TACITA: Threshold Aggregation without Client Interaction

Varun Madathil*, Arthur Lazzaretti, Zeyu Liu, and Charalampos Papamanthou

Yale University

firstname.lastname@yale.edu

Abstract

Secure aggregation enables a central server to compute the sum of client inputs without learning any individual input, even in the presence of dropouts or partial participation. This primitive is fundamental to privacy-preserving applications such as federated learning, where clients collaboratively train models without revealing raw data.

We present a new secure aggregation protocol, TACITA, in the single-server setting that satisfies four critical properties simultaneously: (1) *one-shot communication* from clients with no per-instance setup, (2) *input-soundness*, i.e. the server cannot manipulate the ciphertexts, (3) *constant-size communication* per client, independent of the number of participants per-instance, and (4) *robustness* to client dropouts

Previous works on secure aggregation – Willow and OPA (CRYPTO'25) – achieve one-shot communication but differ in their treatment of *input soundness*. Willow does not provide input soundness, while OPA achieves it under the assumption of a public-key infrastructure (PKI) via signatures. In the absence of input soundness, secure aggregation protocols cannot guarantee full privacy. In contrast, we achieve full privacy at the cost of assuming a PKI. Specifically, TACITA relies on a novel cryptographic primitive we introduce and realize: *succinct multi-key linearly homomorphic threshold signatures (MKLHTS)*, which enables verifiable aggregation of client-signed inputs with constant-size signatures. To encrypt client inputs, we adapt the Silent Threshold Encryption (STE) scheme of Garg et al. (CRYPTO 2024) to support ciphertext-specific decryption and additive homomorphism.

We formally prove security in the Universal Composability framework and demonstrate practicality through an open-source proof-of-concept implementation, showing our protocol achieves scalability without sacrificing efficiency or requiring new trust assumptions.

1 Introduction

Consider a set of N clients, each holding a private input x_i. Out of the N clients, t can be malicious. The goal of a secure aggregation protocol is to enable all clients (both honest and malicious) to learn the sum $\sum x_i$ without learning anything about the inputs of other honest clients. This problem can be straightforwardly solved by applying maliciously-secure generic secure multiparty computation (MPC) protocols[Yao82, GMW87] treating the aggregation as computing the function $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$. A particularly relevant variant of the secure aggregation problem includes a dedicated, potentially malicious server, who is the only one learning the final aggregated output. Typically, the

*Corresponding author: varun.madathil@yale.edu

server itself has no input and simply facilitates the aggregation. This problem too can be solved by treating the aggregation as computing the function

$$f(\perp, x_1, \ldots, x_N) = \left(\sum_{i=1}^{N} x_i, \perp, \ldots, \perp \right)$$

This variant arises in scenarios such as federated learning, where a central server aggregates model updates from multiple distributed clients without compromising their individual data privacy. We will elaborate on this application in detail later.

However, solving secure aggregation using generic MPC protocols typically requires multiple rounds of interaction, often involving direct communication between the clients themselves. Such interactive protocols pose practical challenges in many real-world scenarios, where clients may have intermittent connectivity or limited availability – thus becoming prone to dropouts. To address these challenges, we aim for a more efficient approach: a *one-shot protocol*, where each client interacts only once by sending a single message directly to the server and does not need to remain online afterward. In doing so, we essentially tackle the original MPC problem under stricter constraints, significantly enhancing practical efficiency and robustness.

However, realizing non-interactive or one-shot secure aggregation presents several technical challenges, which we outline below.

Challenge 1: Residual Leakage Non-interactive MPC protocols suffer from a fundamental privacy limitation known as *residual leakage*, first formalized by Halevi et al. [HLP11]. In the absence of interaction, a malicious server can compute the aggregate over arbitrary subsets of received inputs, thereby extracting information about individual client values.

For example, suppose that the server computes $S_1 = x_1 + x_2 + x_3$ and later computes $S_2 = x_1' + x_2 + x_3$ by substituting its own input x_1' in place of x_1. It can then recover x_1 as $S_1 - S_2 + x_1'$. Such residual attacks allow an adversary to learn partial or even complete information about individual inputs, violating the core privacy goal of secure aggregation.

To address this issue, the communication model must be enhanced with mechanisms that prevent the server from selectively aggregating arbitrary subsets. A common approach is to involve an external *committee* to validate or assist in the aggregation process, thereby enforcing threshold constraints and curbing such leakage.

Committee-Based Solutions: Existing Approaches. In this committee-based model, each client encrypts its input under a homomorphic scheme, and the server can only recover the aggregate if the committee jointly performs threshold decryption. This prevents the server from selectively re-aggregating subsets of inputs, since each decryption is bound to a single session.

Two recent protocols at CRYPTO'25 – Willow by Bell-Clark et al. [BCGL+24] and OPA by Karthikeyan et al. [KP24] – employ such committees in one-shot aggregation settings.

At a high-level, clients in Willow encrypt a key-homomorphic encryption (KAHE) key under the public key of the committee, and encrypt their inputs using the KAHE key. The server aggregates the ciphertexts encrypting the keys and sends it to the committee who return partial decryptions of this ciphertext. This allows the server to compute an aggregated key that allows decryption of the aggregated ciphertext. Similarly, in OPA, the clients secret share a seed-homomorphic PRG seed and mask their input with the output of PRG on this seed. They send an encrypted share to each commmittee member via the server. The committee members aggregate the seed shares and allows the server to compute one final aggregated seed. The server then removes the masks after computing the PRG on the aggregated seed.

The crucial insight in both these works is that the server cannot decrypt the aggregate unless the committee participates. If a majority of the committee is honest and only participates *once* per aggregation session, the server is prevented from selectively re-running the aggregation on overlapping subsets of inputs, thus eliminating residual leakage. TACITAis described in the same setting. We present a detailed comparison with OPA and Willow in Section 2.4.

Table 1: Asymptotic computation and communication complexity of one-shot secure aggregation protocols with a committee. N: number of clients participating in one iteration; M: number of committee members. Each cell shows the computation cost and communication cost for that entity, respectively, separated by a '/'. A '-' means no computation or no communication for that metric. Willow and OPA consider inputs vectors of length ℓ, in this table we compare asymptotics with $\ell = 1$. We present a more detailed comparison with Willow and OPA in Section 2.4.

Scheme	**Offline Phase**			**Online Phase**		
	Client	**Server**	**Committee**	**Client**	**Server**	**Committee**
Willow [BCGL+24]	$-/-$	$-/-$	$\mathcal{O}(M)/\mathcal{O}(M)$ (DKG)	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(N+M)/\mathcal{O}(N)$	$\mathcal{O}(\log N)/\mathcal{O}(1)$
OPA [KP24]	$-/-$	$-/-$	$\mathcal{O}(1)/-$	$\mathcal{O}(M)/\mathcal{O}(M)$	$\mathcal{O}(N+M\log M)/\mathcal{O}(N)$	$\mathcal{O}(\log N)/\mathcal{O}(\log N)$
This Work	$\mathcal{O}(N+M)/-$	$-/-$	$\mathcal{O}(N)/-$	$\mathcal{O}(1)/\mathcal{O}(1)$	$\mathcal{O}(N+M)/\mathcal{O}(1)$	$\mathcal{O}(1)/\mathcal{O}(1)$

Note: Computation costs are per party, and communication cost of server is the size of message sent to each committee member. While we require the clients and committee to do computation in the offline phase, their computation and communication are constant-sized in the offline phase. Moreover, we highlight that there is no communication required between committee members in the offline phase, whereas the committee in Willow must run an interactive DKG.

Challenge 2: Input Soundness. A second major challenge in the committee-based, one-shot model is ensuring *input soundness*: the committee must be convinced that a sufficient number of valid client inputs were correctly included in the aggregation, and that the server has not tampered with the ciphertexts. Without such guarantees, a malicious server could attempt to recover individual client contributions. We show that Willow is vulnerable to an attack in which the server manipulates ciphertexts to learn the input of a target client. At a high level, the server discards ciphertexts from all but one target client, fabricates ciphertexts on behalf of the other clients, and forwards this manipulated set to the committee. The committee, acting honestly, produces partial decryptions that allow the server to recover the aggregated sum. By subtracting the values it injected, the server isolates and learns the target client's input. We describe this attack in detail in Section 4.

Naïve Mitigation. Assume the committee know which clients provide inputs, later we describe how this can be realized, and add robustness in case a few clients drop out. A natural countermeasure then is to assume a PKI and require each client to sign its ciphertext. The server would then forward both ciphertexts and signatures to the committee, which could verify that a threshold number of valid client inputs are present. However, this approach introduces significant communication overhead, as the server must transmit each client's ciphertext and signature to every committee member.

A Weaker Mitigation. In applications where PKI is not possible, as in Willow[BCGL+24], the clients may add noise to their inputs, or alternatively have the committee add some noise to the aggregate. For the latter one still needs to assume a PKI for the committee (see Appendix I of Willow [BCGL+24]). However, one can only hope to achieve a notion of differential privacy in this case, which affects accuracy and still does not provide full privacy to the inputs.

Our Mitigation. We show how to achieve full client privacy via input soundness without incurring the communication overhead of naïvely using signatures. Our approach instead shifts the cost into local offline computation performed by both clients and committee members. Importantly, this computation requires no additional interaction among clients. This is unlike Willow, which relies on running a distributed key generation (DKG) among the committee in every iteration. Table 1 compares the asymptotic complexity of our approach with Willow and OPA, assuming their protocols are augmented with a PKI to enforce input soundness.

1.1 Our Contributions

In this work, we present the first secure aggregation protocol - TACITA, within the one-shot, committee-based model that simultaneously satisfies the following desirable properties:

- **Constant-size Communication complexity**: All client-to-server and server-to-committee interactions have message sizes independent of both the number of clients and committee members

unlike any of the previous works (augmented for input soundness). See Table 1 for overview.

- **Input Soundness**: We show that Willow does not provide input soundness and suffers from a man-in-the-middle attack allowing a malicious server to learn the private inputs of honest parties. We then present a protocol that does not suffer from such an attack and also does not affect the communication complexity via a new cryptographic primitive called Multi-key Linearly Homomorphic Threshold Signatures.
- **No external verifiers or fixed committees**: Unlike Willow [BCGL+24], our protocol eliminates the need for additional trusted verifier parties, and does not require a fixed committee. In our work, committee members may be randomly selected clients as in OPA [KP24] and Flamingo [MWA+23].
- **Security in the Universal Composability (UC) Framework**: Our protocol ensures secure aggregation as long as a threshold number of clients submit their inputs, making it inherently robust to intermittent connectivity or client dropout. We prove the security of TACITAin the UC model.

To achieve these goals, we introduce a novel cryptographic primitive: *Succinct Multi-Key Linearly Homomorphic Threshold Signatures (MKLHTS)*. This primitive enables the aggregation of signatures over different messages signed under different keys (i.e., $\sigma_1 = \mathsf{Sign}(\mathsf{sk}_1, m_1), \ldots, \sigma_n = \mathsf{Sign}(\mathsf{sk}_n, m_n)$) into a single succinct signature σ^* on the sum of the messages $m^* = \sum m_i$, verifiable using an aggregated verification key $\mathsf{vk}^* = \sum \mathsf{vk}_i$. Specifically, the aggregate verifies as $\mathsf{Verify}(\mathsf{vk}^*, m^*, \sigma^*) = 1$ if and only if the number of aggregated message-signature pairs exceeds a threshold.

Crucially, MKLHTS ensures *input soundness* without requiring the committee to receive individual signatures or ciphertexts from each client. In contrast to prior approaches such as OPA [KP24], where each committee member must receive a message per client, our scheme requires the committee to process only a single aggregate signature, regardless of the number of clients. This is the core reason our protocol achieves communication complexity that is *constant* in the number of clients while maintaining strong correctness guarantees.

The server aggregates client-submitted signatures and ciphertexts and sends a single succinct message to the committee. The committee verifies the MKLHTS signature and proceeds with threshold decryption only if the signature validates – thereby enforcing input soundness in a lightweight, one-shot manner.

To facilitate threshold decryption, we adapt the *Silent Threshold Encryption (STE)* scheme of Garg et al. [GKPW24]. Our modified STE supports ciphertext-specific partial decryptions and additive homomorphism, allowing the server to compute an encrypted aggregate that can be decrypted only when a threshold number of one-shot committee members provide their shares. This ensures that the committee need not remain online or interact multiple times, and that the protocol remains robust to committee member dropout. Independently and concurrently, Bormet et al. [BCF+25] proposed the same ciphertext-bound, additively homomorphic extension to Silent Threshold Encryption. Our work is independent and was developed in parallel.

Practical Motivation: Federated Learning. The constraint of non-interactivity is not merely theoretical – it arises naturally in practical settings such as *federated learning*, the motivating application also considered in prior work [BIK+17, BBG+20, BCGL+24, KP24]. In such systems, mobile devices periodically transmit locally trained model updates to a central server, which aggregates them using functions such as FEDAVG [MMR+17]. Because mobile devices are often intermittently connected or energy-constrained, protocols must be both *non-interactive* and *one-shot*: clients must be able to submit their updates and then go offline without further participation.

Federated learning further operates in *iterations*, where each round involves a dynamically chosen subset of clients. Since the total client population may number in the millions, it is infeasible to include all devices in a single round. Moreover, clients selected for participation in any given round

cannot be expected to engage in per-instance setup or coordination. This makes *setup-free, one-shot protocols* particularly attractive in practice.

As shown in Table 1, clients in TACITAincur some local computation in an offline phase. In the federated learning setting, this can naturally correspond to preparatory work performed hours before the aggregation – after the dynamic client set for the iteration has been selected. The online phase then requires only a single message from each client of constant size, maintaining the one-shot property while ensuring scalability.

Threat Model. We consider a malicious server that can collude with up to a threshold number of clients and committee members in our secure aggregation protocol. We prove security in the Universal Composability (UC) model. Our protocol achieves privacy as long as the malicious server corrupts only a minority of the clients and committee.

Assumptions. Our protocol design relies on two practical assumptions:

1. **Client subset selection algorithm**: We assume the subset of clients participating in each aggregation round is determined by an externally defined global selection algorithm. Such algorithms are common and well-studied in federated learning contexts; viable secure instantiations already exist, for example, in Flamingo [MWA+23]. The same approach is used in Willow [BCGL+24] and OPA [KP24]. We assume that each client can compute for itself the set of other clients and committee members selected in each iteration of the aggregation protocol.

   Moreover, we assume a Public Key Infrastructure (PKI) and that the clients and committee can retrieve keys via some key distribution mechanism. Recent works on key transparency [LCG+23, LCG+24, FDR+25] can be used to give guarantees that the clients retrieve the correct keys. We remark that without a PKI one-shot secure aggregation schemes may be susceptible to man-in-middle attacks outlined in Section 4.

2. **Small-domain client inputs**: For the majority of the exposition in this work, we assume client inputs belong to a small, finite domain. While this restricts the protocol from directly handling arbitrary large-domain data, it aligns naturally with federated learning scenarios where model updates are typically quantized or otherwise low-entropy [MMR+17, AGL+17]. More specifically, we are required to compute the discrete log in $\mathbb{G}_T$. For typical FL quantization ($q \approx 3329$), precomputed tables (≈ 53 MB) yield constant-time lookup. We note, however, that in Section 8 we show techniques on how to extend the protocol to arbitrary inputs.

3. **Trusted Common Reference String (CRS)**: Our protocols make use of pairings and KZG-style proofs for proving aggregation was done correctly. To this end, our protocols require a trusted CRS. This assumption was not required in previous works like OPA and Willow.

2 Technical Overview

In this section, we provide a technical overview of our protocol - TACITAand the underlying ideas. We begin with a high-level blueprint of our secure aggregation protocol. We then present a simplified warm-up protocol where we assume all parties are honest, which helps illustrate how our cryptographic tools compose. Finally, we discuss how to strengthen the protocol against a malicious server and committee dropout, extend it to vector-valued inputs, and outline the additional considerations required to achieve UC security.

Protocol Blueprint. At a high level, our protocol proceeds in four stages:

1. **Client encryption and signing.** Each client encrypts its private input using the public key of the committee, instantiated via the Silent Threshold Encryption scheme. The client then signs the ciphertext using its secret signing key under the MKLHTS scheme. The encrypted and signed input is sent to the server.

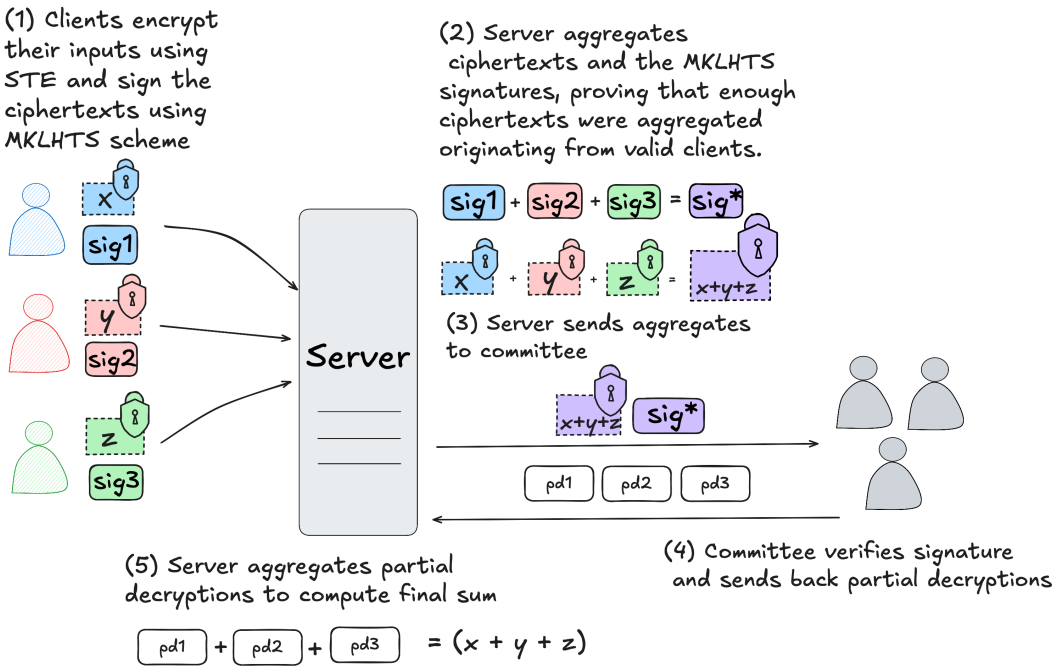

Figure 1: An overview of TACITA. Clients send encrypted and signed inputs to the server, which aggregates and forwards them to the committee. The committee verifies threshold correctness and performs threshold decryption.

2. **Server aggregation.** The server aggregates the received ciphertexts and signatures into a single ciphertext (encrypting the sum of client inputs) and a single aggregate signature. Both are constant in size. The server forwards this pair to the committee.

3. **Committee verification and partial decryption.** The committee verifies that the aggregated signature corresponds to a valid threshold number of client signers. If verification succeeds, each committee member returns a partial decryption share.

4. **Final decryption.** The server combines the partial decryption shares to recover the plaintext sum of the client inputs.

A visual representation of the full protocol is shown in Figure 1.

2.1 A Warm-up Protocol

We begin by describing a simplified version of our protocol in an honest setting, where the server behaves correctly, and neither clients nor committee members drop out. In this setting, we do not require threshold signatures (MKLHTS), since the server can be trusted to compute the aggregation honestly.

The Setting. Assume each client holds a public-private key pair for an additively homomorphic encryption scheme. Let $\mathcal{C}$ denote the set of clients providing inputs, and let Committee denote the committee responsible for decryption.

Each client in $\mathcal{C}$ encrypts its input and sends the ciphertext to the server. A natural question arises: under what key should clients encrypt their inputs? Since the Committee is selected dynamically in each instance of the secure aggregation protocol, it is impractical to expect a traditional distributed key generation (DKG) phase with each new round. Instead, we rely on the *silent threshold* framework

Witness Encryption Scheme:

- $\mathsf{WEnc}((\mathsf{pk}, tag), m)$:
  - Sample $r \leftarrow \mathbb{Z}_p$, compute $ct_1 = r \cdot [1]_1$
  - Compute $ct_2 = r \cdot (\mathsf{pk} \circ H(tag)) + m$
  - Output ciphertext (ct_1, ct_2)
- $\mathsf{WDec}(\sigma, (ct_1, ct_2))$:
  - Output $m = ct_2 - (ct_1 \circ \sigma)$, where $\sigma = \mathsf{sk} \cdot H(tag)$

Figure 2: Witness Encryption from BLS Signatures [DHMW23, MTV+23]

of Garg et al. [GJM+24, GKPW24], which enables threshold decryption by dynamically selected subsets, without requiring per-instance setup.

Witness Encryption for Signatures. The foundation of silent threshold encryption is an elegant adaptation of witness encryption for signatures [DHMW23, MTV+23]. Here, encryption is performed with respect to a public key and a tag (e.g., a session identifier), and decryption is done using signatures on the tag. We describe this core primitive in Figure 2.

Common Notation We assume $m \in \mathbb{G}_T$, and a hash function $H : \{0,1\}^\lambda \to \mathbb{G}_2$. We adopt additive notation throughout, where $+$ denotes group addition, $\cdot$ denotes scalar multiplication, and $\circ$ denotes the bilinear pairing.

Additive Homomorphism. The encryption scheme described in Figure 2 is *additively homomorphic*. Specifically:

$$\mathsf{WEnc}((\mathsf{pk}, tag), m_A) + \mathsf{WEnc}((\mathsf{pk}, tag), m_B) = \mathsf{WEnc}((\mathsf{pk}, tag), m_A + m_B)$$

To see this, let the ciphertexts be:

$$ct_{A,1} = r_A [1]_1 \qquad ct_{A,2} = r_A \cdot (\mathsf{pk} \circ H(tag)) + m_A$$
$$ct_{B,1} = r_B [1]_1 \qquad ct_{B,2} = r_B \cdot (\mathsf{pk} \circ H(tag)) + m_B$$

Summing:

$$ct_1 = r_A + r_B [1]_1$$
$$ct_2 = r_A + r_B \cdot (\mathsf{pk} \circ H(tag)) + m_A + m_B$$

This yields a valid encryption of $m_A + m_B$ under aggregated randomness $r_A + r_B$. Moreover, the ciphertext can be decrypted using the signature $\sigma = \mathsf{sk} \cdot H(tag)$.

Key Aggregation. The scheme also supports *key aggregation*: encryption under a combined key $\mathsf{pk}^* = \mathsf{pk}_A + \mathsf{pk}_B$ can be decrypted using the sum of individual signatures.

Let

$$(ct_1, ct_2) = \mathsf{WEnc}((\mathsf{pk}^*, tag), m)$$

That is,

$$ct_1 = r[1]_1 \quad ct_2 = r \cdot (\mathsf{pk} \circ H(tag)) + m$$

and let each client hold a signature $\sigma_i = \mathsf{sk}_i \cdot H(tag)$. Define the aggregated signature as:

$$\sigma^* = \sigma_A + \sigma_B = (\mathsf{sk}_A + \mathsf{sk}_B) \cdot H(tag) = \mathsf{sk}^* \cdot H(tag)$$

Then decryption yields:

$$\begin{aligned} m &= ct_2 - (ct_1 \circ \sigma^*) \\ &= r \cdot (\mathsf{pk}^* \circ H(tag)) + m - (r[1] \circ \mathsf{sk}^* \cdot H(tag)) \\ &= r \cdot (\mathsf{pk}^* \circ H(tag)) + m - r(\mathsf{sk}^*[1] \circ H(tag)) \\ &= r \cdot (\mathsf{pk}^* \circ H(tag)) + m - r \cdot (\mathsf{pk}^* \circ H(tag)) = m \end{aligned}$$

This follows from the bilinearity of pairings. Hence, multiple clients can encrypt under an aggregated public key, and decryption succeeds using aggregated signatures.

The Warm-up Protocol. Leveraging the homomorphic and key aggregation properties of the encryption scheme described above, the protocol proceeds as follows.

Each client in $\mathcal{C}$ computes the aggregate public key of the Committee as:

$$\mathsf{pk}_{\mathsf{Committee}} = \sum_{j \in \mathsf{Committee}} \mathsf{pk}_j$$

The client then encrypts its input x_i using $\mathsf{WEnc}((\mathsf{pk}_{\mathsf{Committee}}, tag), x_i)$ and sends the resulting ciphertext to the server. The server aggregates all received ciphertexts homomorphically to obtain a ciphertext corresponding to the sum $\sum_{i \in \mathcal{C}} x_i$.

To decrypt, each committee member $j \in \mathsf{Committee}$ returns a signature on the tag, i.e., $\sigma_j = \mathsf{sk}_j \cdot H(tag)$. The server aggregates these signatures to obtain:

$$\sigma^* = \sum_{j \in \mathsf{Committee}} \sigma_j = \left(\sum_{j \in \mathsf{Committee}} \mathsf{sk}_j \right) \cdot H(tag)$$

It then uses $\mathsf{WDec}(\sigma^*, \cdot)$ to decrypt the aggregated ciphertext and recover the plaintext sum.

This warm-up protocol demonstrates the core functionality of TACITAin an honest setting. In the remainder of this section, we extend this protocol to handle malicious servers, enforce input thresholds via MKLHTS, and tolerate dropout among clients and committee members.

2.2 Handling a Corrupt Server

We now modify the warm-up protocol to ensure security even in the presence of a malicious server and client dropouts.

Limitations of the Warm-up Protocol. We first note that even in the honest-but-curious setting, the previous protocol is insecure. The problem stems from the fact that any ciphertext encrypted under the key $(\mathsf{pk}_{\mathsf{Committee}}, tag)$ can be decrypted by the server using the partial decryptions sent by the committee. Since each individual client ciphertext is encrypted under $(\mathsf{pk}_{\mathsf{Committee}}, tag)$, the server can apply the decryption procedure to recover individual inputs thus violating privacy!

Enforcing Ciphertext Binding. To prevent this, we must ensure that partial decryptions returned by the committee can only be used to decrypt exactly one ciphertext.

We first observe the identity:

$$ct_1 \circ \sigma = ct_1 \circ (\mathsf{sk} \cdot H(tag)) = \mathsf{sk} \cdot ct_1 \circ H(tag)$$

This motivates modifying the partial decryption shares to be:

$$\mathsf{part\text{-}dec}_j = \mathsf{sk}_j \cdot ct_1$$

instead of $\sigma_j = \mathsf{sk}_j \cdot H(tag)$. With this binding, the decryption of the aggregated ciphertext becomes:

$$\sum_{i \in \mathcal{C}} x_i = ct_2 - \left(\sum_{j \in \mathsf{Committee}} \mathsf{part\text{-}dec}_j \circ H(tag) \right)$$

This enforces that partial decryptions are tied to the specific aggregate ciphertext and cannot be reused to decrypt individual inputs.

Garg et al. [GKPW24] present a *silent threshold encryption* scheme that builds on the same core idea discussed above, with one crucial enhancement: decryption succeeds as long as a threshold number of partial decryptions are provided. We observe that their scheme preserves the additive homomorphism property essential for secure aggregation, and furthermore, it can be modified to support ciphertext binding in the manner described earlier. This allows us to instantiate a robust, threshold-based decryption mechanism without requiring any per-instance setup or explicit coordination between committee members. Finally, we remark that a very recent work concurrent to ours, by Bormet et al [BCF+25] appeared on eprint that presents the exact same construction of homomorphic STE as in our work.

Enforcing Input Soundness. We now address the challenge of *input soundness*: how can the committee verify that the ciphertext sent by the server is truly aggregated from client ciphertexts? For example, the server could drop honest ciphertexts, or worse, fabricate ciphertexts on behalf of honest clients!

Prior work Willow [BCGL+24] suffers from this issue, enabling the server to recover the input of a target honest party. We sketch the Willow attack below and then explain our mitigation without sacrificing communication efficiency.

An attack on Willow In Willow, clients encrypt their inputs using a Key Additive Homomorphic Encryption (KAHE) scheme, and encrypt the key under the committee's public key via a verifiable additively homomorphic encryption scheme. The server aggregates ciphertexts and sends them to a verifier, who checks correctness and that enough ciphertexts are included. The verifier then issues a certificate, which the server forwards to the committee to obtain partial decryptions, eventually recovering the aggregated KAHE key and decrypting the aggregate.

Crucially, clients do not sign their inputs, so the server can fabricate ciphertexts. Concretely, the server:

1. Discards all but the target client's ciphertexts.
2. Samples random keys and encrypts them honestly under the verifiable AHE scheme.
3. Aggregates ciphertexts and proofs, sends them to the verifier, and obtains a valid certificate.
4. Forwards the certified aggregate to the committee, obtains partial decryptions, and reconstructs the aggregated key.
5. Subtracts its chosen random keys to isolate the target client's key and decrypts its input.

Mitigation assuming a PKI: One simple defense is to require clients to sign their ciphertexts. The server forwards these signatures, and the verifier (or committee in OPA) checks validity before aggregation. While secure, this incurs $\mathcal{O}(N)$ communication, where N is the number of clients.

Multi-Key Linearly Homomorphic Threshold Signatures (MKLHTS). To ensure input soundness without sacrificing efficiency, we introduce a new primitive: *Multi-Key Linearly Homomorphic Threshold Signatures (MKLHTS)*. Our construction builds on Multi-Key Linearly Homomorphic Signatures (MKLHS) [FMNP16, LTWC18, AP19, ABF24], which allow signatures on individual messages under different keys to be aggregated into a single signature on some function of the messages. We extend previous works to the threshold setting, and improve on the size of the overall signature (from $\mathcal{O}(N)$ to $\mathcal{O}(1)$).

In our setting, we only consider the summation function. That is, given:

$$\sigma_1 = \mathsf{Sign}(\mathsf{sk}_1, m_1), \quad \sigma_2 = \mathsf{Sign}(\mathsf{sk}_2, m_2),$$

one can compute an aggregate signature σ^* on $m_1 + m_2$ that verifies under the combined verification keys $\mathsf{vk}_1, \mathsf{vk}_2$.

