ZKSL: Verifiable and Efficient Split Federated Learning via Asynchronous Zero-Knowledge Proofs

Yixiao Zheng*, Changzheng Wei*†✉, Xiaodong Qi*, Hanghang Wu†, Yuhan Wu*, Li Lin†,
Tianmin Song*, Ying Yan†, Yanqin Yang*‡, Zhao Zhang*‡✉, Cheqing Jin*‡, Aoying Zhou*‡
*East China Normal University, China
†Ant Digital Technologies, Ant Group, China
‡Engineering Research Center of Blockchain Data Management, Ministry of Education, Shanghai, China
{yxzheng, xdqi, yhwu, tmsong}@stu.ecnu.edu.cn, {yqyang, zhzhang, cqjin, ayzhou}@dase.ecnu.edu.cn
{changzheng.wcz, hanghang.whh, felix.ll, fuying.yy}@antgroup.com

Abstract*—In Vertical Federated Learning (VFL), prior work has primarily focused on protecting data privacy, while overlooking the risk that participants may manipulate local model execution to mount integrity attacks. Integrating zero-knowledge proofs (ZKPs) into the training process can ensure that each party's computations are verifiable without revealing private data. However, directly encoding deep model training as a monolithic ZKP circuit is impractical due to: (i) complex circuit design and high overhead from frequent parameter commitments, (ii) expensive proof generation for embeddings(cross-party information interface), and (iii) synchronous proof generation that blocks iterative training rounds. To address these challenges, we present ZKSL, an efficient and asynchronous VFL framework that achieves verifiable training under a malicious threat model. ZKSL partitions deep neural networks into layer-wise circuits and generates their proofs in parallel, ensuring input–output consistency via *Privacy-Commitment PLONK* (PC-PLONK), a lightweight extension that supports low-cost, iteration-by-iteration parameter commitments. For embedding layers, ZKSL adopts a probabilistic verification technique that reduces proof complexity from $O(Nnd)$ to $O(nd)$. Furthermore, ZKSL incorporates an asynchronous compute–prove scheduling mechanism to decouple proof generation from training iterations, effectively mitigating pipeline stalls. Experimental results on DeepFM and CNN models show that ZKSL reduces proof generation time by up to 73% while maintaining 99.4% accuracy, demonstrating superior scalability and practicality for real-world federated learning.

I. Introduction

Federated Learning (FL) enables decentralized entities to collaboratively train models without sharing raw data, in line with privacy regulations such as GDPR[1] and CCPA/CPRA[2]. Among its variants, *Vertical Federated Learning* (VFL) [3] considers organizations that hold different feature sets for the same group of users, a common situation in finance, healthcare, and digital marketing. A practical instantiation is *Split Vertical Federated Learning* (Split VFL) [4], where a deep neural network is partitioned across parties. Each participant computes local *embeddings* from its private data, and a coordinator aggregates these intermediate activations to complete the forward and backward passes. This design improves modularity and reduces communication by exchanging activations instead of full gradients or parameters.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.242008
www.ndss-symposium.org

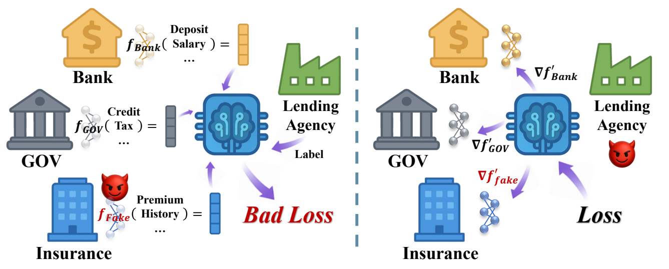


Fig. 1: Loan Example of Vertical Federated Learning

Despite these benefits, Split VFL faces security and trustworthiness challenges: malicious parties may bypass local computation, inject fabricated updates, or infer sensitive data via gradient inversion attacks [5–8], undermining both privacy and model integrity. As illustrated in Fig. 1, in a loan approval pipeline different institutions contribute complementary features for a shared user, yet a dishonest party can submit stale embeddings or tamper with gradients to gain strategic advantage. Existing work primarily protects data privacy using differential privacy [9, 10] or homomorphic encryption [11–13]. However, these privacy mechanisms only limit what can be inferred from exchanged; they *do not* ensure that parties actually execute the prescribed computations.

Fortunately, *zero-knowledge proofs* (ZKPs) [14, 15] allow participants to prove the correctness of private computations without revealing sensitive inputs. In particular, *zero-knowledge succinct non-interactive arguments of knowledge* (zkSNARK) [16, 17] provides compact proofs and fast verification, making them well suited to federated settings. In this paradigm, parties first bind their data and preprocessing through *commitments* [18, 19], and then prove that their training updates are computed strictly from these committed data values. However, existing applications of ZKPs in federated learning largely focus on inference verification or low-level protocol optimization. The problem of building efficient and scalable zero-knowledge training pipelines for deep models in Split VFL remains underexplored, and integrating zkSNARK circuits into Split VFL introduces several challenges:

- **Inefficiency of monolithic ZKP circuits and commitments.** Unlike inference, training requires frequent gradient computations and parameter updates. Each step demands new proofs and fresh commitments, making it impractical to encode the entire training process into a single monolithic ZKP circuit. This leads to scalability issues and heavy proving overhead.
- **High cost of core operators in embedding computation.** In Split VFL, embeddings are the primary intermediate values exchanged between parties. These embeddings are derived via matrix multiplications, which are computationally intensive in ZKP systems and inflate proof generation time.
- **Blocking of iterative training by proof generation.** While proof verification and message transmission are lightweight, generating ZK proofs is computationally intensive and must complete before the next iteration, as the verifier depends on the proof to proceed. This sequential dependency causes delay and becomes a bottleneck in training throughput.

To address the aforementioned challenges, we propose *ZKSL*, a parallel Zero-Knowledge Split Learning framework for VFL built atop the widely adopted zkSNARK protocol PLONK [16]. PLONK's universal trusted setup, support for custom gates, and succinct verification make it particularly well-suited for deep neural network (DNN) workloads in federated settings. ZKSL partitions the model into layer-wise circuits and generates their zero-knowledge proofs concurrently, significantly accelerating training while preserving end-to-end verifiability.

To ensure cross-layer consistency and maintain efficient parameter binding over multiple training rounds, we introduce *Privacy-Commitment PLONK (PC-PLONK)*, a lightweight commitment scheme embedded into PLONK that securely binds each participant's private data and model parameters. Furthermore, we optimize the proof generation for several key operators, notably the embedding layer, through probabilistic verification techniques that reduce circuit complexity. Finally, we design an *asynchronous proof scheduling mechanism* that decouples proof generation from training iteration, preventing proof delays from stalling the model's forward progress.

To our best knowledge, ZKSL is *the first practical framework* that enables verifiable vertical federated learning through zero-knowledge proofs, under the assumption that data authenticity is externally guaranteed and committed before proof generation. Within this setting, ZKSL ensures verifiable learning for all committed data, preserving data privacy and computational integrity. In summary, the main contributions of this paper are:

- We propose an optimized zero-knowledge proof framework for deep neural network training with layer-wise parallel proof construction. To enable this, we introduce Privacy-Commitment PLONK, which supports low-cost parameter commitments and enforces input-output consistency, enabling scalable parallel proof generation.
- We design a lookup-based embedding mechanism with probabilistic verification, replacing traditional one-hot encoding. This reduces the proof complexity from $O(Nnd)$ to $O(nd)$ with respect to embedding dimension, significantly accelerating proof generation for embedding layers.
- We develop an asynchronous pipeline architecture that decouples computation, proof generation, communication, and verification. This design eliminates bottlenecks from delayed proof tasks and improves overall training throughput in federated environments.
- We implement a prototype system, ZKSL[1], integrating all proposed techniques. Extensive experiments on representative models (DeepFM and CNNs) show that ZKSL reduces proof generation overhead by up to 73% compared to state-of-the-art approaches.

The remainder of the paper is organized as follows. Section II reviews background and related work. Section III formalizes the problem. Section IV presents the design of the ZKSL protocol. Section V analyzes its security. Section VI reports experimental results. Section VII concludes.

II. Background and Related Work

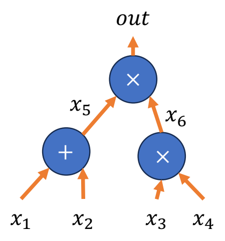
This section presents the fundamental concepts essential for the rest of this paper, including zero-knowledge proofs and vertical federated learning.

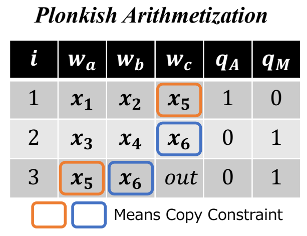
A. Zero-Knowledge Proof

Zero-Knowledge Proofs (ZKPs). ZKPs are cryptographic protocols that allow a prover to convince a verifier that a statement is true while revealing nothing beyond its validity. A well-formed ZKP satisfies three properties: *Completeness*, meaning that an honest prover can always convince the verifier of a true statement; *Soundness*, meaning that no dishonest prover can convince the verifier of a false statement except with negligible probability; and *Zero-Knowledge*, meaning that the verifier learns nothing beyond the fact that the statement is true and gains no information about the prover's private inputs. ZKPs are widely deployed in scenarios where correctness must be verified without exposing sensitive data [14, 15, 20]. A canonical example is Zcash [14], which employs zkSNARKs to prove transaction validity (e.g., sufficient balance and absence of double-spending) while concealing transaction amounts and addresses. In the context of federated learning, where multiple untrusted parties jointly train a model over distributed data, ZKPs provide a means to enforce computation integrity without compromising the confidentiality of private inputs.

ZKPs can be broadly categorized into two types: *interactive* and *non-interactive*. In *interactive zero-knowledge proofs* (IZKPs), the prover and verifier run a multi-round challenge–response protocol [21–24]. By contrast, *non-interactive zero-knowledge proofs* (NIZKPs) produce a single proof verifiable independently, typically under a common reference string (CRS). Among NIZKPs, the *zero-knowledge succinct non-interactive argument of knowledge* (zkSNARK) [17, 25–30] provides succinct proofs and efficient verification, reducing communication and verification overheads. These properties make non-interactive protocols with short proofs and fast verification a natural fit for vertical federated learning, where

[1] https://github.com/YeexiaoZheng/ZKSL





Plonkish Arithmetization

i	w_a	w_b	w_c	q_A	q_M
1	x_1	x_2	x_5	1	0
2	x_3	x_4	x_6	0	1
3	x_5	x_6	out	0	1

🟧 🟦 Means Copy Constraint

Fig. 2: A Simple Plonkish Arithmetization Example

participants hold disjoint feature subsets and must repeatedly attest to local computations without revealing raw data.

zkSNARK and PLONK. Modern general-purpose zkSNARK constructions typically fit into a common algebraic framework with three components: (i) *arithmetization*, which encodes the computation as a system of arithmetic constraints over a finite field; (ii) *a polynomial interactive oracle proof (PIOP)*, which reduces checking constraint satisfiability to verifying identities between low-degree polynomials; (iii) *a polynomial commitment scheme (PCS)*, which lets the prover commit to these polynomials and later open their evaluations at selected points without revealing the underlying witness.

Among zkSNARK systems, the PLONK family [16, 31, 32] stands out for its *universal trusted setup* (reusable across circuits of varying structures), efficient proving, and compatibility with lookup optimizations, making it well suited for encoding complex machine-learning operators such as matrix multiplications and activations. Concretely, PLONK instantiates the arithmetization component with a Plonkish table where computations are arranged in rows of gates selected by polynomial *selectors*; *permutation arguments* enforce copy/equality constraints across rows, while *lookup arguments* enable efficient nonlinear and table-based mappings. At the PCS layer, PLONK typically uses the KZG polynomial commitment scheme [18], which provides constant-size proofs and fast verification for committed polynomials. In this work, we therefore adopt PLONK as our base zkSNARK and later build on it by lightly modifying its arithmetization and PCS.

Arithmetization in PLONK. The PLONK constraint system instantiates arithmetization as a *Plonkish table*, where each row encodes an arithmetic gate and columns represent wire values or selectors. For example Fig 2, addition and multiplication gates are encoded using selector polynomials q_A and q_M, enforcing constraints such as $w_a + w_b = w_c$ or $w_a \cdot w_b = w_c$. To ensure value consistency across dependent gates, *copy constraints* are enforced via a global *permutation argument*, which checks that reused values across rows match under the committed polynomials. This structure allows PLONK to express complex computation graphs compactly while supporting fast proof generation and verification.

Polynomial Commitments in PLONK. A polynomial commitment scheme allows a prover to bind to a hidden polynomial while later proving its evaluations at chosen points without revealing the polynomial itself. In zkSNARKs, the PCS ensures that private inputs and intermediate computations are fixed to a public transcript. PLONK adopts the KZG PCS [18], which uses elliptic curves and bilinear pairings to commit to polynomials, offering constant-size evaluation proofs and fast verification. This makes KZG a natural fit for arithmetic-circuit SNARKs and underpins the commitment layer in PLONK.

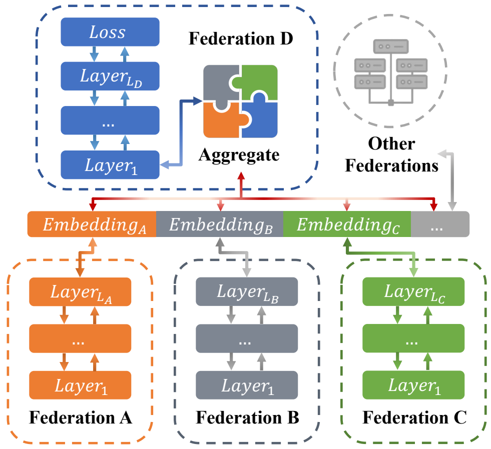


Fig. 3: Neural-Network-Based Split VFL model architecture

Zero-Knowledge Machine Learning (ZKML). ZKML [20, 33–36] combines zero-knowledge proofs with machine learning to ensure both data confidentiality and computational integrity. Using non-interactive proofs, ZKML enables a prover to demonstrate correct model execution (e.g., inference) without revealing inputs or model parameters, an essential capability in outsourced ML settings. Recent work such as Zero-Knowledge Proof of Training (zkPoT) [37, 38] extends ZKML to the training phase, proving that a model was trained on valid data. These approaches often employ interactive protocols with recursive proof composition and Incrementally Verifiable Computation [39–41] to reduce overhead. However, such interactivity is ill-suited to federated learning, where communication efficiency and auditability across dynamic participants are critical. In contrast, our work adopts a non-interactive ZK approach tailored to federated training, enabling public verifiability and long-term trust.

