

Tree-BTE: Batched Threshold Encryption using Additive Homomorphism

Amit Agarwal¹, Sourav Das¹, Babak Poorebrahim Gilkalaye¹, Benny Pinkas², and Peter Rindal¹

¹Category Labs

²Bar-Ilan University

Abstract

Batched threshold encryption (BTE) enables a committee of servers to jointly decrypt any chosen subset of ciphertexts from a large pool, while all remaining ciphertexts stay private. BTE is a key building block for encrypted mempools, where transactions are encrypted until block inclusion to mitigate maximal extractable value (MEV). We give a new tree-based BTE construction from two simple building blocks which support additive homomorphism: a symmetric-key encryption scheme that is homomorphic over key and message space, and a thresholdizable public-key encryption scheme that is homomorphic over message space. These building blocks can be instantiated under a wide variety of assumptions including DDH, DCR, LWR and even LWE for post-quantum security.

In our scheme, each ciphertext is associated with an index from an appropriate index space. We implement the DDH-based instantiation. Given an aggregated batch of 1,000 messages over a small 1024 sized index space, each server performs about 41 ms of single-thread computation, including a 100-of-300 threshold decryption. We also partially implement an RLWE-based post-quantum instantiation over a 2^{128} sized index space for negligible collision probability among indices. A batch of 1024 messages uses approximately 0.89 MB per ciphertext and 20 seconds of single-thread decryption per server.

1 Introduction

Batched threshold encryption (BTE) enables a committee of servers to jointly decrypt an arbitrarily chosen subset (batch) of ciphertexts from a large pool, while all remaining ciphertexts stay private [CGPP24, AFP25, BFOQ24, BCF⁺25]. The defining feature of BTE is scalability: the total server-to-server communication required for decryption grows sublinearly in the batch size and is independent of the overall pool size. This makes BTE a natural building block for encrypted mempools, where transactions are encrypted until they are included in a block. This is a key approach to mitigate maximal extractable value (MEV) [GBH25, LBM⁺]. Practical encrypted mempools operate under tight latency budgets. Reducing both the communication and the computation of batch decryption, while supporting large batch sizes, is therefore an important problem.

This note introduces a new method for designing BTE schemes that departs from current practices. Our new approach draws on prior work about building succinct encodings when garbling a Boolean circuit [AIKW13]. At a high-level, our construction is built generically from two building blocks: a symmetric encryption scheme which is message and key homomorphic, KMHE, and a threshold (public-key) message homomorphic encryption scheme TMHE. In our scheme, for each

batch, the committee performs exactly one threshold decryption of an aggregated TMHE ciphertext, hence making the communication cost independent of the batch size. If the TMHE is silent and/or weighted, our BTE construction will inherit these properties. In Section 2, we describe the building blocks and our construction.

In Section 3, we evaluate a DDH-based [AIKW13] and RLWE-based [BCGL⁺24] instantiations which were implemented primarily using LLMs. As such we consider the implementation as purely a proof of concept. For a batch size $B = 1000$ of 128-bit messages in a 1024-index domain, the DDH prototype generates 7.7 KB ciphertexts and requires about 41 ms of single-thread computation per server, including a 100-of-300 threshold decryption. These measurements use 2 GB of reusable public tables (for computing discrete logarithms of small exponents) and exclude batch construction and network latency. For the RLWE instantiation, we consider a batch size of $B = 1024$ messages over an exponentially large index space of 2^{128} . With honestly sampled keys and errors, we estimate 0.89 MB client ciphertexts and about 20 seconds of local batch-decryption work. We also evaluate larger parameters that accommodate worst-case sums of bounded keys and errors. By our estimates this lattice-based instantiation will perform vastly better than other theoretical lattice based BTE schemes [BLT25, XZC26].

2 Technical Overview

2.1 Cryptographic Ingredients

Our BTE scheme relies on the following two cryptographic primitives.

Ingredient 1: Key and Message Homomorphic Encryption (KMHE). Consider two abelian groups $\mathcal{M}$ and $\mathcal{K}$. The KMHE is a symmetric-key encryption scheme for message vectors in $\mathcal{M}^L$ for some constant length $L \geq 2$. Keys lie in $\mathcal{K}$. Adding a bounded¹ number of ciphertexts adds both their keys and their messages (as vectors):

$$\text{Dec}(k_1 + k_2, \text{Add}(\text{Enc}(k_1, m_1), \text{Enc}(k_2, m_2))) = m_1 + m_2.$$

For ease of illustration, we will assume that the key and message groups are the same, i.e., $\mathcal{K} = \mathcal{M}$ although our construction works even when they are different². For security, we need IND-CPA security only for a single message per key, since every key in our eventual BTE construction will be used to encrypt exactly one message. There exists constructions for such a KMHE scheme under different cryptographic assumptions such as DDH [AIKW13], LWR [KP24, LLPT23], DCR [LLPT23] and (Ring) LWE [AIKW13, KP24, BCGL⁺24]. Note that a one-time pad encryption scheme is not a KMHE as it only supports $L = 1$.

