggregation correctly, we rely on the security of the *PRG*. More precisely, *PRG* guarantees that its output on a seed chosen uniformly at random is computationally indistinguishable from an element of the output space sampled uniformly at random, as long as the seed is hidden from the distinguisher.

Theorem 1. *Provided that* PRG *is secure, VERSA guarantees soundness in the random oracle model.*

Proof. Throughout the proof, we reduce the security of the proposed scheme to the security of the *PRG*. Specifically, we assume that adversary $\mathcal{A}$ wins the Soundness Game with nonnegligible probability ϵ. Then, we show how another adversary $\mathcal{B}$ uses $\mathcal{A}$ to break the security of the *PRG*. Without loss of generality, we assume that dropouts do not occur in favor of the simplicity of proof. However, our proof can easily be extended to consider dropouts by employing the *SS.recon* algorithm. We model an instance of the *PRG* as $\mathcal{O}_P$ and a random oracle $\mathcal{O}_R$ such that $\mathcal{O}_P$ and $\mathcal{O}_R$ are in charge of responding oracle queries to *PRG* and truly random sequences, respectively. Note that $\mathcal{B}$ does not communicate with the oracles directly. Instead, we model $\mathcal{C}$ as an oracle proxy that receives oracle queries from $\mathcal{B}$, flips a coin $c \in \{P, R\}$, and relays oracle queries to $\mathcal{O}_c$. Then, $\mathcal{B}$ interacts with $\mathcal{A}$ as follows:

Setup: $\mathcal{B}$ chooses a *KA* protocol, *SS* protocol, and *AE* and sets the security parameter λ, number of users n, and threshold t. Lastly, $\mathcal{B}$ obtains $pp \leftarrow \mathbf{KA}.\mathbf{param}(\lambda)$ to ensure users agree on cryptographic gadgets and λ, n, t, and pp.

Let $u \in \{1, \ldots, n\}$ be a numerical identifier of user $u \in \mathcal{U}$, where $|\mathcal{U}| = n$. In addition, $\mathcal{B}$ obtains two key pairs $(pk_u^0, sk_u^0) \leftarrow \mathbf{KA}.\mathbf{gen}(pp)$, $(pk_u^1, sk_u^1) \leftarrow \mathbf{KA}.\mathbf{gen}(pp)$. Moreover, $\mathcal{B}$ obtains $\{sk_{u,v}^1\}_{v\in\boldsymbol{\mathcal{U}}_1} \leftarrow \mathbf{SS}.\mathbf{share}(sk_u^1, t, \mathcal{U}_1)$. Then, $\mathcal{B}$ samples $b_u \in_R \mathbb{R}$ and obtains $\{b_{u,v}\}_{v\in\boldsymbol{\mathcal{U}}_1} \leftarrow \mathbf{SS}.\mathbf{share}(\mathbf{b_u}, \mathbf{t}, \boldsymbol{\mathcal{U}}_\mathbf{1})$. For all users $u, v \in \mathcal{U}, u \neq v$, $\mathcal{B}$ obtains $ct_{u,v} \leftarrow \mathbf{AE}.\mathbf{enc}(\mathbf{KA}.\mathbf{agree}(sk_u^0, pk_v^0), sk_{u,v}^1 || b_{u,v})$. Finally, $\mathcal{B}$ returns $(n, t, pp, \{(pk_u^0, pk_u^1)\}_{u\in\mathcal{U}}, \{ct_{u,v}\}_{u,v\in\mathcal{U}})$ and the descriptions of **KA**, **SS**, and **AE** to $\mathcal{A}$.

Query: $\mathcal{A}$ adaptively makes the *Masking Input* queries: $\mathcal{A}$ selects a set of vectors of the same dimension $F = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ and sends F to $\mathcal{B}$. Then, $\mathcal{B}$ encrypts F and sends $\{\mathbf{y}_u, \bar{\mathbf{y}}_u\}_{u\in\mathcal{U}}$ back to $\mathcal{A}$, where each pair $(\mathbf{y}_u, \bar{\mathbf{y}}_u)$ corresponds to $\mathbf{x}_u$. Further, $\mathcal{A}$ continues to query $\mathcal{B}$ for the vector sets F' of its choice. More precisely, $\mathcal{B}$ responds to $\mathcal{A}$'s queries as follows:

- If $\mathcal{A}$ makes a query for F that has not been made, then $\mathcal{B}$ makes oracle queries to $\mathcal{C}$ as follows:
  - (i) $\mathcal{B}$ requests two vectors of the same dimension.
  - (ii) $\mathcal{C}$ flips a coin $c \in \{P, R\}$ and relays the query to $\mathcal{O}_c$.
  - (iii) $\mathcal{O}_c$ responds with $V_{\mathcal{U}} = (\mathbf{a}, \mathbf{b})$, where $\mathbf{a}$ and $\mathbf{b}$ are of the same dimensional vectors, and records the tuple $(\mathcal{U}, V_{\mathcal{U}})$. Then, $\mathcal{C}$ records the tuple $(c, \mathcal{U}, V_{\mathcal{U}})$.
  - (iv) For each user $u \in \mathcal{U}$, $\mathcal{B}$ requests to $\mathcal{C}$ for $4 \cdot (|\mathcal{U}| - 1)$ vectors of the same dimension.
  - (v) $\mathcal{C}$ relays the query to $\mathcal{O}_c$, which responds with $V_u = \{\mathbf{c}_{u,v}, \mathbf{d}_{u,v}, \mathbf{e}_{u,v}, \mathbf{f}_{u,v}\}_{v\in\mathcal{U}, u\neq v}$. For all $v < n$, $\mathcal{O}_c$ looks up the recorded tuples $(1, V_1)$, $\ldots, (u-1, V_{u-1})$ and sets $\mathbf{c}_{u,v} = \mathbf{d}_{v,u}$ and $\mathbf{e}_{u,v} = \mathbf{f}_{v,u}$. Then, $\mathcal{O}_c$ records the tuple (u, V_u), and $\mathcal{C}$ records the tuple (c, u, V_u).
- If $\mathcal{A}$ made a query for the same F previously, then $\mathcal{B}$ requests to $\mathcal{C}$, which retrieves the recorded tuple $(c, \mathcal{U}, V_{\mathcal{U}}, \{u, V_u\}_{u\in\mathcal{U}})$ and returns (V, V_u).
- For each user $u \in \mathcal{U}$, $\mathcal{B}$ computes $G(\mathbf{x}_u) = \mathbf{a} \circ \mathbf{x}_u + \mathbf{b}$ and computes $(\mathbf{y}_u, \bar{\mathbf{y}}_u)$ as follows:

$$\mathbf{y}_u = \mathbf{x}_u + \sum_{v\in\boldsymbol{\mathcal{U}}, u<v} \mathbf{c}_{u,v} - \sum_{v\in\boldsymbol{\mathcal{U}}, u>v} \mathbf{d}_{u,v} \pmod{R},$$
$$\bar{\mathbf{y}}_u = G(\mathbf{x}_u) + \sum_{v\in\boldsymbol{\mathcal{U}}, u<v} \mathbf{e}_{u,v} - \sum_{v\in\boldsymbol{\mathcal{U}}, u>v} \mathbf{f}_{u,v} \pmod{R}.$$

Lastly, $\mathcal{B}$ sends $\{\mathbf{y}_u, \bar{\mathbf{y}}_u\}_{u\in\mathcal{U}}$ to $\mathcal{A}$.

Challenge: $\mathcal{B}$ requests from $\mathcal{A}$ the aggregation of a certain vector set F that $\mathcal{B}$ previously received from $\mathcal{A}$.

Forge: $\mathcal{A}$ generates and sends a proof $(\mathbf{z}^*, \bar{\mathbf{z}}^*)$ to $\mathcal{B}$. Then, $\mathcal{B}$ checks the validity of $(\mathbf{z}^*, \bar{\mathbf{z}}^*)$. Because we assume that $\mathcal{A}$ wins the Soundness Game with non-negligible probability ϵ, $(\mathbf{z}^*, \bar{\mathbf{z}}^*)$ is a valid proof with the same probability. If $\bar{\mathbf{z}}^* = \mathbf{a} \circ \mathbf{z}^* + n \cdot \mathbf{b} \pmod{R}$, then $\mathcal{B}$ returns P, indicating that all the vectors $\mathcal{B}$ received are derived from the instance of the *PRG*; otherwise $\mathcal{B}$ returns R, indicating that all the vectors are derived in a truly random manner.

The proof further proceeds as follows. If the vectors are derived from the instance of the *PRG*, then $\mathcal{B}$ correctly simulated the *Masking Input* queries from $\mathcal{A}$. As the view of $\mathcal{A}$ in the simulation is identical to its view in the Soundness Game, the probability of $\mathcal{B}$ distinguishing between the outputs of the *PRG* and a truly random generator is reduced to the probability of $\mathcal{A}$ winning the Soundness Game. Specifically, the probability ϵ of $\mathcal{A}$ to win this game is reduced to the probability of $\mathcal{A}$ to recover $(\mathbf{a}, \mathbf{b})$ which is derived from the instance of the *PRG*. Therefore, we have

$$\Pr[\mathcal{B}(\mathcal{O}_P, \mathcal{O}_R) = P] = \frac{1}{2} + \epsilon.$$

Even when the vectors are not derived from the instance of the *PRG*, it still holds that $\mathcal{B}$ has simulated the *Masking Input* queries of $\mathcal{A}$ correctly. However, in this case, $\mathcal{B}$ cannot exploit the advantage of $\mathcal{A}$ because the probability of $\mathcal{A}$ to win this game is reduced to the probability of the ability of $\mathcal{A}$ to recover $(\mathbf{a}, \mathbf{b})$ which is derived in a truly random manner. Thus, the probability of $\mathcal{B}$ distinguishing between the outputs of the *PRG* and a truly random generator is no better than flipping a coin. Therefore, we have

$$\Pr[\mathcal{B}(\mathcal{O}_P, \mathcal{O}_R) = R] = \frac{1}{2}.$$

This concludes the proof. □

7 Evaluation

In this section, we evaluate VerSA compared to SA [37] and VerifyNet [30], which are the most relevant state-of-the-art proposals for privacy-preserving cross-device FL. We use these two schemes as a baseline to illustrate how efficiency can be improved while achieving a higher level of security.