B. Vertical Federated Learning

Federated Learning (FL) [42] enables multiple parties to collaboratively train machine learning models without exposing raw data. It is typically categorized into *Horizontal Federated Learning* (HFL) [43–45], where parties share the same feature space but hold different samples, and *Vertical Federated Learning* (VFL), where participants hold disjoint feature subsets over a common user set [3, 11, 46–50]. VFL is particularly relevant in cross-organizational settings such as finance, healthcare, and e-commerce, where different entities possess complementary user attributes. However, its split feature ownership and centralized label aggregation introduce unique privacy and trust challenges. This work aims to enhancing VFL security by enforcing verifiable training under ZKP.

Split VFL Architecture. Our framework adopts a split learning architecture for VFL, as illustrated in Fig. 3. Each feature party (or *passive party*), e.g., banks or institutions, holds vertically partitioned data and locally computes embedding-vectorized

TABLE I: Comparison of functionality with relevant works

Scheme	Cryptographic Technique	Malicious Passive Party Resilience	Malicious Active Party Resilience	Privacy Preserving	Verifiable Inference	Verifiable Training	Auditability
FedVS [49]	Secret Sharing	✗	✗	✓	✗	✗	✗
FedPass [50]	Confusion	✗	✗	✓	✗	✗	✗
RoPA [51]	SINP	✓	✗	✓	✓	✗	✗
ZKML [20]	zkSNARK	N/A	N/A	✓	✓	✗	✓
KAIZEN [37]	GKR+IVC	N/A	N/A	✓	✓	✓	✗
ZKSL (Ours)	zkSNARK	✓	✓	✓	✓	✓	✓

representations derived from raw inputs. These embeddings are then sent to the label-owning coordinator (or the *active party*), which aggregates the multi-source embeddings and performs downstream training. Embedding generation depends on data modality: images may use convolutional neural networks (CNNs), while categorical or tabular data typically use embedding dictionaries. Regardless of the method, the resulting embeddings compress high-dimensional private data into structured, lower-dimensional representations suitable for secure cross-party learning. During each training round, passive parties transform their local data into embeddings using private forward models and transmit them to the active party, the coordinator. It aggregates these embeddings, executes a forward pass through a global model, computes the loss against ground-truth labels, and derives the corresponding gradients. These gradients are returned to each passive party, which uses them to update its local model parameters. This enables collaborative optimization without raw data exchange.

Security Risks in Split VFL. Unlike horizontal federated learning (HFL), where a *shared* model and aligned gradients give the server sufficient signal for sanity checks, Split VFL reveals far less information. It exchanges only intermediate *embeddings* and a single layer of backward gradients, obscuring passive parties' architectures and feature alignment. With *disjoint* feature subsets, cross-party anomaly detection becomes infeasible, so correctness cannot be inferred from aggregate behavior. Verifiability must therefore be enforced at the level of per-party computations. Existing work mainly enhances embedding privacy under semi-honest assumptions, using techniques such as LCC and secret sharing [49], adaptive obfuscation [50], and embedding obfuscation [52]. While these approaches reduce information leakage (e.g., against inversion attacks [5]), they do not ensure that parties perform the prescribed computations, leaving verifiability unaddressed. Under a malicious threat model, Split VFL faces two key challenges: (i) verifying the correctness of local embeddings and (ii) ensuring the integrity of model updates. Passive parties may submit reused or fabricated embeddings, while the coordinator, who controls loss and gradient computation, can return invalid updates or collude with adversarial participants. Existing approaches such as RoPA [51] use secret-shared SNIP proofs to catch misbehavior by feature parties, but they lack end-to-end verification of model updates and do not support auditability or collusion resistance. These limitations motivate ZKSL, a fully verifiable Split VFL framework that provides end-to-end correctness guarantees under malicious adversaries while preserving data privacy; we outline its design in the next section and compare it with existing approaches in Table I.

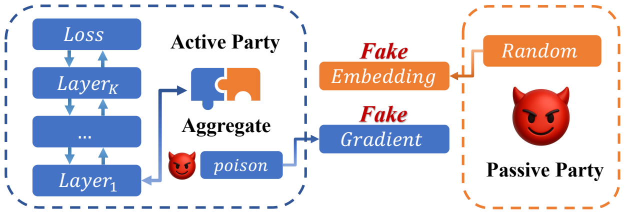


Fig. 4: Attack Examples

III. Problem Formulation

We study a vertical federated learning (VFL) setting involving one active party and multiple passive parties, each holding disjoint feature subsets aligned by a common set of user identifiers. While these parties collaborate to train a shared model, they do not trust each other and may behave adversarially. The goal of ZKSL is to enable verifiable and privacy-preserving collaborative training in this setting. To this end, we consider a malicious threat model and define precise verifiability guarantees that ensure the integrity of local computations and global updates. We assume that, before training, each party obtains its local data through an external trusted process (e.g., organizational controls or audited data pipelines) and binds it to public commitments; parties enter training with these commitments in place. ZKSL then verifies the correctness of computations over the committed inputs, rather than the authenticity of the raw data itself. We now formalize the threat model and the corresponding training semantics enforced by our framework.

A. Threat Model

We consider a malicious adversarial setting in which both the active party (i.e., the label owner) and any subset of passive parties (i.e., feature owners) may deviate arbitrarily from the prescribed protocol. Adversaries are polynomial-time and may collude freely, sharing private information to compromise the training process. We assume the standard cryptographic hardness of underlying primitives, specifically the binding property of commitments and the soundness and zero-knowledge properties of proofs. In addition, all communication is assumed to occur over authenticated and secure channels, preventing impersonation and undetectable message tampering. Within this model, we identify several classes of adversarial behavior and define corresponding security goals:

Malicious Active Party. An adversarial active party controls labels, aggregation, and gradient release. We focus on *computational misbehavior* rather than inference attacks, assuming information leakage (e.g., gradient inversion) is handled by orthogonal privacy defenses. Such a party may inject incorrect or biased gradients (see Fig. 4, left), alter aggregated embeddings, or return inconsistent updates across rounds to embed backdoors. Any forgery of raw labels *prior to commitment* is assumed to be handled by an external data-authenticity layer and is outside the scope of this work.

Malicious Passive Parties. One or more passive parties may also deviate from the prescribed protocol; as before, we focus on *computational* deviations and treat inference attacks as out of scope, assuming they are mitigated by existing privacy-preserving VFL techniques. Such parties may free-ride by reusing stale embeddings, fabricating gradients (right-hand side of Fig. 4), or skipping computation while appearing compliant, or they may actively poison training by corrupting local data or sending inconsistent embeddings to bias convergence. Forgery of raw features *prior to commitment* is likewise delegated to external per-record authenticity mechanisms; once committed, parties can only deviate through incorrect computations.

Collusion. We assume that the active party may collude with an arbitrary subset of passive parties. Such collusion can be leveraged to manipulate global updates or strategically mislead honest participants.

Security Goals. Our proposed framework in this paper targets the following core guarantees:

- *Computation Integrity:* Embeddings and gradients must result from the correct execution of the prescribed training algorithm on the committed inputs.
- *Update Verifiability:* Each global model update is consistent with the committed model parameters and the gradient of the declared loss function.
- *Public Auditability:* Commitments and zero-knowledge proofs form a verifiable audit trail for post hoc validation by participants or external auditors.

We note that auxiliary concerns, such as participant authentication or the genuineness of initial data, are considered orthogonal to our threat model and are assumed to be handled via external mechanisms. The ZKSL framework is designed to satisfy the above security goals; a formal analysis of its guarantees is provided in Section V.

B. Verifiable Training Semantics

We consider a vertical federated learning (VFL) setting involving n passive parties (feature owner) $\mathcal{P}_1, \ldots, \mathcal{P}_n$ that share the same set of user identifiers but hold disjoint subsets of features, i.e., vertically partitioned data. Each passive party $\mathcal{P}_i$ holds private input data x_i, local model parameters w_i, and exposes a public forward function $f_i(\cdot)$. A central active party $\mathcal{S}$ (label owner) holds the label y, local parameters $w_{\mathcal{S}}$, and a public function $\mathcal{F}(\cdot)$ that operates on aggregated embeddings to produce predictions.

Training proceeds in synchronized rounds. At each round t, passive party $\mathcal{P}_i$ draws a local minibatch $x_i^{(t)}$ and computes an embedding:

$$e_i^{(t)} \leftarrow f_i\left(x_i^{(t)}; w_i^{(t)}\right), \quad e_i^{(t)} \in \mathbb{F}^{d_i}.$$

The active party $\mathcal{S}$ aggregates the embeddings $E^{(t)} \leftarrow \bigoplus_{i=1}^{n} e_i^{(t)}$, where $\bigoplus$ denotes the aggregation operator (e.g., concatenation or summation), and computes the prediction $\hat{y}^{(t)} = \mathcal{F}\left(E^{(t)}; w_{\mathcal{S}}^{(t)}\right)$. The loss is computed as $\mathcal{L}^{(t)} = Loss\left(\hat{y}^{(t)}, y^{(t)}\right)$, and the active party updates its local parameters using the gradient

$$\nabla\mathcal{L}^{(t)} = \frac{\partial\mathcal{L}^{(t)}}{\partial w_{\mathcal{S}}^{(t)}}.$$

All participants are assumed to be familiar with the federated data partition and share user identifiers to align their inputs.

Setups. We rely on a *universal* structured reference string (SRS) for KZG polynomial commitments. A standard multi-party "powers-of-tau" ceremony generates an SRS that is secure as long as at least one contributor is honest. For each circuit $\mathcal{C}$ (e.g., a forward or backward circuit), a key-generation algorithm

$$\mathsf{KeyGen}(\mathsf{SRS}, \mathcal{C}) \rightarrow (pk_{\mathcal{C}}, vk_{\mathcal{C}})$$

produces proving and verification keys that can be reused as long as the circuit shape and degree bounds remain unchanged. Henceforth we elide keys and write $\mathsf{Prove}(\cdot)$ and $\mathsf{Verify}(\cdot)$ with the understanding that the appropriate (pk, vk) are used implicitly. We also use the PC-PLONK commitment mechanism and denote commitments by $\mathsf{Com}(\cdot)$.

Proof-Carrying Training Interface. Building on this setup, we require the VFL training process to be *verifiable* despite the private nature of each party's inputs $x_i^{(t)}$ and model parameters $w_i^{(t)}, w_{\mathcal{S}}^{(t)}$. In our ZKSL framework, each participant commits to its private values using $\mathsf{Com}(\cdot)$, and all learning-related computations, including forward evaluation, gradient derivation, and parameter updates, are attested by succinct zero-knowledge proofs over a PLONKish arithmetization. Proofs are produced by $\mathsf{Prove}(\cdot)$ and checked by $\mathsf{Verify}(\cdot)$ against the corresponding commitments. The core verifiability requirement has two components: (i) *verifiable local computation*, which ensures that each claimed embedding $e_i^{(t)}$ is honestly computed from the committed values $x_i^{(t)}$ and $w_i^{(t)}$; and (ii) *verifiable global updates*, which ensure that the coordinator correctly computes gradients and applies the update rule to obtain a new committed model state. Together, these properties guarantee that all learning steps are faithfully executed and form the basis of ZKSL's integrity guarantees.

Verifiable Local Computation. Given the public forward function f_i and a claimed embedding e_i, party $\mathcal{P}_i$ generates a non-interactive zero-knowledge proof that there exist private values x_i and w_i such that

$$e_i = f_i(x_i; w_i),$$

without revealing x_i or w_i. This ensures that each passive party has faithfully executed the local forward pass on its private data and parameters.

Verifiable Global Update. At each training round t, given the

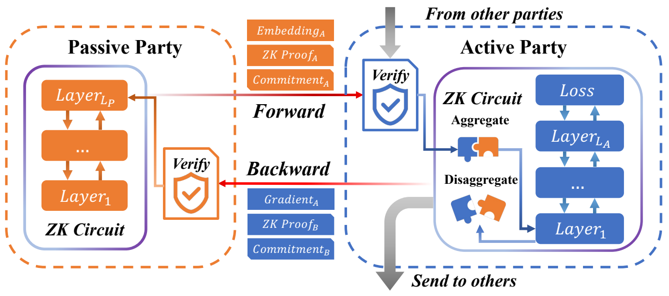


Fig. 5: Zero-Knowledge NN-Based Split VFL architecture

initial commitment $\mathsf{Com}(w_{\mathcal{S}}^{(t)})$, learning rate η, the claimed gradient $\nabla L^{(t)}$, and the updated commitment $\mathsf{Com}(w_{\mathcal{S}}^{(t+1)})$, the coordinator produces a zero-knowledge proof of the following:

(i) *Update Consistency:* There exist openings $w_{\mathcal{S}}^{(t)}$ and $w_{\mathcal{S}}^{(t+1)}$ such that

$$w_{\mathcal{S}}^{(t+1)} = w_{\mathcal{S}}^{(t)} - \eta \nabla \mathcal{L}^{(t)},$$

and $\mathsf{Com}(w_{\mathcal{S}}^{(t+1)})$ is a valid commitment to $w_{\mathcal{S}}^{(t+1)}$ (After each update, model weights are re-committed to maintain verifiable linkage across rounds).

(ii) *Gradient Correctness:* The gradient satisfies

$$\nabla \mathcal{L}^{(t)} = \frac{\partial}{\partial w_{\mathcal{S}}^{(t)}} \mathcal{L}\big(\mathcal{F}(E^{(t)}; w_{\mathcal{S}}^{(t)}); y^{(t)}\big),$$

for some aggregated embedding $E^{(t)} = \bigoplus_{i=1}^{n} e_i^{(t)}$, where each $e_i^{(t)}$ is accompanied by a valid proof of local computation.

This mechanism prevents the active party from introducing incorrect or arbitrary model updates without detection, thereby ensuring the integrity of the global learning process.

IV. Design

This section details the design of ZKSL, a zero-knowledge split VFL framework that makes training both private and publicly verifiable at scale. We first present the verifiable training protocol and its commitments-and-proofs workflow. We then introduce layer-wise parallel proving, Privacy Commitment PLONK, and probabilistic lookup–based embedding verification to cut proving costs without weakening guarantees. Finally, we describe an asynchronous proof scheduling scheme that overlaps compute and proving to maximize throughput in multi-party deployments.