In contrast to existing MKLHS schemes, our construction has two additional properties: (1) it enforces a threshold requirement: the aggregate signature is valid only if it combines signatures from at least t distinct keys, and (2) the signature scheme is fully succinct - the size of the signature is independent of the number of parties.

We achieve this by extending techniques from Aranha et al. [AP19] and Garg et al.'s threshold signatures framework [GJM+24]. However, a key limitation of the scheme by Aranha et al. [AP19] is that the size of the aggregated signature grows linearly with the number of participating signers, making it unsuitable for large-scale applications. In contrast, our construction achieves fully *succinct* aggregation: the final signature remains constant in size, regardless of the number of clients. On the other hand, the scheme by Garg et al. [GJM+24] assumes that all parties sign the same message, which does not hold in our setting. As such, we cannot apply their protocol directly and require new techniques to support aggregation over distinct messages.

Similar to the work of Aranha et al [AP19], each client i computes a signature over its message m_i and a global tag:

$$\sigma_i = \mathsf{sk}_i \cdot (H(tag) + [m_i]).$$

This resembles a BLS-style signature, but introduces a dependency on the message m_i and tag jointly. We prove that this scheme is unforgeable.

The server aggregates the ciphertexts and signatures as:

$$\sigma^* = \sum_i \sigma_i = \left(\sum_i \mathsf{sk}_i \cdot H(tag)\right) + \left(\sum_i \mathsf{sk}_i \cdot [m_i]\right).$$

However, verifying σ^* is non-trivial. A naive check of the form

$$e(\sigma^*, [1]) \stackrel{?}{=} e(H(tag) + \sum_i [m_i], \mathsf{aVK})$$

fails, since in general

$$\sum_i \mathsf{sk}_i \cdot m_i \neq \left(\sum_i \mathsf{sk}_i\right) \cdot \left(\sum_i m_i\right).$$

We resolve this via a generalization of the sumcheck technique from hinTs [GJM+24]. We present details of this new MKLHTS primitive in Section 6.

Putting It All Together. With these components in place – (1) Silent Threshold Encryption for robust, non-interactive encryption and threshold decryption, and (2) MKLHTS for succinct enforcement of input soundness – our full protocol proceeds exactly as described in the blueprint. Each client encrypts and signs their input in a one-shot manner; the server aggregates ciphertexts and signatures; and the committee verifies and decrypts, ensuring both correctness and privacy – even in the presence of malicious behavior and client dropouts.

One subtlety is that while our exposition often refers to messages as group elements in $\mathbb{G}_T$ for clarity, client inputs are in fact elements of $\mathbb{Z}_p$. Thus, after decryption, the server must recover the plaintext by computing a discrete logarithm in $\mathbb{G}_T$ (see section 8.1 for a discussion on how this discrete algorithm can be efficiently computed). This necessitates that inputs lie within a small domain, a design choice justified by many federated learning settings. In Section 8, we discuss how to remove this constraint.

2.3 UC Security and Extensions

UC Security. We prove that our protocol securely realizes an ideal functionality for secure aggregation in the Universal Composability (UC) framework [Can01]. A brief overview is given in Appendix A.1.

A key challenge in proving UC security is that the (potentially malicious) server must compute the *correct sum* of honest client inputs, while the simulator cannot learn honest clients' inputs. To resolve this, we slightly modify the protocol and assume access to a programmable random oracle $\mathcal{G}_{\mathsf{RO}}$ (see Figure 8).

Each client masks its input with $y_i = \mathcal{G}_{\mathsf{RO}}(r_i)$ and encrypts the masked input using our STE scheme and r_i using Garg's STE [GKPW24] (denoted $\overline{\mathsf{STE}}$). With help from the committee, the server decrypts the sum of masked inputs and the individual r_i values (via $\overline{\mathsf{STE}}$), enabling computation of the correct sum. Recall from earlier that to decrypt ciphertexts encrypted under $\overline{\mathsf{STE}}$ the committee only need to provide a BLS signature on the corresponding $\overline{tag}$.

Finally, the simulator programs the random oracle so the real-world output matches the ideal functionality without learning honest inputs. A full UC-style simulation and proof appear in Appendix D.

Prior works also rely on random oracles. In Willow [BCGL+24], the committee must release a fresh decryption key each iteration, requiring a new DKG to be run by the committee in every iteration of the protocol. In OPA [KP24], the committee instead sends $\mathcal{O}(N)$ aggregated shares of the mask, avoiding per-iteration DKG but incurring large communication. Our use of $\overline{\mathsf{STE}}$ achieves the best of both: no per-instance DKG and no $\mathcal{O}(N)$ blowup.

Extension: Vector Inputs. In federated learning, clients submit vectors of model updates rather than scalars. Running the protocol per element would incur linear overhead in the vector length ℓ. To avoid this, we adopt the *Key-Additive Homomorphic Encryption (KAHE)* technique used in Willow [BCGL+24] and OPA [KP24]. Each client encrypts its vector with a KAHE scheme parameterized by a seed ksk_i, and encrypts ksk_i with our threshold encryption. The server aggregates both the vectors and encrypted seeds. Once the committee returns $\sum_i \mathsf{ksk}_i$, the server can decrypt the aggregate vector. Since only the sum of seeds is revealed, individual seeds remain hidden, preserving privacy. We instantiate the KAHE via the Ring Learning with Error assumption as Willow [BCGL+24]. Section 8 provides the full protocol.

2.4 Detailed comparison with Willow and OPA

Here we describe a detailed comparison with OPA [KP24] and Willow [BCGL+24]. Table 1 compares the asymptotic computation and communication complexity of our protocol with OPA and Willow.

Input Soundness. As discussed above and detailed in Section 4, Willowdoes not provide input soundness because the messages sent from clients are not signed. Specifically, in Willow, the server can generate ciphertexts of KAHE keys as if they originated from the clients and send them to the verifier. The verifier then issues a valid certificate, ensuring that the decryptors return valid partial decryptions to the server. This enables the server to learn the input of a specific target client by omitting the keys it selected for non-target clients. With the target client's key, the server can directly compute that client's input.

To prevent this attack, we require that each client's ciphertext be signed. This blocks the attack, since the server cannot replace ciphertexts without forging signatures. Furthermore, in TACITA, we ensure that adding signatures does not increase communication complexity by employing the new primitive we introduce: *Multi-Key Linearly Homomorphic Threshold Signatures*.

Alternatively, one can achieve differential privacy guarantees as suggested by Willow, though this is weaker than full privacy that we achieve.

Distributed Key Generation (DKG) - Willow. In Willow and OPA, as in TACITA, clients must en-

crypt their inputs to the committee or decryptor(s). In Willow, there is a single public key corresponding to the decryptor(s), whereas in OPA, clients encrypt shares separately for each individual decryptor. In Willow, the decryptors are required to run a DKG protocol to jointly generate and announce a public key, under which clients then encrypt their inputs. Because DKG requires interaction among the decryptors, it is infeasible and costly for clients themselves to act as decryptors. In contrast, OPA avoids this issue – its design allows some clients to take on the role of decryptors directly.

In TACITA, we combine the advantages of both approaches: we maintain a single public key, as in Willow, but without requiring distributed key generation. We achieve this using a modified ciphertext-bound, homomorphic variant of the Silent Threshold Encryption scheme by Garg et al. [GKPW24].

Communication and Computation Complexity. Compared to Willow [BCGL+24] and OPA [KP24] our protocols perform much better in terms of communication complexity and have some trade-offs in computation complexity. Below we describe a comparison when the input from the clients are of length 1. Let M be the number of committee members, and N be the number of clients sending inputs.

We consider two phases, an online phase and an offline set up phase for each iteration of the secure aggregation. In the federated learning setting, the offline phase can be thought of as the instant when the clients are notified that they need to participate in an iteration of updates. In TACITA, all operations are done *locally* in the offline setup phase.

Communication Complexity: In the offline phase, as described above, the committee in Willow need to run an interactive DKG before every iteration. In OPA and our work the clients and committee do not need to communicate at all.

In the online phase, in Willow, each client sends only a constant number of ciphertexts, while in OPA clients must send $\mathcal{O}(M)$ ciphertexts – one share per committee member. Our work matches Willow: clients send constant-sized messages to the server. In Willow, the server aggregates and broadcasts a single ciphertext to the committee and incurs $\mathcal{O}(N)$ communication to the verifier. However, if they were to provide input soundness, they will need $\mathcal{O}(N)$ communication to the committee. In OPA the server merely forwards all client ciphertexts. However, in OPA, they cleverly reduce the complexity by sending $\mathcal{O}(\log N)$ ciphertexts to each committee member by increasing the number of committee members. The main idea is to sample $N/\log N$ committee members and assign $\log N$ number of clients to each committee member. Finally, committee members always reply with constant-sized messages – partial decryptions in Willow and our work. In OPA they send aggregated shares that requires $\mathcal{O}(\log N)$ shares per committee member.

Computation Complexity: In the offline phase as mentioned above the committee members of Willow incur $\mathcal{O}(M)$ computation at least, which is the cost of running DKG. In our work the clients need to do local computation to compute hints and keys, this amounts to $\mathcal{O}(M+N)$ work for the clients and $\mathcal{O}(N)$ work for the committee. OPA on the other hand requires no computation in the offline phase.

In the online phase, for clients, Willow and our work requires constant work, while OPA requires $\mathcal{O}(M)$ to compute shares. In the aggregation phase, Willow and our scheme both incur $\mathcal{O}(N)$ server work to combine ciphertexts and signatures, while OPA avoids computation since the server only forwards the ciphertexts to the committee. Committee members do constant work in Willow and TACITA(computing partial decryptions), but $\mathcal{O}(N)$ in OPA to aggregate shares. Finally, the decryption phase costs $\mathcal{O}(M)$ in Willow and our work (aggregating partial decryptions) versus $\mathcal{O}(N + M\log M)$ in OPA (aggregating shares and producing the final output).

3 Preliminaries

Notations. Throughout this paper we use λ as the security parameter, and $\mathsf{negl}(\lambda)$ as the negligible function. In this paper we use the implicit notation, i.e. $[x]_s = g_s^x = x \cdot [1]_s$ and we denote group

operations additively, i.e. $[x]_s + [y]_s = [x+y]_s$. By $([x]_1 \circ [y]_2) = [x \cdot y]_T$ we denote the pairing $e([x]_1, [y]_2)$.

Silent Threshold Encryption. The notion of silent threshold encryption (STE) was introduced by Garg et al [GKPW24]. This protocol allows parties to do a setup one time and generate hints corresponding to their secret keys. Any universe of decryptors can be chosen by running a preprocessing function that outputs an encryption key and an aggregation key.

A Silent Threshold Encryption (STE) consists of a tuple of algorithms (STE.Setup, STE.KGen, STE.HintGen, STE.Preprocess, STE.Enc, STE.PartDec, STE.DecAggr), and achieves the security properties of CPA security and an extended CPA security which we require for our proofs. We present the full syntax and these security properties in Appendix A.2.

Generalized Sumcheck We will use the generalized sumcheck lemma that was used in the construction of Silent Threshold Signatures [GJM+24] and was proven by Ráfols and Zapico in [RZ21].

Lemma 1 (Theorem 1, Generalized Sumcheck[RZ21]). *Let $A(x) = \sum_{i=1}^{\mathbb{H}} a_i \cdot L_i(x)$ and $B(x) = \sum_{i=1}^{\mathbb{H}} b_i \cdot L_i(x)$. It holds that*

$$A(x) \cdot B(x) = \sum_i \frac{a_i b_i}{|\mathbb{H}|} + Q_x(x) \cdot x + Q_Z(x) \cdot Z(x)$$

where Q_x and Q_Z are polynomials with degree $< |\mathbb{H}| - 2$ and are defined as

$$Q_x(x) = \sum_i a_i \cdot b_i \cdot \frac{L_i(x) - L_i(0)}{x}$$

$$Q_z(X) = \sum_i a_i \cdot b_i \frac{L_i^2(x) - L_i(x)}{Z(x)} + \sum_i a_i \sum_{j \neq i} b_j \frac{L_i(x) \cdot L_j(x)}{Z(x)}$$

and $Z(x)$ is the vanishing polynomial defined as $\prod_{i=1}^{|\mathbb{H}|}(x - \omega^i)$.

Programmable Global Random Oracle. The programmable global random oracle functionality $\mathcal{G}_{\mathsf{RO}}$ allows the adversary to program the oracle on a single point at a time. The functionality $\mathcal{G}_{\mathsf{RO}}$ internally keeps a list of preimage-value assignments, and if asked to rewrite one of these assignments it aborts. We define the functionality formally in Appendix A.

3.1 An Ideal Functionality for Secure Aggregation

In this section we present an ideal functionality for secure aggregation. We first outline the guarantees it must satisfy, then detail its interface.

Participants and Threat Model: We denote S as the server, $\mathcal{P}$ as the set of clients, and $\mathcal{P}_{\mathsf{sid}}$ as the set of clients selected to participate in an iteration of the aggregation. A static adversary may corrupt the server and a subset of clients (denoted $\mathcal{C}$). In each session, the environment chooses a subset of clients, ensuring corruptions remain below threshold.

Privacy of Inputs: The server learns only the *sum* of honest inputs, never individual values, provided enough honest clients participate.

Input Soundness: Aggregation proceeds if and only if at least $\mathtt{n_{min}}$ inputs from clients are provided. This notion tolerates dropouts but preventing trivial leakage and other input privacy attacks.

Overview of the functionality. We now describe the ideal functionality and its interface. The `REGISTER` command is used by clients to register with the functionality. The functionality adds the client to $\mathcal{P}$ and notifies the adversary that a client P_i has registered.

To start an iteration of the aggregation, the functionality first sends `START-SESSION` along with the set of parties (denoted $\mathcal{P}_{\mathsf{sid}}$) in this iteration to the adversary, and upon confirmation from the

adversary it requests inputs from parties via REQ-INP command. This ensures that the session starts only if the server agrees participate.

Each client in $\mathcal{P}_{\mathsf{sid}}$ submits its input to the functionality via the AGG-INPUT command. Crucially the functionality only sends the identity of the client to the adversary and not the input of the client. This ensures **privacy of inputs**.

Parameters:

- N: An upper-bound on the total number of clients in the system denoted
- $\mathsf{n}_{\mathsf{min}}$: A parameter that denotes the minimum number of inputs to be aggregated.
- The set of clients that are provide inputs for a session with idenitifier sid are denoted as $\mathcal{P}_{\mathsf{sid}}$.
- $t_{\mathcal{C}}$: A parameter that denotes the maximum number of corrupt parties in $\mathcal{P}_{\mathsf{sid}}$.

Participants: A set of clients denoted $\mathcal{P}$, initialized as $\emptyset$ and upper-bounded by N. A set of corrupt clients $\mathcal{C} \subset \mathcal{P}$ and a server S.
Functionality:

- **Registration** (one time): Upon receiving $(\texttt{REGISTER}, P_i)$ from some party P_i, update $\mathcal{P} \leftarrow \mathcal{P} \cup \{P_i\}$ and send $(\texttt{REGISTER}, P_i)$ to $\mathcal{A}$.
- **Input Phase**:
  1. Send $(\texttt{START-SESSION}, \mathsf{sid}, \mathcal{P}_{\mathsf{sid}})$ to $\mathcal{A}$ and if $(\texttt{START-SESSION}, \mathsf{sid}, \texttt{OK})$ received, send $(\texttt{REQ-INP}, \mathsf{sid}, P_i)$ to each client in $\mathcal{P}_{\mathsf{sid}}$, else output $\perp$.
  2. Upon receiving $(\texttt{AGG-INPUT}, \mathsf{sid}, x_i)$ from each client $P_i \in \mathcal{P}_{\mathsf{sid}}$, send $(\texttt{AGG-INPUT}, \mathsf{sid}, P_i)$ to the adversary $\mathcal{A}$.
- **Adversary Chooses Set**: If S is corrupt, receive $(\texttt{CHOOSE-SET}, \mathcal{D})$ from $\mathcal{A}$, else $\mathcal{D} := \mathcal{P}_{\mathsf{sid}}$
- **Compute Output**: If $|\mathcal{D}| > \mathsf{n}_{\mathsf{min}}$,
  1. send $(\texttt{PROCEED}, \mathsf{sid})$ to $\mathcal{A}$, else output $\perp$. If $\mathcal{A}$ responds $(\texttt{PROCEED}, \mathsf{sid}, \texttt{OK})$, then output $\sum_{P_i \in \mathcal{D}} x_i$ to S, else send $\perp$ to S.
  2. If $|\mathcal{D} \cap \mathcal{C}| > t_{\mathcal{C}}$, send (P_i, x_i) for each honest $P_i \in \mathcal{P}_{\mathsf{sid}}$ to $\mathcal{A}$.

Figure 3: $\mathcal{F}_{\mathsf{agg}}$ ideal functionality

If the server is corrupted, the adversary can choose (using the CHOOSE-SET command) inputs of which subset of clients in $\mathcal{P}_{\mathsf{sid}}$ are to be included in the aggregation. This set is denoted as $\mathcal{D}$.

The output is computed if and only if the size of the set $\mathcal{D}$ is greater than $\mathtt{n}_{\mathtt{min}}$ (a parameter defined in the application). This captures **input soundness** The functionality first asks if the adversary wants to continue using the PROCEED command. The adversary can choose to abort the protocol at this point. If the adversary confirms to proceed, then the functionality computes the sum of the inputs of the clients in $\mathcal{D}$ and sends it to the server.

Finally, if the number of corrupt inputs in $\mathcal{D} \cap \mathcal{C} > t_{\mathcal{C}}$ (a parameter of the application denoting the maximum number of corrupt parties allowed in a session), then the adversary receives each of the individual inputs of the honest clients in that session.

Assumption	Usage in TACITA
Global Random Oracle (GRO)	Simulator programs GRO to align real and ideal executions in UC proof
AGM + DLOG	Security of MKLHTS (unforgeability and extractability).
Trusted CRS	Succinct verification in MKLHTS
PKI with Key Transparency	Ensures consistent client keys visible to all parties (e.g., via verifiable logs).

Table 2: Assumptions at a glance and their role in TACITA.

4 Man-in-the-middle Attack on Willow

In this section we present the man-in-the-middle attacks on Willow where the server can generate ciphertexts on behalf of honest clients and learn the private input of a target honest client.

Below we first present an abridged version of Willow's [BCGL+24] protocol and then present the attack. We present a protocol where there is a single verifier - V and decryptor - D, and the decryptor has already done the key generation step.

Input Phase: Each client P_i does the following:

1. Receive D and $\mathsf{pk}, \mathsf{pk}_{aux}$ from the server.
2. Set $k \leftarrow \mathsf{KAHE.KGen}()$ and $seed \leftarrow \mathsf{PRG.KGen}()$
3. Compute $y := \mathsf{PRG.Expand}(seed, \ell) + x$
4. Set $m := \mathsf{KAHE.Enc}(y, x)$
5. Sample $r \leftarrow \{0,1\}^{\lambda}$ and a nonce $\tau \leftarrow \{0,1\}^{128}$
6. Set $(\mathsf{ct}_0, \mathsf{ct}_1, \pi) \leftarrow \mathsf{AHE.VerifiableEnc}(k, \mathsf{pk}, r, \tau)$
7. Set $c = E.\mathsf{Enc}(seed, \mathsf{pk}_{aux})$
8. Send $(m, (ct_0, ct_1), \pi, c)$ to server S

Aggregation Phase: Server upon receiving (m, ct, π, c) from each client C_i:

1. Compute $encKey = encKey + \mathsf{ct}$
2. Compute $encSum = encSum + m$
3. $encSeeds.\mathsf{Append}(c)$
4. $proofs.\mathsf{Append}(\pi)$

Parse $encKey$ as $(\mathsf{ct}^0, \mathsf{ct}^1)$ and send $(\mathsf{ct}^1, proofs)$ to the verifier V

Verification Phase: The verifier upon receiving $(\mathsf{ct}^1, proofs)$ from S:

1. Check that $|proofs| \geq min_n$
2. Check $\forall \pi \in proofs$: $\mathsf{AHE.VerifyEnc}(\pi) = 1$
3. Check $(\pi.nonce)_{\pi \in proofs}$ contains no duplicates.
4. Check $\sum_{\pi \in proofs} \pi.ct = \mathsf{ct}^1$

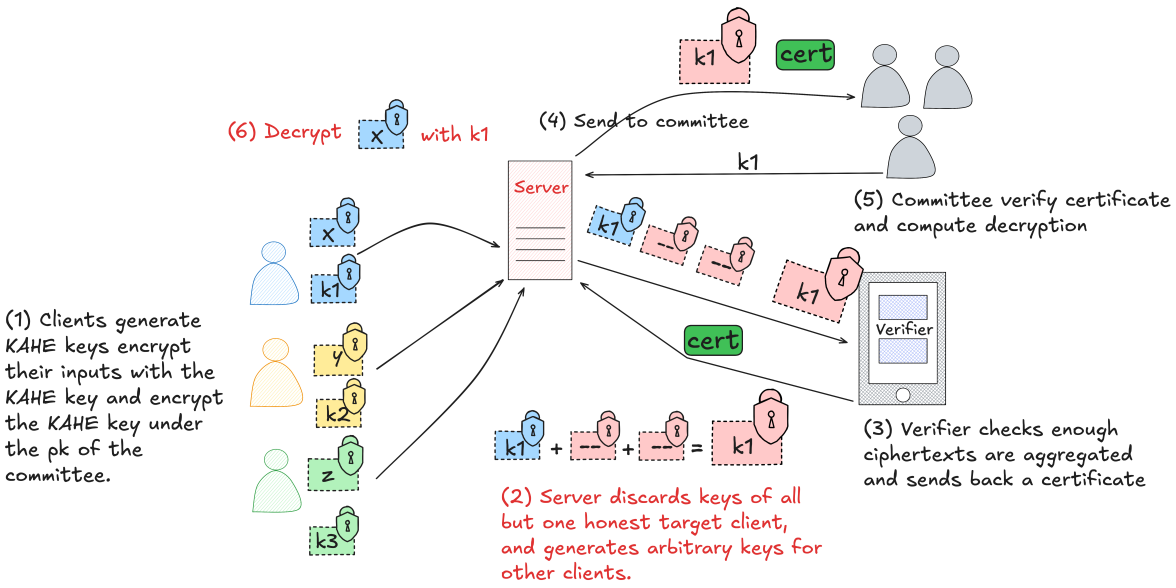

Figure 4: Overview of attack on Willow by malicious server that does not corrupt any clients, but manipulates the ciphertexts.

5. If all checks pass, send $\sigma_V = \mathsf{Sign}(\mathsf{sk}_V, \mathsf{ct}^1)$ to S

Decryption Phase:

1. The server S sends $(\mathsf{ct}^1, \sigma_V)$ to the decryptor D.
2. D computes $k = \mathsf{AHE.Dec}(\mathsf{sk}, ct^1)$ and sends (k, sk_{aux}) to the server S
3. Recover $maskSum = \sum_{c \in encSeeds} \mathsf{PRG.Expand}(E.\mathsf{Dec}(\mathsf{sk}_{aux}, c), \ell)$
4. Output $\mathsf{KAHE.Dec}(encSum, k) - maskSum$

We now present the attack on Willow (See Figure 4 for an overview). For this attack, we only need a malicious server, who does not collude with any of the other entities. The goal of the adversary is to learn the private input of the client P_{tgt}.

The main intuition of the attack is that the adversary can simply compute ciphertexts on behalf of honest clients except the target client. The server learns the final sum as in the honest protocol, and then remove the inputs of the honest clients for which it generated the inputs and computed ciphertexts. This allows the server to learn the exact input of the target client.

The Attack:

1. The input phase is completed as in the protocol and the server receives $(m, (\mathsf{ct}_0, \mathsf{ct}_1), \pi, c)$ from each client $P_i \in \mathcal{C}$, including that of P_{tgt}.
2. The server discards each $(m, (\mathsf{ct}_0, \mathsf{ct}_1), \pi, c)$ except the one of the target. Denote the target's messages as $m_{tgt}, (\mathsf{ct}_0, \mathsf{ct}_1)_{tgt}, \pi_{tgt}, c_{tgt}$.
3. The server locally computes the input phase message corresponding to each party $P_i \in \mathcal{C} \setminus \{P_{tgt}\}$. That is compute $m, (\mathsf{ct}_0, \mathsf{ct}_1, \pi, c)$ for each $P_i \in \mathcal{C} \setminus \{P_{tgt}\}$. Let $x' = \sum_{j \in \mathcal{C} \setminus \{P_{tgt}\}} x_j$
4. Compute the aggregation phase honestly, let $encSum' = \sum_{i \in \mathcal{C} \setminus \{P_{tgt}\}} m$ and let $encSeeds' = encSeeds \setminus c_{tgt}$

5. Interact with V and run the verification phase with the newly generated ciphertexts, and receive σ_V.
6. Send $(\mathsf{ct}^1, \sigma_V)$ to D and receive (k, sk_{aux}).
7. Compute $maskSum$ as in the protocol and output $x_{tgt} = \mathsf{KAHE.Dec}(encSum, k) - maskSum - x'$.

Remark. The CRYPTO proceedings version of OPA[KP25] does not make use of PKI and signatures and therefore is susceptible to a slightly different version of the man-in-the-middle attack. However, after corresponding with the authors, they have updated the latest eprint version[KP24] with a remark that they do need a PKI and signature to prevent the man-in-the-middle attack.

4.1 Sybil Attacks and Differential Privacy

In Willow [BCGL+24] the authors mention that they do not protect against Sybil attacks. While the problem of Input Soundness is related to that of Sybil attacks we remark that they are quite different attacks in terms of adversary capability. In a Sybil attack the malicious server colludes with enough clients in an instance of the protocol. In the attack we describe the malicious server needs to corrupt no clients.

However, Willow also describe a differential privacy approach (Appendix I of [BCGL+24]) to protect against Sybil attacks, and therefore also the input-soundness attack described above. They present two ideas - one leveraging global differential privacy and the other local.

In the global version, the committee/decryptors provide inputs with added noise. Crucially they are required to sign these input messages. The verifier then provides a certificate only if the signatures of the committee verify. Now, the attack on input soundness may proceed as described above, but the adversary only sees a noisy version of the honest client's inputs. However there are a few concerns: 1) the global noise may not be enough to hide the inputs of a single client 2) their protocol is now not "one-shot" since the decryptors need to interact with the server, once in the input phase and once again in the decryption phase.

In the local DP version, the clients provide local noise to their inputs. Here the server can simply remove the noise and inputs from the other parties and learn a noisy version of the target honest client. A concern here is that typically local differential privacy may add too much noise, which may not be suitable for applications.

We note that if one were to assume a PKI and require that clients can authenticate their inputs then the above attacks are mitigated. However, assuming a PKI affects the communication complexity of the protocol since the server needs to forward the signatures on the inputs to the committee. One of the contributions of TACITA is to avoid this blowup in communication complexity but at the same time to achieve input-soundness.

5 Our Silent Threshold Encryption Protocol

The Modified Silent Threshold Encryption (STE). We adopt the Silent Threshold Encryption (STE) framework of Garg et al. [GKPW24], but require two critical extensions to suit our setting:

1. **Additive homomorphism.** We need ciphertexts to support additive homomorphism so that aggregation can be performed without interaction.
2. **Ciphertext-bound partial decryptions.** In the original STE, partial decryption keys are tied to the public tag only, which allows a malicious server to reuse partial decryptions across ciphertexts with the same tag. Our setting demands that partial decryptions be specific to a given ciphertext.

To achieve these, we make two main modifications:

- The encryption tag is made a public parameter, allowing ciphertexts to be additively combined.
- Each partial decryption is computed as a function of a ciphertext-specific component (specifically, the sixth group element in the second part of the ciphertext vector). This binds each decryption share to a unique ciphertext.

This exact construction with efficient chunking and reusing randomness also appears in a recent unpublished manuscript of Bormet et al[BCF+25].

For details on the construction and its intuition we refer the reader to the elegant protocol of Garg et. al.[GKPW24].

Below we present the modified silent threshold protocol, and highlight the differences to the STE protocol by [GKPW24] in red.

- **Setup**(1^λ): Sample $\tau \leftarrow \mathbb{Z}_p$ and output:

$$\mathsf{CRS} = \left([\tau^1]_1, \ldots, [\tau^{M+1}]_1, [\tau^1]_2, \ldots, [\tau^{M+1}]_2\right).$$

- **KGen**(1^λ): Sample $x \leftarrow \mathbb{Z}_p^*$ and output $\mathsf{pk} = [x]_1$, $\mathsf{sk} = x$.

