TreeOnion: Efficient Private Retrieval of Tree Paths

Pranav Shriram Arunachalaramanan*, Chenyang Zhang*, Yue Chen, Ling Ren
University of Illinois Urbana-Champaign
{psa3, zcy331, yuec12, renling}@illinois.edu

***Abstract*—Private information retrieval (PIR) lets clients retrieve entries from a public database hosted on a server without revealing which entries are retrieved. Traditionally, the database is assumed to be an array or a key-value store, and the client has the index or keyword of the entry it wishes to retrieve. However, in practice, the database can have a more complex structure. An important example is when the database is organized as a tree, and the client wants to retrieve all nodes along a root-to-leaf path. Notably, retrieving a Merkle proof is a natural instance of this problem, as is done in applications such as certificate transparency and blockchain systems.**

We present TreeOnion, a single-server scheme that allows clients to privately retrieve a path of nodes from a public database organized as a tree. For small node sizes, the communication and computation costs of TreeOnion are comparable to the costs of privately retrieving a single node from the same database. Compared with alternate approaches, TreeOnion is strictly better in both communication and computation, and is at least an order of magnitude better in one of the two metrics.

1. Introduction

Private information retrieval (PIR) is a cryptographic protocol that allows clients to retrieve entries from a public database hosted on a server *without* revealing any information about *which* entries are retrieved. PIR has been deployed or proposed for various applications, such as caller ID lookup and URL filtering [1], anonymous communication [2], [3], [4], secure advertising [5], [6], [7], [8], password check [9], safe browsing [10], privacy-preserving maps [11], and many more.

Traditional PIR treats the database as an array or a key-value store, and assumes the clients have the indices or keywords of the entries they wish to retrieve. However, in some applications, the database may have a more complex structure. For instance, the database may be a tree, and a client wishes to retrieve all entries on a root-to-leaf path in the tree. A prominent application of such a tree path retrieval is to retrieve authenticated data from a *Merkle tree* [12]. In particular, in addition to retrieving a data entry, the client also retrieves the path of *hashes* to that data entry to verify its integrity. (Sibling hashes are swapped in this Merkle tree; see Section 5.)

*. Equal contribution.

There are several straightforward approaches to the private tree path retrieval problem. For instance, the server could add the entire path to each leaf. In other words, the server treats each path as an entry. Then, the problem reduces to a standard single-query PIR. We call this the *path-per-leaf* approach. The downside of this approach is that it blows up the database size by a factor of $\log N$, where N is the number of leaves in the tree. As a consequence, the server must scan this larger database, increasing the server computation.

Another straightforward approach is to treat each level of the tree as a separate database. Then, the client retrieves one entry from each level using a standard single-query PIR scheme [13], [14], [15], [16]. We call this the *level-wise* approach. The clear downside of this approach is that the client needs to send a (encrypted) location for each level, blowing up the communication.

Alternatively, one can also put all tree nodes in a single database and invoke *batch PIR* [17], [18], [19], [20], which allows a client to retrieve a *set* of entries from the database. This method suffers from the same drawback as the level-wise approach: the client needs to send over the (encrypted) location for each query in the batch, again blowing up communication. Cao et al. [21] propose a custom batch PIR solution for retrieving tree paths, but it only offers a small constant-factor improvement over the straightforward batch PIR approach.

Given the clear downsides of all the above approaches, the goal of this work is to present an efficient solution to the private tree path retrieval problem.

1.1. Our Contributions

We present TreeOnion, a new solution to the private tree path retrieval problem. Our scheme uses a single server. For small entry sizes (e.g., 32 bytes), TreeOnion's communication and computation costs are close to those for retrieving a *single entry* from a database of the same size using a single-server single-query PIR scheme.

TreeOnion can build on many state-of-the-art single-query PIR schemes based on lattice cryptography, such as [15], [16], [22]. In TreeOnion, the nodes in each level of the tree are encoded as a separate database. The client then sends an encrypted query of the underlying single-query scheme that locates *just* the leaf node in the leaf level's database. From this single query for the leaf node, the server can obliviously derive the (encrypted) queries

locating all the nodes on its path. For each level, the server then executes the underlying single-query PIR scheme to obtain a ciphertext encrypting the target node on that level. Next, the server merges the ciphertexts across all levels of the tree (using techniques from the literature [20], [23]) and sends the merged response back to the client. The client recovers all the nodes on the path from the merged response.

Results. We implement TreeOnion using OnionPIRv2 [15] as the underlying single-query PIR scheme. On binary trees of size 256 MiB to 8 GiB with 32-byte nodes, TreeOnion requires 25.5 KiB of online communication, which is exactly the communication cost of OnionPIRv2 for retrieving a single entry from the same database. The server computation cost of TreeOnion is about 10–30% higher than that of an OnionPIRv2 query.

We compare our scheme with the three straightforward baselines (level-wise, batch PIR, and path-per-leaf) and the custom tree path retrieval scheme of Cao et al. [21]. At every tree size we evaluate, TreeOnion always achieves the lowest communication and fastest server computation; for trees of size at least 1 GiB, TreeOnion is at least an order of magnitude better in one of the two metrics.

2. Related Work

Several approaches mentioned in Section 1 require an underlying single-query PIR scheme or a batch PIR scheme. The efficiency of the tree path retrieval scheme naturally depends on the underlying PIR scheme. In this section, we review the PIR literature, again focusing on single-server schemes. We then review the *Merkle tree*, which is our primary application.

Single-query PIR schemes. State-of-the-art single-server PIR schemes [15], [22], [13], [14], [20] utilize somewhat homomorphic encryption (SHE), in particular, lattice-based SHE such as BFV and RGSW (Section 3). At a high level, the server arranges the database as a hypercube, and the client's query encrypts the coordinates of the target entry along every dimension. Using the encrypted coordinates, the server homomorphically selects the entries that match the client's coordinate one dimension at a time, arriving at an encryption of the target entry after the last dimension. Following a paradigm pioneered by SealPIR [18], the client packs the encrypted coordinates into a single ciphertext, and the server *expands* this ciphertext into separate ciphertexts for each coordinate. The expansion step requires key material that the client registers with the server once, creating per-client state that the server must maintain. Among the schemes in this paradigm [18], [15], [22], [13], [14], [20], the state-of-the-art is OnionPIRv2 [15], which we use in the construction of TreeOnion.

Another line of work avoids per-client storage on the server and relies on preprocessing the database to speed up the server's online computation [24], [25], [16]. We believe that the ideas of TreeOnion can be adapted to use these schemes too.

Batch PIR. Batch PIR can be constructed by combining a single-query PIR scheme with *batch codes* or *probabilistic batch codes* (PBC) [18], [17], [26]. This approach requires the server to return at least one ciphertext for each entry in the batch. With lattice ciphertexts, this is costly and wasteful when entries are small. Several batch PIR schemes are custom-designed specifically for small entries [19], [27], [20]. Among these, Respire [20] is the state-of-the-art when the batch size is also small.

Cao et al. [21] optimize batch PIR for the specific setting of retrieving a path of nodes from a binary tree. Instead of using probabilistic batch codes, which replicate the database a constant number of times, they color the nodes of the tree so that every path contains exactly one node of each color, and the client retrieves a path by issuing one PIR query for each color. Their tree coloring does not replicate the database and thus outperforms PBC-based batch PIR by a small constant factor.

Merkle trees and proofs. A Merkle tree [12] is an *authenticated data structure* that allows clients to verify data retrieved from a server against a *digest*. Consider a list of N entries. A Merkle tree of size N is abinary tree consisting of N leaves, where the i-th leaf stores the hash of the i-th entry, and each inner node stores the *hash* of its two children. The root node is the digest.