Ingredient 2: Thresholdizable Message Homomorphic Encryption (TMHE). It is a public-key encryption scheme with messages in an abelian group $\mathbb{G}$, whose secret key sk is shared among N parties with threshold t . Ciphertexts can be added together resulting in a new ciphertext encrypting the sum of underlying messages. Each party j can use its secret key share sk_j to compute a partial decryption share $\text{pd}_j := \text{PartDec}(\text{sk}_j, \text{ct})$ of a ciphertext, and any t such valid partial decryption shares can be combined to decrypt the ciphertext ct to yield the underlying message as $m := \text{Dec}(\text{ct}, \{\text{pd}_j\}_{j \in S})$, where S is any subset of parties of size t . We require the standard

¹We require at most B additions to support a batch of size B .

²When $\mathcal{K} \neq \mathcal{M}$, we will additionally require the existence of efficiently computable maps $\phi: \mathcal{K} \rightarrow \mathcal{M}$ and $\psi: \mathcal{M} \rightarrow \mathcal{K}$. These maps must preserve the group operation under the restriction of at most B additions.

IND-CPA security to hold even when the adversary corrupts fewer than t parties.

2.2 Protocol Overview

In our BTE construction, each ciphertext is generated w.r.t a uniformly random index $i \in \mathcal{D}$ from some domain $\mathcal{D}$ specified by the public parameters of the BTE scheme. We will start by describing a scheme with a domain which is polynomial sized (in the security parameter λ) and then describe a mechanism to extend it to super-polynomial domain sizes using a recursive tree-based approach. We summarize our approach in Figure 1.

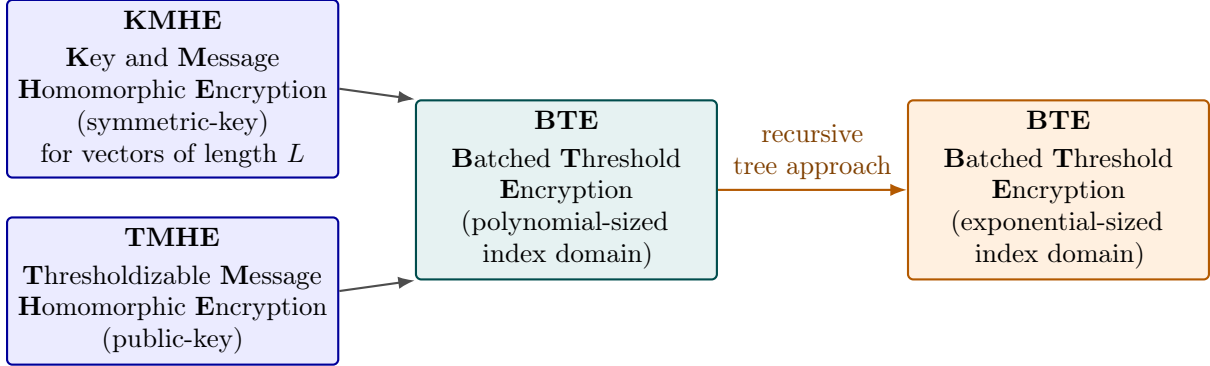


Figure 1: High-level overview of our approach. The two building blocks, KMHE and TMHE, give a BTE scheme for a polynomial-sized index domain (*warm-up*), which the recursive tree-based approach extends to an exponential-sized index domain.

Warm-up (polynomial-sized index domain). Suppose the domain size is $\mathcal{D} = L$. To encrypt a message m , we sample a random index $i \leftarrow \mathcal{D}$ and a KMHE key $k \leftarrow \text{KMHE.KeyGen}(1^\lambda)$ where λ is the security parameter. The ciphertext ct_i is then formed as follows:

$$\text{ct} := \begin{cases} i \\ \text{KMHE.Enc}(k, m \cdot \mathbf{e}_i) \\ \text{TMHE.Enc}(\text{pk}, k) \end{cases}$$

where $\mathbf{e}_i$ denotes a unit vector of length L with 1 in coordinate i (0-indexed), and 0 everywhere else. Thus, $m \cdot \mathbf{e}_i$ denotes the vector with m in coordinate i and 0 everywhere else.

Suppose we want to decrypt a batch of $B \leq L$ ciphertexts, $\text{ct}_0, \dots, \text{ct}_{B-1}$, where ct_i encrypts the message m_i using index i and KMHE key k_i . To do so, the servers add the TMHE components of the ciphertexts to derive a TMHE ciphertext ct_* which encrypts the sum of the keys, $K = \sum_{i=0}^{B-1} k_i$. The committee then performs a threshold decryption operation on ct_* to obtain K .

After this, each server can independently add the KMHE components of the ciphertexts to obtain a KMHE ciphertext $\text{ct}_\dagger$ which encrypts the sum of the vectors, i.e., the message $(m_1, \dots, m_B)$, under the aggregate key K . Using the key K , it is possible to decrypt $\text{ct}_\dagger$ to obtain $(m_1, \dots, m_B)$. This completes the decryption process for the batch.

Extending to an exponential-sized index domain. The issue with a polynomial-sized index domain is that it introduces opportunities for index collisions, whereas the underlying BTE con-

struction can only decrypt batches containing unique indices. This issue has also been highlighted in prior BTE works [BFOQ24, BLT25, ABD⁺25, ACADG26]. Next, we describe how we extend the aforementioned approach to support an exponential-sized index domain.

Consider a tree arity L and depth D for some integer L, D . We start by mapping each index i to the i^{th} leaf (0-indexed) of this tree. This implies a domain size of L^D equal to the number of leaves in the tree. Thus, for $L = O(1)$ and $D = O(\lambda)$ we get a domain size of $2^{O(\lambda)}O(2^\lambda)$. To encrypt a message m w.r.t index i , the encryptor samples D KMHE keys, $k^{(0)}, \dots, k^{(D-1)}$, one for each node along the path from the root to the i -th leaf. We will imagine the message m being at the leaf. We illustrate this in Figure 2.