- **HintGen**$(\mathsf{CRS}, \mathsf{sk}_i, i, M)$: Output

$$\mathsf{hint}_i = \Bigg([\mathsf{sk}_i L_i(\tau)]_1, [\mathsf{sk}_i(L_i(\tau) - L_i(0))]_1, \left[\mathsf{sk}_i \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}\right]_1,$$
$$\left[\mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1, \left\{\left[\mathsf{sk}_i \frac{L_i(\tau)L_j(\tau)}{Z(\tau)}\right]_1\right\}_{j\in[0,M],j\neq i}\Bigg).$$

- **Preprocess**$(\mathsf{CRS}, \mathcal{U}, \{\mathsf{hint}_i, \mathsf{pk}_i\}_{i\in\mathcal{U}})$: Set $\mathsf{sk}_0 = 1$ and $\mathsf{sk}_i = 0$ for each $i \notin \mathcal{U}$ outside the universe.

$$\mathsf{ak} = \Bigg(\mathcal{U}, \{\mathsf{pk}_i\}_{i\in\mathcal{U}\cup\{0\}}, \left\{[\mathsf{sk}_i(L_i(\tau) - L_i(0))]_1\right\}_{i\in\mathcal{U}\cup\{0\}},$$
$$\left\{\left[\mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1\right\}_{i\in\mathcal{U}\cup\{0\}}, \left\{\left[\mathsf{sk}_i \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}\right]_1\right\}_{i\in\mathcal{U}\cup\{0\}},$$
$$\left\{\left[\sum_{j\in S, j\neq i} \mathsf{sk}_j \frac{L_i(\tau)L_j(\tau)}{Z(\tau)}\right]_1\right\}_{i\in\mathcal{U}\cup\{0\}}\Bigg).$$

$$\mathsf{ek} = \left(\left[\sum_{i\in\mathcal{U}} \mathsf{sk}_i L_i(\tau)\right]_1, [Z(\tau)]_2\right) := (\mathcal{C}, Z).$$

- $\mathsf{Enc}(\mathsf{ek}, \mathit{tag}, M, t)$: Parse $\mathsf{ek} := (C, Z)$, set $Z_0 = [\tau - \omega^0]_2$ and set

$$\mathbf{A} = \begin{pmatrix} C & [1]_2 & Z & [\tau]_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & [\tau]_2 & [1]_2 & 0 & 0 & 0 \\ 0 & \mathit{tag} & 0 & 0 & 0 & [1]_1 & 0 & 0 \\ [\tau^t]_1 & 0 & 0 & 0 & 0 & 0 & [1]_2 & 0 \\ [1]_1 & 0 & 0 & 0 & 0 & 0 & 0 & Z_0 \end{pmatrix},$$

$$\mathbf{b} = ([0]_T, [0]_T, [0]_T, [0]_T, [1]_T)^\top.$$

Sample a seed sd and get a vector $\mathbf{s} := \mathsf{RO}(\mathsf{sd}) \in (\mathbb{Z}_p^*)^5$ and output

$$\mathsf{ct} = \left(tag, \mathbf{s}^\top \cdot \mathbf{A}, \mathbf{s}^\top \cdot \mathbf{b} + M\right).$$

- **PartDec**$(\mathsf{sk}_j, \mathsf{ct})$:
  1. Parse $\mathsf{ct} := \left([\gamma]_2, \vec{\mathsf{ct}}_2, \mathsf{ct}_3\right)$
  2. Compute $\mathsf{part\text{-}dec}_j := \mathsf{sk}_j \cdot \vec{\mathsf{ct}}_2[6]$
  3. Compute Schnorr-style proof $\pi_{\mathsf{part\text{-}dec}}$ to prove correctness. (i.e. $\mathsf{pk}_j = [\mathsf{sk}_j] \wedge \mathsf{part\text{-}dec}_j = \mathsf{sk}_j \cdot \vec{\mathsf{ct}}_2[6]$)
  4. Output $(\mathsf{part\text{-}dec}_j, \pi_{\mathsf{part\text{-}dec}})$
- **PartVerify**$(\mathsf{ct}, \mathsf{part\text{-}dec}_j, \pi_{\mathsf{part}}\text{-}\mathsf{dec}, \mathsf{pk}_j)$: Parse $\mathsf{ct} := \left([\gamma]_2, \vec{\mathsf{ct}}_2, \mathsf{ct}_3\right)$ and output 1 if and only if the proof π_{part}-dec verifies.
- **Aggr**$(\vec{\mathsf{ct}})$:
  1. Parse each element i of the vector as $\mathsf{ct}^{(i)} = \left(tag^{(i)}, \vec{\mathsf{ct}}_2{}^{(i)}, \mathsf{ct}_3^{(i)}\right)$ for $i \in [\ell] = |\vec{\mathsf{ct}}|$. For any $i, j \in [\ell]$, if $tag^{(i)} \neq tag^{(j)}$, output $\perp$. Else, output $\mathsf{ct}^{\mathsf{Agg}} \leftarrow \left(tag^{(1)}, \sum_{i\in[\ell]} \vec{\mathsf{ct}}_2{}^{(i)}, \sum_{i\in[\ell]} \mathsf{ct}_3^{(i)}\right)$
- **DecAggr**$(\mathsf{CRS}, \mathsf{ak}, \mathsf{ct}, \{\mathsf{part\text{-}dec}_i\}_{i\in\mathcal{S}})$:
  1. Compute a polynomial $B(X)$ by interpolating 0 on all ω_i with $i \notin \mathcal{S}$ and 1 on ω^0, i.e., interpolate B as $\{(\omega^0, 1), (\omega^i, 0)_{i\notin\mathcal{S}}, (\omega^j, 1)_{j\in\mathcal{S}}\}$ and set
  $$B = \left[\sum_{i=0}^{M} B(\omega^i) L_i(\tau)\right]_2.$$
  2. Set $\mathsf{aPK} = \frac{1}{M+1}\left(\sum_{i\in\mathcal{S}} B(\omega^i)\mathsf{pk}_i + [1]_1\right)$.
  3. Compute polynomials $Q_x(X)$ and $Q_Z(X)$ such that
  $$\mathsf{SK}(X)B(X) = \frac{\mathsf{aSK}}{M+1} + Q_Z(X)Z(X) + Q_x(X)X,$$
  where
  $$\mathsf{aSK} = \sum_{i=0}^{M} \mathsf{SK}(\omega^i)B(\omega^i) := \sum_{i\in\mathcal{S}} \mathsf{sk}_i B(\omega^i) + 1$$
  4. Now, $Q_Z(X)$ and $Q_x(X)$ can be computed as:
  $$Q_Z(X) = \sum_{i=0}^{M+1} B(\omega^i)\left(\mathsf{sk}_i \frac{L_i^2(X) - L_i(X)}{Z(X)}\right) + \left(\sum_{i=0}^{M+1} B(\omega^i) \sum_{j=0, j\neq i}^{M+1} \mathsf{sk}_j \frac{L_i(X)L_j(X)}{Z(X)}\right),$$
  $$Q_x(X) = \sum_{i=0}^{M+1} B(\omega^i)\left(\mathsf{sk}_i \frac{L_i(X) - L_i(0)}{X}\right).$$
  Using ak, compute $Q_Z = [Q_Z(\tau)]_1$

5. Compute $Q_x = [Q_x(\tau)]_1$ and $\hat{Q}_x = [Q_x(\tau) \cdot \tau]_1$
6. Set
$$\mathsf{part\text{-}dec}^* = \frac{1}{M+1}\Big(\sum_{i \in \mathcal{C}_1} B(\omega_i) \cdot \mathsf{part\text{-}dec}_i \circ \mathsf{ct}_1\Big) - (\mathsf{ct}_{2,6} \circ \mathsf{ct}_1)$$
7. Compute
$$\hat{B} = [\tau^t \cdot B(\tau)]_1$$
8. Compute $Q_0(X)$ such that
$$B(X) - 1 = Q_0(X)(X - \omega^0)$$
and compute
$$Q_0 = [Q_0(\tau)]_1$$

Set
$$\mathbf{w} = \Big(B, -\mathsf{aPK}, -Q_Z, -Q_x, \hat{Q}_x, \mathsf{part\text{-}dec}^*, -\hat{B}, -Q_0\Big)^\top$$

Parse $\mathsf{ct} = (\mathsf{ct}_1, \mathbf{ct}_2, \mathsf{ct}_3)$ and output
$$\begin{aligned} M^* = \mathsf{ct}_3 - \Big((\mathsf{ct}_{2,1} \circ w_1) + (\mathsf{ct}_{2,2} \circ w_2) + (\mathsf{ct}_{2,3} \circ w_3) \\ + (\mathsf{ct}_{2,4} \circ w_4) + (\mathsf{ct}_{2,5} \circ w_5) + \mathsf{part\text{-}dec}^* \\ + (\mathsf{ct}_{2,7} \circ w_7) + (\mathsf{ct}_{2,8} \circ w_8)\Big) \end{aligned}$$

Our construction satisfies extended CPA security, even in the presence of adaptively chosen ciphertexts and malicious servers. The proof involves a hybrid argument, and demonstrates that no adversary can distinguish between different aggregated ciphertexts or extract information about individual inputs given the decryption of the aggregated ciphertext. We present this proof in Appendix B.

6 Multi-Key Linearly Homomorphic Threshold Signatures

In this section, we present the definition and our construction of Multi-key Linearly Homomorphic Threshold Signatures (MKLHTS).

Multi-key Homomoprhic Signatures (MKHS) [FMNP16, ABF24] allow one to evaluate a function on data signed by distinct users while producing a succinct and publicly-verifiable certificate of the correctness of the result. Anthoine et al[ABF24] show that given an updateable functional commitment scheme for a class of functions, a somewhere extractable batch argument scheme for NP, a somewhere extractable commitment scheme, and a digital signature scheme one can construct a multi-key homomorphic signature scheme. While elegant, the construction is not concretely efficient.

In our work, we only require a linear homomorphism over these signatures, that is, we require a threshold signature, and for the function $F : \mathbb{G}_1^n \to \mathbb{G}_1$ defined as $F(x_1, x_2, \ldots, x_n) = (x_1 + x_2 + \ldots + x_n)$. Moreover, we require that the signature scheme have only a one-time setup, and therefore must be a silent threshold signature scheme as well. Compared to the original multi-key authenticator scheme of Fiore et al[FMNP16] we require two new algorithms: an algorithm for hint generation (which can be seen as an extension of the key-generation algorithm), and another for pre-processing these hints to generate a verification key and aggregation key for a subset of the keys that were generated.

The following is the syntax for this silent threshold multi-key linearly homomorphic signature scheme:

Definition 1. *A Multi-key Homomorphic Threshold Signature Scheme (MKLHTS) consists of a tuple of algorithms -* $(\mathsf{MTS.Setup}, \mathsf{MTS.KGen}, \mathsf{MTS.HintGen}, \mathsf{MTS.Preprocess}, \mathsf{MTS.Sign}, \mathsf{MTS.PartialVerify}, \mathsf{MTS.SignAggr}, \mathsf{MTS.Verify})$ *with the following syntax.*

- $\mathsf{CRS} \leftarrow \mathsf{MTS.Setup}(1^\lambda, N)$: This algorithm takes as input the maximum possible size of users, denoted by N and outputs a CRS.
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{MTS.KGen}(1^\lambda)$: This algorithm allows a party to generate their private and secret keys.
- $\mathsf{hint}_i \leftarrow \mathsf{MTS.HintGen}(\mathsf{CRS}, \mathsf{sk}, N, i)$: This algorithm allows each party to generate *hints* that are related to their secret keys.
- $(\mathsf{ak}, \mathsf{vk}) \leftarrow \mathsf{MTS.Preprocess}(\mathsf{CRS}, \mathcal{S}, \{\mathsf{hint}_i, \mathsf{pk}_i\}_{i\in\mathcal{S}})$: This algorithm takes as input a subset $\mathcal{S} \subseteq [N]$ and outputs an aggregation key and verification key corresponding to the parties included in the subset $\mathcal{S}$.
- $\sigma, M \leftarrow \mathsf{MTS.Sign}(\mathsf{CRS}, \mathsf{ak}_{sig}, \mathsf{sk}, m, tag)$: This algorithm takes as input a secret key sk, a secret message m and a unique tag, and outputs a signature σ and a public message M.
- $0/1 \leftarrow \mathsf{MTS.PartialVerify}(M, tag, \sigma, \mathsf{pk})$: This algorithm takes as input the public message M, a public tag, the signature σ and public key pk and outputs 1 if the signature is verified and 0 otherwise.
- $M^*, \sigma^* \leftarrow \mathsf{MTS.SignAggr}(\mathsf{CRS}, tag, \mathsf{ak}, \{M_i, \sigma_i\}_{i\in\mathcal{S}}, t)$: This algorithm takes as input CRS, tag, the aggregation key that was computed as part of preprocessing ak for the set $\mathcal{S}$, all the signatures and public messages from the parties in $\mathcal{S}$, the threshold t and outputs a signature σ^* and aggregated message M^*.
- $0/1 \leftarrow \mathsf{MTS.Verify}(M^*, tag, \sigma^*, t, \mathsf{vk})$: This algorithm takes as input the aggregated message, and signature, and the aggregated verification key vk and outputs 0 or 1.

An MKLHTS construction must satisfy the following properties:

Authentication Correctness: This property requires that if a signature is computed correctly for honestly generated keys and CRS will always verify.

$$\mathsf{MTS.PartialVerify}(M, tag, \mathsf{pk}, \mathsf{MTS.Sign}(\mathsf{sk}, m, tag)) = 1$$

Evaluation Correctness: This property guarantees that if an aggregated signature is computed over signatures that are verified, then the aggregated signature verifies under the aggregated verification key on the aggregated message.

$$\left(\mathsf{MTS.Verify}(M^*, \mathsf{tag}, \sigma^*, t, \mathsf{vk}) = 1 \;\middle|\; \begin{array}{l} \forall i \in \mathcal{S} : \sigma_i = \mathsf{MTS.Sign}(\mathsf{sk}_i, M_i, \mathrm{tag}) \\ \wedge\ \mathsf{MTS.PartialVerify}(M_i, \mathrm{tag}, \sigma_i, \mathsf{pk}_i) = 1 \\ \wedge\ (M^*, \sigma^*) \leftarrow \\ \mathsf{MTS.SignAggr}(\mathsf{CRS}, \mathrm{tag}, \mathsf{ak}, \{M_i, \sigma_i\}_{i\in\mathcal{S}}) \end{array}\right)$$

Succinctness Full Succinctness implies the size that the size of the signature must be independent of the dataset size and the number of users and must grow only with respect to the size of security parameters.

Security The standard security of multi-key homomorphic signatures is that of Homomorphic Unforgeability under Chosen Message Attack (HUF-CMA). We adapt the definition of Fiore et al [FMNP16] to our setting below:

We define security via experiment between a challenger $\mathcal{C}$ and an adversary $\mathcal{A}$ defined in Figure 5.

Setup. $\mathcal{C}$ runs $\mathsf{MTS.Setup}(1^\lambda, N)$ to obtain CRS that is sent to the adversary $\mathcal{A}$.

Corruption. The adversary picks n and a subset of parties to corrupt denoted $\mathsf{L_{corr}} \leftarrow \mathcal{A}\mathsf{CRS}$, and sets $\mathsf{L_{corr}}$ to the challenger.

Key Generation. For all honest parties $i \in [n] \setminus \mathsf{L_{corr}}$, the public key and hints are honestly sampled by $\mathcal{C}$. $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{MTS.KGen}$ and $\mathsf{hint}_i \leftarrow \mathsf{MTS.HintGen}(\mathsf{CRS}, \mathsf{sk}_i, n)$. For all $i \in \mathsf{L_{corr}}$, the adversary picks a pk_i and corresponding hint_i.

Preprocess. The challenger preprocess $(\mathsf{ak}, \mathsf{vk}) \leftarrow \mathsf{MTS.Preprocess}(\mathsf{CRS}, \{\mathsf{pk}_i, \mathsf{hint}_i\}_{i\in[n]})$.

Authentication Queries. The adversary may make partial signature queries with some message M for some tag on behalf of an honest party i. Upon receiving (tag, M, i):

- If (tag, M, i) is the first query for tag, then initialize an empty set L_{tag} and proceed as follows.
- If (tag, M, i) is such that $(M, i) \notin \mathsf{L}_{tag}$, $\mathcal{C}$ computes $\sigma_i = \mathsf{MTS.Sign}(\mathsf{sk}_i, M, tag)$, returns σ_i to $\mathcal{A}$ and updates $\mathsf{L}_{tag} \leftarrow \mathsf{L}_{tag} \cup (M, i)$.
- If (tag, M, i) is such that $(*, i) \in \mathsf{L}_{tag}$, then $\mathcal{C}$ just ignores this query, since the adversary has already made a query for some M'.

Verification Queries. $\mathcal{A}$ is also given access to a verification oracle. Namely, the adversary can submit a query $(\mathcal{S}, tag, M, \sigma)$ and $\mathcal{C}$ replies with the output $\mathsf{MTS.Verify}(M, tag, \sigma, T, \mathsf{vk}_\mathcal{S})$, where $\mathsf{vk}_\mathcal{S}$ is the aggregated verification key of the parties in $\mathcal{S}$.

Forgery. In the end, $\mathcal{A}$ outputs a tuple $(\mathcal{S}^*, tag^*, M^*, \sigma^*)$. The experiment outputs 1, if the tuple returned by the adversary is a forgery (defined below), and 0 otherwise.

Definition 2 (Forgery)**.** *. Consider an execution of* HomUF-CMA *where $(\mathcal{S}^*, tag^*, M^*, \sigma^*)$ is the tuple returned by $\mathcal{A}$. This is a valid forgery if $|\mathcal{S}| > T$ and $\mathsf{MTS.Verify}(M^*, tag^*, \sigma^*, T, \mathsf{vk}_\mathcal{S}) = 1$, and for all $i \in \mathcal{S}^*$ we have that $i \notin \mathsf{L_{corr}}$ and either one of the following properties is satisfied:*

Type 1: L_{tag} *was not initialized.*

Type 2: For all $i \in \mathcal{S}$, there exist $(M_i, i) \in \mathsf{L}_{tag}$ but $M^* \neq \sum M_i$

Type 3: There exist an identity $i \in \mathcal{S}$, such that $(*, i) \notin \mathsf{L}_{tag}$

Figure 5: The HomUF-CMA Experiment

Extractability: Notice that the above definition of unforgeability requires that the output forgery requires that $i \notin \mathsf{L_{corr}}$. This implies that the aggregated signature must consist of only honest party inputs. While this is in line with previous definitions of unforgeability, this is not enough for our proof of security for the secure aggregation protocol. Specifically, in the secure aggregation protocol we cannot enforce that a malicious server will include only honestly generated ciphertexts, the adversary may try to compute artificial ciphertexts computed ciphertexts to learn the inputs of honest parties. To this end, we require the property of extractability, which is quite similar to the knowledge soundness property of succinct non-interactive arguments or polynomial commitment schemes. Specifically we require that there exists a polynomial time algorithm that can extract the inputs of the corrupt parties from the aggregated signature (i.e., the input signatures σ_i and message m_i used to generate σ_i).

Definition 3. *A MKLHTS scheme is extractable if there exists a polynomial time algorithm $\mathcal{E}$ such that*

$$\Pr\left[\begin{array}{l}\mathsf{MTS.Verify}(\mathsf{CRS}, t, \mathsf{vk}, M^*, \sigma^*) = 1 \\ \wedge\, (\exists i \in \mathcal{S} \setminus \mathcal{H} :\ \sigma_i \neq \mathsf{MTS.Sign}(\mathsf{CRS}, tag, \mathsf{sk}_i, m_i) \\ \quad \vee\ (M^*, \sigma^*) \neq \mathsf{MTS.SignAggr}\big(\mathsf{CRS}, t, \mathsf{ak}, \{M_i, \sigma_i\}_{i\in\mathcal{S}}\big)\big)\end{array}\right]$$

$$= \mathsf{negl}(\lambda)$$

where

$$\begin{aligned}
&\mathsf{CRS} \leftarrow \mathsf{MTS.Setup} \\
&\wedge\ (\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{KGen}(1^\lambda) \\
&\wedge\ (\sigma^*, M^*) \leftarrow \mathcal{A}(\mathsf{CRS}, \{M_i, \sigma_i\}_{i \in \mathcal{H}}) \\
&\wedge\ (m_i, \sigma_i) \leftarrow \mathcal{E}(\sigma^*, M^*, \mathsf{CRS}, \{M_i, \sigma_i\}_{i \in \mathcal{H}})
\end{aligned}$$

6.1 Overview of our construction

The primitive: *Multi-Key Linearly Homomorphic Threshold Signatures (MKLHTS)* enables clients to sign elements $[s_i] \in \mathbb{G}_1$ and allows an aggregator to produce a succinct signature σ^* on $\sum_i [s_i]$ that is verifiable under an aggregated verification key. The verification succeeds only if a threshold number of signatures were correctly aggregated.

Our Starting Point - hinTs[GJM+24]. hinTs presents a regular threshold signing protocol. In their protocol each party does a one-time setup and generates keys and hints. Crucially, the parties do not need to interact with each other. After this, each party signs a message and publishes the corresponding signature. An aggregator then collects these partial signatures, and aggregates them to compute a final signature. The aggregator must also provide a proof that the aggregation has been done correctly.

Specifically, in their protocols the signature provided by the parties is a simple BLS signature on a message M. That is, $\sigma_i = \mathsf{sk}_i \cdot H(M)$. Now, aggregating BLS signatures is easily done as $\sigma^* = \sigma_1 + \ldots + \sigma_N$, and the corresponding verification key is computed as $\mathsf{aPK} = \mathsf{pk}_1 + \ldots + \mathsf{pk}_N$.

Now the aggregator must prove two things 1) the aPK was computed correctly, 2) at least t signatures were aggregated to compute σ^*. Let B denote the vector of signers that participate in the signing i.e. $B[i] = 1$ if they submit a transaction, and 0 otherwise. Now the aggregator needs to proves that the Hamming weight of B is greater than t to satisfy the second statement. To prove the first statement the strategy is as follows: the aggregator first encodes the vector B as a polynomial $B(X)$. Notice that by definition:

$$\begin{bmatrix} b_1 & b_2 & \ldots & b_n \end{bmatrix} \cdot \begin{bmatrix} \mathsf{pk}_1 \\ \mathsf{pk}_2 \\ \vdots \\ \mathsf{pk}_N \end{bmatrix} = \mathsf{aPK}$$

The aggregator therefore proves such a statement using the generalized sumcheck protocol (Lemma 1) by first committing to the different secret keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_N$ and committing to the polynomial $B(X)$ and computes the polynomials $Q_x(X)$ and $Q_Z(X)$ such that

$$SK(X) \cdot B(X) = \mathsf{aSK} + Q_Z(X) \cdot Z(X) + Q_x(X) \cdot X$$

That is the aggregator will compute the polynomial commitments $[Q_Z(\tau)]_1$ and $[Q_x(\tau)]$ and any verifier just verifies the pairing equation:

$$\begin{aligned}
([SK(\tau)]_1 \circ [B(\tau)]_2) = (\mathsf{aPK} \circ [1]_2) + ([Q_Z(\tau)]_1 \circ [Z(\tau)]_2) \\
+([Q_x(\tau)]_1, [\tau]_2)
\end{aligned}$$

Now, of course the aggregator does not have access to the secret keys of the parties. Therefore, to compute these polynomial commitments, the aggregator will need to use some hints that are published by the parties before the start of the protocol.

Specifically, the parties publish $\mathsf{hint}_i =$

$$\Bigg([\mathsf{sk}_i \cdot L_i(\tau)]_1, [\mathsf{sk}_i \cdot (L_i(\tau) - L_i(0))]_1, \left[\mathsf{sk}_i \cdot \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}\right]_1,$$
$$\left[\mathsf{sk}_i \cdot \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1, \left\{ \left[\mathsf{sk}_i \cdot \frac{L_i(\tau)L_j(\tau)}{Z(\tau)}\right]_1 \right\}_{j\in[0,M],j\neq i} \Bigg).$$

And in a preprocessing step, the aggregator combines these hints to get an aggregation key as:

$$\left\{ [\mathsf{sk}_i(L_i(\tau) - L_i(0))]_1 \right\}_{i\in\mathcal{U}\cup\{0\}},$$
$$\left\{ \left[\mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1 \right\}_{i\in\mathcal{U}\cup\{0\}}, \left\{ \left[\mathsf{sk}_i \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}\right]_1 \right\}_{i\in\mathcal{U}\cup\{0\}},$$
$$\left\{ \left[\sum_{j\in S, j\neq i} \mathsf{sk}_j \frac{L_i(\tau)L_j(\tau)}{Z(\tau)} \right]_1 \right\}_{i\in\mathcal{U}\cup\{0\}} \Bigg).$$

Thus the aggregator is able to compute the quotient polynomial commitments $[Q_Z(\tau)]_1$ and $[Q_x(\tau)_1]$, which are computed as

$$[Q_Z(\tau)]_1 = \sum_{i\in[n]} b_i \cdot (\mathsf{sk}_i \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}) + \sum_{i\in[n]} b_i \cdot \sum_{j\neq i} \mathsf{sk}_j \frac{L_i(\tau)L_j(\tau)}{Z(\tau)}$$

and

$$[Q_x(\tau)]_1 = \sum_{i\in[n]} b_i \cdot \mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau}$$

Technical Challenges in the MKLHTS setting. While hinTs assumes all clients sign the *same message*, our secure aggregation setting requires each client to sign a *distinct message* – their masked input. This gives rise to two key challenges:

1. The aggregator must prove that σ^* corresponds to the *sum* $\sum_i [s_i]$, and that each s_i was signed under a valid key.
2. The aggregated signature must be constant-sized, and crucially must not include the set of verification keys of the participating clients.

Our Approach. The BLS signature ($\sigma = \mathsf{sk} \cdot H(m)$) scheme enjoys a natural and efficient aggregatability property, which we aim to preserve in our construction. However, adapting BLS to our secure aggregation setting does not work directly.

First, note that the hash function in BLS (modeled as a random oracle) is not homomorphic, thus we cannot hope to get a homomorphism over the messages with BLS.

Second, we need to bind each signature to a specific aggregation instance to prevent replay across different sessions. To achieve this, we incorporate a public session identifier, denoted tag, into each signature.

We achieve linear homomorphism and session binding, by defining a new signing algorithm where each client P_i signs their input $s_i \in \mathbb{Z}_p$ by computing:

$$\sigma_i = \mathsf{sk}_i \cdot (H(\mathsf{tag}) + [s_i])$$

A similar construction was defined by Aranha et al [AP19] for a simple linearly homomorphic multikey signature scheme. However, they do not achieve succinctness.

To verify the signature, one does not need the s_i itself, but can do so with the corresponding group element $S_i = [s_i]$ by checking the following pairing check:

$$(\mathsf{vk}_i \circ H(tag)) + (\mathsf{vk}_i \circ S_i) = ([1] \circ \sigma_i)$$

Now each client computes σ_i and sends $(\sigma_i, [s_i])$ to the aggregator. The aggregator computes:

$$\sigma^* = \sum_i \sigma_i, \quad S^* = \sum_i [s_i], \quad \mathsf{vk}^* = \sum_i \mathsf{vk}_i.$$

Thus the overall flow of the scheme is as follows:

- Each party will sign a group element S_i (i.e. compute $\sigma_i = \mathsf{sk}_i \cdot (H(tag) + S_i)$), and the aggregator aggregates these signatures and group elements as $\sigma^* = \sum_{i\in[n]} \sigma_i$ and $R^* = \sum_{i\in[n]} S_i$
- The aggregator then proves that the aggreation was done correctly. Specifically, it must prove that S^*, σ^* and aPK were computed correctly and that greater than t parties provided the signatures. We elaborate below on how this achieved.

Proving Correct Aggregation. The aggregator now must prove that the aggregated S^*, σ^* and vk^* are valid, and that they are aggregated using $\geq t$ signatures and messages.

To prove the latter, the server computes a commitment to the vector B and proves that the Hamming weight of B is greater than t. This is exactly as in hinTs [GJM+24].

Proving well-formedness of vk^*: The aggregator commits to a polynomial $B(X)$ encoding the set of signers and uses a generalized sumcheck argument (Lemma 1) to verify:

$$SK(X) \cdot B(X) = \mathsf{vk}^* + Q_Z(X)Z(X) + Q_x(X)X.$$

Proving well-formedness of S^*:To show well-formedness of S^* and σ^*, the server will need to compute additional quotient polynomials. That is, the aggregator needs to prove that the signature is computed using the partial signatures of the clients, and that this is a valid signature on the S^*.

Let $S(X)$ denote the polynomial encoding each $s_i \in \mathbb{Z}_p$ such that $S_i = [s_i]_1$. Recall that the aggregator commits to the vector B which indicates the set of parties that contribute the partial signatures. Thus, the aggregator must prove that:

$$S(X) \cdot B(X) = \sum_{i\in[n]} s_i + Q_x^1(X) \cdot X + Q_Z^1(X) \cdot Z(X)$$

Proving well-formedness of σ^*: Next, the aggregator must prove the well-formedness of $\sigma*$. One natural idea would be to use the generalized sumcheck protocol (see Lemma 1) using $S(X)$ and $SK(X)$.

That is, consider the following polynomial equation that must be satisfied:

$$SK(X) \cdot S(X) = \sum_{i\in[n]} sk_i \cdot s_i + Q_x^2(X) \cdot X + Q_Z^2(X) \cdot Z(X)$$

This translates to the following pairing check:

$$([SK(\tau) \circ [S(\tau)]]) = (\sum_{i\in[n]} \mathsf{sk}_i \cdot S_i \circ [1]) + ([Q_x^2(\tau)] \circ [\tau]) + ([Q_Z^2(\tau)] \circ [Z(\tau)]) \tag{1}$$

However, this pairing check is not verifiable (and not even computable by the aggregator) since the aggregator cannot compute $\sum_{i\in[n]} \mathsf{sk}_i \cdot S_i$. Instead, the aggregator only has:

$$\sigma^* = \sum_{i\in[n]} \sigma_i = \sum_{i\in[n]} \mathsf{sk}_i \cdot (H(tag) + S_i)$$

However, observe that:

$$(\sigma^* \circ [1]) = \Big(\sum_{i\in[n]} \mathsf{sk}_i \cdot (S_i + H(tag)) \circ [1] \Big)$$

$$= \Big(\sum_{i\in[n]} \mathsf{sk}_i \cdot S_i \circ [1] \Big) + \Big(\sum_{i\in[n]} (\mathsf{sk}_i \cdot H(tag)) \circ [1] \Big)$$

$$= \Big(\sum_{i\in[n]} \mathsf{sk}_i \cdot S_i \circ [1] \Big) + \Big(\sum_{i\in[n]} (\mathsf{sk}_i) \cdot H(tag) \circ [1] \Big)$$

$$= \Big(\sum_{i\in[n]} \mathsf{sk}_i \cdot S_i \circ [1] \Big) + \Big(H(tag) \circ \sum_{i\in[n]} \mathsf{sk}_i [1] \Big)$$

$$= \Big(\sum_{i\in[n]} \mathsf{sk}_i \cdot S_i \circ [1] \Big) + \Big(H(tag) \circ \mathsf{vk}^* \Big)$$

Thus adding $(H(tag) \circ \mathsf{vk})$ to both sides of Equation 1:

$$([SK(\tau) \circ [S(\tau)]]) + \boxed{(H(tag) \circ \mathsf{vk})} = (\sum_{i\in[n]} S_i^{\mathsf{sk}_i} \circ [1]) \cdot ([Q_x^2(\tau)] \circ [\tau]) \cdot ([Q_Z^2(\tau)] \circ [Z(\tau)] + \boxed{(H(tag) \circ \mathsf{vk})}$$

$$= ([SK(\tau) \circ [S(\tau)]]) + (H(tag) \circ \mathsf{vk}) = \boxed{(\sigma^*, [1])} + ([Q_x^2(\tau)] \circ [\tau]) + ([Q_Z^2(\tau)] \circ [Z(\tau)]$$

Thus with this pairing check, the signature verification can be accomplished and that σ^* has been computed correctly.