When the client wants to retrieve the i-th entry, the server sends the entry together with a *Merkle proof*, which consists of the siblings of the nodes on the path from the root to the i-th leaf. Given the proof and the entry, the client recomputes the root hash and accepts the entry only when the result matches the digest. The server cannot make the client accept a wrong entry due to the *collision resistance* of the hash function.

Merkle trees are used in a wide range of applications, such as certificate transparency [28], key transparency systems for end-to-end encryption [29], cryptocurrencies and blockchains [30], [31], and version control systems like Git.

3. Background

3.1. Notation

For an integer x, we write $[x] = \{0, \ldots, x-1\}$. For an array A, we write $A[\colon k]$ to denote its first k entries.

Let α be a bit string of length a. We use α_b, for $b \in [a]$, to denote the b-th least significant bit of α. For $k \le a$, we use $\alpha[\colon k]$ to denote the k most significant bits of α.

We naturally also view α as an integer in $[2^a]$ whose value is $\sum_{b\in[a]} 2^b \alpha_b$. We use 1_α to denote the one-hot encoding of α; 1_α is a bit string of length 2^a that is 1 at position α (i.e., $(1_\alpha)_\alpha = 1$) and 0 at all other positions.

3.2. Private Tree Path Retrieval

A perfect binary tree T with $N = 2^L$ leaves has levels $\ell = 0, \ldots, L$, with the root on level 0 and the leaves on level L. Let

$$T_\ell[0], T_\ell[1], \ldots, T_\ell[2^\ell - 1]$$

denote the nodes at level ℓ. The children of $T_\ell[j]$ are $T_{\ell+1}[2j]$ and $T_{\ell+1}[2j+1]$.

The *path* to $T_L[i]$, i.e., the i-th leaf, is the set of nodes on the path from the root to $T_L[i]$. In particular, it includes the following nodes:

$$\mathsf{Path}(i) = \{T_\ell[i[:\ell]] : \ell \in \{0, 1, \ldots, L\}\}. \quad (1)$$

Observe that the index for the node at level ℓ of $\mathsf{Path}(i)$ is the integer represented by the first ℓ bits of i.

Definition 1 (Private Tree Path Retrieval)**.** The server has a tree T of height L, and the client has a leaf index i. A Private Tree Path Retrieval protocol between the client and the server must satisfy the following:

- ***Correctness:*** For *any* leaf index i, the client outputs $\mathsf{Path}(i)$.
- ***Privacy*** (informal): The server does not learn *any* information about the client's leaf index i.

More formally, privacy can be defined with the following game between the client and the server/adversary:

- The server/adversary selects two leaf indices i_0 and i_1, and sends i_0, i_1 to the client.
- The client samples a bit $b \leftarrow \$\{0, 1\}$ randomly and executes a private tree path retrieval query for i_b with the server.
- The server tries to guess b.

If all computationally bounded adversary guesses b with probability at most $\frac{1}{2}+\epsilon$, where ϵ is negligible in the security parameter, the scheme satisfies privacy.

3.3. Single-query Private Information Retrieval

In this section, we introduce several procedures from a recent single-query PIR scheme OnionPIRv2 [15]. TreeOnion uses these procedures in black boxes. Before that, we first introduce the BFV [32], [33] and RGSW [34] *Somewhat Homomorphic Encryption* (SHE) schemes, which these procedures build on.

BFV. Consider polynomial rings $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^n + 1)$ and $\mathcal{R}_t = \mathbb{Z}_t[X]/(X^n+1)$ for a power of two n, a ciphertext modulus q, and a plaintext modulus $t \ll q$. A plaintext is a polynomial $m \in \mathcal{R}_t$; we write $m[u]$ for the coefficient of X^u in m.

In BFV [32], [33], the secret key s is a polynomial in $\mathcal{R}_q$, and the ciphertext for a plaintext polynomial $m \in \mathcal{R}_t$ is a pair

$$C = (c_0, c_1) = (-as + e + \Delta m,\ a)$$

where $\Delta = \lfloor q/t \rfloor$, $a \in \mathcal{R}_q$ is a uniformly random polynomial, and $e \in \mathcal{R}_q$ is a small *noise*. Let $\mathsf{BFV}(m)$ denote a BFV ciphertext that encrypts message m. The security of BFV rests on the Ring Learning with Errors (RLWE) assumption [35].

BFV supports efficient homomorphic linear operations on ciphertexts, in particular,

- *Addition:* $\mathsf{BFV}(m) + \mathsf{BFV}(m')$ gives $\mathsf{BFV}(m+m')$.
- *Plaintext multiplication:* For any plaintext $y \in \mathcal{R}_t$, $y \cdot \mathsf{BFV}(m)$ gives $\mathsf{BFV}(ym)$.

Homomorphic operations increase noise in the resulting ciphertext. A ciphertext decrypts correctly only if its noise stays small. Thus, somewhat homomorphic encryption schemes like BFV support only a limited number of homomorphic operations on ciphertexts. Higher-level protocols must therefore be carefully designed to keep noise growth within budget.

BFV also supports homomorphic multiplication between two ciphertexts. But state-of-the-art PIR schemes such as OnionPIRv2 intentionally avoid this operation, because it is computationally expensive and leads to high noise growth. Instead, to support ciphertext multiplications, OnionPIRv2 relies on the RGSW encryption scheme, introduced next.

RGSW. RGSW [34] encryption can use the same secret key as BFV encryption. We use $\mathsf{GSW}(\mu)$ to denote a RGSW ciphertext encrypting a bit μ. RGSW gives rise to a special homomorphic ciphertext multiplication operation between an RGSW ciphertext and a BFV ciphertext under the same secret key. The result is a BFV ciphertext that encrypts the product of the underlying messages. This special multiplication operation has low noise growth.

For a bit string β of b bits, we use the following notation to denote a vector of ciphertexts encrypting β bit by bit, i.e.,

$$\mathsf{BFV}[\beta] = [\mathsf{BFV}(\beta_{b-1}), \ldots, \mathsf{BFV}(\beta_0)],$$

$$\mathsf{GSW}[\beta] = [\mathsf{GSW}(\beta_{b-1}), \ldots, \mathsf{GSW}(\beta_0)].$$

QueryPack and Expand. In OnionPIRv2, the client wishes to encrypt the bits of its query index $i \in [N]$ and sends them to the server. However, encrypting each bit in a separate ciphertext would substantially blow up the communication. Instead, the client packs these bits into a single BFV ciphertext and sends it to the server. The server then expands the packed ciphertext to obtain ciphertexts encrypting each individual bit.

Concretely, many state-of-the-art schemes (including OnionPIRv2) view the database as a $2^{\mathsf{f}} \times 2 \times \cdots \times 2$ hypercube, where the *first dimension size* $2^{\mathsf{f}} \leq N$ is an important protocol parameter. The i-th database entry can be located using the bits of index i, split into two sequences α, β as follows

$$i = (\underbrace{\alpha}_{\mathsf{f}\text{ bits}} \mid \underbrace{\beta}_{L-\mathsf{f}\text{ bits}})_2, \quad (2)$$

where α is the first dimension coordinate of entry i and β locates entry i in the other dimensions.

Given the query index i, the client encodes the first-dimension bits α as a one-hot vector 1_α and keeps the bits of β as they are. The client then runs a procedure $\mathsf{QueryPack}(1_\alpha; \beta)$ that packs all these bits into a single BFV ciphertext Q, and sends Q to the server.