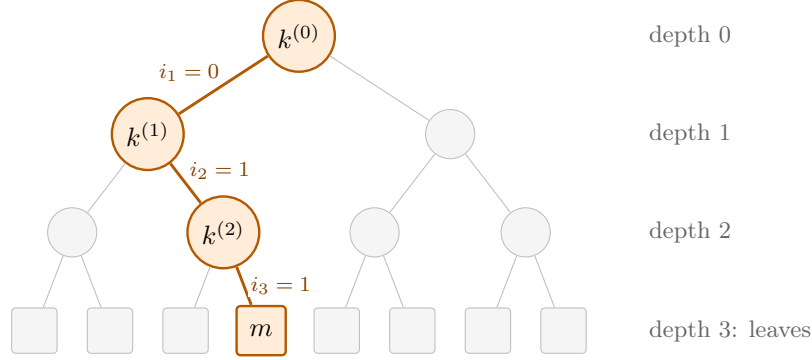


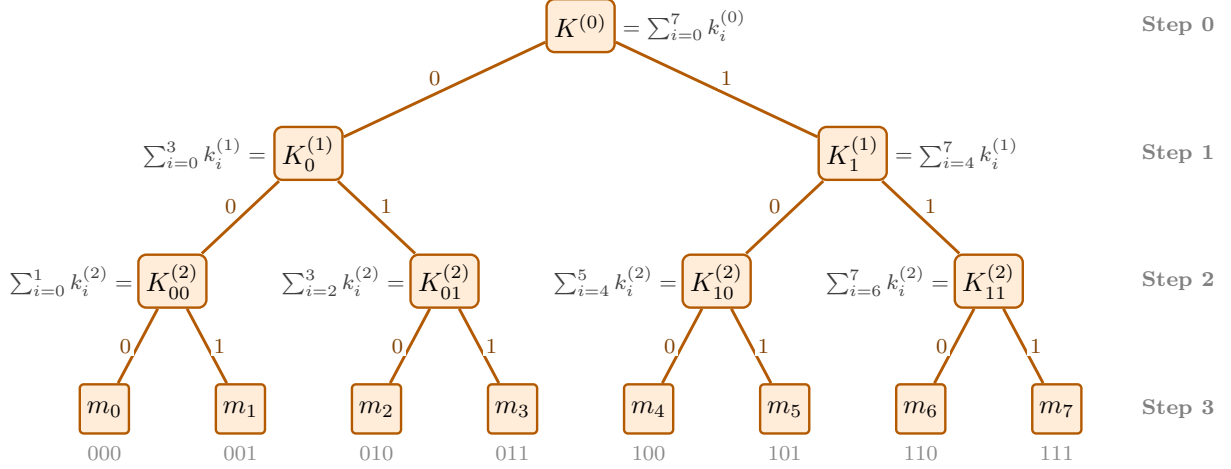
Figure 2: The tree for arity $L = 2$ and depth $D = 3$, from the point of view of a client at index $i = 3$, written in binary as $(i_1, i_2, i_3) = (0, 1, 1)$. The client samples keys $k^{(0)}, k^{(1)}, k^{(2)}$ for the nodes on the path to leaf 3 (highlighted). The root encrypts $k^{(1)}$ in coordinate $i_1 = 0$, the next node encrypts $k^{(2)}$ in coordinate $i_2 = 1$, and the last node encrypts the payload m in coordinate $i_3 = 1$; $k^{(0)}$ is encrypted with TMHE. The client’s ciphertext involves no other node (grey).

Let $(i_1, \dots, i_D)$ denote the base L representation of the index i , with i_1 and i_D being the most and least significant digits, respectively. Also, for each $j \in \{1, \dots, D\}$, let $\mathbf{e}_{i_j}$ denote the unit vector of length L with 1 in coordinate i_j and 0 elsewhere. We will form D independent KMHE ciphertexts $\text{ct}^{(1)}, \dots, \text{ct}^{(D)}$ as follows. For each $j \in \{1, \dots, D-1\}$, $\text{ct}^{(j)}$ encrypts the key $k^{(j)}$, encoded as $k^{(j)} \mathbf{e}_{i_j}$, under the key $k^{(j-1)}$. The leaf ciphertext $\text{ct}^{(D)}$ encrypts the message m , encoded as $m \cdot \mathbf{e}_{i_D}$, under the key $k^{(D-1)}$. Besides this, we will encrypt the root key $k^{(0)}$ under the committee’s public key using TMHE. The overall ciphertext ct will be of the following form:

$$\text{ct} = \begin{cases} i \\ \underbrace{\text{KMHE.Enc}(k^{(0)}, k^{(1)} \cdot \mathbf{e}_{i_1}), \dots, \text{KMHE.Enc}(k^{(D-2)}, k^{(D-1)} \cdot \mathbf{e}_{i_{D-1}})}_{\text{ct}^{(1)}}, \underbrace{\text{KMHE.Enc}(k^{(D-1)}, m \cdot \mathbf{e}_{i_D})}_{\text{ct}^{(D)}} \\ \text{TMHE.Enc}(\text{pk}, k^{(0)}) \end{cases}$$

Suppose we want to decrypt a batch of $B \leq L^D$ ciphertexts, $\text{ct}_0, \dots, \text{ct}_{B-1}$, where ct_i encrypts the message m_i using index i and KMHE keys $(k_i^{(0)}, \dots, k_i^{(D-1)})$. The servers will perform the following steps (see Figure 3 for an illustration):

- Step 0 (decrypting the root): The servers add the TMHE components of the ciphertexts to derive a TMHE ciphertext ct_* that encrypts the sum of the root keys, $K^{(0)} = \sum_{i=0}^{B-1} k_i^{(0)}$. The committee then performs a threshold decryption operation on ct_* to obtain $K^{(0)}$.