Hints to compute the quotient polynomial commitments One final aspect that we need to consider is that the aggregator will need enough information to compute these polynomials. That is, the aggregator needs *hints* from the signers to compute these values. We first recall, that the signers will have already computed hints via MTS.HintGen before the start of the protocol. There are two issues we need to consider (1) How can we use these hints to compute new hints that aid the server in computing the quotient polynomials $Q_Z^1, Q_x^1, Q_Z^2, Q_x^2$ (2) How can we ensure that the size of these hints is constant-sized? The hints output by MTS.HintGen are dependent on the number of parties in the system. Thus simply having the parties send these hints as part of the signature is not practical since the size of the signature is now linear in the number of parties. Specifically, the output of MTS.Sign must include a constant-sized hint for the message s_i denoted as s-hint and constant-sized hints for the product of $s_i \cdot \mathsf{sk}_i$ denoted as $(\mathsf{s} \cdot \mathsf{sk})$-hint. We show in the next subsection how to generate these hints such that the correct quotient polynomials can be computed.

6.2 Computing hints for quotient polynommials

Computation of Q_x^1 and Q_Z^1. Recall that Q_x^1 and Q_Z^1 are two polynomials such that

$$S(X)B(X) = \sum_{i\in[n]} s_i + Q_x^1(X) \cdot X + Q_Z^1(X) \cdot Z(X)$$

And from the Generalized Sumcheck (Lemma 1) we see that

$$Q_Z^1(X) = \sum_{i\in[n]} \left(s_i \cdot b_i \frac{L_i^2(X) - L_i(X)}{Z(X)} \right) + \sum_{i\in[n]} s_i \cdot \sum_{i\neq j} \left(b_j \cdot \frac{L_i(X) L_j(X)}{Z(X)} \right)$$

Now one thing we immediately observe is that the first term, $\left(s_i \cdot b_i \frac{L_i^2(X)-L_i(X)}{Z(X)}\right)$ can be locally computed by party P_i. Thus the party just appends

$$\mathsf{s\text{-}hint}_{i,1} = \left[\left(s_i \frac{L_i^2(X) - L_i(X)}{Z(X)}\right)\right]$$

to the signature.

But the second term $s_i \cdot \sum_{i \neq j} \left(b_j \cdot \frac{L_i(X)L_j(X)}{Z(X)}\right)$ cannot be computed by party P_i since it does not know ahead of time which parties will participate (i.e. $b_j = 1$) in the protocol.

To get around this issue, observe that a party not participating is equivalent to saying that the input of the party P_j is simply zero. Thus if we assume that all parties participate in the protocol, and the parties that do not participate have their inputs set as zero, then each party can compute the second terms locally as well. More specifically each party P_i now computes

$$\mathsf{s\text{-}hint}_{i,2} = \left[s_i \cdot \sum_{i \neq j} \left(\frac{L_i(\tau)L_j(\tau)}{Z(\tau)}\right)\right]$$

and appends this to the signature as well.

Similarly, for the computation of $Q_x(X)$, a client can locally compute

$$\mathsf{s\text{-}hint}_{i,3} = \left[s_i \cdot \frac{L_i(\tau) - L_i(0)}{\tau}\right]$$

To compute the polynomial commitment $[S(\tau)]$, the client also computes $\mathsf{s\text{-}hint}_{i,0} = s_i \cdot [L_i(\tau)]_1$. The aggregator also needs $\mathsf{s\text{-}hint}_{i,4} = s_i \cdot [L_i(\tau) - L_i(0)]_1$ for the PLONK style proof

These hints are sent to the aggregator, who now computes the quotient polynomials for the following polynomial equation:

$$S(X) \cdot A(X) = \sum_{i \in [n]} s_i + Q_x^1(X) \cdot X + Q_Z^1(X) \cdot Z(X)$$

where $A(X)$ encodes the all one vector.

The aggregator computes

$$[Q_x^1(\tau)] = \sum_{i \in [n]} \mathsf{s\text{-}hint}_{i,3}$$

and computes

$$[Q_Z^1(\tau)] = \sum_{i \in [n]} (\mathsf{s\text{-}hint}_{i,1} + \mathsf{s\text{-}hint}_{i,2})$$

Computation of Q_x^2 and Q_Z^2. Recall that Q_x^2 and Q_Z^2 are polynomials such that

$$= ([SK(\tau) \circ [S(\tau)]]) \cdot (H(tag) \circ \mathsf{vk}) =$$

$$(\sigma^*, [1]) \cdot ([Q_x^2(\tau)] \circ [\tau]) \cdot ([Q_Z^2(\tau)] \circ [Z(\tau)]$$

To compute Q_x^2 and Q_Z^2 we basically need polynomials that satisfy the following polynomial equation:

$$SK(X) \cdot S(X) = \sum_{i \in [n]} sk_i \cdot s_i + Q_x^2(X) \cdot X + Q_Z^2(X) \cdot Z(X)$$

Now let us observe the terms in $Q_Z^2(X)$ and $Q_x^2(X)$.

$$Q_Z^2(X) = \sum_{i \in [n]} \left(s_i \cdot \mathsf{sk}_i \frac{L_i^2(X) - L_i(X)}{Z(X)} \right) + \sum_{i \in [n]} s_i \cdot \sum_{i \neq j} \left(\mathsf{sk}_j \cdot \frac{L_i(X) L_j(X)}{Z(X)} \right)$$

As before the first term can be computed locally by party P_i

$$(\mathsf{s} \cdot \mathsf{sk})\text{-}\mathsf{hint}_{i,1} = \left[s_i \cdot \mathsf{sk}_i \frac{L_i^2(X) - L_i(X)}{Z(X)} \right]$$

Moreover notice that the MTS.Preprocess algorithm outputs

$$\left[\sum_{j \in [n], j \neq i} \mathsf{sk}_j \frac{L_i(\tau) L_j(\tau)}{Z(\tau)} \right]$$

as part of the aggregation key, for each i. Thus the party P_i can simply compute the second term as

$$(\mathsf{s} \cdot \mathsf{sk})\text{-}\mathsf{hint}_{i,2} = s_i \cdot \left[\sum_{j \in [n], j \neq i} \mathsf{sk}_j L_i(\tau) L_j(\tau) Z(\tau) \right]$$

And finally for Q_x^2, the party P_i can locally compute

$$(\mathsf{s} \cdot \mathsf{sk})\text{-}\mathsf{hint}_{i,3} = s_i \cdot \left[\mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau} \right]$$

The client also computes $(\mathsf{s} \cdot \mathsf{sk})\text{-}\mathsf{hint}_{i,4} = s_i \cdot [L_i(\tau) - L_i(0)]_1$ for the PLONK-style proof.
The aggregator computes

$$[Q_x^2(\tau)] = \sum_{i \in [n]} (\mathsf{s} \cdot \mathsf{sk})\text{-}\mathsf{hint}_{i,3}$$

and computes

$$[Q_Z^2(\tau)] = \sum_{i \in [n]} ((\mathsf{s} \cdot \mathsf{sk})\text{-}\mathsf{hint}_{i,1} + \mathsf{s}\text{-}\mathsf{hint}_{i,2})$$

Thus the final signature sent by party P_i now not only includes the signature σ_i, but also includes the hints - $\{\mathsf{s}\text{-}\mathsf{hint}_{i,j}\}_{j \in [0,4]}$, $\{(\mathsf{s} \cdot \mathsf{sk})\text{-}\mathsf{hint}_{i,j}\}_{j \in [1,4]}$, which are 9 additional group elements.

6.3 Our Full MKLHTS Construction

- **Setup**(1^λ): Sample $\tau \leftarrow \mathbb{Z}_p$ and output:

$$\mathsf{CRS} = \left([\tau^1]_1, \ldots, [\tau^N]_1, [\tau^1]_2, \ldots, [\tau^N]_2 \right).$$

- **KGen**(1^λ): Sample $x \leftarrow \mathbb{Z}_p^*$ and output $\mathsf{pk} = [x]_1, \mathsf{sk} = x$.

- **HintGen**$(\mathsf{CRS}, \mathsf{sk}_i, i, N)$:
  Compute hint_i as follows:

$$\Bigg([\mathsf{sk}_i \cdot L_i(\tau)]_1, [\mathsf{sk}_i \cdot L_i(\tau)]_2, [\mathsf{sk}_i \cdot (L_i(\tau) - L_i(0))]_1, \left[\mathsf{sk}_i \cdot \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}\right]_1,$$
$$\left[\mathsf{sk}_i \cdot \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1, \left\{\left[\mathsf{sk}_i \cdot \frac{L_i(\tau)L_j(\tau)}{Z(\tau)}\right]_1\right\}_{j\in[0,N],j\neq i}\Bigg).$$

  Output (hint_i)

- **Preprocess**$(\mathsf{CRS}, \mathcal{U}, \{\mathsf{hint}_i, \mathsf{pk}_i\}_{i\in\mathcal{U}})$: Set $\mathsf{sk}_0 = 1$ and $\mathsf{sk}_i = 0$ for each $i \notin \mathcal{U}$ outside the universe.

$$\mathsf{ak} = \Bigg(\mathcal{U}, \{\mathsf{pk}_i\}_{i\in\mathcal{U}\cup\{0\}}, \left\{[\mathsf{sk}_i(L_i(\tau) - L_i(0))]_1\right\}_{i\in\mathcal{U}\cup\{0\}},$$
$$\left\{\left[\mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1\right\}_{i\in\mathcal{U}\cup\{0\}}, \left\{\left[\mathsf{sk}_i \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}\right]_1\right\}_{i\in\mathcal{U}\cup\{0\}},$$
$$\left\{\left[\sum_{j\in S, j\neq i} \mathsf{sk}_j \frac{L_i(\tau)L_j(\tau)}{Z(\tau)}\right]_1\right\}_{i\in\mathcal{U}\cup\{0\}}\Bigg).$$

$$\mathsf{SK}(\tau) = \sum \mathsf{sk}_i L_i(\tau) \quad W(\tau) = \sum L_i(\tau) \quad Z(\tau) = \prod_{i\in[n+1]} (\tau - \omega^i)$$

  The verification key is

$$\mathsf{vk} = [\mathsf{SK}(\tau)]_1, \mathsf{SK}(\tau)]_2, [W(\tau)]_1, [Z(\tau)]_2$$

- **Sign**$(\mathsf{sk}, \mathsf{ak}_i, tag, m_i, \mathcal{S})$:
  Let $M_i = [m_i]_1$.
  Compute $\sigma_i = sk_i \cdot ([m_i]_2 + H(tag))$
  Compute $\{\mathsf{s\text{-}hint}_{i,j}\}_{j\in[0,4]}$ as follows:

$$\mathsf{s\text{-}hint}_{i,0} = m_i \cdot [L_i(\tau)]_1$$

$$\mathsf{s\text{-}hint}_{i,1} = \left[\left(m_i \frac{L_i^2(X) - L_i(X)}{Z(X)}\right)\right]_1$$

$$\mathsf{s\text{-}hint}_{i,2} = \left[m_i \cdot \sum_{i\neq j}\left(\frac{L_i(\tau)L_j(\tau)}{Z(\tau)}\right)\right]_1$$

$$\mathsf{s\text{-}hint}_{i,3} = \left[m_i \cdot \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1$$

$$\mathsf{s\text{-}hint}_{i,4} = m_i \cdot [L_i(\tau) - L_i(0)]_1$$

  Compute $\{(\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,j}\}_{j\in[1,4]}$ as follows:

$$(\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,1} = \left[m_i \cdot \mathsf{sk}_i \frac{L_i^2(X) - L_i(X)}{Z(X)}\right]_1$$

$$(\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,2} = m_i \cdot \left[\sum_{j\in[n], j\neq i} \mathsf{sk}_j L_i(\tau) L_j(\tau) Z(\tau)\right]_1$$

$$(\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,3} = m_i \cdot \left[\mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau}\right]_1$$

$$(\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,4} = m_i \cdot [L_i(\tau) - L_i(0)]_1$$

Output $(M_i, \sigma_i, \mathsf{s}\text{-}\mathsf{hint}_i, (\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_i)$

- **SignAggr**$(\mathsf{CRS}, \mathsf{ak}, \{\sigma_i, M_i, r_i\}_{i\in\mathcal{S}})$
  Compute aggregated public key and signature as

$$\mathsf{aPK} = \sum_{i\in\mathcal{S}} pk_i \quad \sigma^* = \sum_{i\in\mathcal{S}} \sigma_i \quad M^* = \sum_{i\in\mathcal{S}} M_i$$

  The final signature is $\mathsf{aPK}, \sigma^*, \pi$ and the aggregated message is M^*.
  The proof π is as follows:

  1. Commitment to the set $\mathcal{S}$ as
  $$[B(\tau)]_2 = \sum_{i\in\mathcal{S}} [L_i(\tau)]_2$$
  2. Let $A(X)$ be the polynomial encoding the all ones vector. Compute commitments to quotient polynomials satisfying

$$\mathsf{SK}(X)\cdot B(X) - \mathsf{aSK} = Q_Z(X)\cdot Z(X) + Q_x(X)\cdot X$$

$$S(X)\cdot A(X) - \sum m_i = Q_Z^1(X)\cdot Z(X) + Q_x^1(X)\cdot X$$

$$\mathsf{SK}(X)\cdot S(X) - \sum_{i\in\mathcal{S}} \mathsf{sk}_i \cdot m^i = Q_Z^2(X)\cdot Z(X) + Q_x^2(X)\cdot X$$

  where the polynomial commitments $[Q_x(\tau)]$ and $Q_Z(\tau)$ are computed as follows:

$$[Q_Z(\tau)]_1 = \sum_{i\in[n]} b_i \cdot \left(\mathsf{sk}_i \frac{L_i^2(\tau) - L_i(\tau)}{Z(\tau)}\right) + \sum_{i\in[n]} b_i \cdot \sum_{j\neq i} \mathsf{sk}_j \left(\frac{L_i(\tau) L_j(\tau)}{Z(\tau)}\right)$$

$$[Q_x(\tau)]_1 = \sum_{i\in[n]} b_i \cdot \mathsf{sk}_i \frac{L_i(\tau) - L_i(0)}{\tau}$$

  $Q_x^1(\tau)$ and $Q_Z^1(\tau)$ are computed as

$$[Q_x^1(\tau)] = \sum_{i\in[n]} \mathsf{s}\text{-}\mathsf{hint}_{i,3}$$

$$[Q_Z^1(\tau)] = \sum_{i\in[n]} (\mathsf{s\text{-}hint}_{i,1} + \mathsf{s\text{-}hint}_{i,2})$$

and $Q_x^2(\tau)$ and $Q_Z^2(\tau)$ are computed as

$$[Q_x^2(\tau)] = \sum_{i\in[n]} (\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,3}$$

$$[Q_Z^2(\tau)] = \sum_{i\in[n]} ((\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,1} + (\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,2})$$

3. Compute

$$[Q_x(\tau)\cdot\tau]_1 = \sum_{i\in\mathcal{S}} [sk_i\cdot(L_i(\tau) - L_i(0))]_1$$

$$[Q_x^1(\tau)\cdot\tau]_1 = \sum_{i\in\mathcal{S}} \mathsf{s\text{-}hint}_{i,4}$$

$$[Q_x^2(\tau)\cdot\tau]_1 = \sum_{i\in\mathcal{S}} (\mathsf{s}\cdot\mathsf{sk})\text{-}\mathsf{hint}_{i,4}$$

4. Let ParSum be the following polynomial:

$$\mathsf{ParSum}(x) = \sum_{i\in\mathcal{S}} L_i(x)$$

Compute $[\mathsf{ParSum}(\tau)]_1$

5. Compute quotient polynomials $Q_1(x)$ and $Q_2(x)$ such that

$$\mathsf{ParSum}(x\cdot\omega) - \mathsf{ParSum}(x) - (W(x) - t\cdot L_{n+1}(x))\cdot B(x) = Z(x)\cdot Q_1(x) \tag{2}$$

and

$$B(x)\cdot(1 - B(x)) = Z(x)\cdot Q_2(x) \tag{3}$$

Commit to them by computing $[Q_1(\tau)]_1$ and $[Q_2(\tau)]_1$

6. Random challenge r is computed using random oracle as $r \leftarrow \mathrm{RO}\Big(t, [B(\tau)]_2, [Q_Z(\tau)]_1, [Q_x(\tau)]_1, [Q_x(\tau)\cdot\tau]_1, [\mathsf{ParSum}(\tau)]_1, [Q_1(\tau)]_1, [Q_2(\tau)]_1\Big)$

Compute the following opening and batch opening proof of the committed polynomials

- Open and prove $\mathsf{ParSum}(\omega) = 0$
- Open and prove $B(\omega^{n+1}) = 1$
- Open and prove the following at $x = r$: $\{\mathsf{ParSum}(x), W(x), B(x), Q_1(x), Q_2(x)\}$
- Open and prove $\mathsf{ParSum}(x)$ at $x = r\cdot\omega$

- **Verify**$([M]_2, \sigma, t, \mathsf{vk})$ Verify the proof π as
  - Verify all opening proof of the polynomials.
  - Check that Equation 2 and 3 hold at the evaluation point r.

– Check that

$$([\mathsf{SK}(\tau)]_1 \circ [B(\tau)]_2) - (\mathsf{aPK} \circ [1]_2) =$$
$$([Q_Z(\tau)_1 \circ Z(\tau)]_2) + ([Q_x(\tau)]_1 \circ [\tau]_2)$$

$$([S(\tau)]_1 \circ [\mathsf{SK}(\tau)]_2) + (\mathsf{aPK}, H(tag)) - (\sigma^* \circ [1]_2) =$$
$$([Q_Z^2(\tau)]_1 \circ [Z(\tau)]_2) + ([Q_x^2(\tau)]_1 \circ [\tau]_2)$$

$$([S(\tau)]_1 \circ [A(\tau)]_2) - (M^* \circ [1]_2) =$$
$$([Q_Z^1(\tau)]_1 \circ [Z(\tau)]_2) + ([Q_x^1(\tau)]_1 \circ [\tau]_2)$$

– Run the degree check for $Q_x(\tau), Q_x^1(\tau), Q_x^2(\tau)$ as

$$([Q_x(\tau)]_1 \circ [\tau]_2) = ([Q_x(\tau) \cdot \tau]_1 \circ [1]_2)$$

$$([Q_x^1(\tau)]_1 \circ [\tau]_2) = ([Q_x^1(\tau) \cdot \tau]_1 \circ [1]_2)$$

$$([Q_x^2(\tau)]_1 \circ [\tau]_2) = ([Q_x^2(\tau) \cdot \tau]_1 \circ [1]_2)$$

If all checks pass output 1, else output 0

Theorem 1. *The MKLHTS scheme presented above is unforgeable and extractable in the AGM model under the discrete log assumption.*

Proof. (Sketch) The unforgeability of our MKLHTS scheme follows from the unforgebality of the underlying signature scheme $\sigma_i = \mathsf{sk}_i \cdot (H(tag) + [s_i])$ and the soundness guarantees of the PLONK-style proof from hinTs. We show that if an adversary is able to break the unforgeability of the our MKLHTS scheme, we can write a reduction to the discrete log assumption in the AGM.

For extractability we resort to the knowledge soundness of the underlying polynomial commitment scheme (KZG commitments) to extract the inputs of the parties.

We defer the full proofs of security to Appendix C.

□

7 Our Protocol - TACITA

In this section we present our full protocol - TACITA.

Public Selection: Via public randomness a set of clients and committee members are selected to participate in an iteration of the secure aggregation protocol. We remark that as long as a threshold number of clients and committee members are honest, the protocol provides correctness of output and privacy for the client inputs. A threshold encryption public key pk^{enc} and an aggregate verification key is computed using the public keys of the committee members and clients respectively. We consider this an offline setup phase when the clients and the committee may locally compute the encryption key and the aggregation key.

Input Phase: In the simplest form, the client encrypts its private input x_i using pk^{enc} and then signs the ciphertext using sk_i^{sig} for the MKLHTS scheme. However, for UC security against malicious adversaries, we need to modify the encryption scheme slightly. The client first samples a random value rand_i and evaluates the random oracle on rand_i to get a mask s_i. The client then masks its input x_i with s_i to compute $\hat{x}_i = x_i + s_i$. The clients then compute two ciphertexts, one using our STE

scheme defined in Section 5 and another using $\overline{\mathsf{STE}}$ which is the STE scheme of Garg et al[GKPW24]. Specifically, the clients encrypt the masked input $\hat{x}_i$ using our STE and the randomness rand_i using $\overline{\mathsf{STE}}$. Why? Recall that $\overline{\mathsf{STE}}$ allows decryption of any ciphertext encrypted under the corresponding tag and public key. This enables the server to decrypt and learn the individual rand_i and remove the mask with a single decryption key.

This improves upon previous work Willow [BCGL+24] and OPA [KP24] who also use the same technique of masking with a random oracle output to achieve malicious security. Specifically, in Willow the committee need to run a DKG for each iteration, and have the clients encrypt the rand_i under this public key. The commitee then provides the secret key to the server allowing it to decrypt each ciphertext encrypting rand_i. In OPA on the other hand the clients secret share rand_i and encrypt it to the committee via the server. The committee decrypts the ciphertexts and sends back the shares of rand_i to the server, who then aggregates for each client to compute rand_i. This incurs $\mathcal{O}(N)$ communication to the committee.

Our use of STE avoids the communication blow-up as in OPA and does not require the committee to run a DKG for each iteration. Now after the computing these ciphertexts, the clients sign only the 6-th ciphertext of $\vec{\mathsf{ct}}_2$ the STE ciphertext. In Appendix E we discuss why this is sufficient. The clients then send the $\mathsf{STE}, \overline{\mathsf{STE}}$ ciphertexts and MKLHTS signature to the server.

Aggregation Phase: The server aggregates the STE ciphertexts to compute $\mathsf{CT}_1, \vec{\mathsf{CT}}_2, \mathsf{CT}_3$ and aggregates the signatures using the $\mathsf{MTS.SignAggr}$ function of the MKLHTS signature scheme. This aggregated signature σ^* proves that enough ciphertexts were aggregated and that they originated from the clients. The server then sends the aggregated signature and aggregated STE ciphertext, along with the $\overline{tag}$ to the committee.

Decryption Phase: Each committee member first verifies the aggregated signature using the aggregate verification key. This ensures that the server did not replace any of the ciphertexts, that the ciphertexts that are being decrypted correspond exactly to the clients selected in that iteration, and that atleast a threshold number of client inputs have been aggregated. The committee members then compute the partial decryption of the STE ciphertext, computed as $\mathsf{sk}_j \cdot \vec{\mathsf{CT}}_2[6]$, and the partial decryption required to decrypt the $\overline{\mathsf{STE}}$ ciphertext, computed as $\mathsf{sk}_j \cdot H(\overline{tag})$, and send these two partial decryptions back to the server. The server then aggregates the STE partial decryptions. This results in the computation of X which is equal to $\sum_i [\hat{x}_i]$. Now the server needs to remove the masks which is equal to $\sum_i s_i$. The server first aggregates the $\overline{\mathsf{STE}}$ partial decryptions. Recall that the server can now decrypt any ciphertext encrypted under $\overline{tag}$ the public key of the commitee. This allows the server to decrypt each of the rand_i. The server then evaluates $s_i = H(\mathsf{rand}_i)$ and outputs $[\sum_i x_i] = X - [\sum_i s_i]$. Finally, the server computes the discrete log to get the actual sum $\sum_i x_i$. This concludes our protocol.

One-time Setup

Setup:

1. Run $\mathsf{CRS}_{enc} \leftarrow \mathsf{STE.Setup}(1^\lambda, N)$ and $\mathsf{CRS}_{sig} \leftarrow \mathsf{MTS.Setup}(1^\lambda, N)$

Key Generation: Each client P_i

1. Compute $(\mathsf{pk}_i^{enc}, \mathsf{sk}_i^{enc}) \leftarrow \mathsf{STE.KGen}(1^\lambda)$ and compute $\mathsf{hint}_i^{enc} \leftarrow \mathsf{STE.HintGen}(\mathsf{CRS}_{enc}, \mathsf{sk}_i^{enc}, N, i)$
2. Compute $(\mathsf{pk}_i^{sig}, \mathsf{sk}_i^{sig}) \leftarrow \mathsf{MTS.KGen}(1^\lambda)$ and compute $\mathsf{hint}_i^{sig} \leftarrow \mathsf{MTS.HintGen}(\mathsf{CRS}_{enc}, \mathsf{sk}_i^{sig}, N, i)$
3. Publish $(\mathsf{pk}_i^{enc}, \mathsf{hint}_i^{enc}, \mathsf{pk}_i^{sig}, \mathsf{hint}_i^{sig})$

Aggregation Phase

Aggregation: Let $\mathcal{C}$ denote set of input-providing clients Committee be clients enabling decryption in the current *iteration* of the protocol. Let $|\mathcal{C}| = n$ and $|\mathsf{Committee}| = \ell$ and $tag = H(iteration|aux)$ be the identifier for the current iteration. (Note that for FL use-cases, aux will include the model data)

1. Server S (Clients may also run steps (a) and (b)):
   (a) Run $\mathsf{STE.Preprocess}(\mathsf{CRS}, \mathsf{Committee}, \{(\mathsf{pk}_i^{enc}, \mathsf{hint}_i^{enc})\}_{i \in \mathsf{Committee}})$ and output $(\mathsf{ek}, \mathsf{ak}_{enc})$.
   (b) Run $\mathsf{MTS.Preprocess}(\mathsf{CRS}, \mathcal{C}, \{(\mathsf{pk}_i^{sig}, \mathsf{hint}_i^{sig})\}_{i \in \mathcal{C}})$ and output $\mathsf{vk}, \mathsf{ak}_{sig}$.
   (c) Send $(\mathsf{Committee}, \mathsf{ek}, tag, aux)$ to each client $P_i \in \mathcal{C}$, and $(\mathcal{C}, \mathsf{vk}, tag)$ to Committee
2. Each $P_i \in \mathcal{C}$ with input x_i verifies that $\mathsf{ek}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ak}_{sig}$ and tag were computed correctly and then:
   (a) Sample $\mathsf{rand}_i \leftarrow \{0,1\}^\lambda$ and compute $s_i \leftarrow H_{\mathsf{RO}}(\mathsf{rand}_i)$. Compute $\hat{x}_i = x_i + s_i$
   (b) Compute $(\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i \leftarrow \mathsf{STE.Enc}(\mathsf{ek}, tag, t, \hat{x}_i)$
   (c) Compute $(\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i \leftarrow \overline{\mathsf{STE.Enc}}(\mathsf{ek}, \overline{tag}, t, \mathsf{rand}_i)$
   (d) Compute $\sigma_i \leftarrow \mathsf{MTS.Sign}(\mathsf{sk}_i^{sig}, \mathsf{ak}_{sig}, tag, \vec{\mathsf{ct}}_2[6])$
   (e) Send $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ to the Server.
3. Server S upon receiving $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ from $i \in \mathcal{C}$:
   (a) Compute $(\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3) = \sum_{i \in \mathcal{C}} \mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i$
   (b) Compute $\sigma^* = \mathsf{MTS.SignAggr}(\mathsf{CRS}, tag, \mathsf{ak}_{sig}, \{(\vec{\mathsf{ct}}_2[6])_i\}_{i \in \mathcal{C}})$
   (c) Send $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ to each client $P_j \in \mathsf{Committee}$
4. Each client $P_j \in \mathsf{Committee}$ does:
   (a) Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
   (b) If yes, compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$ and $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$ and return $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$ to the Server.
5. The server S upon receiving $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$ from each of the $P_j \in \mathsf{Committee}$ does:
   (a) Compute $X \leftarrow \mathsf{STE.DecAggr}(\mathsf{CRS}, \mathsf{ak}_{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \{\mathsf{part\text{-}dec}\}_{j \in \mathsf{Committee}})$
   (b) For $i \in [n]$: compute $\mathsf{rand}_i \leftarrow \overline{\mathsf{STE.DecAggr}}(\mathsf{CRS}, \mathsf{ak}_{enc}, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \{\overline{\mathsf{part\text{-}dec}}\}_{j \in \mathsf{Committee}})$ and compute $s_i = H_{\mathsf{RO}}(\mathsf{rand}_i)$
   (c) Compute discrete log of $X - [\sum_i s_i]_T$ via look-up table and output $\sum_i x_i$ (see section 8.1 for a discussion on how this discrete algorithm can be efficiently computed).

Figure 6: TACITA: Our Secure Aggregation Protocol

We are now ready to prove our main theorem:

Theorem 2. *The secure aggregation protocol presented in Fig 6 UC-realizes the $\mathcal{F}_{\mathsf{agg}}$ ideal functionality (Fig 3) assuming a secure silent threshold encryption scheme, and a multi-key homomorphic silent threshold signature scheme in the programmable $\mathcal{G}_{\mathsf{RO}}$-hybrid model.*

Proof. To prove UC-security, we construct a PPT simulator for $\mathcal{F}_{\mathsf{agg}}$ that produces a transcript indistinguishable from the real protocol. We distinguish two cases:

Honest server. The simulator receives encrypted inputs from corrupt clients. Since it controls honest committee keys (under the honest-majority assumption), it can decrypt these inputs. To emulate server–committee interaction, the simulator sends encryptions of zero on behalf of the server, then forwards $(\texttt{AGG-INPUT}, x_i)$ for each corrupt client to $\mathcal{F}_{\mathsf{agg}}$, sends `PROCEED`, and outputs the aggregated sum.