The server then runs an Expand procedure that takes the above BFV ciphertext Q and outputs a list of ciphertexts each encrypting a single bit, in particular,

$$\begin{aligned}\mathsf{BFV}[1_\alpha] &= [\mathsf{BFV}((1_\alpha)_{2^\mathsf{f}-1}), \ldots, \mathsf{BFV}((1_\alpha)_0)],\\ \mathsf{GSW}[\beta] &= [\mathsf{GSW}(\beta_{L-\mathsf{f}-1}), \ldots, \mathsf{GSW}(\beta_0)].\end{aligned} \tag{3}$$

Select. The server uses the expanded ciphertexts $\mathsf{BFV}[1_\alpha], \mathsf{GSW}[\beta]$ to select the database entry at index i. Concretely, the database is encoded as plaintext polynomials

$$\mathbf{D} = [D_0, D_1, \ldots, D_{N-1}].$$

The server runs a procedure

$$\mathsf{Select}(\mathbf{D}, \mathsf{BFV}[1_\alpha], \mathsf{GSW}[\beta]),$$

which outputs $\mathsf{BFV}(D_i)$. The server sends $\mathsf{BFV}(D_i)$ back to the client. The client decrypts and obtains D_i.

3.4. Additional SHE Procedures

We also introduce some additional SHE procedures from the batch PIR literature [20], [23] that TreeOnion relies on.

Rotate. The Rotate [20] procedure takes a BFV ciphertext and an encrypted shift amount, and cyclically shifts the coefficients of the underlying plaintext by that amount.

Concretely, consider a BFV ciphertext $\mathsf{BFV}(m)$ where $m \in \mathcal{R}_t$. The shift amount is a value $\gamma \in [n]$ whose binary representation is encrypted as $\mathsf{GSW}[\gamma]$ bit by bit. The $\mathsf{Rotate}(\mathsf{BFV}(m), \mathsf{GSW}[\gamma])$ procedure [20] outputs a BFV ciphertext:

$$\mathsf{BFV}(m \cdot X^{-\gamma})$$

Merge. The Merge [23] procedure takes as input several BFV ciphertexts whose "useful" values sit at coefficients that are multiples of a power of two apart, and packs these useful values into a single BFV ciphertext.

Concretely, consider a parameter ρ that is a power of two, and a list of BFV ciphertexts

$$[\mathsf{BFV}(m_0), \ldots, \mathsf{BFV}(m_{k-1})],$$

encrypting plaintext polynomials $m_0, m_1, \ldots, m_{k-1}$ in $\mathcal{R}_t$; each of these plaintexts contains useful values at coefficients that are multiples of ρ, i.e., $X^0, X^\rho, X^{2\rho}, \ldots$ The $\mathsf{Merge}(\rho; \mathsf{BFV}(m_0), \ldots, \mathsf{BFV}(m_{k-1}))$ procedure [23] outputs a single BFV ciphertext $\mathsf{BFV}(m')$ whose underlying plaintext polynomial m' contains all the useful values in $m_0, m_1, \ldots, m_{k-1}$ in public, distinct coefficients of m'. Note that the merge procedure requires the total number of useful values across all k ciphertexts to be at most n. Fig. 1 shows an example.

In TreeOnion, the server uses the rotate and merge procedures to merge several ciphertexts, each encrypting one node on the path, into a single ciphertext that contains all nodes on the path.

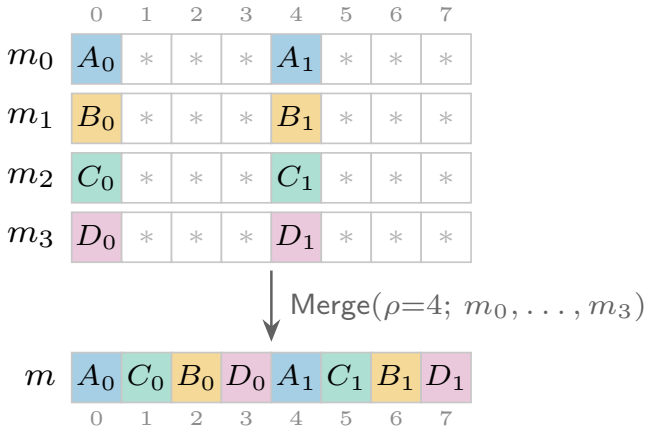

Figure 1. An illustration of the Merge procedure from [23]; in this illustration, the number of input ciphertexts $k = 4$, the polynomial size $n = 8$, and $\rho = 4$.

4. TreeOnion

In this section, we specify the design of TreeOnion in detail. Recall the setting from Section 3.2. A server holds a public database that is organized as a tree with N leaf nodes, where each tree node (leaf or internal) contains a w-bit entry. The client holds a leaf index $i \in [N]$ and wants to retrieve all the nodes on the path from the root to leaf i.

For ease of exposition, we *first* consider the case with a perfect binary tree T of height L with $N = 2^L$ leaves, and each node fits into *one* coefficient of a plaintext polynomial $\mathcal{R}_t$ (i.e., $w < \log t$). Later, we extend the scheme to support larger nodes (Section 4.5) and d-ary trees (Section 4.6).

4.1. Encoding the Binary Tree

Let ρ denote the number of nodes that a plaintext polynomial can hold. For now, $\rho = n$, as each node fits in one coefficient. The server encodes each level of the tree separately. Level ℓ needs

$$R_\ell = \lceil 2^\ell / \rho \rceil$$

plaintext polynomials, denoted $D_\ell[0], \ldots, D_\ell[R_\ell - 1]$.

The server processes the 2^ℓ nodes of the level from left to right and encodes them across the R_ℓ polynomials in round-robin order, filling the lowest-degree coefficients first: the first R_ℓ nodes go to coefficient 0, the next R_ℓ nodes go to coefficient 1, and so on. In other words, the j-th node $T_\ell[j]$ of level ℓ is stored at the $\lfloor j/R_\ell \rfloor$-th coefficient of polynomial $D_\ell[j \bmod R_\ell]$.

The encoding procedure is described in Algorithm 1.

4.2. Client Query

The client holds the leaf index i. Given a parameter $\mathsf{f} \le L - \log \rho$ for the first dimension size (see Section 3.3), the client splits the bits of i into three bit strings γ, α, β:

$$i = (\underbrace{\gamma}_{\log \rho \text{ bits}} | \underbrace{\alpha}_{\mathsf{f} \text{ bits}} | \underbrace{\beta}_{L-\log\rho-\mathsf{f} \text{ bits}})_2 \tag{4}$$

Algorithm 1: Algorithm to encode the binary tree as plaintext polynomials.

Input: Binary tree T of height L, parameter ρ
Output: For every level ℓ, plaintext polynomials $D_\ell[0], \ldots, D_\ell[R_\ell - 1]$

for $\ell = 0, 1, \ldots, L$ **do**
  $R_\ell \leftarrow \lceil 2^\ell / \rho \rceil$
  Initialize empty polynomials $D_\ell[p]$ for all $p < R_\ell$
  for $j = 0, 1, \ldots, 2^\ell - 1$ **do**
    Set slot $\lfloor j / R_\ell \rfloor$ of $D_\ell[j \bmod R_\ell]$ to $T_\ell[j]$
return $\{D_\ell[j] : \ell \in [L+1], j \in [R_\ell]\}$

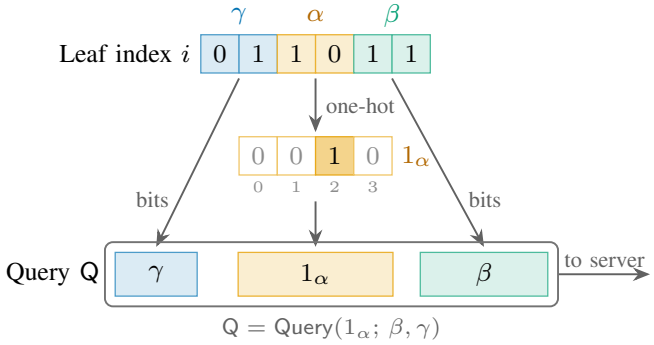

Figure 2. An illustration showing how the client computes the query using the leaf index i; in this illustration, the number of levels L of the tree is 6 and the client's leaf index $i = 27$.