A. Training Protocol

In each training round t, the passive and active parties jointly execute a verifiable training protocol. All commitments are public and included in the global transcript. Zero-knowledge proofs expose the corresponding commitments as public inputs, ensuring that any deviation, such as using values inconsistent with commitments, fabricating embeddings, gradients, or update, will be detected through verification failure.

As depicted in Fig. 5 and formalized in Algorithm 1, each round proceeds in three stages. In the *embedding phase*

Algorithm 1: ZKSL Training Protocol for Round t

```
Data: Passive Party: $\{x_i^{(t)}, w_i^{(t)}\}_{i=1}^n$; Active Party: $y^{(t)}, w_{\mathcal{S}}^{(t)}$.
Result: $\{w_i^{(t+1)}\}_{i=1}^n$, $w_{\mathcal{S}}^{(t+1)}$ and accepted zk proofs.
1  Passive Party Forward:
2      $C_{x_i}^{(t)} \leftarrow \mathsf{Com}(x_i^{(t)}),\ C_{w_i}^{(t)} \leftarrow \mathsf{Com}(w_i^{(t)})$
3      $e_i^{(t)} \leftarrow f_i(x_i^{(t)}; w_i^{(t)})$
4      $\pi_{i,\mathrm{fwd}}^{(t)} \leftarrow \mathsf{ProveFwd}(C_{x_i}^{(t)}, C_{w_i}^{(t)}, e_i^{(t)})$
5      Send $\big(e_i^{(t)}, C_{x_i}^{(t)}, C_{w_i}^{(t)}, \pi_{i,\mathrm{fwd}}^{(t)}\big)$ to Active Party
6  Active Party Round:
7      if $\forall i \in [n]$, $\mathsf{Verify}(C_{x_i}^{(t)}, C_{w_i}^{(t)}, e_i^{(t)}, \pi_{i,\mathrm{fwd}}^{(t)}) = true$
8          $E^{(t)} \leftarrow \bigoplus_{i=1}^{n} e_i^{(t)}$
9          $\hat{y}^{(t)} \leftarrow \mathcal{F}(E^{(t)}; w_{\mathcal{S}}^{(t)})$
10         $\mathcal{L}^{(t)} \leftarrow \mathsf{Loss}(\hat{y}^{(t)}, y^{(t)})$
11         $C_{\mathcal{L}}^{(t)} \leftarrow \mathsf{Com}(\mathcal{L}^{(t)}),\ C_{w_{\mathcal{S}}}^{(t)} \leftarrow \mathsf{Com}(w_{\mathcal{S}}^{(t)})$
12         Compute $\nabla \mathcal{L}^{(t)}$
13         $w_{\mathcal{S}}^{(t+1)} \leftarrow w_{\mathcal{S}}^{(t)} - \eta \nabla \mathcal{L}^{(t)}$
14         $C_{w_{\mathcal{S}}}^{(t+1)} \leftarrow \mathsf{Com}(w_{\mathcal{S}}^{(t+1)})$
15         $\pi_{\mathcal{S}}^{(t)} \leftarrow \mathsf{ProveSrv}(C_{\mathcal{L}}^{(t)}, C_{w_{\mathcal{S}}}^{(t)}, C_{w_{\mathcal{S}}}^{(t+1)}, \nabla \mathcal{L}^{(t)}, E^{(t)})$
16         Broadcast $\big(\{\nabla \mathcal{L}_i^{(t)}\}_{i=1}^n,\ C_{w_{\mathcal{S}}}^{(t+1)},\ \pi_{\mathcal{S}}^{(t)}\big)$
17     else
18         Reject
19 Passive Party Update:
20     if $\mathsf{Verify}(C_E^{(t)}, C_{\mathcal{L}}^{(t)}, C_{w_B}^{(t)}, C_{w_B}^{(t+1)}, \nabla \mathcal{L}_i^{(t)}, \pi_B^{(t)}) = true$
21         $w_i^{(t+1)} \leftarrow w_i^{(t)} - \eta \nabla \mathcal{L}_i^{(t)}$
22         $C_{w_i}^{(t+1)} \leftarrow \mathsf{Com}(w_i^{(t+1)})$
23         $\pi_{i,\mathrm{upd}}^{(t)} \leftarrow \mathsf{ProveUpd}(C_{w_i}^{(t)}, C_{w_i}^{(t+1)}, \nabla \mathcal{L}_i^{(t)})$
24         send $\big(C_{w_i}^{(t+1)}, \pi_{i,\mathrm{upd}}^{(t)}\big)$ to Active Party
25     else
26         Reject
27 Iterate for $t+1$ until convergence
```

(Lines 1–5), each passive party commits to its private minibatch and model weights using a binding scheme $\mathsf{Com}(\cdot)$, computes a local embedding, and generates a succinct zkSNARK (based on a PLONKish arithmetization) proving that the embedding results from a correct forward pass over the committed values. The embedding, commitments, and proof are then sent to the active party. In the *aggregation and update phase* (Lines 6–18), the active party verifies all received proofs, aggregates the embeddings, performs forward computation, and derives the loss and gradient. It then commits to the aggregate input, loss, and its own parameters, updates its weights, and produces a proof attesting to the correctness of this computation. The active party broadcasts the per-party gradient slices, the new commitment, and the proof. In the *response and verification phase* (Lines 19–26), each passive party verifies the proof, applies its local update using the received gradient slice, commits to the updated weights, and generates a proof that the update matches the declared gradient. The round counter is then incremented (Line 27), and the protocol repeats until convergence.

This design ensures that all computation steps, by both passive and active parties, are publicly verifiable, while preserving the privacy of data and parameters via commitments and zero-knowledge proofs. For efficiency, we leverage PC-PLONK's dedicated commitment column to bind parameters inside the circuit, avoiding hash-heavy commitment gadgets and reducing overall proving overhead.

Layer-Wise Parallel Proof Generation. Generating zero-knowledge proofs for deep neural networks (DNNs) is computationally intensive due to the depth of layered operations, including large matrix multiplications and nonlinear activations. Monolithic zk circuits scale poorly under such workloads. To improve scalability, we exploit the inherent layer-wise structure of DNNs to enable *layer-wise parallel* proof generation as demonstrated in Fig. 6. Each layer is treated as an independent computational unit, allowing zk proofs to be generated in parallel with localized resource allocation. These per-layer proofs are then linked via cross-layer consistency checks, reducing proving latency while preserving end-to-end correctness.

Let $\{L_k\}_{k=1}^n$ denote the n layers of a DNN, each with parameters $\{\theta_k\}_{k=1}^n$ and initial input z_0 (such as the aggregated embedding $E^{(t)}$). The computation proceeds through the network via the recurrence

$$z_k := L_k(z_{k-1}; \theta_k), \quad \text{for } k = 1, \ldots, n.$$

For each intermediate value and parameter tensor, we publish binding commitments:

$$C_{z_k} \leftarrow \mathsf{Com}(z_k), \qquad C_{\theta_k} \leftarrow \mathsf{Com}(\theta_k).$$

Let P_{L_k} denote the zero-knowledge proof corresponding to layer L_k, and let Π_{total} represent the final aggregated proof. We assume a PLONKish proof system $\mathsf{PC-PLONK}$ equipped with privacy-preserving commitment columns and copy constraints to enforce consistency across circuit boundaries.

The protocol is detailed in Algorithm 2. First, a full forward pass is executed to compute all intermediate values $\{z_k\}$, and binding commitments $\{C_{z_k}, C_{\theta_k}\}$ are published. Next, for each layer L_k, a circuit $\mathcal{C}_k$ is constructed with private inputs (z_{k-1}, θ_k, z_k), constrained to be consistent with their respective commitments. The zk proof for this layer is then computed as

$$P_{L_k} \leftarrow \mathsf{ProveLayer}_{\mathsf{PC-PLONK}}(C_{z_{k-1}}, C_{\theta_k}, C_{z_k}),$$

and all such proofs are generated in parallel across the n layers.

After proof generation, a lightweight consistency check is performed to ensure that for each k, the output commitment C_{z_k} of layer L_k matches the input commitment C_{z_k} expected by layer L_{k+1}. This step stitches the individually generated layer proofs into a coherent, end-to-end proof chain. Finally, the individual proofs $\{P_{L_k}\}_{k=1}^n$ are aggregated into a single succinct proof $\Pi_{\text{total}} \leftarrow \mathsf{Aggregate}(\{P_{L_k}\}_{k=1}^n)$.

This layer-wise strategy enables scalable, parallel proof generation while preserving global correctness through commitment-bound consistency.

Privacy Commitment PLONK (PC-PLONK). Before generating computation proofs, all private inputs and weights must be committed without disclosure (e.g., $C_{x_i}^{(t)} \leftarrow \mathsf{Com}(x_i^{(t)})$, $C_{w_i}^{(t)} \leftarrow \mathsf{Com}(w_i^{(t)})$, $C_{z_k} \leftarrow \mathsf{Com}(z_k)$). However, directly integrating standard commitments, particularly hash-based schemes [19], into zk-SNARK circuits is impractical, as in-circuit hash evaluation significantly increases arithmetic overhead and prover cost.

Algorithm 2: Layer-Wise Parallel Proofs

```
   Input: Layers $\{L_k\}_{k=1}^n$ with params $\{\theta_k\}$; input $z_0$ (e.g.,
          $x_i^{(t)}, E^{(t)}$).
   Output: Aggregated proof $\Pi_{\text{total}}$.
 1 Compute (sequential):
 2 for $k = 1$ to $n$ do
 3     $z_k \leftarrow L_k(z_{k-1}; \theta_k)$
 4     $C_{z_k} \leftarrow \mathsf{Com}(z_k)$
 5     $C_{\theta_k} \leftarrow \mathsf{Com}(\theta_k)$
 6 Parallel proving:
 7 for $k \in \{1, \ldots, n\}$ in parallel do
 8     Build circuit $\mathcal{C}_k$ with privacy column $P$ holding
         $z_{k-1}, \theta_k, z_k$ consistent with $C_{z_{k-1}}, C_{\theta_k}, C_{z_k}$
 9     Enforce copy constraints for all uses from $P$
10     $P_{L_k} \leftarrow \mathsf{ProveLayer}_{\mathsf{PC-PLONK}}(C_{z_{k-1}}, C_{\theta_k}, C_{z_k})$
11 Consistency check (lightweight):
12 for $k = 1$ to $n - 1$ do
13     Check equality of adjacent commitments: $C_{z_k}$ in $P_{L_k}$
         equals the input commitment referenced by $P_{L_{k+1}}$
14 Aggregate proofs:
15     $\Pi_{\text{total}} \leftarrow \mathsf{Aggregate}(P_{L_1}, \ldots, P_{L_n})$
16 return $\Pi_{\text{total}}$
```

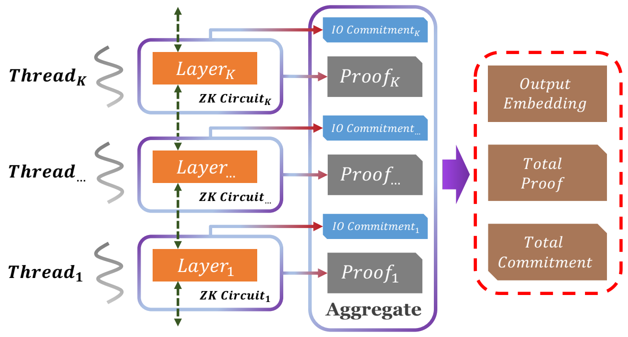
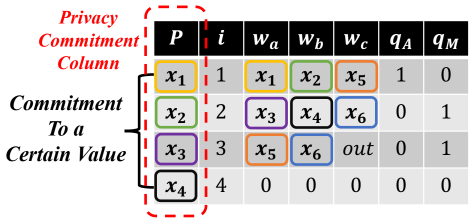


Fig. 6: Design of Layer-Wise Parallel Proof Generation

To address this, we introduce *Privacy Commitment PLONK (PC-PLONK)*, a refinement of the PLONK protocol that *separates commitment binding from computation*. PC-PLONK augments the PLONK trace with a *privacy-commitment column* P, which holds committed private values. Every use of a private input or parameter within the circuit must reference its corresponding entry in P, and permutation constraints enforce consistency. This design ensures that values are reused without redundancy, while eliminating the need to evaluate commitment logic (e.g., hashing) inside the circuit. Binding to public commitments is deferred to PLONK's polynomial commitment layer, preserving verifiability with minimal overhead.

Example 1. *Extending Fig. 2, we introduce column P to store private inputs and model parameters used in subsequent computations (Fig. 7). All gates access these values from P, and copy constraints enforce consistency across usage. Any deviation from the committed values breaks the proof, ensuring low-cost integrity enforcement while preserving input privacy.*

Compared to in-circuit hash-based commitments such as Poseidon Merkle trees, which must be enforced by additional PIOP constraints, PC-PLONK shifts commitment binding to the



P	i	w_a	w_b	w_c	q_A	q_M
x_1	1	x_1	x_2	x_5	1	0
x_2	2	x_3	x_4	x_6	0	1
x_3	3	x_5	x_6	out	0	1
x_4	4	0	0	0	0	0

Fig. 7: Design of Privacy Commitment PLONK

PCS layer and keeps the arithmetic circuit almost unchanged. A Poseidon gadget needs to encode each hash inside the Plonkish table, adding $\Theta(m(R_F + R_P))$ hash-gate rows for m committed values, whereas PC-PLONK introduces a few advice columns to store privacy data (one conceptual column P, or several columns in our implementation) plus $\Theta(u)$ additional permutation constraints for u uses of these values. Since these permutation constraints are handled by PLONK's existing global permutation argument and require no new custom gates, the incremental PIOP cost of PC-PLONK is negligible compared to a Poseidon-based commitment circuit.

B. Embedding Generation

Embedding layers are essential in vertical federated learning for encoding categorical inputs into dense vectors, allowing local data to be abstracted before sharing. In ZKSL, this abstraction also protects raw features from being directly exposed to other participants. However, in the zero-knowledge proof (ZKP) setting, embedding computation becomes a major bottleneck. Unlike computational frameworks such as PyTorch [53], which optimize embedding lookups via indexing, our verification logic must explicitly enforce **Constraints** within the circuit. Consequently, standard implementations, such as those used in frameworks like EZKL [54], typically perform dot-product verification using one-hot encodings and matrix multiplications. This approach is highly inefficient in ZK circuits, especially as the batch size increases: most entries in the one-hot vectors are zero, yet multiplication constraints must still be generated, leading to proving overhead in large-scale models. To address this inefficiency, we reformulate embedding as a sparse, index-based operation and explore ZKP-compatible designs that significantly reduce constraint complexity.