7.1 Implementation Setup

We measured the computation time for four phases: the *sharing key*, *masking input*, *unmasking input and returning output*, and *validating output*. We ignored nondominant costs, such as running the *setup* and *advertising key*, which is considered a one-time cost. The experiment was conducted using Java on a desktop machine with a 3.00GHz Intel Core i7-9700 processor and 16 GB RAM. We used elliptic-curve Diffie-Hellman, t-out-of-n Shamir secret sharing, Advanced Encryption Standard Galois/Counter Mode with a 128-bit private key, and SHA-256 to implement the *KA*, *SS*, *AE*, and *PRG*, respectively. We set $t = 10$ for *SS*. We used randomly generated 10K-entry vectors with 64-bit for each entry,

TABLE 2
Comparative Analysis Result

	SA		VerifyNet		VERSA	
	$n = 500$	$n = 1,000$	$n = 500$	$n = 1,000$	$n = 500$	$n = 1,000$
			0% dropouts			
Sharing Key	516 ms	690 ms	526 ms	683 ms	531 ms	766 ms
Masking Input	6,842 ms	13,941 ms	801,587 ms	825,599 ms	13,157 ms	26,711 ms
Unmasking Input and Returning Output	101 ms	218 ms	919,699 ms	1,885,900 ms	104 ms	208 ms
Validating Output	N/A	N/A	1,245,452 ms	1,206,534 ms	16 ms	18 ms
			10% dropouts			
Sharing Key	547 ms	703 ms	526 ms	683 ms	562 ms	736 ms
Masking Input	6,983 ms	14,029 ms	806,534 ms	840,345 ms	13,366 ms	26,209 ms
Unmasking Input and Returning Output	289,955 ms	604,929 ms	1,215,335 ms	2,534,754 ms	586,571 ms	1,240,015 ms
Validating Output	N/A	N/A	1,225,234 ms	1,203,572 ms	16 ms	18 ms
			20% dropouts			
Sharing Key	562 ms	722 ms	563 ms	697 ms	559 ms	719 ms
Masking Input	7,262 ms	14,837 ms	812,653 ms	852,769 ms	13,318 ms	26,509 ms
Unmasking Input and Returning Output	524,155 ms	1,210,150 ms	1,455,922 ms	2,954,534 ms	1,035,677 ms	2,346,821 ms
Validating Output	N/A	N/A	1,301,652 ms	1,304,239 ms	15 ms	18 ms

while varying the number of users and user dropout ratio, to gain a general perspective on how the two varying factors affect the performance of the proposed four phases. To implement VerifyNet, we utilized a pairing-based cryptography library written in Java [63]. The experiment was performed on two user groups, consisting of 500 and 1,000 users, respectively.

7.2 Experimental Result

We provide the comparative experimental result in Table 2. The table presents the overall performance of three schemes while varying the number of users and the user dropout ratio. In the *sharing key* phase, we did not observe any noticeable performance gap among the three schemes. In the *masking input* phase, SA exhibits the best efficiency over VerifyNet and VERSA because SA only supports the privacy preservation of gradients, whereas the others support verification of aggregate gradients as a supplementary functionality. The cost of VERSA approximately doubles the cost of SA. VERSA performs vector expansion though the *PRG* twice compared to SA, where the vector expansion is the computationally dominant operation in SA and VERSA.

In contrast, VerifyNet incurs significant costs compared with both SA and VERSA. This is because of the extensive usage of group operations in VerifyNet. The computation cost of these group operations overwhelms that of the vector expansion through the *PRG*. In the *unmasking input and returning output* phase, VERSA and SA show the similar cost when the dropouts do not occur. However, these schemes exhibit a noticeable cost difference when the dropouts occur because the server reconstructs these vectors twice compared with SA. The cost of VerifyNet overwhelms the costs of SA and VERSA for the same reason in the *masking input* phase. Lastly, in the *validating output* phase, VerifyNet and VERSA incur a constant cost irrespective of the number of users and the user dropout ratios. However, the cost of VERSA is orders of magnitude smaller than that of VerifyNet. The main reason for such a performance gap between two schemes is that VERSA uses only a computationally lightweight *PRG* operations, whereas VerifyNet performs pairing operations, which are much more intensive than the *PRG* in terms of computation.

7.3 Accuracy

7.3.1 Dataset

We conducted an evaluation using the following three datasets:

- MNIST [40] is a dataset of grayscale images of handwritten numbers from 0 to 9, consisting of 60,000 training, and 10,000 testing images. Each image has a size of $28 \times 28 \times 1$ and one of 10 labels.
- SVHN [41] is a dataset of RGB image of house numbers obtained from Google Street View. Similar to MNIST, it contains images of small, cropped digits but incorporates much more labeled data from significantly more challenging and unresolved real-world problems. It is divided into 73,257 digits for training and 26,032 digits for testing. This dataset has 10 classes for each number, and each image size is $32 \times 32 \times 3$.
- CIFAR100 [42] is a collection of photos of 100 objects in RGB. It has 100 classes containing 500 training images and 100 testing images per class, each of size $32 \times 32 \times 3$.

The characteristics of the above three datasets are summarized in Table 3.

7.3.2 Neural Networks

Each dataset was trained using different models. For the MNIST dataset, we used a three-layer network with two

TABLE 3
Dataset

Dataset	# Training data	# Test data	# Label	Image size
MNIST	60,000	10,000	10	28x28x1
SVHN	73,257	26,032	10	32x32x3
CIFAR100	50,000	10,000	100	32x32x3

hidden, fully connected layers with 256 neurons and rectified linear units. The output layer is fully connected with 10 output neurons and softmax activation. For the SVHN and CIFAR100 datasets, a convolution neural network, consisting of seven convolutional layers with 3×3 filters and a stride of 1, was used. Each convolutional layer was followed by rectified linear units and 2×2 max pooling with a stride of 2. The fully connected layer used softmax activation.

For the three datasets, we also used ResNet[64], a model used in real-world environments. ResNet consists of 50 convolution layers, rectified linear units and 1 fully connected layer with 2,048 neurons. Originally used for ImageNet, we reduced the number of output neurons from 1K to the number of labels for each dataset.

The stochastic gradient descent optimizer was used for the MNIST dataset with a learning rate of 0.001 and the Adam optimizer was used for the simple convolution neural networks with a learning rate of 0.001. In terms of ResNet, we used stochastic gradient descent optimizer again to follow the training method in [65], i.e., train at different learning rates for each epoch.

