Mario: Multi-round Multiple-Aggregator Secure Aggregation with Robustness against Malicious Actors

Truong Son Nguyen
Arizona State University

Tancrède Lepoint
Amazon Web Services Inc

Ni Trieu
Arizona State University

Abstract

Federated Learning (FL) enables multiple clients to collaboratively train a machine learning model while keeping their data private, eliminating the need for data sharing. Two common approaches to secure aggregation (SA) in FL are the single-aggregator and multiple-aggregator models.

Existing multiple-aggregator protocols such as Prio (NSDI 2017), Prio+ (SCN 2022), Elsa (S&P 2023) either offer robustness only in the presence of semi-honest servers or provide security without robustness and are limited to two aggregators. We introduce Mario, the first multi-aggregator SA protocol that is both secure in a malicious setting and provides robustness. Similar to prior work of Prio and Prio+, Mario provides secure aggregation in a setup of n servers and m clients. Unlike previous work, Mario removes the assumption of semi-honest servers, and provides a complete protocol with robustness against less than $n/2$ malicious servers, defense with input validation of upto $m-2$ corrupted clients, and dropout of any number of clients. Our implementation shows that Mario is 3.40× and 283.4× faster than Elsa and Prio+, respecitively.

1 Introduction

Federated learning (FL) allows multiple clients to collaboratively train a machine learning model while keeping their data private on-device. A crucial component is secure aggregation (SA), which computes aggregated model without revealing individual client models. While many SA protocols have been proposed in the past decade [11, 12, 36, 43, 44, 45, 53, 64, 68, 69], they have limitations. First, many use the blueprint from SecAgg [64], which requires pairwise keys between client for each aggregation rounds and increases the total number of rounds of communication. Recently, Flamingo [53] introduced a "multi-round single-server" solution. Flamingo only requires a one-time setup phase and can perform multiple secure aggregation rounds with a single server. However, it has a limitation: a predefined set of parties must participate in the key generation process. This constraint is problematic for large-scale FL scenarios where the participating parties can dynamically change over time. Secondly, there is a lack of emphasis on asynchronous (AsyncFL) updates, allowing late model updates to contribute to the training process. Existing works [57, 67] on AsyncFL have scaling issues, in particular for lightweight devices, which limits their applicability.

More importantly, recent research highlights that a single-server setup is not secure for clients. Loki [71] demonstrated an attack where a malicious server, by modifying the network architecture, can extract inputs from honest clients. This attack is difficult to defend against in a single-server setup because clients lack the ability to verify the integrity of the model architecture and weight assignments they receive. A malicious server can easily disguise compromised weight matrices by embedding them with noisy row or column vectors.

Thus, it is desirable to consider a multiple servers setup, where the servers share the same model architecture and weights so that honest servers could tell malicious action from malicious ones. Unfortunately, existing multiple-aggregator protocols [5, 10, 16, 26, 31, 39, 40, 50, 63], which eliminate client-to-client communication, have focused mainly on the semi-honest settings (like Prio [26], Prio+ [5, 39, 40]) or two-party malicious settings (like Poplar [16], Popstar [50], Elsa [63]). Recent work such as Willow [13] can be extended to multiple servers, but only addresses the case where a single server performing aggregation is malicious. This study tackles designing an SA protocol that solves all three main challenges: one-time key setup with dynamic client inclusion, asynchronous updates, and multiple malicious aggregators.

In contrast to existing approaches in multiple-aggregator schemes, which use secret sharing or multi-party computation, we leverage an additive homomorphic encryption (ThHE) scheme based on the ring-learning-with-errors (RLWE) problem. We introduce a new variant, referred to as dynamic ThHE, which supports dynamic party participation in a malicious setting, thereby enhancing existing multi-aggregator schemes that are currently limited to two parties [16, 50, 63]. This approach is also advantageous when compared to the

FL Protocol	Malicious Server	Malicious Client	Robust w. Mal. Server	Async. support	Multi server
SecAgg+ [12]	✓	✗	✗	✗	○
ACORN [11]	✓	✓	✗	✗	○
BASecAgg [67]	✗	✗	✗	✓	○
Flamingo [53]	✓	✗	✗	✗	◐
Prio+ [5]	✗	✓	✗	✗	●
ELSA [63]	✓	✓	✗	✗	2
Mario (Ours)	✓	✓	✓	✓	●

Table 1: **Qualitative Comparison of FL Protocols**. Only representative works are shown. Malicious Client indicates scheme support input validation; For multi-server column, ● indicates multi-server support, ◐ indicates single-server decryptor-committee setting, '2' indicates two-server only setting, ○ indicates single server.

single-aggregator model. Concretely, it eliminates the need to re-run key setup when new clients join which is a limitation of Flamingo [53]. It achieves constant amortized communication and computation costs per client, unlike SecAgg++ [12], Flamingo [53] and ACORN [11] with logarithmic client communication costs. It also supports both synchronous and asynchronous FL settings and prevents malicious servers from distributing inconsistent models [60] or customized convolutional kernels [71] to clients.

While the past year has seen a surge in research about lattice-based ThHE [9, 17, 18, 24, 27, 55], state-of-the-art schemes face practical limitations when used in federated learning. These limitations include: (1) the requirement for a trusted key setup or malicious DKG to protect against malicious servers in multi-server FL; (2) the inability to add new servers during computation, a crucial feature for ensuring robustness in multi-server FL. Thus, the direct use of such methods lacks both secure and dynamic performance. This work follows the ThHE framework of [55] and proposes a simple scheme that simultaneously satisfies a compactness property (i.e., the ciphertext size is independent of the number of participants), a dynamic property, and has a decentralized key generation.

Problem Statement. The ideal functionality of multiple-aggregator secure aggregation in the synchronous FL (SyncFL) and asynchronous FL (AsyncFL) is described as follows. Let $\{U_1,\dots,U_m\}$ be a set of clients, each holding a secret input v_i (often a vector or tensor in the FL setting). The protocol Π is a secure SA if it securely computes $v = \sum_{i\in I} v_i$, where $I = [m]$ in the SyncFL setting, and $I \subset [m]$ is a dynamic set of size k in the AsyncFL setting. The secure aggregation computation makes use of n servers $P_{i\in[n]}$. Figure 1 presents the ideal functionality for multi-aggregator SA.

Threat Model and Setting. Following the model in SecAgg+ [12], Elsa [63], ACORN [11], we consider a scenario where at least two clients remain honest, while the other $n-2$ clients may be malicious or controlled by malicious servers. This scenario sets the lower bound for the number of

PARAMETERS: m clients $\{U_1,\dots,U_m\}$, n servers $\{P_1,\dots,P_n\}$, a threshold $t < n$ and set size $k \in [1,m]$.

FUNCTIONALITY:

- Waiting for input v_i from the $U_{i\in I}$, where $I = [m]$ in the SyncFL settings and $I \subset [m]$ is the dynamic set of size k in the AsyncFL settings.
- Waiting for no input from the servers $P_{i\in[n]}$.
- Give each client $v = \sum_{i=1}^{m} v_i$

Figure 1: Multi-server Secure Aggregation (SA) Ideal Functionality

honest clients in secure aggregation (SA). This is because if $n-1$ parties are malicious, the adversary can learn the honest party's input from the SA output.

For the server setting, we assume that an adversary can control a static set of up to $t-1$ malicious servers throughout the entire protocol execution. While this static set assumption is strong, it is worth noting that this is the first work to address the challenge of malicious multi-party servers, and removing this assumption can be explored in future work. Nevertheless, we introduce a mechanism to reset the shares of the secret keys using the new function of ThHE (i.e., ThHE.ShareRefresh), which mitigates the impact of this assumption by allowing the static set of malicious servers to be reset when the share refresh is called after a few iterations.

Our Mario is robust to up to any number of client dropouts and up to $n-t$ servers dropouts during execution. We summarize the properties and parameters of our scheme based on the threat model in Table 2. The parameters represent the ideal values that the best possible protocol can achieve.

- *Malicious clients*: We consider the threat of malicious clients that may deviate from the protocol to gain additional information, such as other clients' model updates, or send manipulated encrypted models that could bias the final aggregated result. In our protocol, we defend against these threats by implementing input validation, which allows us to ignore invalid inputs from malicious clients.
- *Malicious servers*: We consider malicious servers that may deviate from the protocol to try to recover the raw client model updates v_i. Our scheme defends against two recent attacks: model inconsistency attack [60] and attacks using customized convolutional kernels [71], which can occur even when the model updates have been securely aggregated before reaching the servers.
- *Robustness against malicious adversary*: We study an additional feature named robustness, or in some paper [42] mentioned as *"guarantee output delivery (GOD)"*. Given a minority of malicious servers (less than half of the total number of servers), our protocol allows honest servers to jointly compute the correct output without interference from the malicious servers. This extends the security and robustness of the two-server model (e.g., Elsa [63]), and the multi-server model (e.g., Prio [26],Prio+ [5]).

Privacy	t	# mal. servers	# mal. clients
Client's Input	≥ 1	$\leq n-1$	$m-2$
w. Robustness	$> n/2$	$< n/2$	$m-2$

Table 2: **Threat Model Condition and Privacy Guarantee of Our Mario**. mal. shorts for malicious. We assure privacy of input in dishonest majorty malicious servers, and assure privacy with robustness in honest majority set of servers, and under up to 2 honest uncorrupted clients. m: number of clients, n: number of servers.

To ensure robustness in the multiparty computation (MPC), it is generally assumed that less than half of servers can be malicious [41, 42, 49]. Indeed, if more than half of the servers are malicious, they can collude and override the semi-honest servers, compromising the correctness of the output and making robustness unattainable.

Our Contributions. They are twofold:

- We propose Mario, the first multi-server secure aggregation protocol that provides "privacy" against dishonest majority of servers and clients, provides "privacy with robustness" against honest majority set of servers and any set of corrupted clients, provide input validation against malicious clients. Our protocol achieves $3-9\times$ better performance than single-server SyncFL ACORN [11], $450-600\times$ faster runtime than the state-of-the-art single-server AsyncFL BASecAgg [67], while running $3.40\times$ faster than Elsa [63], and $283.4\times$ faster than Prio+.

- We realize a practical threshold additive homomorphic encryption (ThHE) scheme that simultaneously ensures malicious privacy and compactness in a dynamic system where any party can join or drop out of the computation midway. This work designs a novel efficient, compact, and robust ThHE scheme that does not require any trusted setup.

Theoretical Comparison. Table 1 outlines the security levels and various features supported by existing protocols closely related to our work. Note that while ACORN and Prio+ offer robustness, they do so only in a semi-honest server setting, which does not meet the requirement for "robustness with malicious servers."

The key advantages of our protocol are: (1) Client complexity independent of number of clients; (2) Security against malicious adversaries; (3) Defense against invalid input from malicious clients that poison the aggregated result (input validation); (4) Support for any number of clients dropouts without affecting correctness/security; (5) Support privacy with correctness. These features make Mario a comprehensive solution for real-world applications.

Overview of Our Techniques. We make use of n servers which are relatively stable, and a set of m clients join the training for every round. Assume we have an efficient and compact threshold additive homomorphic encryption (ThHE) scheme. The n servers first jointly compute the public key and distribute the secret key using $\mathsf{ThHE.KeyGen}$. Then, they send the public key pk to every client join training in the secure aggregation computation.

We begin with a basic secure aggregation protocol in the semi-honest setting. The process works as follows: each client encrypts their local input model v using a public key, resulting in $\mathsf{ThHE.Encrypt}(\mathsf{pk}, v)$, and sends the encrypted model to the leader server. The server then leverages the homomorphic properties of the encryption scheme to compute the encrypted sum. Afterward, a subset of t servers collaboratively decrypts the sum and forwards the final result to the clients, who then use it to update their local models.

To provide robustness against malicious clients, we leverage existed zero knowledge proof technique in ACORN [11] that provides efficient range proof for client's input. We adapt the protocol into our system with the optimization using the structure of our protocol: as the clients use public key encryption to encrypt their model, they do not need to communicate with any other clients for proving the range of their input, thus neglecting the additional $O(\log^2(m))$ communication and computation cost for each client of ACORN-robust.

For malicious servers, we ensure the privacy of an honest client's input even in the presence of up to $t-1$ malicious servers. This is achieved through the use of a threshold additive homomorphic encryption (ThHE) scheme, where only a group of t or more servers can decrypt a given ciphertext, while fewer than t servers cannot, thereby keeping the client's input secure under encryption. However, relying solely on existing ThHE schemes is insufficient to guarantee complete security in malicious settings. The challenges are twofold: (1) the adversary may perform a chosen ciphertext attack—specifically, the aggregating server could replace the encrypted sum with any ciphertext, such as a specific client input v_i, and requrest the t decryption servers into decrypting it; (2) the distributed key generation is not robust against malicious adversaries.

To address (1), we implement a check on the correctness of the ciphertext form, ensuring that the decryption servers only decrypt the correct ciphertexts that are indeed the summation of clients' encrypted messages. To address (2), we rely on a verifiable secret sharing scheme and propose an optimization where verification can be performed in batches, given that the secret share of the key is a long vector. Details of these defenses can be found in Section 3.2.

In our FL scheme, we also ensure that the final result remains correct (robustness property). We implement a consistency check based on the assumption of honest majority set of servers. Specifically, if the majority of server outputs are consistent and/or proven to be correctly evaluated, the set of clients will accept the final result as valid, allowing the training process to continue.

As shown in Table 2, Mario provides input privacy to honest

		Synchronous						Asynchronous	
		SecAgg+ [12]	ACORN [11]	Flamingo [53]	Prio+ [5]	Elsa [63]	**Mario**	BASecAgg [67]	**Mario**
Client	Comm	$L+\log m$	$L\log m$	$L\log m$	Ln	Ln	$L+n$	$Lm/(k-t)$	$L+n$
	Comp	$L\log m+\log^2 m$	$L(\log m+\log L)$	$L\log m$	Ln	Ln	$L\log L$	$Lm\log m/(k-t)$	$L\log L$
Server	Comm	$Lm+m\log m$	$Lm\log m$	$m(L+\log m)$	Lmn	Lmn	Lmn	Lk	Lkn
	Comp	$Lm\log m+m\log^2 m$	$Lm\log m$	$Lm\log m$	Lm	Lm	Lm	$Lk\log k$	Lk

Table 3: **Complexity Comparison of FL protocols.** Only representative works are shown. The description of the parameters presented in Table 4. The threshold in Mario is for the ThHE (the number of non-colluding servers).

Table 4: **Definition of Parameters Used in This Paper**

Parameter	**Description**
n	number of servers
m	number of clients
k	buffer size (async. setting)
L	model size
t	threshold for servers
τ	threshold for clients
nb	# neighbors for each client
N	Degree of plaintext & ciphertext polynomial
q	ciphertext modulo
p	plaintext modulo
κ	computational security parameter
λ	statistical security parameter
$[a]_q$	unique integer in $\mathbb{Z}_q$ with $[a]_q = a \mod q$
$[x]$	a set $\{1,\dots,x\}$
$(\mathbf{x};\mathbf{y})$	concatenation of two vector x and y

client against the presence of up to $t-1$ malicious servers, and provides correct aggregation to set of honest servers and clients against up to less than half of the malicious servers.

The remaining question is how to construct the efficient ThHE protocol in the dynamic setting. We build on the underlying framework presented in the work of [55, 56, 59], where servers utilize Shamir secret sharing to distribute their local secret values among each other. These shares are then summed to obtain the final "secret key share". However, we extend this approach to support a dynamic setting and improve both communication and computation costs. At a simple idea, our approach introduces t pivot parties that communicate with other parties and secret share their inputs. This allows us to implement mechanisms for new servers to join the training process through ThHE.Join functionality. Specifically, new servers can compute their share key using Lagrange interpolation when t existing servers send a portion of the computation data. Additionally, we provide ThHE.ShareRefresh, which enables all servers to collectively refresh the existing shares into a new set of shares—these are secret shares of the same secret but under updated (t',n) parameters.

Furthermore, we extend the work of [55, 56, 59] to achieve an efficient, parallelizable, and verifiable key generation process, which we present in Appendix D.

2 Preliminary and Related Work

2.1 Secure Aggregation in Federated Learning

Federated Learning with Single Aggregator. Masking with One-Time Pads (SecAgg [64]) is a popular technique used in SA for SyncFL. It allows clients to mask the updates before sending to the server. The masks are used to protect their models and will be cancelled out when the server computes the final aggregation. Flamingo [53] and ACORN [11] represent the state-of-the-art in SyncFL. Nonetheless, they exhibit limitations such as the heavy communication between clients, discussed in the introduction section.

For AsyncFL, [57] proposes a novel buffered asynchronous aggregation method, which allows the users to send their updates asynchronously while ensuring privacy by storing the updates in a trusted execution environment. The BASecAgg construction [67] removes the need for hardware support by carefully designing the mask technique of SecAgg [64] such that the masks cancel out even if they correspond to different rounds. However, they only consider the threat model of semi-honest server and semi-honest clients, and their client's communication and computation complexity depends on k, the number of clients in buffer .

Federated Learning with Multiple Aggregators. Within this model, existing methodologies encompass tailored systems for FL [5, 10, 16, 26, 31, 39, 40, 50, 63], alongside systems designed for privacy-preserving aggregation of statistical data, such as Prio[26] and Prio+ [5]. Although this approach effectively leverages lightweight federated learning, the existing protocols either ensure security under semi-honest conditions or are applicable solely to two-party scenarios (Elsa [63]).

Federated Learning with Aid of Decryption Committee. Between Single Aggregator and Multiple Aggregators in Federated Learning (FL), there are protocols that use committees to assist with decryption. The involvement of these committees helps reduce the communication overhead for general clients, which are often resource-constrained and may have unstable connections. There are three recent works that follow such settings: Flamingo [53], Willow [13], and OPA [47]. Although the setup are promising, they either do not cover case of malicious committees, or have limitation on fixed set of clients, and they only provide privacy with abort in case of malicious adversaries, without considering correctness of final result.

Secure Aggregation from HE. Indeed, ThHE has been used to compute SA [23, 65, 72], and one of the most representative is the threshold Paillier-HE which is expensive for a large n. Since our ThHE is based on Polynomial-LWE (PLWE) and its encryption/evaluation has mostly the same cost as the traditional PLWE single-key HE. Therefore, our SA (as well as our FL scheme) is more efficient than previous work.

Recent Attacks on SA. Regardless of implementing the aggregation securely in the standard security definition [46], there are many attacks to FL. For example, the malicious user can perform poisoning/Byzantine attacks (inject poisoned updates into the learner) to reduce the global model accuracy or implant backdoors [14, 15]; and/or the malicious server can provide different global model updates to different users to infer information on users' datasets (so-called model inconsistency attack which was recently discovered by [60, 71]). Also, malicious server can manipulate the training model architecture, injecting a carefully designed kernel to steal client's input [71]. This work aims to prevent all of such attacks.

2.2 Threshold Homomorphic Encryption

Homomorphic encryption (HE) is a form of encryption that allows performing arbitrary computations on encrypted data without access to the secret key. Threshold Homomorphic Encryption (ThHE) aims to protect an HE secret key by distributing it into n shares, each stored by a different party. Any t parties can use the secret without reconstructing it whereas any $t-1$ parties should not be able to recover or use the secret.

ThHE scheme of [17] does not provide compactness: the evaluated ciphertext size is linear in number of input to homomorphic evaluation (e.g. n). Recent works [9, 18, 24, 27, 28] enhance the asymptotic complexity of the ciphertext size of [17], yet it remains linear in the number of parties. Within the FL framework, this size is directly proportional to the number of clients, which poses efficiency challenges. Our work builds upon the approach of [55, 56, 59], where Shamir secret sharing is used to keep the secret key compact with new functionalities (i.e., ThHE.Join, ThHE.ShareRefresh) that fit with FL setup.