Malicious server. Upon receiving $(\texttt{AGG-INPUT}, P_i)$ from $\mathcal{F}_{\mathsf{agg}}$, the simulator samples random values, encrypts them under $\mathsf{STE.Enc}$ and $\overline{\mathsf{STE.Enc}}$, and signs them for the server. Acting as an honest committee member, it returns aggregated ciphertexts and a signature; if the signature verifies, it provides a partial decryption. The simulator then issues `PROCEED` to $\mathcal{F}_{\mathsf{agg}}$ and learns the honest clients' sum. Finally, when the server queries the random oracle on an honest input, the simulator programs the oracle so the aggregate matches the ideal sum. □

We present our full proof of security in Appendix D.

8 Secure Aggregation for vectors of arbitrary length

In this exposition thus far we have only described the protocol as if client input is one object. However, in applications like federated learning client inputs are typically a vector, say of length ℓ. One of the main contributions of Willow [BCGL+24] and OPA [KP24] was to ensure that the communication complexity of the server and committee members in their protocols is independent of ℓ.

A naive approach of applying STE and MKLHTS to each element in the vector of inputs would lead to the communication complexity to be linear in the length of the vector. In this section, we show how we can apply the clever techniques in Willow and OPA to TACITA. Our main idea is to use a Key Additive Homomorphic Encryption Scheme (or alternatively seed-homomorphic PRGs). We briefly recall the definition below:

Essentially (symmetric-key) key-additive homomorphic encryption allows one to add S ciphertexts encrypting different messages, and the resulting ciphertext is encrypting the summation of the messages under all the S keys added. Formally, it contains the following PPT algorithms, adapted from [BCGL+24, Appendix C].

- $\mathsf{pp} \leftarrow \mathsf{KAHE.Setup}(1^\lambda, S)$: takes a security parameter λ; ouptuts a public parameter pp.
- $\mathsf{sk} \leftarrow \mathsf{KAHE.KeyGen}(\mathsf{pp})$: takes a public parameter pp; outputs a secret key sk
- $\mathsf{ct} \leftarrow \mathsf{KAHE.Enc}(\mathsf{pp}, \mathsf{sk}, i, m)$: takes a public parameter pp, a secret key sk, an index i, and a message m; outputs a ciphertext ct
- $m \leftarrow \mathsf{KAHE.Dec}(\mathsf{pp}, \mathsf{sk}, i, \mathsf{ct})$: takes a public parameter pp, a secret key sk, an index i, and a ciphertext ct; outputs a message m

Definition 4 (Correctness). *Let $\lambda > 0$ and $S = \mathsf{poly}(\lambda)$, let $\mathsf{pp} \leftarrow \mathsf{KAHE.Setup}(1^\lambda, S)$, for any $S' \leq S$, let $\mathsf{sk}_j \leftarrow \mathsf{KAHE.KeyGen}(\mathsf{pp})$ for $j \in [S']$, for any messages $m_1, \ldots, m_{S'}$, and any $i = \mathsf{poly}(\lambda)$, let $\mathsf{ct}_j \leftarrow \mathsf{KAHE.Enc}(\mathsf{pp}, \mathsf{sk}, i, m_j)$, let $\mathsf{ct} \leftarrow \sum_{j\in[S']} \mathsf{ct}_j$ and $\mathsf{sk} \leftarrow \sum_{j\in[S']} \mathsf{sk}_j$, it holds that $\Pr[\mathsf{Dec}(\mathsf{sk}, i, \mathsf{ct}) = \sum_{j\in[S']} m_j] = 1 - \mathsf{negl}(\lambda)$.*

Definition 5 (Security). *Let $\lambda > 0$, $S = \mathsf{poly}(\lambda)$, and $M = \mathsf{poly}(\lambda)$, let $\mathsf{pp} \leftarrow \mathsf{KAHE.Setup}(1^\lambda, S)$, and $\mathsf{sk}_j \leftarrow \mathsf{KAHE.KeyGen}(\mathsf{pp})$ for $j \in [S]$, we say that a KAHE scheme is S-semantic secure under leakage of sum of secret keys if there exists an efficient simulator Sim such that, for any sequences of input vectors $(m_{1,i}, m_{2,i}, \ldots, m_{S,i})$ for $i \in [M]$, the following distribution*

$$D_0 = \left\{ \left(\sum_{j\in[S]} \mathsf{sk}_j, \{c_{j,i}\}_{j\in[S],i\in[M]} \right) | c_{j,i} \leftarrow \mathsf{Enc}(\mathsf{pp}, \mathsf{sk}_j, i, m_{j,i}) \right\}$$

is computationally indistinguishable from

$$D_1 = \left\{ \mathsf{Sim}(\mathsf{pp}, (\sum_{j\in[S]} m_{j,i})_{i\in[M]}) \right\}$$

The Ring Learning with Error (RLWE) assumption. Before introducing a concrete instantiation of KAHE, we first introduce the RLWE assumption, since the security of the instantiation is based on the RLWE assumption.

Definition 6 (Decisional ring learning with error problem [LPR13]). *Let $N, q, \mathcal{D}, \chi$ be parameters dependent on λ and N being a power of two. Let $\mathcal{R} = \mathbb{Z}[X]/(X^N + 1)$. The decisional ring learning with error (RLWE) problem $\mathsf{RLWE}_{N,q,\mathcal{D},\chi}$ is the following: distinguish $(a, a \cdot s + e)$ and (a, b) (with noticeable advantage), where $a \leftarrow_\$ \mathcal{R}_q, s \leftarrow \mathcal{D}, e \leftarrow \chi$ and $b \leftarrow_\$ \mathcal{R}_q$.*

RLWE-based KAHE. KAHE can be instantiated using the RLWE assumption as follows.

- $\mathsf{KAHE.Setup}(1^\lambda)$: Let N be a power-of-two denoting the ring dimension, and let ring $R_q := \mathbb{Z}[X]/(q, X^N + 1)$ for some modulus $q > 0$. Let t be an integer coprime to q and the plaintext space is $\mathbb{Z}_t^N$. Let χ_σ denote the distribution over degree-N polynomials suhc that the coefficients are independent discrete Gaussians with standard deviation σ and mean 0. Let $\sigma_{\mathsf{sk}}, \sigma_e > 0$ be Gaussian parameters for the secret and error distributions respectively. Let s be a PRG seed. $\mathsf{pp} := (N, q, t, \sigma_s, \sigma_e, s)$.
- $\mathsf{KAHE.KeyGen}(\mathsf{pp})$: Sample $\mathsf{sk} \leftarrow D_{\sigma_{\mathsf{sk}}}$
- $\mathsf{KAHE.Enc}(\mathsf{pp}, \mathsf{sk}, i, m \in \mathbb{Z}_t^N)$: First computes $a \leftarrow \mathsf{PRG}(s, i)$ (where PRG is a PRG that takes a seed s and outputs polynomial number of ring elements in R_q and i denotes the i-th one). Then compute $\mathsf{ct} := a\mathsf{sk} + t \cdot e + m$.
- $\mathsf{KAHE.Dec}(\mathsf{pp}, \mathsf{sk}, i, \mathsf{ct})$: $m := \mathsf{ct} - \mathsf{PRG}(s, i)\mathsf{sk} \mod t$

Lemma 2 (Lemma 1, Willow [BCGL+24]). *For any $\lambda > 0$, the RLWE-based KAHE scheme above is correct (definition 4) and secure (definition 5) under the $\mathsf{RLWE}_{N,q,\chi_\sigma,\chi_\sigma}$ assumption if $\sigma_{\mathsf{sk}} = \sqrt{2}\sigma, \sigma_e = 2\sigma$.*

Remark 1. *Note that for cleanness, the interface above does not take S (i.e., the upper bound on the number of ciphertexts to-be-aggregated) as an input. However, since the scheme is based on RLWE, to guarantee correctness, S is needed to determine q and σ_e to guarantee correctness (such that the accumulated error does not affect decryption). For simplicity, we assume that Setup takes S implicitly to correctly determine q and σ_e. This is the same case for [BCGL+24] as well.*

Now given the KAHE scheme we describe an appraoch similar to Willow: Clients generate a KAHE key denoted k_i. As before they sample a random string rand_i and compute $\vec{s}_i[j] = H_{\mathsf{RO}}(\mathsf{rand}_i, j)$ for $j \in [\ell]$. They then mask their inputs vector using $\vec{s}$ as

$$\hat{\vec{x}}_i = \vec{s}_i + \vec{x}$$

This masked input vector is then encrypted using the KAHE scheme:

$$C_i = \mathsf{KAHE.Enc}(k_i, \hat{\vec{x}}_i)$$

Now instead of encrypting the $\hat{x}_i$ in the single input case (Figure 6), the client encrypts the KAHE key k_i using STE. As before the randomness rand_i is encrypted using $\overline{\mathsf{STE}}$ and the clients need to sign only the STE ciphertext using the MKLHTS scheme.

The server aggregates the STE ciphertexts and each of the C_i. Notice that the aggregate of the STE ciphertexts now encrypt $\sum_i k_i$ and the aggregation of the C_i results in an encryption of $\sum_i \hat{\vec{x}}_i$ under $\sum_i k_i$. The server also aggregates the MKLHTS signatures.

The server only sends the aggregated STE ciphertext and the aggegated signature to the committee as before. Notice that the server only needs to send constant sized messages here to the committee and crucially does not depend on the input size ℓ. The committee perform the same actions as before and give partial decryptions that enable the server to decrypt the aggregated STE ciphertext and the individual $\overline{\mathsf{STE}}$ ciphertexts. This allows the server to compute $K = \sum_i k_i$. The server then decrypts $\sum_i C_i$ to cpmpute $maskedSum = \sum_i \hat{\vec{x}}_i$. After decrypting the individual $\overline{\mathsf{STE}}$ ciphertexts as before, the server computes the rand_i. The server then computes each vector $\vec{s}_i$ as $\vec{s}_i[j] = H_{\mathsf{RO}}(\mathsf{rand}_i, j)$ and unmasks $\sum_i \hat{\vec{x}}_i$ as

$$\sum_i \vec{x}_i = maskedSum - \sum_i \vec{s}_i$$

Additional optimizations. One thing to note is that in our STE scheme above, encrypting a message m first computes g^m. Thus, decrypting a message requires the computation of discrete logrithm. Thus, m cannot be large. However, recall that the seed space $|\mathcal{S}| \geq 2^\lambda$ to guarantee λ-bit security. Computing discrete log over $\mathcal{S}$ can take super-polynomial time.

One direct idea is to then separate s into $\log(\lambda)$ bits and encrypt them using $\lambda/\log(\lambda)$ ciphertexts (i.e., client i separate $s_i := s_{i,1}||\ldots||s_{i,m}$ where $m = \lambda/\log(\lambda)$ and encrypts $s_{i,j}$ individually). However, recall that the committee does not get each individual ciphertexts, but instead getting the summed ciphertexts. This means that each partition of the seed is added separately, but in this case, the seed-homomorphism may get broken. In other words, $\sum_i s_i \neq \sum_i s_{i,1}||\ldots||\sum_i s_{i,m}$.

To avoid this issue, we use a specific type of KAHE, where $\mathcal{S} = \int^n$ where $q = \mathsf{poly}(\lambda)$, i.e., the seed is a vector of length n over $\int$. Then, we have $\vec{s}_i := (s_{i,1}, \ldots, s_{i,m})$ and $\sum s_i = \sum_i \vec{s}_i = \sum_i (s_{i,1}, \ldots, s_{i,m}) = (\sum_i s_{i,1}, \ldots, \sum_i s_{i,m})$. And since $q = \mathsf{poly}(\lambda)$, solving for $g^s_{i,j}$ takes at most $\mathsf{poly}(\lambda)$ time.

Lastly, for M summed ciphertexts, naively decoding $S[j] := \sum_i \vec{s}_i[j] \in \mathbb{Z}$ ($j \in [n]$) takes $O(q \cdot M)$ time (note that $S[j]$ is not in $\mathbb{Z}_q$ but in $\mathbb{Z}$). To reduce the runtime, we observe that the resulting $S[j]$ has an expected value of $Mq/2$. Therefore, we can choose a, b such that $S[j] \in (a, b)$ with probability $1 - \mathsf{negl}(\lambda)$ using Chernoff bound. In particular, $\Pr[S[j] \leq a] = \Pr[S[j] \leq Mq/2 - \Delta_a] \leq e^{\frac{-2\Delta_a^2}{Mq^2}}$, so as long as $\Delta_a = \Omega(\sqrt{M} \cdot \log^2(M) \cdot q)$, we have $\Pr[S[j] \leq a] = \mathsf{negl}(\lambda)$. Similarly, we can compute b. In this case, $b - a = \tilde{O}(\sqrt{Mq})$ instead of $O(Mq)$.

With all these, we formally present our scheme in Figure 7.

Remark 2. *With this change,* TACITA *allows the clients to have arbitrary input (i.e., input of size $\mathbb{Z}_t^\ell$) as prior works [BCGL+24, KP24]. Note that DLOG does* not *make our scheme restricted to have a small input. This is because under RLWE, we can simply set σ_s to output small secrets (i.e., each secret element bounded by $O(\sigma_s) = \mathsf{poly}(\lambda)$ for some security parameter λ. Then, DLOG only takes $\mathsf{poly}(\lambda)$ time (and how to compute DLOG more efficiently is discussed in section 8.1).*

However, one caveat is that this t must be chosen such that RLWE remains secure. Specifically, if t is exponentially large, the gap between the ciphertext modulus and noise is exponentially large, which results in an insecure RLWE assumption. Thus, t is restricted to be sub-exponential (but can be super-polynomially large). This restriction applies to [BCGL+24, KP24] as well since they rely on RLWE as well.

One-time Setup

Setup:

1. Run $\mathsf{CRS}_{enc} \leftarrow \mathsf{STE.Setup}(1^\lambda, N)$, $\mathsf{CRS}_{sig} \leftarrow \mathsf{MTS.Setup}(1^\lambda, N)$ and $\mathsf{CRS}_{\mathsf{KAHE}} \leftarrow \mathsf{KAHE.Setup}(1^\lambda)$

Key Generation: Each client P_i

1. Compute $(\mathsf{pk}_i^{enc}, \mathsf{sk}_i^{enc}) \leftarrow \mathsf{STE.KGen}(1^\lambda)$ and compute $\mathsf{hint}_i^{enc} \leftarrow \mathsf{STE.HintGen}(\mathsf{CRS}_{enc}, \mathsf{sk}_i^{enc}, N, i)$
2. Compute $(\mathsf{pk}_i^{sig}, \mathsf{sk}_i^{sig}) \leftarrow \mathsf{MTS.KGen}(1^\lambda)$ and compute $\mathsf{hint}_i^{sig} \leftarrow \mathsf{MTS.HintGen}(\mathsf{CRS}_{enc}, \mathsf{sk}_i^{sig}, N, i)$
3. Publish $(\mathsf{pk}_i^{enc}, \mathsf{hint}_i^{enc}, \mathsf{pk}_i^{sig}, \mathsf{hint}_i^{sig})$

Aggregation Phase

Aggregation: Let $\mathcal{C}$ denote set of input-providing clients Committee be clients enabling decryption in the current *iteration* of the protocol. Let $|\mathcal{C}| = n$ and $|\mathsf{Committee}| = \ell$ and $tag = H(iteration|aux)$ be the identifier for the current iteration. (Note that for FL use-cases, aux will include the model data)

1. Server S (Clients may also run steps (a) and (b)):
   (a) Run $\mathsf{STE.Preprocess}(\mathsf{CRS}, \mathsf{Committee}, \{(\mathsf{pk}_i^{enc}, \mathsf{hint}_i^{enc})\}_{i \in \mathsf{Committee}})$ and output $(\mathsf{ek}, \mathsf{ak}_{enc})$.
   (b) Run $\mathsf{MTS.Preprocess}(\mathsf{CRS}, \mathcal{C}, \{(\mathsf{pk}_i^{sig}, \mathsf{hint}_i^{sig})\}_{i \in \mathcal{C}})$ and output $\mathsf{vk}, \mathsf{ak}_{sig}$.
   (c) Send $(\mathsf{Committee}, \mathsf{ek}, tag, aux)$ to each client $P_i \in \mathcal{C}$, and $(\mathcal{C}, \mathsf{vk}, tag)$ to Committee
2. Each $P_i \in \mathcal{C}$ with input $\vec{x}_i$
   (a) Sample $\mathsf{rand}_i \leftarrow \{0,1\}^\lambda$
   (b) Generate KAHE secret key $k_i \leftarrow \mathsf{KAHE.KGen}(\mathsf{CRS}_{\mathsf{KAHE}})$
   (c) Compute $\vec{s}_i \leftarrow H_{\mathsf{RO}}(\mathsf{rand}_i, j)$ for $j \in [\ell]$. Compute $\hat{\vec{x}}_i = \vec{x}_i + \vec{s}_i$
   (d) Compute $\vec{C}_i \leftarrow \mathsf{KAHE.Enc}(\mathsf{CRS}_{\mathsf{KAHE}}, k_i, \hat{\vec{x}}_i)$
   (e) Compute $(\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i \leftarrow \mathsf{STE.Enc}(\mathsf{ek}, tag, \mathsf{Committee}, t, k_i)$
   (f) Compute $(\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i \leftarrow \overline{\mathsf{STE.Enc}}(\mathsf{ek}, \overline{tag}, \mathsf{Committee}, t, \mathsf{rand}_i)$
   (g) Compute $\sigma_i \leftarrow \mathsf{MTS.Sign}(\mathsf{sk}_i^{sig}, n, tag, \vec{\mathsf{ct}}_2[6])$
   (h) Send $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i, \vec{C}_i)$ to the Server.
3. Server S upon receiving $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i, \vec{C}_i)$ from $i \in \mathcal{C}$:
   (a) Compute $\vec{C}^* = \sum_i C_i$
   (b) Compute $(\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3) = \sum_{i \in \mathcal{C}} \mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i$
   (c) Compute $\sigma^* = \mathsf{MTS.SignAggr}(\mathsf{CRS}, tag, \mathsf{ak}_{sig}, \{(\vec{\mathsf{ct}}_2[6])_i\}_{i \in \mathcal{C}})$
   (d) Send $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ to each client $P_j \in \mathsf{Committee}$
4. Each client $P_j \in \mathsf{Committee}$ does:
   (a) Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
   (b) If yes, compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$ and $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$ and return $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$ to the Server.
5. The server S upon receiving $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$ from each of the $P_j \in \mathsf{Committee}$ does:
   (a) Compute $[K]_T \leftarrow \mathsf{STE.DecAggr}(\mathsf{CRS}, \mathsf{ak}_{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \{\mathsf{part\text{-}dec}\}_{j \in \mathsf{Committee}})$
   (b) For $i \in [n]$: compute $\mathsf{rand}_i \leftarrow \overline{\mathsf{STE.DecAggr}}(\mathsf{CRS}, \mathsf{ak}_{enc}, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \{\overline{\mathsf{part\text{-}dec}}\}_{j \in \mathsf{Committee}})$ and compute $\vec{s}_i = H_{\mathsf{RO}}(\mathsf{rand}_i, j)$ for $j \in [\ell]$
   (c) Compute discrete log of $[K]$ and output $\sum_i \vec{x}_i = \mathsf{KAHE.Dec}(K, \vec{C}^*) - \sum_i \vec{s}_i$

Figure 7: Our Extended Secure Aggregation Protocol with KAHE

8.1 Computing Discrete Logarithm

Lastly, we briefly discuss how to compute the discrete logarithm directly. Instead of directly brute-force all possible solutions, which, as discussed above, takes $\tilde{O}(\sqrt{Mq})$ time, we could alternatively store all the possible solutions as a constant-time lookup table. This instead takes $\tilde{O}(|G| \cdot \sqrt{M} \cdot q))$ storage. Concretely, for the largest parameters ($M = 1024$) we are interested in, set $q = 3329$ (as used in Crystals-Kyber [BDK+18]), it takes $\leq 16 \cdot 256 \cdot \sqrt{1024} \cdot 3329 < 53$MB (where 32 guarantees the overflow probability is $< 2^{-128}$ given the Chernoff bound above). In this case, only 53MB of storage is needed to compute the discrete logarithm in *constant* time.

Alternatively, one can give a further tradeoff between storage and computation. Specifically, in the idea above, let the storage be S. One can reduce the storage to S/s at the cost of doing s multiplications (or elliptic-curve additions). The idea is simple: instead of storing all possible V values, divide all the V values into V/s chunks each with s values. Then, store the first value in each chunk. To compute discrete log for $y = g^x$, for compute $y, y \cdot g, \ldots, y \cdot g^{s-1}$. Then, one of these s values ($y \cdot g^i$ for $i \in [0, s-1]$) must be in the table. Then, we can use the table and i to recover x easily. This takes s multiplications. For example, to reduce the storage to under 10MB, only 6 multiplications are needed.

8.2 Federated Learning Specific Challenges

Below we highlight some additional challenges that have been identified in the federated learning literature. We discuss how we can extend our secure aggregation scheme to resolve these challenges.

Input Validation. In a poisoning attack, malicious clients craft updates to bias or degrade the global model. In Tacita, such risks can be mitigated by combining our cryptographic aggregation with established defenses. Optimization-based methods like FedOpt [RCZ+20] and Byzantine-robust rules like bRSA [LXC+19] are directly compatible, since Tacita outputs a standard aggregated update that can be post-processed without altering the protocol. Thus, the server can apply FedOpt to handle heterogeneity or bRSA to limit malicious influence. A full specification of these integrations is left to future work. Another option is to require clients to provide zero-knowledge proofs that updates are well-formed (e.g., bounded norm, consistent quantization), which we also leave to future investigation.

Model Inconsistency Attacks. In a model inconsistency attack [PFA22], a server sends different global models to different clients during updates. The main idea is to craft these models so that the updates from certain clients cancel out during aggregation, enabling the server to infer the inputs of a target client. In our protocol, this can be easily mitigated by leveraging the *tag*. We can enforce that the tag is deterministically generated from the global model, for instance, $tag = H(\text{iteration}|\text{model})$. Note that in our protocols, the aggregated ciphertext can be decrypted only if the same *tag* was used in each individual ciphertext; likewise, the aggregated signature will fail verification if the individual signatures were computed over different *tag*s. OPA [KP24] uses a similar approach in their protocols.

Table 3: Computation and Communication Overhead of Our Protocol

Measurement	N	**Client**			**Server (Aggregation)**			**Server (Decryption)**	**Committee**	
		Sign	Encrypt	Total	Sign	Encrypt	Total		Verify	Partial Decrypt
Computation Time	32	59.31ms	4.26ms	63.57ms	21.39ms	0.54ms	21.94ms	14.3ms	8.8ms	0.89ms
	64	0.17s	4.18ms	0.18s	36.59ms	1.09ms	37.68ms	22.3ms	8.8ms	0.89ms
	128	0.53s	4.22ms	0.53s	58.81ms	2.15ms	60.96ms	33.08ms	8.8ms	0.89ms
	256	1.93s	4.21ms	1.93s	0.10s	4.65ms	0.10s	56.05ms	8.8ms	0.89ms
	512	8.08s	4.21ms	8.08s	0.21s	8.68ms	0.21s	89.24ms	8.8ms	0.89ms
	1024	32.59s	4.20ms	32.59s	0.41s	16.39ms	0.42s	0.18s	8.8ms	0.89ms
Communication Size	-	**Client → Server**			**Server → Committee**			-	**Committee → Server**	
	-	0.42kB	1.32kB	1.74kB	0.83kB	1.32kB	2.15kB	-	0.18kB	

Note: Communication complexity is constant across all roles and does not grow with the number of clients.

9 Implementation and Evaluation

We implemented our secure aggregation protocol – including the Multi-Key Linearly Homomorphic Threshold Signature (MKLHTS) and the modified Silent Threshold Encryption (STE) scheme – in Rust. Our implementation builds on two existing open-source libraries:

- `hinTs` [1], which implements the Silent Threshold Signature scheme of Garg et al. [GJM+24];
- `silent-threshold-encryption` [2], which provides a baseline implementation of the original STE scheme from [GKPW24].

All benchmarks were run on a MacBook Air (Apple M2, 16GB RAM), using the BLS12-381 curve and compiled in release mode with optimizations enabled. We emphasize that our benchmarks reflect a research-grade implementation: no effort was made to optimize the underlying cryptographic libraries. As such, the results reflect protocol-level behavior and asymptotics rather than peak performance.

9.1 Evaluation Metrics

We evaluate the performance of our protocol along two dimensions: computation time and communication size. Our experiments vary both the number of clients and committee members, denoted N and M respectively. We assume the input length to be 1 (i.e. $\ell = 1$) throughout in our comparisons with previous work. We justify this via the argument that the only real change in terms of communication is that clients need to send ℓ extra KAHE ciphertexts and in terms of computation the server needs to aggregate these ciphertexts and decrypt them.

We record:

- Client-side cost: time to encrypt and sign the input;
- Server-side cost: time to aggregate ciphertexts and signatures, and to perform final decryption;
- Committee-side cost: time to verify the aggregate MKLHTS signature and compute a partial decryption;
- Communication cost: size of each message exchanged in the protocol.

9.2 Performance Overview

Table 3 summarizes the computation and communication costs of our protocol as N scales. We now highlight the key observations:

[1] https://github.com/hintsrepo/hints

[2] https://github.com/guruvamsi-policharla/batched-silent-threshold-encryption/tree/main/src/ste

Table 4: Computation overheads (OPA vs. Tacita). Abbreviations: C=Client, S=Server, Com=Committee. Tacita's higher per-client time is due to online hint generation; hints are deterministic and can be moved offline (expected $\sim 10\times$ reduction in online client time).

N	C		S		Com
	OPA	Tacita	OPA	Tacita	Tacita
100	28 ms	0.50 s	26 ms	50 ms	< 10 ms
500	27 ms	8.0 s	26 ms	200 ms	< 10 ms
1000	28 ms	32 s	26 ms	400 ms	< 10 ms

Table 5: Communication per direction (OPA vs. Tacita) with M=50. Sizes in kB (decimal). OPA kB values use calibration 1 ciphertext + signature $\approx$ 128 B. S $\rightarrow$ Com is per-committee communication.

Direction	N	OPA (kB)	Tacita (kB)
C$\rightarrow$S uses $64M+1$ elems.			
C $\rightarrow$ S	128 512 1024	400.1	**1.2**
S$\rightarrow$Com uses $64 \log N + \log N$ elems.			
S $\rightarrow$ Com	128 512 1024	39.4 50.6 56.3	**2.15**
Com$\rightarrow$S uses $\log N + 64$ elems .			
Com $\rightarrow$ S	128 512 1024	8.6 8.7 8.8	**0.18**

Client-side Cost. Client computation increases linearly with N, since each client generates hints for the full participant set. For $N = 1024$, the average client-side cost is 32.59s. This reflects the cost of generating s-hint and $(\mathsf{s} \cdot \mathsf{sk})$-hint from scratch for each run. However, these hints can be precomputed when the client knows its cohort in advance, reducing runtime to near-constant. Our implementation does currently not incorporate this optimization.

Notably, client computation time is dominated by MKLHTS signing. While this time is higher than that of OPA – which does not provide succinct aggregate signatures – we emphasize that our scheme achieves constant communication complexity and avoids any external verifier dependence. We present a more detailed comparison with OPA in Table 4, 5. We do not directly compare with Willow since their setting is different from ours (they use DKG and have other entities like Verifier).

Server-side Cost. Server runtime is divided into two phases: aggregation and final decryption. For $N = 1024$, aggregation requires 0.42s and decryption 0.18s. These costs grow modestly with N, reflecting the cost of aggregating BLS-based commitments and computing quotient polynomials. Server-side cost is comparable to that of OPA, despite the added structure of threshold signatures and verifiable input soundness.

Committee-side Cost. Committee operations are lightweight and constant across all values of N. Signature verification takes an average of 8.8ms, and computing a partial decryption takes 0.89ms. This efficiency is a direct consequence of our use of MKLHTS, which compresses all client attestations into a single verifiable signature. In contrast, OPA require the committee to process one message per client, resulting in logarithmic in N cost, since each committee member only receives $\log N$ number of ciphertexts from the server.

Communication Overhead. The communication cost in our protocol is constant in N for all roles. Clients send fixed-size ciphertexts and signatures, and the server sends a single aggregated message to the committee. Table 3 confirms that these message sizes do not scale with the number of clients. In prior protocols like OPA, committee-side communication is logarithmic in N, since each committee member is assigned $\log N$ number of clients. Our use of MKLHTS and ciphertext aggregation avoids this overhead.

We leave it to future work to improve performance through optimized cryptographic libraries and structured precomputation.

9.3 Other Related Work

Secure aggregation has been studied extensively in both cryptographic and federated learning settings. A wide range of protocols have been proposed with varying tradeoffs in setup assumptions, interactivity, trust models, and efficiency. Liu et. al. [LGY+22] and Kairouz et al [KMA+21] present surveys on applications of secure aggregation to federated learning.

Table 6 summarizes key qualitative distinctions across prior work. We do not compare solutions that require two or more servers such as ELSA [RSWP23], SAFEFL [GMS+23] or the distributed server setting. We categorize protocols based on whether they require per-instance setup, support non-interactive execution, rely on designated (persistent) committees, and whether they operate over arbitrary or small input domains.

Flamingo [MWA+23] does not require a designated committee since the committee members can proactively secret share and forward the shares to the next committee. This assumes some communication between the current committee members and future committee members. Alternatively, they could also run a setup in every instance and not have committee members forward any shares.

At a high level, our protocol differs from prior works in the following ways: (1) It achieves full non-interactivity for both clients and committee members, with only a single message per party. (2) It supports ephemeral, one-shot committees without requiring external verifiers or designated decryption parties. (3) It attains constant-size communication and constant committee-side computation, enabled by our use of a novel MKLHTS primitive.