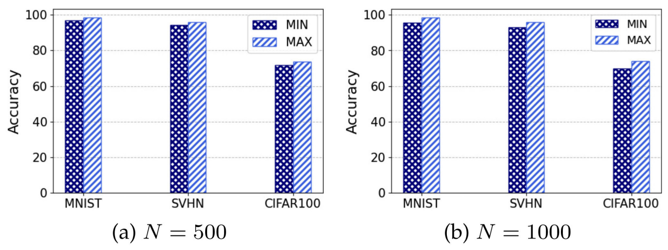
The gradient size of each dataset's output of the fully connected layer was calculated by multiplying the input and output size, adding the bias size, as listed in Table 4.

7.3.3 Default Model Accuracy

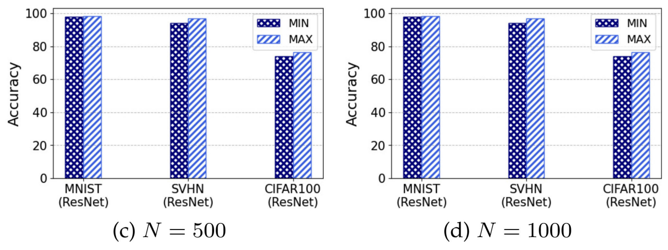
In the experiment, N models were trained, where each model was trained by an individual user with 1,000 randomly selected data from each dataset. Each user received a pre-trained model and trained only the fully-connected layer while holding the parameters of the convolution layers. Then, the accuracy was evaluated for each test set, where $N \in \{500, 1000\}$. The minimum and maximum accuracy value of N models are displayed in Fig. 5. The figures plot the model accuracy evaluated by each user. The minimum/maximum accuracy difference becomes slightly greater as the number of trained models increases from 500 to 1,000. We provide the model accuracy results in these figures as a baseline to measure the FL accuracy.

TABLE 4
Gradient Size of Fully Connected Layer

Dataset	Model	In × Out	Bias	Gradient size
MNIST	DNN	256×10	10	2,570
MNIST	ResNet	$2,048 \times 10$	10	20,490
SVHN	CNN	256×10	10	2,570
SVHN	ResNet	$2,048 \times 10$	10	20,490
CIFAR100	CNN	$1,024 \times 100$	100	102,500
CIFAR100	ResNet	$2,048 \times 100$	100	204,900



(a) $N = 500$ (b) $N = 1000$



(c) $N = 500$ (d) $N = 1000$

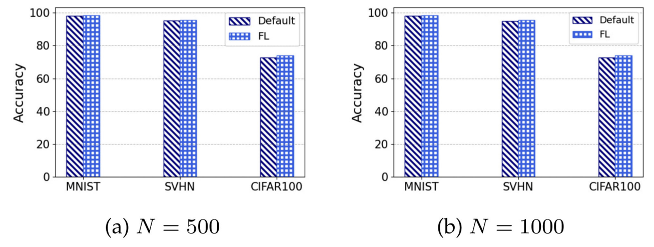
Fig. 5. Min/Max accuracy (%) of models.

7.3.4 FL Accuracy

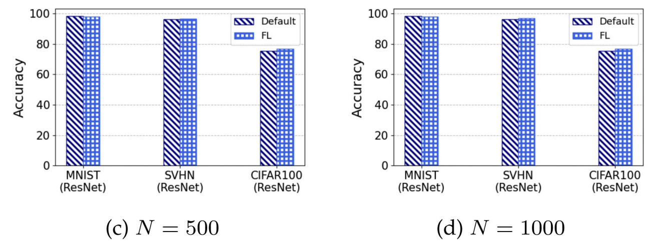
We compared the accuracy of N default models with that of FL in Fig. 6. From the experimental result, we did not observe any noticeable accuracy difference between the two groups of models. Nevertheless, the accuracy of FL is higher than that of the default models. Although the tested accuracy was measured only on the four types of model, rooms still exist to advocate that FL is slightly better than the default models.

7.3.5 Accuracy Comparison

The gradient entries should be in the form of an integer to encrypt the gradient of the model. Because all gradient entries are decimal numbers, we transformed the numbers to integers following the float-to-integer conversion method [52], [53]. We quantized a floating-point value v into an integer $\lfloor v \cdot \alpha \rfloor$ with a scaling factor α, where larger values of α leads to low quantization errors. In the experiment, we set $\alpha = 10^x$, where x is called a *walk*, and each walk is a positive integer. Such a transformation has a trade-off between computational overhead and accuracy. If we set a larger

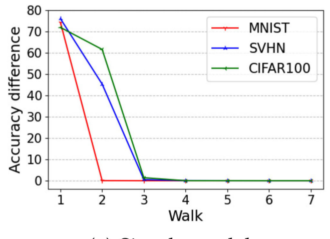


(a) $N = 500$ (b) $N = 1000$

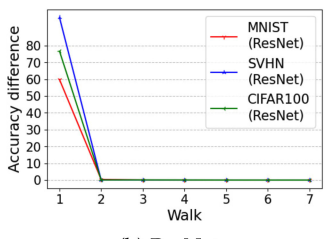


(c) $N = 500$ (d) $N = 1000$

Fig. 6. Comparison of the accuracy between default models and FL.



(a) Simple model



(b) ResNet

Fig. 7. Accuracy difference between floating-point type FL and integer type FL models.

walk, then we have greater computation time. In contrast, if we set a smaller walk, then the accuracy of the FL model is reduced.

