ZipPIR: High-throughput Single-server PIR without Client-side Storage

Rasoul Akhavan Mahdavi Abdulrahman Diaa Florian Kerschbaum
University of Waterloo
{rasoul.akhavan.mahdavi, abdulrahman.diaa, florian.kerschbaum}@uwaterloo.ca

Abstract

Private Information Retrieval (PIR) allows a client to privately access a database without revealing which element is accessed. Initial PIR protocols based on Ring Learning with Errors (RLWE) demonstrated the practicality of PIR, but achieve limited throughput. Alternatively, high-throughput protocols leverage an offline phase that requires substantial client-side storage (e.g., hints in SimplePIR) or involve prohibitive communication costs during the offline phase (e.g., Piano). These limitations conflict with the practical constraints of resource-limited clients and are further exacerbated by dynamic databases, where updates necessitate costly regeneration and retransmission of hints.

To address these challenges, we propose ZipPIR, a high-throughput PIR protocol that compresses LWE ciphertexts into significantly smaller Paillier ciphertexts. ZipPIR leverages the offline phase to obtain this size reduction without incurring the associated computational cost in the online phase. Moreover, under computational assumptions, ZipPIR features an almost silent offline phase, requiring no communication beyond an initial public key, enabling the server to independently generate and update hints during idle times without client interaction. ZipPIR achieves over 2 GB/s of throughput — comparable to state-of-the-art protocols such as SimplePIR — without the need for a large client-stored hint. For PIR over a 1 GB database, ZipPIR has up to 10x higher throughput than existing protocols with no client-side storage, while requiring less than 200 KB of server-side storage per client, significantly enhancing scalability for practical deployments. While prior PIR protocols using Paillier are very inefficient, ZipPIR is the first PIR protocol using Paillier that achieves throughput that is competitive with state-of-the-art PIR protocols.

1 Introduction

Private Information Retrieval (PIR) enables a client to privately retrieve an element from a database without revealing to the server which element was accessed. PIR is an essential functionality for privacy-enhancing applications such as compromised credential checks [48], private contact discovery [64], metadata-private applications [6], private blocklist lookups [46], private STC auditing [44], private messaging [14], and private distributed file systems [53].

Table 1: Performance of PIR over a 1 GB database for different protocols. ∗ Results for a 1 GB database are extrapolated from the paper. † The only client-side storage is a long-term secret key.

Property	PIR-with-keys† [12, 13, 54]	Stateful PIR* [1]	LWE-based PIR (SimplePIR [34, 44])	Sublinear PIR (Piano [66])	**ZipPIR†**
Client-side Storage	-	1 MB	128 MB	66 MB	-
Offline Communication	600 KB - 13 MB	1 GB	128 MB	1 GB	400 B
Throughput	< 1 GB/s	< 500 MB/s	10 GB/s	126 GB/s	3 GB/s

Existing PIR protocols operate in the multi-server or single-server model. Multi-server PIR, while being an efficient solution, relies on a strong assumption that the servers hold identical copies of the database and do not collude. In contrast, single-server PIR [30], where a single server holds the database, is compelling for the applications mentioned above since ensuring multiple servers do not collude can be challenging in practice. Moreover, maintaining a consistent view of a dynamic database among servers requires additional coordination. Hence, in this paper, we focus our attention on single-server PIR schemes.

Clients in the applications mentioned above typically have limited storage and computational resources, e.g., browsers and smartphones. Hence, we can classify existing protocols based on the burden they place on the client.

PIR using client-specific keys. This approach to PIR originated from XPIR [4] and was followed by recent works [6, 12, 13, 54, 56]. While these protocols require no client-side storage, they still suffer from low throughput, up to hundreds of megabytes per second [54]. The primary reason for this low throughput is that these protocols require a large number of public-key operations, usually linear in the database size.

PIR with client storage. Private stateful information retrieval [59] reduces server computation but requires clients to

maintain a large, database-dependent private state, which is difficult for a resource-constrained client. Additionally, the hint must be updated when the database is updated. More recent high-throughput protocols based on Learning With Errors (LWE), such as SimplePIR [44], FrodoPIR [34], and DoublePIR [44], achieve high throughput that approaches memory bandwidth, requiring only simple matrix-vector multiplication in the online phase. However, they require clients to store large hints (128 MB for SimplePIR) that must be updated whenever the database changes, making them impractical for real-world scenarios. Another paradigm takes throughput optimization to the extreme by proposing protocols with sublinear online time [31, 41, 63, 66]. Such protocols perform offline preprocessing such that the online work is only sublinear in the database size, which is a substantial improvement for very large databases. However, these protocols impose impractical client-side requirements. For instance, Piano [66] requires streaming the entire database to the client during preprocessing, while other approaches [31, 41, 63] demand substantial client-side storage and computation, contradicting the resource constraints of limited client devices.

Table 1 summarizes the limitations of existing work. Given these limitations in existing approaches, we pose the following question:

Can we construct a high-throughput PIR protocol without placing storage burden on the client?

In this paper, we answer this question affirmatively by introducing ZipPIR, a new PIR protocol that achieves both high throughput and practical client-side requirements. Compared to existing PIR protocols that fulfill the mentioned requirements such as HintlessPIR and YPIR, we show that ZipPIR is up to 2.2x faster. Moreover, ZipPIR is the first PIR protocol using Paillier that achieves throughput that is competitive with state-of-the-art PIR protocols.

1.1 Technical Overview

At the core of our approach is a novel technique for compressing homomorphic ciphertexts based on LWE and Ring-LWE. While our approach is independently applicable to protocols using homomorphic encryption, we apply it to address the challenges of PIR.

Compressing LWE Ciphertexts. (R)LWE ciphertexts are commonly sent over the network in modern single-server PIR protocols [13, 44, 54, 55] and other applications that use homomorphic encryption [7, 25, 26, 52]. However, they suffer from high ciphertext expansion factors, which is the ratio of ciphertext size to its corresponding plaintext size. This expansion factor directly impacts communication costs in client-server architectures such as PIR [50, 62], through both query uploads (client-generated ciphertexts) and response downloads (ciphertexts processed by the server). While there are effective methods to reduce upload costs [10, 12, 23, 29, 36, 37], the same techniques can not be applied to reduce download costs. Previous efforts to address ciphertext size include switching to smaller parameters in LWE-based schemes [57], performing modulus switching in RLWE-based schemes [2, 27], and LWE-to-RLWE packing [24]. However, an LWE ciphertext size remains proportional to $O(n \log q)$, where n and q represent the ciphertext dimension and modulus, respectively.

Our proposed compression technique reduces the size of and (R)LWE ciphertext by exploiting the linear step in the decryption of (R)LWE ciphertexts, which reduces a ciphertext along its dimension, n. Since this step is linear, it can be performed homomorphically on the server side using an additive homomorphic encryption scheme (e.g., Paillier [58]), allowing the server to return a single additive ciphertext to the client. This compression technique achieves size reduction of approximately 89% (from 6.8KB to 768 B) for a single LWE ciphertext and up to 99% for a batch of LWE ciphertexts. The only additional requirement is a small, reusable compression key — the encrypted (R)LWE secret under the additive scheme.

Constructing ZipPIR. Utilizing this compression technique, ZipPIR addresses the main bottleneck of the high-throughput approach of SimplePIR [44]: the large client-side hint. However, a direct application of compression would result in prohibitive computational costs due to expensive Paillier operations. PIR using additive schemes such as Paillier have been known to be very slow due to these expensive operations [1].

ZipPIR employs an offline/online paradigm to push all expensive Paillier operations to the offline phase that can occur during the server's idle time. We use an alternative representation for Paillier ciphertexts which enables an offline/online split, whilst maintaining security. Importantly, under computational assumptions, the client does not need to communicate with the server during the offline phase, except for a one-time Paillier public key and cryptographic seed. This leaves the online phase with only two matrix-vector multiplications, which are sufficient to achieve throughput comparable to SimplePIR. Since the second matrix multiplication is over a large modulus, we represent the operands in RNS form to enable the use of efficient machine-level operations. Importantly, upon any database updates, the server can redo the precomputation without any communication with the client. While this requires the server to store a per-client state, the nature of this state in ZipPIR differs fundamentally from related work and is novel in the literature. Other protocols store multiple megabytes of static, per-client information that does not require updates over time. In contrast, ZipPIR maintains a much smaller per-client state—only a few hundred kilobytes—which is refreshed per query, though this update can also be performed silently, i.e., without interaction with the client, and during the server's idle time. While using an offline phase is very common for PIR protocols using lattice-based encryption schemes, our work is the first to propose a

practical offline/online protocol using Paillier.

Summary of Results. Our results demonstrate that ZipPIR achieves a throughput of 3 GB/s for a 1GB database and 3.5 GB/s for a 2GB database, while maintaining only a 120 and 170 KB per-client per-query state, respectively. ZipPIR's throughput is up to 10x higher than that of protocols using client-specific keys. Aside from competitive runtime, our work is the first to demonstrate that PIR using additive schemes such as Paillier can be efficient and competitive with pure lattice-based schemes.

2 Background

In this section and throughout the paper, we index the i^{th} element of the vector $\mathbf{a}$ as $\mathbf{a}[i]$. We also define $[n] = \{0, 1, \cdots, n-1\}$ and $\lfloor \cdot \rceil$ denotes rounding to the nearest integer. $x \leftarrow D$ denotes the variable x sampled from a distribution D and $x \leftarrow\!\!\$\ S$ denotes sampling x uniformly from a set S.