Distributed Key Generation (DKG) [2, 3, 4, 7, 8, 61, 70] allows a group of n parties to jointly compute a shared pair of secret key. However, these protocols either have to assume that the dealer are trusted or have to protect against the malicious dealer with an expensive cost. In the context of FL, the fact that a dealer knows the secret key would directly lead to leak in user's weight when the dealer collude with our computing servers. Even more crucially, current DKG protocols lack support for computing the public key in the unique format required by ThHE.

3 Protocol Building Blocks

3.1 Our Dynamic Threshold HE

In the context of FL, we consider the ThHE as a tuple of algorithms (ThHE.SecretKeyGen,ThHE.PublKeyComp, ThHE.Join, ThHE.Encrypt,ThHE.PartDec, ThHE.FinalDec, ThHE.Add, ThHE.ShareRefresh) where their ideal functionalities are described in Definition 1.

Definition 1. *A dynamic threshold additive homomorphic encryption ThHE scheme is a tuple of PPT algorithms ThHE =(ThHE.SecretKeyGen, ThHE.PublKeyComp, ThHE.Join, ThHE.Encrypt, ThHE.PartDec, ThHE.FinalDec, ThHE.Add, ThHE.ShareRefresh) with the following properties:*

- *ThHE.SecretKeyGen$(\lambda, n) \to (\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$: Given the security parameters λ and the number of parties n, output to each party P_i the secret share sk_i, such that sk_i is the Shamir share at ID i of a common secret key sk.*
- *ThHE.PublKeyComp$(\mathsf{pk}_1, \ldots, \mathsf{pk}_n) \to \mathsf{pk}$: Given the local public keys pk_i from n parties, output the global public key pk such that $(\mathsf{sk}, \mathsf{pk})$ forms a valid pair of secret key and public key, achieving the security parameter λ.*
- *ThHE.Join$(\nu) \to \mathsf{sk}_\nu$: When a new party with ID ν requests to join, output the corresponding secret share of sk to party P_ν, i.e., output sk_ν.*
- *ThHE.ShareRefresh$(\mathcal{P} = \{P_1, \ldots, P_n\}, t') \to (\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$: Given the input new threshold t' and a set of n parties $\mathcal{P}$, output the new threshold shares of the secret key and distribute these shares to each corresponding party.*
- *ThHE.Encrypt$(\mathsf{pk}, m) \to C$: Given the message m and public key pk, output the corresponding ciphertext $C = (C[1], C[0])$ that encrypts m.*
- *ThHE.PartDec(C, sk_i): Given input C and sk_i, the party P_i performs local partial decryption to compute $C' = C[1] + C[0] \cdot \mathsf{sk}_i$.*
- *ThHE.FinalDec$(p_1, \ldots, p_t) \to m$: On t partial decryptions, output the decrypted message m*
- *ThHE.Add$(C_1, \ldots, C_k) \to C_{add}$: Given k ciphertexts as input, output the ciphertext that encrypts the sum of the plaintexts of $C_{i \in [1,k]}$.*

Protocol Overview. This section presents an overview of our ThHE protocol. The complete protocol description can be found in Appendix D, along with a detailed security analysis in Appendix E.

We build on the approach of [55, 56, 59], but introduce t pivot parties, $P_1, \ldots, P_t$. Each pivot samples a small secret key s_i and uses t-out-of-n Shamir secret sharing to distribute s_i to all other servers. After receiving all the Shamir shares, each server sums them to obtain a "local secret key" sk_i.

The common public key pk is computed as follows: a leader server generates a polynomial a as the second part of the public key and sends a to the other servers. If all t pivot servers are online, the public key is computed by aggregating the partial public keys $\mathsf{pk}_i = [-a \cdot c_{i,0} + e_i]_q$ sent by each pivot P_i, resulting in $\mathsf{pk} = \left([\sum_{i=1}^{t} \mathsf{pk}_i]_q, a\right)$. Here, $c_{i,0}$ represents an additional component of the local secret key sk_i (i.e., the additive share of the secret key sk), and e_i is a small noise polynomial sampled by the i-th pivot. If not all pivot servers are available, a set A of t random servers is chosen, and the partial public keys take the form $\mathsf{pk}_i = \left(\frac{-i}{j-i} \cdot a \cdot \mathsf{sk}_i + e_i\right)$. Thus, we can use Lagrange interpolation to compute the public key, leveraging the fact that $a \cdot \mathsf{sk}_i$ represents the Shamir share of the public key at server P_i. Encryption, partial decryption, and decryption follow the protocol detailed in [55].

We further customize the threshold HE framework in [55] by introducing three new functionalities: (1) verifiable key generation, (2) parallelizable key generation, and (3) dynamic joining for new servers during protocol execution.

Verifiable Key Generation. Firstly, to address the presence of malicious actors, we need a method to verify the correctness of key generation. We achieve this by introducing the function BatchSumVerify, as defined in Figure 2. This function is utilized for both key verification and ciphertext verification in the subsequent sections.

Parallelizable Key Generation. Secondly, we implement parallelizable key generation. Since each Shamir share is a function evaluation of a local polynomial at the server ID, we can optimize the process by evaluating the polynomial in batch. Specifically, we preprocess a Vandermonde matrix of server indices and then create another matrix from the Shamir polynomial coefficients. The Shamir shares for all parties are obtained through matrix multiplication. By leveraging this approach, we can efficiently perform key generation using GPUs for matrix multiplication. The algorithm is illustrated in Figure 13, with further details of this optimization provided in Appendix D.4.

Dynamic Property. Finally, we introduce two new functions: ThHE.Join and ThHE.ShareRefresh, to facilitate the inclusion of new servers during the training process. When a new party P_ν requests to join (ThHE.Join), a random committee A of t servers is selected to compute the Shamir share of sk for P_ν using Lagrange interpolation. Specifically, each party $P_{k \in A}$ computes $u_k = \mathsf{sk}_k \cdot L_A(\nu, k)$, where $L_A(\nu, k) = \prod_{j \in A \setminus k} \frac{\nu - k}{j - k}$ is the Lagrange coefficient and sk_k is the secret-shared key of P_k. The secret-shared key sk_ν is then computed as $[\sum_{i \in A} v_i]_q$. However, directly sending u_k to P_ν could leak sk_k to them. To prevent this leakage, we use zero-sharing. Specifically, the servers in A perform zero-sharing [64] to obtain a zero share, which they then add to u_k before sending it to P_ν. The zero share is cancelled at the end. This method is similar to secure aggregation on $L_A(\nu, k) \cdot \mathsf{sk}_k$, but without dropout handling, assuming a stable connection among the servers.

While ThHE.Join allows new servers to join the system, it introduces potential security concerns for Mario, as the number of malicious servers might exceed the pre-defined threshold t. To mitigate this, we introduce the ThHE.ShareRefresh feature, which allows the same secret key to be reshared (keeping the public key unchanged) while updating the threshold t to a new threshold t'. At a high level, the idea involves adding the existing t-out-of-n Shamir share sk_i of the secret key sk with a new t'-out-of-n Shamir share sk_i^0 of zero. This approach works because (1) sk_i can be treated as a t'-out-of-n Shamir share where the highest $t' - t$ coefficients of the Shamir polynomial are all zeros, and (2) Shamir shares have an additive property. The remaining question is how to generate sk_i^0. This can be done by using our ThHE.SecretKeyGen protocol, where the secret key is set to zero.

3.2 Our Variant of Batch Sum's Verification

Given a set of input vectors $\{\vec{x}_i \mid 1 \le i \le \eta\}$, where each party $P_{i<t}$ holds a vector $\vec{x}_i$, and the last party P_t holds the remaining $\eta - t$ vectors $\vec{x}_{j \in [t,\eta]}$ along with a vector $\vec{x}$ that potentially represents the sum of all η vectors, the goal is for the party P_ν to verify whether $\vec{x} = \sum_{i=1}^{\eta} \vec{x}_i$ in a single batch operation. We define this functionality as BatchSumVerify, and present its protocol in Figure 2.

In our FL scheme, we utilize this building block to verify the integrity of ciphertexts. Specifically, we set η as the total number of valid client messages, with $\vec{x}_i$ representing ciphertext ct_i. If $\eta < t$ (there are fewer than t valid client ciphertexts), we set $\vec{x}_{j \in [\eta, t]}$ to zero to ensure it does not affect the final verification result. Then, our BatchSumVerify function allows a specific verifier server P_v to confirm that the ciphertext (the sum/aggregation) it is about to decrypt is correct.

Additionally, we use this building block to prevent malicious behavior during the public key computation in our dynamic ThHE. In this process, each server P_i generates a local public key pk_i, and the final public key is computed as the aggregate of these local public keys. Therefore, when we set $\eta = t$, with $\vec{x}_i$ representing the local public key pk_i held by server P_i and $\vec{x}$ representing the final public key pk, the function BatchSumVerify helps the verifier P_v confirm that the output of ThHE.PublKeyComp is correct.

Protocol. Our batch verification of the sum (BatchSumVerify) proceeds as follows: The t parties $P_1, \ldots, P_t$ form a committee of provers. Each party P_i has an input vector $\vec{x}_i$ of length ℓ, and P_t has additional input vectors $\vec{x}_{j \in [t+1,\eta]}$, which helps the verifier P_v verify that their input vector $\vec{x}$ is the sum of all the parties' inputs, without revealing information about $\vec{x}_{i \in [\eta]}$.

First, P_v generates a challenge vector $\vec{\alpha} \in (\mathbb{Z}_q \setminus 0)^\ell$ and sends it to all $P_{i \in [t]}$. Each party $P_{i \in [t-1]}$, upon receiving the challenge, computes $[\![v]\!]_i = \langle \vec{\alpha}, \vec{x}_i \rangle$, compressing their input vector of length ℓ into a single value (Step 3). P_t is special because it holds multiple vectors; therefore, it compresses its input vectors by computing $[\![v]\!]_t = \langle \vec{\alpha}, \sum_{j \in [t,\eta]} \vec{x}_j \rangle$ as shown in Step 4. Each party then sends $g^{[\![v]\!]_i}$ to P_v, who checks if $g^{\langle \vec{\alpha}, \vec{x} \rangle} = \prod_{i=1}^{t} g^{[\![v]\!]_i}$. If the check fails, P_v declares that the vec-

PARAMETERS: The space $R_q = \mathbb{Z}_q[X]/(X^N+1)$, a generator g of $\mathbb{Z}_q$. A party P_v acts as the verifier, $\eta \geq t$ total number of vectors.
INPUT:

- a set of t parties $\mathcal{S} = \{P_1, \ldots, P_t\}$
- Party $P_{i<t}$ has $\vec{x}_i$;
- Party P_t has $\vec{x}_{j \in [t,\eta]}$;
- P_v has $\vec{x}$;

OUTPUT: P_v does "accept" if the $\vec{x} = \sum_{i=1}^{\eta} \vec{x}_i$, else "reject".

```
1: procedure BatchSumVerify($\vec{x}, \vec{x}_1, \ldots, \vec{x}_\eta, \mathcal{S}$)
2:     The party $P_v$ generates $\ell$ values $\vec{\alpha}_{i\in[\ell]} \in \mathbb{Z}_q \backslash \{0\}$ as challenges and sends them to everyone.
3:     Each party $P_{i<t}$ locally computes $[\![v]\!]_i = \langle \vec{\alpha}, \vec{x}_i \rangle$
4:     Party $P_t$ locally computes $[\![v]\!]_t = \langle \vec{\alpha}, \sum_{j=t}^{\eta} \vec{x}_j \rangle$
5:     $P_i$ sends $g^{[\![v]\!]_i}$ to $P_v$.
6:     $P_v$ computes $C_v = \prod_{i=1}^{t} g^{[\![v]\!]_i}$
7:     $P_v$ checks if $C_v = g^{\langle \vec{\alpha}, \vec{x} \rangle}$, return "accept", else return "reject"
8: end procedure
```

Figure 2: Our Batch Sum's Verification (BatchSumVerify).

tor $\vec{x}$ is not the sum of the vectors $\vec{x}_i$

Our protocol BatchSumVerify is designed to ensure that it always outputs "accept" when the sum $\vec{x} = \sum_{i\in[m]} \vec{x}_i$ is correct. If the sum is incorrect, the protocol outputs "accept" only with a negligible probability, specifically $\frac{1}{q-1}$. In our implementation, we set q of 54-bit. This negligible probability arises because the challenge α is sampled randomly at each iteration of BatchSumVerify by the verifier P_v. Furthermore, during the execution of BatchSumVerify, there is no gain of knowledge about the values of x_i for parties $j \neq i$. The sketched security proof for our protocol BatchSumVerify is presented in Appendix C.

3.3 Zero-knowledge Argument of Knowledge

Informally, a zero-knowledge argument of knowledge is a set of PPT algorithms $(\mathcal{P}, \mathcal{V})$ that enables two parties, the prover P and the verifier V, to exchange information such that after the evaluation, V can verify whether P possesses certain knowledge without revealing that knowledge. To prevent malicious adversaries, we rely heavily on techniques from Bulletproofs [22] for the correctness of dot product evaluations, the input validation technique from ACORN [11], and the correctness of ring operation from Pino et al. [29].

At a high level in Bulletproofs, to prove the knowledge of vectors $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_q^L$ that satisfy $\langle \mathbf{x}, \mathbf{y} \rangle = a$ for some public value a, the prover recursively compresses $\mathbf{x}$ and $\mathbf{y}$ into smaller vectors $\mathbf{x'}, \mathbf{y'}$ of half the size, which satisfy $\langle \mathbf{x'}, \mathbf{y'} \rangle = a'$ for a public value a'. The communication overhead for this process is $2\lceil \log(L) \rceil$ group elements and two $\mathbb{Z}_q$ elements.

In Pino et al. [29], the goal is to verify the correctness of ring operations, specifically proving that $[\sum_{i=1}^{k} U_i T_i + V]_q = Z$ in the ring $R_q = \mathbb{Z}[X]/(X^N+1)$. Here, k is a constant, and $U_{i\in[k]}$, $T_{i\in[k]}$, V, and Z are polynomials in R_q. Both the verifier and prover know $U_{i\in[k]}$ and Z.

The prover begins by extending the original equation to $\sum U_{i\in[k]} T_{i\in[k]} + V + D_1 q + D_2(X^N+1) = Z$, where D_1 is a polynomial of degree at most $2(N-1)$, and D_2 is a polynomial of degree at most $N-1$. Upon receiving a challenge $\alpha \in \mathbb{Z}_q$ from the verifier, the prover must show that $\sum_{i=1}^{k} U_i(\alpha) T_i(\alpha) + V(\alpha) + D_1(\alpha) q + D_2(\alpha)(\alpha^N+1) = Z(\alpha)$. This can be rewritten as $\big(U_1(\alpha), \ldots, U_k(\alpha); 1; q; \alpha^N+1\big) \cdot \big(T_1, \ldots, T_k(\alpha); V; D_1; D_2\big) \cdot \big(1, \alpha, \ldots, \alpha^{N-1}\big) = Z(\alpha)$.

The prover can then use Bulletproofs to prove the correctness of this dot product. Since the vectors $(U_1(\alpha), \ldots, U_k(\alpha); 1; q; \alpha^N + 1)$, $(1, \alpha, \ldots, \alpha^{N-1})$, and $Z(\alpha)$ are known to the verifier, while only $(T_1(\alpha), \ldots, T_k(\alpha); V(\alpha); D_1(\alpha); D_2(\alpha))$ remain secret, Pino et al. can leverage the techniques from Gentry et al. [38] to efficiently complete the proof of knowledge.

Bulletproofs also provide a zero-knowledge proof of knowledge for range proofs, where the prover proves knowledge of a variable x within a specific range $[a,b]$. However, this method is less efficient when x is a long vector rather than a single value. For this reason, we employ the technique from ACORN [11] for such cases. We review the technique used in ACORN in Section 5.2.

4 Semi-Honest Secure Aggregation

Given our building blocks, we design our secure aggregation protocol (Mario) for a multi-server setting. This basic version operates under the semi-honest threat model. In subsequent sections, we introduce various defenses to adapt the protocol for use in a malicious threat model.

We outline the blueprint of our construction in Figure 3. In Step 1, the leader server P_1 collects encrypted models from the clients and forwards them to a set O of t online servers. Each online server then performs homomorphic addition on the ciphertexts individually (Steps 3-4) to obtain the ciphertext of the sum. Subsequently, each server $P_i \in O$ requests any t servers to jointly decrypt the ciphertext (Step 4). Once the sum is decrypted, P_i sends the result to every client, allowing them to update their local models.

Remark. In our semi-honest secure aggregation protocol, the leader P_1 is responsible for performing all tasks in Steps 3 to 6, making Step 2 redundant. All other servers in O, with the exception of those involved in decrypting the message for P_1, remain inactive and do not perform any additional tasks.

5 Achieving Malicious Secure Aggregation

If the servers and/or clients are malicious, the protocol depicted in Figure 3 is vulnerable to various attacks. Below, we present all possible ways in which a group of malicious

PARAMETERS:

- a set of m clients $\mathcal{U} = \{U_1, \ldots, U_m\}$
- a set of n servers $\mathcal{P} = \{P_1, \ldots, P_n\}$
- A dynamic THE scheme ThHE with malicious key generation.
- A hash function $H : \{0,1\}^\star \to \{0,1\}^\star$

INPUT: The client $U_j \in \mathcal{U}$ has input v_j. The server P_i has no input

ONE-TIME SETUP: The n servers together run ThHE.KeyGen which consist of (ThHE.SecretKeyGen,ThHE.PublKeyComp) to agree on a public key pk and secretly shared a common secret key sk.

INITIALIZATION: The first server, P_1, initializes a model M that the servers aim to train and sends this model to all clients in $\mathcal{U}$.

EVERY SA ROUND:

1. Client U_j in $\mathcal{U}$ sends $\mathsf{ct}_j = \mathsf{ThHE.Encrypt}(\mathsf{pk}, v_j)$ to the leader server P_1
2. Leader server P_1 forwards the ciphertext ct_j to a set O of t online servers
3. For SyncFL, the server $P_i \in O$ performs an additive evaluation on all encrypted $\mathsf{ct}_j^{(i)}$ as $\mathsf{ct}^{(i)} = \mathsf{ThHE.Add}(\mathsf{ct}_1^{(i)}, \ldots, \mathsf{ct}_m^{(i)})$.
4. For AsyncFL, server $P_i \in O$ stores the received $\mathsf{ct}_j^{(i)}$ in the buffer B. When B is full, P_i computes $\mathsf{ct}^{(i)} = \mathsf{ThHE.Add}(\{\mathsf{ct} \in B\})$.
5. $P_i \in O$ chooses a set A_i of $t-1$ online servers, and sends $\mathsf{ct}^{(i)}$ to $P_\tau \in A_i$. If $A_i \cup P_i = O$, the $P_\tau \in A_i$ uses their $\mathsf{ct}^{(\tau)}$, obtained from Step (3-4), as $\mathsf{ct}^{(i)}$ for the below computation.
   - Each server $P_{\tau \in A_i}$ executes a partial decryption as $\mathsf{p}_\tau = \mathsf{ThHE.PartDec}(\mathsf{ct}^{(i)}, \mathsf{sk}_\tau)$ and sends it to P_i
   - Server P_i performs a final decryption as $v = \mathsf{ThHE.FinalDec}(p_i, \{\mathsf{p}_\tau\}_{\tau \in A_i})$, where $\mathsf{p}_i = \mathsf{ThHE.PartDec}(\mathsf{ct}^{(i)}, \mathsf{sk}_i)$
6. Server $P_i \in O$ sends v to all clients in $\mathcal{U}$.