With the encoding in Section 4.1, these three bit strings locate leaf i: α and β concatenated locate the plaintext polynomial holding leaf i, and γ is the coefficient that leaf i occupies within that plaintext polynomial. The client only sends the encrypted bits of i to the server. From those, the server will derive the ciphertexts locating all other nodes on the path (Section 4.3).

The client packs these bits into a single ciphertext $\mathsf{Q} \leftarrow \mathsf{QueryPack}(1_\alpha; \beta, \gamma)$ (Section 3.3), and sends Q as the query to the server. Note again that α is encoded in the one-hot form 1_α. Fig. 2 illustrates the construction.

4.3. Query Expansion

Expanding the client query. The server first expands the client query Q with Expand (Section 3.3). The expansion procedure outputs ciphertexts of the bits of $1_\alpha, \beta, \gamma$. In particular, the server obtains

$$\mathsf{BFV}[1_\alpha], \mathsf{GSW}[\beta], \mathsf{GSW}[\gamma].$$

Obliviously locating the nodes on the path. Using these ciphertexts, the server needs to (obliviously) compute the ciphertexts locating every other node on the path. We call these the *target nodes*.

For each target node, the server needs the ciphertexts that locate: i) the plaintext polynomial containing the node, and ii) the coefficient occupied by the node within that plaintext polynomial. Observe that the location of each target node is simply a prefix of the bits of the leaf node index i. In particular, the target node on level ℓ is located by the prefix $i[:\ell]$ of the leaf index: its first $\min(\ell, \log \rho)$ bits give the coefficient, and the remaining bits give the plaintext polynomial index.

The server computes these encrypted locations by starting at the leaf and moving up level by level to the root. Essentially, the server drops one bit of the encrypted leaf index at a time. As the server moves up the tree, the next bit to be dropped moves from β into α and eventually into γ (Fig. 3). This partitions the levels into three ranges: *deep, middle, and shallow*. If every bit of i were encrypted in its own ciphertext, the server could simply drop one ciphertext at a time. This is the case for γ and β, whose bits are encrypted individually as $\mathsf{GSW}[\gamma]$ and $\mathsf{GSW}[\beta]$. However, α is encoded in one-hot form 1_α and requires a different strategy.

Concretely, for each *deep* level ℓ, with $\ell > \log \rho + \mathsf{f}$, the server simply drops the GSW ciphertexts of the $L - \ell$ least significant bits of β, and obtains the ciphertexts

$$\mathsf{BFV}[1_\alpha], \mathsf{GSW}[\beta[:\ell - \log \rho - \mathsf{f}]]$$

that represent the target node's plaintext index. $\mathsf{GSW}[\gamma]$ still represents the target node's coefficient.

For each *middle* level ℓ, with $\log \rho < \ell \leq \log \rho + \mathsf{f}$, the target node's plaintext index is $\alpha[:k]$, where $k = \ell - \log \rho$. Recall that the server has a ciphertext for each bit of the one-hot encoding of α. The server can efficiently compute the ciphertexts of the one-hot representation of $\alpha[:k]$

$$\mathsf{BFV}[1_{\alpha[:k]}]$$

one level at a time, by leveraging the natural relationship between one-hot encodings $1_{\alpha[:k]}$ and $1_{\alpha[:k+1]}$. In particular, starting from the most significant bit, each bit of $1_{\alpha[:k]}$ is the sum of the next two contiguous bits of $1_{\alpha[:k+1]}$. Thus, the ciphertext of each bit of $1_{\alpha[:k]}$ can be obtained by a simple homomorphic addition of each adjacent pair of ciphertexts of $1_{\alpha[:k+1]}$. The BFV portion of Fig. 3 illustrates this. Like deep levels, the ciphertexts $\mathsf{GSW}[\gamma]$ represent the target node's coefficient.

For each *shallow* level $\ell \leq \log \rho$, the whole level fits in a single plaintext polynomial. So the server only needs to derive the ciphertexts that represent the target node's coefficient within this single plaintext polynomial. For this, the server simply needs to drop the GSW ciphertexts for the $\log \rho - \ell$ least significant bits of γ and uses

$$\mathsf{GSW}[\gamma[:\ell]].$$

Fig. 3 illustrates how the server derives the ciphertexts of every level from those of the leaf level.

4.4. Selecting and Merging the Target Nodes

Selecting the target node from each level. Using the ciphertexts derived above, the server computes, for every level

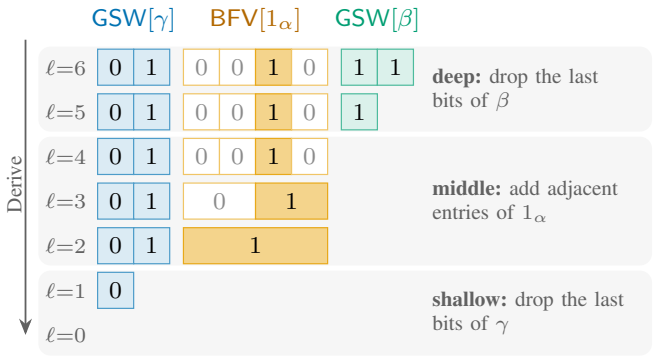

Figure 3. An illustration showing how the server computes the encrypted location of target nodes using the encrypted location of the leaf node; in this illustration, the number of levels L of the tree is 6.

ℓ, a ciphertext C_ℓ that encrypts the plaintext polynomial of level ℓ containing the target node, without learning which polynomial it is.

Let $\mathbf{D}_\ell = (D_\ell[0], D_\ell[1], \ldots, D_\ell[R_\ell - 1])$ denote the plaintext polynomials of level ℓ. For each deep and middle level ℓ, the server computes the ciphertext C_ℓ by using the Select procedure (Section 3.3) on $\mathbf{D}_\ell$ and the ciphertexts of the target node's plaintext index (computed in the previous step). For each shallow level ℓ, there is only one plaintext, so the server computes its ciphertext as the product of the plaintext and an encryption of 1.

Merging ciphertexts across levels. The server now has $L + 1$ ciphertext polynomials, each encrypting a target node. The server can simply return all of them to the client. However, each ciphertext carries only one useful coefficient, and sending back these $L+1$ ciphertexts would be communication-inefficient. Instead, the server can merge these ciphertexts into a single ciphertext before returning it to the client.

This is carried out by the server in two steps. First, the server has ciphertexts that encrypt the coefficients of all target nodes, which are $\mathsf{GSW}[\gamma]$ for deep and middle levels and $\mathsf{GSW}[\gamma[:\ell]]$ for each shallow level ℓ (Section 4.3). The server moves each target node at level ℓ to the 0-th coefficient using Rotate (Section 3.3).

Then, it merges all the rotated ciphertexts into a single ciphertext C using

$$C \leftarrow \mathsf{Merge}(\rho; C_0, \ldots, C_L)$$

(Section 3.4). Merge ensures that each target node is placed in a distinct, public coefficient of the plaintext underlying the output ciphertext C.

The server returns C to the client. The client decrypts and obtains the target nodes from distinct, public coefficients of the plaintext polynomial.

Algorithm 2 details the steps of the client query to the response.