2.1 Homomorphic Encryption

Homomorphic Encryption (HE) is a form of public-key cryptography which permits computation on messages while in encrypted form, without the need to access the secret key. Similar to other public-key cryptosystems, homomorphic ciphertexts are larger than the underlying plaintext. The ratio between the ciphertext and plaintext is denoted as the *expansion factor*.

2.1.1 LWE & RLWE ciphertexts

For this work, we describe a simple version of an encryption system based on the Learning With Errors (LWE) [62] assumption which we will denote by $\mathcal{E}_{\text{LWE}}$. The most prominent encryption schemes that have ciphertexts of this format are Regev [62], FHEW [38], and TFHE(CGGI) [28].

$\mathcal{E}_{\text{LWE}}$ uses the following parameters: dimension n, ciphertext modulus q, plaintext modulus p, noise scale $\Delta = \lfloor q/p \rceil$, a discrete error distribution over $\mathbb{Z}_q$ called χ. We sample the secret key, $\mathtt{sk}$, from $\mathbb{Z}_q^n$. The encryption and decryption procedure for $\mathcal{E}_{LWE}$ is shown in Algorithm 1.

Algorithm 1 Encryption and Decryption of $\mathcal{E}_{LWE}$

procedure LWEENCRYPT($\mathtt{sk}, \mu$)
  Sample $\mathbf{a} \xleftarrow{\$} \mathbb{Z}_q^n$ and $e \leftarrow \chi$
  $b = \sum_{i\in[n]} \mathbf{a}[i] \cdot \mathtt{sk}[i] + \Delta \cdot \mu + e \mod q$
  return $\mathsf{ct} = (\mathbf{a}, b)$

procedure LWEDECRYPT($\mathtt{sk}, \mathsf{ct} = (\mathbf{a}, b)$)
  $\mu^* = \left(b - \sum_{i\in[n]} \mathbf{a}[i] \cdot \mathtt{sk}[i]\right) \mod q$
  $\mu' = \lfloor \mu^*/\Delta \rceil$
  return μ'

Fresh ciphertexts can be compressed to reduce network costs. Since $\mathbf{a}$ is sampled at random, we can send the seed used to generate $\mathbf{a}$ instead of the vector itself. Concretely, instead of sending $c = (\mathbf{a}, b)$, the client can produce $\bar{c} = (\theta, b)$ where $\theta \leftarrow \{0,1\}^\lambda$ is the seed of a cryptographically secure PRG used to generate $\mathbf{a}$, i.e., $\mathbf{a} \leftarrow \mathtt{PRG}(\theta)$. With this technique, fresh ciphertexts are only $\lambda + \log_2 q$ bits instead of $n \log_2 q$.

Similar to LWE, we can also construct an encryption scheme based on the Ring Learning with Errors (RLWE) [50] assumption. Cryptosystems such as BGV [20], BFV [18, 40], and CKKS [27] have ciphertexts of a similar format. More details regarding RLWE cryptosystems are provided in Appendix B.

2.1.2 Paillier Cryptosystem

The Paillier cryptosystem [58] is an additive cryptosystem based on the computational intractability of the quadratic residuosity problem. The plaintext space of Paillier is $\mathbb{Z}_m$ for $m = pq$, for two secret primes p and q and a Paillier ciphertext is an element $c \in \mathbb{Z}_{m^2}$. The cryptosystem supports homomorphic additions between two ciphertexts, and plaintext multiplications. Throughout this paper, we denote Paillier addition and plaintext multiplication as $\oplus$ and $\otimes$, respectively. We also define $\text{PAILLIERMATMUL}_m(C, P)$ which denotes a matrix multiplication between a matrix of ciphertexts $X \in \mathbb{Z}_{m^2}^{a\times b}$ and matrix of plaintext values $P \in \mathbb{Z}_m^{b\times c}$.

2.2 Private Information Retrieval

Private Information Retrieval (PIR) is a protocol where a client retrieves an element from a database such that the query is not revealed. A specific variant of PIR dubbed *Offline/Online PIR* aims to push the costly operations to the offline phase, such that the online phase is very efficient. We rely on the definition proposed by Corrigan-Gibbs and Kogan [31], where an offline/online PIR scheme is a tuple $\Pi = (\text{SETUP}, \text{HINT}, \text{QUERY}, \text{ANSWER}, \text{RECONSTRUCT})$ of five algorithms:

- $(\mathsf{ck}, q_h) \leftarrow \text{SETUP}(1^\lambda, N)$: a randomized algorithm that outputs a client key (ck) and hint request (q_h), given the security parameter and database size.
- $h \leftarrow \text{HINT}(\mathsf{db}, q_h)$: a deterministic algorithm that outputs a client-specific hint, given the database and the hint request.
- $\mathsf{qu} \leftarrow \text{QUERY}(\mathsf{ck}, i)$: a randomized algorithm that generates a query, given the client key and a desired index.
- $\mathsf{ans} \leftarrow \text{RESPONSE}(\mathsf{db}, \mathsf{qu})$: a deterministic algorithm that produces an answer, given the query and the database.
- $d \leftarrow \text{EXTRACT}(h, \mathsf{ans})$: a deterministic algorithm that outputs a final response, given the hint and the answer.

Definition 1 (Correctness)**.** *A PIR protocol with preprocessing consisting of the five aforementioned routines is* δ*-correct, if for a domain* $\mathcal{D}$*, any database* $\mathsf{db} \in \mathcal{D}^N$ *and any* $i \in [N]$*,*

$$\mathbb{P}\left[\mathsf{db}[i] = f \;\middle|\; \begin{array}{rcl} (\mathsf{ck}, q_h) & \leftarrow & \textsc{Setup}(1^\lambda, N) \\ h & \leftarrow & \textsc{Hint}(\mathsf{db}, q_h) \\ \mathsf{qu} & \leftarrow & \textsc{Query}(\mathsf{ck}, i) \\ \mathsf{ans} & \leftarrow & \textsc{Response}(\mathsf{db}, \mathsf{qu}) \\ f & \leftarrow & \textsc{Extract}(h, \mathsf{ans}) \end{array}\right] > 1 - \delta \tag{1}$$

Intuitively, the two pieces of information that the server observes, q_h and qu, should not reveal any information about the client's query. This is formalized in the following definition for single-server security of offline/online PIR [32, Definition 46]:

Definition 2 (Security)**.** *A PIR protocol is* ε*-secure if for any PPT adversary* $\mathcal{A}$ *and any* $i, j \in [N]$*,*

$$\left|\mathbb{P}\left[\mathcal{A}(1^N, (q_h, \mathsf{qu})) = 1 \;\middle|\; \begin{array}{c} (\mathsf{ck}, q_h) \leftarrow \textsc{Setup}(1^\lambda, N) \\ \mathsf{qu} \leftarrow \textsc{Query}(\mathsf{ck}, i) \end{array}\right] - \mathbb{P}\left[\mathcal{A}(1^N, (q_h, \mathsf{qu})) = 1 \;\middle|\; \begin{array}{c} (\mathsf{ck}, q_h) \leftarrow \textsc{Setup}(1^\lambda, N) \\ \mathsf{qu} \leftarrow \textsc{Query}(\mathsf{ck}, j) \end{array}\right]\right| \leq \varepsilon$$

3 Additive HE for Smaller FHE Responses

We propose a technique to compress LWE/RLWE ciphertexts using auxiliary information provided by the client.

Exploiting Linear Phase Evaluation. In LWE and RLWE decryption, we compute an intermediate value which is commonly referred to as the *phase*, i.e., μ^* in Algorithm 1. Phase evaluation is linear and the phase is much smaller than the ciphertext itself. The main insight behind our solution is to homomorphically compute the phase on the server using encrypted values of the secret key, encrypted under an additive encryption scheme. Since the phase is much smaller than the original ciphertext, this results in a smaller response size. In general, our technique can be applied to any encryption scheme that has a linear phase evaluation. Examples of encryption schemes with this property are Regev [62], FHEW [38], TFHE [28], BFV [18, 40], and BGV [20].

In the following section, we describe the procedure for compressing LWE ciphertexts and follow with additional optimizations to this procedure. In Appendix B, we describe compression for RLWE-based ciphertexts as well.

The Additive Encryption Scheme. For the compression protocol, we require an additive encryption scheme which we denote $\mathcal{E}_A$ such that the plaintext space is $\mathbb{Z}_m$, for some m. Also, denote the ciphertext space of $\mathcal{E}_A$ as $\mathcal{C}$. $\mathcal{E}_A$ supports addition and plaintext multiplications. We denote addition and plaintext multiplication with $\oplus$ and $\otimes$, respectively. Moreover, denote the secret key generated by $\mathcal{E}_A$ as sk_A and the corresponding encryption and decryption algorithms as $\mathtt{AEnc}$ and $\mathtt{ADec}$.

Paillier [58], Damgard-Jurik [33], Exponential ElGamal [39], and Benaloh [17] are examples of cryptosystems that can be used for this purpose.

3.1 Compressing LWE Ciphertexts

The ciphertext compression algorithm for LWE and the corresponding modified decryption algorithm are given in Algorithm 2.

Algorithm 2 LWE compression, performed by the server and the corresponding modified decryption process, performed by the client over a compressed ciphertext. The compression key $\mathsf{ck} \in \mathcal{C}^n$ is such that $\mathsf{ck}[i] = \mathtt{AEnc}(\mathsf{sk}_A, \mathsf{sk}[i])$.