Table 6: Qualitative comparison of aggregation protocols. Here Setup Type differentiates between protocols that require a setup (for e.g. DKG) to be done between clients or committee in each instance of the secure aggregation. Input Domain captures whether there are any limitations on the inputs. One-shot captures the property that clients and committee members need to speak only once, and finally Designated Committee considers the case when protocol require a fixed committee or if the committee can be randomly sampled from the clients themselves.

Work	Setup Type	Input Domain	One-Shot	Designated Committee
Bonawitz et al. [BIK+17]	Per-instance	Arbitrary	✗	–
Bell et al. [BBG+20]	Per-instance	Arbitrary	✗	–
SASH [LCY+22]	Per-instance	Arbitrary	✗	–
LERNA [LLPT23]	One-time	Arbitrary	✗	–
MicroSecAgg [GPS+24]	One-time	Small	✗	–
Flamingo [MWA+23]	One-time	Arbitrary	✗	✗
Behnia et al. [BRE+24]	Per-instance	Arbitrary	✗	✓
Willow [BCGL+24]	Per-instance	Arbitrary	✓	✓
OPA [KP24]	One-time	Arbitrary	✓	✗
This work	One-time	Small*	✓	✗

**Notes: In Section 8 we show to extend the input domain to arbitrary inputs as for [BCGL+24, KP24].*

10 Conclusion and Future Work

We presented a new secure aggregation protocol called TACITAthat is one-shot, robust to client dropouts, and achieves constant communication complexity without relying on designated verifiers or committees. Our construction is based on two key innovations: a succinct Multi-Key Linearly Homomorphic Threshold Signature (MKLHTS) scheme for enforcing input soundness, and a modified Silent Threshold Encryption (STE) protocol that enables ciphertext-bound threshold decryption. We formally prove security in the Universal Composability framework and support our design with an open-source implementation and benchmark evaluation.

An important direction for future work is to ensure client input validation. This is particularly important in the Federated Learning setting, where malicious clients can mount poisoning attacks

to skew the output of the aggregation. Improving the concrete efficiency of our implementation – particularly the client-side signing step – is another practical goal.

Acknowledgements

Varun Madathil would like to thank Antigoni Polychroniadou for discussions on OPA and the attacks to OPA and Willow.

References

[ABF24] Gaspard Anthoine, David Balbás, and Dario Fiore. Fully-succinct multi-key homomorphic signatures from standard assumptions. In *Annual International Cryptology Conference*, pages 317–351. Springer, 2024.

[AGL+17] Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. Qsgd: Communication-efficient sgd via gradient quantization and encoding. *Advances in neural information processing systems*, 30, 2017.

[AP19] Diego F Aranha and Elena Pagnin. The simplest multi-key linearly homomorphic signature scheme. In *International Conference on Cryptology and Information Security in Latin America*, pages 280–300. Springer, 2019.

[BBG+20] James Henry Bell, Kallista A Bonawitz, Adrià Gascón, Tancrède Lepoint, and Mariana Raykova. Secure single-server aggregation with (poly) logarithmic overhead. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 1253–1269, 2020.

[BCF+25] Jan Bormet, Arka Rai Choudhuri, Sebastian Faust, Sanjam Garg, Hussien Othman, Guru-Vamsi Policharla, Ziyan Qu, and Mingyuan Wang. Beast-mev: Batched threshold encryption with silent setup for mev prevention. *Cryptology ePrint Archive*, 2025.

[BCGL+24] James Bell-Clark, Adrià Gascón, Baiyu Li, Mariana Raykova, and Phillipp Schoppmann. Willow: Secure aggregation with one-shot clients. *Cryptology ePrint Archive*, 2024.

[BDK+18] Joppe Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. Crystals-kyber: a cca-secure module-lattice-based kem. In *2018 IEEE European symposium on security and privacy (EuroS&P)*, pages 353–367. IEEE, 2018.

[BIK+17] Keith Bonawitz, Vladimir Ivanov, Ben Kreuter, Antonio Marcedone, H Brendan McMahan, Sarvar Patel, Daniel Ramage, Aaron Segal, and Karn Seth. Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1175–1191, 2017.

[BRE+24] Rouzbeh Behnia, Arman Riasi, Reza Ebrahimi, Sherman SM Chow, Balaji Padmanabhan, and Thang Hoang. Efficient secure aggregation for privacy-preserving federated machine learning. In *2024 Annual Computer Security Applications Conference (ACSAC)*, pages 778–793. IEEE, 2024.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[CDG+18] Jan Camenisch, Manu Drijvers, Tommaso Gagliardoni, Anja Lehmann, and Gregory Neven. The wonderful world of global random oracles. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 280–312. Springer, 2018.

[DHMW23] Nico Döttling, Lucjan Hanzlik, Bernardo Magri, and Stella Wohnig. Mcfly: verifiable encryption to the future made practical. In *International Conference on Financial Cryptography and Data Security*, pages 252–269. Springer, 2023.

[FDR+25] Vivian Fang, Emma Dauterman, Akshay Ravoor, Akshit Dewan, and Raluca Ada Popa. Legolog: A configurable transparency log. *Cryptology ePrint Archive*, 2025.

[FFR24] Antonio Faonio, Dario Fiore, and Luigi Russo. Real-world universal zksnarks are non-malleable. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3138–3151, 2024.

[FKL18] Georg Fuchsbauer, Eike Kiltz, and Julian Loss. The algebraic group model and its applications. In *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II 38*, pages 33–62. Springer, 2018.

[FMNP16] Dario Fiore, Aikaterini Mitrokotsa, Luca Nizzardo, and Elena Pagnin. Multi-key homomorphic authenticators. In *International conference on the theory and application of cryptology and information security*, pages 499–530. Springer, 2016.

[GJM+24] Sanjam Garg, Abhishek Jain, Pratyay Mukherjee, Rohit Sinha, Mingyuan Wang, and Yinuo Zhang. hints: Threshold signatures with silent setup. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 3034–3052. IEEE, 2024.

[GKPW24] Sanjam Garg, Dimitris Kolonelos, Guru-Vamsi Policharla, and Mingyuan Wang. Threshold encryption with silent setup. In *Annual International Cryptology Conference*, pages 352–386. Springer, 2024.

[GMS+23] Till Gehlhar, Felix Marx, Thomas Schneider, Ajith Suresh, Tobias Wehrle, and Hossein Yalame. Safefl: Mpc-friendly framework for private and robust federated learning. In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 69–76. IEEE, 2023.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In Alfred V. Aho, editor, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229. ACM, 1987.

[GPS+24] Yue Guo, Antigoni Polychroniadou, Elaine Shi, David Byrd, and Tucker Balch. Microsecagg: Streamlined single-server secure aggregation. *Proc. Priv. Enhancing Technol.*, 2024(3):246–275, 2024.

[HLP11] Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*, pages 132–150. Springer, 2011.

[KMA+21] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and trends® in machine learning*, 14(1–2):1–210, 2021.

[KP24] Harish Karthikeyan and Antigoni Polychroniadou. Opa: One-shot private aggregation with single client interaction and its applications to federated learning. *Cryptology ePrint Archive*, 2024.

[KP25] Harish Karthikeyan and Antigoni Polychroniadou. Opa: one-shot private aggregation with single client interaction and its applications to federated learning. In *Annual International Cryptology Conference*, pages 319–353. Springer, 2025.

[LCG+23] Julia Len, Melissa Chase, Esha Ghosh, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. Elektra: Efficient lightweight multi-device key transparency. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2915–2929, 2023.

[LCG+24] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. {OPTIKS}: An optimized key transparency system. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4355–4372, 2024.

[LCY+22] Zizhen Liu, Si Chen, Jing Ye, Junfeng Fan, Huawei Li, and Xiaowei Li. Sash: Efficient secure aggregation based on shprg for federated learning. In *Uncertainty in Artificial Intelligence*, pages 1243–1252. PMLR, 2022.

[LGY+22] Ziyao Liu, Jiale Guo, Wenzhuo Yang, Jiani Fan, Kwok-Yan Lam, and Jun Zhao. Privacy-preserving aggregation in federated learning: A survey. *IEEE Transactions on Big Data*, 2022.

[LLPT23] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. Lerna: secure single-server aggregation via key-homomorphic masking. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 302–334. Springer, 2023.

[LPR13] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6), November 2013.

[LPS24] Helger Lipmaa, Roberto Parisella, and Janno Siim. Constant-size zk-snarks in rom from falsifiable assumptions. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–64. Springer, 2024.

[LTWC18] Russell WF Lai, Raymond KH Tai, Harry WH Wong, and Sherman SM Chow. Multi-key homomorphic signatures unforgeable under insider corruption. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 465–492. Springer, 2018.

[LXC+19] Liping Li, Wei Xu, Tianyi Chen, Georgios B Giannakis, and Qing Ling. Rsa: Byzantine-robust stochastic aggregation methods for distributed learning from heterogeneous datasets. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 1544–1551, 2019.

[MMR+17] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Artificial intelligence and statistics*, pages 1273–1282. PMLR, 2017.

[MTV+23] Varun Madathil, Sri Aravinda Krishnan Thyagarajan, Dimitrios Vasilopoulos, Lloyd Fournier, Giulio Malavolta, and Pedro Moreno-Sanchez. Cryptographic oracle-based conditional payments. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[MWA+23] Yiping Ma, Jess Woods, Sebastian Angel, Antigoni Polychroniadou, and Tal Rabin. Flamingo: Multi-round single-server secure aggregation with applications to private federated learning. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 477–496. IEEE, 2023.

[PFA22] Dario Pasquini, Danilo Francati, and Giuseppe Ateniese. Eluding secure aggregation in federated learning via model inconsistency. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2429–2443, 2022.

[PPS14] Alexandre Pinto, Bertram Poettering, and Jacob C.N. Schuldt. Multi-recipient encryption, revisited. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, page 229–238, New York, NY, USA, 2014. Association for Computing Machinery.

[RCZ+20] Sashank Reddi, Zachary Charles, Manzil Zaheer, Zachary Garrett, Keith Rush, Jakub Konečnỳ, Sanjiv Kumar, and H Brendan McMahan. Adaptive federated optimization. *arXiv preprint arXiv:2003.00295*, 2020.

[RSWP23] 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. IEEE, 2023.

[RZ21] Carla Ràfols and Arantxa Zapico. An algebraic framework for universal and updatable snarks. In *Annual International Cryptology Conference*, pages 774–804. Springer, 2021.

[Yao82] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

A Extended Preliminaries

A.1 Universal Composability

In UC security[Can01] we consider the execution of the protocol in a special setting involving an environment machine $\mathcal{Z}$, that chooses the inputs for the honest parties, and interacts with an adversary who is the party that participates in the protocol on behalf of dishonest parties. At the end of the protocol execution, the environment receives the output of the honest parties as well as the output of the adversary which one can assume to contain the entire transcript of the protocol. When the environment activates the honest parties and the adversary, it does not know whether the parties and the adversary are running the real protocol –they are in the real world, or they are simply interacting with the trusted ideal functionality, in which case the adversary is not interacting with any honest party, but is simply "simulating" to engage in the protocol. In the ideal world the adversary is therefore called simulator, that we denote by $\mathcal{S}$. To prove security we need to show that the environment cannot distinguish between the real and the ideal worlds.

Our protocols make use of the Global Random Oracle Functionality [CDG+18] defined in Figure 8

A.2 Silent Threshold Encryption

In this section, we present the definition of Silent Threshold Encryption (STE) and its properties, defined first in Garg et al. [GKPW24]. Our STE protocol, based on their work, has some important modifications in order to fit with our final secure aggregation scheme. We describe our protocol later in Appendix B. We present here our modified definitions, although we also use the standard STE scheme presented in [GKPW24] in our final protocol.

Parameters: output size $\ell(\lambda)$
Variables: initially empty list List_H
Functionality:

- Upon receiving $(\texttt{HASH-QUERY}, m)$ from some party P:
  1. Find h such that $(m, h) \in \mathsf{List}_H$. If no such entry exists, sample random $h \leftarrow \{0,1\}^{\ell(n)}$ and store (m, h) in List_H.
  2. Output $(\texttt{HASH-CONFIRM}, (m, h))$ to P.
- Upon receiving $(\texttt{PROGRAM-RO}, m, h)$ from adversary $\mathcal{A}$:
  1. If $\exists h' \in \{0,1\}^{\ell(n)}$ such that $(m, h') \in \mathsf{List}_H$ and $h \neq h'$ then $\texttt{abort}$.
  2. Else add (m, h) to List_H and output $(\texttt{PROGRAM-CONFIRM}, (m, h))$ to $\mathcal{A}$.

Figure 8: $\mathcal{G}_{\mathsf{RO}}$ ideal functionality

An STE scheme, informally, allows a set of parties to generate an encryption key ek independently from only a public key bulletin board, such that, if a party encrypts a message using ek and a parameter t of choice, then this message will only be decryptable if at least t of the parties in the bulletin board participate in the decryption.

Furthermore, the efficiency achieved in [GKPW24] is that both the size of ek, the size of the ciphertext and the size of partial decryptions are constant in the number of parties participating (generating ek requires linear time at a message independent and threshold independent setup phase). Finally, decrypting time is in total proportional to the threshold t. The scheme is heavily based on earlier work by Garg et al. [GJM+24]. We now present the interface for STE.

Definition 7 (Silent Threshold Encryption (modified from [GKPW24]))**.** *A Silent Threshold Encryption* (STE) *consists of a tuple of algorithms* $(\mathsf{STE.Setup}, \mathsf{STE.KGen}, \mathsf{STE.HintGen}, \mathsf{STE.Preprocess}, \mathsf{STE.Enc}, \mathsf{STE.PartDec}, , \mathsf{STE.Aggr}, \mathsf{STE.DecAggr})$ *with the following syntax.*

- $\mathsf{CRS} \leftarrow \mathsf{STE.Setup}(1^\lambda, M)$*: On input the security parameter* λ *and an upper bound* M *on the maximum number of users, the* ***Setup*** *algorithm outputs a common reference string* CRS*.*
- $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{STE.KGen}(1^\lambda)$*: On input the security parameter* λ*, the* STE.KGen *algorithm outputs a public/secret key pair* $(\mathsf{pk}, \mathsf{sk})$*.*
- $\mathsf{hint}_i \leftarrow \mathsf{STE.HintGen}(\mathsf{CRS}, \mathsf{sk}, M, i)$*: On input the* CRS*, the secret key* sk*, the number of parties* M*, and a position* $i \in [M]$*, the* STE.HintGen *algorithm outputs a hint* hint_i*.*
- $(\mathsf{ak}, \mathsf{ek}) \leftarrow \mathsf{STE.Preprocess}(\mathsf{CRS}, \mathcal{U}, \{\mathsf{hint}_i, \mathsf{pk}_i\}_{i \in \mathcal{U}})$*: On input the* CRS*, a universe* $\mathcal{U} \subseteq [M]$*, all pairs* $\{\mathsf{hint}_i, \mathsf{pk}_i\}_{i \in \mathcal{U}}$*, the* STE.Preprocess *algorithm computes an aggregation key* ak *and an encryption key* ek*.*
- $\mathsf{ct} \leftarrow \mathsf{STE.Enc}(\mathsf{ek}, tag, M, t)$*: On input an encryption key* ek*, a message* M*, a public tag, and a threshold* t*, it outputs a ciphertext* ct*.*
- $\mathsf{part\text{-}dec} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}, \mathsf{ct})$*: On input a secret key* sk*, and a ciphertext* ct*, the* STE.PartDec *algorithm outputs a partial decryption* part-dec*.*
- $b \leftarrow \mathsf{STE.PartVerify}(\mathsf{ct}, \mathsf{part\text{-}dec}, \mathsf{pk})$ *: On input ciphertext* ct*, partial decryption* σ *and public key* pk *return* 1 *iff the partial decryption verifies.*

1. Run $\mathsf{CRS} \leftarrow \mathsf{STE.Setup}(1^\lambda, M)$, pick random $[\gamma_2]$
2. Adversary outputs $\mathcal{U}^*, \mathsf{Cor} \leftarrow \mathcal{A}(\mathsf{CRS})$, the universe $\mathcal{U}^*$ and set of corrupted parties $\mathsf{Cor} \subset \mathcal{U}^*$.
3. For all $i \in \mathcal{U}^* \setminus \mathsf{Cor}$, let $(\mathsf{pk}_i \mathsf{sk}_i) \leftarrow \mathsf{STE.KGen}(1^\lambda)$, and $\mathsf{hint}_i \leftarrow \mathsf{STE.HintGen}(\mathsf{CRS}, \mathsf{sk}_i, M, i)$. Let $\mathsf{CRS}, \mathcal{U}^*$, and these public keys and hints become public parameters pp.
4. $\{(\mathsf{pk}_j, \mathsf{hint}_j)\}_{j \in \mathsf{Cor}} \leftarrow \mathcal{A}(\mathsf{pp})$. Append these outputs to pp.
5. Adversary picks two sets of messages to encrypt S_0, S_1 (to be a valid experiment, both sets must be of the same size (size $|\mathcal{U}^*| - |\mathsf{Cor}|$), and the sum of the messages in S_0 must be equal to the sum of the messages in S_1.
6. Challenger generates samples bit b uniformly, generates $|U| - |\mathsf{Cor}|$ honest ciphertexts encrypting the elements of S_b with tag $[\gamma_2]$ and outputs them to the adversary.
7. Adversary outputs at most $|\mathsf{Cor}|$ ciphertexts, along with the message and randomness used to generate the ciphertext.
8. Challenger verifies that the new ciphertexts output by the adversary are correct (if not, output 0). The challenger then aggregates all ciphertexts and outputs the aggregated ciphertext, along with the partial decryption for it.
9. Adversary outputs b', its guess of what set was encrypted.
10. Output 1 iff $|\mathsf{Cor}| < t$ AND $b' = b$.

Figure 9: Extended CPA Security game for STE.

- $\mathsf{ct}^{(agg)} \leftarrow \mathsf{STE.Aggr}(\mathsf{ek}, \vec{\mathsf{ct}})$: *On input encryption key and a vector of ciphertexts, output an aggregation of the ciphertexts which encrypts the sum of the messages, denoted as* $\mathsf{ct}^{(agg)}$.
- $M \leftarrow \mathsf{STE.DecAggr}(\mathsf{CRS}, \mathsf{ak}, \mathsf{ct}, \{\mathsf{part\text{-}dec}_i\}_{i \in S})$*: On input the* CRS*, an aggregation key* ak*, and a set of partial decryptions* $\{\mathsf{part\text{-}dec}_i\}_{i \in S}$*, the* STE.DecAggr *algorithm outputs a message* M.

In addition, the size of ak, ct*, and the partial decryption circuit should be independent of* M.

We now also present the CPA definition of security for STE. The security will basically guarantee semantic (CPA) security for any adversary corrupting at most $t-1$ parties (for a ciphertext generated with threshold t). Notice that the subset of parties is picked after setup but before picking messages. This is the same in [GKPW24].[3]

Definition 8 (CPA security for our modified STE)**.** *An STE scheme* STE *satisfies CPA security if for every* $M = \mathsf{poly}(\lambda)$ *and any PPT adversary* $\mathcal{A}$*, the output of the game Figure 9 is* 1 *with probability less than or equal to* $1/2 + \mathsf{negl}(\lambda)$.

In the correctness definition, the adversary can corrupt any number of parties and use them to maliciously generated incorrect partial decryptions. The correctness says that given enough correctly generated partial decryptions (independent of how many malicious partial decryptions were generated) an honest party will always correctly decrypt any honestly generated ciphertext. It will never decrypt to something different. This is defined formally below.

Definition 9 (Correctness for STE)**.** *An STE scheme* STE *satisfies correctness, if for any* $M = \mathsf{poly}(\lambda)$*, and any PPT adversary* $\mathcal{A}$*, the output of the game Figure 10 is* 1 *with probability less than or equal to* $\mathsf{negl}(\lambda)$.

[3] One can imagine a stronger version of the security game where the adversary can corrupt any $t-1$ parties adaptively, whenever it would like. This is not discussed in [GKPW24] and is also outside the scope of this work.

1. Experiment runs $\mathsf{CRS} \leftarrow \mathsf{STE.Setup}(1^\lambda, M)$.
2. Adversary outputs $\mathsf{Cor}, \mathcal{U}^* \leftarrow \mathcal{A}(\mathsf{CRS})$ where $\mathsf{Cor} \subseteq \mathcal{U}^*$ is the set of corrupted parties and universe, respectively.
3. For all $i \in \mathcal{U}^* \setminus \mathsf{Cor}$, let $(\mathsf{pk}_i \mathsf{sk}_i) \leftarrow \mathsf{STE.KGen}(1^\lambda)$, and $\mathsf{hint}_i \leftarrow \mathsf{STE.HintGen}(\mathsf{CRS}, \mathsf{sk}_i, M, i)$. Let $\mathsf{CRS}, \mathcal{U}^*$, and these public keys and hints become public parameters pp.
4. $\{(\mathsf{pk}_j, \mathsf{hint}_j)\}_{j \in \mathsf{Cor}} \leftarrow \mathcal{A}(\mathsf{pp})$. Append these outputs to pp.
5. Let $(\mathsf{ek}, \mathsf{ak}) \leftarrow \mathsf{STE.Preprocess}(\mathsf{pp})$. Adversary outputs the message and threshold $(M, t) \leftarrow \mathcal{A}^{\mathsf{STE.PartDec}(\cdot,\cdot)}_{\{sk_j : j \in \mathsf{Cor}\}}(\mathsf{pp}, \mathsf{ek}, \mathsf{ak})$, where part-dec is an oracle to partial decryption oracle of the honest parties given to the adversary.
6. Experiment outputs either $\mathsf{ct}\mathsf{STE.Enc}(\mathsf{ek}, M, t)$ or generates a set of K ciphertexts where for each $i \in [K]$, $\mathsf{ct}_i \mathsf{STE.Enc}(\mathsf{ek}, M_i, t)$ and outputs $\mathsf{ct} \leftarrow \mathsf{STE.Aggr}(\mathsf{ek}, \{\mathsf{ct}_i\}_{i \in [K]})$. In this case, let $M = \sum_{i \in [K]} M_i$.
7. The adversary outputs $\{\sigma_i\}_{i \in S} \leftarrow \mathcal{A}^{\mathsf{STE.PartDec}(\cdot,\cdot)}_{\{sk_j : j \in \mathsf{Cor}\}}(\mathsf{pp}, \mathsf{ek}, \mathsf{ak}, \mathsf{ct})$.
8. Let S' be the set $\{\sigma_i : \mathsf{STE.PartVerify}(\mathsf{ct}, \sigma_i, pk_j) = 1\}_{i \in S}$; where j, the partial decryptor, is included as part of σ_i. If $|S'| \geq t$, the experiment computes $M' \leftarrow \mathsf{STE.DecAggr}(\mathsf{CRS}, \mathsf{ak}, \mathsf{ct}, \{\sigma_i\}_{i \in S'})$.
9. Experiment outputs 1 iff $|S'| \geq t$ and $M' \neq M$.

Figure 10: Correctness game for STE.

B Proof of Security of the Modified Silent Threshold Encryption

Theorem 3. *In the Random Oracle model, the STE scheme (Appendix B) above satisfies the extended CPA security (Figure 9).*

Proof. We split the proof in two parts. First, we show computational indistinguishability between the game in Figure 11 and the game defined in Figure 9. Next, we show that given our STE scheme, no adversary can win Figure 11 except with negligible probability.

Indistinguishability between Figure 11 and Figure 9. Notice that fixing the randomness, the encryption algorithm becomes deterministic. Since the challenger will check the ciphertext and the given randomness and verify that the ciphertext was correctly generated from that randomness, the only difference between the game in Figure 9 and Figure 11 is that the adversary can pick the randomness and potentially try many different seeds. It can use this to potentially poison the aggregated ciphertext and have it so that the decryption of the aggregated ciphertext contains information about the decryption of some intermediate honest ciphertext output in step (6).

Suppose there exists a non-neglible set of sets of ciphertexts for which the aggregation of these and the honestly generated ciphertexts from step (6) followed by signatures on the $\mathsf{ct}_{2,6}$ of the aggregated ciphertext allows the adversary to decryption some honestly generated ciphertext from step (6). Then, it must be that for the game in Figure 11 there is a non-negligible chance that the challenger picks these ciphertexts in step 7. Then, it must be that either (a) the games are computationally equivalent or (b) the set of sets of ciphertexts which allow the adversary to win in either game is negligible. Notice for case (b), if the set of sets of ciphertexts is negligible, then, since our construction takes the adversary's input seed and passes it through a random oracle to generate the randomness for the ciphertext, the adversary cannot find any set of ciphertexts which would help in winning the game except with negligible probability. Then, in both cases, the games are computationally equivalent.

Given our STE scheme, no adversary can win Figure 11 except with negligible probability.

1. Suppose in step 5, adversary picks sets S_0 and S_1, where each set S_b be denoted by elements

1. Run $\mathsf{CRS} \leftarrow \mathsf{STE.Setup}(1^\lambda, M)$, pick random $[\gamma_2]$
2. Adversary outputs $\mathcal{U}^*, \mathsf{Cor} \leftarrow \mathcal{A}(\mathsf{CRS})$, the universe $\mathcal{U}^*$ and set of corrupted parties $\mathsf{Cor} \subset \mathcal{U}^*$.
3. For all $i \in U^* \setminus \mathsf{Cor}$, let $(\mathsf{pk}_i \mathsf{sk}_i) \leftarrow \mathsf{STE.KGen}(1^\lambda)$, and $\mathsf{hint}_i \leftarrow \mathsf{STE.HintGen}(\mathsf{CRS}, \mathsf{sk}_i, M, i)$. Let $\mathsf{CRS}, \mathcal{U}^*$, and these public keys and hints become public parameters pp.
4. $\{(\mathsf{pk}_j, \mathsf{hint}_j)\}_{j \in \mathsf{Cor}} \leftarrow \mathcal{A}(\mathsf{pp})$. Append these outputs to pp.
5. Adversary picks messages two sets of messages to encrypt S_0, S_1 (to be a valid experiment, both sets must be of the same size (size $|\mathcal{U}^*| - |\mathsf{Cor}|$), and the sum of the messages in S_0 must be equal to the sum of the messages in S_1.
6. Challenger generates samples bit b uniformly, generates $|U| - |\mathsf{Cor}|$ honest ciphertexts encrypting the elements of S_b with tag $[\gamma_2]$ and outputs them to the adversary.
7. Adversary outputs at most $|\mathsf{Cor}|$ messages to be encrypted.
8. Challenger encrypts messages by the adversary, and outputs the new ciphertexts, then aggregates all ciphertexts (including those from (5)) and outputs the aggregated ciphertext, along with the partial decryption for it.
9. Adversary outputs b', its guess of what set was encrypted.
10. Output 1 iff $|\mathsf{Cor}| < t$ AND $b' = b$.

Figure 11: Hybrid

$M_{b,1}, \ldots, M_{b,k}$, where $k = |S_b|$. (If it doesn't pick according to this specification, then the experiment is cancelled and the adversary loses.)

2. In step 6, the adversary gets back encryptions of the elements of S_b, for b picked by the challenger. We will denote these $\mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{(k)}$.

3. Note that up to step 8 in Figure 9, adversary cannot distinguish between set of ciphertexts, this follows from CPA security of the scheme in [GKPW24]. The ciphertexts it sees are exactly the same as the ciphertexts from before, and it has no additional information (decryption).

4. At step 8, the adversary receives the decryption of the aggregated ciphertext, $\mathsf{ct}^{(\mathsf{Agg})} = \mathbf{Aggr}(\mathsf{ct}^{(1)}, \ldots, \mathsf{ct}^{k+|Cor|})$. This decryption is a set of keys $\gamma := \{\mathsf{sk}_i \circ \mathsf{ct}_{2,6}^{(\mathsf{Agg})}\}_{i \in m}$. First, notice that for $k = 1$, the adversary can only send a single message and so by definition b is indistinguishable. So for now, we look at the case where $k \geq 2$.

5. Consider an equivalent hybrid where the challenger runs the entire encryption protocol beforehand, except the step where it adds the message (it preprocesses k tuples $\mathsf{ct}^{(i)} = (\mathsf{tag}, s^T \cdot A, s^T \cdot b)$ and when the adversary picks the messages to encrypt, it samples b and adds $M_{b,i}$ to $\mathsf{ct}^{(i)}[3]$ and outputs that. Clearly the adversary's view is equivalent to the current experiment.

6. Suppose, now, that γ which adversary receives at step 8 allows the adversary to distinguish between the set of messages encrypted.

7. Then, it must be that the distributions $D(\{\mathsf{ct}^{(j)}\}_{j \in [k]} | \gamma, b = 0) \not\approx D(\{\mathsf{ct}^{(j)}\}_{j \in [k]} | \gamma, b = 1)$. In other words, the conditional distributions of the intermediate ciphertexts output by the challenger, given the signature on the aggregated $\mathsf{ct}^{(\mathsf{Agg})}$ is computationally distinguishable for the cases $b = 0$ and $b = 1$.

   In both cases, the only change is to the terms $\mathsf{ct}^{(i)}[3]$, and the only terms related to the decryptions γ are $\vec{\mathsf{ct}}_2[2], \vec{\mathsf{ct}}_2[6]$, and ct_3. Let us denote $s^{(i)}$ the randomness sampled to encrypt ciphertext i. More sepcifically, rewriting the claim above, it must be that:

$$D(\{\vec{ct_2}^{(j)}[2], \vec{ct_2}^{(j)}[6], (s^T)^{(j)} \cdot b + M_{0,j}\}_{j\in[k]}|\gamma) \not\approx D(\{\vec{ct_2}^{(j)}[2], \vec{ct_2}^{(j)}[6], (s^T)^{(j)} \cdot b + M_{1,j}\}_{j\in[k]}|\gamma) \tag{4}$$

where

$$\vec{ct_2}[2] = s^{(j)}[1] \cdot [1]_2 + s^{(j)}[3] \cdot tag, \quad \vec{ct_2}[6] = s^{(j)}[3] \cdot [1]_1$$

and

$$\gamma = \left\{ \mathsf{sk}_i \cdot \left(\sum_{j\in k+|Cor|} s^{(j)}[3] \cdot [1]_1 \right) \right\}_{i\in m}.$$

8. Then, since the original scheme is CPA, there are only three cases for which the adversary can earn an advantage. (1) It can decrypt the aggregated ciphertext and learn information about the intermediary ciphertexts picked; (2) it can learn the secret keys from the partial decryptions provided; or (3) it can decrypt some intermediate ciphertext or learn partial information about intermediate ciphertexts given the partial decryptions of the aggregated ciphertext.