Step 0: threshold-decrypt $\text{ct}_* := \sum_{i=0}^{B-1} \text{TMHE.Enc}(\text{pk}, k_i^{(0)})$ to get $K^{(0)}$.

Step 1: locally decrypt and obtain $K^{(1)} = (K_0^{(1)}, K_1^{(1)})$ using $K^{(0)}$.

Step 2: locally decrypt and obtain $K_0^{(2)} = (K_{00}^{(2)}, K_{01}^{(2)})$ and $K_1^{(2)} = (K_{10}^{(2)}, K_{11}^{(2)})$ using $K_0^{(1)}$ and $K_1^{(1)}$, respectively.

Step 3: locally decrypt and obtain $(m_0, m_1), (m_2, m_3), (m_4, m_5), (m_6, m_7)$ using $K_{00}^{(2)}, K_{01}^{(2)}, K_{10}^{(2)}, K_{11}^{(2)}$, respectively.

Figure 3: Illustration of the merged tree of arity $L = 2$ and depth $D = 3$ corresponding to a full batch of $B = 8$ ciphertexts, $\text{ct}_0, \dots, \text{ct}_7$, where ct_i is formed w.r.t index i , binary decomposed as (i_1, i_2, i_3) , and message m_i . Each node holds the sum of the keys of the ciphertexts whose index path passes through it. Adding the KMHE ciphertexts below a node gives an encryption, under that node's key, of the vector of its children's keys. Servers decrypt the tree top-down starting from $K^{(0)}$.

- Step 1 (decrypting the first layer): Each server can independently add the first KMHE component of the ciphertexts in the batch, $\text{ct}_0^{(1)}, \dots, \text{ct}_{B-1}^{(1)}$, to obtain a KMHE ciphertext $\text{ct}_\dagger^{(1)}$ which encrypts the sum of the first-layer unit-vector encoded keys, $K^{(1)} = \sum_{i=0}^{B-1} k_i^{(1)} \cdot \mathbf{e}_{i_1}$, under the sum of the root keys $K^{(0)} = \sum_{i=0}^{B-1} k_i^{(0)}$. The server locally decrypts $\text{ct}_\dagger^{(1)}$ using $K^{(0)}$ to recover an L length vector $K^{(1)} := (K_0^{(1)}, \dots, K_{L-1}^{(1)})$ where

$$\forall \ell \in [0, L-1], \quad K_\ell^{(1)} = \sum_{i \in [0, B-1]: i_1 = \ell} k_i^{(1)}.$$

In other words, $K_\ell^{(1)}$ corresponds to the sum of the first-layer keys $k_i^{(1)}$ for all $i \in [0, B-1]$ whose most significant digit i_1 is ℓ .

- Step 2 (decrypting the second layer): For each $\ell \in [0, L-1]$, each server locally forms the following aggregate KMHE ciphertext:

$$\text{ct}_{\dagger, \ell}^{(2)} := \sum_{i \in [0, B-1]: i_1 = \ell} \text{ct}_i^{(2)}$$

which encrypts the sum of the second-layer unit-vector encoded keys under the corresponding first-layer aggregate key $K_\ell^{(1)}$. The server then locally decrypts $\text{ct}_{\dagger, \ell}^{(2)}$ using $K_\ell^{(1)}$ to recover a

vector $K_\ell^{(2)} := (K_{\ell 0}^{(2)}, \dots, K_{\ell(L-1)}^{(2)})$ where

$$\forall \ell' \in [0, L-1], \quad K_{\ell \ell'}^{(2)} = \sum_{i \in [0, B-1]: i_1=\ell, i_2=\ell'} k_i^{(2)}$$

In other words, $K_{\ell \ell'}^{(2)}$ corresponds to the sum of the second-layer keys $k_i^{(2)}$ for all $i \in [0, B-1]$ whose most significant and second most significant digits are $i_1 = \ell$ and $i_2 = \ell'$ respectively.

- Step d (decrypting layer d): The aforementioned process is repeated for each layer $d \in [D]$ until we reach the leaf layer, where the servers locally decrypt and obtain the payload messages m_i using the corresponding keys from the previous layer.