Embedding retrieval can be efficiently implemented via lookup tables, avoiding the need for one-hot encoding followed by matrix multiplication. This is achieved through a *set-membership argument*, which verifies that a selected embedding vector appears at the correct index within a committed dictionary. Compared to polynomial evaluation, this method incurs fewer constraints and directly reflects the index-based semantics of embedding layers.

However, standard lookup mechanisms in zero-knowledge proof (ZKP) circuits face scalability challenges. As the embedding dimension d grows, the number of required lookup tables and associated constraints increases proportionally. For example, an embedding matrix $E \in \mathbb{F}^{99\times 9}$ requires verifying membership in 99 rows of 9-dimensional embeddings, each

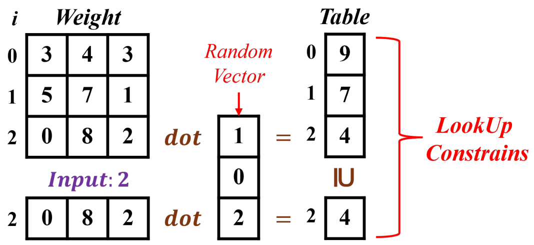


Fig. 8: Lookup-based Embedding with Probability Optimization

prepended with an index, forming 10-dimensional lookup entries, such as:

$$\{(1, e_{1,1}, \ldots, e_{1,9}), \ldots, (99, e_{99,1}, \ldots, e_{99,9})\}.$$

To verify the correctness of retrieving, say, row 2, one must prove that the tuple $(2, e_{2,1}, \ldots, e_{2,9})$ belongs to this set. As d increases, the lookup burden quickly becomes significant, and in some cases even exceeds the cost of conventional one-hot-based matrix multiplication.

To address this, we propose a novel *probabilistic verification* approach for embedding retrieval, inspired by similar techniques in matrix multiplication. This method reduces the verification cost to be independent of the embedding dimension d. Its security relies on a probabilistic projection argument: if two vectors have equal random projections under a shared random challenge, then with high probability the vectors themselves are equal. We formalize this result in the following theorem and defer its proof to the appendix.

Theorem 1. *Let $E \in \mathbb{F}_p^{n\times d}$ be an embedding matrix, where each row $E_i \in \mathbb{F}_p^d$ is an embedding vector. Let r be a random vector uniformly sampled from $\mathbb{F}_p^d$. Suppose a prover returns $\hat{E}_i \in \mathbb{F}_p^d$ in response to a query for E_i. If $\hat{E}_i \neq E_i$, then the probability that $\hat{E}_i \cdot r = E_i \cdot r$ is at most $1/p$.*

To reduce the cost of verifying embedding lookups in zero-knowledge circuits, we use a random vector $r \in \mathbb{F}^d$ to compress high-dimensional embeddings into scalar values via inner product $v_i = E_i \cdot r = \sum_{j=1}^{d} e_{i,j} \cdot r_j$, where E_i is the i-th row of the embedding matrix E. During proof generation, the prover computes this compressed value v_i, and the verifier checks correctness probabilistically using a lookup table that contains only index–projection pairs. Verification proceeds in three steps:

1) The lookup table stores compressed entries $\{(i, v_i)\}$;
2) The prover computes $v_i = E_i \cdot r$ for a given index i;
3) A lookup constraint ensures that the claimed pair (i, v_i) exists in the table.

This technique decouples the constraint cost from the embedding dimension d, dramatically reducing overhead. Instead of verifying all d components of E_i, we check only the projection v_i, with high-probability sufficiency, due to the above theorem.

Example 2. *As shown in Fig. 8, suppose $E \in \mathbb{F}^{3\times 3}$ and $r \in \mathbb{F}^3$. For the third row, we compute:*

$$v_2 = e_{2,1}r_1 + e_{2,2}r_2 + e_{2,3}r_3.$$

The prover submits $(2, v_2)$, and the verifier enforces a lookup

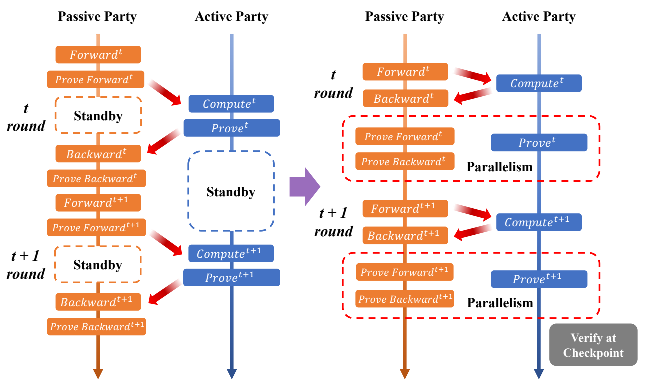


Fig. 9: Computation-Validation Asynchronous Framework

constraint to check this pair against a preconstructed table. Since the projection uniquely identifies the embedding with high probability, individual components $e_{2,j}$ need not be verified.

This probabilistic lookup approach offers a scalable and efficient solution for embedding verification in ZKP. It reduces constraint complexity from $O(Nnd)$ to $O(nd)$, where N is the batch size, n is the vocabulary size, and d is the dimension size (see Appendix VIII-B for details), eliminates redundant computation from sparse matrix operations, and avoids the overhead of one-hot encoding. By replacing explicit matrix multiplications with a dot-product-based commitment and a simple lookup constraint, the circuit enforces consistency between the claimed index and embedding while minimizing proving cost. This enables practical certification of embedding layers even in large-scale models.

C. Asynchronous Proof Scheduling

In conventional execution, forward and backward computations are tightly coupled with their corresponding zero-knowledge proof (ZKP) generation. As illustrated on the left of Fig. 9, a participant must complete a computation stage, wait for the ZKP to be generated, and only then transmit the result and proof to downstream parties. This strict serialization introduces idle "Standby" periods, underutilizes resources, and degrades throughput, especially in multi-party VFL, where such latency compounds across participants.

To address this bottleneck, ZKSL introduces an *asynchronous proof scheduling* mechanism that decouples model computation from ZKP generation. The key idea is to allow a participant to commit to the output of a computation stage and forward it, along with its commitment, to downstream parties without waiting for the proof. This triggers the next computation round (e.g., round $t{+}1$) to begin promptly, while the ZKP for round t is generated asynchronously in the background. As shown in Fig. 9 (right), ZKPs attest to computations from the previous round and are verified at a checkpoint in the following round. The red dotted boxes represent deferred proof tasks for round t, executed in parallel across federations. This design overlaps computation and proof generation across rounds, improving pipeline utilization and system throughput.

To implement this pipeline, each participant maintains a Prove-Queue (PQ) for deferred proof tasks and enforces

Algorithm 3: Asynchronous Compute/Prove Scheduling with K-Window Verification

```
   Data: $x^{(t)}$, $w^{(t)}$; window size $K$.
   Result: Serial compute overlapped with proving; windowed
           verification.
1  Initialize
2      $PQ, Ready \leftarrow \emptyset$; $\pi[\cdot] \leftarrow$ null; $t \leftarrow 0$; $verified \leftarrow 0$
3  Main loop (serial compute; async proving)
4      while not converged do
5          $t \leftarrow t+1$
6          Checkpoint at round boundary
7              wait until $\{t-K,\dots,t-1\} \subseteq Ready$
8              $\Pi_{\text{local}} \leftarrow \mathsf{Aggregate}\big(\{\pi[r]\}_{r=t-K}^{t-1}\big)$
9              send $\Pi_{\text{local}}$ to peers
10             $\Pi_{\text{peers}} \leftarrow \mathsf{ReceiveAll}(\text{peers})$
11             if not $\mathsf{VerifyAll}(\Pi_{peers})$ then
12                 $\mathsf{Rollback}(t-K)$
13                 for $r \leftarrow t-K$ to $t-1$ do
14                     $\pi[r] \leftarrow$ null; $Ready \leftarrow Ready \setminus \{r\}$
15                 $t \leftarrow t-K-1$; continue
16             $verified \leftarrow t-1$
17         $y^{(t)} \leftarrow \mathsf{ComputeStage}(x^{(t)}, w^{(t)})$
18         $C_x^{(t)} \leftarrow \mathsf{Com}(x^{(t)})$,   $C_w^{(t)} \leftarrow \mathsf{Com}(w^{(t)})$
19         send $\langle t,\, y^{(t)} \rangle$ to peers
20         enq $(PQ, \langle t,\, C_x^{(t)}, C_w^{(t)}, y^{(t)} \rangle)$
21         wait until $t - verified < K$
22 Prover workers (run concurrently)
23     while $PQ \neq \emptyset$ do
24         $\langle r,\, C_x, C_w, y \rangle \leftarrow$ deq$(PQ)$
25         $\pi[r] \leftarrow \mathsf{Prove}(C_x, C_w, y)$
26         $Ready \leftarrow Ready \cup \{r\}$
```

a window size K that bounds how far computation can run ahead of proving. As formalized in Algorithm 3, the scheduler interleaves sequential computation with asynchronous proving under this sliding-window constraint. At the start of each round, the node checks whether proofs for the past K rounds are available (Line 7). If ready, it aggregates and exchanges them with peers and performs collective verification (Lines 8–11). If any proof fails to verify, the node rolls back up to K rounds by clearing local state and retrying (Lines 12–15). Otherwise, it proceeds with the computation for round t (Line 18), commits the output (Line 19), and sends it downstream (Line 20), allowing the next round to start. Importantly, the corresponding ZKP is *not* generated inline. Instead, the proof task is appended to the Prove-Queue and handled asynchronously by prover workers (Lines 21–22). A scheduler guard ensures that the number of in-flight rounds without completed proofs does not exceed the window size K; if the limit is reached, computation pauses until progress is made (Line 23). Meanwhile, dedicated prover threads drain the queue by proving each task and marking it ready for future verification (Lines 26–29).

This asynchronous scheduling allows computation to proceed without waiting for proofs. By overlapping compute and prove phases and deferring verification to boundaries, the system balances throughput, latency, and rollback safety. The window size K serves as a tunable knob for controlling concurrency depth, backpressure, and recovery cost, enhancing robustness under varying workloads and network conditions.

V. Security Analysis

We analyze ZKSL's security under the malicious threat model outlined in Section III, distinguishing formal cryptographic guarantees from empirical robustness. ZKSL achieves *completeness*, *soundness*, and *zero-knowledge* through PLONK and our PC-PLONK extension. For brevity, we summarize the core structure of the security proofs here; full proofs and reductions are provided in the appendix.

A. Unified Security Definition: Computation Integrity, Update Verifiability and Public Auditability

Let λ be the security parameter. Parties $\mathcal{P} = \{P_0, \dots, P_n\}$ run a VFL protocol for T rounds. Each party P_i has local data D_i and randomness r_i and (unless corrupted before commit) posts an immutable commitment $\mathsf{Com}_i = \mathsf{Com}(D_i; r_i)$ to a public append-only log at setup. We assume Com is binding and hiding.

Let S_t denote the global model state after round t (with S_0 initial). For each party P_i fix a canonical local operator

$$\mathsf{Exec}_i(S_{t-1}, D_i; \eta) \mapsto (e_i^t, g_i^t, \mathsf{aux}_i^t),$$

which deterministically (given randomness) models the declared forward/backward computation producing an embedding e, a gradient g and auxiliary values aux for round t.

Define the basic predicates:

$$\mathsf{ConsistentWithCommit}(\mathsf{Com}_i, D) \overset{\text{def}}{\iff} \exists r: \ \mathsf{Com}(D; r) = \mathsf{Com}_i.$$

$$\mathsf{ValidExec}_i(e, g, \mathsf{aux}, S_{t-1}, \mathsf{Com}_i) \overset{\text{def}}{\iff} \exists D, r, \eta, \ \ s.t.$$
$$\mathsf{Com}(D; r) = \mathsf{Com}_i \ \wedge \ (e, g, \mathsf{aux}) = \mathsf{Exec}_i(S_{t-1}, D; \eta).$$

(Thus ValidExec expresses that the triple (e, g, aux) can be produced by correctly running Exec_i on some data consistent with the published commitment.)

Let $\ell(S; \mathbf{D})$ be the declared global loss (with $\mathbf{D} = (D_0, \dots, D_n)$) and let $\mathsf{Agg}(\{o_i\})$ be the declared aggregation mapping local contributions to a global update. Define global consistency of accepted local contributions:

$$\mathsf{GlobalConsistent}\big(\{o_i\}, S_{t-1}, \mathsf{Com}_{0:n}\big) \overset{\text{def}}{\iff}$$
$$\exists D_0, \dots, D_n, r_0, \dots, r_n \ \Big(\bigwedge_{i=0}^{n} \mathsf{Com}(D_i; r_i) = \mathsf{Com}_i$$
$$\wedge \ \mathsf{Agg}(\{o_i\}) = \nabla_S \ell\big(S_{t-1}; (D_0, \dots, D_n)\big)\Big).$$

(Thus the aggregated accepted contributions equal the declared loss gradient on some data consistent with commitments.)

We assume the protocol specifies a deterministic verification algorithm $\mathsf{Verify}(\mathsf{Trans}, S_{t-1}) \in \{\mathsf{Accept}, \mathsf{Reject}\}$ that, given the round transcript (including non-interactive ZK proofs, openings, etc.) and previous state, either accepts and returns the set $\mathsf{Accepted}^t$ of tuples used for aggregation, or rejects. We also assume a deterministic post-hoc auditor $\mathsf{Audit}(\mathsf{Trans}) \in \{\mathsf{Accept}, \mathsf{Reject}\}$ that checks commitments and proofs in a transcript and derives the same logical predicates as Verify.

Adaptive security experiment. Define the experiment $\mathbf{Exp\_Unified}(A, \lambda)$ for a PPT adversary A that may adaptively corrupt parties via an oracle $\mathsf{Corrupt}(i)$ (corruption is irrevocable). The experiment proceeds:

1) **Commit phase.** For each party P_i:
   - If A issues $\mathsf{Corrupt}(i)$ *before* the commit action, A chooses (D_i, r_i) and computes $\mathsf{Com}_i = \mathsf{Com}(D_i; r_i)$.
   - Otherwise (honest at commit time), sample honest (D_i, r_i) and compute $\mathsf{Com}_i = \mathsf{Com}(D_i; r_i)$.