Figure 3: **Our Blueprint Multi-server Secure Aggregation**. Note: For the semi-honest protocol, a single leader server P_1 can perform all the tasks assigned to servers $P_i \in O$ in Steps 3-6, and Step 2 is redundant.

servers and clients can either disrupt the final result or extract information from semi-honest clients.

(A) At one-time setup, malicious servers can disrupt the protocol by sending incorrect shares or computing a wrong public key during the key generation phase, potentially causing the entire protocol to fail.

(B) At model initialization, a malicious server can assign specific weights, as proposed in Loki [71], to attempt to infer the clients' inputs.

(C) At Step 1, malicious clients can send an encryption of an incorrect message (Encryption Verification) as well as send the message with wrong form (Input Validation), interfering the training procedure.

(D) At Step 2, a malicious leader server may forward incorrect encryptions received from clients to the other servers.

(E) At Steps 3-4, a malicious server could conduct a chosen ciphertext attack by reporting an incorrect computation of the sum or substituting the result with a completely different ciphertext.

(F) At Step 5, a malicious server involved in decryption might send an incorrect result during partial decryption.

(G) At Step 6, a malicious server can send incorrect updates to the clients in $\mathcal{U}$.

Section 5.1 addresses **(B)**, **(D)**, and **(E)** to ensure input privacy against up to $t-1$ malicious servers and any set of $m-2$ corrupted clients. Subsequently, Section 5.2 tackles **(A)**, **(C)**, **(F)**, and **(G)** to achieve robustness against fewer than $n/2$ malicious servers and $m-2$ corrupted clients. Figure 4 presents our Mario construction in the malicious and robustness setting.

5.1 Input Privacy

Defense in Model Initialization. To defend against **(B)**, we propose additional steps for the servers to jointly initialize the model architecture and model weight. A naive approach is to have all the servers in $\mathcal{P}$ send the initialized model to the clients. However, it incurs communication overhead, both for clients and servers. Therefore, a simple trick is to have the leader server send the model to the clients, and have each server send the hash of the initialized model. The client would then check for inconsistency between the model he received and the hash.

To ensure input privacy as well as defend against model modification attacks like Loki [71], each client in $\mathcal{U}$ verifies that the model M matches at least $t-1$ of the received hashes. If this condition is not met, the client must abort (Step 5, Initialization, Figure 4). For robustness, each client in $\mathcal{U}$ identifies the hash that matches more than $n/2$ of the received hashes and requests the corresponding model M from one of the servers. If no matching hash is found, the client must abort (Step 6).

Verification of Forwarded Ciphertext from Leader. To address the issue of verifying the ciphertext $\mathsf{ct}_j = \mathsf{ThHE.Encrypt}(\mathsf{pk}, v_j)$ being sent from the leader server to a set O of t online servers—issue **(D)**—a naive approach would involve having the client U_j send the encrypted model ct_j to all $P_i \in O$. However, this would cause a significant communication on the client's side. Instead, we use a simple trick as before – using a hashing technique to mitigate this cost. The client additionally sends the hash of the encrypted model as $h'_j = H(\mathsf{ct}_j)$ to each server other than the leader, which incurs only a small additional computational and communication cost compared to the naive approach. This process is described in Step 1, Secure Aggregation, as shown in Figure 4.

Verifiable Ciphertext to Decrypt. To address issue **(E)**, where a server might send an incorrect ciphertext ct to the decryption servers to learn more information, we need a verification mechanism. For instance, if server P_i sends the ciphertext of the first client, ct_1, instead of the correct sum $\mathsf{ct} = \sum \mathsf{ct}_j$, a malicious server could potentially learn the input of this specific client.

To counter this, we enable the decryption servers to verify that the ciphertext ct they receive from P_i is indeed the sum of the ciphertexts ct_j from each client. This can be indeed achieved by using our building block described in Section 3.2—the batch verification of sums ($\mathsf{BatchSumVerify}$), i.e., verifying $\mathsf{ct} = \sum ct_j$.

5.2 Achieving Privacy with Robustness

Verifiable Key Generation for Threshold HE. In the context of issue **(A)**, where malicious servers might distribute incorrect threshold shares during Key Generation, we can mitigate this risk by employing a technique from Feldman's scheme for verifiable secret sharing [35]. Feldman's scheme provides a method to verify the correctness of the shares distributed among servers. By incorporating this technique, we ensure that each server receives and validates threshold shares accurately. We provide details on our construction of the verifiable secret key sharing in Appendix D.3.

In addition, during the computation of the public key, there is a risk that malicious servers involved in summing the local public keys might produce an incorrect result. To address this issue, we employ our building block – the batch verification of the sum, $\mathsf{BatchSumVerify}$, as illustrated in Figure 2.

Client Input Validation and Encryption Verification (π_{valid}, $\pi_{encverif}$). To address issue **(C)**, we implement two defense mechanisms: input validation and encryption verification.

Input validation. Our protocol is compatible with existing techniques proposed in ACORN [11]. Additionally, the multi-server model in our protocol allows for further optimization of the ZKP approach used in ACORN.

To prove that each entry $\mathbf{x}_j$ in a long vector $\mathbf{x}$ lies within the range $[0, b]$, ACORN [11] shows that it is equivalent for the prover (i.e., the client) to show knowledge of three vectors $\mathbf{u}, \mathbf{v}, \mathbf{w}$ such that:

$$(\mathbf{x}'; \mathbf{u}; \mathbf{v}; \mathbf{w}) \cdot (\mathbf{x}'; \mathbf{u}; \mathbf{v}; \mathbf{w}) = a \tag{1}$$

where $a = -1-(b+1)^2$ and $\mathbf{x}' = 2\mathbf{x} - (b-1) \cdot \mathbf{1}$. The proof of Equation (1) can be achieved using Bulletproofs, which incurs a computational cost of $O(L \log L)$ and a communication cost of $O(\log L)$ on the client. Similar to ACORN, we convert the range proof into a dot product proof. However, our model improves upon ACORN-robust (ACORN with robustness against semi-honest servers) by eliminating the need for additional $O(\log^2(m))$ computation and communication with other clients for the proof of correct secret sharing.

Encryption Verification. This requires the client to prove that the ciphertext sent is indeed the encryption of the message used in input validation, we first present the encryption formula in BFV encryption used to realize our ThHE (a more detailed explanation can be found in Figure 9). The encryption of a message m is given by:

$$\mathsf{ThHE.Encrypt}(\mathsf{pk}, m) = ([\mathsf{pk}[0]u + \mathbf{e}_1 + \Delta m]_q, [\mathsf{pk}[1]u + \mathbf{e}_2]_q) \tag{2}$$

where $u \leftarrow R_q$ is a random polynomial, and $\mathbf{e}_1, \mathbf{e}_2 \leftarrow \chi$ are small noise polynomials.

To prove the correctness of encryption, we require two proofs: (i) a proof to demonstrate the correct evaluation of the ring operation, and (ii) a proof to show the smallness of the noise polynomials $\mathbf{e}_1$ and $\mathbf{e}_2$.

For (i), we apply the technique from Pino et al. [29], as discussed in Section 3.3 . By doing so, we can convert the proof into a dot product proof and use Bulletproofs to verify the correctness of the operation.

For (ii), it is equivalent to range proofs specifically tailored for small vectors since the coefficients of $\mathbf{e}_i$ should be within the range $[-1, 1]$. We then leverage the ACORN technique to handle this efficiently. Specifically, we need a proof showing the correctness of Equation 1, but with $a = -1-(2+1)^2 = 10$ and $\mathbf{x} = \mathbf{e}_1 + \mathbf{1}, \mathbf{e}_2 + \mathbf{1}$. This adjustment is because the bound for $\mathbf{e}_i$ is [-1, 1], making the bound for $\mathbf{e}_i + \mathbf{1}$ [0, 2].

To sum up, to defend against **(C)**, we ask from client the proof π_{valid} showing the correct evaluation of Equation 1, the proof $\pi_{encverif}$ verifying the correctness of the encryption.

PARAMETERS:

- a set of m clients $\mathcal{U} = \{U_1, \dots, U_m\}$, a set of n servers $\mathcal{P} = \{P_1, \dots, P_n\}$
- A dynamic THE scheme ThHE with malicious key generation defined in Definition 1.
- A Variant of Batch Sum's Verification, BatchSumVerify, described in Figure 2.
- Proofs $\pi_{valid}, \pi_{encverif}$, and $\pi_{\tau,partial}$ described in Section 5.2.
- A hash function $H : \{0,1\}^\star \to \{0,1\}^\star$

INPUT: The client $U_j \in \mathcal{U}$ has input v_j. The server P_i has no input

ONE-TIME SETUP: The n servers together run ThHE.KeyGen which consists of (ThHE.SecretKeyGen,ThHE.PublKeyComp) to agree on a public key pk and secretly shared a common secret key sk.

INITIALIZATION:

1. The servers in $\mathcal{P}$ agree on a random seed for initialization.
2. Each server in $\mathcal{P}$ initializes the model M using the agreed seed.
3. Server P_1 sends the initialized model M to all clients in $\mathcal{U}$.
4. The other servers $P_{i\neq 1}$ in $\mathcal{P}$ send the hash $h_i = H(M)$ of the model to the clients in $\mathcal{U}$.
5. For input privacy, each client in $\mathcal{U}$ verifies that the model M matches at least $t-1$ of the received hashes. If not, abort.
6. For robustness, each client in $\mathcal{U}$ identifies the hash that matches more than $n/2$ of the received hashes and requests the corresponding model M from one of the servers. If no matching hash is found, abort.

SECURE AGGREGATION:

1. The client $U_{j\in[m]}$ distributes $\mathsf{ct}_j = \mathsf{ThHE.Encrypt}(\mathsf{pk}, v_j)$ to a set O of t online servers as follows:
   - U_j sends ct_j to the leader P_1, and sends the hash $h_j = H(\mathsf{ct}_j)$ to servers $P_i \in O \setminus \{P_1\}$.
   - The leader P_1 forwards ct_j to the $P_i \in O \setminus \{P_1\}$.
   - Each server $P_i \in O$ verifies that $H(\mathsf{ct}_j) = h_j$. If the hashes do not match, the server aborts.

   Clients also distribute $\pi_{valid}, \pi_{encverif}$ to the servers for input validation. The server $P_i \in O$ then do the following:

   - Verifies the proofs $\pi_{valid}, \pi_{encverif}$
   - For SyncFL, stores the "honest" client ID in set V if he accepts the proof. Then he computes $\mathsf{ct}^{(i)} = \mathsf{ThHE.Add}(\mathsf{ct}_{i\in V}))$
   - For AsyncFL, stores the "honest" client ID in the buffer B and set V if he accepts the proof. If B is full, he computes $\mathsf{ct}^{(i)} = \mathsf{ThHE.Add}(\mathsf{ct}_{i\in B}))$
2. $P_i \in O$ forwards $\mathsf{ct}^{(i)}$ and set V from Step 1 to all servers in $\mathcal{P}$
3. Upon receiving $\mathsf{ct}^{(i)}$ and V, $P_k \in \mathcal{P}$ run $\mathsf{BatchSumVerify}(\mathsf{ct}^{(i)}, \mathsf{ct}_{i\in V}, O)$ to verify he receives the correct ciphertext.
4. $P_i \in \mathcal{P}$ chooses a set A_i of t servers, sends the corresponding Lagrange coefficients to each server in A_i. P_i and the servers in A_i perform the following
   (a) Each server $P_{\tau\in A_i}$ executes a partial decryption as $\mathsf{p}_\tau = \mathsf{ThHE.PartDec}(\mathsf{ct}^{(i)}, \mathsf{sk}_\tau)$ and a short proof $\pi_{\tau,partial}$ and sends $(\mathsf{p}_\tau, \pi_{\tau,partial})$ to P_i
   (b) The server P_i verify $\pi_{\tau,partial}$ for every $\tau \in A_i$, record all servers $M \subset A_i$ that failed verification, **rerun** this step for a new set $A'_i = (A_i \setminus M) \cup S'$ where S' is a new set of servers, $|S'| = |M|$
   (c) P_i performs a final decryption as $v^{(i)} = \mathsf{ThHE.FinalDec}(p_i, \{\mathsf{p}_\tau\}_{\tau\in A})$, where $\mathsf{p}_\tau = \mathsf{ThHE.PartDec}(\mathsf{ct}^{(\tau)}, \mathsf{sk}_\tau)$
5. Client $U_j \in \mathcal{U}$ do the following step:
   (a) request $h_i = H(v^{(i)})$ from $P_i \in \mathcal{P}$, find the hash value h that is most frequent in the list $\{h_1, \dots, h_n\}$
   (b) record the set $\mathcal{P}_H = \{P_i \| h_i = h\}$
   (c) in sequence, ask for $v^{(i)}$ from $P_i \in \mathcal{P}_H$. Check if $H(v^{(i)}) = h$, stop if true, else request for the next server in $\mathcal{P}_H$

Figure 4: **Mario** – Privacy with Correctness Multiserver Malicious Secure Aggregation

Even though it might seem complicated for the client, our Mario is more efficient than ACORN-robust. In Mario, the client can directly send all the proofs to the servers, whereas in ACORN-robust, the client incurs additional $\log^2(m)$ computation and communication costs to prove and verify the correctness of secret sharing and aggregation.

Verifiable Partial Decryption ($\pi_{\tau,partial}$). In our protocol's decryption phase, the server responsible for partial decryption, P_τ, performs the following evaluation to compute the partial decryption:

$$p_\tau = \mathsf{ThHE.PartDec}(\mathsf{ct}^{(i)}, \mathsf{sk}_\tau) = [\mathsf{ct}^{(i)}[1]L_A(0,\tau)\mathsf{sk}_\tau + e_\tau]_q \tag{3}$$

where $\mathsf{ct}^{(i)}$ is the ciphertext to decrypt, $L_A(0,\tau) = \prod_{j\in A\setminus\tau} \frac{-\tau}{j-\tau}$ is the Lagrange coefficient, and e_τ is a small noise.

To verify the correctness of the evaluation, the server P_τ must prove two things: the ring operations in Equation (3) are correct; and the noise e_τ is within the allowable bounds. For the former, we use Pino et al.'s technique to prove the correct evaluation of ring operations. For the latter, we use a zero-knowledge range proof from ACORN [11] to ensure that the noise e_τ is within the allowable bounds. Both techniques are discussed in Section 3.3.

Thus, to address issue **(F)**—ensuring the correctness of partial decryption—we apply the proof techniques (so-called $\pi_{\tau,partial}$) discussed above. With these proofs, the server holding the encrypted sum can verify the correctness of all received partial decryptions. Additionally, the server can identify any malicious parties that submitted incorrect partial decryptions. If needed, the servers can redo the partial decryption round and replace the malicious servers with new ones.

Verifiable Final Result. To address **(G)**, we employ a hashing technique for consistency checks. Specifically, we have Step 2 for the selected servers in O to send the aggregated ciphertext to all servers in the set $\mathcal{P}$. Then, in Step 5, each server in $\mathcal{P}$ sends the hash of its result to the client. In a setting with an honest majority, the hash of the correct model will be provided by more than half of the servers.

Upon receiving these hashes, the client identifies the servers that provided the majority hash and requests the model from one of them. If a server returns a model that does not match the hash it provided, the client detects the discrepancy and requests the model from another server. On average, the client will need to make two requests to obtain the correct model from an honest server.

Putting Everything Together. By integrating all the aforementioned defenses, our Mario ensures both privacy and correctness within the malicious setting. The protocol outputs the correct summation. We formally state the robustness against malicious adversaries in the following lemma:

Lemma 1. *Given the multi-server secure aggregation protocol presented in Figure 4, and the threat model of an honest majority of servers and a dishonest majority of clients as stated in Table 2, the following two statements hold:*

1. *Malicious servers cannot convince the client of an incorrect aggregated result.*
2. *Malicious clients' inputs are ignored in the final summation and cannot interfere with the training process.*

Sketched Proof. For the first statement, a malicious server at Step 1 cannot distribute an incorrect encryption of the client's model due to the hashing consistency check. At Step 2-3, a malicious server cannot decrypt the wrong message because of $\mathsf{BatchSumVerify}$. Furthermore, the server cannot provide an incorrect partial decryption due to the check in Step 4-b, which includes the proof $\pi_{\tau,partial}$. In Step 5, the client verifies the decrypted result and accepts the result output by the majority, ensuring that a minority of malicious servers cannot interfere with the final result.

The second statement follows from Step 1. The servers validate the client's input and ignore any invalid inputs in the final summation. They record valid clients in a set V and perform aggregation only on this set.

6 Experiment

We evaluate FL protocols using a series of benchmarks on a local machine (11th Gen Intel(R) Core(TM) i9-11900KF Processor with an all-core CPU frequency of 3.50GHz, 16 vCPU, 32GB RAM). Based on the FHE standard [6], we set $N = 2048$, $q =$0x3fffffff000001 (54 bits), $p = 2^{16}$.

6.1 Comparison to Prior Work

We evaluate the performance of both our semi-honest and malicious protocols and compare them with state-of-the-art protocols in both synchronous and asynchronous settings. The results are summarized in Table 5. For Prio+ and Elsa, we utilized their publicly available implementations on GitHub. For BASecAgg, we obtained the runtime using the implementation provided by the paper's authors. Since the implementation of ACORN is not publicly available, we estimated its runtime based on the homomorphic encryption (HE) operations described in their paper.

For the semi-honest setup, our Mario outperforms Flamingo and ACORN by factors of $2-70\times$ and outperforms BASecAgg by a factor of $356\times$. In terms of malicious multi-server performance, we achieve a total runtime that is $3.40\times$ faster compared to Elsa and reduce the communication cost for clients by $3.19\times$. This is due to our protocol's design where only one encrypted model is sent to the leader and the hash is sent to the remaining servers.

Regarding communication cost, Mario shows a significant improvement in client communication for the multi-aggregator setup. This dues to fact that in Mario, the client only needs to send the encrypted model to the leader aggregators and the hash to all other servers, eliminating the additional communication cost associated with $n-1$ servers.

Table 5: **Empirical Comparison of Federated Learning Schemes Setting. SH indicates semi-honest performance with no input validation, MA indicates malicious adversaries performance. IV indicates Input Validation**. The runtime and communication cost are for secure aggregation only, no local training involved. The common setup is $L = 44426, m = 1000$. For ACORN, $(nb, \tau) \in \{(24,2),(72,46),(118,95)\}$. For Asyncronous setting, $k = 100, \tau = 50$ for BASecAgg and $k = 100, n = t = 2$ for Ours. The "/" notation in server communication indicates the cost of leader server / normal server in our protocol.

Setting	Protocol	Runtime (s)			Client Comm. (MB)			Server Comm. (MB)			Multiple servers
		0%	25%	50%	0%	25%	50%	0%	25%	50%	
Sync (SH)	ACORN[11]	0.32	0.69	1.60	**0.18**	**0.19**	**0.20**	228.95	**229.56**	**229.56**	✗
	Prio+[5]	51.02			10.93			5423			✓
	Mario	**0.18**			3.64			5003 / **3.89**			✓
Sync (MA)	Elsa [63]	38.69			11.61			7118			✓
	Mario (w.o IV)	**11.38**			**3.64**			**5003 / 5.95**			✓
	Mario (incl. IV)	123.08			5.70			**5003 / 5.95**			✓
Async (SH)	BASecAgg [67]	11.09	11.09	15.31	1.15	1.15	23.00	**0.79**	**0.79**	**11.61**	✗
	Mario	**0.043**			**3.64**			504 / 3.67			✓

6.2 Our Mario Performance

To understand the performance breakdown of our protocol, we benchmark each component individually, including the semi-honest protocol, key generation, input validation, and decryption verification.