2.3 Analysis

Theorem 2.1 (informal). *Assuming an IND-CPA secure KMHE scheme for vectors of length $L = O(1)$ and an IND-CPA secure TMHE scheme for N servers, there exists an IND-CPA secure BTE scheme with depth parameter $D = \text{poly}(\lambda)$, batch size $B = \text{poly}(\lambda)$ and N servers with the following properties:*

- *Ciphertext size:* $D \times \text{CiphertextSize}_{\text{KMHE}} + \text{CiphertextSize}_{\text{TMHE}} = O_\lambda(1)$ for $D = O(\lambda)$
- *Communication cost (per server):* $N \times \text{PartialDecryptionShareSize}_{\text{TMHE}} = O_\lambda(1)$ for $N = \text{poly}(\lambda)$
- *Computation cost (per server):* $(B + \log_L \frac{L^D}{B}) \times \text{DecryptionTime}_{\text{KMHE}} + D \cdot B \times \text{KMHE.Add} + B \times \text{TMHE.Add} + \text{DecryptionTime}_{\text{TMHE}} = B \cdot \text{poly}(\lambda)$ for $D = O(\lambda)$

Security proof (informal). Consider a ciphertext that is not part of the batch. Its root key is encrypted only under TMHE, and no partial decryption of a sum containing it is ever released, so security of TMHE hides its root key. Once the root key is hidden, it is a key that encrypts a single message, namely the next key on the path, so security of KMHE hides that next key. Continuing down the path hides every key and, finally, the payload message.

Corollary 2.2 (informal). *Assuming an IND-CPA secure BTE scheme and a simulation-secure NIZK proof system, there exists an IND-CCA secure BTE scheme, obtained using standard techniques.*

3 Evaluation

We evaluate a proof-of-concept implementation of our approach where we use the encryption scheme from [AIKW13] as our underlying KMHE scheme, and threshold ElGamal as our underlying TMHE scheme. We develop our implementation using LLM assistance. Additionally, we also provide performance estimates for a (ring)-LWE-based construction based on the parameters from [BCGL⁺24].

3.1 DDH-based instantiation

Setup. The implementation supports arity $L \in \{2, 4, 8, 16\}$ with $L = 4$ resulting the best running time. We use depth $D = 5$ and thus the domain size is $L^D = 1024$.³ We use a batch consisting of $B = 1000$ ciphertexts encrypted with distinct indices, each encrypting a 128-bit message.

³In future, we plan to evaluate this for a domain size of 2^{128} or consider cuckoo-hashing based techniques to have small collision probability on the index space.

The TMHE secret key sk is secret shared among $N = 300$ servers with threshold $t = 100$. All measurements use a single thread on one core of an AMD Ryzen 9 7950X, compiled with GCC 15.2. We report local computation only, and exclude network latency as well as the generation of the public parameters, of sk , and of the shares sk_j .

Instantiation. We instantiate the DDH scheme over the ristretto255 group $\mathbb{H}$ and let $g \in \mathbb{H}$ be its generator. Our KMHE encryption scheme is a direct instantiation of [AIKW13]: a message x is encrypted under a key k as $H^k g^x$, where H is a public group element obtained by hash to group. We implement the Ristretto operations using eight-lane AVX-512 IFMA instructions. Decryption is performed by removing H^k and computing a bounded discrete logarithm to recover x from g^x and therefore x must be small, e.g. less than 2^{32} . Larger messages are supported by decomposing x into chunks and encoding each with a distinct H chosen via a hash to group modeled as a random oracle. For TMHE, we use threshold ElGamal encryption, $(g^r, \text{pk}^r g^x)$, where $\text{pk} = g^{\text{sk}}$ and sk is threshold secret-shared among the N servers. TMHE decryption follows the same discrete log approach. In our parameterization we therefore split the key and message space of KMHE into 22-bit chunks and encrypt each chunk separately. For $B = 1000$, the servers compute discrete logarithms whose exponents have at most 32 bits. For these parameters the implementation performs a total of 10,080 discrete logarithms in Steps 1 to D .

For the fixed parameter, a ciphertext is 7,732 bytes: 6,912 bytes for $\text{ct}^{(1)}, \dots, \text{ct}^{(D)}$, 768 bytes for the TMHE ciphertext, and 52 bytes for the index i and identifiers of the public parameters.

Batch aggregation. Upon selecting the batch, ciphertexts are deserialized and ciphertexts with an overlapping path KMHE in the tree are added together as described in Steps 0 to D of Section 2. The TMHE components are added to obtain ct_* . Deserializing and validating that the ciphertexts are on the curve took 118 ms. Adding the ciphertexts of the overlapping paths took only 10 ms. Most deserialization time is spent computing inverse square roots during Ristretto decoding. Clients can supply these values as hints, which the receiver checks using inexpensive field operations. This brings the deserialization time down to 12 ms, a $\sim 10\times$ reduction, but at the expense of almost doubling the size of the client ciphertext.

The aggregated batch contains one aggregate KMHE ciphertext per occupied internal node and one aggregate TMHE ciphertext. When a constant fraction of the leaves is occupied, this representation contains $O(B)$ node ciphertexts. The builder can broadcast it instead of the individual client ciphertexts. Assuming the aggregated batch is sent, we consider the optimization where the party building the batch also includes the “Ristretto inverse square root” hint in the batch. Computing these hints once took 4.8 ms. An aggregated batch with hints requires just 0.8 ms to deserialize and is 659 KB in size. Serializing an aggregated batch with hints required 5 ms. Without hints a batch requires 5.5 ms to deserialize and is 332 KB in size.

Server computation and communication. After broadcasting the aggregated batch, each server (1) computes and broadcasts its partial decryption share pd_j of ct_* , (2) receives and combines $t = 100$ shares to obtain $K^{(0)}$ (Step 0), (3) performs Steps 1 to D to recover the keys $K^{(1)}, \dots, K^{(D-1)}$ of each layer and finally the messages $m_0, \dots, m_{B-1}$. Table 1 reports the cost of each part. Steps 1 to D take about 21 ms, or about 21 μs per message.