9. For (1), notice that γ allows for decryption of $\mathsf{ct}^{(\mathsf{Agg})}$. However, in both experiments ($b = 0, b = 1$), $\mathsf{ct}^{(\mathsf{Agg})}$ decrypts to the same sum (by definition, the sum of the set of messages is the same). Then, the decryption of $\mathsf{ct}^{(Agg)}$ gives no information of which set was picked. For (2) by the static DDH assumption [PPS14] it follows that each sk_i is not recoverable from $\mathsf{sk}_i \cdot \left(\sum_{j\in k+|Cor|} s^{(j)}[3] \cdot [1]_1\right)$.

10. Finally, for (3) assume there exists some $\mathcal{P} \subset [k]$ such that if we replace $[k]$ by $\mathcal{P}$ in Equation (4) the distributions in are computationally distinguishable.

    Notice that for any $\mathcal{P} \subset [k]$, the sum $\sum_{j\in[k]\setminus\mathcal{P}} s^{(j)}$ is uniformly distributed, and therefore γ is uniformly distributed with respect to any subset of ciphertexts. Then, it must be for any subset of $[k]$, the two distributions conditional on γ look the same. This is a contradiction and concludes our proof.

□

C Proof of Security of the MKLHTS scheme

C.1 Unforgeability

We prove that the protocol presented above is unforgeable under the definition presented in Definition 1.

Our proof strategy is as follows:

1. We first show that the signature scheme $\mathsf{MTS.Sign}(\mathsf{sk}_i, tag, M) = \mathsf{sk}_i \cdot (H(tag) + M)$ is an unforgeable signature scheme.

2. Next, we show that if the output signature σ^* is valid, then a threshold number of parties have participated, i.e. $|\mathcal{S} \cup A| > T$, where A is the set of parties corrupted by the adversary.

3. Finally, we show that if the adversary outputs such a valid signature then it cannot win the game via *type-2* or *type-3* forgery.

Lemma 3. *The signature scheme* $\mathsf{MTS.Sign}(\mathsf{sk}_i, tag, M) = \mathsf{sk}_i \cdot (H(tag) + M)$ *where* $M = [m]_2$ *is unforgeable under the co-CDH assumption.*

Game $\mathbf{G}^{\mathcal{A}_{\mathsf{alg}}}$

Game Execution

1. $x \leftarrow \mathbb{Z}_p$
2. $X = [x]_1$
3. $Q := \emptyset$
4. $(m^*, [\sigma^*]_{\vec{a}}) \leftarrow \mathcal{A}_{\mathsf{alg}}^{\mathcal{O}(\cdot),\mathsf{H}(\cdot)}(X)$

Signature Queries $\mathcal{O}(tag_j, m_i)$

1. $Q := Q \cup (tag_j, m_i)$
2. $\sigma_{ij} = x \cdot (\mathsf{H}(tag_j) + [m_i])$
3. Return σ_i

Hash Queries $\mathsf{H}(tag_i)$

1. $h_i \leftarrow \mathsf{H}(tag_i)$
2. Return h_i

Figure 12: Unforgeability Game of the signature scheme against an algebraic adversary

Lemma 4. *Under the discrete log assumption, the signature scheme* MTS.Sign *is EUF-CMA secure in the AGM + ROM model.*

Proof. Let $\mathcal{A}_{\mathsf{alg}}$ be the algebraic adversary. We first define the security game for unforgeability relative to this adversary.

As $\mathcal{A}_{\mathsf{alg}}$ is an algbraic adversary at the end of the game it will output a forgery σ^* on a message $(m*, tag^*)$ where $(tag^*, m^*) \notin Q$ together with an algebraic representation of σ^*, denoted $\vec{a} = (\hat{a}, a', \bar{a}_1, \ldots, \bar{a}_q, \tilde{a}_1, \ldots, \tilde{a}_q)$

Let $h_i = [r_i]_2$ and $\mathsf{H}(tag^*) = [r^*]_2$. The above equation is then equivalent to:

$$x(m^* + r^*) \equiv_p x(a' + \sum_i (r_i + m_i)\tilde{a}_i) + (\hat{a} + \sum_i r_i \bar{a}_i)$$

We will now describe the adversaries $\mathcal{B}^1_{\mathsf{alg}}$ and $\mathcal{B}^2_{\mathsf{alg}}$ that play in the discrete log game that will run $\mathcal{A}_{\mathsf{alg}}$ as a sub-routine. The two adversaries will simulate the game $\mathbf{G}$ to the adversary $\mathcal{A}_{\mathsf{alg}}$ in different ways.

Reduction $\mathcal{B}^1_{\mathsf{alg}}(Z = g^z)$

1. Set $X = Z$ and send X to $\mathcal{A}_{\mathsf{alg}}$
2. **Simulating hash queries**: Upon receiving a request $\mathsf{H}(tag_i)$
   (a) If $\mathsf{H}(tag_i) \neq \perp$, return $\mathsf{H}(tag_i)$
   (b) Else, sample $r_i \leftarrow \mathbb{Z}_p$
   (c) Set $h_i = [r_i]_2$
   (d) Return h_i.
3. **Simulating signing queries:** Upon receiving a request $\mathcal{O}(tag_i, m_i)$:
   (a) If $\mathsf{H}(tag_i) = \perp$, sample $r_i \leftarrow \mathbb{Z}_p$ and set $h_i = [r_i]_2$
   (b) Compute $\sigma_i = (r_i + m_i) \cdot X$
   (c) Return σ_i
4. **Output forgery:** Upon receiving a forgery $((m^*, tag^*), \sigma^*)$ from $\mathcal{A}$:
   (a) From the equation above we have:

$$z(m^* + r^*) \equiv_p z(a' + \sum_i r_i \tilde{a}_i) + (\hat{a} + \sum_i r_i \bar{a}_i)$$

(b) Output

$$z = (\hat{a} + \sum_i r_i \bar{a}_i)(m^* + r^* - a' - \sum_i (r_i + m_i)\tilde{a}_i)^{-1}$$

Simulation of $\mathcal{B}^1_{\mathsf{alg}}$ is perfect, and there is no loss in security.

Now we describe the reduction $\mathcal{B}^2_{\mathsf{alg}}$:

Reduction $\mathcal{B}^2_{\mathsf{alg}}(Z = g^z)$

1. Sample $x \leftarrow \mathbb{Z}_p$ $X = [x]$ and send X to $\mathcal{A}_{\mathsf{alg}}$
2. **Simulating hash queries**: Upon receiving a request $\mathsf{H}(tag_i)$
   (a) If $\mathsf{H}(tag_i) \neq \perp$, return $\mathsf{H}(tag_i)$
   (b) Else, sample $\hat{r}_i, b_i \leftarrow \mathbb{Z}_p$
   (c) Set $H_i = Z^{b_i} + [\hat{r}_i]$
   (d) Return H_i.
3. **Simulating signing queries:** Upon receiving a request $\mathcal{O}(tag_i, m_i)$:
   (a) If $\mathsf{H}(tag_i) = \perp$, sample $r_i \leftarrow \mathbb{Z}_p$ and set $H_i = [r_i]_2$
   (b) Compute $\sigma_i = x \cdot (H_i + [m_i])$
   (c) Return σ_i.
4. **Output forgery:** Upon receiving a forgery $((m^*, tag^*), \sigma^*)$ from $\mathcal{A}$:
   (a) Similar to the equation above we have:

$$(zb^* + \hat{r}^* + m^*)x \equiv_p a'x + x(\sum_i (m_i + \hat{r}_i + zb_i)\tilde{a}_i)$$

$$+(\hat{a} + \sum_i (zb_i + \hat{r}_i)\bar{a}_i)$$

   (b) Output

$$z = ((\sum_i (m_i + \hat{r}_i)\tilde{a}_i) - m^* - \hat{r}^* + a') \cdot (b^* - \sum_i b_i \tilde{a}_i)^{-1}$$

Analysis:

Let F denote the event that $(m^* + r^* - a' - \sum_i (r_i + m_i)\tilde{a}_i) \not\equiv_p 0$. Then we can show that

$$\mathbf{Pr}[\mathbf{G}^{\mathcal{B}^1_{\mathsf{alg}}}_{\mathsf{dlog}}] = \mathbf{Pr}[\mathbf{G}^{\mathcal{A}_{\mathsf{alg}}} = 1 | F]$$

and

$$\mathbf{Pr}[\mathbf{G}^{\mathcal{B}^2_{\mathsf{alg}}}_{\mathsf{dlog}}] \geq \frac{p-1}{p} \mathbf{Pr}[\mathbf{G}^{\mathcal{A}_{\mathsf{alg}}} = 1 | \neg F]$$

The case when $F = 1$ is quite straightforward, and the reduction $\mathcal{B}^1_{\mathsf{alg}}$ is able to output z as described above.

The case when $F = 0$, recall that

$$(m^* + r^* - a' - \sum_i (r_i + m_i)\tilde{a}_i) \equiv_p 0$$

Recall that in this case $r^* = (\hat{r}^* + zb^*)$ and $r_i = (\hat{r}_i + zb_i)$

This implies we can rearrange the above equation to get

$$z = ((\sum_i (m_i + \hat{r}_i)\tilde{a}_i) - m^* - \hat{r}^* + a') \cdot (b^* - \sum_i b_i \tilde{a}_i)^{-1}$$

This implies the reduction succeeds whenever $b^* \neq \sum_i b_i \tilde{a}_i$
which occurs with probability $1 - \frac{1}{p}$.
Hence

$$\mathbf{Pr}[\mathbf{G}^{\mathcal{B}^2_{\mathsf{alg}}}_{\mathsf{dlog}}] \geq \frac{p-1}{p} \mathbf{Pr}[\mathbf{G}^{\mathcal{A}_{\mathsf{alg}}} = 1 | \neg F]$$

□

Now we are ready to prove that if the signature is valid then $|\mathcal{S} \cup A| > T$, where T is the threshold and A is the set of parties corrupted by the adversary. We prove this via a sequence of lemmas as in hinTS [GJM+24] in the AGM model.

Lemma 5 (Similar to Lemma 3 in [GJM+24]). *Suppose at the end of the forgery game, $\mathcal{A}$ outputs (M^*, σ^*) such that $\mathsf{MTS.Verify}(\mathsf{aPK}, \sigma^*, M^*, tag^*) = 1$. Then with $1 - \mathsf{negl}(\lambda)$ probability, we can extract multivariate polynomials: $\mathsf{ParSum}(x), B(x), Q_1(x), Q_2(x), Q_x(x, \{\mathsf{sk}_i\}), Q^*_x(x, \{\mathsf{sk}_i\}), Q_Z(x, \{\mathsf{sk}_i\}), Q^1_x(x, \{\mathsf{sk}_i\}), Q^{1*}_x(x, \{\mathsf{sk}_i\}), Q^1_Z(x, \{\mathsf{sk}_i\}), Q^2_x(x, \{\mathsf{sk}_i\}), Q^{2*}_x(x, \{\mathsf{sk}_i\}), Q^2_Z(x, \{\mathsf{sk}_i\}), \mathsf{aSK}(x, \{\mathsf{sk}_i\})$ from $\mathcal{A}$ such that:*

$$\mathsf{ParSum}(\omega) = 0$$

$$B(\omega^{n+1}) = 0$$

$$\mathsf{ParSum}(x \cdot \omega) - \mathsf{ParSum}(x) = (W(x) - t \cdot L_{n+1}(x)) \cdot B(x) = Z(x) \cdot Q_1(x)$$

$$B(x) \cdot (1 - B(x)) = Z(x) \cdot Q_2(x)$$

$$SK(X) \cdot B(X) = \mathsf{aSK}(x, \{\mathsf{sk}_i\}) + Q_Z(X) \cdot Z(X) + Q_x(X) \cdot X$$

$$S(X) \cdot A(X) = \sum_{i \in [n]} m_i + Q^1_x(X) \cdot X + Q^1_Z(X) \cdot Z(X)$$

$$SK(X) \cdot S(X) = \sum_{i \in [n]} sk_i \cdot m_i + Q^2_x(X) \cdot X + Q^2_Z(X) \cdot Z(X)$$

Proof. This proof directly follows from the proof of Lemma 3 in hinTS [GJM+24]. □

Lemma 6 (Similar to Lemma 4 in [GJM+24]). *The terms in polynomials $Q_x(x, \{\mathsf{sk}_i\}), Q^2_x(x, \{\mathsf{sk}_i\})$ that depend on sk_i has degree $< \mathbb{H} - 2$ in terms of x.*

Proof. This proof directly follows from the proof of Lemma 4 in hinTS [GJM+24]. □

Lemma 7 (Similar to Lemma 5 in [GJM+24]). *For polynomial $B(x)$ it must hold that $\sum_{i=1}^{n} B(\omega^i) = t$*

Proof. This proof directly follows from the proof of Lemma 5 in hinTS [GJM+24]. □

Lemma 8 (Similar to Lemma 6 in [GJM+24]). *Let the sets $\mathcal{S}$ and A be defined as in the unforgeability game. If the signature verifies under aPK, that is,*

$$([\mathsf{SK}(\tau)]_1 \circ [S(\tau)]_2) + (\mathsf{aPK} \circ H(tag)) - (\sigma^* \circ [1]_2) =$$

$$([Q^2_Z(\tau)_1 \circ Z(\tau)]_2) + ([Q^2_x(\tau)]_1 \circ [\tau]_2)$$

then with $1 - \mathsf{negl}(\lambda)$ probability, the polynomial identity $\mathsf{aSK}(\{\mathsf{sk}_i\}) = \sum_{i \in \mathcal{S}} \mathsf{sk}_i \cdot v_i + v_0$ for some $\{v_i\}$.

Proof. This lemma states that aSK will depend only on the sk_i that have signed the message M_i and tag. In particular it cannot depend on other honest parties sk_i.

Observe that

$$([\mathsf{SK}(\tau)]_1 \circ [S(\tau)]_2) + (\mathsf{aPK} \circ H(tag)) - (\sigma^* \circ [1]_2) =$$
$$([Q_Z^2(\tau)_1 \circ Z(\tau)]_2) + ([Q_x^2(\tau)]_1 \circ [\tau]_2)$$

This implies $\sigma^* = \mathsf{aSK} \cdot (H(tag) \cdot M^*)$. Now the group elements related to $H(tag)$ and M^* the adversary sees are $\{\mathsf{sk}_i \cdot (H(tag) \cdot M_i)\}$. Now consider the following lemma from [FKL18] which says the following:

Let $[f_1(x_1, \ldots, x_\ell)], \ldots, [f_t(x_1, \ldots, x_\ell)]$ be a sequence of group elements (in either $\mathbb{G}_1$ or $\mathbb{G}_2$) given to an algebraic adversary $\mathcal{A}$ as input, where $x_1, ..., x_\ell \leftarrow \mathbb{F}$. Let (g_1, g_2, h_1, h_2) be the output of $\mathcal{A}$. If it holds that $(g_1 \circ h_1) = (g_2 \circ h_2)$, with $1 - \mathsf{negl}(\kappa)$ probability, the adversary $\mathcal{A}$ must know the corresponding polynomials $g_1(x_1, \ldots, x_\ell) = \sum_{i=1}^t \alpha_i \cdot f_i$, $h_1(x_1, \ldots, x_\ell) = \sum_{i=1}^t \beta_i \cdot f_i$, $g_2(x_1, \ldots, x_\ell) = \sum_{i=1}^t \gamma_i \cdot f_i$, $h_2(x_1, \ldots, x_\ell) = \sum_{i=1}^t \delta_i \cdot f_i$ such that

$$g_1(x_1, \ldots, x_\ell) \cdot h_1(x_1, \ldots, x_\ell) = g_2(x_1, \ldots, x_\ell) \cdot h_2(x_1, \ldots, x_\ell)$$

holds as a multivariate polynomial identity.

Thus the proof of this lemma is a consequence of the lemma from [FKL18].

□

Now let us suppose that the adversary outputs a signature σ^* such that it verifies under aPK. This implies the following polynomial equations are valid

$$SK(X) \cdot B(X) = \mathsf{aSK}(x, \{\mathsf{sk}_i\}) + Q_Z(X) \cdot Z(X) + Q_x(X) \cdot X$$

And as shown in Lemma 8, we know that $\mathsf{aSK} = \sum_{i \in \mathcal{S}^*} \mathsf{sk}_i \cdot v_i + v_0$.

Now we extract the set $\mathcal{S}'$ from $B(x)$. This is the set $\{i \in [n] : B(\omega^i) = 1\}$. Let $\mathsf{aSK}' = \sum_{i \in \mathcal{S}'} \mathsf{sk}_i$

Now there are also honestly sampled $Q'_x(x, \{\mathsf{sk}'\})$ and $Q'_Z(x, \{\mathsf{sk}'\})$ such that

$$SK(X) \cdot B(X) = \mathsf{aSK}' + Q'_Z(X) \cdot Z(X) + Q'_x(X) \cdot X$$

This implies

$$\mathsf{aSK} - \mathsf{aSK}' = (Q_Z - Q'_Z) \cdot Z(x) + (Q_x - Q'_x) \cdot x$$

Thus to forge a signature, the adversary computes polynomials Δ_1 and Δ_2 such that:

$$\mathsf{aSK} - \mathsf{aSK}' = \Delta_1 \cdot Z(x) + \Delta_2 \cdot x$$

Since the size of $\mathcal{S}'$ is atleast t, such that $t > \sum_{i \in \mathcal{S} \cup A}$, there exists atleast one party j such that $j \in \mathcal{S}'$ and $j \notin \mathcal{S}^*$.

This implies

$$\mathsf{aSK} - \mathsf{aSK}' = \sum_{i \in \mathcal{S}^*} \mathsf{sk}_i \cdot v_i + v_0 - \sum_{i \in \mathcal{S}'} \mathsf{sk}_i / |\mathbb{H}|$$
$$= L(\{\mathsf{sk}_i\}_{i \neq j}) + \mathsf{sk}_j / |\mathbb{H}|$$

where $L(\cdot)$ is an affine combination of $\{\mathsf{sk}_i\}_{i \neq j}$ and whose coefficients depend on $v_0, \{v_i\}$.

Now observe that by Lemma 6 we know that the terms in $\Delta_2(x)$ that depend on sk_j have degree $< |\mathbb{H}| - 2$. Therefore the terms in $\Delta_1 \cdot Z(x) + \Delta_2 \cdot x$ that depend on sk_j can never be of the form $c \cdot \mathsf{sk}_j$, and in particular cannot equal $\mathsf{sk}_j / |\mathbb{H}|$. Thus the polynomial identity,

$$\mathsf{aSK} - \mathsf{aSK}' = \Delta_1 \cdot Z(x) + \Delta_2 \cdot x$$

does not hold, which implies the adversary can compute a valid signature that does not meet the threshold only with negligible probability.

A *type 2* forgery. Now that we have proved that the adversary has produced a valid signature which meets the threshold, our next step is to prove that the computed output is valid. That is, the adversary is not able to compute a *type 2* forgery. More specifically, the adversary should not be able to output (M^*, σ^*), such that the signature verifies but $M^* \neq \sum_{i \in \mathcal{S}^*} M_i$.

Recall that since the signatures verify the following two equations hold:

$$([\mathsf{SK}(\tau)]_1 \circ [S(\tau)]_2) + (\mathsf{aPK}, H(tag)) - (\sigma^* \circ [1]_2) =$$
$$([Q_Z^2(\tau)_1 \circ Z(\tau)]_2) + ([Q_x^2(\tau)]_1 \circ [\tau]_2)$$

$$([A(\tau)]_1 \circ [S(\tau)]_2) - ([1]_1 \circ [M^*]_2) =$$
$$([Q_Z^1(\tau)_1 \circ Z(\tau)]_2) + ([Q_x^1(\tau)]_1 \circ [\tau]_2)$$

Previously we have proved that σ^* cannot be forged by the adversary. This implies that $S(x)$ encodes $\{m_i\}_{i \in \mathcal{S}^*}$.

This implies, there exist Q_x^{1*} and Q_Z^{1*} such that

$$([A(\tau)]_1 \circ [S(\tau)]_2) - ([1]_1 \circ \sum_i m_i \cdot [1]_2) =$$
$$[Q_Z^{1*}(\tau)]_1 \circ [Z(\tau)]_2) + (Q_x^{1*}(\tau)]_1 \circ [\tau]_2)$$

This implies

$$(Q_Z^{1*} - Q_Z^1) \cdot Z(x) + (Q_x^{1*} - Q_x^1) \cdot x = M^* - \sum_i m_i$$

Thus to compute a *type 2* forgery, the adversary needs to output two polynomials Γ_1 and Γ_2 such that

$$M^* - \sum_i m_i = \Gamma_1 \cdot Z(x) + \Gamma_2 \cdot x$$

By a similar argument as above, we can show that such a polynomial cannot hold, which implies that the adversary cannot compute a *type 2* forgery.

A *type 3* forgery. To prove that the adversary cannot win with a type 3 forgery we need to show that the adversary cannot output a valid σ_i on behalf of some honest party P_i. To this end, we need to show that the signature scheme $\mathsf{MTS.Sign}(\mathsf{sk}_i, tag, M) = \mathsf{sk}_i \cdot (H(tag) \cdot M)$ is an unforgeable signature scheme. Note that this directly follows from Lemma 3.

C.2 Extractability

Let us first recall the definition of extractability. A MKLHTS scheme is extractable if there exists a PPT algorithm $\mathcal{E}$, such that with overwhelming probability, whenever an algebraic adversary outputs a valid aggreagted signature σ^* and message M^*, which includes some corrupt parties partial signatures σ_i, the extractor can extract the messages m_i corresponding to these partial signatures m_i.

Concretely, the scheme ensures that if σ^* passes the MTS.Verify procedure but does not arise from honest calls to MTS.Sign for each corrupted signer, then $\mathcal{E}$ can still *extract* the corrupted signers' message inputs from the structure of σ^* itself. In particular, we will show that this can be done from the polynomial commitments used by MTS.SignAggr.

The adversary $\mathcal{A}$ receives the CRS. It can query honest signers for partial signatures on some messages m_i for $i \in \mathcal{H}$. For $i \in \mathcal{C} := \mathcal{S} \setminus \mathcal{H}$, it can produce *any* partial signatures $\sigma_i^\star$ on messages

$m_i^\star$ (possibly not matching the real sk_i). Finally, $\mathcal{A}$ outputs (M^*, σ^*), claiming the aggregator has combined the partial signatures $\{\sigma_i\}$ for $i \in \mathcal{S}$ into σ^* on $\sum_{i\in\mathcal{S}} m_i$.

Now recall that σ^* includes polynomial commitments to the polynomials $B(X), SK(X)$ and $S(X)$, where

1. $B(\omega^i) = 1$, if the i-th client's signature is aggregated.
2. $S(\omega^i) = m_i$, which is the input message of the i-th party
3. $SK(\omega^i) = \mathsf{sk}_i$ which is the secret key of the i-th party.

Now we can leverage the knowledge-soundness of the underlying polynomial commitment scheme to extract each of these polynomials. Recent work by Faonio et al [FFR24] and Lipmaa et al [LPS24] show how to extract the polynomial from the polynomial commitment given at least one evaluation. Thus there exist a polynomial time algorithm that allows extraction of the polynomials $B(X), S(X)$ and $SK(X)$.

Now computing $S(\omega^i)$ for $i \in \mathcal{C}$ outputs m_i for $i \in \mathcal{C}$. Now by Lemma 3, we can claim that the extracted signature must verify, and by the unforgeability (Type 2 forgery) of the underlying MKLHTS scheme we can show that the probability with which MTS.SignAggr outputs a different (M^*, σ^*) on the extracted inputs is negligible.

D Proof of Theorem 2

To prove security, we need to define a simulator. For an intuition of the proof please see the main body. Here we first present the formal description of the simulator in Figure 13, then we will present a proof by hybrids to show that the simulated world is indistinguishable from the real-world. In the simulator below we assume the server to be corrupted and less than a threshold number of clients and committee members to be corrupt.

Proof Intuition: Below we present an intuition on how we simulate each of the operations.

Setup and Key Generation: The simulator generates keys and hints for all honest parties, and receives the keys and hints from the corrupt parties.

Aggregating input: Here we must consider two cases: 1) when the server is malicious and 2) when the server is honest.

When the server is *honest*, the simulator receives the encrypted inputs from the corrupt parties. Since the simulator generated the secret keys for the honest clients in the committee, we assume an honest majority in the committee, the simulator can decrypt the inputs of the corrupt clients directly. To simulate, the interaction between the server and the committee, the simulator sends encryptions of zero on behalf of the server. Now, the simulator sends $(\texttt{AGG-INPUT}, x_i)$ for each corrupt client. The simulator then sends $\texttt{PROCEED}$ to the $\mathcal{F}_{\mathsf{agg}}$ functionality and outputs the aggregated sum received from the functionality.

When the server is *malicious*, the simulation is slightly trickier. When the simulator receives $(\texttt{AGG-INPUT}, P_i)$ from $\mathcal{F}_{\mathsf{agg}}$, the simulator samples two random values and encrypts them using $\mathsf{STE.Enc}$ and $\overline{\mathsf{STE.Enc}}$ and computes a signature and sends it to the server. Now on behalf of an honest committee member, receive aggregated ciphertexts and an aggregated signature. If the signature verifies, send a partial decryption on behalf of the honest committee member to the server. The simulator now sends $\texttt{PROCEED}$ to the $\mathcal{F}_{\mathsf{agg}}$ ideal functionality and learns the sum of the input of the honest parties. Now upon receiving a random oracle query for one of the honest parties' encrypted input, the simulator programs the random oracle such that the aggregated sum is equal to the sum of the honest parties.

Now through a sequence of hybrids we show that the simulated world and the real world are indistinguishable. We show in red what we change in each hybrid.

Hybrid 0: Let this be the real world. Specifically, the transcript includes:

The simulator $\mathcal{S}_{\mathsf{agg}}$

Simulating Registration of Honest Clients: Upon receiving $(\texttt{REGISTER}, P_i)$ from $\mathcal{F}_{\mathsf{agg}}$, run the **Key Generation** algorithms and publish $(\mathsf{pk}_i^{enc}, \mathsf{hint}_i^{enc}, \mathsf{pk}_i^{sig}, \mathsf{hint}_i^{sig})$.

Simulating Registration of Malicious Clients: Upon receiving $(\mathsf{pk}_j^{enc}, \mathsf{hint}_j^{enc}, \mathsf{pk}_j^{sig}, \mathsf{hint}_j^{sig})$ from the adversary on behalf of a corrupt party, send $(\texttt{REGISTER}, P_i)$ to the ideal functionality $\mathcal{F}_{\mathsf{agg}}$.

Simulating Input Phase: The simulator gets the set of clients involved in this session sid denoted as $\mathcal{P}_{\mathsf{sid}}$ from the environment. Let $\mathcal{M}_{\mathsf{sid}}$ denote the set of corrupt clients that provide inputs.

1. Upon receiving ek and vk from the server S on behalf of honest parties, check if ek corresponds to the encryption key for the parties that form the committee set, and check that vk corresponds to the verification key for the parties in the client set.
2. Upon receiving $\texttt{START-SESSION}$ from the functionality, the simulator responds with $\texttt{OK}$ if the two checks above output True, else output $\perp$
3. Upon receiving $(\texttt{AGG-INPUT}, P_i)$ from $\mathcal{F}_{\mathsf{agg}}$:
   (a) Sample a random string rand_i and compute $\overline{\mathsf{STE.Enc}}(\mathsf{ek}, \overline{tag}, \mathsf{ek}, t, \mathsf{rand}_i)$
   (b) Compute $\mathsf{STE.Enc}(\mathsf{ek}, tag, t, 0)$
   (c) Compute $\sigma_i \leftarrow \mathsf{MTS.Sign}(\mathsf{sk}_i^{sig}, \mathsf{ak}_{sig}, tag, \vec{\mathsf{ct}}_2[6])$
   (d) Send $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ to $\mathcal{A}$.
4. Upon receiving $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the server on behalf of an honest Committee member, P_j:
   (a) Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
   (b) Run the extractor of the MTS scheme and extract a set $\mathcal{D} = (\mathsf{sk}_i, s_i, \sigma_i)$ for $i \in \mathcal{M}_{\mathsf{sid}}$. If extraction fails abort with error message $\mathsf{ExtractionError}_1$. Else,
      - if the size of the extracted set $|\mathcal{D}| < t$, abort with error message $\mathsf{MTSFailure}_0$.
      - if there exists an σ_i that corresponds to an honest client's signature, abort with $\mathsf{MTSFailure}_1$.
      - If the aggregated sum of the inputs $\sum_{i \in \mathcal{P}_{\mathsf{sid}}} s_i$ is not equal to $\vec{\mathsf{ct}}_2[6]$ abort with $\mathsf{MTSFailure}_2$.
   (c) If not aborted, compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$ and $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$ and return $\mathsf{part\text{-}dec}_j$ to the adversary.

Simulating Output Phase: The simulator needs to program the random oracle so that it outputs the correct output to the server.

1. Send $(\texttt{CHOOSE-SET}, \mathcal{H}_{\mathsf{sid}})$ to the $\mathcal{F}_{\mathsf{agg}}$ ideal functionality (where $\mathcal{H}_{\mathsf{sid}}$ can be inferred from the set of honest contributions in $\mathcal{D}$) and receive X which is the sum of the inputs of the parties in $\mathcal{H}_{\mathsf{sid}}$.
2. *Program the random oracle*: Upon receiving a random oracle query rand_1 for an honest party's input P_1:
   - Send $(\texttt{PROGRAM-RO}, \mathsf{rand}_1, X)$ to $\mathcal{G}_{\mathsf{RO}}$ ideal functionality.
   - Send $(\texttt{PROGRAM-RO}, \mathsf{rand}_i, 0)$ for all $i \in \mathcal{H}_{\mathsf{sid}} \setminus \{1\}$ to $\mathcal{G}_{\mathsf{RO}}$ ideal functionality.