1: **procedure** $\textsc{LWECompress}_q(\mathsf{ck}, \mathsf{ct} = (\mathbf{a}, b))$ ▷ $\mathsf{ct} \in \mathbb{Z}^n \times \mathbb{Z}$
2: $\quad x = b$
3: $\quad$ **for** $i \in [n]$ **do**
4: $\quad\quad x \leftarrow x \oplus ((q - \mathbf{a}[i]) \otimes \mathsf{ck}[i])$
5: $\quad$ **return** x ▷ $\mu^* = \mathtt{ADec}(\mathsf{sk}_A, x)$

6: **procedure** $\textsc{ModifiedLWEDecrypt}_{q,p}(\mathsf{sk}_A, x)$
7: $\quad \mu^{**} = \mathtt{ADec}(\mathsf{sk}_A, x) \mod q$
8: $\quad \mu'' = \lfloor \mu^{**}/\Delta \rceil$ ▷ $\Delta = \lfloor q/p \rceil$
9: $\quad$ **return** $\mu'' \in \mathbb{Z}_p$

Theorem 1 (Correctness)**.** *For an LWE ciphertext* $\mathsf{ct} \in \mathbb{Z}_q^{n+1}$*, if* $m > q + nq^2$*, then* $\textsc{LWECompress}_q$ *produces a compressed ciphertext which decrypts to the correct message if decrypted using* $\textsc{ModifiedLWEDecrypt}$*. More formally, if*

$$x \leftarrow \textsc{LWECompress}_q(\mathsf{ck}, \mathsf{ct})$$
$$\mu'' \leftarrow \textsc{ModifiedLWEDecrypt}_{q,p}(\mathsf{sk}_A, x)$$

then $\mu'' = \textsc{LWEDecrypt}(\mathsf{sk}, \mathsf{ct})$

Proof. In the $\textsc{ModifiedLWEDecrypt}_{q,p}$ procedure (Line 7 of Algorithm 2), we calculate $b + \sum_{i\in[n]}(q - \mathbf{a}[i]) \cdot \mathsf{sk}[i]$, encrypted under additive encryption, which is achievable due to the linear properties of the additive encryption. We know that $\mathsf{sk}[i], \mathbf{a}[i]$ and b are elements in $\mathbb{Z}_q$ so $0 \leq \mathsf{sk}[i], \mathbf{a}[i], b < q$ and

$$b + \sum_{i\in[n]} (q - \mathbf{a}[i]) \cdot \mathsf{sk}[i] \leq q + \sum_{i\in[n]} q \cdot q = q + nq^2 < m. \tag{2}$$

so there is no overflow in the plaintext space of the additive ciphertext. In $\textsc{ModifiedLWEDecrypt}_{q,p}$ (Algorithm 2), we have

$$\begin{aligned} \mu^{**} \mod q &= \mathtt{ADec}(\mathsf{sk}_A, x) \mod q \\ &= \left(\left(b + \sum_{i\in[n]} (q - \mathbf{a}[i]) \cdot \mathsf{sk}[i]\right) \mod m\right) \mod q \\ &= \left(b + \sum_{i\in[n]} (q - \mathbf{a}[i]) \cdot \mathsf{sk}[i]\right) \mod q \\ &= b - \sum_{i\in[n]} \mathbf{a}[i] \cdot \mathsf{sk}[i] \mod q \end{aligned}$$

This is identical to μ^* in line 1 of Algorithm 1, hence, since the subsequent steps of LWEDECRYPT and MODIFIEDLWEDECRYPT are identical, they produce the same response, and the theorem is proven. □

In cryptosystems such as TFHE [28], the secret key is sampled from a binary distribution. In such a case, we can tighten the inequality required in Theorem 1 for correctness because $0 \leq \mathsf{sk}[i] \leq 1$. The following corollary summarizes this fact.

Corollary 1. *If the LWE secret key is binary and $m > q + nq$,* LWECOMPRESS *produces a compressed ciphertext which decrypts to the correct message if decrypted using* MODIFIEDLWEDECRYPT.

Security. In Gentry's original construction of a bounded depth encryption scheme, he proposed the idea of using a chain of semantically secure cryptosystems, such that each cryptosystem encrypts the secret key of the next [42]. Gentry proved that if the secret key of each cryptosystem is sampled independently, the composed scheme is also semantically secure.

Let $\mathcal{E}'$ denote the cryptosystem which is the chaining of $\mathcal{E}_{\text{LWE}}$ and $\mathcal{E}_A$. The encryption and decryption procedure of $\mathcal{E}'$ is shown in Algorithm 1 and Algorithm 2, respectively. The secret key of $\mathcal{E}'$ is the combination of the secret keys of $\mathcal{E}_{\text{LWE}}$ and $\mathcal{E}_A$. The same holds for the public key as well. Moreover, we also release encryptions of the bits of the secret key of $\mathcal{E}_{\text{LWE}}$ under the secret key of $\mathcal{E}_A$.

Proposition 1 (Security)**.** *If $\mathcal{E}_{LWE}$ and $\mathcal{E}_A$ are semantically secure, then $\mathcal{E}'$ is also semantically secure.*

3.2 Using Smaller Compression Keys

In practice, the plaintext space of the additive encryption system could be much larger than is required for the correctness of the compression technique to hold, i.e., $m \gg q + nq^2$. For example, the plaintext space of Paillier for 128-bit security is 3072 bits, which is much larger than $q + nq^2$ for any common choice of LWE parameters. We can use this gap to pack multiple digits of the LWE secret key within one additive ciphertext. Instead of encrypting each bit of the LWE key separately, we encrypt the first t digits of the secret key together into one packed additive ciphertext as $\mathtt{pck}_{0-t} = \mathtt{AEnc}(\mathsf{sk}_A, \sum_{i\in[t]} \mathsf{sk}[i] \cdot \delta^i)$ for a large enough δ. Specifically, δ should be such that $\delta > q + nq^2$ (or $\delta > q + nq$ in the case of binary keys). On the server side, the server unpacks the secret key by computing $\mathsf{ck}[i] = \delta^{t-1-i} \otimes \mathtt{pck}_{0-t}$ for $i \in [t]$. Compression proceeds as before, with the only difference being that the encrypted phase, calculated by the server in the additive ciphertext, is scaled by a factor of δ^{t-1}. Appendix A details the procedures for generating the packed key, unpacking it, and the corresponding modified LWE decryption function. We use the same function for compressing the ciphertext.

3.3 Batched Compression

To achieve better compression, multiple LWE ciphertexts (encrypted using the same secret key) can be compressed within the same additive ciphertext, which we denote as *batched compression*. Each LWE ciphertext takes up $\log_2(q + nq^2)$ bits of the total bitwidth of the plaintext space. So, if m is the modulus of the plaintext space, then $\lfloor \log_2 m / \log_2(q + nq^2) \rfloor$ LWE ciphertexts can be compressed into one ciphertext from the additive cryptosystem.

Algorithm 3 illustrates how to compress ℓ LWE ciphertexts within one additive ciphertext. The corresponding decryption procedure is also shown. Using LWECOMPRESS as a subprocedure allows for better parallelization when compressing many LWE ciphertexts.

Algorithm 3 Batch Compression of LWE ciphertexts by the server and the modified decryption procedure, performed by the client. The compression key ck is such that $\mathsf{sk}[i] = \mathtt{ADec}(\mathsf{sk}_A, \mathsf{ck}[i])$ and $cts = \{c_j\}_{j\in[\ell]}$ such that $c_j = (\mathbf{a}_j, b_j) \in \mathbb{Z}_q^n \times \mathbb{Z}$ and $\gamma = q + nq^2$.

1: **procedure** BATCHEDLWECOMPRESS$_{q,\gamma}(\mathsf{ck}, cts = \{c_j\}_{j\in[\ell]})$
2: **for** $j \in [\ell]$ **do**
3: $x_j \leftarrow \text{LWECOMPRESS}_q(\mathsf{ck}, c_j)$
4: $x \leftarrow x \oplus \gamma^j x_j$
return x

5: **procedure** MODIFIEDBATCHEDLWEDECRYPT$_{q,p,\gamma}(\mathsf{sk}_A, x)$
6: $\mu^{**} = \mathtt{ADec}(\mathsf{sk}_A, x)$
7: **for** $j \in [\ell]$ **do**
8: $\mu_j^{**} = \lfloor \mu^{**}/\gamma^j \rceil \mod \gamma$
9: $\mu_j'' = \lfloor \frac{\mu_j^{**} \bmod q}{\Delta} \rceil$ ▷ $\Delta = \lfloor q/p \rceil$
return $\{\mu_j'' \in \mathbb{Z}_p\}_{j\in[\ell]}$

Theorem 2 (Correctness)**.** *Let $\mathsf{ct} = \{c_j\}_{j\in[\ell]}$ be a vector of ℓ LWE ciphertexts. For $\gamma \geq q + nq^2$, if $m > \gamma^\ell$, then* BATCHEDLWECOMPRESS$_{q,\gamma}$ *produces a compressed ciphertext which, if decrypted using the corresponding modified decryption, decrypts to the vector of ℓ plaintexts. More formally, if*

$$x \leftarrow \text{BATCHEDLWECOMPRESS}(\mathsf{ck}, \mathsf{ct}, k)$$
$$\{\mu_j'\}_{j\in\ell} \leftarrow \text{MODIFIEDBATCHEDLWEDECRYPT}(\mathsf{sk}_A, x)$$

then $\mu_j' = LWEDecrypt(\mathsf{sk}, c_j)$.