Each server sends a single partial decryption share of 420 bytes, which includes an identifier of the batch and of the server; the $t = 100$ shares needed to obtain $K^{(0)}$ occupy 42 KB.

We measured three batches and, for each batch, executed the computation of all 300 servers; Table 1 reports medians over these 900 executions for the share computation and Steps 0 to D . The

Table 1: Single-thread computation (median, in milliseconds) of one server for the DDH-based instantiation on an AMD Ryzen 9 7950X, for a batch of $B = 1000$ ciphertexts with arity $L = 4$ and depth $D = 5$, and $N = 300$ servers with threshold $t = 100$. Client ciphertexts are sent without decoding hints, while the aggregated batch includes them. [†]The total excludes deserializing the client ciphertexts and aggregating the batch.

Operation	Median ms
Deserialize $B = 1000$ ciphertexts (w/o hints)	117.99
Batch aggregation	20.12
TMHE partial decryption	0.12
TMHE combine t shares (Step 0)	19.30
KMHE decryption (Steps 1 to D)	21.78
Total computation per server [†]	41.20

servers were executed one after another on the same machine, which may make these measurements optimistic. In every execution, the servers recovered the expected keys $K^{(j)}$ and messages m_i .

Our DDH-based KMHE decryption uses 2 GB of precomputed public data: 1.37 GB for a table that speeds up the discrete logarithms, and 0.63 GB that speeds up computing the powers H^K . It is generated once, in about 6.1 seconds, and reused for every batch; this cost is not included in the tables above. The chunk size trades this memory against ciphertext size and decryption time: with 16-bit chunks, a table of about 84 MB suffices, but the D KMHE ciphertexts $\text{ct}^{(1)}, \dots, \text{ct}^{(D)}$ of each ciphertext grow from 6,912 to 9,216 bytes, and more discrete logarithms are needed.

3.2 Lattice-based instantiation

We instantiate KMHE and TMHE with Willow’s RLWE-based KAHE and AHE schemes, respectively [BCGL⁺24]. Unlike our DDH based instantiation, for our RLWE-based instantiation, we use an exponential size domain of $L^D \geq 2^{128}$ and batches of $B = 1024$ ciphertexts.

We measure single-thread timings on an Apple M4 Max using Google’s implementation⁴ together with a prototype of our tree on top of it. Our prototype builds the tree of Section 2, aggregates a batch, and decrypts it top-down.

Schemes. For a power of two N and an integer q , let $R_q = \mathbb{Z}_q[X]/(X^N + 1)$. In the RLWE-based KMHE, a key is an element $k \in R_{q_1}$ whose N_1 coefficients are sampled from a discrete Gaussian distribution with standard deviation 4.5. Position $b \in [0, L - 1]$ of a node ciphertext encrypts $x_b \in R_{p_1}$ as $c_b = a_b \cdot k + p_1 \cdot e_b + x_b$, where $a_b \in R_{q_1}$ is public and e_b has discrete Gaussian coefficients with standard deviation 6.36; decryption under K returns $(c_b - a_b \cdot K) \bmod p_1$. Adding n ciphertexts and decrypting under the sum K of their keys leaves $p_1 \sum e + \sum x$: the term $a_b \cdot K$ cancels exactly, so the error depends on the number of ciphertexts added and not on K . The key and message groups differ, $\mathcal{K} = R_{q_1}$ and $\mathcal{M} = R_{p_1}$, and the maps of Section 2 are coefficient-wise: ϕ reduces a key modulo p_1 , and ψ lifts each coefficient to $(-p_1/2, p_1/2]$. For keys $k_1, \dots, k_n$ with $\|\sum_i k_i\|_\infty < p_1/2$, we have $\psi(\sum_i \phi(k_i)) = \sum_i k_i$. Decryption of a batch is therefore correct if

$$p_1 > 2 \max \left\| \sum k \right\|_\infty \quad \text{and} \quad q_1 > p_1 \left(2 \max \left\| \sum e \right\|_\infty + 1 \right),$$

⁴<https://github.com/google/secure-aggregation>, commit 24da9d2.

where the maxima range over the sums of at most B keys and errors that occur at a node. At the last level, the 128-bit message is split into $\lceil 128/\log_2 p_1 \rceil$ coefficients, and only these coefficients of each position are sent, since decryption never reads the others.

The **TMHE** is Willow’s threshold AHE over R_{q_2} with $N_2 = 4096$, ternary secrets, and flooding noise added to each partial decryption. The root key $k^{(0)}$ occupies the first N_1 of the N_2 plaintext coefficients, and the plaintext modulus p_2 exceeds $2 \max \|\sum k^{(0)}\|_\infty$. A **TMHE** ciphertext $(\text{ct}_0, \text{ct}_1)$ is sent with all N_2 coefficients of ct_1 , which the servers multiply by their key shares, and only the first N_1 coefficients of ct_0 ; a partial decryption pd_j is likewise truncated to N_1 coefficients.

Parameters. We consider two ways of bounding the sums above, for $B = 1024$. The *probabilistic* bound assumes that keys and errors are sampled from the prescribed distributions, so that a sum of n of them has standard deviation $\sqrt{n}$ times that of a single one, and requires the bounds to hold except with probability 2^{-40} over all coefficients of a batch. The *deterministic* bound only assumes that every key coefficient is at most 37 and every error coefficient at most 51 in absolute value (the samplers cut the distributions off at eight standard deviations) and adds B such values; it holds for every batch, including one that contains ciphertexts whose norms are only guaranteed by a proof, as in the CCA-secure variant of Corollary 2.2. Table 2 lists the resulting parameters. We estimate security with the lattice estimator [APS15], reporting both its core-SVP model and its full estimate; all parameter sets reach 128 bits under both.