Semi-honest Performance. We present the runtime and communication cost of our semi-honest protocol in Figure 5. The figure shows that the communication cost for the client does not depend on the number of clients participating in the round and varies linearly with and the size of the model L. Interestingly, the communication cost of client is independent from number of servers. This is because the client only needs to send a hash to servers except for the leader.

Key Generation. In Table 6, we present the performance metrics for key generation. The table shows that by generating local keys in parallel, the cost of the key generation step is reduced by factors of $8-47\times$, with higher speedups observed in setups with more servers. Additionally, we report the overhead runtime for public key verification, which incurs a small overhead ranging from 1.17 to 2.70 milliseconds, thanks to the compression/batch technique (BatchSumVerify) described in Figure 2. Most of the runtime in key generation is attributed to the secret key verification step, which involves N group exponentiations and N group multiplications, with costs ranging from 5 to 13 seconds.

Table 6 also presents the costs associated with the join (ThHE.Join) and key refresh (ThHE.ShareRefresh) operations. We do not include the secret share verification costs in the join and refresh metrics. The Join function requires between 14 to 48 milliseconds per new server joining, while the share refresh operation costs between 120 to 456 milliseconds, depending on the number of servers.

Input Validation and Encryption Verification. The performance of input validation and encryption verification involves two key components: range proof generation by clients and range proof verification by servers. We benchmark both the runtime and communication cost for input validation. These

Process	**Runtime (ms)**			
	(6,10)	(8,15)	(11,20)	(16,30)
KeyShare (vanilla)	1023	4022	6680	21127
KeyShare (parallel) (Speed up)	121 ($8\times$)	175 ($23\times$)	271 ($25\times$)	456 ($46\times$)
Public Key Verif.	1.17	1.47	1.95	2.70
Secret Share Verif.	4895	6526	8974	13053
Join	14	14	30	48
ShareRefresh	120	175	271	456

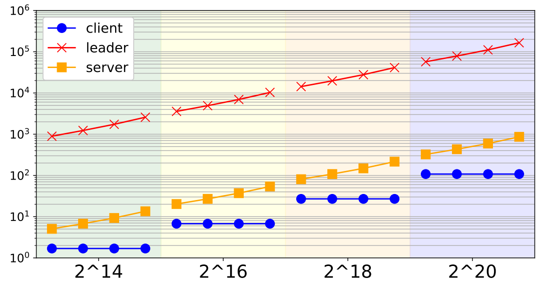
Table 6: **Runtime of Each Section of Key Generation in Milliseconds**. $(\cdot,\cdot)$ denotes different (t,n) assignments. The Join and Share Refresh times exclude the secret share verification costs.

Model size	Client comm. cost	Computation cost (s)	
		client (gen)	server (verify)
2^{14}	270 KB	22	4.63
2^{16}	1.03 MB	94	17.7
2^{18}	4.10 MB	381	71.7
2^{20}	17 MB	1482	286.8

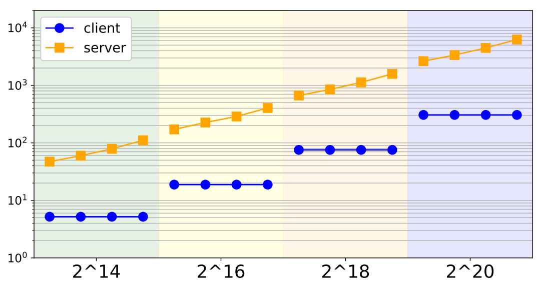
Table 7: **Input Validation and Encryption Verification Estimate**. Total number of clients to verify is $m = 1000$. Server is using 4 cores.

are estimated using Figure 5 from ACORN [11] paper, adding with the Bulletproofs for linear dot product proof.

Note that our approach incurs roughly $5\times$ more runtime compared to ACORN-detect (ACORN protocol to detect malicious client, no robustness), due to the additional proofs of encryption. However, this provides robustness and improves upon ACORN-robust, which would result in approximately $\log^2(m)$ (≈ 100) times more runtime and communication cost than our protocol due to the additional proof required for correct secret sharing.



(a) Communication Cost (MB)



(b) Runtime (ms)

Figure 5: **Mario's Performance in the Semi-Honest Setting**. Numbers on x-axis represent size of model L. Each line consists of 4 points representing the measures for each assignment of (t,n): (6,10),(8,15),(11,20),(16,30).

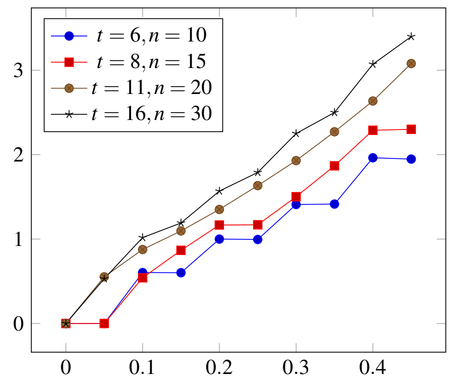


Figure 6: Average number of re-selection over 10000 runs. x-axis: ratio of malicious servers over total number of servers, y-axis: average number of re-selection needed until hitting a set of all honest servers.

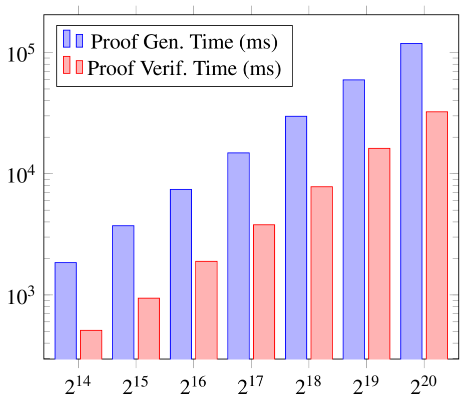


Figure 7: Proof generation and proof verification time (milliseconds) at servers. Blue: generation time, Red: verification time

Number of Re-Selection for Robust. In our privacy-with-robustness protocol, servers must re-select a set of decryptors until they find a set composed entirely of honest servers. This experiment evaluates the expected number of re-selections required given an unknown pre-defined set of malicious servers. The results, shown in Figure 6, indicate that for a setup of 30 servers with at least 16 honest ones, an honest server typically needs to re-select decryptors about 3 times before finding a set of all honest decryptors. The number of re-selections decreases linearly with the ratio of malicious servers. This relatively small number of re-selections is due to the protocol's ability to identify and exclude malicious decryptors during the partial decryption phase, thus improving the efficiency of subsequent re-selections.

Partial and Final Decryption Verification. The cost of decryption verification is divided into two main costs: (1) the proof generation and verification of the linear dot product, and (2) the proof generation and verification of the range of error term e_τ for a party P_τ. Figure 7 presents the runtime for both proof generation and verification across various model sizes.

7 Conclusion

In this work, we present Mario, the first multi-aggregator protocol designed to achieve robustness against both malicious servers and malicious clients, addressing a gap in previous multi-aggregator secure aggregation research [5, 26, 63]. Mario supports both synchronous and asynchronous settings and is secure against recent attacks such as model inconsistency [60] and modifications to model weights and architecture [71]. Furthermore, while not explicitly covered, Mario is fully compatible with Differential Privacy (DP), allowing clients to add DP noise to their local gradients to enhance protection against future attacks targeting secure aggregation.

Future work. In the future, we would like to extend upon this paper (1) a single-aggregator equivalence of Mario; (2) support of other aggregated statistics: min, max, frequency count similar to Prio [26] by extending ThHE to fully homomorphic encryption; (3) improve input validation by using encrypted value of binary input, which was inspired by the approach used in Prio+ [5].

References

[1] https://github.com/Microsoft/SEAL.

[2] Adaptively secure non-interactive threshold cryptosystems. *Theoretical Computer Science*, 2013.

[3] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. ePrint, 2022/1759.

[4] Ittai Abraham, Philipp Jovanovic, Mary Maller, Sarah Meiklejohn, and Gilad Stern. Bingo: Adaptivity and asynchrony in verifiable secret sharing and distributed key generation. Cryptology ePrint Archive, Paper 2022/1759, 2022. https://eprint.iacr.org/2022/1759.

[5] Srinivas Addanki, Kristin Garbe, Eliot Jaffe, Rafail Ostrovsky, and Antigoni Polychroniadou. Prio+: Privacy preserving aggregate statistics via boolean shares. *IACR ePrint Archive*, 2021:576, 2021.

[6] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[7] Renas Bacho, Daniel Collins, Chen-Da Liu-Zhang, and Julian Loss. Network-agnostic security comes (almost) for free in dkg and mpc. Cryptology ePrint Archive, Paper 2022/1369, 2022. https://eprint.iacr.org/2022/1369.

[8] Renas Bacho and Julian Loss. On the adaptive security of the threshold bls signature scheme. CCS '22.

[9] Saikrishna Badrinarayanan, Aayush Jain, Nathan Manohar, and Amit Sahai. Secure MPC: Laziness leads to GOD. pages 120–150.

[10] Laasya Bangalore, Mohammad Hossein Faghihi Sereshgi, Carmit Hazay, and Muthuramakrishnan Venkitasubramaniam. Flag: A framework for lightweight robust secure aggregation. New York, NY, USA, 2023. Association for Computing Machinery.

[11] James Bell, Adrià Gascón, Tancrède Lepoint, Baiyu Li, Sarah Meiklejohn, Mariana Raykova, and Cathie Yun. Acorn: Input validation for secure aggregation. ePrint, 2022/1461, 2022.

[12] James Henry Bell, Kallista A. Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly)logarithmic overhead.

[13] James Bell-Clark, Adrià Gascón, Baiyu Li, Mariana Raykova, and Phillipp Schoppmann. Willow: Secure aggregation with one-shot clients. Cryptology ePrint Archive, Paper 2024/936, 2024. https://eprint.iacr.org/2024/936.

[14] Arjun Nitin Bhagoji, Supriyo Chakraborty, Prateek Mittal, and Seraphin Calo. Analyzing federated learning through an adversarial lens. ICML'19.

[15] Battista Biggio, Blaine Nelson, and Pavel Laskov. Poisoning attacks against support vector machines. ICML'12.

[16] Dan Boneh, Elette Boyle, Henry Corrigan-Gibbs, Niv Gilboa, and Yuval Ishai. Lightweight techniques for private heavy hitters. Cryptology ePrint Archive, Paper 2021/017, 2021. https://eprint.iacr.org/2021/017.

[17] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter M. R. Rasmussen, and Amit Sahai. Threshold cryptosystems from threshold fully homomorphic encryption.

[18] Katharina Boudgoust and Peter Scholl. Simple threshold (fully homomorphic) encryption from lwe with polynomial modulus. Cryptology ePrint Archive, Paper 2023/016, 2023. https://eprint.iacr.org/2023/016.

[19] Ronald Newbold Bracewell and Ronald N Bracewell. *The Fourier transform and its applications*. McGraw-Hill New York, 1986.

[20] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. ITCS '12.

[21] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. 2011.

[22] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. Cryptology ePrint Archive, Paper 2017/1066, 2017. https://eprint.iacr.org/2017/1066.

[23] T. H. Hubert Chan, Elaine Shi, and Dawn Song. Privacy-preserving stream aggregation with fault tolerance. In Angelos D. Keromytis, editor, *Financial Cryptography and Data Security*, 2012.

[24] Jung Hee Cheon, Wonhee Cho, and Jiseung Kim. Improved universal thresholdizer from threshold fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 545, 2023.

[25] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In *ASIACRYPT 2017*.

[26] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[27] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-fhe using noise flooding. In *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '23, page 35–46, New York, NY, USA, 2023. Association for Computing Machinery.

[28] Morten Dahl, Daniel Demmler, Sarah El Kazdadi, Arthur Meyre, Jean-Baptiste Orfila, Dragos Rotaru, Nigel P. Smart, Samuel Tap, and Michael Walter. Noah's ark: Efficient threshold-fhe using noise flooding. Cryptology ePrint Archive, Paper 2023/815, 2023. https://eprint.iacr.org/2023/815.

[29] Rafael del Pino, Vadim Lyubashevsky, and Gregor Seiler. Short discrete log proofs for FHE and ring-LWE ciphertexts. Cryptology ePrint Archive, Paper 2019/057, 2019. https://eprint.iacr.org/2019/057.

[30] Yvo Desmedt. Threshold cryptosystems. In *Advances in Cryptology — AUSCRYPT '92*.

[31] F. Betül Durak, Chenkai Weng, Erik Anderson, Kim Laine, and Melissa Chase. Precio: Private aggregate measurement via oblivious shuffling. Cryptology ePrint Archive, Paper 2021/1490, 2021.

[32] Taher ElGamal. On computing logarithms over finite fields.

[33] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. ePrint, 2012/144, 2012.

[34] Serge Fehr. Span programs over rings and how to share a secret from a module. Master's thesis, ETH Zurich, Institute for Theoretical Computer Science, 1998.

[35] Frank A. Feldman. Fast spectral tests for measuring nonrandomness and the DES. pages 243–254, 1988.

[36] Qi Gao, Yi Sun, Xingyuan Chen, Fan Yang, and Youhe Wang. An efficient multi-party secure aggregation method based on multi-homomorphic attributes. *Electronics*, 13(4), 2024.

[37] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure applications of pedersen's distributed key generation protocol. In *Topics in Cryptology — CT-RSA 2003*.

[38] Craig Gentry, Shai Halevi, and Vadim Lyubashevsky. Practical non-interactive publicly verifiable secret sharing with thousands of parties. Cryptology ePrint Archive, Paper 2021/1397, 2021. https://eprint.iacr.org/2021/1397.

[39] Ming Hao, Hongyi Li, Guanhong Xu, Hui Chen, and Tianwei Zhang. Efficient, private and robust federated learning. In *ACSAC*, 2021.

[40] Li He, Sai Praneeth Karimireddy, and Martin Jaggi. Secure byzantine-robust machine learning. *CoRR*, 2020.

[41] Martin Hirt and Jesper Buus Nielsen. Robust multiparty computation with linear communication complexity. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference*, volume 4117 of *Lecture Notes in Computer Science*, pages 463–482. Springer, 2006.

[42] Yuval Ishai, Eyal Kushilevitz, Yehuda Lindell, and Erez Petrank. On combining privacy with guaranteed output delivery in secure multiparty computation. In *Advances in Cryptology - CRYPTO 2006, 26th Annual International Cryptology Conference*, volume 4117 of *Lecture Notes in Computer Science*, pages 483–500. Springer, 2006.

[43] Y. Jiang, X. Luo, Y. Wu, X. Xiao, and B. Ooi. Protecting label distribution in cross-silo federated learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 112–112, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[44] E. Kabir, Z. Song, M. Rashid, and S. Mehnaz. Flshield: A validation based federated learning framework to defend against poisoning attacks. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 140–140, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[45] Swanand Kadhe, Nived Rajaraman, Onur Ozan Koyluoglu, and Kannan Ramchandran. Fastsecagg: Scalable secure aggregation for privacy-preserving federated learning. *ArXiv*, abs/2009.11248, 2020.

[46] Seny Kamara, Payman Mohassel, and Mariana Raykova. Outsourcing multi-party computation. ePrint, 2011/272, 2011.

[47] Harish Karthikeyan and Antigoni Polychroniadou. OPA: One-shot private aggregation with single client interaction and its applications to federated learning. Cryptology ePrint Archive, Paper 2024/723, 2024. https://eprint.iacr.org/2024/723.

[48] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. CCS '17.

[49] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. SWIFT: Super-fast and robust privacy-preserving machine learning. Cryptology ePrint Archive, Paper 2020/592, 2020.

[50] Hanjun Li, Sela Navot, and Stefano Tessaro. Popstar: Lightweight threshold reporting with reduced leakage. Cryptology ePrint Archive, Paper 2024/320, 2024. https://eprint.iacr.org/2024/320.

[51] Yehuda Lindell and Ariel Nof. A framework for constructing fast mpc over arithmetic circuits with malicious adversaries and an honest-majority. Cryptology ePrint Archive, Paper 2017/816, 2017. https://eprint.iacr.org/2017/816.

[52] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Advances in Cryptology – EUROCRYPT 2010*.

[53] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. Flamingo: Multi-round single-server secure aggregation with applications to private federated learning. ePrint, 2023/486, 2023.

[54] R. Moenck and A. Borodin. Fast modular transforms via division. In *13th Annual Symposium on Switching and Automata Theory (swat 1972)*, 1972.

[55] Christian Mouchet, Elliott Bertrand, and Jean-Pierre Hubaux. An efficient threshold access-structure for rlwe-based multiparty homomorphic encryption. *J. Cryptol.*, 36(2):10, 2023.

[56] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. Cryptology ePrint Archive, Paper 2020/304, 2020.

[57] John Nguyen, Kshitiz Malik, Hongyuan Zhan, Ashkan Yousefpour, Mike Rabbat, Mani Malek, and Dzmitry Huba. Federated learning with buffered asynchronous aggregation. In *AISTATS 2022*.

[58] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*.

[59] Jeongeun Park. Homomorphic encryption for multiple users with less communications. ePrint, 2021/1085.

[60] Dario Pasquini, Danilo Francati, and Giuseppe Ateniese. Eluding secure aggregation in federated learning via model inconsistency. *CCS*, 2022.

[61] Torben Pryds Pedersen. Non-interactive and information-theoretic secure verifiable secret sharing. In Joan Feigenbaum, editor, *CRYPTO '91*.

[62] Michael Rabin. Verifiable secret sharing. In *Proceedings of the 21st Annual ACM Symposium on Theory of Computing (STOC '89)*, 1989.

[63] Mayank Rathee, Conghao Shen, Sameer Wagh, and Raluca Ada Popa. Elsa: Secure aggregation for federated learning with malicious actors. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1961–1979, 2023.

[64] Aaron Segal, Antonio Marcedone, Benjamin Kreuter, Daniel Ramage, H. Brendan McMahan, Karn Seth, K. A. Bonawitz, Sarvar Patel, and Vladimir Ivanov. Practical secure aggregation for privacy-preserving machine learning. In *CCS*, 2017.

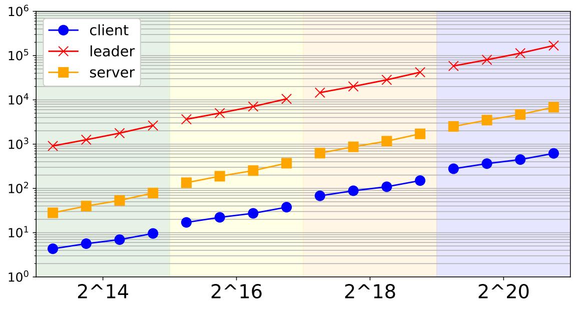
[65] Elaine Shi, T.-H. Hubert Chan, Eleanor Gilbert Rieffel, Richard Chow, and Dawn Song. Privacy-preserving aggregation of time-series data. In *NDSS 2011*.

[66] Gustavus J. Simmons. How to (really) share a secret. In Shafi Goldwasser, editor, *CRYPTO' 88*.