Fig. 7 demonstrates the accuracy difference between the floating-point type FL and float-to-integer transformed FL models. The accuracy of floating-point type FL models is always greater than or equal to that of the integer type FL models. When we set walk less than 3, the accuracy of the floating-point type FL model is always higher than that of the float-to-integer transformed FL model. However, when we increased the walk, we observed no significant difference between the two types of models. Moreover, when we set walk larger than 6, no accuracy difference occurred at all. In Fig. 8, we plotted the details of the effect of the walk on accuracy. As we used a smaller walk (e.g., $walk = 1$), the accuracy decreased significantly compared with the floating-point type FL models. However, such an accuracy gap was drastically reduced when $walk \geq 3$.

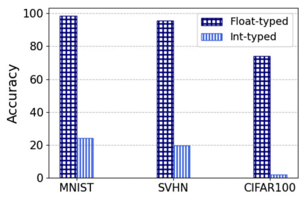
8 Related Work

We describe previous studies for secure FL in two main approaches (i.e., privacy preservation and verifiable computation).

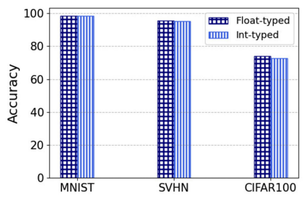
8.1 Privacy-Preserving Deep Learning

Training a machine learning model on a third-party provider, such as machine learning as a service, may leak sensitive training data despite its promising aspect. For example, neural network parameters trained locally on the user side could be exploited to leak sensitive training examples [7], [8], [10]. Some proposals employ differential privacy techniques over gradients obtained by the stochastic gradient descent rule to protect against training data leakage attacks [9], [20]. However, these schemes rely on a trusted party who can access the training data to add noise.

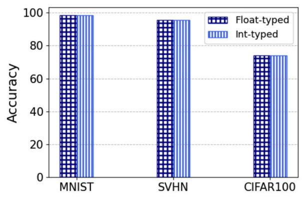
Bonawitz *et al.* [37] proposed an SA protocol for the cross-device federated learning settings. In SA, the local model parameters are encrypted while a central server can derive a global, aggregated model parameter from the encrypted local model parameters without decryption. The SA uses a *PRG* to mask model parameters, making it best suitable for encrypting the model parameters of high-dimensional vectors in resource-constrained user devices. Moreover, the SA is resilient to dropouts by users, which would occur frequently in a cross-device setting. Some prior work has relied on homomorphic encryption for privacy preservation during training [21], [22], yielding impractical performance concerns.



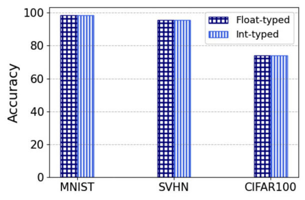
(a) Walk = 1



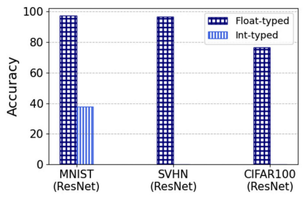
(b) Walk = 3



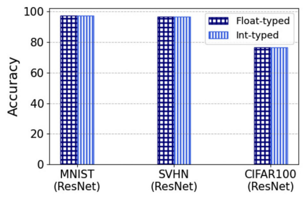
(c) Walk = 6



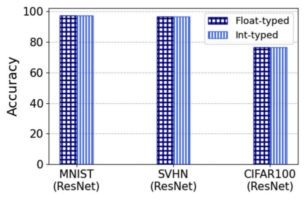
(d) Walk = 7



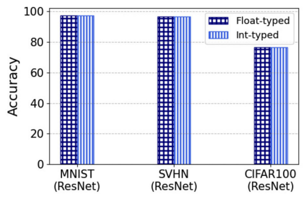
(e) Walk = 1 / ResNet



(f) Walk = 3 / ResNet



(g) Walk = 6 / ResNet



(h) Walk = 7 / ResNet

Fig. 8. Accuracy comparison of float-typed and int-typed FL models.

Sav *et al.* [23] recently proposed POSEIDON, a privacy-preserving FL protocol. Their scheme highly relies on a multiparty variant of homomorphic encryption, and the user dropout issue remains untouched, both of which we consider crucial in cross-device FL. Therefore, we designed VERSA especially for SA, to inherit all the beneficial properties of SA.

8.2 Deep Learning with Verifiable Computation

8.2.1 Verifiability in Federated Learning

Only a few studies have addressed verifiability in the context of FL. Recently, Xu *et al.* [30] proposed VerifyNet, an SA protocol with verifiable computation. VerifyNet is similar to VERSA for the two reasons. First, both schemes are designed to run on top of SA for the cross-device FL settings [37]. Second, they aim to provide the verifiability of global model parameters in the training phase. In VerifyNet, all users must share a secret value that is used to generate an evaluation key for a parameter. Each user runs a local training model and encodes the output parameter to an evaluation key. The server receives the encrypted local model parameters and evaluation keys and returns the proof by aggregating all evaluation keys and computing the sum of all parameters. The user verifies the correctness of the sum using the proof.

VerifyNet has two limitations. First, the user-side computational cost is too extensive to process high-dimensional data. Specifically, for each entry in a gradient, VerifyNet requires six exponentiations in a cyclic group $\mathbb{G}$ to generate an evaluation key and four pairings and one exponentiation to verify the result. Second, VerifyNet could allow malicious users to learn a victim's local model parameters. As we show in Section 4, the attack results on VerifyNet demonstrate that an adversary can recover most of the victim's local model parameters in a reasonable time.