Table 2: RLWE parameters $B = 1024$. For **KMHE**, N_1 is the ring degree, q_1 the ciphertext modulus, and p_1 the plaintext modulus. **TMHE** is Willow’s RLWE-based threshold scheme with ring degree $N_2 = 4096$, ciphertext modulus q_2 , and plaintext modulus p_2 . Security is in bits, estimated with the lattice estimator as core-SVP / full estimate.

Bound	KMHE				TMHE		
	N_1	$\log_2 q_1$	p_1	Security	$\log_2 q_2$	p_2	Security
Probabilistic	1024	26	2^{12}	129 / 153	76	$2^{13} + 1$	157 / 184
Deterministic	2048	35	2^{17}	204 / 226	80	$2^{17} + 1$	147 / 174

Ciphertext size. A ciphertext contains the index i , $D - 1$ internal **KMHE**, a last-level **KMHE** ciphertext of, and the **TMHE** ciphertext. Its size in bits is

$$|i| + (D - 1) L N_1 \log_2 q_1 + L \left\lceil \frac{128}{\log_2 p_1} \right\rceil \log_2 q_1 + (N_2 + N_1) \log_2 q_2.$$

The public elements a_b and the **TMHE** public key are derived from a seed and are not part of the ciphertext. Table 3 gives this size for $L = 4$ and $D = 64$. The internal **KMHE** ciphertexts account for over 94% of it, because every internal position carries a full child key of N_1 coefficients.

Computation. Table 4 reports the cost of each operation. The timings include the overhead of Willow’s implementation, which decodes plaintexts through 256-bit integers.

Choice of arity. Table 5 estimates costs for different arities, with $D = \lceil 128/\log_2 L \rceil$. Ciphertext size grows with $(D - 1)L$ and the number of occupied nodes in a batch is about $B(D - \log_L B)$, and the server decrypts L positions at each of them, so server time grows with L as well. Nodes of

Table 3: Client ciphertext size for 2^{128} leaves, $L = 4$, and $D = 64$, using the parameters of Table 2. The last-level component $\text{ct}^{(64)}$ keeps only the coefficients needed for a 128-bit message.

Bound	Index	$\text{ct}^{(1)}, \dots, \text{ct}^{(63)}$	$\text{ct}^{(64)}$	TMHE	Total
Probabilistic	16 B	838.7 KB	143 B	48.6 KB	887.5 KB
Deterministic	16 B	2.26 MB	140 B	61.4 KB	2.32 MB

Table 4: Single-thread operation times using parameters of Table 2. A KMHE node ciphertext has $L = 4$ positions, and decrypting an aggregated ciphertext costs the same as decrypting a single one.

[‡]Measured with $\log_2 q_2 = 96$ and two shares instead of $\log_2 q_2 = 80$ and 100 shares.

Operation	Probabilistic	Deterministic
KMHE key generation	55 μs	112 μs
KMHE encryption of one node	280 μs	560 μs
KMHE decryption of one node	330 μs	710 μs
TMHE encryption	490 μs	500 $\mu\text{s}^{\dagger}$
TMHE partial decryption pd_j	21.7 ms	22.2 ms [‡]
Combine $t = 100$ shares and recover $K^{(0)}$	450 μs	140 $\mu\text{s}^{\dagger}$

the same level are independent, so this work parallelizes across cores. Each server sends one partial decryption of 9.7 KB in all cases, independently of B , and aggregating a batch takes about 0.16 s.

Table 5: Estimated RLWE-based instantiation cost for 2^{128} leaves and $B = 1024$ for various arity L and depth $D = \lceil 128 / \log_2 L \rceil$. KMHE uses $N_1 = 1024$, a 26-bit q_1 , and $p_1 = 2^{12}$; TMHE uses $N_2 = 4096$ and a 76-bit q_2 ; these parameters assume that keys and errors are sampled from the prescribed distributions (failure probability at most 2^{-40} per batch). Ciphertext is the size of one BTE ciphertext. Client encryption and server decryption are single-thread times; server decryption time is per batch and includes threshold decryption and tree-traversal decryption time. We show the number of occupied nodes in the fifth column. The $L = 3$ row is extrapolated from the per-position costs measured for the other arities.

L	D	Ciphertext	Client encryption	Occupied nodes	Server decryption
2	128	894 KB	26 ms	120,944	20.5 s
3	81	847 KB	22 ms	76,365	19.4 s
4	64	887 KB	22 ms	60,216	20.4 s
8	43	1,167 KB	27 ms	40,316	27.3 s
16	32	1,700 KB	38 ms	29,838	40.4 s

With the deterministic bound and $L = 4$, we estimate 44 ms for client encryption and 43.0 s for server decryption per batch on one core.