Proof. By the proof of Theorem 1 we know that if $\mu_j^{**} = \mathtt{ADec}_\mathsf{s}(x_j)$, then $0 \leq \mu_j^{**} < \gamma = q + nq^2$. Hence, we have

$$\mu^{**} = \sum_{j\in[\ell]} \gamma^j \mu_j^{**} \leq \sum_{j\in[\ell]} \gamma^j(\gamma - 1) = \gamma^\ell - 1 < \gamma^\ell < m.$$

Hence, the plaintext corresponding to x, i.e., μ^{**}, does not overflow in the plaintext space of the additive ciphertext. If

Algorithm 4 Batch compression of LWE ciphertexts using precomputed powers. The compression key ck is such that $\mathsf{sk}[i] = \mathtt{ADec}(\mathsf{sk}_A, \mathsf{ck}[i])$ and $cts = \{c_j\}_{j\in[\ell]}$ such that $c_j = (\mathbf{a}_j, b_j) \in \mathbb{Z}_q^n \times \mathbb{Z}$.

1: **procedure** ExpandCompressionKey$_q$(ck)
2: $\quad\mathsf{eck}[0] = \mathsf{ck}$
3: $\quad$**for** $i \in [t-1]$ **do** ▷ $t = \lceil \log_2 q \rceil$
4: $\quad\quad$**for** $j \in [n]$ **do**
5: $\quad\quad\quad\mathsf{eck}[i+1][j] = \mathsf{eck}[i][j] \oplus \mathsf{eck}[i][j]$
$\quad$**return** eck

6: **procedure** FastLWECompress$_q$($\mathsf{eck}, \mathsf{ct} = (\mathbf{a}, b)$)
7: $\quad x = b$
8: $\quad$**for** $i \in [n]$ **do**
9: $\quad\quad(b_{t-1} \cdots b_1 b_0)_2 \leftarrow (q - \mathbf{a}[i]) \mod q$ ▷ $t = \lceil \log_2 q \rceil$
10: $\quad\quad$**for** $j \in [t]$ **do**
11: $\quad\quad\quad$**if** $b_j = 1$ **then**
12: $\quad\quad\quad\quad x \leftarrow x \oplus \mathsf{eck}[j][i]$
$\quad$**return** x ▷ $\mu^* = \mathtt{ADec}(\mathsf{sk}_A, x)$

13: **procedure** FastBatchedLWECompress$_{q,\gamma}$($\mathsf{ck}, cts = \{c_j\}_{j\in[\ell]}$)
14: $\quad\mathsf{eck} \leftarrow$ ExpandCompressionKey$_q$(ck)
15: $\quad\gamma = q + nq^2$
16: $\quad$**for** $j \in [\ell]$ **do**
17: $\quad\quad x_j \leftarrow$ FastLWECompress$_q$(eck, c_j)
18: $\quad\quad x \leftarrow x \oplus \gamma^j x_j$
$\quad$**return** x

μ_j^* is equivalent to μ^* in the LWEEncrypt procedure, then for some value t,

$$\mu_j'' = \lfloor \mu^{**}/\gamma^j \rfloor \mod \gamma = (\mu_j^* + \gamma \cdot t) \mod \gamma = \mu_j^*$$

and the subsequent steps are similar, which proves the theorem. □

3.3.1 Faster Batched Compression with Expanded Key

Compression makes use of expensive operations in the additive scheme. The plaintext multiplication in Line 4 of Algorithm 2 is the most expensive operation. In additive schemes such as Paillier and ElGamal, this is equivalent to a modular exponentiation in a large group.

In the batched setting, we can reduce the overhead by precomputing and reusing multiples of the bits of the secret key. If we decompose $(q - \mathbf{a}[i])$ as $(b_{t-1} \cdots b_1 b_0)_2 = (q - \mathbf{a}[i]) \mod q$ we compute the plaintext multiplication as follows

$$(q - \mathbf{a}[i]) \otimes \mathsf{ck}[i] \tag{3}$$
$$= 2^{t-1} b_{t-1} \mathsf{ck}[i] + \cdots + 2 b_1 \mathsf{ck}[i] + b_0 \mathsf{ck}[i] \tag{4}$$

and we can precompute an *extended compression key*, eck, such that $\mathsf{eck}[i][j] = 2^j \mathsf{ck}[i]$ for $j \in [t]$, which can be reused for all LWE ciphertexts we want to compress. Algorithm 4 shows the procedure extending the compression key and compressing LWE ciphertexts using the extended compression key.

Corollary 2. *Let* $\mathsf{ct} = \{c_j\}_{j\in[\ell]}$ *be a vector of* ℓ *LWE ciphertexts. For* $\gamma \geq q + nq^2$*, if* $m > \gamma^\ell$*, if*

$$x \leftarrow \text{FastBatchedLWECompress}(\mathsf{eck}, \mathsf{ct}, k)$$
$$\{\mu_j'\}_{j\in\ell} \leftarrow \text{ModifiedBatchedLWEDecrypt}(\mathsf{sk}_A, x)$$

then $\mu_j' = LWEDecrypt(\mathsf{sk}, c_j)$.

3.3.2 Rescaling for Compression

In some instances, it is possible to rescale the elements in the ciphertext to a smaller modulus without altering the underlying message. This technique, also called modulus switching, is commonly used in the literature to simplify the decryption procedure or control noise growth [18]. However, rescaling is only possible if the noise of the underlying LWE ciphertext is less than a given bound. In Appendix **??**, we prove how rescaling is possible for LWE ciphertexts with binary keys, if the noise is less than a certain bound, i.e., less than $\Delta/4$. Rescaling to a smaller modulus accelerates our compression technique since the scalar multiplication in the additive encryption scheme is done with a smaller scalar.

3.3.3 Better compression with a smaller scale

The number of LWE ciphertexts that fit within each additive ciphertext is determined by the scale, i.e., $\gamma = q + nq^2$. Using a smaller scale would allow us to pack more LWE ciphertexts within each additive ciphertext. There are two instances where we can use a smaller scale. First, when the LWE secret key is binary. In that case, we can use $\gamma = q + nq$ as the scale. This follows from the fact that in the case of binary keys, $0 < \mu_j^{**} \leq \gamma = q + nq$.

The second instance where we can reduce the scale is by allowing μ_j^{**} and μ_{j+1}^{**} to overlap in the additive scheme. Intuitively, this is possible because the high-order bits of μ_j^{**} are removed when it is taken modulo q as part of the modified decryption. The lower order bits of μ_{j+1}^{**} are also rounded during the modified decryption so it is possible to add additional error, as long as it does not interfere with the message. Specifically, if $|e| < \Delta/4$ (instead of the usual condition where $|e| < \Delta/2$ for correct decryption), we can reduce the scale to $\gamma = q^2$ and $\gamma = q$ in the case of non-binary and binary keys, respectively. Due to space restrictions, we provide proof of the correctness of this technique using a smaller scale under these conditions in the full version of the paper.

4 Our PIR Construction: ZipPIR

In this section, we present ZipPIR, a high-throughput single-server PIR protocol optimized for resource-bound clients. ZipPIR uses the compression technique from the previous section but pushes all expensive operations to the offline phase. We show how we benefit from our proposed compression technique whilst maintaining an efficient online phase.

4.1 Leveraging the offline phase

As we saw in the previous section, our compression technique is very effective in reducing the size of LWE ciphertexts.

However, this comes at a significant cost due to the expensive operations in the additive scheme. In the case of Paillier, multiplications are done via expensive exponentiations.

To avoid performing expensive operations in the online phase, we note an observation by Beck [16], which states that a Paillier encryption of a message $\mu \in \mathbb{Z}_m$ can be represented as

$$\mathsf{Enc}(\mu) = \mathsf{Enc}(r) + (\mu - r) \tag{5}$$

where r is a uniformly random value over the plaintext space of the encryption scheme. Moreover, instead of sampling r and encrypting it, we can directly sample the ciphertext $\mathsf{Enc}(r)$ from $\mathbb{Z}_{m^2}^2$ using a PRNG seed. While valid Paillier ciphertexts are elements of $\mathbb{Z}_{m^2}^*$, the likelihood of sampling an invalid ciphertext is negligible in the security parameter (roughly $2/\sqrt{m}$).

Using this observation, we can evaluate any linear function (such as the compression function) over Paillier ciphertexts in two steps. First, we compute the function over the random component $\mathsf{Enc}(r)$, which does not depend on the message μ. Then we compute the function over $(\mu - r)$, which we call the *offset*, and add this to the result. Using this approach, all expensive operations over the additive ciphertexts are performed in an offline phase, before the value of μ is known.

4.2 ZipPIR Description

In this section, we describe our construction, ZipPIR. We also show how, through simple modifications, ZipPIR requires no storage overhead for the client and, under computational assumptions, the offline processing can be done silently, except for an initial public key and seed.

ZipPIR builds on SimplePIR [44] but eliminates the need for the client to store a large database-dependent hint by compressing the hint using the algorithms described in Section 3. However, since the compression is a linear operation, we can leverage the online phase, as described in the previous section. While we do not use the precise procedures described in Section 3, the operations in ZipPIR perform the same function in principle, albeit in an offline/online manner.

Algorithm 5 shows a simple description of ZipPIR, and we describe additional modifications and enhancements in the following sections. At a high level, the protocol works as follows. In the offline phase:

- At setup (SETUP), the client generates the necessary Paillier public keys and samples the Paillier ciphertexts, which are the random components of the compression keys.
- In the offline hint generation (HINT), the server generates a compressed version of the Paillier hint given the client's Paillier public keys and the sampled ciphertexts. To avoid client-side storage, the server holds this hint until the client issues a query.

Once the client's query is known, the online phase occurs as follows:

- The client's query consists of two components: the offset of the compression key and the LWE query (from SimplePIR).
- The server performs two matrix multiplications — one over the database and another over the hint. Both matrix multiplications are done over modular integers. The server sends the response and the associated hint to the client.
- The client extracts the answer from the server's response.