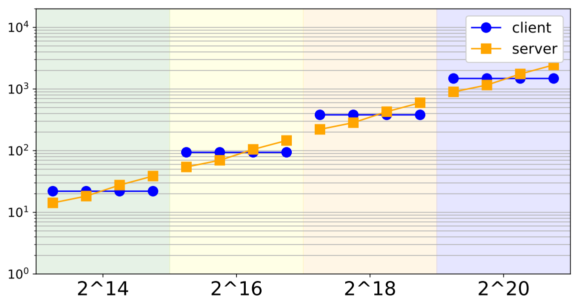
[67] Jinhyun So, Ramy E. Ali, Basak Güler, and Amir Salman Avestimehr. Secure aggregation for buffered asynchronous federated learning. *CoRR*, abs/2110.02177.

[68] Jinhyun So, Basak Guler, and A. Salman Avestimehr. Turbo-aggregate: Breaking the quadratic aggregation barrier in secure federated learning. ePrint, 2020/167.

[69] Elina van Kempen, Qifei Li, Giorgia Azzurra Marson, and Claudio Soriente. Lisa: Lightweight single-server secure aggregation with a public source of randomness, 2023.



(a) Communication Cost (MB)



(b) Runtime (s)

Figure 8: Privacy with robustness performance. Numbers on x-axis represent size of model L. Each line consists of 4 points representing the measures for each assignment of (t,n): (6,10),(8,15),(11,20),(16,30).

[70] Haibin Zhang, Sisi Duan, Chao Liu, Boxin Zhao, Xuanji Meng, Shengli Liu, Yong Yu, Fangguo Zhang, and Liehuang Zhu. Practical asynchronous distributed key generation: Improved efficiency, weaker assumption, and standard model. Cryptology ePrint Archive, Paper 2022/1678, 2022. https://eprint.iacr.org/2022/1678.

[71] J. Zhao, A. Sharma, A. Elkordy, Y. H. Ezzeldin, S. Avestimehr, and S. Bagchi. Loki: Large-scale data reconstruction attack against federated learning through model manipulation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 30–30, Los Alamitos, CA, USA, may 2024. IEEE Computer Society.

[72] Wenting Zheng, Raluca Ada Popa, Joseph E. Gonzalez, and Ion Stoica. Helen: Maliciously secure coopetitive learning for linear models. In *SP 2019*.

A Estimate time of robustness protocol

Based on the number presented in Section 6.2, we estimate the runtime and communication cost of the whole protocol. The results are shown in Figure 8. Note that the client runtime is slow due to the costly input validation operation, which is the common issue shared with state-of-the-art ACORN [11].

B Preliminaries

Let N be a power of two and q be an integer. We denote by $R = \mathbb{Z}[X]/(X^N+1)$ the ring of integers of the $(2N)$-th cyclotomic field and $R_q = \mathbb{Z}_q[X]/(X^N+1)$ the residue ring of R modulo q. We represent an element $a = \sum_{i=0}^{(N-1)} a_i X^i \in R$ by the vector of its coefficients $(a_0, \dots, a_{(N-1)})$. Given a probability distribution D, we use $x \leftarrow D$ to denote that x is sampled from D. A distribution χ over the integers is called B-bounded if it is supported on $[-B, B]$.

Let g be a generator of the field $\mathbb{Z}_q$, let $a \in R_q$ be a arbitrary polynomial in R_q, $A = (A_1, \dots, A_N), B = (B_1, \dots, B_N) \in \mathbb{Z}_q^N$ be two arbitrary vectors size N. We write as $a = \sum_{i=0}^{N} a_i x^i$, then we define the following three operators:

- Power of g: $g^a := (g^{a_0}, \dots, g^{a_N})$
- Multiplication: $AB = (A_1B_1, A_2B_2, \dots, A_NB_N)$
- Power on Z_q^N: $A^n = AA\dots A$ (for n times) for any $n \in \mathbb{N}$

B.1 Polynomial-LWE based HE

We construct our ThHE based on traditional lattice HE schemes from the polynomial learning with error (PLWE) assumption [21] (See Def 2) which is a simplified version of the Ring-LWE [52]. These lattice HE schemes are faster compared to other HE (based on Elgamal or Paillier) [32, 58] and provide quantum resistance.

For easy understanding of our ThHE, we choose to present a generic PLWE-based single-key HE scheme in Figure 9. Note that our ThHE can be straightforwardly modified to work with other HE schemes such as FV [33], BGV [20], and CKKS [25].

PARAMETERS: The secret key space χ_1, the noise space χ_2, the plaintext space $R_p = \mathbb{Z}_p[X]/(X^N+1)$, the ciphertext and public key space $R_q = \mathbb{Z}_q[X]/(X^N+1)$, and $\Delta = \lfloor q/p \rfloor$

```
HE.KeyGen() → (sk, pk)
    • Sample $s \leftarrow \chi_1$
    • Sample $a, a_0 \leftarrow R_q$ and $e, e_0 \leftarrow \chi_2$
    • Output $(\mathsf{sk}, \mathsf{pk})$ where $\mathsf{sk} = s$ and $\mathsf{pk} = ([-a \cdot s + e]_q, a)$
HE.Encrypt(pk, $m$) → ct
    • Sample $u \leftarrow \chi_1$ and $e_1, e_2 \leftarrow \chi_2$
    • Let $p_0 = \mathsf{pk}[0]$, $p_1 = \mathsf{pk}[1]$
    • Compute $c_0 = [p_0 \cdot u + e_1 + \Delta \cdot m]_q$
      and $c_1 = [p_1 \cdot u + e_2]_q$
    • Output $(c_0, c_1)$
HE.Decrypt(sk, ct) → $m$
    • Let $c_0 = \mathsf{ct}[0]$, $c_1 = \mathsf{ct}[1]$
    • Compute $v = [c_0 + c_1 \cdot s]_q$
    • Output $[\frac{1}{\Delta} \cdot v]_p$
HE.Add(ct$_1, \dots,$ ct$_k$) → ct
    • Output $\mathsf{ct}_1 + \dots + \mathsf{ct}_k$
```

Figure 9: PLWE-based Additive HE Construction.

Definition 2. *(**Polynomial-LWE** [21]) For security parameter λ and $q = q(\lambda) > 1$, set $R = \mathbb{Z}[X]/(X^N + 1)$ and $R_q = \mathbb{Z}_q[X]/(X^N + 1)$. For a random element $s \in R_q$ and a distribution $\chi = \chi(\lambda)$ over R, denote with $A^{(q)}_{s,\chi}$ the distribution obtained by choosing a uniformly random element $a \leftarrow R_q$ and a noise term $e \leftarrow \chi$ and outputting $(a, [a \cdot s + e]_q)$. The Polynomial-LWE(PLWE) problem is to distinguish between the distribution $A^{(q)}_{s,\chi}$ and the uniform distribution.*

B.2 Threshold Scheme and Secret Sharing

Threshold Scheme & Shamir Secret Sharing. A threshold scheme (TS) with a threshold t over a secret space $\mathcal{K}$ is two probabilistic polynomial-time (PPT) algorithms $(\mathsf{Share}_t, \mathsf{Res}_t)$ where:

- $\mathsf{Share}_t(k, n) \to (s_1, \ldots, s_n)$: On input a secret $k \in \mathcal{K}$ and $n > t$, the sharing algorithm returns n shares $\{s_1, \ldots, s_n\}$.
- $\mathsf{Res}_t(s_{i,1} \ldots, s_{i,t}) \to k$: On input a set of shares $(s_{i,1} \ldots, s_{i,t})$, the combining algorithm outputs the secret $k \in \mathcal{K}$

The scheme must satisfy the following properties:

- Correctness: $\mathsf{Res}_t(s_{i,1} \ldots, s_{i,t}) = k$
- Security: Any subset $S = \{s_i\}_{i \in [n]}$ for $|S| < t$ reveals nothing about the secret k.

Shamir secret sharing (S3) is a threshold scheme based on Lagrange interpolation. To share a secret s in a finite field $\mathbb{Z}_p$, the sharing phase of S3 chooses a polynomial f of degree $t-1$ such that $[f(0)]_p = s$, with all other coefficients chosen uniformly in $\mathbb{Z}_p$. Each party P_i receives a share in the form of the point $(i, y_i) = (i, [f(i)]_p)$ on the polynomial. The interpolation theorem guarantees that any t of the shares can uniquely determine the polynomial f, and hence recover the secret $f(0)$. The traditional Shamir secret-sharing scheme is typically implemented over a field. Therefore, this paper adopts Lagrange interpolation in $\mathbb{Z}_p$ for simplicity. For a more comprehensive understanding of Shamir secret sharing over a ring, we refer the reader to [34].

Verifiable Secret Sharing. Given a secret s to be shared to n parties, namely $[\![s]\!]_i$ to party P_i, so that any t parties can together reconstruct s, Verifiable Secret Sharing (VSS) [62] is the problem that ensure each of the n parties receive the correct share, even in the present of a malicious dealer. In terms of Shamir secret sharing, it is equivalent to the verification that $[\![s]\!]_i = s + \sum_{j=1}^{t-1} a_j i^j$ where $s + a_1 x + \cdots + a_{t-1} x^{t-1}$ represents the polynomial that was used in Shamir sharing.

Additive Secret Sharing. Given a set of n parties $\{P_1, \ldots, P_n\}$, to additively secret share $[\![x]\!]$ an ℓ-bit value x of P_i to other parties, he first chooses $x^i \leftarrow \mathbb{Z}_{2^\ell}$ uniformly at random such that $x = \sum_{j=1}^{n} x^j \mod 2^\ell$, and then sends each x^j to the party P_j. For ease of composition, we omit the mod. To reconstruct an additive shared value $[\![x]\!]$, all parties P_j sends $[\![x]\!] = x^j$ to P_i, who locally reconstructs the secret value by computing $x \leftarrow \sum_{i=1}^{n} x^j$. In this work, we define the additive secret sharing of a value x as $[\![x]\!]$.

C Sketched security proof for BatchSumVerify

- *Completeness:* If $x = x_1 + \cdots + x_m$, then $C_v = \prod_{i=1}^{t} g^{[\![v]\!]_i} = g^{\sum [\![v]\!]_i} = g^{\sum \langle \alpha, x_i \rangle} = g^{\langle \alpha, x \rangle}$. Therefore, if the statement is correct, then P_v will always accept.
- *Soundness:* If $x \neq x_1 + \cdots + x_m$, the equation holds with a probability of only $\frac{1}{q-1}$ for $\lceil \log q \rceil = 54$ in our implementation.
- *Zero-knowledge:* The simulator on verifier P_v can simply generate $g^{[\![v]\!]_i}$ for each P_i by randomly sampling values from $\mathbb{Z}_q$.

D Dynamic Threshold HE

We realize the dynamic threshold HE in Definition 1 in Figure 11. The verifiable secret share of secret key is inspired by Feldman's verifiable secret sharing scheme [35]. In addition, we use additive verification function in Section 3.2 to verify if the public key are jointly computed correctly. The encryption/decryption functions are similar to BFV encryption scheme [33].

Throughout this section, we refer to the *servers as 'parties'* to describe our ThHE scheme employed in our FL framework. It is less likely for those parties (servers) to drop out frequently during the protocol execution compare to remote clients in FL.

D.1 System Design

In our ThHE, we classify participants into two categories: pivot and non-pivot. Although any party can act as a pivot, we recommend selecting a party with powerful devices and stable connectivity, as pivots perform more work in $\mathsf{ThHE.SecretKeyGen}()$. We assume that the IP addresses of participants are publicly available, enabling each party to determine whether a particular party is online.

During $\mathsf{ThHE.Join}()$, which generates a secret key for the new P_v, it requires a set A of at least t online parties. If the set of t pivot parties is available and online, P_v selects them as set A. Otherwise, P_v selects a random online set A and announces the IP addresses of the selected parties. In the event that a party drops out in the middle of the process, P_v can re-select another online party.

Our system also includes a combiner that aggregates information from different parties. Any party can act as a combiner, including pivot/non-pivot parties, or the leader server in our FL application.

D.2 Construction in the Semi-honest Setting

Our dynamic ThHE protocol builds on the foundational framework presented in [55, 56, 59], where servers use Shamir secret sharing to distribute their local secret values among themselves. For completeness, we provide a detailed description of the full construction in this section. Some aspects of this approach are derived from previous work.

In our ThHE, there is a *single public key* pk, under which all messages can be encrypted by different parties. Hence, encryption and additive homomorphic computation are efficient and independent of the number of decryptors/parties as they are performed in a similar fashion to single-key HE. In this section, we show how key generation algorithm and decryption work in ThHE using the Shamir secret sharing (S3) scheme. Figure 11 formally presents our ThHE construction. We provide a detailed discussion on the correctness and security for our ThHE construction in Appendix E.

D.2.1 Key Generation in ThHE

In the previous THE schemes [9, 17], linear secret sharing (LSS) [66] is used in the key setup procedure. While LSS can handle the noise blow-up in the decryption process, it needs either a trusted third party or an expensive MPC to generate the shares (the shared secret keys) for the parties. Moreover, LSS requires a fixed set of parties as input. Thus, LSS is not suitable for multi-round SA with one-time setup. We propose to use distributed Shamir secret schemes (S3) [30, 37] for ThHE, similar to [55]. However, direct use of the S3 causes decryption fail because of a large Lagrange coefficient, which will be discussed in detail below.

Secret Key Generation. We assume that there is a set of t pivot parties $\{P_1,\ldots,P_t\}$. Adapting S3 protocols [30, 37], we first present how the pivots generate secret shares for other parties. Our protocol starts with each pivot choosing a random polynomial $f_i(X)=\sum_{k=0}^{(t-1)} c_{i,k}X^k$ of degree $(t-1)$ where $c_{i,0}\leftarrow\chi_1$ (the B_1-bounded distribution) and $c_{i,k>0}\leftarrow R_q$. The pivot then sends a share $[f_i(j)]_q$ to each party $P_{j\in[n]}$. The $P_{j\in[n]}$ can add up its obtained shares to get a final share (i.e., the shared secret key) as $\mathsf{sk}_j=[\sum_{i=1}^t f_i(j)]_q$. Due to linearity, the result is also a valid Shamir secret share where the final secret value $\mathsf{sk}=[\sum_{i=1}^t c_{i,0}]_q$ is additively shared by t pivots. For the pivot $P_{i\in[t]}$, their final share also consists of the coefficient $c_{i,0}$ so that the t pivots can generate a new share for a new user in the same manner as described above. Additionally, keeping the $c_{i,0}$ allows the t pivots to easily compute the public key and the decryption as we will discuss later.

Indeed, the sum $\sum_{i=1}^t c_{i,0}$ is bounded by tB_1 as each term $c_{i,0}$ is sampled from the B_1-bounded distribution χ_1. Consequently, this sum is considerably smaller than the q value of the HE setting. For example, we evaluate our ThHE application (Federated Learning) with $t\le 1000$, and utilize the single-key HE parameters from SEAL [1] library where $\lceil\log q\rceil=54$ and B_1 is relatively small (such as $B_1=1$ in the ternary distribution or $B_1=26$ in the Gaussian distribution with standard deviation $\sigma=3.2$). Thus, we can omit the module q in the secret key sk and represent sk as $\sum_{i=1}^t c_{i,0}$. When the secret B_1-bounded distribution χ_1 (e.g., Gaussian distribution) has an additive property, we can infer that sk is sampled from the tB_1-bounded distribution χ_1'.

Public Key Computation. Computing pk can be naturally completed using generic techniques from MPC, with each party holding a shared key sk_i as a local input, and a common vector a. However, this solution might be inefficient. In the following, we present an efficient protocol to compute ThHE.PublKeyComp. For ease of composition, we omit the second part of the pk (a vector a).

In the case that all pivots are alive, the pk can be computed easily as follows. Each pivot broadcasts a partial public key $\mathsf{pk}_i=[-a\cdot c_{i,0}+e_i]_q$ for a small sample noise e_i from the B_2-bounded distribution χ_2. A combiner computes $\mathsf{pk}=\left[\sum_{i=1}^t \mathsf{pk}_i\right]_q=[a\cdot s+e]_q$, where $e=\sum_{k\in A} e_k$ is in the tB_2-bounded distribution χ_2'. However, when one of the pivots is not available, it becomes more challenging to compute pk from the S3 shares.

Using a similar approach proposed in generating secret keys, one can apply Lagrange linear combination. Concretely, given t online parties $P_{k\in A}$, each can compute the inner product $\mathsf{pk}_k=-a\cdot\mathsf{sk}_k$ using its secret share $\mathsf{sk}_k\in R_q$. Then, pk_k is sent to the combiner which can compute pk as $\left[\sum_{k\in A}\mathsf{pk}_k L_A(0,k)+e\right]_q$ for an additive noise e. Unfortunately, this construction is insecure as the secret sk_i can be easily recovered from pk_k by the combiner.

Relying on the PLWE assumption, a possible approach to resolve this issue is for a party $P_{k\in A}$ to add small additive noise e_k to the inner product, and reduce the result to module q, i.e., $\mathsf{pk}_k=[-a\cdot\mathsf{sk}_k+e_k]_q$. The combiner computes pk as $\left[\sum_{k\in A}\mathsf{pk}_k L_A(0,k)\right]_q$. However, this solution might not preserve the correct computation of the public key for two reasons: (1) the reduction of each term $a\cdot\mathsf{sk}_k$ before multiplying with a "rational" Lagrange coefficient $L_A(0,k)$ might cause an incorrect reconstruction of the secret key; and (2) the noise increases significantly because of a large Lagrange coefficient when the combiner computes pk which has a noise $\left[\sum_{k\in A} e_k L_A(0,k)\right]_q$. The second issue has been pointed out in [17] for a different but related problem. To address this, the authors proposed to scale Lagrange coefficients to be integers by multiplying with a term $(n!)^2$, and also to increase the modulus q of the scheme by $\log(n!)$ to support its additional noise growth. However, the modulus q depends on the value n, which leads to a non-compact ciphertext size.

Hence, we propose a different approach to ensure compactness property while allowing decryption success. Specifically, we assume that the set of A is publicly known by any party in A, and each party $P_{k\in A}$ computes $\mathsf{pk}_k=[-a\cdot\mathsf{sk}_k L_A(0,k)+$

$e_k]_q$ where $e_k \leftarrow \chi_2$. One can consider that the pk_k value is an additive share of the joint public key pk. This allows the combiner to compute pk as $\mathsf{pk} = [\sum_{k\in A} \mathsf{pk}_k]_q$ without making the noise blowup significantly. Note that pk_k reveals no information about sk_k since the PLWE assumption says that $\big([-aL_A(0,k)]_q, [-a\cdot \mathsf{sk}_k L_A(0,k)+e_k]_q\big)$ hides sk_k. However, this solution raises another problem: the coefficients of $-a\cdot\mathsf{sk}_k L_A(0,k)$ might not be integers. In Section D.2.5, we show how to convert the term $[-a\cdot \mathsf{sk}_k L_A(0,k)]_q$ to R_q using our simple algorithm $\mathsf{ConvMult}_q()$. The key idea is to find another value that makes the term $-a\cdot\mathsf{sk}_k$ divisible by the denominator of $L_A(0,k)$. In summary, the partial public key has a formula $\mathsf{pk}_k = [\mathsf{ConvMult}_q(-a\cdot\mathsf{sk}_k, L_A(0,k)) + e_k]_q$. The joint public key is $\mathsf{pk} = [\sum_{k\in A}\mathsf{pk}_k]_q = [a\cdot s+e]_q$ where the noise $e = \sum_{k\in A} e_k$ is in the space χ_2'.