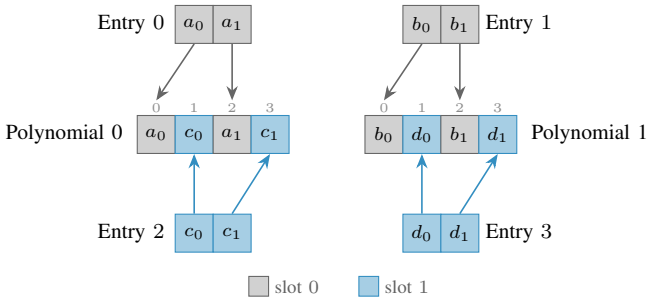

Figure 4. An illustration showing how the server encodes nodes that occupy more than one plaintext polynomial coefficient. The level has four nodes 0, 1, 2, 3, each occupying $g = 2$ coefficients, and is encoded into $R = 2$ polynomials of $n = 4$ coefficients.

4.5. Extending to Multi-coefficient Sized Nodes

We now extend TreeOnion to handle larger nodes that occupy multiple coefficients. For ease of exposition, we first consider the case where the number of coefficients occupied by a node is a power of two, denoted g. This case requires minimum modifications. The server simply encodes each plaintext to contain $\rho = n/g$ nodes instead of $\rho = n$ nodes. In particular, the coefficients of each plaintext polynomial are partitioned into $\rho = n/g$ *slots*, where slot u contains the coefficients

$$u, u + \rho, \ldots, u + (g - 1)\rho$$

(Fig. 4). The server scans every level ℓ, and places node j of level ℓ in slot $\lfloor j/R_\ell \rfloor$ of polynomial $D_\ell[j \bmod R_\ell]$, as described in Algorithm 1.

The procedure to privately retrieve paths is exactly as described in Algorithm 2. The client query and the derivation of each level's query ciphertexts do not depend on the layout within a polynomial, so they are unchanged. The Select step, which selects the plaintext polynomial holding the target node of level ℓ, is also unchanged, since it operates on whole polynomials and is not affected by the internal structure of the plaintext polynomials. Placing each node at coefficients spaced ρ apart allows the rotation and merge procedures to remain unchanged as well. Concretely, if the target node of level ℓ is in slot u, then Rotate multiplies the underlying plaintext by X^{-u}, which moves every coefficient down by u, and places the target node at coefficients that are multiples of ρ. Then, Merge with parameter ρ takes exactly these coefficients from every level and places them in a single response ciphertext (assuming they fit). The client decrypts the response ciphertext, reads the g coefficients of every level and concatenates them to obtain the target node of that level.

Splitting nodes to reduce padding. When the number of coefficients occupied by a target node is not a power of two, we could pad it to the next power of two, but this wastes many coefficients in the polynomials. So instead, we can split every node into P smaller pieces of equal size, such that the number of coefficients needed for each piece is close to a power of two (again denoted g). Each piece

Algorithm 2: Algorithm to retrieve $\mathsf{Path}(i)$.

Client input: Leaf index i, parameters ρ, f
Server input: Plaintext polynomials $\mathbf{D}_\ell$ of every level ℓ (from Algorithm 1), parameters ρ, f
Client output: $\mathsf{Path}(i)$

Client
Split i into $(\gamma \mid \alpha \mid \beta)_2$ by Eq. (4)
$\mathsf{Q} \leftarrow \mathsf{QueryPack}(1_\alpha;\, \beta, \gamma)$
Send Q to the server

Server
$(\mathbf{A},\, \mathbf{B},\, \mathbf{G}) \leftarrow \mathsf{Expand}(\mathsf{Q})$ ▷ $\mathbf{A} = \mathsf{BFV}[1_\alpha]$, $\mathbf{B} = \mathsf{GSW}[\beta]$, $\mathbf{G} = \mathsf{GSW}[\gamma]$
for $\ell = L, L-1, \ldots, \log\rho + \mathsf{f} + 1$ **do** ▷ deep levels
  $C_\ell \leftarrow \mathsf{Select}(\mathbf{D}_\ell,\, \mathbf{A},\, \mathbf{B}[:\, \ell - \log\rho - \mathsf{f}])$
  $C_\ell \leftarrow \mathsf{Rotate}\big(C_\ell,\, \mathbf{G}\big)$
for $\ell = \log\rho + \mathsf{f}, \ldots, \log\rho + 1$ **do** ▷ middle levels
  $C_\ell \leftarrow \mathsf{Select}(\mathbf{D}_\ell,\, \mathbf{A})$ ▷ $\mathbf{A} = \mathsf{BFV}[1_{\alpha[:k]}]$
  $C_\ell \leftarrow \mathsf{Rotate}\big(C_\ell,\, \mathbf{G}\big)$
  $k \leftarrow \ell - \log\rho$
  for $x < 2^{k-1}$ **do** $\mathbf{A}[x] \leftarrow \mathbf{A}[2x] + \mathbf{A}[2x+1]$
  $\mathbf{A} \leftarrow \mathbf{A}[:\, 2^{k-1}]$
for $\ell = \log\rho, \ldots, 0$ **do** ▷ shallow levels
  $C_\ell \leftarrow \mathbf{A}[0] \cdot D_\ell[0]$ ▷ $\mathbf{A}[0]$ encrypts 1
  $C_\ell \leftarrow \mathsf{Rotate}\big(C_\ell,\, \mathbf{G}[:\, \ell]\big)$
$C \leftarrow \mathsf{Merge}(\rho; C_0, \ldots, C_L)$
Send C

Client
Decrypt the response C and output the target node of each level ℓ from the plaintext

is stored and encoded as part of a separate tree database. Since all P databases share the same structure (as the piece sizes are the same), the client still sends a single query; the server runs selection and rotation on each database with the same expanded query ciphertexts and merges *all* $P(L+1)$ resulting ciphertexts into one response. The client reassembles each node from its $P \cdot g$ pieces in the decrypted response.

Target nodes do not fit in one plaintext. When the target nodes do not fit in a single plaintext, the server merges the level ciphertexts into several ciphertexts using multiple Merge operations. The client then obtains the target nodes from these ciphertexts as described before.

4.6. d-ary Trees

TreeOnion can also be naturally extended from a binary tree to a perfect d-ary tree where d is a power of two.

First, during the encoding step, the server places the d^ℓ tree nodes of each level ℓ in R_ℓ polynomials where

$$R_\ell \leftarrow \lceil d^\ell / \rho \rceil,$$

following the same procedure described before.

When the client wants to retrieve a tree path ending at leaf $i \in [d^L]$, it writes i in the binary representation and splits these bits into γ, α, and β, exactly as in Eq. (4). As described for the binary tree case, the client packs these bits into a query ciphertext Q and sends it to the server, and the server expands the query.

Since d is a power of two, the target node on level ℓ has index $i[:\, \ell \log d]$, i.e., moving up one level drops $\log d$ least significant bits of the index, instead of a single bit. Therefore, after expanding Q, the server derives the ciphertexts at each level as in Section 4.3, except that it drops $\log d$ bits per level. The remaining steps are unchanged.

4.7. Correctness, Privacy, and Cost

Theorem 1 (Correctness)**.** *In Algorithm 2, the client with leaf index i outputs* $\mathsf{Path}(i)$.

Proof. The proof composes the guarantees of QueryPack, Expand, Select, Rotate, and Merge.

First, by the encoding of Algorithm 1, the target node of level ℓ lies at the following location:

- on a deep level ℓ, in slot γ of polynomial $(\alpha \,|\, \beta[:\, \ell - \log\rho - f])_2$, where $|$ denotes bit concatenation;
- on a middle level ℓ, in slot γ of polynomial $\alpha[:\, \ell - \log\rho]$;
- on a shallow level ℓ, in slot $\gamma[:\, \ell]$ of the single polynomial in that level.

These are exactly the locations that the server derives, so Select returns an encryption of the polynomial that contains the target node.