Algorithm 5 Simple description of ZipPIR. q, n, and p are the LWE ciphertext modulus, ciphertext dimension, and plaintext modulus, respectively. Database $\mathsf{db} \in \mathbb{Z}_p^{d_0 \times d_1}$ where $N = d_0 d_1$. Also, assume $\mathbf{A} \leftarrow\$ \mathbb{Z}_q^{d_0 \times n}$ and the server also computes $\mathbf{H} = -\mathsf{db}^T \cdot \mathbf{A} \in \mathbb{Z}_q^{d_1 \times n}$. PAILLIERMATMUL denotes a multiplication between a plaintext matrix and an encrypted vector.

procedure SETUP(1^λ, (d_0, d_1)) ▷ Client
  Generate Paillier keys $(\mathsf{sk}_P, \mathsf{pk}_P = m)$ with λ-bit security
  Sample $\mathsf{ck}_r \leftarrow\$ \mathbb{Z}_{m^2}^n$
  $\mathsf{pt}_r \leftarrow$ PAILLIERDECRYPT$(\mathsf{sk}_P, \mathsf{ck}_r)$
  return $((\mathsf{sk}_P, \mathsf{pt}_r), (m, \mathsf{ck}_r))$

procedure HINT($\mathbf{H} \in \mathbb{Z}_q^{d_1 \times n}$, (m, ck_r)) ▷ Server
  $\mathbf{k} \leftarrow$ PAILLIERMATMUL$_m(\mathbf{H}, \mathsf{ck}_r)$ ▷ $\mathbf{k} \in \mathbb{Z}_{m^2}^{d_1}$
  return $\mathbf{k}$

procedure QUERY($i_0 \in [d_0]$, pt_r) ▷ Client
  Sample LWE key $\mathsf{sk} \leftarrow\$ \{0,1\}^n$
  $\mathsf{ck}_o = (\mathsf{sk} - \mathsf{pt}_r) \mod m$
  $u_0 =$ selection vector for index i_0
  Sample $e \leftarrow \chi_e^{d_0}$
  $\mathsf{qu}_0 = \mathbf{A} \cdot \mathsf{sk} + e + \Delta \cdot u_0 \mod q$ ▷ $\Delta = \lfloor q/p \rfloor$
  return $(\mathsf{ck}_o, \mathsf{qu}_0)$ ▷ $\mathsf{ck}_o \in \mathbb{Z}_m^n, \mathsf{qu}_0 \in \mathbb{Z}_q^{d_0}$

procedure RESPONSE($\mathsf{db}, \mathbf{H}, \mathsf{qu} = (\mathsf{ck}_o, \mathsf{qu}_0)$) ▷ Server
  $b = \mathsf{db}^T \cdot \mathsf{qu}_0 \mod q$
  $t = b + \mathbf{H} \cdot \mathsf{ck}_o \mod m$
  return t

procedure EXTRACT(sk_P, $(\mathbf{k}, t)$) ▷ Client
  $\mu = (t + $ PAILLIERDECRYPT$(\mathsf{sk}_P, \mathbf{k})) \mod m$
  return $f = \lfloor (\mu \mod q)p/q \rceil \mod p$

The following theorems hold for the correctness and security of ZipPIR. The full proof of correctness and security are provided in Appendix D.

Theorem 3. *(ZipPIR Correctness) For LWE parameters (n, q, χ_e, χ_s) where χ_s is a discrete Gaussian with standard deviation σ, and χ_s is a binary distribution, and plaintext modulus p, and failure rate δ such that*

$$q/p > 2p\sigma\sqrt{2d_0 \ln(2/\delta)} \tag{6}$$

for random $\mathbf{A} \in \mathbb{Z}_q^{d_0 \times n}$, for any database $\mathsf{db} \in \mathbb{Z}_p^{d_0 \times d_1}$, $\mathbf{H} = -\mathsf{db}^T \cdot \mathbf{A}$, Paillier modulus m such that $m > q + nq$, and any

query $i_0 \in [d_0]$*, if*

$$\begin{aligned}((\mathsf{sk}_P, \mathsf{pt}_r), (m, \mathsf{ck}_r)) &\leftarrow \textsc{Setup}(1^\lambda, (d_0, d_1)) \\ \boldsymbol{k} &\leftarrow \textsc{Hint}(\boldsymbol{H}, (m, \mathsf{ck}_r)) \\ (\mathsf{ck}_o, \mathsf{qu}_0) &\leftarrow \textsc{Query}(i_0, \mathsf{pt}_r) \\ t &\leftarrow \textsc{Response}(\mathsf{db}, \boldsymbol{H}, (\mathsf{ck}_o, \mathsf{qu}_0)) \\ f &\leftarrow \textsc{Extract}(\mathsf{sk}_P, (\boldsymbol{k}, t))\end{aligned}$$

then $\mathbb{P}[\mathsf{db}[i_0] = f] > 1 - \delta - 2d_0/\sqrt{m}$.

Theorem 4. *(ZipPIR Security) Assume we have secure LWE parameters* (n, q, χ_e, χ_s) *where* χ_s *is a discrete Gaussian with standard deviation* σ*, and* χ_s *is a binary distribution. Also, assume we have secure Paillier parameters. Then for any PPT adversary* $\mathcal{A}$ *and any* $i, j \in [d_0]$*, we have*

$$\left| \mathbb{P}\left[\mathcal{A}(1^\lambda, (q_h, \mathsf{qu})) = 1 \;\middle|\; \begin{array}{c} ((\_, \mathsf{pt}_r), q_h) \leftarrow \textsc{Setup}(1^\lambda, (d_0, d_1)) \\ \mathsf{qu} \leftarrow \textsc{Query}(i, \mathsf{pt}_r) \end{array} \right] \right.$$
$$\left. - \mathbb{P}\left[\mathcal{A}(1^\lambda, (q_h, \mathsf{qu})) = 1 \;\middle|\; \begin{array}{c} ((\_, \mathsf{pt}_r), q_h) \leftarrow \textsc{Setup}(1^\lambda, (d_0, d_1)) \\ \mathsf{qu} \leftarrow \textsc{Query}(j, \mathsf{pt}_r) \end{array} \right] \right| < \varepsilon$$

where ε *is negligible in the security parameter.*

4.3 Additional Optimizations & Tradeoffs.

The description of ZipPIR from Algorithm 5 can be modified to provide bandwidth savings and reduce computational and storage burden for the client.

Faster matrix multiplication using RNS. The second matrix multiplication in the online phase ($\mathbf{H} \cdot \mathsf{ck}_o$) is performed over large integers, which is slow. To be able to speed up the matrix multiplication, we represent the two operands of the multiplication in the Residue Number System (RNS) format. The hint ($\mathbf{H}$) can be converted to RNS in the offline phase so we only require the offset vector (ck_o) to be converted during the online phase. Using this technique, we can perform the second matrix multiplication over native machine operations.

Batched compression for smaller responses. A more efficient instantiation of ZipPIR uses our batched compression technique instead of single compression. This is possible because the scaling used in batched compression is also a linear operation, and hence can be split into offline and online operations. Batched compression not only reduces the size of the response, but also results in smaller matrix-vector multiplication in Line 18. We can also use a small scale as mentioned in Section 3.3.3.

Lowering offline communication costs. To achieve a (nearly) silent offline phase, we generate ck_r using a PRG seed. The client and server only need to agree on the initial seed for the pseudorandom generator. Moreover, this seed can be used to generate a large number of client hints (polynomial in the security parameter) for all future queries. The client can also store the seed, alongside its Paillier private key, to enable queries in the future, all of which require constant-size storage. One consequence is that the client delays decryption of ck_r (Line 4) to the Query step.

Smaller responses w/ Paillier addition. To further reduce the response size, the online response (t), can be combined with the offline hint ($\mathbf{k}$) instead of being sent separately, at the cost of a Paillier addition. We can then send only one Paillier ciphertext instead of one ciphertext and one plaintext, reducing the response size by 30%.

Supporting multiple queries. The current description of ZipPIR supports one query from the client. To extend this to multiple queries, the client can generate multiple ck_r values, for as many queries as it wishes to issue, and send them to the server. Using a PRG, as mentioned in the previous paragraph, the offline bandwidth stays constant, even for multiple queries. The server offline time and server per-client storage scales linearly with the number of queries that the server wants to support for that client, without additional offline communication.

Reusability of the offline phase. Similar to SimplePIR, the computation of the global hint $\mathbf{H}$ is reused for all queries from all clients. The client-specific hint computation (Line 7) is not reusable across clients because it depends on the client's public key and must be computed separately for each prospective client.

ZipPIR with client storage. Although we presented ZipPIR as a client-efficient PIR protocol, it can also be modified to use client storage, similar to the work of Patel et al. [59]. To achieve this, the client simply retrieves and stores the hint ($\mathbf{k}$) in the offline phase for as many queries as it will make. A hint is consumed for each query made by the client, similar to the client state used in PSIR [59].

4.4 Concrete Costs of ZipPIR

To achieve the best concrete performance (demonstrated in Section 5.3), we include the first three techniques from Section 4.3 in our final construction. We report the concrete costs of this augmented protocol in this section.