   Publish all $\{\mathsf{Com}_i\}$ to the immutable public log. From now on commitments are fixed. Initialize S_0.
2) **Oracles available to A.**
   - $\mathsf{Corrupt}(i)$: when invoked on an uncorrupted i, mark P_i corrupted and return the full internal state of P_i (including D_i, r_i, secret keys, randomness used so far). After this, A controls P_i.
   - Normal protocol interaction: A may send arbitrary messages on behalf of corrupted parties; the experiment simulates honest parties below.
3) **Rounds.** For each round $t = 1, \dots, T$:
   a) For each currently honest (not-yet-corrupted) P_i: sample fresh local randomness η_i^t and compute $(e_i^t, g_i^t, \mathsf{aux}_i^t) := \mathsf{Exec}_i(S_{t-1}, D_i; \eta_i^t)$. Produce the protocol messages/proofs for honest P_i (including commitments openings, ZK proofs) and append to the round transcript.
   b) For each corrupted P_j: let A supply arbitrary messages, proofs, openings, embeddings/gradients, etc., for round t; append these to the transcript.
   c) Run Verify on the round transcript and S_{t-1}. If Verify outputs Reject then set $S_t \leftarrow S_{t-1}$. If Verify outputs Accept then it also returns an accepted set $\mathsf{Accepted}^t$ of tuples $(i, t, e, g, \mathsf{aux})$ which are used by the declared Agg and Update to derive S_t.
4) **Post-hoc transcript output.** At the end, A may also output an arbitrary transcript Trans (possibly equal to the real transcript); an external auditor runs $b \leftarrow \mathsf{Audit}(\mathsf{Trans})$.

Bad events (security failure). The experiment outputs 1 (i.e. a security failure occurs) iff one of the following holds:

- **(Computation Integrity violation)** There exists a round t and an accepted tuple $(i, t, e, g, \mathsf{aux}) \in \mathsf{Accepted}^t$ such that
  $$\neg \ \mathsf{ValidExec}_i(e, g, \mathsf{aux}, S_{t-1}, \mathsf{Com}_i).$$
  (An accepted embedding/gradient cannot be produced by running the declared Exec_i on any data consistent with Com_i.)
- **(Update Verifiability violation)** There exists a round t for which the set of accepted local contributions $\{o_i^t\} := \{(e_i^t, g_i^t, \mathsf{aux}_i^t) \in \mathsf{Accepted}^t\}$ satisfies
  $$\neg \ \mathsf{GlobalConsistent}\big(\{o_i^t\}, S_{t-1}, \mathsf{Com}_{0:n}\big).$$
  (The aggregated update is not equal to the declared loss gradient on any data vector consistent with the commitments.)

- **(Public Auditability violation)** The auditor accepts a transcript yet the transcript contains a round t that satisfies either of the above violations with respect to the commitments appearing in Trans:

$$\mathsf{Audit}(\mathsf{Trans}) = \mathsf{Accept}$$
$$\wedge \Big(\exists t \text{ s.t. EITHER Computation Integrity}$$
$$\text{OR Update Verifiability violation w.r.t. } \mathsf{Trans}\Big).$$

(An auditable transcript is accepted while some accepted update in it is invalid.)

Adversary advantage and security requirement. Define

$$\mathsf{Adv}_{\mathsf{Unified}}(A, \lambda) \;:=\; \Pr\big[\mathbf{Exp\_Unified}(A, \lambda) = 1\big],$$

with probability over honest randomness and A's randomness and over the behavior of Corrupt oracle. The protocol achieves unified security (Computation Integrity, Update Verifiability and Public Auditability) iff for all PPT adversaries A,

$$\mathsf{Adv}_{\mathsf{Unified}}(A, \lambda) \le \mathsf{negl}(\lambda).$$

Completeness. If all parties are honest (or any corruption queries occur only after the simulator hands over internal honest state) and honest proofs/signatures/openings are generated, then:

$$\Pr\big[\mathbf{Exp\_Unified}(A_{\mathsf{honest}}, \lambda) = 0\big] \ge 1 - \mathsf{negl}(\lambda),$$

i.e. honest executions are accepted by Verify and by Audit except with negligible probability.

B. Security Theorem and Reductions

We state the theorem and give a concise, rigorous reduction proof in appendix VIII-D. Let λ be the security parameter, n the number of parties, T the number of rounds, and let Q be an upper bound on the number of checked objects (proofs/openings/statements) produced by each party in each round. Let $\mathsf{Adv}_{\mathsf{Unified}}(A, \lambda)$ denote the advantage of any PPT adversary A in the unified experiment. Let $\mathsf{Adv}_{\mathsf{PLONK}}(\lambda)$ and $\mathsf{Adv}_{\mathsf{ComBind}}(\lambda)$ denote the maximum PPT advantages to break PLONK soundness and commitment binding respectively. Assume the reduction may use standard simulation oracles (PLONK simulator / CRS trapdoor) and let $\delta_{\mathsf{sim}}(\lambda)$ be the total simulation error (negligible under the usual setup). Then there exists the explicit factor $M := n \cdot T \cdot Q$ such that:

Theorem 2 (Reduction to PLONK soundness and commitment binding)**.** *For every PPT adversary A there holds*

$$\mathsf{Adv}_{\mathsf{Unified}}(A, \lambda) \;\le M \cdot \big(\mathsf{Adv}_{\mathsf{PLONK}}(\lambda) + \mathsf{Adv}_{\mathsf{ComBind}}(\lambda)\big) \;+\; \delta_{\mathsf{sim}}(\lambda).$$

Consequently, if $\mathsf{Adv}_{\mathsf{PLONK}}(\lambda)$, $\mathsf{Adv}_{\mathsf{ComBind}}(\lambda)$ and $\delta_{\mathsf{sim}}(\lambda)$ are negligible in λ, then $\mathsf{Adv}_{\mathsf{Unified}}(A, \lambda)$ is negligible.

VI. Evaluation

A. Implementation

We implemented our framework on top of the PLONK proof system using Halo2 [55] in Rust. Specifically, we designed 18 custom gates as fundamental operators for constructing chips within the circuit, covering common DNN operations for efficient and flexible circuit representation. We developed 12 operators targeting the ONNX [56] intermediate representation, enabling the system to directly parse ONNX files and build training models, streamlining model import and circuit conversion. For the federated learning scenario, we adopted tonic [57], a high-performance gRPC framework in Rust, to facilitate communication between federated entities. To further improve efficiency, we utilized Rayon [58], a Rust data-parallelism library, enabling concurrent proof generation through multi-threaded execution. The entire ZKSL prototype is implemented in Rust, comprising more than 20,000 lines of code.

B. Experiment Setup

Testbed. Our experiments were conducted on a machine with an AMD Ryzen Threadripper PRO 7975WX CPU (32 cores, 64 threads) and 256GB of RAM, running Ubuntu 22.04. For GPU acceleration, we conducted our experiments on a cloud platform equipped with an AMD EPYC 7543 32-core CPU, 512 GB of RAM, and eight NVIDIA RTX 4090 GPUs (24 GB VRAM each). Computations were performed over the BN256 field to support zero-knowledge proof calculations using the PLONK proof system. Unless otherwise stated, all experiments use a default communication window of $\boldsymbol{K = 1}$ described above. We deployed CNNs across two federated entities, where one holds the data and the other holds the labels, a common real-world scenario where a party with extensive data collaborates with a label provider. For DeepFM, we used a three-party federated setting, simulating multi-institutional joint training, with only one federation(the active party) holding labels. This setting is also widely adopted by other privacy-preserving vertical federated learning works [9, 48, 59].

Workloads. We implemented the **DeepFM** [60] model for financial applications, where feature interactions and deep feature extraction were circuitized using custom gates to enhance efficiency in zero-knowledge proofs. Additionally, we developed **CNN (LeNet / AlexNet / VGG)** [61] model to compare performance and accuracy against other frameworks such as ZKML and KAIZEN. The vertical federated partitioning of these networks, along with their parameter counts and FLOPs, is summarized in Table II. The integration of KZG commitments enables efficient commitment generation and verification while ensuring parameter integrity within the PLONK framework. By leveraging the flexibility of PLONK and the efficiency of KZG commitments, our implementation achieves significant performance improvements while maintaining security.

Datasets. In our experiments, we utilized two distinct datasets tailored to the requirements of our models.

Loan Default [62] dataset is used for training the DeepFM model. It contains approximately 174,000 loan records with 33 features, including credit scores, loan amounts, terms, interest rates, income levels, and debt-to-income ratios. This rich feature set supports the construction of effective credit scoring models for accurate risk assessment.

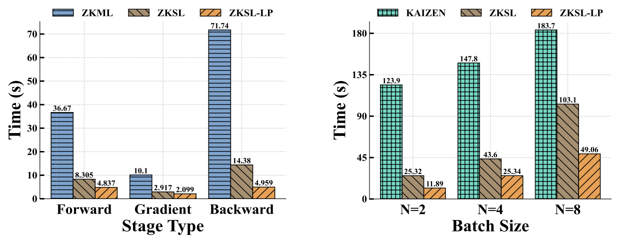
MNIST [63] dataset is used for the LeNet model. It consists of 70,000 grayscale handwritten digit images (60,000 training,

TABLE II: CNN and DeepFM Models and Split.

Model	Party	Layers	#Params	FLOPs
LeNet	Passive (data)	2 Conv	2.5K	0.7M
	Active (label)	3 FC	59K	0.1M
AlexNet	Passive (data)	5 Conv	0.3M	39M
	Active (label)	3 FC	0.2M	0.2M
VGG11	Passive (data)	5 Conv	0.5M	57M
	Active (label)	3 Conv, 3 FC	0.9M	11M
DeepFM	Passive-1 (data)	Embedding	0.2M	0.2M
	Passive-2 (data)	Embedding	0.1M	0.1M
	Passive-3 (data)	Embedding	0.1M	0.1M
	Active (label)	FM, 2 FC	0.1K	0.2K

TABLE III: Single-Node Proving Performance of CNN Models

Model	ZKML	KAIZEN	ZKSL	ZKSL-LP
LeNet	118.5s	123.9s	25.32s	**11.89s**
AlexNet	N/A	213.2s	131.6s	**98.13s**
VGG11	N/A	386.5s	246.9s	**213.7s**



(a) Stage Proof Generation ($N = 2$) (b) Total Proof Generation

Fig. 10: LeNet Training Comparison

10,000 testing), each of size 28×28 and labeled from 0 to 9. Its moderate scale and well-defined structure make MNIST a standard benchmark for evaluating image classification models.

CIFAR-10 [64] dataset is used for AlexNet and VGG11. It includes 60,000 color images of size 32×32 across 10 classes, with 50,000 for training and 10,000 for testing. Due to its higher visual diversity compared to MNIST, CIFAR-10 serves as a challenging benchmark for assessing the representation and generalization capabilities of CNNs.

Comparisons. Since research on zero-knowledge proofs for neural network training is still limited, we selected two representative baselines for comparison:

- ZKML [20]: A Plonk-based system with KZG commitments for verifying deep neural network inference without exposing model parameters. Unlike ZKSL, which verifies the full training process, ZKML is restricted to inference. For a fair comparison, we extend ZKML to support backpropagation.
- KAIZEN [37]: A zkPoT framework based on Multilinear Extensions (MLE) and Sumcheck (GKR-style proofs). While KAIZEN supports training verification, its use of MLE and Sumcheck leads to larger proofs and inherent interactivity, making it unsuitable for federated learning.

Our Methods. We designed multiple versions of ZKSL to evaluate its performance under different optimization strategies.

- ZKSL: The baseline Zero-Knowledge Proof Federated Learning framework.
- ZKSL-G: ZKSL accelerated with GPU.
- ZKSL-LP: ZKSL with hierarchical parallel proof generation.
- ZKSL-LP-G: ZKSL-LP accelerated with GPU.

C. Evaluation of Single-Node

We first focus on the local learning scheme, evaluating proof generation efficiency, computational overhead, and model performance on a single machine. This includes measuring proof latency for *forward propagation*, *gradient computation*, and *backpropagation*, as well as the effects of parallel optimization. We conduct these single-node microbenchmarks to enable a fair comparison with prior systems such as ZKML and Kaizen, which are evaluated only in centralized settings, thereby isolating the cost of the proving backend from federated communication overheads.

We compare the proving time of CNN models across different baseline approaches. Table III summarizes the single-node proving performance of three CNN models under ZKML, KAIZEN, and our proposed ZKSL and ZKSL-LP. ZKSL already offers substantial improvements: for LeNet, it reduces proving time from 118s (ZKML) and 123s (KAIZEN) to 25.32s, showing the effectiveness of our structured circuit optimization. Building on this, ZKSL-LP incorporates hierarchical layer-wise parallelization and further lowers latency to 11.89s, achieving an additional 53% reduction over ZKSL. For AlexNet and VGG11, ZKSL-LP also delivers notable speedups, reducing proving time to 98.13s and 213.7s. The smaller improvement ratios compared to LeNet mainly stem from limited parallel processing capacity on a single machine. With stronger hardware or multi-node parallel proving, ZKSL-LP would yield even larger gains for deeper neural networks.

Figure 10(a) shows the proving time of ZKML, ZKSL, and ZKSL-LP across three stages. ZKSL significantly reduces proving overhead compared to ZKML, achieving 3.6×, 4.7×, and 5.1× speedups for gradient computation, forward propagation, and backward propagation, respectively. Building on ZKSL, ZKSL-LP introduces hierarchical parallel proof generation, further improving efficiency by 57.2%, 75.1%, and 65.2% across the three stages, demonstrating the effectiveness of structured parallelization. Figure 10(b) examines proving time under different batch sizes ($N = 2, 4, 8$). Although proving time increases with batch size, ZKSL-LP consistently outperforms KAIZEN, highlighting the benefit of structured proof optimizations. At $N = 8$, ZKSL-LP reduces proving time by 73.3% compared to KAIZEN and improves upon ZKSL by 52.4%, confirming the advantage of layer-wise parallel proof generation.

D. Evaluation of Federated Learning

Next, we evaluate the efficiency of proof generation in the federated learning setting, focusing on proof latency for both passive-side and active-side computations. This analysis includes the impact of hierarchical parallelization and GPU acceleration. Since other frameworks lack a federated structure, we compare only our own methods to assess scalability and optimization effectiveness.