D.2.2 ThHE.Join Computation

The main challenge is how to generate a new share sk_ν for a new party P_ν when a subset of pivot parties drop out after the initial setup (ThHE.KeyGen). To address the issue, we propose to use Lagrange linear combination. Specifically, it is assumed that there are t alive (either pivot or non-pivot) parties $P_{k\in A}$ for some party's IDs $k\in A$. The new share sk_ν can be computed as $\sum_{k\in A}\mathsf{sk}_k L_A(\nu,k)$ where $L_A(\nu,k) = \prod_{j\in A\setminus\{k\}} \frac{\nu-k}{j-k}$ is a Lagrange coefficient. However, it is insecure if each $P_{k\in A}$ sends a term $\mathsf{sk}_k L_A(\nu,k)$ to P_ν since the P_ν can factorize the product and learn the sk_k. To handle this, we propose to mask each term $\mathsf{sk}_k L_A(\nu,k)$ with a zero share z_k where $\sum_{k\in A} z_k = 0$. The masking prevents the attacker from learning an individual sk_k and the zero-shares will be canceled out in the final sum to maintain the correctness of the sk_ν computation.

We now describe the zero-share generation [48, 64] as follow. Each party $P_{k\in A}$ chooses a random $z_{k,j}$ for $j\in A, j>k$, and sends $z_{k,j}$ to the party P_j. After receiving $z_{j,k}$ from the party $P_{j\in A, j<k}$, the P_k computes the zero share $z_k = \sum\limits_{j\in A, j<k} z_{j,k} - \sum\limits_{j\in A, j>k} z_{k,j}$. It is easy to see that $\sum_{k\in A} z_k = 0$ as desired.

D.2.3 ThHE.ShareRefresh Computation

The protocol calls ThHE.ShareRefresh to redistribute the shares among servers. When many new servers join, the protocol ensures that a combination of new and existing servers cannot form a sufficiently large committee (e.g., more than t servers) capable of reconstructing the secret key. Therefore, assuming the maximum number of malicious servers is $t-\mu$ (for some $\mu>0$) before a new server joins, ThHE.ShareRefresh must be invoked before every μ new servers join, in order to increase the threshold t to $t+\mu$

The way ThHE.ShareRefresh works is straightforward: a set of t' servers is selected to act as the new pivot. Each of these t' servers generates a t' out of n Shamir sharing of zero

PARAMETERS: The space $R_q = \mathbb{Z}_q[X]/(X^N+1)$ and a set of indices A
INPUT: $x = (x_1,\ldots,x_N)\in R_q$ and $L_A(0,k) = \prod_{j\in A\setminus\{k\}} \frac{-k}{j-k}$ is a Lagrange coefficient.
OUTPUT: $[xL_A(0,k)]\in R_q$

```
1: procedure ConvMult_q($x, L_A(0,k)$)
2:     Find $a,b$ such that $\frac{a}{b} = L_A(0,k)$ and $\gcd(a,b)=1$
3:     Find $q_{inv}$ such that $q_{inv}q = 1 \mod b$
4:     for $i = 1,\ldots,N$ do
5:         Compute $\bar{x}_i = -q_{inv}x_i \mod b$
6:         Compute $x_i' = x_i + \bar{x}_i * q$          $\triangleright\ x_i' = x_i \mod q$
7:     end for
8:     return $[x'L_A(0,k)]_q$ where $x' = (x_1',\ldots,x_N')$
9: end procedure
```

Figure 10: Computation in Ring

(by setting the less significant coefficient $c_{0,k}$ of the polynomial $f_i(X)$ to zero) and sends the share $v_{i\in[t'],j\in[n]}$ to the corresponding server P_j. Each server $P_{j\in[n]}$ then updates its current share sk_j with $[\mathsf{sk}_j + \sum_{i=1}^{t'} v_{i,j}]_q$. Since the original share can be viewed as a t' out of n Shamir share of a polynomial where the highest $t'-t$ coefficients are zero, the updated share effectively becomes a Shamir share of the secret key under the new threshold. The security is maintained because the new shares are derived from a polynomial of degree $t'-1$.

D.2.4 Encryption and Decryption

Our encryption follows directly from the encryption algorithm of the conventional single-key HE. The encryption process involves utilizing the noise space χ_2'. This is for consistency with the underlying HE construction, as depicted in Figure 9. Note that the noise added to the public key pk also in this domain.

In most existing HE schemes, the decryption has the form $[c_0 + c_1\cdot s]_q$ where (c_0,c_1) is a ciphertext. Using the techniques proposed in Section D.2.1, the decryption algorithm can be implemented in a similar way to compute the public key without revealing the secret key. Concretely, for any set A of t alive parties, each $P_{k\in A}$ can locally compute a partial decryption $\mathsf{p}_k = [\mathsf{ConvMult}_q(c_1\cdot\mathsf{sk}_k, L_A(0,k)) + e_k]_q$ for a small noise $e_k\leftarrow\chi_2$. The final decryption algorithm outputs $\big[p/q[c_0+\sum_{k\in A}\mathsf{p}_k]_q\big]_p$ which is the desired plaintext when the noise is sufficiently small.

If all t pivot parties are available, the partial decryption can be simply implemented as follows. Each pivot $P_{i\in[t]}$ sends $\mathsf{p}_i = [c_1\cdot c_{i,0} + e_i]_q$ to a combiner, where $e_i\leftarrow\chi_2$ is a small random noise. The combiner can perform a final decryption by computing $\big[p/q[c_0+\sum_{i=1}^t \mathsf{p}_i]_q\big]_p$.

D.2.5 Computation in Ring

Recall that the pk is equal to $\big[\sum_{k\in A}[-a\cdot\mathsf{sk}_k L_A(0,k)+e_k]_q\big]_q$

where each of the additive terms can be locally computed by each party $P_{k\in A}$, and the $L_A(0,k)$ is not an integer. We present a method named ConvMult to correctly evaluate the value as a member of the finite field. The method is presented in Figure 10 .

To make the public key computation returns a correct form of $\mathsf{pk} = [-a \cdot s + e]_q$, we have to deal with non-integer Lagrange coefficients $L_A(0,k)$. To this end, we introduce a function $\mathsf{ConvMult}_q$ which aims to convert a term $xL_A(0,k)$ to a polynomial in R_q, where $x = (x_1,\dots,x_N) \in R_q$ (in the public key computation, x is $-a\cdot \mathsf{sk}_k$). The $\mathsf{ConvMult}_q$ works as below.

We present $L_A(0,k)$ as a fraction $\frac{a}{b}$ where $gcd(a,b) = 1$. Our goal is to make $x_{i\in[N]}$ to be divisible by b. It can be done as follow. We first find $\bar{x}_i$ satisfying that

$$x_i + q\bar{x}_i = 0 \mod b \tag{4}$$

In our ThHE scheme, q is a large prime number. In addition, b is less than $\prod_{j\in A\setminus\{k\}}(j-k)$ for the set of parties' indices A of size t. Thus, we have $\gcd(q,b) = 1$. Consequently, Equation (4) has the unique solution $\bar{x}_i = -x_i q_{inv} \mod b$, where $q_{inv}q = 1 \mod b$. Furthermore, when we define $x'_i = x_i + \bar{x}_i q$, we have $x'_i = x_i \mod q$ and $x'_i = x_i + -x_i q_{inv} q = 0 \mod b$. Therefore, x'_i is dividable by b as desired and $[x_i L_A(0,k)]_q = [x'_i L_A(0,k)]_q$. The function $\mathsf{ConvMult}_q$ is presented in Figure 10.

D.3 Verifable Key Generation

When distributing threshold secret shares of the secret key, it is essential to prevent malicious (pivot) parties from sending incorrect shares. To address this, we use a method analogous to Feldman's verifiable secret sharing scheme [35].

We integrate the batch checking of share correctness from [51] with Feldman's verifiable sharing approach. This combination enables the parties in the protocol to verify the correctness of the shares they receive. Details on how we implement share correctness are presented in Figure 12.

- *Completeness:* If the share sk_i is correct $\forall i \in A$, then SecretKeyVerify always outputs "accept".
- *Soundness:* If some of the share sk_i is incorrect, then SecretKeyVerify outputs "accept" with probability of $\frac{1}{q-1}$.
- *Zero-knowledge:* The simulator on parties P_i would generate their own random α at step 2, sampling commitment $C_1,\dots,C_{t-1}$ uniformly from $\mathbb{Z}_q$, generate a random value for $[\![r]\!]_i$ at step 4 while uniformly sample C_0 at step 6.

D.4 Parallel Polynomial Evaluation for Key Generation

The parallel key generation algorithm consists of two phases:

- The first phase called Preprocessing, where each party at the beginning just store values of $[j^k]_q$ for all possible $1 \le j \le n$ and all $0 \le k \le t$. This preprocessing step can be done once and can be updated if total number of parties changed without having to recompute all existed values.
- The second phase is the actual evaluation phase where the party P_i receive request to evaluate $f_i(\cdot)$ from n parties $\{x_1,\dots,x_n\}$. The party retrieves previously computed values of x_j^k for all $j \in [1,n]$ and forms a matrix. It also form a matrix C from its own secret value $c_{i,0},\dots,c_{i,t-1}$. As the matrix multiplication result in a matrix with column j which has value of $[\sum_k c_{i,k}x_j^k]_q$, the party i then outputs polynomials from the result of the matrix multiplication.

We note that there are two main benefits by doing a parallel evaluation in this manner:

- If the pivot parties are parties with high computational power, they can use GPU for faster key generation as all matrix multiplications can be done in parallel.
- We can concurrently evaluate requests from multiple clients, thus increasing throughput of ThHE.SecretKeyGen.

To prove the correctness of the algorithm presented in Figure 13, we just write down the formula of C and X we formed earlier as follow:

$$C = \begin{bmatrix} c_{i,0}^{(0)} & c_{i,1}^{(0)} & \dots & c_{i,t-1}^{(0)} \\ \dots & \dots & \dots & \\ c_{i,0}^{(t-1)} & c_{i,1}^{(t-1)} & \dots & c_{i,t-1}^{(t-1)} \end{bmatrix} \tag{5}$$

$$X = \begin{bmatrix} x_1^0 & x_2^0 & \dots & x_n^0 \\ \dots & \dots & \dots & \\ x_1^{t-1} & x_2^{t-1} & \dots & x_n^{t-1} \end{bmatrix} \tag{6}$$

$$\text{Hence, } C\cdot X = \begin{bmatrix} \sum_{k=0}^{t-1} c_{i,k}^{(0)} x_1^k & \dots & \sum_{k=0}^{t-1} c_{i,k}^{(0)} x_n^k \\ \dots & \dots & \dots \\ \sum_{k=0}^{t-1} c_{i,k}^{(t-1)} x_1^k & \dots & \sum_{k=0}^{t-1} c_{i,k}^{(t-1)} x_n^k \end{bmatrix}$$

We also have that for all $X \in \mathbb{Z}_q$

$$\begin{aligned} f_i(x_j)(X) &= \sum_{k=0}^{t-1} c_{i,k} x_j^k(X) \\ &= \sum_{k=0}^{t-1}\Big(\sum_{h=0}^{t-1} c_{i,k}^{(h)} X^h\Big) x_j^k \\ &= \sum_{h=0}^{t-1}\Big(\sum_{k=0}^{t-1} c_{i,k}^{(h)} x_j^k\Big) X^h \\ &= \sum_{h=0}^{t-1} (C\cdot X)_{h,j} X^h \end{aligned} \tag{7}$$

Therefore, the representation of $f_i(x_j)$ is the j-th column vector of $C\cdot X$, verifying the correctness of our output.

PARAMETERS:

- A threshold t, parties $P_1,\dots,P_m$ for $m \geq t$, a number $n \in [t,m]$. The B_1-bounded and tB_1-bonded secret spaces χ_1 and χ_1', respectively. The B_2-bounded and tB_2-bonded noise spaces χ_2 and χ_2', the plaintext space $R_p = \mathbb{Z}_p[X]/(X^N+1)$, the ciphertext and public key space $R_q = \mathbb{Z}_q[X]/(X^N+1)$, generator g of $\mathbb{Z}_q$, and $\Delta = \lfloor q/p \rfloor$ for sufficiently large primes p and q.
- A single-key HE, a FuncEval, SecretKeyVerify, and a $\mathsf{ConvMult}_q$ algorithm described in Figures 9, 13, 12, and 10, respectively.

PROTOCOL – $\mathsf{ThHE.SecretKeyGen}(\lambda, \mathsf{TS}(t), \mathcal{P} = \{P_1,\dots,P_n\})$

1. For $i \in [t]$, the pivot party P_i does the followings:
   (a) Choose a random value $c_{i,0} \leftarrow \chi_1$, and $t-1$ random values $c_{i,k} \leftarrow R_q$ for $k \in [1, t-1]$
   (b) Form a polynomial of the degree $(t-1)$ as $f_i(X) = \sum_{k=0}^{(t-1)} c_{i,k} X^k$, public commitments of $c_{i,k}$: $\mathsf{comm}_{i,k} = g^{c_{i,k}}$
   (c) Compute and send $v_{i,j} = \mathsf{FuncEval}(1,\dots,t)[j] = [f_i(j)]_q$ to each party $P_{j\in[n]}$
   (d) Each pivot $P_{i\in[t]}$ outputs a shared secret key $\mathsf{sk}_i = (c_{i,0}, [\sum_{j=1}^{t} v_{j,i}]_q)$
   (e) The server outputs the commitment $C_k = \prod_{i=1}^{t} \mathsf{comm}_{i,k} = g^{\sum_{i=1}^{t} c_{i,k}}$ for all $0 \leq k \leq t-1$
2. For $i \in [t+1, n]$, P_i outputs a shared secret key $\mathsf{sk}_i = [\sum_{j=1}^{t} v_{j,i}]_q$
3. In case of malicious adversary, party $P_{1\leq i\leq n}$ runs $\mathsf{SecretKeyVerify}(\mathsf{sk}_i, C_0, C_1, \dots, C_{t-1})$.

PROTOCOL – $\mathsf{ThHE.ShareRefresh}(\mathcal{P} = \{P_1,\dots,P_n\}, t')$

1. Sample a new set of t' pivot parties, WLOG $\{P_1,\dots,P_{t'}\}$
2. For $i \in [t']$, the new pivot party P_i does the followings:
   (a) Choose $t'-1$ random values $c_{i,k} \leftarrow R_q$ for $k \in [1, t-1]$
   (b) Form a polynomial of the degree $(t'-1)$ as $f_i(X) = \sum_{k=1}^{(t-1)} c_{i,k} X^k$, public commitments of $c_{i,k}$: $\mathsf{comm}_{i,k} = g^{c_{i,k}}$
   (c) Compute and send $v_{i,j} = \mathsf{FuncEval}(1,\dots,t')[j] = [f_i(j)]_q$ to each party $P_{j\in[n]}$
   (d) The server computes the commitment $C_k' = \prod_{i=1}^{t} \mathsf{comm}_{i,k} = g^{\sum_{i=1}^{t'} c_{i,k}}$ for all $1 \leq k \leq t'-1$
   (e) The server updates the commitment $C_k = C_k * C_k'$ for all $1 \leq k \leq t-1$ and $C_k = C_k'$ for $t \leq k \leq t'-1$
3. For $i \in [1, n]$, P_i outputs a shared secret key $\mathsf{sk}_i = \mathsf{sk}_i + [\sum_{j=1}^{t} v_{j,i}]_q$
4. In case of malicious adversary, party $P_{1\leq i\leq n}$ runs $\mathsf{SecretKeyVerify}(\mathsf{sk}_i, C_0, C_1, \dots, C_{t-1})$.

PROTOCOL – $\mathsf{ThHE.Join}(\lambda, \{\mathsf{sk}_i\}_{i\in A}, P_{\nu\in[n+1,m]})$

1. If all t pivots $\{P_1,\dots,P_t\}$ are online (e.g. $A = \{1,\dots,t\}$), they do the following: each $P_{i\in[t]}$ computes and sends $v_{i,\nu} = [f_i(\nu)]_q$ to a new party P_ν who outputs a shared secret key $\mathsf{sk}_\nu = [\sum_{i=1}^{t} v_{i,\nu}]_q$
2. If some subset of $\{P_1,\dots,P_t\}$ dropout, a random t parties $P_{(k\in A)}$ for a set of indices A do the following:
   (a) The party $P_{(k\in A)}$ computes $u_k = \mathsf{sk}_k L_A(\nu,k)$ where $L_A(\nu,k) = \prod_{j\in A\setminus\{k\}} \frac{\nu-k}{j-k}$ is a Lagrange coefficient and sk_k is the Shamir secret share of P_k
   (b) Each party P_k chooses a random value $z_{k,j}$ for $j \in A, j > k$ and then sends $z_{k,j}$ to P_j
   (c) The party $P_{k\in A}$ computes $v_k = u_k + \sum\limits_{j\in A, j<k} z_{j,k} - \sum\limits_{j\in A, j>k} z_{k,j}$ and sends it to P_ν
   (d) P_ν outputs a shared secret key $\mathsf{sk}_\nu = [\sum_{i\in A} v_i]_q$
3. In case of malicious adversary, P_ν runs $\mathsf{SecretKeyVerify}(sk_\nu, C_1, \dots, C_{t-1})$

Figure 11: Our Additive ThHE Construction from Polynomial-LWE (Text highlighting malicious key generation is colored in blue)

PROTOCOL – $\mathsf{ThHE.PublKeyComp}(\lambda, \{\mathsf{sk}_i\}_{i\in A})$

1. If all t pivots $\{P_1,\ldots,P_t\}$ are online (e.g. $A=\{1,\ldots,t\}$), each pivot P_i broadcasts a partial public key $\mathsf{pk}_i = [-a\cdot c_{i,0}+e_i]_q$ for $e_i \leftarrow \chi_2$, and a combiner computes the public key $\mathsf{pk} = \left(\left[\sum_{i=1}^{t}\mathsf{pk}_i\right]_q, a\right)$.
2. If some subset of $\{P_1,\ldots,P_t\}$ dropout, a random t parties $P_{(k\in A)}$ for a set of indices A do the following. Each P_k broadcasts a partial public key $\mathsf{pk}_k = [\mathsf{ConvMult}_q(-a\cdot\mathsf{sk}_k, L_A(0,k))+e_k]_q$ for a random noise $e_k\leftarrow\chi_2$. A combiner computes the public key as $\mathsf{pk} = ([\sum_{k\in A}\mathsf{pk}_k]_q, a)$.
3. In case of malicious adverary, the parties run $\mathsf{BatchSumVerify}(\mathsf{pk}[0], \mathsf{pk}_{i\in A}[0])$ to verify if public key is correctly computed.