Guo *et al.* [66] recently proposed VERIFL to reduce the communication cost of VerifyNet when processing high-dimensional gradient vectors. To achieve verifiability in FL, VERIFL employs a homomorphic hash in the same way as VerifyNet, i.e., all users in VERIFL share the secret key of the homomorphic hash. Thus, as our model recovery attack on VerifyNet exploits the shared key, it is straightforward that VERIFL is also vulnerable to the attack. Fu *et al.* [32] proposed VFL, a verifiable FL protocol for cross-device settings. However, similar to VerifyNet and VERIFL, VFL relies on the assumption that all users share a predefined secret parameter in advance to verify the global model parameters, which is not practical in cross-device federated learning settings.

While VerSA and prior work [30], [32], [66] aim at achieving the verifiability of aggregated models under the untrusted central server, several works addressed it under different threat models. Specifically, Zhang *et al.* [67] showed how to guarantee the correctness of aggregated models from misbehaving users who may not execute training tasks as intended. Peng *et al.* [68] addressed the setting where the role of the single central server is distributed to multiple parties. They then achieved the verifiability of aggregated models, provided that a majority of parties are honest.

8.2.2 *Verifiability in Various Settings*

In addition to the verifiable FL schemes described above, several proposals have introduced different verifiable machine learning approaches. Ghodsi *et al.* [52] proposed Safetynets, a neural net framework for verifying the correctness of inference tasks in a single client-server setting. Xu *et al.* [69] demonstrated how to support verifiability for inference tasks using a homomorphic encryption. Tramèr *et al.* [70] proposed Slalom to verify machine learning tasks employing a trusted execution environment. Slalom is faster than computationally expensive cryptographic technique-based approach like [69], but it highly relies on the trust assumption concerning chip vendors, such as Intel's software guard extensions [71]. Niu *et al.* [72] proposed a verifiable inference model protocol for a typical client-server setting, but its computational workload is extensive due to the extensive usage of bilinear pairings.

9 Conclusion

In this paper, we studied the problem of verifying the model aggregation of a neural network in the presence of a federation of users and an untrusted central server. We designed a novel attack against prior work and demonstrated that local model parameters are revealed within a reasonable time (e.g., 98% of encrypted local model parameters sent to the central server during the training phase are recovered with 21 h), violating privacy. Then, we proposed VerSA, a verifiable secure aggregation protocol for cross-device federated learning. VerSA effectively supports privacy-preserving model aggregation and verifiability, even when the model parameters are high-dimensional vectors, in a communication-efficient and failure-robust manner. VerSA does not require any trusted setup between users while enabling both the central server and users to use only a *PRG* to prove and verify the correctness of model aggregation. We experimentally confirmed the efficiency of VerSA under three datasets (MNIST, SVHN, and CIFA100), demonstrating VerSA is orders of magnitude faster than models in prior work.

Acknowledgments

The authors would like to thank the reviewers for their insightful feedback.

References

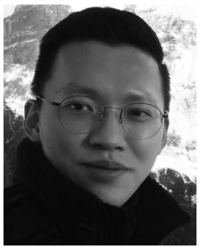
[1] L. Deng and D. Yu, "Deep learning: Methods and applications," *Found. Trends Signal Process.*, vol. 7, no. 3–4, pp. 197–387, 2014.
[2] S. Khan and T. Yairi, "A review on the application of deep learning in system health management," *Mech. Syst. Signal Process.*, vol. 107, pp. 241–265, 2018.
[3] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, "State-of-the-art in artificial neural network applications: A survey," *Heliyon*, vol. 4, no. 11, 2018, Art. no. e00938.
[4] I. Stoica *et al.*, "A berkeley view of systems challenges for AI," 2017, *arXiv:1712.05855*.
[5] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, "Federated optimization: Distributed machine learning for on-device intelligence," 2016, *arXiv:1610.02527*.
[6] H. B. McMahan, E. Moore, D. Ramage, and B. A. y Arcas, "Federated learning of deep networks using model averaging," 2016, *arXiv:1602.05629*.
[7] M. Fredrikson, S. Jha, and T. Ristenpart, "Model inversion attacks that exploit confidence information and basic countermeasures," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1322–1333.
[8] R. Shokri and V. Shmatikov, "Privacy-preserving deep learning," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 1310–1321.
[9] M. Abadi *et al.*, "Deep learning with differential privacy," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 308–318.
[10] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," in *Proc. IEEE Symp. Secur. Privacy.*, 2017, pp. 3–18.
[11] B. Hitaj, G. Ateniese, and F. Perez-Cruz , "Deep models under the GAN: Information leakage from collaborative deep learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 603–618.
[12] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru , and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in *Proc. IEEE Symp. Secur. Privacy.*, 2018, pp. 19–35.
[13] N. Carlini, C. Liu, Ú. Erlingsson, J. Kos, and D. Song, "The secret sharer: Evaluating and testing unintended memorization in neural networks," in *Proc. 28th* USENIX *Secur. Symp.*, 2019, pp. 267–284.
[14] L. Melis, C. Song, E. De Cristofaro , and V. Shmatikov, "Exploiting unintended feature leakage in collaborative learning," in *Proc. IEEE Symp. Secur. Privacy.*, 2019, pp. 691–706.
[15] M. Nasr, R. Shokri, and A. Houmansadr, "Comprehensive privacy analysis of deep learning: Passive and active white-box inference attacks against centralized and federated learning," in *Proc. IEEE Symp. Secur. Privacy.*, 2019, pp. 739–753.
[16] Z. Wang, M. Song, Z. Zhang, Y. Song, Q. Wang, and H. Qi, "Beyond inferring class representatives: User-level privacy leakage from federated learning," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2019, pp. 2512–2520.