The concrete number of operations in each routine, along with the party that performs those operations, is listed below. Let $\gamma = \lceil (\log_2 m - \log_2 n)/\log_2 q \rceil$ and $d_1' = \lfloor d_1/\gamma \rfloor$, then

- Setup (Client): Paillier key generation
- Hint (Server) : $d_1' n$ Paillier plaintext multiplications and additions (Line 7)
- Query (Client): d_0 LWE encryptions, (Line 14), n random samples in $\mathbb{Z}_m$, and n Paillier decryptions (Line 4)
- Response (Server): $N = d_0 d_1$ multiplications and additions in $\mathbb{Z}_q$ (Line 17) and $d_1' n$ multiplications and additions in $\mathbb{Z}_m$ (Line 18).
- Extract (Client): d_1' Paillier decryptions

Similarly, the concrete communication costs (in bits) of the protocol are listed below.

Table 2: Evaluation of the ciphertext compression technique for a single LWE ciphertext. Three sample parameter sets are chosen for LWE-based ciphertexts. The first three columns are common parameter sets used in the Concrete library [65]. The last configuration is the STD128 configuration for CGGI in OpenFHE [8].

Parameters	LWE $(n, \log_2 q)$				RLWE $(N, \log_2 q)$			
	(630, 64)	(742, 64)	(870, 64)	(1305, 11)	(1024,27)	(2048,54)	(4096,36)	(8192,43)
Compression Time	9.7 ms	11.0 ms	12.9 ms	16.6 ms	7.2 ms	23.8 ms	33.8 ms	83.3 ms
Compressed Ciphertext	768 B	768 B	768 B	768 B	768 B	768 B	768 B	768 B
Uncompressed Ciphertext	5.05 KB	5.94 KB	6.97 KB	1.80 KB	3.46 KB	13.83 KB	18.44 KB	44.04 KB
Size Reduction	**84.78 %**	**87.08 %**	**88.98%**	**57.23%**	**77.80%**	**94.45%**	**95.83%**	**98.26%**

- Client to Server (Offline, one-time): $2\log_2 m + \lambda$
- Client to Server (Online, per-query) : $n \log_2 m + d_0 \log_2 q$
- Server to Client (Online, per-query) : $3d_1' \log_2 m$

The server-side storage per client is $2d_1' \log_2 m$ bits for each query. The client-side storage is constant, only the Paillier private key and the seed used to generate ciphertexts, which is $\log_2 m + \lambda$ bits in total.

5 Evaluation

We present our evaluation in two sections. First, evaluating the compression of LWE ciphertexts using our proposed method compared to existing approaches in the literature. Second, we compare our PIR construction, ZipPIR, with PIR protocols in the literature.

5.1 Evaluating Ciphertext Compression

We implemented our compression technique as a library in C++ using GMP. We use Paillier as the additive encryption scheme, which is extended to Damgard-Jurik when we require a larger plaintext space. We use a 3072-bit modulus for Paillier, composed of two 1536-bit primes, which is the recommended modulus size for 128-bit security [15]. We experiment with LWE and RLWE parameters satisfying 128-bit security but our methods can be applied to other LWE and RLWE parameters without any change.

We also parallelized our implementation to minimize the latency of the compression. Specifically, we parallelize over the LWE dimension n or the number of LWE ciphertexts that are compressed, depending on whichever is larger. Using this dynamic approach, we use existing cores on our machines even when compressing few ciphertexts.

We experiment under two scenarios: 1) compressing a single LWE ciphertext or RLWE coefficient and 2) compressing many LWE ciphertexts or multiple RLWE coefficients. The former is useful in applications with small outputs such as private inference, whereas the latter is used for applications with large outputs such as image processing.

5.1.1 Single Compression Evaluation

Table 2 summarizes the results for compressing a single LWE ciphertext. We choose LWE parameters adopted in existing libraries implementing variants of LWE encryption [8, 65]. The results show that we consistently provide high compression rates. Notably, for $\log_2 q = 64$, our compression rates are over 84%.

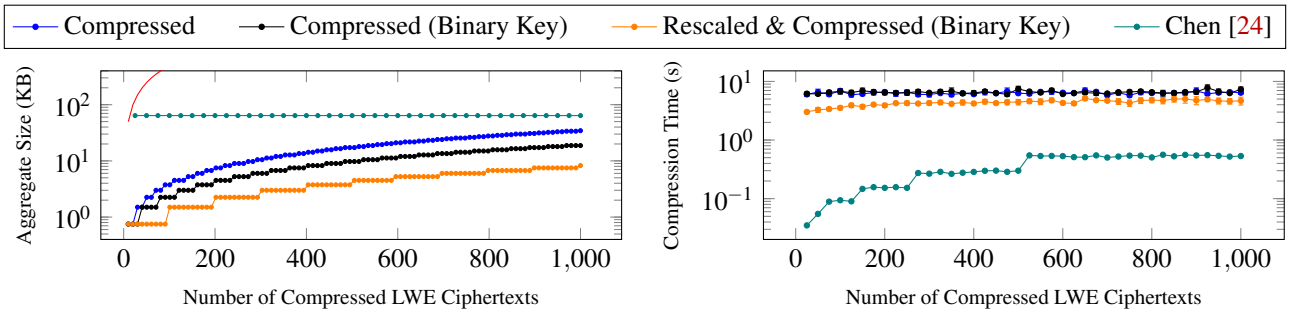
Similarly, for compression of a single RLWE coefficient, we use common parameters for RLWE-based schemes such as BFV [18, 40] and BGV [20], which are used in libraries such as SEAL [2], Lattigo [3] and OpenFHE [8]. Recall that compression is compatible with modulus switching so we choose the parameters corresponding to the lowest level in a BFV/BGV parameter set. We achieve over 85% compression and up to 98%.

5.1.2 Measuring Compression Key Sizes

Using the technique described in Section 3.2, we can pack the compression key and have the server unpack the key. The unpacking can be done offline, as soon as the server receives the packed key, to reduce latency during compression. The compression procedure is identical after the key has been unpacked, so we do not report the runtime of compression again. Instead, we measure the size of the compression key, with and without packing, and report the time required to unpack the key. We also distinguish two cases, non-binary and binary keys. In the case of binary keys, we use $\delta = q + nq$ so more bits of the secret key can fit within the same ciphertext. Table 3 shows the size of the packed compression keys in the two cases. Note that even the size of the unpacked key is much smaller than commonly used cryptographic keys such as relinearization keys, automorphism keys, and bootstrapping keys, which could be as large as 100 MB.

5.2 Batched Compression Evaluation

Next, we evaluate the batched compression of LWE ciphertexts. In both cases, we use the batched compression algorithm to compress ℓ ciphertexts into additive Paillier ciphertexts. Note that for batched compression, we can not use packed



(a) Compressed Size

(b) Time for Compression

Figure 1: Compressed size and compression time required for compressing LWE ciphertexts with $(n,q) = (630, 2^{64})$ using batched compression. The red line denotes the baseline size of uncompressed LWE ciphertexts.

compression keys.

We distinguish the case of binary keys from non-binary keys in the experiment, denoted by blue and black lines, respectively. As mentioned in Section 3.3, we set the scale to $\gamma = q + nq^2$ and $\gamma = q + nq$ in the case of non-binary and binary keys, respectively. We also visualize a case where we rescale to a smaller modulus before compression.

The alternative approach to packing many LWE ciphertexts is RLWE packing [24, 28]. Our reference point for RLWE packing is the work of Chen et al. [24] which is state-of-the-art in compression and has better runtime than related work. This work maps LWE ciphertexts in $\mathbb{Z}_q^n \times \mathbb{Z}$ to an RLWE ciphertext in $\mathbb{Z}_q[X]/(X^n+1)$.

Figure 1a shows the size after compression as a function of the number of compressed LWE ciphertexts. Figure 1b also shows the runtime of batched compression. The blue and black plots correspond to non-binary and binary keys, respectively. The orange plot also shows batched compression but over ciphertexts that are rescaled to $r = 2^{20}$. In the case of non-binary keys, we only need one Paillier ciphertext for up to 14 LWE ciphertexts and for binary keys, it is about 26. Overall, compressing more LWE ciphertexts offers more compression compared to compressing only one LWE ciphertext. This is because more of the plaintext space of Paillier is being utilized as more ciphertexts are compressed. We also observe that rescaling improves the runtime and allows more compression, as is expected. In comparison to the work of Chen et al., our compression protocol offers more than one order of magnitude better compression compared to RLWE packing. However, our compression approach is slower than RLWE packing, particularly due to the use of expensive modular exponentiations. In summary, in cases where there are fewer items to pack, our compression techniques offers much better compression while in cases with many LWE ciphertexts to pack, the work of Chen et al. is the more suitable solution.

Table 3: Size of packed compression keys and unpacking time. We distinguish the case of binary and non-binary keys since binary keys can be packed more than non-binary keys. The Paillier modulus is 3072 bits in all cases.

$(n, \log_2 q)$	(630,64)	(742,64)	(870,64)	(1305,11)
Unpacked Key	240 KB	284 KB	334 KB	501 KB
Packed Non-binary Key	22 KB	26 KB	30 KB	11 KB
Unpacking Time	14 ms	25 ms	74 ms	15 ms
Packed Binary Key	12 KB	14 KB	16 KB	7 KB
Unpacking Time	13 ms	12 ms	13 ms	15 ms

5.3 PIR Evaluation

In our PIR evaluation, we compare with two classes of protocols. First, we compare with other protocols where the server stores a state for the client. We show how the state in ZipPIR differs from that of related work and how we achieve higher throughput than similar works in this model. We also evaluate ZipPIR as a protocol with client-side storage.