PROTOCOL – $\mathsf{ThHE.Encrypt}(\mathsf{pk},\mu)$:

1. Sample $u\leftarrow\chi_1'$ and $e_1,e_2\leftarrow\chi_2'$
2. Compute $c_0=[p_0\cdot u+e_1+\Delta\cdot\mu]_q$ and $c_1=[p_1\cdot u+e_2]_q$ where $p_0=\mathsf{pk}[0]$, $p_1=\mathsf{pk}[1]$
3. Output (c_0,c_1)

PROTOCOL – $\mathsf{ThHE.Add}(\mathsf{ct}_1,\ldots,\mathsf{ct}_n)$: Output $\mathsf{HE.Add}(\mathsf{ct}_1,\ldots,\mathsf{ct}_n)$

PROTOCOL – $\mathsf{ThHE.PartDec}(\mathsf{ct},\mathsf{sk}_i)$:

1. Let $c_0=\mathsf{ct}[0]$ and $c_1=\mathsf{ct}[1]$
2. Output a partial decryption $\mathsf{p}_i=(c_0,[\mathsf{ConvMult}_q(c_1\cdot\mathsf{sk}_i, L_A(0,i))+e_i]_q)$ for a small noise $e_i\leftarrow\chi_2$.
3. If P_i is a pivot, P_i outputs an additional partial decryption $\mathsf{p}_i'=(c_0,[c_1\cdot c_{i,0}+e_i]_q)$

PROTOCOL – $\mathsf{ThHE.FinalDec}(\{\mathsf{p}_i\}_{i\in A})$: Given any set A of t alive parties,

1. If A is a set of t partial decryptions from pivots, outputs $\left[p/q[c_0+\sum_{i=1}^{t}\mathsf{p}_i']_q\right]_p$
2. Otherwise, outputs $\left[p/q[c_0+\sum_{k\in A}\mathsf{p}_k]_q\right]_p$

Figure 11: Our Additive ThHE Construction from Polynomial-LWE (Text highlighting malicious key generation is colored in blue) (cont.)

PARAMETERS: The space $R_q=\mathbb{Z}_q[X]/(X^N+1)$ and a set of online parties A
INPUT: Party P_ν has a share sk_ν that he wants to verify. P_ν also has $C_1,\ldots,C_{t-1}$ where $C_i=g^{c_i}$ being the commitment of party P_i Shamir polynomial
OUTPUT: "accept" if the sk_ν is the correct Shamir share of party P_ν, else "reject".

```
1: procedure SECRETKEYVERIFY($\mathsf{sk}_\nu, C_0,\ldots,C_{t-1}$)
2:     $P_\nu$ check if
           $g^{\mathsf{sk}_\nu[i]} = C_0[i](C_1[i])^{\nu}(C_2[i])^{\nu^2}\ldots(C_{t-1}[i])^{\nu^{t-1}}, \forall i\in[N]$
       and abort if they are not equal
3: end procedure
```

Figure 12: Secret Key Verification.

E ThHE Properties

E.1 Compactness and Complexity

As the encryption and homomorphic computation of ThHE are performed in a similar fashion to single-key HE. Thus, the (evaluated) ciphertext size and the size of the partial decryption result are independent of the number of decryptors/parties. Assume that the ciphertext size of the underlying single-key HE only depends on the security parameter λ, so is the ThHE ciphertext size. For completeness, we present the complexity of each algorithm in Table 8.

E.2 Correctness

The ThHE's correctness follows from the correctness of the underlying threshold scheme and single-key HE schemes.

Definition 3. *(Correctness) We say that a ThHE scheme is correct if for all security parameter λ, a threshold scheme TS with a threshold t, the k messages $\mu_i\in\{0,1\}^\star$ for $i\in[k]$, sets $\mathcal{P}=\{P_1,\ldots,P_m\}$ and $\mathcal{P}'=\{P_1,\ldots,P_n\}$ for $m\geq n\geq t$. For $(pk, sk_1,\ldots,sk_n)\leftarrow ThHE.KeyGen(\lambda, TS(t),\mathcal{P}')$; for $\nu\in[n+1,m]$, $sk_\nu\leftarrow ThHE.Join(\lambda,\{sk_i\}_{i\in A'},P_\nu)$ where A' is an index set of t shared secret keys; $ct_i = ThHE.Encrypt(pk,\mu_i), \forall i\in$*

	Computation			Communication		
	Pivot	Non-Pivot	Combiner	Pivot	Non-Pivot	Combiner
ThHE.SecretKeyGen	$O(n\log(t))$	$O(\log(t))$	–	$O(nN\log(q))$	$O(tN\log(q))$	–
ThHE.PublKeyComp (1)	$O(N\log(N))$	–	$O(tN)$	$O(N\log(q))$	–	$O(tN\log(q))$
ThHE.PublKeyComp (2)	–	$O(N\log(N)+t)$	$O(tN)$	–	$O(N\log(q))$	$O(tN\log(q))$
ThHE.Join (1)	$O(\log(t))$	$O(\log(t))$	–	$O(N\log(q))$	$O(tN\log(q))$	–
ThHE.Join (2)	–	$O(t^2)$	–	–	$O(tN\log(q))$	–
ThHE.Encrypt	$O(N\log(N))$	$O(N\log(N))$	–	–	–	–
ThHE.Add	$O(N)$	$O(N)$	–	–	–	–
ThHE.PartDec	$O(N\log(N))$	$O(N\log(N))$	–	$O(N\log(q))$	$O(N\log(q))$	–
ThHE.FinalDec	–	–	$O(tN)$	–	–	$O(tN\log(q))$

Table 8: **The Complexity of Our ThHE Construction**. We have three types of parties: pivot, non-pivot, and combiner. The combiner can be any party including pivot, non-pivot, or a third party such as a server in our FL application. The ThHE key generation and decryption algorithms rely on Shamir secret sharing which contains the polynomial operations. Using fast Fourier transform (FFT) [19, 54], the two t-degree polynomial multiplication has computational complexity of $O(t\log t)$, and n-point evaluation has the complexity $O(n\log t)$. (1): t pivots are online. (2): A subset of pivots are offline but at least t parties are online. n,t represents the number of parties, the threshold for our ThHE. The ciphertext space is $R_q = \mathbb{Z}_q[X]/(X^N+1)$. Cells with $-$ denote computation/communication that is not valid by the protocol.

$[k]$; and $\mathsf{ct} = \mathsf{ThHE.Add}(\mathsf{ct}_1,\dots,\mathsf{ct}_k)$, we have:

$$\mathsf{Prob}[\mathsf{ThHE.FinalDec}(D) = \mu_1+\ldots+\mu_k)] = 1-\mathsf{negl}(\lambda)$$

where $D = \{\mathsf{ThHE.PartDec}(\mathsf{ct},\mathsf{sk}_i)) \mid i\in A\}$ and A is an index set of t shared secret keys.

Theorem 2. *(Correctness) Suppose TS is a threshold scheme that satisfy correctness. Then, the ThHE construction described in Fig. 11 satisfies correctness defined in Def. 3.*

Proof. We prove the following:

- The ThHE.KeyGen and ThHE.Join algorithms gives the valid share of the secret key $\mathsf{sk} = \sum_{i=1}^{t} c_{i,0}$ to each party in $\mathcal{P}$ who joins the computation at the beginning (the share generated by ThHE.SecretKeyGen) or in the middle (the share generated by ThHE.Join) of the process. Additionally, given any t valid shares, the algorithm correctly computes the public key pk which has the same formula as the public key of the single-key HE scheme described in Figure 9.
- If the underlying (conventional) HE scheme satisfies correctness with the noise bound, the ThHE.PartDec algorithm returns the correct plaintext p given the same noise bound, and the encryption ct of the p which is computed by either ThHE.Encrypt or ThHE.Add.

Fix the security parameter λ, threshold scheme TS with a threshold t, the k messages $\mu_i \in \{0,1\}^\star$ for $i\in[k]$, sets $\mathcal{P} = \{P_1,\dots,P_m\}$ and $\mathcal{P}' = \{P_1,\dots,P_n\}$ for $m\ge n\ge t$, the following holds. For $(\mathsf{pk},\mathsf{sk}_1,\dots,\mathsf{sk}_n) \leftarrow \mathsf{ThHE.KeyGen}(\lambda,\mathsf{TS}(t),\mathcal{P}')$; for $\nu\in[n+1,m], \mathsf{sk}_\nu \leftarrow \mathsf{ThHE.Join}(\lambda,\{\mathsf{sk}_i\}_{i\in A'},P_\nu)$ where A' is an index set of t shared secret keys; $\mathsf{ct}_i = \mathsf{ThHE.Encrypt}(\mathsf{pk},\mu_i), \forall i\in[k]$; and $\mathsf{ct} = \mathsf{ThHE.Add}(\mathsf{ct}_1,\dots,\mathsf{ct}_k)$. We must show that given $D = \{\mathsf{ThHE.PartDec}(\mathsf{ct},\mathsf{sk}_i)) \mid i\in A\}$ where A is an index set of t shared secret keys, $\mathsf{Prob}[\mathsf{ThHE.FinalDec}(\mathsf{pk},\mathsf{ct},D) = \mu_1+\ldots+\mu_k)] = 1-\mathsf{negl}(\lambda)$. To this end, we show the followings.

(1) The ThHE.KeyGen generates the valid share of the secret key $\mathsf{sk} = \sum_{i=1}^{t} c_{i,0}$ to each party in $\mathcal{P}'$.
Let consider the t-degree polynomial $f(X) = \sum_{i=1}^{t} f_i(X)$, where the polynomial $f_i(X) = \sum_{k=0}^{(t-1)} c_{i,k}X^k$ has been chosen by the pivot party $P_{i\in[t]}$. As each $c_{i,0}$ is sampled from the B_1-bounded distribution χ_1, the value of $f(0)$ is bounded by tB_1. We first show that each party $P_{i\in[n]}$ receives the valid S3 share $[f(i)]_q$ from ThHE.KeyGen. Subsequently, we prove that any t shares can reconstruct the jointed secret key $\mathsf{sk} = \sum_{i=1}^{t} c_{i,0}$.
Recall that the pivot party's key comprises of two components, namely $\mathsf{sk}_i = (c_{i,0}, [\sum_{j=1}^{t} v_{j,i}]_q)$. For the sake of simplicity, we shall disregard the first component of the key. According to ThHE.SecretKeyGen, it is easy to see that $\mathsf{sk}_i = [f(i)]_q$ holds for all $i\in[m]$, given that $\mathsf{sk}_i = [\sum_{j=1}^{t} v_{j,i}]_q = [\sum_{j=1}^{t}[f_j(i)]_q]_q$.

(2) The ThHE.KeyGen algorithm correctly computes the public key pk.
Given t partial public keys $\mathsf{pk}_i = [-a\cdot c_{i,0}+e_i]_q$ from t pivots, the first term of the public key is computed correctly as $\mathsf{pk} = \left[\sum_{i=1}^{t}\mathsf{pk}_i\right]_q = \left[\sum_{i=1}^{t} -a\cdot c_{i,0}+e_i\right]_q = [-a\cdot s+e]_q$ for the noise $e = \sum_{i=1}^{t} e_i$. Note that the secret key $\mathsf{sk} = \sum_{i=1}^{t} c_{i,0}$ and the noise $e = \sum_{i=1}^{t} e_i$, where all $c_{i,0}$ and e_i are sampled from χ_1 and χ_2, respectively, thus, the sk and e are sampled from the distribution χ_1' and χ_2', respectively. In a general case when there is a set A of t alive parties, each $P_{k\in A}$ publishes its partial public key $\mathsf{pk}_k = [\mathsf{ConvMult}_q(-a\cdot\mathsf{sk}_k, L_A(0,k))+e_k]_q$. This allows a combiner to correctly compute the joint public key which has the same formula as the public key of the single-key HE

```
PARAMETERS: The space $R_q = \mathbb{Z}_q[X]/(X^N+1)$, total number of parties $n$, threshold $t$, number of points to evaluate $\rho$
INPUT: $c_{i,0},..,c_{i,t-1} \in R_q$; $x_1,\dots,x_\rho \in [n]$
OUTPUT: $f_i(x_j) = [\sum_{k=0}^{t-1} c_{i,k}x_j^k]_q$, $\forall j \in [\rho]$

1: procedure PREPROCESSING($\rho$)
2:     for $j = 1,\dots,\rho$ do
3:         Compute $z_j = j^k \mod q$
4:         store a vector $X_j = \{z_0, z_1, \dots, x^{t-1}\}$ for later usage
5:     end for
6: end procedure
7: procedure FUNCEVAL($x_1,\dots,x_\rho$)
8:     for $k = 1,\dots,\rho$ do
9:         Get $X_{x_k}$ = Preprocessing($\rho$)[$x_k$]
10:    end for
11:    Form a matrix $X$ with $k$-th column equals $X_{x_k}$
12:    for $j = 0,\dots,t-1$ do
13:        Form a vector $C_j = \{c_{i,j}^{(0)},\dots,c_{i,j}^{(t-1)}\}$ where the elements satisfies: $c_{i,j}$: $c_{i,j} = \sum_{k=0}^{t-1} c_{i,j}^{(k)} X^k$
14:    end for
15:    Form a matrix $C$ with $k$-th column equals $C_k$
16:    Compute the matrix multiplication: $Y = [C \cdot X]_q$
17:    Form $n$ polynomials from $Y$: $y_k(x) = \sum_{j=1}^{t} Y_{j,k} x^j$
18:    return $y_1,\dots,y_n$
19: end procedure
```

Figure 13: Parallel Polynomial Evaluation

scheme described in Figure 9. It due to the fact that

$$\begin{aligned}\mathsf{pk} &= \big[\sum_{k\in A}\mathsf{pk}_k\big]_q\\ &= \big[\sum_{k\in A}[\mathsf{ConvMult}_q(-a\cdot\mathsf{sk}_k, L_A(0,k)) + e_k]_q\big]_q\\ &= \big[\sum_{k\in A}[-a\cdot\mathsf{sk}_k L_A(0,k) + e_k]_q\big]_q\\ &= \big[-a[\sum_{k\in A}\mathsf{sk}_k L_A(0,k)]_q + [\sum_{k\in A} e_k]_q\big]_q\\ &= [-a\cdot s + \sum_{k\in A} e_k]_q\end{aligned}$$

(3) The ThHE.Join give the valid share of the secret key $\mathsf{sk} = \sum_{i=1}^{t} c_{i,0}$ to each new party.
In ThHE.Join, which generates a new secret key sk_ν for a new user $P_{\nu\in[m+1,n]}$, the computation of sk_ν is identical to that in ThHE.SecretKeyGen, provided all t pivots are online. Thus, we have $\mathsf{sk}_\nu = [f(\nu)]_q$. For a general case scenario where a set A of any t parties is available, the computation[1] of the sk_ν is given by $\mathsf{sk}_\nu = [\sum_{k\in A}\mathsf{sk}_k L_A(\nu,k)]_q$. Here, $L_A(\nu,k) = \prod_{j\in A\setminus\{k\}} \frac{\nu-k}{j-k}$ is a Lagrange coefficient. Using Lagrange linear combination, the sk_ν is a valid Shamir share of sk since we have $\mathsf{sk}_\nu = [\sum_{k\in A}[f(k)]_q L_A(\nu,k)]_q = [f(\nu)]_q$.
Given a set A of any t valid Shamir secret shares, one can reconstruct sk by computing $\sum_{k\in A} sk_k L_A(0,k) = \sum_{k\in A}[f(k)]_q L_A(0,k) = [f(0)]_q = \mathsf{sk}$.

[1]For simplicity, we disregard the zero share, as it finally cancels out

(4) Assuming that the underlying (conventional) homomorphic encryption scheme satisfies correctness with a noise bound of tB_2, the ThHE.PartDec algorithm will return the correct plaintext p when given the same noise bound tB_2 and the encryption ct of p that was computed by either ThHE.Encrypt or ThHE.Add algorithm.
Recall that the constant coefficient sk of $f(X)$ is bounded by tB_1, which means that the secret key sk belongs to the distribution χ_1'. Additionally, the noise term e in the public key pk is equal to $[\sum_{k\in A} e_k]_q$, where each local noise e_k is sampled from the B_2-bounded distribution χ_2, and therefore, the e also belongs to the tB_2-bounded distribution χ_2'.
The ThHE.Add calls the additive evaluation algorithm of the single-key HE. Hence, when using the underlying single-key HE with the noise space χ_2' to prove that $\mathsf{Prob}[\mathsf{ThHE.FinalDec}(D) = C(\mu_1,\dots,\mu_k)] = 1 - \mathsf{negl}(\lambda)$ where $D = \{\mathsf{ThHE.PartDec}(\mathsf{ct},\mathsf{sk}_i)) \mid i\in A\}$, it is sufficient to prove that $\mathsf{Prob}[\mathsf{ThHE.FinalDec}(D) = \mu] = 1 - \mathsf{negl}(\lambda)$ where ct is the encryption of $\mu = C(\mu_1,\dots,\mu_k)$.
In ThHE.Encrypt, the ciphertext $\mathsf{ct} = (c_0, c_1)$ is computed as follows: $c_0 = [(-a\cdot s + e)\cdot u + e_1' + \Delta\cdot\mu]_q$; $c_1 = [a\cdot u + e_2']_q$; $u \leftarrow \chi_1'$; $e_1', e_2' \leftarrow \chi_2'$; and $\Delta = \lfloor q/p \rfloor$. When the set A consists of the t pivot parties, they perform the partial decryption $\mathsf{p}_i' = [c_1\cdot c_{i,0} + e_i]_q$ and then compute $\big[p/q[c_0 + \sum_{i=1}^{t}\mathsf{p}_i']_q\big]_p$. We have:

$$\begin{aligned}[c_0 + \sum_{i=1}^{t}\mathsf{p}_i']_q &= [c_0 + \sum_{i=1}^{t}[c_1\cdot c_{i,0} + e_i]_q]_q\\ &= [c_0 + c_1\cdot s + \sum_{i=1}^{t} e_i]_q\\ &= \big[[(-a\cdot s + e)\cdot u + e_1' + \Delta\cdot\mu]_q\\ &\quad + [a\cdot u + e_2']_q\cdot s + \sum_{i=1}^{t} e_i\big]_q\\ &= [\Delta\cdot\mu + e\cdot u + e_1' + e_2'\cdot s + e']_q\end{aligned}$$

where $e' = \sum_{i=1}^{t} e_i \in \chi_2'$. Since all the variables e, u, e_1', e_2', e', s are from the bounded distribution, the $\big[p/q[c_0 + \sum_{i=1}^{t}\mathsf{p}_i']_q\big]_p$ gives the desired plaintext μ.
In a general case when there is a set A of t alive parties, the final decryption of the ciphertext $\mathsf{ct} = (c_0, c_1)$ is given by $\big[p/q[c_0 + \sum_{k\in A}\mathsf{p}_k]_q\big]_p$ where $\mathsf{p}_k = [\mathsf{ConvMult}_q(c_1 \cdot$

$\mathsf{sk}_k, L_A(0,k)) + e_k]_q$. We have:

$$\begin{aligned}[c_0 + \sum_{k\in A} \mathsf{p}_k]_q &= \big[c_0 + \sum_{k\in A}[c_1 \cdot \mathsf{sk}_k L_A(0,k) + e_k]_q\big]_q \\ &= \big[c_0 + c_1 \cdot \sum_{k\in A}\big(\mathsf{sk}_k L_A(0,k)\big) + \sum_{k\in A} e_k]_q\big]_q \\ &= [c_0 + c_1 \cdot s + \sum_{i=1}^{t} e_k]_q\end{aligned}$$

Similar to the above case, the final decryption $\big[p/q[c_0 + \sum_{k\in A}\mathsf{p}_k]_q\big]_p$ gives the correct plaintext μ.

□

E.3 Security

The security definition of ThHE is stated in accordance with the threshold security of [17] which includes the semantic/simulation security in Definition 4 and Definition 5. The challenge with these definitions is how to accurately reflect the dynamics of the setting. For simplicity, we assume that the adversary controls the malicious parties only after the join or refresh share functions are called.