Second, Rotate moves the g coefficients of the target node to the coefficients $0, \rho, \ldots, (g-1)\rho$ in each level's ciphertext. Finally, Merge places the coefficients at multiples of ρ of every level at distinct, public coefficients of the response, from which the client obtains every target node. □

Theorem 2 (Privacy)**.** *The scheme in Algorithm 2 satisfies privacy as defined in Definition 1.*

Several procedures in TreeOnion from Section 3.3, inherited from OnionPIRv2, require *key material*, which consists of encryptions of values that depend on the client's secret key s. The client generates this key material once and sends it to the server offline. The server stores it and reuses it for all of the client's later queries. Since the key material encrypts functions of s under s itself, TreeOnion relies on the so-called *circular security* assumption of RLWE, just like many prior RLWE-based PIR schemes [13], [15], [20], [23].

Apart from the key material, the server in TreeOnion sees only the query Q, a BFV ciphertext. Under the RLWE assumption with circular security, Q is indistinguishable from any other RLWE ciphertext, independently of i.

Communication. The client query packs the 2^{f} entries of 1_α, and the $L - \mathsf{f}$ bits of β and γ using QueryPack. This takes just one BFV ciphertext for all practical tree sizes.

The response is about as large as the path itself, $(L+1)\,w$ bits, packed into ciphertexts that each hold n coefficients' worth of data. For small w (e.g., 32-byte), the response is a single ciphertext.

Computation. The client encrypts one query and decrypts one response. The server's computation during selection is linear in the size of the tree. All other steps, such as expanding the query, rotating and merging the responses, depend only on the height of the tree, and are thus not the bottleneck for the server's computation.

5. Applications to Merkle Trees and Authenticated PIR

In this section, we discuss how TreeOnion can be used to privately retrieve a Merkle proof and to enable *authenticated PIR*.

Merkle proofs. Recall Merkle trees from Section 2. Observe that a Merkle proof is not itself a path: it consists of the siblings of the nodes on a path rather than of the nodes themselves. Following Cao et al. [21], the Merkle tree can be preprocessed so that every node is swapped with its sibling, so the Merkle proof of the i-th data entry is exactly the path of nodes from the root to leaf i in the preprocessed Merkle tree.

In some applications, the client already holds the data entry and only needs the Merkle proof. For example, in Certificate Transparency, a browser holds the certificate and wants to verify its inclusion in the log. In this case, TreeOnion solves the problem as described.

If the client also needs to retrieve the data entry, the server should place the data entries in the same order as the leaves of the preprocessed Merkle tree in a separate database. Since this database has the same layout as the leaf level of the Merkle tree, the server can reuse the leaf-level query, and the client sends no additional query. The server selects entry i from this database using the ciphertexts for the leaf level of the Merkle tree, so the client sends no additional query. If the data entry is small, it can be merged into the TreeOnion response. If the entries are larger than the hashes, the server splits them across multiple databases so that each database has the same layout.

Authenticated PIR. In traditional PIR, the server is assumed to be semi-honest, and privacy is the only concern. If the server is malicious instead, *integrity* also becomes a concern. This motivates *Authenticated PIR* [36], which adds an *integrity* requirement: given a public trusted digest of the database, the client either outputs the correct entry or rejects the retrieved entry if the server has tampered with the database.

TreeOnion gives a simple solution to authenticated PIR. Like in all authenticated PIR schemes, the client is assumed to have the Merkle root digest of the database. To retrieve entry i, the client retrieves the entry together with its Merkle proof, i.e., the path to leaf i in the preprocessed Merkle tree. The client then recomputes the Merkle root using the retrieved nodes, and accepts only if it matches the digest. By the collision resistance of the hash function, the server cannot make the client accept a wrong entry.

This scheme, however, does not protect against *selective failure* attacks [36]. In such an attack, a malicious server tampers with parts of the Merkle tree, so that verification fails for some query indices and succeeds for others. If the server can then observe whether the client accepted or rejected the retrieved Merkle proof, for example whether the client continues to use the service, it learns that one bit of information about the client's query.

We remark that, in certain applications, it may be reasonable to assume that the server cannot observe whether the client has accepted or rejected. For example, the client may use the service only once even when it accepts, or the client may intentionally issue dummy queries to hide its rejection. In these scenarios, the simple authenticated PIR based on TreeOnion is much more efficient, at almost no additional cost over PIR without integrity. In contrast, single-server authenticated PIR schemes that also protect against selective failure attacks [36], [37], [38], [39], [40] are a lot more expensive. Some of these schemes require client storage that is a constant fraction of the database (e.g., over 1 GiB for an 8 GiB database [38]), some incur large server computation and communication [36], [37], [40], and some require the client and the server to periodically rerun a costly per-client offline phase [39].

6. Evaluation

We implement TreeOnion in C++ on top of the OnionPIRv2 codebase, and implement the rotation and merging operations from Respire and SPIRIT from scratch. The implementation is available at https://github.com/zcy05331/TreeOnion.

Our implementation of TreeOnion uses OnionPIRv2's RLWE parameters: ring degree $n = 2048$, ciphertext modulus $q \approx 2^{58}$, and plaintext modulus $t \approx 2^{12}$. The lattice estimator [41] puts this configuration at 116 bits of security.

6.1. Experimental Setup

Baselines. We compare TreeOnion with the following baselines. Several of them likewise require an underlying single-query PIR scheme. For a fair comparison, we use OnionPIRv2 [15] as the underlying single-query PIR scheme for all such schemes.

- *Level-wise*: The server stores every level of the tree as a separate database. The client retrieves the target node of each level using the single-query PIR scheme to that level's database.
- *Path-per-leaf*: The server stores each path as one entry of a single database. The client retrieves a path using a single-query PIR scheme.
- *Batch PIR*: The server stores all tree nodes in a single database, and the client retrieves all the nodes on the path with a batch PIR. One generic way to construct batch PIR is to combine a probabilistic batch code (PBC) [18] with a single-query PIR scheme.
- *Respire* [20]: a state-of-the-art custom batch PIR scheme for small entry sizes and small batches.
- *Cao et al.* [21]: This scheme optimizes batch PIR specifically for tree-path queries.

Execution environment. We run all our experiments with a single thread on Intel Xeon Platinum 8358 processors at 2.60 GHz with 256 GiB of RAM.

Database size. We run TreeOnion and all the baselines on trees with 2^{22} to 2^{27} leaves of synthetic 32-byte nodes, which correspond to 256 MiB to 8 GiB of data in the tree.

Metrics. We primarily compare the following metrics of the various schemes:

- *Throughput*, which is the tree size divided by the mean server computation time,
- *Upload*, which is the number of bits sent from the client to the server during a query, and
- *Download*, which is the number of bits sent from the server back to the client during a query.

We also report the size of the key material that the server stores for each client, server preprocessing time, and client computation time.

TABLE 1. COMMUNICATION AND THROUGHPUT OF PRIVATE TREE PATH RETRIEVAL SCHEMES FOR A TREE WITH 2^{24} LEAVES, WITH EACH NODE OF SIZE 32 BYTES.

	Online communication (KiB)			Server throughput
Scheme	Upload	Download	Total	(MB/s)
Level-wise	262	202	464	894
Path-per-leaf	14.5	11.0	25.5	137
Batch PIR (PBC)	538	407	945	240
Respire	121	8.0	129	37
Cao et al.	349	264	613	573
TreeOnion	**14.5**	**11.0**	**25.5**	**1223**

6.2. Evaluation Results

In Table 1, we compare TreeOnion with all baselines at 1 GiB. Fig. 5 shows the throughput and online communication of every scheme across tree sizes. At every size, TreeOnion has the highest throughput, and requires the least communication among all schemes.