[17] L. Zhu and S. Han, "Deep leakage from gradients," in *Proc. Federated Learn.*, 2020, pp. 17–31.
[18] H. B. McMahan, D. Ramage, K. Talwar, and L. Zhang, "Learning differentially private recurrent language models," 2017, *arXiv: 1710.06963*.
[19] W. Li *et al.*, "Privacy-preserving federated brain tumour segmentation," in *Proc. Int. Workshop Mach. Learn. Med. Imag.*, 2019, pp. 133–141.
[20] L. Yu, L. Liu, C. Pu, M. E. Gursoy, and S. Truex, "Differentially private model publishing for deep learning," in *Proc. IEEE Symp. Secur. Privacy.*, 2019, pp. 332–349.
[21] E. Hesamifard, H. Takabi, M. Ghasemi, and R. N. Wright, "Privacy-preserving machine learning as a service," *Proc. Privacy Enhancing Technol.*, vol. 2018, no. 3, pp. 123–142, 2018.
[22] K. Nandakumar, N. Ratha, S. Pankanti, and S. Halevi, "Towards deep neural network training on encrypted data," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit. Workshops*, 2019, pp. 40–48.
[23] S. Sav *et al.*, "Poseidon: Privacy-preserving federated neural network learning," 2020, *arXiv:2009.00349*.
[24] P. Mohassel and Y. Zhang, "SecureML: A system for scalable privacy-preserving machine learning," in *Proc. IEEE Symp. Secur. Privacy.*, 2017, pp. 19–38.
[25] P. Mohassel and P. Rindal, "ABY3: A mixed protocol framework for machine learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 35–52.
[26] S. G. Choi, J. Katz, R. Kumaresan, and C. Cid, "Multi-client non-interactive verifiable computation," in *Proc. Theory Cryptogr. Conf.*, 2013, pp. 499–518.
[27] S. D. Gordon, J. Katz, F.-H. Liu, E. Shi, and H.-S. Zhou, "Multi-client verifiable computation with stronger security guarantees," in *Proc. Theory Cryptogr. Conf.*, 2015, pp. 144–168.
[28] C. Cachin, E. Ghosh, D. Papadopoulos, and B. Tackmann, "Stateful multi-client verifiable computation," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2018, pp. 637–656.
[29] X. Ma, F. Zhang, X. Chen, and J. Shen, "Privacy preserving multiparty computation delegation for deep learning in cloud computing," *Inf. Sci.*, vol. 459, pp. 103–116, 2018.
[30] G. Xu, H. Li, S. Liu, K. Yang, and X. Lin, "VerifyNet: Secure and verifiable federated learning," *IEEE Trans. Inf. Forensics Secur.*, vol. 15, pp. 911–926, Jul. 2019.
[31] W. Zheng, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Helen: Maliciously secure coopetitive learning for linear models," in *Proc. IEEE Symp. Secur. Privacy.*, 2019, pp. 724–738.
[32] A. Fu, X. Zhang, N. Xiong, Y. Gao, H. Wang, and J. Zhang, "VFL: A verifiable federated learning with privacy-preserving for big data in industrial IoT," *IEEE Trans. Ind. Inf.*, early access, Nov. 06, 2020, doi: 10.1109/TII.2020.3036166.
[33] P. Kairouz *et al.*, "Advances and open problems in federated learning," 2019, *arXiv:1912.04977*.
[34] S. Shi, X. Chu, K. C. Cheung, and S. See, "Understanding top-K sparsification in distributed deep learning," 2019, *arXiv:1911.08772*.
[35] M. N. Krohn, M. J. Freedman, and D. Mazieres, "On-the-fly verification of rateless erasure codes for efficient content distribution," in *Proc. IEEE Symp. Secur. Privacy,.*, 2004, pp. 226–240.
[36] D. Fiore, R. Gennaro, and V. Pastro, "Efficiently verifiable computation on encrypted data," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 844–855.
[37] K. Bonawitz *et al.*, "Practical secure aggregation for privacy-preserving machine learning," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 1175–1191.
[38] J. Chen, X. Pan, R. Monga, S. Bengio, and R. Jozefowicz, "Revisiting distributed synchronous SGD," 2016, *arXiv:1604.00981*.
[39] I. Goodfellow, Y. Bengio, A. Courville, and Y. Bengio, *Deep Learning*. Cambridge, MA, USA: MIT Press Cambridge, 2016.
[40] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[41] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning," *NIPS Workshop Deep Learn. Unsupervised Feature Learn.*, 2011. [Online]. Available: http://ufldl.stanford.edu/housenumbers/nips2011_housenumbers.pdf
[42] A. Krizhevsky, G. Hinton *et al.*, "Learning multiple layers of features from tiny images," Tech. Rep. Univ. Toronto, 2009.
[43] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, *arXiv:1610.05492*.
[44] K. Bonawitz *et al.*, "Towards federated learning at scale: System design," 2019, *arXiv:1902.01046*.
[45] M. J. Sheller *et al.*, "Federated learning in medicine: Facilitating multi-institutional collaborations without sharing patient data," *Sci. Rep.*, vol. 10, no. 1, pp. 1–12, 2020.
[46] C. Zhang, S. Li, J. Xia, W. Wang, F. Yan, and Y. Liu, "BatchCrypt: Efficient homomorphic encryption for cross-silo federated learning," in *Proc.* USENIX *Annu. Tech. Conf.*, 2020, pp. 493–506.
[47] A. Shamir, "How to share a secret," *Commun. ACM*, vol. 22, no. 11, pp. 612–613, 1979.
[48] F. Brezing and A. Weng, "Elliptic curves suitable for pairing based cryptography," *Des., Codes Cryptogr.*, vol. 37, no. 1, pp. 133–141, 2005.
[49] M. Abe and S. Fehr, "Perfect NIZK with adaptive soundness," in *Proc. Theory Cryptogr. Conf.*, 2007, pp. 118–136.
[50] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th* USENIX *Symp. Oper. Syst. Des. Implementation*, 2016, pp. 265–283.
[51] B. Lynn *et al.*, "PBC library manual 0.5. 11.," Stanford Univ., 2006. Accessed: 2021. [Online]. Available: http://crypto.stanford.edu/pbc/
[52] Z. Ghodsi, T. Gu, and S. Garg, "SafetyNets: Verifiable execution of deep neural networks on an untrusted cloud," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 4672–4681.
[53] L. T. Phong, Y. Aono, T. Hayashi, L. Wang, and S. Moriai, "Privacy-preserving deep learning via additively homomorphic encryption," *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 5, pp. 1333–1345, May 2018.
[54] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *Proc. Int. Conf. Mach. Learn.*, 2015, pp. 1737–1746.
[55] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 7675–7684.
[56] P. C. Van Oorschot and M. J. Wiener, "On diffie-hellman key agreement with short exponents," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 1996, pp. 332–343.
[57] P. Rogaway, "Authenticated-encryption with associated-data," in *Proc. 9th ACM Conf. Comput. Commun. Secur.*, 2002, pp. 98–107.
[58] J. Håstad, R. Impagliazzo, L. A. Levin, and M. Luby, "A pseudorandom generator from any one-way function," *SIAM J. Comput.*, vol. 28, no. 4, pp. 1364–1396, 1999.
[59] U. Maurer, "Modelling a public-key infrastructure," in *Proc. Eur. Symp. Res. Comput. Secur.*, 1996, pp. 325–350.
[60] R. Cramer, I. Damgård, and U. Maurer, "General secure multiparty computation from any linear secret-sharing scheme," in *Proc. Int. Conf. Theory Appl. Cryptographic Techn.*, 2000, pp. 316–334.
[61] P. Mukherjee and D. Wichs, "Two round multiparty computation via multi-key FHE," in *Proc. Annu. Int. Conf. Theory Appl. Cryptographic Techn.*, 2016, pp. 735–763.
[62] C. Gentry *et al.*, *A Fully Homomorphic Encryption Scheme*. Stanford, CA, USA: Stanford Univ. Stanford, 2009.
[63] A. De Caro and V. Iovino, "jPBC: Java pairing based cryptography," in *Proc. IEEE Symp. Comput. Commun.*, 2011, pp. 850–855.
[64] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 770–778.
[65] T. Devries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," 2017, *arXiv:1708.04552*.
[66] X. Guo *et al.*, "V eri FL: Communication-efficient and fast verifiable aggregation for federated learning," *IEEE Trans. Inf. Forensics Secur.*, vol. 16, pp. 1736–1751, Dec. 2020.
[67] X. Zhang, F. Li, Z. Zhang, Q. Li, C. Wang, and J. Wu, "Enabling execution assurance of federated learning at untrusted participants," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, 2020, pp. 1877–1886.
[68] Z. Peng *et al.*, "VFchain: Enabling verifiable and auditable federated learning via blockchain systems," *IEEE Trans. Netw. Sci. Eng.*, early access, Jan. 12, 2021, doi: 10.1109/TNSE.2021.3050781.
[69] G. Xu *et al.*, "Secure and verifiable inference in deep neural networks," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2020, pp. 784–797.
[70] F. Tramer and D. Boneh, "Slalom: Fast, verifiable and private execution of neural networks in trusted hardware," 2018, *arXiv:1806.03287*.