4 Discussions

- Instantiations of KMHE

- The instantiation of KMHE used in prior works on secure aggregation [KP24, LLPT23, BCGL⁺24] need to rely on an extra leakage-resilience property to avoid leaking non-trivial information about the individual items that are being aggregated. This extra property is not needed for our BTE scheme as our goal is to reveal each item in the batch.
 - BEAT-MEV [BFOQ24] presents a BTE scheme for polynomial sized index space using key-homomorphic punctured PRF (KHPPRF). Each ciphertext ct_i in the batch has an independent KHPPRF key k_i and encrypts the message m_i w.r.t an index i from a polynomial sized index space. An interesting property of their construction is the following: Once the sum of all KHPPRF keys k_i in the batch is revealed, anyone can publicly recover all the individual messages m_i . Moreover, if two ciphertexts ct_i and ct_i^* have a colliding index i , then we instead end up recovering the sum of messages $m_i + m_i^*$. In essence, their BTE scheme for domain size L has an implicit KMHE scheme for vectors of length L with the useful property that the implied KMHE ciphertext size is independent of L . This property can be leveraged by instantiating our tree-based idea with a larger arity (e.g. 100 to 1000) and reducing the depth, thus leading to smaller overall BTE ciphertext size.
- Index space and collisions.
 - The encryption scheme is easier to define and analyze by relying on one security parameter, λ . This value sets both key lengths and the index space. When using a tree with constant arity and $O(\lambda)$ depth, the index space grows exponentially with λ which helps in keeping the index collision probability negligible. Some deployments may choose a smaller index range to reduce the ciphertext size, at the cost of accepting a small but non-negligible collision risk.
 - When using our BTE scheme for applications like encrypted mempool, where users send BTE ciphertexts towards a public mempool, we need to additionally defend against “index-copy” attacks. In such an attack, the attacker first observes the index i of a ciphertext ct_i generated by an honest user and then creates and sends a new ciphertext ct_i^* w.r.t the same index i to force trigger a collision. Having an exponential-sized index space only reduces inadvertent collision probability to negligible but doesn’t (directly) address such a maliciously generated collision. However, this can be handled using the techniques described in [AFP25, CGPP24].

References

- [ABD⁺25] Amit Agarwal, Kushal Babel, Sourav Das, Babak Poorebrahim Gilkalaye, Arup Mondal, Benny Pinkas, Peter Rindal, and Aayush Yadav. Weighted batched threshold encryption with applications to mempool privacy. Cryptology ePrint Archive, Paper 2025/2115, 2025.
- [ACADG26] Amit Agarwal, Rutchathon Chairattana-Apirom, Sourav Das, and Babak Poorebrahim Gilkalaye. Batched and weighted threshold encryption with silent setup. Cryptology ePrint Archive, Paper 2026/2087, 2026.
- [AFP25] Amit Agarwal, Rex Fernando, and Benny Pinkas. Efficiently-thresholdizable batched identity based encryption, with applications. In *Advances in Cryptology – CRYPTO 2025, Part III*, volume 16002 of *Lecture Notes in Computer Science*, pages 69–100. Springer, 2025.

- [AIKW13] Benny Applebaum, Yuval Ishai, Eyal Kushilevitz, and Brent Waters. Encoding functions with constant online rate or how to compress garbled circuit keys. In *Advances in Cryptology—CRYPTO 2013, Part II*, volume 8043 of *Lecture Notes in Computer Science*, pages 166–184. Springer, 2013.
- [APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 2015.
- [BCF⁺25] Jan Bormet, Arka Rai Choudhuri, Sebastian Faust, Sanjam Garg, Hussien Othman, Guru-Vamsi Policharla, Ziyang Qu, and Mingyuan Wang. BEAST-MEV: Batched threshold encryption with silent setup for MEV prevention. Cryptology ePrint Archive, Paper 2025/1419, 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, Paper 2024/936, 2024.
- [BFOQ24] Jan Bormet, Sebastian Faust, Hussien Othman, and Ziyang Qu. BEAT-MEV: Epochless approach to batched threshold encryption for MEV prevention. Cryptology ePrint Archive, Paper 2024/1533, 2024.
- [BLT25] Dan Boneh, Evan Laufer, and Ertem Nusret Tas. Threshold batch identity-based encryption without epochs. Cryptology ePrint Archive, Paper 2025/1254, 2025. Revised May 23, 2026.
- [CGPP24] Arka Rai Choudhuri, Sanjam Garg, Julien Piet, and Guru-Vamsi Policharla. Mempool privacy via batched threshold encryption: Attacks and defenses. In *33rd USENIX Security Symposium*, pages 3513–3529. USENIX Association, 2024.
- [GBH25] Pranav Garimidi, Joseph Bonneau, and Lioba Heimbach. On the limits of encrypted mempools. *a16z Crypto*, July 2025.
- [KP24] Harish Karthikeyan and Antigoni Polychroniadou. OPA: One-shot private aggregation with single client interaction and its applications to federated learning. Cryptology ePrint Archive, Paper 2024/723, 2024.
- [LBM⁺] Frederik Lührs, Luis Bezenberger, Francesco Mosterts, Sebastian Faust, and Andreas Erwig. The road towards an encrypted mempool on Ethereum. Shutter Docs.
- [LLPT23] Hanjun Li, Huijia Lin, Antigoni Polychroniadou, and Stefano Tessaro. LERNA: Secure single-server aggregation via key-homomorphic masking. Cryptology ePrint Archive, Paper 2023/1936, 2023.
- [XZC26] Saisi Xiong, Yijian Zhang, and Jie Chen. Efficient batched IBE from lattices in the standard model. *IACR Communications in Cryptology*, 3(2), 2026.