Implementation & Experimental Considerations. We implement ZipPIR in C++ using AVX512 and the libhcs library, which supports large integer arithmetic for Paillier operations. For the LWE parameter, we use $(n,q) = (1400, 2^{32})$ and a discrete Gaussian distribution with a standard deviation of 6.4, which gives over 128-bit security as shown by the lattice-estimator [9]. We use the same parameters for the Paillier encryption that we used in the previous section. Offline computation, which involves Paillier operations, is parallelized but online computation is single-threaded for a fair comparison with related work. We perform the second matrix multiplication $(\mathbf{H} \cdot \mathsf{ck}_o)$ over an RNS basis consisting of 240 prime moduli of 27-bit length. This allows us to perform the matrix multiplication within the machine word size, without the need to perform modular reduction until the end of the computation, significantly enhancing performance.

For our experiments, we use the reference implementation of each protocol when it is provided. All experiments are performed 5 times and the average statistics are reported. We

Table 4: Comparison of communication and computation costs with PIR protocols with per-client storage. All protocols retrieve a payload of at least 16 KB. * In all protocols except ZipPIR, the offline communication is equal to the per-client storage. In ZipPIR, the offline communication is 400 B.

DB	Metric	SealPIR [13]	FastPIR [6]	OnionPIR [56]	Spiral [54]	HintlessPIR [47]	YPIR [55]	ZipPIR
256 MB	Per-client Storage	1215 KB	670 KB	5538 KB	12432 KB	-	-	60 KB
	Query	90 KB	832 KB	256 KB	14 KB	424 KB	518 KB	617 KB
	Response	181 KB	64 KB	128 KB	20 KB	964 KB	120 KB	91 KB
	Total Comm.	271 KB	896 KB	384 KB	34 KB	1388 KB	638 KB	708 KB
	Server Time	2455 ms	1851 ms	3378 ms	1633 ms	662 ms	388 ms	171 ms
	Throughput	104 MB/s	138 MB/s	75 MB/s	156 MB/s	383 MB/s	659 MB/s	1497 MB/s
512 MB	Per-client Storage	1215 KB	670 KB	5538 KB	12656 KB	-	-	85 KB
	Query	90 KB	1664 KB	256 KB	14 KB	448 KB	574 KB	655 KB
	Response	181 KB	64 KB	128 KB	20 KB	964 KB	120 KB	128 KB
	Total Comm.	271 KB	1728 KB	384 KB	34 KB	1452 KB	694 KB	783 KB
	Server Time	4564 ms	2808 ms	5865 ms	3031 ms	674 ms	425 ms	227 ms
	Throughput	112 MB/s	182 MB/s	87 MB/s	168 MB/s	759 MB/s	1204 MB/s	2255 MB/s
1 GB	Per-client Storage	1350 KB	670 KB	5538 KB	12656 KB	-	-	120 KB
	Query	90 KB	3328 KB	256 KB	14 KB	488 KB	686 KB	708 KB
	Response	181 KB	64 KB	128 KB	20 KB	1748 KB	120 KB	181 KB
	Total Comm.	271 KB	3392 KB	384 KB	34 KB	2236 KB	806 KB	889 KB
	Server Time	9071 ms	4698 ms	9662 ms	3080 ms	971 ms	515 ms	339 ms
	Throughput	112 MB/s	217 MB/s	105 MB/s	332 MB/s	1053 MB/s	1988 MB/s	3020 MB/s
2 GB	Per-client Storage	1350 KB	670 KB	5538 KB	14224 KB	-	-	170 KB
	Query	90 KB	6592 KB	256 KB	14 KB	616 KB	910 KB	782 KB
	Response	181 KB	64 KB	128 KB	20 KB	1748 KB	120 KB	255 KB
	Total Comm.	271 KB	6656 KB	384 KB	34 KB	2364 KB	1030 KB	1037 KB
	Server Time	17692 ms	8388 ms	19569 ms	4435 ms	1232 ms	704 ms	580 ms
	Throughput	115 MB/s	244 MB/s	104 MB/s	461 MB/s	1661 MB/s	2909 MB/s	3531 MB/s

conduct all experiments on an Intel(R) Xeon(R) Platinum 8276 CPU running Ubuntu 24.04. Our code is open-source and available [1].

5.3.1 Evaluation of PIR without Client Storage

In this section, we compare with state-of-the-art client-efficient single-server PIR protocols. This consists of protocols that require per-client storage on the server, i.e., SealPIR [13], FastPIR [6], OnionPIR [56], Spiral [54]. Protocols with silent preprocessing (that also support large payloads), such as YPIR (using SimplePIR) [55], and HintlessPIR [47] are also relevant and are included in this category. In each protocol, we measure the per-client storage required by the server, the communication costs, and the runtime. We report the online server throughput, a metric proposed by Henzinger et al. [44], which shows how many database elements are processed per unit of time. Table 4 summarizes the results of this comparison.

ZipPIR has a much higher throughput than all works in this category. This can be attributed to the simple online phase, which only consists of two matrix multiplications. In contrast, all other protocols involve operations over RLWE ciphertexts. Compared to the fastest alternative, YPIR, ZipPIR is at least 20% faster in all cases, and over 2x faster for some database sizes.

The setup of ZipPIR only requires a one-time communication cost of 400 B, which consists of the clients Paillier public key and cryptographic seed. In all instances of ZipPIR, 527 KB of the total query size consists of the compression key offsets (ck_o), and the remainder is the LWE query vector (qu_o). The online communication cost of ZipPIR is similar to that of HintlessPIR and YPIR. SealPIR, OnionPIR, and Spiral have smaller per-query communication costs, since they are highly optimized for this metric at the cost of throughput and server-side storage.

5.3.2 Evaluation of PIR with Client-side Storage

As described in Section 4.3, we can extend ZipPIR to leverage client-side storage as well. ZipPIR with client storage requires less communication in the online phase. Given that ZipPIR is a protocol with linear online time, we compare it with other protocols that also fall in this category. Existing protocols

[1] https://github.com/RasoulAM/ZipPIR

Table 5: Comparison of communication and computation costs with PSIR variants [1] over a database with n items of size 288 B. In the case of ZipPIR, the stated client state size supports one query for that client.

n	Metric	SealPSIR	PaillierPSIR	XPSIR	ZipPIR
2^{16}	Client Storage	129 KB	129 KB	129 KB	6 KB
	Query	74 KB	12 KB	305 KB	591 KB
	Response	256 KB	18 KB	262 KB	3 KB
	Online Total	330 KB	30 KB	567 KB	594 KB
	Online Server	70 ms	760 ms	20 ms	16 ms
2^{18}	Client Storage	258 KB	258 KB	258 KB	12 KB
	Query	84 KB	24 KB	413 KB	655 KB
	Response	256 KB	18 KB	262 KB	6 KB
	Online Total	340 KB	42 KB	675 KB	661 KB
	Online Server	210 ms	1020 ms	120 ms	40 ms
2^{20}	Client Storage	524 KB	524 KB	524 KB	24 KB
	Query	104 KB	48 KB	597 KB	783 KB
	Response	256 KB	18 KB	262 KB	12 KB
	Online Total	360 KB	66 KB	895 KB	795 KB
	Online Server	710 ms	1750 ms	520 ms	120 ms

with sublinear online time impose client-side costs that are inconsistent with our model. We compare with the work of Patel et al. [59], which uses a client-side state. The authors instantiate PSIR using SealPIR, PaillierPIR, and XPIR in their paper. They do not provide the implementation, so we compare with the numbers presented in their paper [59, Table 2 & 3]. The result of this comparison is summarized as follows:

Online Costs. In almost all instances, ZipPIR is faster than all instantiations of PSIR. This can be attributed to the fact that PSIR performs PIR as a subroutine, which necessitates the use of public key operations, even though the number of such operations is sub-linear in the database size. In contrast, ZipPIR only performs plaintext operations in the online phase.

With regards to network costs, ZipPIR requires less online communication compared to XPSIR, which is the fastest variant of PSIR. The communication cost is comparable to that of SealPSIR and strictly worse than PaillierPSIR. However, these two protocols are much slower, as mentioned before, which shows a communication-computation tradeoff.

Offline Costs. In PSIR, the database is streamed to the client, and the state, which is sublinear in the database size, is stored. In ZipPIR with client-side storage, only the hint is sent to the client, which is much smaller than the database and the PSIR hint, but only supports one query.

6 Discussion

Why not only use Paillier? We construct ZipPIR as a combination of LWE-based PIR and Paillier. One might suggest using only Paillier for the full construction. Corrigan-Gibbs and Kogan [32, Appendix E.2] also suggested a solution that uses additive homomorphic encryption. However, such a protocol requires a large number of Paillier plaintext multiplications, linear in the database size. While this expensive step is performed in the offline phase, it is required to be done once for each query, which is a high computational cost, even in the offline phase. In contrast, the offline phase of ZipPIR only requires a sublinear number of Paillier operations, which is much more practical.

ZipPIR+DoublePIR. We can also construct a protocol, similar to ZipPIR, that builds on top of DoublePIR instead of SimplePIR. In this variant, the size of the hint would be reduced and the matrix multiplication between the hint and the compression key offset (Line 18) would be faster. This would, however, limit the size of the payload to only one byte, similar to DoublePIR and other protocols built on it, such as YPIR [55]. We leave a comparison of protocols focused on small payloads [22, 44, 55] for future work.

ZipPIR using a Packed Compression Key. ZipPIR can also be modified to use the packed compression key from Appendix A. Given that the bulk of the query cost consists only of the compression key, this can reduce the query size at the cost of a larger response.

Advantages for database updates. ZipPIR delegates all offline storage and computation to the server (except constant storage for keys and the PRNG seed). The benefit of this model is that database updates require no interaction with the client. This is compatible with our assumption that the client is resource-constrained. It is also efficient from a bandwidth standpoint since the server does not need to communicate with all available clients upon each database update, which is not scalable if many clients query the server. The disadvantage is that client-specific hints must be updated when the database changes, but this can be mitigated with additional server resources, without any client involvement.