Federated CNN. Table IV reports the per-round time breakdown of federated CNN training across different proving config-

TABLE IV: Per-round Stage-wise Time Breakdown of Federated CNNs (seconds)

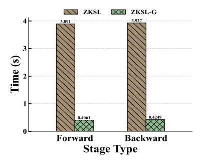
Model	Scheme	Passive Party				Data Transfer	Active Party				
		Compute$_P$	ProveF$_P$	ProveB$_P$	Verify$_P$		Compute$_A$	ProveF$_A$	ProveG$_A$	ProveB$_A$	Verify$_A$
LeNet	**ZKSL**	0.016	8.26	8.61	0.006	0.372	0.017	3.72	2.13	7.18	0.005
	ZKSL-G		0.676	0.665				0.732	0.404	0.741	
	ZKSL-LP		2.66	3.91		0.481		2.35	2.06	2.89	
	ZKSL-LP-G		0.498	0.462				0.532	0.395	0.551	
AlexNet	**ZKSL**	0.161	23.24	116.3	0.006	0.578	0.078	9.56	3.67	51.56	0.006
	ZKSL-G		6.08	8.32				2.29	0.883	1.63	
	ZKSL-LP		26.67	46.44		0.797		5.53	3.59	20.54	
	ZKSL-LP-G		3.78	4.79				1.82	0.832	1.08	
VGG11	**ZKSL**	0.225	63.54	64.67	0.007	1.13	0.175	61.65	3.72	62.54	0.007
	ZKSL-G		6.72	6.79				3.55	0.808	3.95	
	ZKSL-LP		23.06	40.58		1.44		11.74	3.73	22.14	
	ZKSL-LP-G		2.86	4.22				1.48	0.828	1.49	

urations. Compute denotes the time for forward, backward, and gradient computation. ProveF, ProveG, and ProveB represent the proving times for forward propagation, gradient computation, and backward propagation, respectively. Verify denotes verification time per party, while Data Transfer measures the communication cost of exchanging intermediate activations and proofs between passive and active parties.

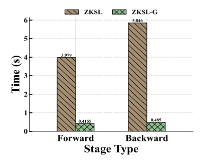
Across all models, ZKSL-LP substantially reduces proving overhead compared to ZKSL by leveraging layer-wise parallel proof generation. For LeNet, ZKSL-LP reduces the passive party's proving time from 8.26s (ProveF) and 8.61s (ProveB) to 2.66s and 3.91s, respectively, with similar improvements on the active side. This demonstrates that parallelism effectively reduces the dominant proving stages in federated settings. The GPU-accelerated variants (ZKSL-G and ZKSL-LP-G) deliver even greater speedups. For LeNet, ZKSL-LP-G lowers the passive party's proving time to just 0.498s (ProveF) and 0.462s (ProveB), and brings active-party proving costs below one second. The benefits of GPU acceleration persist in deeper models such as AlexNet and VGG11: although the overall proving time grows with model size, ZKSL-LP-G consistently achieves the lowest latency across all proving stages. Data-transfer overhead remains relatively small compared with proving time, confirming that proof generation, particularly backward propagation proofs, is the primary bottleneck in federated training. Overall, the combination of layer-wise parallelism (LP) and GPU acceleration (G) provides the most efficient configuration, significantly reducing per-round latency for secure federated CNN learning.

Federated DeepFM. Figure 11 reports the proving time for passive and active parties in DeepFM. The passive party handles embedding layers, while the active party processes feature interaction and fully connected layers. Since the passive party performs only a single embedding lookup, parallelization is infeasible, and we compare only ZKSL and ZKSL-G.

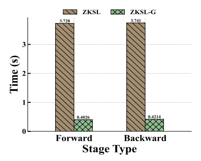
For the passive party (Figs. 11(a)–11(c)), ZKSL incurs 3.891s–3.979s for forward proving and 3.927s–5.846s for backward proving, while ZKSL-G reduces these costs to 0.415s and 0.48s, achieving 9.58× and 12.18× speedups. For the active party (Fig. 11(d)), ZKSL requires 4.945s, 2.144s, and 8.147s for forward, gradient, and backward proving, respectively; ZKSL-LP lowers these times to 2.397s, 2.032s, and 3.54s (51.5%, 5.2%, and 56.5% overhead reduction), and ZKSL-LP-G further accelerates them to 0.818s, 0.435s, and 0.578s, corresponding to 6.04×, 4.93×, and 14.09× speedups over ZKSL.



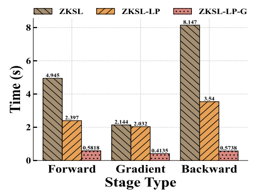
(a) Passive Party 1



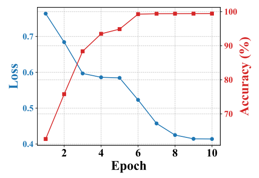
(b) Passive Party 2



(c) Passive Party 3



(d) Active Party



(e) Loss & Accuracy

Fig. 11: DeepFM Federated Training Cost

TABLE V: Performance Comparison on Loan Default Dataset

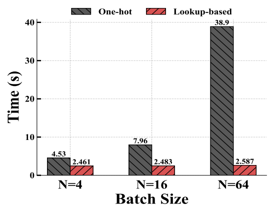
Model	Accuracy	Recall	Precision	F1
ZKSL-DeepFM	99.39%	98.17%	99.38%	98.77%
Pytorch-DeepFM	98.89%	100%	95.68%	97.79%

Fig. 11(e) shows stable convergence across 10 training epochs, with loss decreasing and accuracy rising to 99.4%. Table V further reports that ZKSL-DeepFM achieves 99.39% accuracy and 99.38% precision on the Loan Default dataset, exceeding PyTorch [53] (99.39% vs. 98.89% accuracy). These results confirm that ZKSL-DeepFM maintains competitive model quality even with integrated ZK proofs.

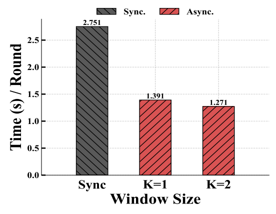
Overall, hierarchical parallelization and GPU acceleration work synergistically to significantly improve proof-generation efficiency and scalability in federated DeepFM.

E. Evaluation of Optimization

This subsection evaluates the effectiveness of our optimization techniques. Figure 12(a) measures the *embedding proving* subroutine on CPU, while Figure 12(b) reports end-to-end per-round LeNet training time on a GPU-enabled setup that incorporates all proposed optimizations, including layer-wise parallelization and asynchronous scheduling.



(a) Embedding Cost



(b) Round Cost

Fig. 12: Comparison of Optimization

Embedding Performance. Figure 12(a) reports embedding proving time for batch size $N \times 1$ with a 20000×9 embedding matrix. The one-hot encoding baseline scales poorly, reaching $38.9\,\text{s}$ at $N = 64$ due to expensive matrix multiplications. In contrast, our lookup-based probability-optimized scheme maintains near-constant latency, with $2.461\,\text{s}$, $2.483\,\text{s}$, and $2.587\,\text{s}$ for $N = 4, 16, 64$, respectively. Thus, our method avoids redundant computation and constraint growth, enabling efficient and scalable embedding verification.

Asynchronous Scheduling. Figure 12(b) shows per-round LeNet training time under synchronous and asynchronous scheduling with window size K. The synchronous baseline is $2.751\,\text{s}$ per round. Asynchronous execution reduces this to $1.391\,\text{s}$ at $K = 1$ (49.4% reduction; $1.98\times$ speedup) and $1.271\,\text{s}$ at $K = 2$ (53.8% reduction; $2.16\times$; 8.6% over $K = 1$). These results show that decoupling proof generation from the training loop and overlapping it within a small window lowers per-round latency. Since our experimental machine could not saturate the proof queue, additional parallel workers or accelerators would likely yield larger speedups.

VII. Conclusion

We introduce ZKSL, a PLONK-based zkSNARK framework that brings verifiable computation to vertical federated learning without sacrificing practicality. By extending a privacy-commitment column into PLONK (PC-PLONK), ZKSL binds each participant's private inputs and parameters to commitments, eliminating the hash-heavy overhead of conventional schemes. The framework stratifies deep neural networks into layer-level circuits whose proofs are generated in parallel, verifies embedding operations with lookup, and overlaps proof generation with training via an asynchronous compute–prove scheduler, yielding a scalable, privacy-preserving, integrity-assured training pipeline for vertical federated learning in finance and healthcare.

Acknowledgement

This work was supported by National Key Research and Development Program of China (No.2023YFC3304700), Program of Shanghai Academic/Technology Research Leader (No.23XD1401100) and Ant Digital Technologies, Ant Group Research Fund.

References

[1] P. Regulation, "Regulation (eu) 2016/679 of the european parliament and of the council," *Regulation (eu)*, vol. 679, p. 2016, 2016.

[2] R. Bonta, "California consumer privacy act (ccpa)," *Retrieved from State of California Department of Justice: https://oag. ca. gov/privacy/ccpa*, 2022.

[3] Y. Liu, Y. Kang, T. Zou, Y. Pu, Y. He, X. Ye, Y. Ouyang, Y.-Q. Zhang, and Q. Yang, "Vertical federated learning: Concepts, advances, and challenges," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 3615–3634, 2024.

[4] C. Thapa, P. C. M. Arachchige, S. Camtepe, and L. Sun, "Splitfed: When federated learning meets split learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 36, no. 8, 2022, pp. 8485–8493.

[5] L. Zhu, Z. Liu, and S. Han, "Deep leakage from gradients," *Advances in neural information processing systems*, vol. 32, 2019.

[6] Z. He, T. Zhang, and R. B. Lee, "Model inversion attacks against collaborative inference," in *Proceedings of the 35th Annual Computer Security Applications Conference*, 2019, pp. 148–162.

[7] Y. Zhang, R. Jia, H. Pei, W. Wang, B. Li, and D. Song, "The secret revealer: Generative model-inversion attacks against deep neural networks," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 253–261.

[8] Y. Huang, S. Gupta, Z. Song, K. Li, and S. Arora, "Evaluating gradient inversion attacks and defenses in federated learning," *Advances in neural information processing systems*, vol. 34, pp. 7232–7241, 2021.

[9] R. Xu, N. Baracaldo, Y. Zhou, A. Anwar, J. Joshi, and H. Ludwig, "Fedv: Privacy-preserving federated learning over vertically partitioned data," in *Proceedings of the 14th ACM workshop on artificial intelligence and security*, 2021, pp. 181–192.

[10] G. Wang, B. Gu, Q. Zhang, X. Li, B. Wang, and C. X. Ling, "A unified solution for privacy and communication efficiency in vertical federated learning," *Advances in Neural Information Processing Systems*, vol. 36, pp. 13 480–13 491, 2023.

[11] Y. Wu, N. Xing, G. Chen, T. T. A. Dinh, Z. Luo, B. C. Ooi, X. Xiao, and M. Zhang, "Falcon: A privacy-preserving and interpretable vertical federated learning system," *Proceedings of the VLDB Endowment*, vol. 16, no. 10, pp. 2471–2484, 2023.

[12] M. Gong, Y. Zhang, Y. Gao, A. K. Qin, Y. Wu, S. Wang, and Y. Zhang, "A multi-modal vertical federated learning framework based on homomorphic encryption," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 1826–1839, 2023.

[13] Z. Chen, Z. Gu, Y. Lu, X. Ren, R. Zhong, W.-J. Lu, J. Zhang, Y. Zhang, H. Wu, X. Zheng *et al.*, "Safe: A scalable homomorphic encryption accelerator for vertical federated learning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[14] "Zcash," 2022. [Online]. Available: https://z.cash/

[15] V. Farzaliyev, C. Pärn, H. Saarse, and J. Willemson, "Lattice-based zero-knowledge proofs in action: Applica-

tions to electronic voting," *Journal of Cryptology*, vol. 38, no. 1, p. 6, 2025.

[16] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical non-interactive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[17] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology–EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II 35*. Springer, 2016, pp. 305–326.

[18] A. Kate, G. M. Zaverucha, and I. Goldberg, "Constant-size commitments to polynomials and their applications," in *International conference on the theory and application of cryptology and information security*. Springer, 2010, pp. 177–194.

[19] R. C. Merkle, "A certified digital signature," in *Conference on the Theory and Application of Cryptology*. Springer, 1989, pp. 218–238.

[20] B.-J. Chen, S. Waiwitlikhit, I. Stoica, and D. Kang, "Zkml: An optimizing system for ml inference in zero-knowledge proofs," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 560–574.

[21] T. Xie, J. Zhang, Y. Zhang, C. Papamanthou, and D. Song, "Libra: Succinct zero-knowledge proofs with optimal prover computation," in *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 2019, pp. 733–764.

[22] J. Zhang, T. Liu, W. Wang, Y. Zhang, D. Song, X. Xie, and Y. Zhang, "Doubly efficient interactive proofs for general arithmetic circuits with linear prover time," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 159–177.

[23] J. Zhang, T. Xie, Y. Zhang, and D. Song, "Transparent polynomial delegation and its applications to zero knowledge proof," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 859–876.

[24] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell, "Bulletproofs: Short proofs for confidential transactions and more," in *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018, pp. 315–334.

[25] N. Bitansky, R. Canetti, A. Chiesa, S. Goldwasser, H. Lin, A. Rubinstein, and E. Tromer, "The hunting of the snark," *Journal of Cryptology*, vol. 30, no. 4, pp. 989–1066, 2017.

[26] J. Kilian, "A note on efficient zero-knowledge proofs and arguments," in *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, 1992, pp. 723–732.

[27] S. Micali, "Computationally sound proofs," *SIAM Journal on Computing*, vol. 30, no. 4, pp. 1253–1298, 2000.

[28] E. Ben-Sasson, A. Chiesa, D. Genkin, E. Tromer, and M. Virza, "Snarks for c: Verifying program executions succinctly and in zero knowledge," in *Annual cryptology conference*. Springer, 2013, pp. 90–108.

[29] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," *Communications of the ACM*, vol. 59, no. 2, pp. 103–112, 2016.

[30] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Advances in Cryptology–EUROCRYPT 2013: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings 32*. Springer, 2013, pp. 626–645.

[31] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 499–530.

[32] A. Gabizon and Z. J. Williamson, "plookup: A simplified polynomial protocol for lookup tables," *Cryptology ePrint Archive*, 2020.

[33] S. Lee, H. Ko, J. Kim, and H. Oh, "vcnn: Verifiable convolutional neural network based on zk-snarks," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 4, pp. 4254–4270, 2024.