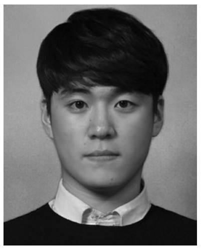
[71] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 2016, no. 86, pp. 1–118, 2016.

[72] C. Niu, F. Wu, S. Tang, S. Ma, and G. Chen, "Toward verifiable and privacy preserving machine learning prediction," *IEEE Trans. Dependable Secure Comput.*, early access, Nov. 03, 2020, doi: 10.1109/TDSC.2020.3035591.

Changhee Hahn received the BS and MS degrees in computer science from Chung-Ang University, Seoul, South Korea, in 2014 and 2016, respectively, and the PhD degree from the Department of Computer Science and Engineering, College of Informatics, Korea University, South Korea, in 2020. From 2020 to 2021, he was a postdoctoral researcher with Korea University. He is currently an assistant professor with the Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul, South Korea. His research interests include information security and cloud computing security.



Hodong Kim received the BS degree in computer science and engineering from Chung-Ang University, Seoul, South Korea, in 2017 and the MS degree in computer science from Korea University, Seoul, in 2020. He is currently working toward the PhD degree with the Department of Computer Science and Engineering, College of Informatics, Korea University, South Korea. His research interests include information security, cloud computing security, and microarchitectural security.



Minjae Kim received the BS degree in computer science from Korea University, Seoul, South Korea, in 2021. He is currently working toward the MS degree with the Department of Computer Science and Engineering, College of Informatics, Korea University, South Korea. His research interests include SafeAI and Blockchain.

Junbeom Hur received the BS degree in computer science from Korea University, Seoul, South Korea, in 2001, and the MS and PhD degrees in computer science from KAIST in 2005 and 2009, respectively. From 2009 to 2011, he was a postdoctoral researcher with the University of Illinois at Urbana-Champaign. From 2011 to 2015, he was an assistant professor with the School of Computer Science and Engineering, Chung-Ang University, South Korea. He is currently a professor with the Department of Computer Science and Engineering, the Korea University, South Korea. His research interests include information security, cloud computing security, network security, and applied cryptography.