Level-wise. The level-wise approach requires 16× to 21× more online communication than TreeOnion, since every level needs its own query and response. On small trees, the server time of the level-wise approach is about 1.4× that of TreeOnion because it performs query expansion at every level, whereas TreeOnion expands a single query. On larger trees, the query expansion cost becomes insignificant, and the server time of the level-wise approach is comparable to that of TreeOnion.

Path-per-leaf. The path-per-leaf approach has the same online communication as TreeOnion, 25.5 KiB, since it also uses one query and one response ciphertext. However, every node is stored in all the paths that contain it, so its database holds $L/2$ times as many nodes as the tree. As a result, its server computation time is 5.8× to 15.7× worse than TreeOnion from 256 MiB to 2 GiB; for even larger trees, its inflated database does not fit in the memory of our test platform.

Batch PIR. PBC-based batch PIR is 3.9× to 5.6× slower than TreeOnion and requires 34× to 42× more online communication. The computation blowup is due to PBC replicating database entries, and the communication blowup comes from the server returning a BFV ciphertext for each node on the path.

Respire requires 3.3× to 8.5× more online communication than TreeOnion and is 22× to 49× slower than TreeOnion from 256 MiB to 4 GiB; its preprocessed database does not fit in memory at 8 GiB. Respire compresses the queries and responses of independent target entries in the batch, and thereby achieves better communication than PBC. But it does not exploit the fact that the nodes to be retrieved form a path, leading to communication and computation blowups.

Prior art in custom tree path retrieval. Cao et al. requires 22× to 27× more online communication than TreeOnion and is 1.4× to 2.5× slower than TreeOnion. It achieves a

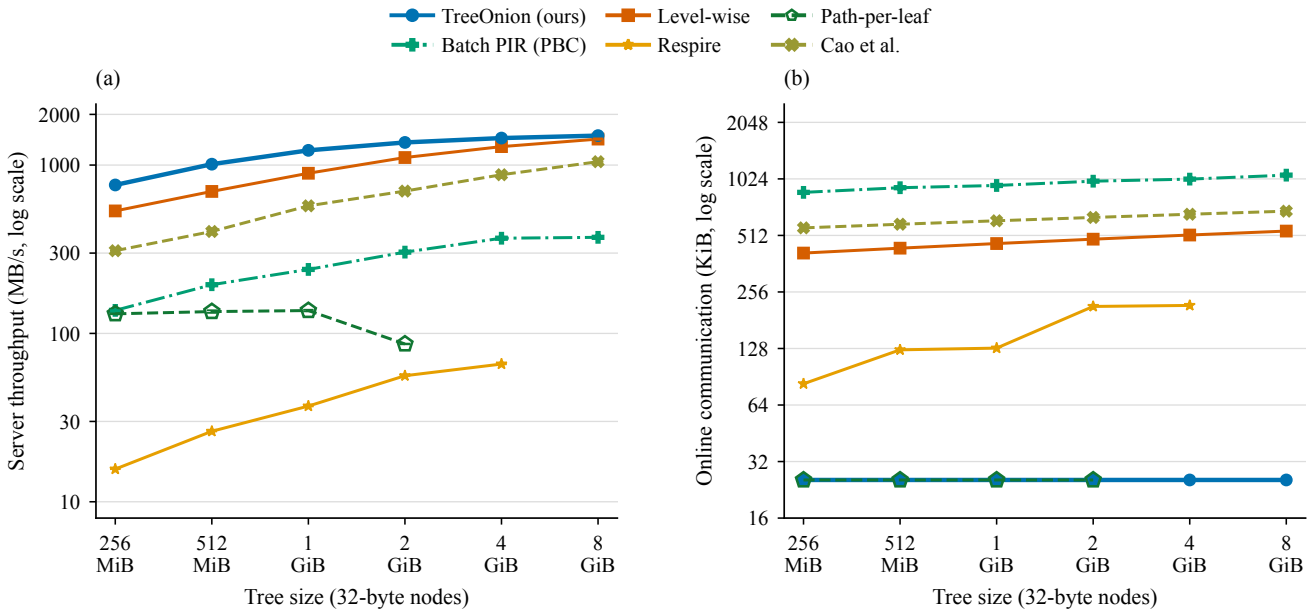

Figure 5. Server computation throughput and online communication across tree sizes.

small constant-factor improvement over PBC as it does not replicate the database, but still requires costly communication as it requires one query ciphertext per target node.

Other metrics. The online client computation for all schemes is less than 1 ms, and in particular, 0.3 ms for TreeOnion.

In all schemes, the client sends key material to the server during an offline phase, and the server stores it. The key material size for each client is about 1.5 MiB for all schemes built on OnionPIRv2. For Respire, it is about 4.6 MiB.

In all schemes, the server preprocesses the plaintext polynomials using Number Theoretic Transform (NTT). The preprocessing cost mainly depends on the amount of database replication of each scheme. Therefore, TreeOnion, level-wise, and Cao et al. have lower preprocessing costs than the other three schemes that replicate the database. Concretely, at 1 GiB, preprocessing takes about 13s for TreeOnion, compared to about 10s for level-wise and Cao et al., 31s for PBC-based batch PIR, 120s for path-per-leaf, and 189s for Respire.

6.3. Ablation

To understand where the savings of TreeOnion come from, Fig. 6 starts with the level-wise approach and adds our techniques one at a time. At the 1 GiB tree, the level-wise approach requires 464 KiB of communication, roughly evenly split between upload and download.

Shared query. Shared query refers to the single query the client sends for the leaf level, which the server expands and processes to locate the other nodes on the path, as in Algorithm 2. The server computes the response for each level,

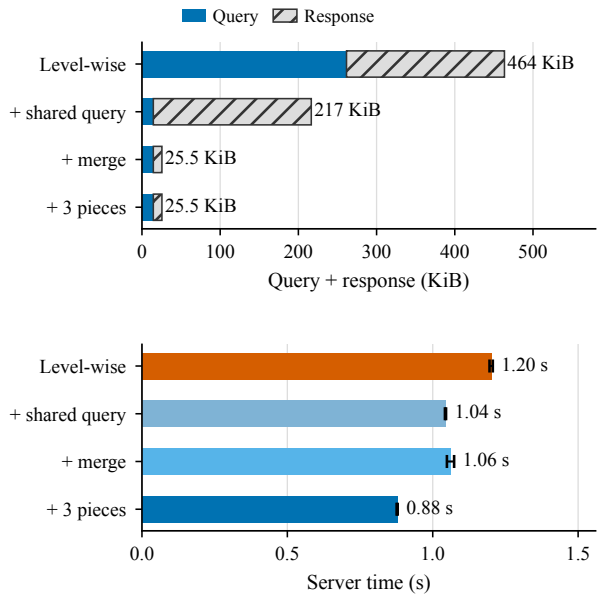

Figure 6. Ablation of TreeOnion when the tree size is 1 GiB and the entry size is 32 bytes.

and sends back each level's response directly without further processing. This significantly reduces the client upload.

Merging responses. Merge refers to the ciphertext merging across levels, so that a single ciphertext is sent back to the client. This significantly reduces the download.

Splitting nodes into pieces. With 32-byte nodes and our

BFV parameters, each node requires 22 coefficients to store. By splitting every node into $P = 3$ pieces of $g = 8$ coefficients, each node now occupies 24 coefficients instead of being padded to 32. This reduces the number of plaintext polynomials needed to encode the tree, and hence reduces the server computation time.

7. Conclusion

We presented TreeOnion, a single-server scheme that allows clients to privately retrieve a path of nodes from a public binary tree. TreeOnion exploits the fact that the location of every node on a path is a prefix of the leaf index. The client sends a single query that locates only the leaf; from there, the server derives the query for target nodes at other levels. The server then merges the selected nodes from all levels into a single ciphertext as the response. As a result, retrieving a whole path with TreeOnion costs about as much as retrieving a single entry.