[34] T. Liu, X. Xie, and Y. Zhang, "zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2968–2985.

[35] Y. Fan, K. Ma, L. Zhang, X. Lei, G. Xu, and G. Tan, "Validcnn: A large-scale cnn predictive integrity verification scheme based on zk-snark," *IEEE Transactions on Dependable and Secure Computing*, 2024.

[36] H. Sun, T. Bai, J. Li, and H. Zhang, "Zkdl: Efficient zero-knowledge proofs of deep learning training," *IEEE Transactions on Information Forensics and Security*, 2024.

[37] K. Abbaszadeh, C. Pappas, J. Katz, and D. Papadopoulos, "Zero-knowledge proofs of training for deep neural networks," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4316–4330.

[38] S. Garg, A. Goel, S. Jha, S. Mahloujifar, M. Mahmoody, G.-V. Policharla, and M. Wang, "Experimenting with zero-knowledge proofs of training," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 1880–1894.

[39] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Scalable zero knowledge via cycles of elliptic curves," *Algorithmica*, vol. 79, pp. 1102–1160, 2017.

[40] N. Bitansky, R. Canetti, A. Chiesa, and E. Tromer, "Recursive composition and bootstrapping for snarks and proof-carrying data," in *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, 2013, pp. 111–120.

[41] P. Valiant, "Incrementally verifiable computation or proofs

of knowledge imply time/space efficiency," in *Theory of Cryptography: Fifth Theory of Cryptography Conference, TCC 2008, New York, USA, March 19-21, 2008. Proceedings 5.* Springer, 2008, pp. 1–18.

[42] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[43] J. Bell, A. Gascón, T. Lepoint, B. Li, S. Meiklejohn, M. Raykova, and C. Yun, "{ACORN}: input validation for secure aggregation," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 4805–4822.

[44] H. Lycklama, L. Burkhalter, A. Viand, N. Küchler, and A. Hithnawi, "Rofl: Robustness of secure federated learning," in *2023 IEEE Symposium on Security and Privacy (SP).* IEEE, 2023, pp. 453–476.

[45] A. Roy Chowdhury, C. Guo, S. Jha, and L. van der Maaten, "Eiffel: Ensuring integrity for federated learning," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022, pp. 2535–2549.

[46] B. Gu, A. Xu, Z. Huo, C. Deng, and H. Huang, "Privacy-preserving asynchronous vertical federated learning algorithms for multiparty collaborative learning," *IEEE transactions on neural networks and learning systems*, vol. 33, no. 11, pp. 6103–6115, 2021.

[47] Y. Wu, N. Xing, G. Chen, T. T. A. Dinh, Z. Luo, B. C. Ooi, X. Xiao, and M. Zhang, "Falcon: A privacy-preserving and interpretable vertical federated learning system," *Proceedings of the VLDB Endowment*, vol. 16, no. 10, pp. 2471–2484, 2023.

[48] Y. Wu, S. Cai, X. Xiao, G. Chen, and B. C. Ooi, "Privacy preserving vertical federated learning for tree-based models," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2090–2103, 2020.

[49] S. Li, D. Yao, and J. Liu, "Fedvs: Straggler-resilient and privacy-preserving vertical federated learning for split models," in *International conference on machine learning*. PMLR, 2023, pp. 20 296–20 311.

[50] H. Gu, J. Luo, Y. Kang, L. Fan, and Q. Yang, "Fedpass: Privacy-preserving vertical federated deep learning with adaptive obfuscation."

[51] L. Wang, Z. Zhang, M. Huang, K. Gai, J. Wang, and Y. Shen, "Ropa: Robust privacy-preserving forward aggregation for split vertical federated learning," *IEEE Transactions on Network and Service Management*, 2025.

[52] S. Wang, K. Gai, J. Yu, and L. Zhu, "Bdvfl: Blockchain-based decentralized vertical federated learning," in *2023 IEEE International Conference on Data Mining (ICDM).* IEEE, 2023, pp. 628–637.

[53] "Pytorch," 2025. [Online]. Available: https://pytorch.org/

[54] "Ezkl," 2022. [Online]. Available: https://github.com/zkonduit/ezkl

[55] "Halo2," 2022. [Online]. Available: https://github.com/zcash/halo2

[56] "Open neural network exchange," 2025. [Online]. Available: https://onnx.ai/

[57] "tonic," 2025. [Online]. Available: https://github.com/hyperium/tonic

[58] "rayon," 2025. [Online]. Available: https://github.com/rayon-rs/rayon

[59] D. Zhu, J. Chen, X. Zhou, W. Shang, A. E. Hassan, and J. Grossklags, "Vulnerabilities of data protection in vertical federated learning training and countermeasures," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 3674–3689, 2024.

[60] H. Guo, R. Tang, Y. Ye, Z. Li, and X. He, "Deepfm: a factorization-machine based neural network for ctr prediction," *arXiv preprint arXiv:1703.04247*, 2017.

[61] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[62] "Loan default dataset," 2022. [Online]. Available: https://www.kaggle.com/datasets/yasserh/loan-default-dataset

[63] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE signal processing magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[64] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009. [Online]. Available: https://www.cs.toronto.edu/~kriz/cifar.html

VIII. Appendix

A. Lookup-based Embedding Theorems and Proof

Theorem 1. *Let $E \in \mathbb{F}_p^{n\times d}$ be an embedding matrix, where each row $E_i \in \mathbb{F}_p^d$ represents an embedding vector. Let $r \in \mathbb{F}_p^d$ be a uniformly random vector over the finite field $\mathbb{F}_p$. Suppose a prover returns a vector $\hat{E}_i \in \mathbb{F}_p^d$ in response to a query for E_i. If $\hat{E}_i \neq E_i$, then the probability that $\hat{E}_i \cdot r = E_i \cdot r$ is at most $\frac{1}{p}$.*

Proof. Let $d = E_i - \hat{E}_i$. Since $\hat{E}_i \neq E_i$, it follows that $d \neq \mathbf{0}$. Consider the dot product:

$$d \cdot r = \sum_{j=1}^{d} d_j r_j,$$

where $d = (d_1, d_2, \ldots, d_d)$ and $r = (r_1, r_2, \ldots, r_d)$. Since $d \neq \mathbf{0}$, at least one coefficient d_j is nonzero, so $d \cdot r$ is a nonzero linear form in the components of r. A basic fact in finite fields is that for any nonzero linear form $L(r)$ over $\mathbb{F}_p$, the probability that $L(r) = 0$ is exactly $\frac{1}{p}$ when r is chosen uniformly at random.

Thus, we have

$$\Pr[d \cdot r = 0] = \frac{1}{p}.$$

Since $\hat{E}_i \cdot r = E_i \cdot r$ implies that $d \cdot r = 0$, it follows that

$$\Pr[\hat{E}_i \cdot r = E_i \cdot r] = \Pr[d \cdot r = 0] = \frac{1}{p}.$$

B. Batch Complexity under Lookup-based Embedding

Theorem 2 (From $O(Nnd)$ to $O(nd)$)**.** *Let $E \in \mathbb{F}^{n\times d}$ be an embedding matrix and suppose a batch of N index queries* *is verified in zero knowledge. Measure cost by the number of multiplicative (or lookup) constraints in a PLONKish circuit.*

(1) *(*One-hot baseline*) The batch verification cost is*

$$C_{\text{one-hot}} = \Theta(Nnd).$$

(2) *(*Lookup-based approach*) Sample a single random $r \in \mathbb{F}^d$, form once the table $\{(i, \langle E_i, r\rangle)\}_{i=1}^n$, and verify each query by a lookup against this table. The batch cost is*

$$C_{\text{lookup}} = \Theta(nd + Nd^2).$$

Under the large-vocabulary regime $n \gg Nd$, we have $C_{\text{lookup}} = \Theta(nd)$. Hence the overall complexity drops from $O(Nnd)$ to $O(nd)$; equivalently, the amortized per-query cost drops from $O(nd)$ to $O(1)$, i.e., from $O(N)$ to $O(1)$ after normalizing by the common factor nd.

Proof. Item (1) is standard: for each of the N queries and each output coordinate (d of them), the computation

$$Y_{i,j} = \sum_{k=1}^{n} X_{i,k}\, E_{k,j}$$

incurs n multiplicative/nonlinear constraints, yielding $\Theta(Nnd)$ in total (the one-hotness constraints $\sum_k X_{i,k} = 1$ and $X_{i,k}^2 = X_{i,k}$ are lower order).

For item (2), forming $v_i = \langle E_i, r\rangle = \sum_{j=1}^d E_{i,j} r_j$ for all $i \in [n]$ costs $\Theta(nd)$ once, and each query is then checked by a constant-cost lookup (plus at most $O(d^2)$ local work), so $C_{\text{lookup}} = \Theta(nd + Nd^2)$. Under $n \gg Nd$,

$$C_{\text{lookup}} = nd + Nd^2 = nd\left(1 + \frac{Nd}{n}\right) = \Theta(nd).$$

Therefore,

$$\frac{C_{\text{one-hot}}}{C_{\text{lookup}}} = \Theta\left(\frac{Nnd}{nd}\right) = \Theta(N),$$

which exactly states the drop from $O(Nnd)$ to $O(nd)$ in batch cost.

C. Combined Soundness of Permutation and Commitment in PC-PLONK

Problem Setting. In PC-PLONK, correctness of private values used in the circuit must be ensured in two ways:

- **In-circuit consistency**: Enforced via the PLONK permutation argument to ensure that all occurrences of a private value are consistent across the circuit.
- **External binding to origin**: Enforced via a cryptographic commitment (e.g., Merkle root or KZG polynomial commitment) to ensure that all private values originate from a committed source.

Circuit and Commitment Model. Let the prover maintain a private column $P = \{v_1, v_2, \dots, v_m\} \in \mathbb{F}^m$ containing all private inputs and parameters.

- The prover computes a commitment $C = \text{Com}(P)$ using a binding scheme (Merkle or KZG).
- Each circuit witness w_i must satisfy $w_i = P[\sigma(i)]$ for a fixed permutation σ mapping circuit locations to P.

Combined Constraint System. The verifier enforces two simultaneous constraints:

(a) Permutation Argument. Let $f(X)$ be the witness polynomial evaluated on a multiplicative subgroup $H = \{\omega^i\}_{i=0}^{N-1}$. The grand product polynomial $Z(X)$ satisfies

$$Z(\omega^{i+1}) = Z(\omega^i) \cdot \frac{f(\omega^i) + \beta \cdot \omega^i + \gamma}{f(\sigma(\omega^i)) + \beta \cdot \sigma(\omega^i) + \gamma}$$

for random $\beta, \gamma \in \mathbb{F}$, ensuring $f(\omega^i) = f(\sigma(\omega^i))$ unless verification fails.

(b) Commitment Binding. Each value $P[j]$ must be provably bound to the public commitment C. This is done via a KZG opening proof for $P(j)$ under the committed polynomial.

Theorem 3. *Let $P \in \mathbb{F}^m$ be the column of private values, and $C = Com(P)$ its public commitment. Suppose all witness values w_i in the circuit are constrained via a permutation to entries in P, and all openings are properly verified against C. Then, any attempt to forge w_i or alter P without breaking either:*

1) the soundness of the PLONK permutation argument
2) the binding of the commitment scheme,

will be detected by the verifier with overwhelming probability.

Proof Sketch. *Case 1: Forging inconsistent values in the circuit.* If $f(\omega^i) \neq f(\sigma(\omega^i))$ for some i, the permutation check fails with high probability due to the Schwartz–Zippel lemma applied over the random challenges β and γ.

Case 2: Modifying committed values in P. If P is altered but the prover reuses old openings:

- For Merkle: a second-preimage/collision is required to match the root, contradicting hash collision resistance.
- For KZG: a forged polynomial opening is required, which breaks KZG binding under the discrete-log assumption.

Conclusion. By combining the PLONK permutation argument with a binding commitment over the private column, PC-PLONK ensures:

- In-circuit consistency of all private values, and
- External verifiability that those values originate from a committed and unforgeable source.

This combined mechanism guarantees correctness and prevents witness forgery, under standard cryptographic assumptions.

D. Proof of Unified Security (reduction to PLONK soundness and commitment binding)

In this section we prove the unified security claim (Computation Integrity, Update Verifiability and Public Auditability) by reduction to two underlying assumptions: PLONK (zkSNARK) soundness and commitment binding. The proof refers to the combined reduction algorithm 4 $\mathcal{R}$ which simulates the unified experiment with an adaptive $\mathsf{Corrupt}(\cdot)$ oracle, uses a PLONK prover/simulator to generate honest proofs, records all commitment openings, and upon observing a violating accepted object at a sampled target (i^*, t^*, c^*) attempts to extract either (a) a PLONK forgery for a false statement or (b) two distinct valid openings for the same commitment (a binding break).