Figure 13: The simulator $\mathcal{S}_{\mathsf{agg}}$

1. The keys generated from Preprocessing of MTS and STE: $\mathsf{ak}_{sig}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ek}$
2. The ciphertexts and the signatures sent by the honest clients: $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ computed as
   (a) Sample $\mathsf{rand}_i \leftarrow \{0,1\}^\lambda$
   (b) Send $(\texttt{HASH-QUERY}, \mathsf{rand}_i)$ to $\mathcal{G}_{\mathsf{RO}}$ and get back s_i.
   (c) Compute $\hat{x}_i = x_i + s_i$
   (d) Compute $(\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i \leftarrow \mathsf{STE.Enc}(\mathsf{ek}, tag, t, \hat{x}_i)$
   (e) Compute $(\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i \leftarrow \overline{\mathsf{STE.Enc}}(\mathsf{ek}, \overline{tag}, t, \mathsf{rand}_i)$
   (f) Compute $\sigma_i \leftarrow \mathsf{MTS.Sign}(\mathsf{sk}_i^{sig}, \mathsf{ak}_{sig}, tag, \vec{\mathsf{ct}}_2[6])$
   (g) Send $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ to the Server (which is the $\mathcal{A}$).
3. Receive $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the adversary.
4. Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
5. On behalf of honest committee members output $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$:
   (a) Compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$
   (b) Compute $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$
   (c) Send $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$
6. The server (adversary $\mathcal{A}$) outputs the aggregated sum.

Hybrid 1: This hybrid is the same as the previous hybrid except that the simulator now extracts the signatures and messages from the aggregated signature. If the extractor does not terminate, abort with the error $\mathsf{ExtractionError}_1$.

1. The keys generated from Preprocessing of MTS and STE: $\mathsf{ak}_{sig}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ek}$
2. The ciphertexts and the signatures sent by the honest clients: $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ computed as in previous hybrid.
3. Receive $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the adversary.
4. Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
5. Run the extractor $\mathcal{E}$ of the MTS scheme. If the extractor fails in outputting $\mathcal{D}$, abort with the error message $\mathsf{ExtractionError}_1$
6. On behalf of honest committee members output $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$:
   (a) Compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$
   (b) Compute $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$
   (c) Send $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$
7. The server (adversary $\mathcal{A}$) outputs the aggregated sum.

From the proof of extractability of our MKLHTS scheme we know that the extractor runs in polynomial time, and will fail except with negligible probability. And therefore the **Hybrid 1** and **Hybrid 0** are indistinguishable.

Hybrid 2: This hybrid is the same as the previous hybrid, except that the simulator aborts if the size of the set $\mathcal{D}$ is less than the threshold t. Specifically:

1. The keys generated from Preprocessing of MTS and STE: $\mathsf{ak}_{sig}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ek}$
2. The ciphertexts and the signatures sent by the honest clients: $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ computed as in previous hybrid.
3. Receive $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the adversary.
4. Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
5. Run the extractor $\mathcal{E}$ of the MTS scheme. If the extractor fails in outputting $\mathcal{D}$, abort with the error message $\mathsf{ExtractionError}_1$
   - if the size of the extracted set $|\mathcal{D}| < t$, abort with error message $\mathsf{MTSFailure}_0$.
6. On behalf of honest committee members output $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$:
   (a) Compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$
   (b) Compute $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$
   (c) Send $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$
7. The server (adversary $\mathcal{A}$) outputs the aggregated sum.

Notice that the event $\mathsf{MTSFailure}_0$ occurs if and only if the adversary is able to forge a signature without t contributions. By unforgeability property of the underlying MKLHTS scheme the occurrence of this event is negligible, and therefore the **Hybrid 1 and Hybrid 2** are indistinguishable.

Hybrid 3: This hybrid is the same as the previous hybrid except that if there exists an extracted signature that corresponds to an honest party that was not signed by the simulator, abort with message $\mathsf{MTSFailure}_1$. Specifically:

1. The keys generated from Preprocessing of MTS and STE: $\mathsf{ak}_{sig}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ek}$
2. The ciphertexts and the signatures sent by the honest clients: $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ computed as in previous hybrid.
3. Receive $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the adversary.
4. Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
5. Run the extractor $\mathcal{E}$ of the MTS scheme. If the extractor fails in outputting $\mathcal{D}$, abort with the error message $\mathsf{ExtractionError}_1$
   - if the size of the extracted set $|\mathcal{D}| < t$, abort with error message $\mathsf{MTSFailure}_0$.
   - if there exists an $\sigma_i \in \mathcal{D}$ that corresponds to an honest client's signature, abort with $\mathsf{MTSFailure}_1$.
6. On behalf of honest committee members output $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$:
   (a) Compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$

(b) Compute $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$

(c) Send $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$

7. The server (adversary $\mathcal{A}$) outputs the aggregated sum.

Notice that this event occurs if and only if the adversary is able to forge a signature σ_i such that it corresponds to that of an honest party. This corresponds to exactly the Type 3 forgery in the HomUF-CMA game. Since we show that our MKLHTS scheme is HomUF-CMA secure, the probability that this event occurs is negligible and therefore **Hybrid 2** and **Hybrid 3** are indistinguishable.

Hybrid 4: This hybrid is the same as the previous hybrid except that if the sum of the extracted inputs in $\mathcal{D}$ is not equal to the aggregated sum $\mathsf{ct}_2[6]$, the simulator aborts with the error message $\mathsf{MTSFailure}_2$.

Specifically:

1. The keys generated from Preprocessing of MTS and STE: $\mathsf{ak}_{sig}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ek}$
2. The ciphertexts and the signatures sent by the honest clients: $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ computed as in previous hybrid.
3. Receive $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the adversary.
4. Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
5. Run the extractor $\mathcal{E}$ of the MTS scheme. If the extractor fails in outputting $\mathcal{D}$, abort with the error message $\mathsf{ExtractionError}_1$
   - if the size of the extracted set $|\mathcal{D}| < t$, abort with error message $\mathsf{MTSFailure}_0$.
   - if there exists an $\sigma_i \in \mathcal{D}$ that corresponds to an honest client's signature, abort with $\mathsf{MTSFailure}_1$.
   - If the aggregated sum of the inputs $\sum_{i \in \mathcal{P}_{\mathsf{sid}}} s_i$ is not equal to $\vec{\mathsf{ct}}_2[6]$ abort with $\mathsf{MTSFailure}_2$.
6. On behalf of honest committee members output $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$:

   (a) Compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$

   (b) Compute $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$

   (c) Send $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$
7. The server (adversary $\mathcal{A}$) outputs the aggregated sum.

As above this corresponds to a Type 2 forgery of the HomUF-CMA game. Since we prove that our MKLHTS scheme is HomUF-CMA unforgeable, the probability of this event occuring is negligible, and therefore the simulator does not abort except with negligible probability and therefore **Hybrid 3** and **Hybrid 4** are indistinguishable.

Hybrid 5: This hybrid is the same as the previous hybrid except that the random oracle outputs are programmed such that the output of the first honest party is set as $(\mathsf{rand}_1, \sum_{i \in \mathcal{H}_{\mathsf{sid}}} x_i - \hat{x}_i)$ and for all other rand_i it is set as $(\mathsf{rand}_i, 0)$. More specifically:

1. The keys generated from Preprocessing of MTS and STE: $\mathsf{ak}_{sig}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ek}$
2. The ciphertexts and the signatures sent by the honest clients: $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ computed as in previous hybrid.

3. Receive $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the adversary.
4. Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
5. Run the extractor $\mathcal{E}$ of the MTS scheme. If the extractor fails in outputting $\mathcal{D}$, abort with the error message $\mathsf{ExtractionError}_1$
   - if the size of the extracted set $|\mathcal{D}| < t$, abort with error message $\mathsf{MTSFailure}_0$.
   - if there exists an $\sigma_i \in \mathcal{D}$ that corresponds to an honest client's signature, abort with $\mathsf{MTSFailure}_1$.
   - If the aggregated sum of the inputs $\sum_{i \in \mathcal{P}_{\mathsf{sid}}} s_i$ is not equal to $\vec{\mathsf{ct}}_2[6]$ abort with $\mathsf{MTSFailure}_2$.
6. On behalf of honest committee members output $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$:
   (a) Compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$
   (b) Compute $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$
   (c) Send $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$
7. *Program the random oracle*: Upon receiving a random oracle query rand_1 for an honest party's input P_1:
   - Send $(\texttt{PROGRAM-RO}, \mathsf{rand}_1, (\mathsf{rand}_1, \sum_{i \in \mathcal{H}_{\mathsf{sid}}} x_i - \hat{x}_i))$ to $\mathcal{G}_{\mathsf{RO}}$ ideal functionality.
   - Send $(\texttt{PROGRAM-RO}, \mathsf{rand}_i, 0)$ for all $i \in \mathcal{H}_{\mathsf{sid}} \setminus \{1\}$ to $\mathcal{G}_{\mathsf{RO}}$ ideal functionality.
8. The server (adversary $\mathcal{A}$) outputs the aggregated sum.

Since the aggregated sum output by the adversary is the same as in the previous hybrid and the only difference is in the programming of the random oracle, **Hybrid 4** and **Hybrid 5** are indistinguishable.

Hybrid 6: This hybrid is the same as the previous hybrid except that the ciphertexts encrypted under STE.Enc are replaced by encryptions of 0.

1. The keys generated from Preprocessing of MTS and STE: $\mathsf{ak}_{sig}, \mathsf{vk}, \mathsf{ak}_{enc}, \mathsf{ek}$
2. The ciphertexts and the signatures sent by the honest clients: $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ computed as:
   (a) Compute $(\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i \leftarrow \mathsf{STE.Enc}(\mathsf{ek}, tag, t, 0)$
   (b) Compute $(\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i \leftarrow \overline{\mathsf{STE.Enc}}(\mathsf{ek}, \overline{tag}, t, \mathsf{rand}_i)$
   (c) Compute $\sigma_i \leftarrow \mathsf{MTS.Sign}(\mathsf{sk}_i^{sig}, \mathsf{ak}_{sig}, tag, \vec{\mathsf{ct}}_2[6])$
   (d) Send $((\mathsf{ct}_1, \vec{\mathsf{ct}}_2, \mathsf{ct}_3)_i, (\overline{\mathsf{ct}}_1, \vec{\overline{\mathsf{ct}}}_2, \overline{\mathsf{ct}}_3)_i, \sigma_i)$ to the Server (which is the $\mathcal{A}$).
3. Receive $(tag, \overline{tag}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3), \sigma^*)$ from the adversary.
4. Check $\mathsf{MTS.Verify}(\mathsf{CRS}, \vec{CT}_2[6], tag, \sigma^*, t, \mathsf{vk}) = 1$.
5. Run the extractor $\mathcal{E}$ of the MTS scheme. If the extractor fails in outputting $\mathcal{D}$, abort with the error message $\mathsf{ExtractionError}_1$
   - if the size of the extracted set $|\mathcal{D}| < t$, abort with error message $\mathsf{MTSFailure}_0$.
   - if there exists an $\sigma_i \in \mathcal{D}$ that corresponds to an honest client's signature, abort with $\mathsf{MTSFailure}_1$.

- If the aggregated sum of the inputs $\sum_{i \in \mathcal{P}_{\mathsf{sid}}} s_i$ is not equal to $\vec{\mathsf{ct}}_2[6]$ abort with $\mathsf{MTSFailure}_2$.

6. On behalf of honest committee members output $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$:
   (a) Compute $\mathsf{part\text{-}dec}_j \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, (\mathsf{CT}_1, \vec{CT}_2, \mathsf{CT}_3))$
   (b) Compute $\overline{\mathsf{part\text{-}dec}_j} \leftarrow \mathsf{STE.PartDec}(\mathsf{sk}_j^{enc}, \overline{tag})$
   (c) Send $(\mathsf{part\text{-}dec}_j, \overline{\mathsf{part\text{-}dec}_j})$
7. *Program the random oracle*: Upon receiving a random oracle query rand_1 for an honest party's input P_1:
   - Send $(\texttt{PROGRAM-RO}, \mathsf{rand}_1, (\mathsf{rand}_1, \sum_{i \in \mathcal{H}_{\mathsf{sid}}} x_i - \hat{x}_i))$ to $\mathcal{G}_{\mathsf{RO}}$ ideal functionality.
   - Send $(\texttt{PROGRAM-RO}, \mathsf{rand}_i, 0)$ for all $i \in \mathcal{H}_{\mathsf{sid}} \setminus \{1\}$ to $\mathcal{G}_{\mathsf{RO}}$ ideal functionality.
8. The server (adversary $\mathcal{A}$) outputs the aggregated sum.

Notice that in this hybrid, the aggregated ciphertext that is output by the adversary has a component $\mathsf{CT}_2[6]$. And from a previous hybrid the extractor $\mathcal{E}$ of the MKLHTS scheme allows for an extraction of each of the individual randomness that was used in computing the malicious ciphertexts. This matches with our extended CPA definition, By the CPA security of the underlying encryption scheme the two hybrids are indistinguishable.

Now **Hybrid 6** is the same as the simulated world, and therefore we have shown that through a sequence of hybrids the two worlds are indistinguishable. And this completes our proof.

E Manipulation of ciphertexts in TACITA

E.1 Background on STE

We first present a background on our modified STE protocol, we show exactly what constitutes the ciphertexts, and how the final decryption step works.

The silent threshold encryption protocol of Garg [GKPW24] is based on the following five pairing based equations:

1. $$[\mathsf{SK}(\tau)]_1 \circ [B(\tau)]_2 = [1]_2 \circ \mathsf{aPK} + [Z(\tau)]_2 \circ [Q_Z(\tau)]_1 + [\tau]_2 \circ [Q_x(\tau)]_1$$
2. $$[\tau]_2 \circ [Q_x(\tau)]_1 = [1]_2 \circ [\hat{Q}_x(\tau)]_1$$
3. $$[\gamma]_2 \circ \mathsf{aPK} = [1]_1 \circ \sigma^*$$
   (Here $\sigma^* = \mathsf{sk} \cdot [\gamma]$)
4. $$[\tau^t]_1 \circ [B(\tau)]_2 = [1]_2 \circ [\hat{B}(\tau)]_1$$
5. $$[1]_1 \circ [B(\tau)]_2 = [\tau - 1]_2 \circ [Q_0(\tau)]_1 + 1$$

Let us recall the main parts of the protocol.

- $\mathsf{Enc}(\mathsf{ek}, M, t)$: Sample $[\gamma]_2 \leftarrow \mathbb{G}_2$, parse $\mathsf{ek} := (C, Z)$,

$$\mathsf{ek} = \left(\Big[\sum_{i \in \mathcal{U}} \mathsf{sk}_i L_i(\tau) \Big]_1, [Z(\tau)]_2 \right) := (\mathcal{C}, Z).$$

set $Z_0 = [\tau - \omega^0]_2$ and set

$$\mathbf{A} = \begin{pmatrix} C & [1]_2 & Z & [\tau]_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & [\tau]_2 & [1]_2 & 0 & 0 & 0 \\ 0 & [\gamma]_2 & 0 & 0 & 0 & [1]_1 & 0 & 0 \\ [\tau^t]_1 & 0 & 0 & 0 & 0 & 0 & [1]_2 & 0 \\ [1]_1 & 0 & 0 & 0 & 0 & 0 & 0 & Z_0 \end{pmatrix}$$

$$\mathbf{b} = ([0]_T, [0]_T, [0]_T, [0]_T, [1]_T)^\top.$$

Sample a vector $\mathbf{s} \leftarrow (\mathbb{Z}_p^*)^5$ and output

$$\mathsf{ct} = \Big([\gamma]_2, \mathbf{s}^\top \cdot \mathbf{A}, \mathbf{s}^\top \cdot \mathbf{b} + M \Big).$$

- **DecAggr**$(\mathsf{CRS}, \mathsf{ak}, \mathsf{ct}, \{\sigma_i\}_{i \in \mathcal{S}})$ Set

$$\mathbf{w} = \Big(B, -\mathsf{aPK}, -Q_Z, -Q_x, \hat{Q}_x, [\gamma]_2, -\hat{B}, -Q_0 \Big)^\top$$

+

Recall that this is computed in the previous steps of **DecAggr** after aggregating the partial decryptions received from the decryptors.

Parse $\mathsf{ct} = (\mathsf{ct}_1, \mathbf{ct}_2, \mathsf{ct}_3)$ and output

$$M^* = \mathsf{ct}_3 - (\mathbf{ct}_2 \circ \mathbf{w} + \mathsf{part\text{-}dec}^*)$$

Recall that

$$\mathsf{part\text{-}dec}^* = \frac{1}{M+1} \Big(\sum_{i \in \mathcal{C}_1} B(\omega_i) \cdot \mathsf{part\text{-}dec}_i \circ \mathsf{ct}_1 \Big) - (\mathsf{ct}_{2,6} \circ \mathsf{ct}_1)$$

This can be represented as

$$\mathsf{part\text{-}dec}^* = (\mathsf{sk} \cdot s_3 [1]_1 \circ [\gamma]_2) - (s_3 [1]_1 \circ [\gamma]_2])$$

First let us open up $\mathbf{ct}_2$:

$$\mathbf{ct}_2 = \mathbf{s}^\top \cdot \mathbf{A}$$

$$= \begin{pmatrix} s_1 & s_2 & s_3 & s_4 & s_5 \end{pmatrix} \cdot \begin{pmatrix} C & [1]_2 & Z & [\tau]_2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & [\tau]_2 & [1]_2 & 0 & 0 & 0 \\ 0 & [\gamma]_2 & 0 & 0 & 0 & [1]_2 & 0 & 0 \\ [\tau^t]_1 & 0 & 0 & 0 & 0 & 0 & [1]_2 & 0 \\ [1]_1 & 0 & 0 & 0 & 0 & 0 & 0 & Z_0 \end{pmatrix}$$

Expanding the multiplication:

$$\mathbf{ct}_2 = \mathbf{s}^\top \mathbf{A} = \begin{pmatrix} s_1 C + s_4[\tau^t]_1 + s_5[1]_1, \\ s_1[1]_2 + s_3[\gamma]_2, \\ s_1 Z, \\ s_1[\tau]_2 + s_2[\tau]_2, \\ s_2[1]_2, \\ s_3[1]_1, \\ s_4[1]_2, \\ s_5 Z_0 \end{pmatrix}.$$

ct_3 is of the form:

$$\begin{aligned} \mathsf{ct}_3 &= \mathbf{s}^\top \cdot \mathbf{b} + M \\ &= \begin{pmatrix} s_1 & s_2 & s_3 & s_4 & s_5 \end{pmatrix} \cdot \begin{pmatrix} [0]_T \\ [0]_T \\ [0]_T \\ [0]_T \\ [1]_T \end{pmatrix} + M \\ &= s_5[1]_T + M \end{aligned}$$

Compute $\mathbf{ct}_2 \circ \mathbf{w} + \mathsf{part\text{-}dec}^*$:

$$\begin{pmatrix} (s_1 C + s_4[\tau^t]_1 + s_5[1]_1 \circ B) \\ +(s_1[1]_2 + s_3[\gamma]_2 \circ -\mathrm{aPK}) \\ +(s_1 Z \circ -Q_Z) \\ +(s_1[\tau]_2 + s_2[\tau]_2 \circ -Q_x) \\ +(s_2[1]_2 \circ \hat{Q}_x) \\ +(s_3[1]_1 \circ [\gamma]_2) \\ +(s_4[1]_2 \circ -\hat{B}) \\ +(s_5 Z_0 \circ -Q_0) \end{pmatrix} + \mathsf{part\text{-}dec}^*$$

Note that each line is a pairing. And the final result is the product of these pairings. We can break the pairings up as follows:

$$\begin{pmatrix} (s_1 C \circ B) + (s_4[\tau^t]_1 \circ B) + (s_5[1]_1 \circ B) \\ +(s_1[1]_2 \circ -\mathrm{aPK}) + (s_3[\gamma]_2 \circ -\mathrm{aPK}) \\ +(s_1 Z \circ -Q_Z) \\ +(s_1[\tau]_2 \circ -Q_x) + (s_2[\tau]_2 \circ -Q_x) \\ +(s_2[1]_2 \circ \hat{Q}_x) \\ + (s_3[1]_1 \circ [\gamma]_2) + \mathsf{sk}s_3[1] \circ [\gamma]_2 - s_3[1] \circ [\gamma]_2 \\ +(s_4[1]_2 \circ -\hat{B}) \\ +(s_5 Z_0 \circ -Q_0) \end{pmatrix} = \begin{pmatrix} (s_1 C \circ B) + (s_4[\tau^t]_1 \circ B) + (s_5[1]_1 \circ B) \\ +(s_1[1]_2 \circ -\mathrm{aPK}) + (s_3[\gamma]_2 \circ -\mathrm{aPK}) \\ +(s_1 Z \circ -Q_Z) \\ +(s_1[\tau]_2 \circ -Q_x) + (s_2[\tau]_2 \circ -Q_x) \\ +(s_2[1]_2 \circ \hat{Q}_x) \\ + s_3[1] \circ \mathsf{sk}[\gamma]_2 (= s_3[1] \circ \sigma^*) \\ +(s_4[1]_2 \circ -\hat{B}) \\ +(s_5 Z_0 \circ -Q_0) \end{pmatrix}$$

Rearranging these pairings we get:

$$\begin{pmatrix} (s_1 C \circ B) + (s_1[1]_2 \circ -\mathrm{aPK}) + (s_1 Z \circ -Q_Z) + (s_1[\tau]_2 \circ -Q_x) \\ +(s_3[1]_1 \circ \sigma^*) + (s_3[\gamma]_2 \circ -\mathrm{aPK}) \\ +(s_2[1]_2 \circ \hat{Q}_x) + (s_2[\tau]_2 \circ -Q_x) \\ +(s_4[1]_2 \circ -\hat{B}) + (s_4[\tau^t]_1 \circ B) \\ +(s_5 Z_0 \circ -Q_0) + (s_5[1]_1 \circ B) \end{pmatrix}$$

Taking out the common field elements s_i

$$\begin{pmatrix} s_1\Big((C \circ B) + ([1]_2 \circ -\mathsf{aPK}) + (Z \circ -Q_Z) + ([\tau]_2 \circ -Q_x)\Big) \\ +s_3\Big(([1]_1 \circ \sigma^*) + ([\gamma]_2 \circ -\mathsf{aPK})\Big) \\ +s_2\Big(([1]_2 \circ \hat{Q}_x) + ([\tau]_2 \circ -Q_x)\Big) \\ +s_4\Big(([1]_2 \circ -\hat{B}) + ([\tau^t]_1 \circ B)\Big) \\ +s_5\Big((Z_0 \circ -Q_0) + ([1]_1 \circ B)\Big) \end{pmatrix}$$

Now recall the five pairing equations presented at the beginning of this section as in [GKPW24]:

1. $$[\mathsf{SK}(\tau)]_1 \circ [B(\tau)]_2 = [1]_2 \circ \mathsf{aPK} + [Z(\tau)]_2 \circ [Q_Z(\tau)]_1 + [\tau]_2 \circ [Q_x(\tau)]_1$$

2. $$[\tau]_2 \circ [Q_x(\tau)]_1 = [1]_2 \circ [\hat{Q}_x(\tau)]_1$$

3. $$[\gamma]_2 \circ \mathsf{aPK} = [1]_1 \circ \sigma^*$$

4. $$[\tau^t]_1 \circ [B(\tau)]_2 = [1]_2 \circ [\hat{B}(\tau)]_1$$

5. $$[1]_1 \circ [B(\tau)]_2 = [\tau - 1]_2 \circ [Q_0(\tau)]_1 + 1$$

If the aggregation was done correctly we have:

$$\begin{pmatrix} s_1[0]_T \\ +s_3[0]_T \\ +s_2[0]_T \\ +s_4[0]_T \\ +s_5[1]_T \end{pmatrix} = s_5[1]_T$$

Thus

$$\mathbf{ct}_2 \circ \mathbf{w} + \mathsf{part\text{-}dec}^* = (s_5[1]_T)$$

Finally, the decryption step

$$\mathsf{ct}_3 - (\mathbf{ct}_2 \circ \mathbf{w} + \mathsf{part\text{-}dec}^*) = (s_5[1]_T + M) - (s_5[1]_T) = M$$

E.2 Proving that the adversary cannot control the decryption

Now that we have a background on STE, we show how the adversary cannot manipulate the ciphertexts.

The simplest approach to prevent manipulation might to be enforce the encryptors to sign every group element in $\mathbf{ct}_2$. This enforces the adversary to use the same $s_1, \ldots, s_5$ as the encryptors chose implying that it cannot modify the ciphertext such that it decrypts to some other value M'. However, we notice that we do not need to sign the full ciphertext, and only need to sign one group element in the ciphertext.

Below we first show why signing $ct_2[6]$ is the most efficient way to ensure that a threshold number of ciphertexts have been aggregated. Then we show why the adversary cannot manipulate the ciphertexts to decrypt to messages of their choice.

Recall that $\mathbf{ct}_2 \circ \mathbf{w} + \mathsf{part\text{-}dec}^*$ rearranges blockwise as

$$\mathbf{ct}_2 \circ \mathbf{w} + \mathsf{part\text{-}dec}^* = s_1\Phi_1 + s_3\Phi_3 + s_2\Phi_2 + s_4\Phi_4 + s_5\Phi_5,$$

where the s_3-block equals $\Phi_3 = ([1]_1 \circ \sigma^*) + ([\gamma]_2 \circ -\mathrm{aPK})$ and the s_5-block equals $\Phi_5 = (Z_0 \circ -Q_0) + ([1]_1 \circ B)$.

Then we can show that requiring the server to present a valid (*succinct, threshold*) MKLHTS aggregate signature *only* on the single G_1-element $\mathbf{ct}_2[6] = s_3[1]_1$ ensures that any accepting aggregate necessarily binds the s_3-block to an actual set of at least t client signers, and hence the server cannot fabricate the s_3-contribution without forging the MKLHTS.

Now, given that one of the ciphertexts ($\mathbf{ct}_2[6]$) is fixed, we need to show that the adversary cannot output new ciphertexts that decrypt to a message of its choice. We capture this notion via a property called No-Target Control (NTC). We define the NTC game below for a fixed index i of the ciphertext:

1. Generate $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(\lambda)$ and send pk to the adversary $\mathcal{A}$
2. $\mathcal{A}$ fixes a message distribution $\mathcal{D} \leftarrow \mathcal{M}$ and sends $\mathcal{D}$ to the challenger.
3. Sample $m \leftarrow \mathcal{D}$, and compute $\mathbf{c} \leftarrow \mathsf{Enc}(\mathsf{pk}, m)$ and send $\mathbf{c}$ to $\mathcal{A}$.
4. Receive a pair $(d, t) \in \mathcal{M} \times \mathcal{M}$ and $\mathbf{c}'$ from $\mathcal{A}$
5. $\mathcal{A}$ wins if
   (a) $\mathbf{c}'[i] = \mathbf{c}[i]$, and
   (b) $\mathsf{Dec}(\mathsf{sk}, \mathbf{c}') = t$ and $t = d + m$

Lemma 9. *Given a CPA-secure STE ciphertext $(ct_1, \mathbf{ct}_2, ct_3)$ encrypting a message m under public key pk and tag. If the 6th element of $\mathbf{ct}_2$ is fixed, then a PPT adversary does not win the NTC game except with negligible probability.*

Proof. To prove Lemma 9, we show that an adversary winning the NTC game can break the CPA security of the underlying STE scheme.

Reduction $\mathcal{B}$:

1. Run $\mathcal{A}$ to output a distribution $\mathcal{M}$
2. Choose $m_0, m_1 \leftarrow \mathcal{M}$, and submit (m_0, m_1) to the CPA challenger.
3. Receive a challenge ciphertext $\mathbf{c}$ for a hidden bit b.
4. Give $\mathbf{c}$ to the adversary
5. Receive $(d, t, \mathbf{c}')$. Set $\hat{m} = t - d$. Set $b' = 0$ if $m_0 = \hat{m}$ and $b' = 1$ if $m_1 = \hat{m}$. Else sample $b' \leftarrow \{0, 1\}$.
6. Send b' to the CPA challenger.

Advantage Analysis:

Let W be defined as the following event:

$$W := \{\mathcal{A} \text{ wins the NTC game } \wedge\, \mathsf{Dec}(\mathsf{sk}, \mathbf{c}') = t\}$$

Let $\Pr[W] = \epsilon$. Then

$$\Pr[b = b'] = \Pr[W] \cdot \Pr[b = b' \mid W] + \Pr[\neg W] \cdot \Pr[b = b' \mid \neg W]$$

As noted $\Pr[b = b' \mid W] = 1$, since $\hat{m} = m_b$ when W is true.

For the failure case $\neg W$, we have $\Pr[b = b' \mid \neg W] = \frac{1}{2}$
Thus we have

$$\Pr[b = b'] = \epsilon \cdot 1 + (1 - \epsilon) \cdot \frac{1}{2} = \frac{\epsilon + 1}{2}$$

And therefore

$$\mathbf{Adv}_{\mathsf{IND\text{-}CPA}}(\mathcal{B}) = \Pr[b = b'] - \frac{1}{2} = \frac{\epsilon}{2}$$

Equivalently,

$$\epsilon \leq 2 \cdot \mathbf{Adv}_{\mathsf{IND\text{-}CPA}}$$

Since the STE scheme is CPA secure, $\mathbf{Adv}_{\mathsf{IND\text{-}CPA}}$ is negligible, and therefore the advantage of the adversary winning the NTC game is negligible, and that concludes our proof.

□