We state the semantic security and simulation of our ThHE in Theorem 3 and Theorem 4, respectively in Appendix. At the high-level idea, the security of our ThHE relies on the PLWE assumption, the security of threshold scheme (e.g., Shamir secret sharing) and the single-key HE schemes. First, we observe that all the broadcast messages in the ThHE construction have a form $(a, [a \cdot x + e]_q)$ for $a \leftarrow R_q$ and $e \leftarrow \chi_2$. Based on the PLWE problem, these messages reveal nothing about the underlying secret x. Second, we ensure that any set of $t-1$ parties cannot reconstruct the original secret because of the Shamir secret sharing scheme. Thus, no one can learn the secret key as well as other parties' input unless t parties collude. Finally, the encryption and evaluation algorithms are computed in a similar way to single-key HE. Therefore, if the underlying single-key HE is secure, so are these two algorithms of ThHE. For achieving malicious security, we can refer directly to the discussion in Section 5 and Appendix D.3.

Definition 4. *(Semantic Security) We say that a ThHE scheme satisfies semantic security if for all security parameter λ, the following holds. For any PPT adversary Adv, the following experiment $\mathsf{Expt}^{sem}_{\mathsf{Adv},ThHE}(1^\lambda)$ outputs 1 with negligible probability.*

$\mathsf{Expt}^{sem}_{\mathsf{Adv},ThHE}(1^\lambda)$:

- *On input the security parameter λ, the adversary Adv outputs a threshold scheme TS with a threshold t.*
- *The challenger runs $(\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n) \leftarrow \mathsf{ThHE.KeyGen}(\lambda, \mathsf{TS}(t), \mathcal{P}')$, $\mathsf{sk}_\nu \leftarrow \mathsf{ThHE.Join}(\lambda, \{\mathsf{sk}_i\}_{i\in A'}, P_\nu)$, for $\nu \in [n+1, m]$, and $\{\mathsf{sk}_1, \ldots, \mathsf{sk}_n\} \leftarrow \mathsf{ThHE.ShareRefresh}(\mathcal{P} = \{P_1, \ldots, P_n\}, t')$, where $\mathcal{P}' = \{P_1, \ldots, P_n\}$ and A' is an index set of t shared secret keys. The challenger provides pk to Adv.*
- *Adv outputs an invalid set $V \in \{P_1, \ldots, P_m\}$ for such that $|V| < t$. Note that this occurs after $\mathsf{ThHE.Join}$ to ensure the security of the dynamic model. For $\mathsf{ThHE.ShareRefresh}$, we can consider the security with respect to the new threshold t' to be similar to that of the original threshold t.*
- *The challenger provides $\{\mathsf{sk}_i \mid i \in V\}$ along with the $\mathsf{ct} = \mathsf{ThHE.Encrypt}(\mathsf{pk}, \mu)$ for $\mu \leftarrow \{0,1\}^\star$ to Adv.*
- *The challenger provides the ciphertext $\mathsf{ct} = \mathsf{ThHE.Encrypt}(\mathsf{pk}, \mu)$ for $\mu \leftarrow \{0,1\}^\star$ to Adv.*
- *Adv outputs a guess μ'. The experiment outputs 1 if $\mu = \mu'$.*

Definition 5. *(Simulation Security) We say that a ThHE scheme satisfies simulation security if for all security parameter λ, the following holds. There exists a stateful PPT algorithm $T = (T_1, T_2, T_3, T_4)$ such that for any PPT adversary Adv, the following experiments $\mathsf{Expt}_{\mathsf{Adv},ThHE,\mathsf{REAL}}(1^\lambda)$ and $\mathsf{Expt}_{\mathsf{Adv},ThHE,\mathsf{IDEAL}}(1^\lambda)$ are indistinguishable:*

$\mathsf{Expt}_{\mathsf{Adv},ThHE,\mathsf{REAL}}(1^\lambda)$:

- *On input the security parameter λ, the adversary Adv outputs a threshold scheme TS with a threshold t.*
- *The challenger runs the key generation $(\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n) \leftarrow \mathsf{ThHE.KeyGen}(\lambda, \mathsf{TS}(t), \mathcal{P}')$, the join algorithm $\mathsf{sk}_\nu \leftarrow \mathsf{ThHE.Join}(\lambda, \{\mathsf{sk}_i\}_{i\in A'}, P_\nu)$, for $\nu \in [n+1, m]$, and $\{\mathsf{sk}_1, \ldots, \mathsf{sk}_n\} \leftarrow \mathsf{ThHE.ShareRefresh}(\mathcal{P} = \{P_1, \ldots, P_n\}, t')$ where $\mathcal{P}' = \{P_1, \ldots, P_n\}$ and A' is an index set of t shared secret keys. The challenger provides the pk to Adv.*
- *Adv outputs an invalid set $V \in \{P_1, \ldots, P_m\}$ such that $|V| < t$, and k messages $\mu_{j\in[k]} \leftarrow \{0,1\}^\star$. Note that this occurs after $\mathsf{ThHE.Join}$ to ensure the security of the dynamic model. For $\mathsf{ThHE.ShareRefresh}$, we can consider the security with respect to the new threshold t' to be similar to that of the original threshold t.*
- *The challenger provides $\{\mathsf{sk}_i \mid i \in V\}$ and the ciphertexts $\mathsf{ct}_{j\in[k]} = \mathsf{ThHE.Encrypt}(\mathsf{pk}, \mu_j)$ to Adv.*
- *Adv issues a polynomial number of queries for the form $(S \subset \{P_1, \ldots, P_m\}, C \subset \{1, \ldots, m\})$. For each query, the challenger computes $\hat{\mathsf{ct}} = \mathsf{ThHE.Add}(\{\mathsf{ct}_i \mid i \in C\})$, and provides the partial decryption $\mathsf{p}_i = \mathsf{ThHE.PartDec}(\hat{\mathsf{ct}}, \mathsf{sk}_i), \forall i \in S$ to Adv.*
- *Adv outputs a distinguishing bit b*

$\mathsf{Expt}_{\mathsf{Adv},ThHE,\mathsf{IDEAL}}(1^\lambda)$:

- *On input the security parameter λ, the adversary Adv outputs a threshold scheme TS with a threshold t.*
- *The challenger runs $(\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n) \leftarrow T_1(\lambda, d, \mathsf{TS}(t), \mathcal{P}')$ and $\mathsf{sk}_\nu \leftarrow T_2(\lambda, \{\mathsf{sk}_i\}_{i\in A'}, P_\nu)$, for $\nu \in [n+1, m]$ where $\mathcal{P}' = \{P_1, \ldots, P_n\}$ and A' is an index set of t shared secret keys. The challenger provides the pk to Adv.*
- *Adv outputs an invalid set $V \in \{P_1, \ldots, P_m\}$ such that $|V| < t$, and k messages $\mu_{j\in[k]} \leftarrow \{0,1\}^\star$,*
- *The challenger provides $\{\mathsf{sk}_i \mid i \in V\}$ and the ciphertexts $\mathsf{ct}_{j\in[k]} = \mathsf{ThHE.Encrypt}(\mathsf{pk}, \mu_j)$ to Adv.*
- *Adv issues a polynomial number of queries for the form $(S \subset \{P_1, \ldots, P_m\}, C \subset \{1, \ldots, m\})$. For each query, the*

challenger computes $\hat{\mathsf{ct}} = T_3(\{\mathsf{ct}_i \mid i \in C\})$, and provides the partial decryption $\mathsf{p}_i = T_4(\hat{\mathsf{ct}}, \mathsf{sk}_i), \forall i \in S$ to Adv.

- *Adv outputs a distinguishing bit b*

Intuitively, the security definitions say that given an invalid set V of parties (i.e., $|V| < t$), (1) the adversary should not be able to learn anything about μ given the encryption of a message μ chosen by the challenger; (2) when the adversary Adv gives the ciphertexts ct_j of the chosen messages μ_j, requests to perform computations C on the ciphertexts, and their partial decryptions, the adversary should not learn any information about the shared secret key of other honest parties or the secret key sk from the partial decryptions. The Adv executes the final decryption, but learns nothing except the $\sum_{i \in C} \mu_i$.

Theorem 3. *(Semantic Security) Suppose HE is an additive homomorphic encryption scheme and TS is a threshold scheme that satisfy security. Then, the ThHE construction described in Figure 11 also satisfies semantic security defined in Definition 4 under the PLWE assumption.*

Proof. The semantic security of our ThHE follows from the semantic security of the underlying FHE and the property of threshold scheme in a straightforward way.

The encryption algorithm of our ThHE scheme follows the same encryption of the single-key HE where the noise is sample from χ_2'. In the $\mathsf{ThHE.PublKeyComp}$, the noise term e of the public key pk is obtained by summing up randomly sampled noise terms e_i from a B_2-bounded distribution χ_2. As the χ_2 (such as the Gaussian distribution) has an additive property, the resulting e can be considered to be sampled from a tB_1-bounded distribution χ_1'. This means that both encryption and key generation sample noises from the same distribution χ_2', which is identical with the encryption procedure of the single-key HE.

The adversary Adv obtains access to shared keys from the invalid set V. Thanks to the security guarantee of the threshold scheme, Adv gains no knowledge about the secret key. With no knowledge of the secret key, the security of the single-key HE encryption (as is the case with our ThHE) ensures that no information about the plaintext is disclosed during encryption without knowing the secret key. Consequently, the adversary's ability to correctly guess the value $\mu' = \mu$ is negligible.

□

Theorem 4. *(Simulation Security) Suppose HE is an additive homomorphic encryption scheme and TS is a threshold scheme that satisfy security. Then, the ThHE construction described in Figure 11 also satisfies simulation security defined in Definition 5 under the PLWE assumption.*

Proof. To prove the theorem, we simulate that the real-world and ideal-world executions are indistinguishable. In our ThHE, the $T = (T_1, T_2, T_3, T_4)$ implements the ideal functionality of ($\mathsf{ThHE.KeyGen}$
, $\mathsf{ThHE.Join}$, $\mathsf{ThHE.Add}$, $\mathsf{ThHE.PartDec}$). Specifically,

- $T_1(\lambda, \mathsf{TS}(t), \mathcal{P}')$: On input the security parameter λ, a TS with the threshold t, and $\mathcal{P}'$, the algorithm outputs $(\mathsf{pk}, \mathsf{sk}_1, \dots, \mathsf{sk}_n)$ where $(\mathsf{sk}_1, \dots, \mathsf{sk}_n)$ is generated from a threshold scheme $\mathsf{TS}(t)$ in which any t values sk_i can construct a secret value sk. The pk has a same formula as the public key described in Figure 9.
- $T_2(\lambda, \{\mathsf{sk}_i\}_{i \in A'}, P_v)$: On input λ, an index set A' of t shared secret keys, the algorithm outputs sk_v which is a valid share of the $\mathsf{TS}(t)$.
- $T_3(\mathcal{P} = \{P_1, \dots, P_n\}, t')$: On an parties set $P_1, \dots, P_n$ and new threshold t', the algorithm outputs $\mathsf{sk}_{i \in [n]}$ which is a valid share of the $\mathsf{TS}(t')$.
- $T_4(\{\mathsf{ct}_i \mid i \in C\})$: On input a set $\{\mathsf{ct}_i \mid i \in C\}$, the algorithm outputs the ciphertext $\hat{\mathsf{ct}}$, which is encryption of the the sum on $\{\mu_i \mid i \in C\}$, where μ_i is the plaintext of ct_i.
- $T_5(\hat{\mathsf{ct}}, \mathsf{sk}_i)$: On input $\hat{\mathsf{ct}}$, and sk_i, the algorithm outputs p_i such that any t values p_i can compute $\hat{\mu}$ which is the plaintext of ct.

We now prove the theorem using a sequence of hybrid experiments between a challenger and an adversary Adv. For the correctness of the algorithm T, we refer the reader to Theorem 2.

Hybrid 0: This is $\mathsf{Expt}_{\mathsf{Adv}, ThHE, \mathsf{REAL}}(1^\lambda)$ – the ThHE real security experiment, where all parties run the ThHE scheme honestly.

Hybrid 1: The same as the real interaction (Hybrid 0), except that the challenger now samples $(\mathsf{pk}, \mathsf{sk}_1, \dots, \mathsf{sk}_n)$ using T_1. Specifically, instead of computing $(\mathsf{sk}_1, \dots, \mathsf{sk}_n) \leftarrow$ $\mathsf{ThHE.SecretKeyGen}$
$(\lambda, \mathsf{TS}(t), \mathcal{P}')$ and $\mathsf{pk} \leftarrow \mathsf{ThHE.KeyGen}(\lambda, \{\mathsf{sk}_i\}_{i \in A})$ for a subset $A \subset \mathcal{P}', |A| = t$, the challenger now samples these values $(\mathsf{pk}, \mathsf{sk}_1, \dots, \mathsf{sk}_n) \leftarrow T_1(\lambda, \mathsf{TS}(t), \mathcal{P}')$. The rest of the experiment remains unchanged.

In the $\mathsf{ThHE.SecretKeyGen}$, each corrupt party $P_{j \in V}$ receives $v_{i,j} = [f_i(j)]_q$ from the honest party P_i. Relying on the security property of the threshold scheme, the adversary Adv which controls the invalid set V of the size less than the threshold t should have no different view from the shared keys generated by T_1. Moreover, when V consists of only pivots parties, Adv sees no difference between the set of ideal shares and the set of real shares of secret key due to the fact that the remaining shares of honest pivot parties are unknown values in χ_1 domain to Adv and the sk is additively shared to t pivots, which of them is honest. Therefore, the adversary view of two worlds from $\mathsf{ThHE.SecretKeyGen}$ is identical.

To simulate the $\mathsf{ThHE.PublKeyComp}$ which computes the public key pk, we consider two following cases based on the method to compute the keys:

- All t pivots are online: In this case, the pivot publishes $\mathsf{pk}_i = [-a \cdot c_{i,0} + e_i]_q$ where $c_{i,0} \leftarrow \chi_1$ and $e_i \leftarrow \chi_2$. According to the PLWE assumption, the pk_i reveals nothing about $c_{i,0}$ to the adversary.
- Some subset of pivots $\{P_1, \dots, P_t\}$ dropout, a random t parties $P_{(k \in A)}$ for a set of indices A is chosen for invoking

ThHE.PublKeyComp computation: In this case, the honest P_k broadcasts a partial public key $\mathsf{pk}_k = [\mathsf{ConvMult}_q(-a \cdot \mathsf{sk}_k, L_A(0,k)) + e_k]_q$ for a random noise $e_k \leftarrow \chi_2$. We can present pk_k as $[-(aL_A(0,k)) \cdot \mathsf{sk}_k + e_k]_q$ where both a and $L_A(0,k)$ are the public values; and the values $\mathsf{sk}_k \in R_q, e_k \in \chi_2$. Note that $[aL_A(0,k)]_q$ are uniformly sampled in R_q due to the fact that a is uniformly sampled in R_q and $\gcd(L_A(0,k),q) = 1$ for any $k \in Z_q, k \neq 0$. According to the PLWE assumption, a pair $\big([aL_A(0,k)]_q,$ $[-(aL_A(0,k)) \cdot \mathsf{sk}_k + e_k]_q\big)$ hides sk_k. Therefore, pk_k reveals nothing to the adversary.

In summary, all the messages that were sent/received during the ThHE.PublKeyComp procedures are either under a form of $(a, [a \cdot x + e]_q)$, which can be replaced with random ones. Hence, Hybrid 0 and Hybrid 1 are indistinguishable.

Hybrid 2: The same as Hybrid 1, except that the challenger now uses T_2 to compute sk_v for the new P_v. Specifically, instead of computing $\mathsf{sk}_v \leftarrow \mathsf{ThHE.Join}(\lambda, \{\mathsf{sk}_i\}_{i \in A}, P_v)$ for a subset $A \subset \mathcal{P}', |A| = t$, the challenger now samples the $\mathsf{sk}_v \leftarrow T_2(\lambda, \{\mathsf{sk}_i\}_{i \in A'}, P_v)$. The rest of the experiment remains unchanged.

In the ThHE.Join when the new key sk_v is computed by all t pivot parties (Step 1), the simulation is simple as it is similar to the case of the ThHE.SecretKeyGen. Consider a scenarios when a random t parties $P_{(k \in A)}$ computes sk_v, the adversary receives v_k from the honest party P_k. The v_k equals to $v_k = u_k + z_k$ where the value $z_k = \sum\limits_{j \in A, j<k} z_{j,k} - \sum\limits_{j \in A, j>k} z_{k,j}$ is considered as the share of zero, which was distributed by t parties (i.e., z_k looks random to the adversary as $|V| < t$). Thus, the corrupt parties cannot unmask the $v_k = u_k + z_k$ to learn u_k as well as the secret key sk_k. Therefore, the ThHE.Join computation is secure against up to t colluding parties. In other words, ThHE.Join implements the ideal functionality of T_1 securely. Thus, Hybrid 1 and Hybrid 2 are indistinguishable.

Hybrid 3: The same as Hybrid 2, except that the challenger now uses T_3 to compute $\mathsf{sk}_{i \in [n]}$ for the new threshold t.

Generating the t'-out-of-n Shamir secret shares of zero follows the concept of the ThHE.SecretKeyGen protocol, where the secret key is initialized to zero. This approach is secure, allowing us to replace the transcript with random values. Additionally, we leverage the additive property of Shamir shares, ensuring that the secret key (sk) is neither reconstructed nor revealed during the computation. Therefore, Hybrid 1 and Hybrid 2 are indistinguishable.

Hybrid 4: The same as Hybrid 3, except that the challenger now computes $\hat{\mathsf{ct}}, \{\mathsf{p}_i\}_{i \in S}$ from T_3, T_4, respectively. Specifically, instead of computing the $\hat{\mathsf{ct}} = \mathsf{ThHE.Add}(\{\mathsf{ct}_i \mid i \in C\})$ and $\mathsf{p}_i \leftarrow \{\mathsf{ThHE.PartDec}(\hat{\mathsf{ct}}, \mathsf{sk}_i)\}_{i \in S}$, the challenger now samples the $\hat{\mathsf{ct}} \leftarrow T_3(\{\mathsf{ct}_i \mid i \in C\})$ and $\mathsf{p}_i \leftarrow T_4(\hat{\mathsf{ct}}, \mathsf{sk}_i)$. The rest of the experiment remains unchanged.

Note that, the μ_j were chosen by the adversary. In addition, the $\mathsf{ThHE.Add}(\{\mathsf{ct}_i \mid i \in C\})$ calls the subroutine $\mathsf{HE.Add}(\{\mathsf{ct}_i \mid i \in C\})$ of the single-key HE. Thus, the simulation for $\hat{\mathsf{ct}}$ and $\{\mathsf{ct}_j\}_{j \in [k]}$ is elementary. The only information sent to the adversary is the set of values p_i.

Recall that the partial decryption p_i has a form $\mathsf{p}_i = [c_1 \cdot c_{i,0} + e_i]_q$ or $\mathsf{p}_i = [\mathsf{ConvMult}_q(c_1 \cdot \mathsf{sk}_i, L_A(0,i)) + e_i]_q = [c_1 L_A(0,i) \cdot \mathsf{sk}_i + e_i]_q$ where $e_i \leftarrow \chi_2$ is a small noise. According to the PLWE assumption, a pair $(c_1, [c_1 \cdot c_{i,0} + e_i]_q)$ or $([c_1 L_A(0,i)]_q, [c_1 L_A(0,i) \cdot c_{i,0} + e_i]_q)$ protects information about $c_{i,0}$ or sk_i from the adversary Adv. Therefore, the Adv gains no information about the secret key of the honest party, regardless of the chosen C or plaintexts μ_j. The Adv might perform the final decryption to obtain $\sum_{i \in C} \mu_i$). However, this provides the same information that Adv also receives in the ideal world. Therefore, the Hybrid 2 and Hybrid 3 are indistinguishable.

$\square$