Algorithm 4: Reduction $\mathcal{R}$ for PLONK soundness & Commitment binding

```
Input: PLONK challenger (public params / prover-simulator), adversary $\mathcal{A}$.
Output: Either a PLONK forgery $(\mathsf{stmt}^*, \pi^*)$ with $\mathsf{Verify}_{\mathrm{PLONK}}(\mathsf{stmt}^*, \pi^*) = 1$ and $\mathsf{stmt}^* \notin \mathcal{L}$, or two openings $(v_1, v_2)$ s.t.
        $\mathsf{Commit}(v_1) = \mathsf{Commit}(v_2)$.
Choose random target party $i^* \in \{0, \ldots, n\}$ and target round $t^* \in \{1, \ldots, T\}$;
Initialize corrupted set $\mathcal{C} \leftarrow \emptyset$, storage tables State, Openings;
Provide A with oracle $\mathsf{Corrupt}(i)$ implemented as: begin
    if $i \in \mathcal{C}$ then
        return "already corrupted";
    if commitment for i not yet posted (pre-commit) then
        Request $(D_i, r_i)$ from $\mathcal{A}$; compute and publish $\mathsf{Com}_i \leftarrow \mathsf{Commit}(D_i; r_i)$;
    else
        Return stored internal state (including $D_i, r_i$) to $\mathcal{A}$;
    Mark $i$ as corrupted: $\mathcal{C} \leftarrow \mathcal{C} \cup \{i\}$;
For each party $i$ not pre-corrupted: sample honest $(D_i, r_i)$; compute and publish $\mathsf{Com}_i$; store $(D_i, r_i)$ in State;
Initialize global model state $S_0$;
Run $\mathcal{A}$; answer Corrupt queries as above. For each round $t = 1, \ldots, T$ do: begin
    For each party $i \notin \mathcal{C}$: sample local randomness $\eta_i^t$; compute $(e_i^t, g_i^t, \mathsf{aux}_i^t) \leftarrow \mathsf{Exec}_i(S_{t-1}, D_i; \eta_i^t)$; generate PLONK proof $\pi_i^t$ for
      statement "$\exists D, r, \eta :\ \mathsf{Commit}(D; r) = \mathsf{Com}_i \wedge (e_i^t, g_i^t, \mathsf{aux}_i^t) = \mathsf{Exec}_i(S_{t-1}, D; \eta)$" using prover/simulator; append
      proof/opening to transcript; record any opening info for $\mathsf{Com}_i$ in Openings; For each party $j \in \mathcal{C}$: accept arbitrary
      messages/proofs/openings from $\mathcal{A}$ for $(j, t)$ and append to transcript; record any opening-like data into Openings; Run the
      protocol verifier Verify on the round transcript and $S_{t-1}$. If Verify accepts, obtain accepted set $\mathsf{Accepted}^t$ and update $S_t$ per
      protocol; else set $S_t \leftarrow S_{t-1}$; If Verify accepted and there exists an accepted tuple $(i^*, t^*, e, g, \mathsf{aux}, \pi)$ in $\mathsf{Accepted}^{t^*}$ such that
      $(e, g, \mathsf{aux})$ is not explainable by $\mathsf{Exec}_{i^*}$ on any $D$ consistent with $\mathsf{Com}_{i^*}$ then: begin
        Let stmt be the PLONK statement validated by $\pi$;
        if stmt is false (no witness exists) then
            return $(\mathsf{stmt}, \pi)$ ;                                        // PLONK soundness forgery
        else
            Search Openings for two entries $(\mathsf{Com}_{i^*}, \mathrm{opening}_a, \mathrm{val}_a)$ and $(\mathsf{Com}_{i^*}, \mathrm{opening}_b, \mathrm{val}_b)$ with $\mathrm{val}_a \neq \mathrm{val}_b$ and both
              validating to $\mathsf{Com}_{i^*}$;
            if found such $(val_a, val_b)$ then
                return $(\mathrm{val}_a, \mathrm{val}_b)$ ;                              // Commitment binding break
return FAIL;
```

Proof. Let $\varepsilon := \mathsf{Adv}_{\mathsf{Unified}}(A, \lambda)$ be the probability that the unified experiment outputs a failure. Run the reduction $\mathcal{R}$ referenced above which simulates the unified experiment for A, implements the adaptive oracle $\mathsf{Corrupt}(\cdot)$ faithfully, uses the PLONK simulator to generate honest proofs, and records all commitment openings.

1. (Sampling / averaging.) $\mathcal{R}$ selects uniformly at random a target location u among the M checked-object locations (i.e. a triple (i^*, t^*, c^*)). If A causes a failure somewhere with probability ε, then by averaging the probability that the failure occurs at the sampled u is at least ε/M.

2. (Decomposition of failure event.) Condition on a failure at the sampled location u. Since the verifier accepted an object that is invalid (not explainable by the declared local execution on any data consistent with the published commitment), at least one of the verifier's primitive checks was circumvented. Restricting to PLONK and commitments, the only possibilities are: (a) PLONK verification accepted a false statement (soundness break), or (b) an inconsistent opening / two different openings for the same commitment was accepted (commitment binding break). Hence the failing event at u implies the occurrence of $\mathcal{E}_{\mathrm{PLONK}} \vee \mathcal{E}_{\mathrm{ComBind}}$.

3. (Extraction and simulation error.) When $\mathcal{R}$ observes a failure at u it attempts extraction: if the acceptance relied on a PLONK proof for a false statement, $\mathcal{R}$ outputs that proof as a PLONK forgery; otherwise $\mathcal{R}$ searches recorded openings and, if two distinct valid openings for the same commitment are found, outputs them as a binding break. The simulation uses the PLONK simulator / CRS trapdoor as assumed; the distinguishing / extraction failure probability introduced by simulation is bounded by $\delta_{\mathsf{sim}}(\lambda)$.

4. (Probability accounting.) Let p_{PLONK} and p_{ComBind} be the probabilities that $\mathcal{R}$ successfully outputs a PLONK forgery or a binding break, respectively. From steps 1–3 we obtain

$$p_{\mathrm{PLONK}} + p_{\mathrm{ComBind}} \ \geq\ \frac{\varepsilon}{M} - \delta_{\mathsf{sim}}(\lambda).$$

By definition $p_{\mathrm{PLONK}} \ \leq\ \mathsf{Adv}_{\mathsf{PLONK}}(\lambda)$ and $p_{\mathrm{ComBind}} \ \leq\ \mathsf{Adv}_{\mathsf{ComBind}}(\lambda)$, hence

$$\mathsf{Adv}_{\mathsf{PLONK}}(\lambda) + \mathsf{Adv}_{\mathsf{ComBind}}(\lambda) \ \geq\ \frac{\varepsilon}{M} - \delta_{\mathsf{sim}}(\lambda).$$

Rearranging gives

$$\varepsilon \ \leq\ M \cdot \big(\mathsf{Adv}_{\mathsf{PLONK}}(\lambda) + \mathsf{Adv}_{\mathsf{ComBind}}(\lambda)\big) \ +\ M \cdot \delta_{\mathsf{sim}}(\lambda).$$

5. (Conclusion.) Absorbing the polynomial factor into the negligible simulation term yields the stated bound with $\delta_{\mathsf{sim}}(\lambda)$ replaced by a (still negligible) $\delta'_{\mathsf{sim}}(\lambda)$, completing the proof.

ARTIFACT APPENDIX

A. Description & Requirements

We release the ZKSL prototype and scripts to reproduce all core results in our paper: layer-wise parallel proving, the PC-PLONK privacy-commitment column, probabilistic verification for embedding layers, and the asynchronous compute-prove scheduling. The repository contains complete pipelines for split/federated training and for proof generation/verification, plus ready-to-run configurations for *LeNet* and *DeepFM*.

1) How to access:

- Code repository (source, build/run guides, configs): https://anonymous.4open.science/r/ZKSL-FA48
- Artifact package (code + configs + sample data splits + pre-generated example outputs): **Submission (Option 1) - Packaged Artifact** [2]

2) Hardware dependencies:

- **CPU:** AMD Ryzen Threadripper PRO 7975WX CPU (32 cores, 64 threads)
- **GPU:** NVIDIA RTX 4090 or higher
- **RAM:** 256GB or higher

2) Software dependencies:

- **Operating System:** Ubuntu 22.04
- **Python:** Python 3.10
- **PyTorch:** PyTorch 2.0.1
- **Rust:** Rust nightly-2024-10-30 (for the PLONK/Halo2 implementation)
- **NVIDIA:** Driver Version 580.95.05 & CUDA Version 13.0
- **Key Crates** (declared in `Cargo.toml`): Halo2, Rayon (parallelism), tonic (gRPC), ONNX operator subset

Benchmarks. Default scripts cover end-to-end time for *LeNet* (two-party) and *DeepFM* (three-party), per-stage breakdown (forward/gradient/backward), batch-size scaling, and comparisons to baseline systems.

B. Artifact Installation & Configuration

1) Build. Clone the repo and run:

```
cargo build --release
```

This fetches Halo2/PLONK dependencies on first build.

2) Datasets.

- *Loan Default* (for DeepFM): ∼33 features, ∼174K records.
- *MNIST* (for CNN variants): 70K images (28×28 grayscale).

Each config includes dataset paths and train/test splits aligned with the paper.

3) Configuration. Users can modify the model definition files under the corresponding example directories (e.g., `examples/lenet/model.py`) to adjust network architectures or parameters. After editing, the model can be exported to the ONNX format by executing `python model.py`. Subsequently, `converter.py` converts the generated ONNX file into a `model.json` format compatible with our Rust-based ZKSL framework. Finally, the corresponding binary executable within the project (`cargo run --release --bin <project_name>`) can be invoked to perform inference and verification.

[2]https://doi.org/10.5281/zenodo.17866076

C. Experiment Workflow

Our workflow has five stages:

1) **Collect/Prepare:** Prepare data and apply federated partitioning (passive parties hold features; the active party holds labels).
2) **Model Setup:** Select model/operators, build the PLONK-ish circuit and the PC-PLONK privacy-commitment column, publish commitments.
3) **Proving (Layer-wise):** Generate proofs per layer in parallel, enforce inter-layer equality via commitments, then aggregate.
4) **Async Scheduling:** Enable decouple computation and proving; proofs for round t are verified in round $t+1$.
5) **Evaluation:** Record per-stage costs, end-to-end time, scaling, accuracy, and compare against baselines.

D. Major Claims

- **(C1) Layer-wise parallel proving reduces latency** and scales better than monolithic circuits.
- **(C2) PC-PLONK integrates commitments efficiently** while preserving privacy and enforcing cross-layer consistency. *A baseline that integrates commitments via in-circuit hashing (i.e., without resorting to off-circuit hashing) is essentially infeasible at our scale; conversely, moving hashes outside the circuit directly breaks the ZK trust model and is not a valid baseline. Therefore, we report absolute runtimes for PC-PLONK and omit a head-to-head runtime comparison for this part.*
- **(C3) Probabilistic embedding checks cut complexity to $\mathcal{O}(nd)$** with respect to the number of categorical fields n and the embedding dimension d, shrinking circuit cost.
- **(C4) Asynchronous Compute–Prove improves throughput** by avoiding training stalls on proof generation.

E. Evaluation

Unless stated otherwise, default parameters follow Table VI. Each task provides three blocks: *Preparation*, *Execution*, and *Results*.

E1) *Impact of Layer-wise parallel*

[Preparation] Build the project in release mode with all parties on the same commit/toolchain; use the paper-aligned dataset splits with fixed batch size, number of iterations, and random seeds; configure reachable IP/ports in the configuration; **before launching any party, run `bash net.sh` on each machine to simulate network fluctuations (jitter/latency)**; enable layer-wise proving by setting the code constant `PARALLEL=true` (set `PARALLEL=false` for the non–layer-wise baseline) with `K=1`; and capture logs so that each stage's "Proof Generation Time" can be aggregated later.

[Execution — LeNet (2-party)] Start the two parties on separate terminals/machines (MLP side and Conv side). After handshake, the system proceeds through `forward`, `gradient`, and `backward`, and each stage prints its *Proof Generation Time* to the logs.

```
cargo run -r --bin test_cnn_mlp
cargo run -r --bin test_cnn_conv
```

[Execution — DeepFM (3-party)] Launch the server first, then start three clients with IDs 0, 1, and 2 on their respective terminals/machines; each stage (`forward`, `gradient`, `backward`) emits a *Proof Generation Time* entry in the logs.

```
cargo run -r --bin test_deepfm_server
cargo run -r --bin test_deepfm_client 0
cargo run -r --bin test_deepfm_client 1
cargo run -r --bin test_deepfm_client 2
```

[Results] Consistent with Figs. 12 and 13, enabling hierarchical (layer-wise) parallelism (`PARALLEL=true`) significantly reduces *Proof Generation Time* in the `forward` and `backward` stages across LeNet and DeepFM, compared to a non–layer-wise configuration (`PARALLEL=false`).

E2) Benefit of probabilistic embedding checks

[Preparation] On *DeepFM*, fix the embedding weight matrix to 20000×9; keep all other hyperparameters, seeds, and data splits identical across runs. Evaluate two implementations: the baseline `one-hot + mm` (default) and the probabilistic lookup variant enabled by flag `-p`. Enable logging to capture per-run "Proof Generation Time" for the embedding proving step.

[Execution] Hold all parameters fixed and vary the batch size $N \in \{4, 16, 64\}$.

```
# one-hot + mm (baseline)
cargo run -r --bin test_emb -- -N 4

# probabilistic projection + lookup
cargo run -r --bin test_emb -- -N 4  -p
```

[Results] Consistent with Fig. 14(a), the `one-hot + mm` baseline shows a clear increase in proving time as the batch size N grows, whereas the probabilistic projection+lookup keeps the embedding proving time nearly constant across N. This indicates that the probabilistic method removes redundant matrix multiplications and prevents constraint growth, enabling scalable and efficient ZK embedding verification.

E3) Impact of Asynchronous Compute–Prove

[Preparation] Fix the model (LeNet) and batch size; keep seeds and all other hyperparameters identical; run a synchronous baseline ($K = 0$) and asynchronous runs with window sizes $K \in \{1, 2\}$; enable logging of per-round end-to-end time.

[Execution] Use two machines/terminals and launch the same two binaries as in E1 (`test_cnn_mlp` and `test_cnn_conv`). Repeat the run for $K = 0, 1, 2$ (set K via config or CLI).

TABLE VI: Default settings used in our experiments.

Parameter	**Default**
Proof system	PLONK (Halo2, Rust)
Field	BN256
Async window (K)	1
LeNet parties	2 (1 passive + 1 active)
DeepFM parties	3 (active holds labels)
Datasets	Loan Default (DeepFM), MNIST (CNNs)
Parallel proving	Layer-wise(on)
Embedding check	Prob. projection + lookup (on)
Reference HW	High-core CPU / $\geq$128 GB RAM / 24 GB GPU

```
cargo run -r --bin test_cnn_mlp -- -k 0
cargo run -r --bin test_cnn_conv -- -k 0
```

Collect per-round times and rollback counts for each K.

[Results] Compared to the synchronous baseline, enabling asynchronous execution with a small window substantially lowers per-round latency and improves throughput; $K{=}1$ already brings a large gain, and $K{=}2$ provides additional but diminishing improvement while slightly delaying checkpoints.

F. Customization

Key knobs exposed in configs:

- **Federation & roles:** `num_parties`, active vs. passive roles.
- **Parallelism & scheduling:** op_per_subgraph, parallel, K-window size.
- **Proof system:** GPU enable/disable, proof aggregation, logging verbosity.
- **Model & data:** operator set, dataset paths/splits, batch size, epochs, learning rate.
- **Security/auditing:** output commitments and proof logs for offline re-verification.

G. Notes

- All experiments use a PLONK/Halo2 implementation over BN256 by default; asynchronous strategy defaults to $K{=}1$ unless changed.
- Federated communication uses gRPC (`tonic`); multithreading uses Rayon. Both are pre-wired in the repo.
- Hardware/driver differences may cause variance.