Acknowledgments. This work is funded in part by the National Science Foundation Award #2246386. ChatGPT Codex and Claude Code were used to assist with the implementation and writing.

References

[1] "Combining machine learning and homomorphic encryption in the Apple ecosystem," https://machinelearning.apple.com/research/homomorphic-encryption.

[2] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg, "PIR-Tor: Scalable anonymous communication using private information retrieval," in *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[3] A. H. Kwon, D. Lazar, S. Devadas, and B. Ford, "Riffle: An efficient communication system with strong anonymity," *Proceedings on Privacy Enhancing Technologies*, vol. 2016, no. 2, pp. 115–134, 2016.

[4] S. Angel and S. Setty, "Unobservable communication over fully untrusted infrastructure," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 551–569.

[5] A. Juels, "Targeted advertising... and privacy too," in *Cryptographers' Track at the RSA Conference*. Springer, 2001, pp. 408–424.

[6] M. Backes, A. Kate, M. Maffei, and K. Pecina, "ObliviAd: Provably secure and practical online behavioral advertising," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 257–271.

[7] M. Green, W. Ladd, and I. Miers, "A protocol for privately reporting ad impressions at scale," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, 2016, pp. 1591–1601.

[8] S. Servan-Schreiber, K. Hogan, and S. Devadas, "AdVeil: A private targeted advertising ecosystem," *Cryptology ePrint Archive*, 2021.

[9] K. Thomas, J. Pullman, K. Yeo, A. Raghunathan, P. G. Kelley, L. Invernizzi, B. Benko, T. Pietraszek, S. Patel, D. Boneh, and E. Bursztein, "Protecting accounts from credential stuffing with password breach alerting," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 1556–1571.

[10] D. Kogan and H. Corrigan-Gibbs, "Private blocklist lookups with Checklist," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 875–892.

[11] P. S. Arunachalaramanan, A. Appan, D. Heath, and L. Ren, "Efficient private range queries on public data," 2027.

[12] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology – CRYPTO '87*, ser. Lecture Notes in Computer Science, vol. 293. Springer, 1988, pp. 369–378.

[13] M. H. Mughees, H. Chen, and L. Ren, "OnionPIR: Response efficient single-server PIR," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2021, pp. 2292–2306.

[14] S. J. Menon and D. J. Wu, "Spiral: Fast, high-rate single-server PIR via FHE composition," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2022, pp. 930–947.

[15] Y. Chen and L. Ren, "OnionPIRv2: Efficient single-server PIR," Cryptology ePrint Archive, Paper 2025/1142, 2025.

[16] R. A. Mahdavi, S. Patel, J. Y. Seo, and K. Yeo, "InsPIRe: Communication-efficient PIR with server-side preprocessing," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2026, pp. 1463–1482.

[17] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai, "Batch codes and their applications," in *36th Annual ACM Symposium on Theory of Computing (STOC)*. ACM, 2004, pp. 262–271.

[18] S. Angel, H. Chen, K. Laine, and S. T. V. Setty, "PIR with compressed queries and amortized query processing," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2018, pp. 962–979.

[19] M. H. Mughees and L. Ren, "Vectorized batch private information retrieval," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2023, pp. 437–452.

[20] A. Burton, S. J. Menon, and D. J. Wu, "Respire: High-rate PIR for databases with small records," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2024, pp. 1463–1477.

[21] Q. Cao, S. H. Dau, R. Gagiano, D. Huynh, X. Yi, P. L. Le, Q.-H. Luu, E. Viterbo, Y.-C. Huang, J. Zhu, M. M. Jalalzai, and C. Feng, "TreePIR: Efficient private retrieval of Merkle proofs via tree colorings with fast indexing and zero storage overhead," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2025, pp. 4476–4494.

[22] M. Luo, F.-H. Liu, and H. Wang, "Faster FHE-based single-server private information retrieval," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2024, pp. 1405–1419.

[23] Z. Zhang, R. Mao, Z. Zhao, H. Pan, Y. Fu, Y. Liu, D. Li, Y. Chen, B. Zhang, C. Ren, J. Dong, Z. Guan, and S. Bian, "SPIRIT: Batch hintless single-server PIR via stateful ciphertext conversion," in *35th USENIX Security Symposium (USENIX Security)*. USENIX Association, 2026, pp. 4523–4542.

[24] B. Li, D. Micciancio, M. Raykova, and M. Schultz-Wu, "Hintless single-server private information retrieval," in *Advances in Cryptology – CRYPTO 2024*, ser. Lecture Notes in Computer Science, vol. 14928. Springer, 2024, pp. 183–217.

[25] S. J. Menon and D. J. Wu, "YPIR: High-throughput single-server PIR with silent preprocessing," in *33rd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2024, pp. 5985–6002.

[26] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[27] J. Liu, J. Li, D. Wu, and K. Ren, "PIRANA: Faster multi-query PIR via constant-weight codes," in *IEEE Symposium on Security and Privacy (S&P)*. IEEE, 2024, pp. 4315–4330.

[28] B. Laurie, E. Messeri, and R. Stradling, "Certificate Transparency version 2.0," RFC Editor, RFC 9162, Dec. 2021.

[29] M. S. Melara, A. Blankstein, J. Bonneau, E. W. Felten, and M. J. Freedman, "CONIKS: Bringing key transparency to end users," in *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, Aug. 2015, pp. 383–398. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/melara

[30] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," White paper, 2008. [Online]. Available: https://bitcoin.org/bitcoin.pdf

[31] V. Buterin, "The Ethereum Whitepaper," White paper, 2013. [Online]. Available: https://ethereum.org/en/whitepaper/

[32] J. Fan and F. Vercauteren, "Somewhat practical fully homomorphic encryption," Cryptology ePrint Archive, Paper 2012/144, 2012.

[33] Z. Brakerski, "Fully homomorphic encryption without modulus switching from classical GapSVP," in *Advances in Cryptology – CRYPTO 2012*, ser. Lecture Notes in Computer Science, vol. 7417. Springer, 2012, pp. 868–886.

[34] C. Gentry, A. Sahai, and B. Waters, "Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based," in *Advances in Cryptology – CRYPTO 2013*, ser. Lecture Notes in Computer Science, vol. 8042. Springer, 2013, pp. 75–92.

[35] V. Lyubashevsky, C. Peikert, and O. Regev, "On ideal lattices and learning with errors over rings," in *Advances in Cryptology – EUROCRYPT 2010*, ser. Lecture Notes in Computer Science, vol. 6110. Springer, 2010, pp. 1–23.

[36] S. Colombo, K. Nikitin, H. Corrigan-Gibbs, D. J. Wu, and B. Ford, "Authenticated private information retrieval," in *32nd USENIX Security Symposium (USENIX Security)*. USENIX Association, 2023, pp. 3835–3851.

[37] M. Dietz and S. Tessaro, "Fully malicious authenticated PIR," in *Advances in Cryptology – CRYPTO 2024*, ser. Lecture Notes in Computer Science, vol. 14928. Springer, 2024, pp. 113–147.

[38] L. de Castro and K. Lee, "VeriSimplePIR: Verifiability in SimplePIR at no online cost for honest servers," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[39] P. S. Arunachalaramanan and L. Ren, "Single-server stateful PIR with verifiability and balanced efficiency," in *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2026, pp. 1385–1402.

[40] M. Rathee, K. Lee, and R. A. Popa, "Verifiable PIR with small client storage," in *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2026, pp. 1423–1442.

[41] M. R. Albrecht, R. Player, and S. Scott, "On the concrete hardness of learning with errors," *Journal of Mathematical Cryptology*, vol. 9, no. 3, pp. 169–203, 2015.