7 Private SCT Auditing at Fixed Intervals

Henzinger et al. proposed the use of PIR in context of certificate transparency, specifically for auditing of signed certificate timestamps (SCT) [44]. The authors build on Google's recent solution of using an SCT auditor [35].

The initial solution of Henzinger et al. provided high throughput but came at the cost of a large hint stored by the clients. The problem is that this hint must be updated upon each change in the database. To combat this problem, the authors suggested that clients download hints periodically.

We propose a different approach to handle database updates, while maintaining throughput similar to that of SimplePIR. Typically, the set of clients performing SCT checks are known to the auditor (e.g. Google), so we assume that clients can register to perform tests and submit their public keys. Moreover, clients collect SCTs from TLS handshakes during browsing and perform an audit in a batch. So in our

proposed solution, we assume that clients audit a fixed-sized batch of SCTs at regular intervals. Using this assumption, the server can perform precomputation in anticipation of the client's request. The precomputation can be done on the latest version of the database, hence allowing to detect malicious behavior sooner.

8 Related Work

8.1 Related work on compression

Compression of homomorphic ciphertexts is achieved by performing scheme-switching, i.e., changing the scheme under which the message is encrypted.

Compression from Precise Scheme-switching. Techniques such as modulus switching [27] and dimension reduction [21] change the parameters of LWE and RLWE ciphertexts, resulting in smaller ciphertexts. LWE-to-RLWE packing [24] compresses a list of LWE ciphertexts into one RLWE ciphertext, which reduces the total size of the ciphertexts. However, this method is less effective if less than N ciphertexts are compressed, where N is the degree of the RLWE ciphertext. Coefficient extraction performs the inverse functionality of LWE-to-RLWE packing, i.e., extracting specific coefficients of the RLWE ciphertext and discarding the rest, which can reduce the overall size.

Compression from Imprecise Scheme-switching. Imprecise scheme-switching changes the scheme of the ciphertext, but encrypts to a message that is slightly different than the original message. Imprecise scheme-switching is sufficient if the final result is decrypted and no further computation is performed.

Hu [45] introduced the concept of *secure converters* for converting between cryptographic schemes. This is achieved by homomorphically evaluating (part of) the decryption circuit of the source scheme under the destination scheme. Within that framework, the author proposed homomorphically converting from LTV ciphertexts [49] to Paillier ciphertexts to reduce bandwidth usage from the server to the client. The conversion, however, is not precise and the Paillier ciphertexts encrypt a noisy version of the initial plaintexts. Using this approach, a 256x compression rate is achieved whilst communicating ciphertexts back to the client. However, the LTV cryptographic scheme is not adopted as a practical homomorphic encryption scheme.

Brakerski et al. [19] showed how a high-rate compression, arbitrarily close to one, can be achieved over ciphertexts with the *linear-decrypt-and-multiply* characteristic. Cryptosystems with linear-decrypt-and-multiply can decrypt to any multiple of the message. Based on the authors, among prevalent encryption schemes, only GSW falls into that category. Assuming the goal is to encrypt $\{m_0, m_1, \cdots, m_{\ell-1}\}$, then the compression is done by homomorphically decrypting these messages to $\{m_0 + e_0, \Delta m_1 + e_1, \cdots, \Delta^{\ell-1} m_{\ell-1} + e_{\ell-1}\}$, where e_i's are noise introduced from the homomorphic cryptosystem, similar to LWE. By adding these messages together, the server obtains one large plaintext, encrypted under an additive ciphertext which it sends to the client.

Gentry et al. [43] also proposed a method to compress many GSW ciphertexts into high-rate PVW [61] ciphertexts. The ratio between the plaintext and ciphertext can be arbitrarily close to one in their construction. However, this can only be achieved if the underlying aggregate plaintext is very large. Specifically, for the ratio to be $1-\varepsilon$, the aggregate plaintext must be proportional to $1/\varepsilon^3$. The authors described how to construct a PIR protocol from this technique, but their compression technique is not applicable to any other type of ciphertext.

8.2 Related Work on PIR

Computational PIR (CPIR) protocols follow one of two approaches: 1) the server (with or without the help of the client) computes a *hint* that is given to the client 2) the client sends cryptographic keys to the server which are stored. We describe each approach briefly, the advantages and disadvantages of each approach and list related work.

8.2.1 PIR with client-storage

One approach to PIR is to compute a database-dependent hint which is stored by the client before issuing queries. This hint can speed up subsequent queries. SimplePIR [44] and FrodoPIR [34] propose a PIR protocol based on LWE with a client-independent hint. The hint size is $O(\sqrt{N}n)$ for N database rows and LWE dimension of n. All clients use the same hint, which helps quickly respond to PIR queries and achieves high throughput (up to 10 GB/s). However, the hint is a high upfront network cost (100 MB for a 1 GB database), requires large storage by the client, and must be recalculated and redistributed to the clients every time the database is updated. DoublePIR extends SimplePIR such that the hint is smaller, but the overall throughput is less and it is limited to retrieval of one byte of information per query.

Another similar line of work proposes PIR with online time that is sublinear in the size of the database [31, 66]. The client stores some information in the form of a hint, which is used to issue queries and must be updated after a certain number of queries have been made. Despite the very high throughput of these protocols, the requirements are highly impractical for a resource-constrained client. In some protocols [66], the server streams the entire database to the client in the offline phase, and the storage requirements for the client are sublinear in the size of the database.

8.2.2 PIR with per-client server-side storage

Another category of works assumes auxiliary information is sent before the start of the protocol, usually in the form of

cryptographic keys. The cost of sending these keys is amortized over many queries, but requires per-client storage on the server. The per-query communication costs in such protocols are small, and if sufficient queries are made, the runtime and communication cost of setup are amortized. Works that follow this model include SealPIR [13], MulPIR [11], OnionPIR [56], Constant-weight PIR [51], Pantheon [5], FastPIR [6], Spiral (and its variants) [54], and SparsePIR [60].

HintlessPIR [47] and YPIR [55] remove the need for client-specific storage and generate one large preprocessed hint, which depends on the database. This hint, which is stored by the server, is used to respond to all client queries.

9 Conclusion

In this work, we proposed ZipPIR, a client-efficient single-server PIR protocol. ZipPIR is designed to minimize client-side storage requirements, while maintaining high throughput. After sending an initial setup, the offline phase can be performed silently without any interaction with the client or client-side computation or storage. By delegating all expensive offline computation and storage to the server, database updates can be handled silently, without communication with all clients. At the heart of our construction is a new technique for compressing LWE ciphertexts into extremely small additive ciphertexts. Our compression technique may be of independent interest and is applicable in many applications which send (R)LWE ciphertexts over the network, not only PIR. Our evaluation shows that our compression can achieve up to 99% size reduction for LWE ciphertexts. Moreover, our evaluation of our ZipPIR shows how we can achieve over 3 GB/s of throughput in the online phase, without imposing any client-side burden. Our works demonstrates that PIR using additive schemes such as Paillier, which were thought to be inefficient, can be practical and competitive with PIR schemes which are purely based on lattice-based schemes.

Ethical Considerations

In this work, the main goal is to improve the efficiency of Private Information Retrieval (PIR), a privacy-enhancing technology that enables users to retrieve data without revealing their specific items of interest. We have identified several key stakeholders and evaluated the potential impacts of our research on each:

Potential Stakeholders

- **End-Users of digital services.** The primary stakeholders are individuals seeking to use digital services. By making PIR easier to deploy in real-world applications (such as password leak checks or SCT auditing), this research empowers users to verify their security status without surrendering sensitive metadata to service providers. The benefit to these users is removing the trade-off between utility and privacy.
- **Service Providers.** Improving efficiency lowers the computational and financial barriers for organizations to adopt privacy-preserving protocols.
- **Potential Misusers (Negative Stakeholders).** We acknowledge that PIR may allow bad actors to attempt to access data while using the protocol's privacy properties to mask illegal activities.

Balancing Benefits and Risks Our decision to conduct and publish this research was motivated by the significant barriers currently preventing the widespread adoption of PIR. We weighed the potential for misuse against the tangible benefits of empowering users to remain safe on the internet while maintaining their privacy. Because PIR only protects the query and not the authorization (i.e., it does not bypass access control mechanisms already in place by the data provider), we believe the risk of masking illegal activity is mitigated by existing security layers.

Ultimately, we believe the potential gains—evidenced by current real-world deployments such as Password Checkup—outweigh the potential negative outcomes. We are unaware of any further ethical considerations or overlooked stakeholders regarding the improved efficiency of PIR.

Open Science

We open-source our implementation of our proposed protocols and the scripts used to perform our experimental evaluation such that it may be evaluated for both functionality and reproducibility. [2]

References

[1] Private Stateful Information Retrieval | Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. https://dl.acm.org/doi/10.1145/3243734.3243821.

[2] Microsoft SEAL (release 4.1), January 2023.

[3] Lattigo v6. Online: https://github.com/tuneinsight/lattigo, August 2024. EPFL-LDS, Tune Insight SA.

[4] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private Information Retrieval for Everyone. *Proceedings on Privacy Enhancing Technologies*, 2016.

[2] https://zenodo.org/records/17907225

[5] Ishtiyaque Ahmad, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Pantheon: Private Retrieval from Public Key-Value Store. *Proceedings of the VLDB Endowment*, 16(4):643–656, December 2022.

[6] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, 2021.

[7] Rasoul Akhavan Mahdavi, Haoyan Ni, Dimitry Linkov, and Florian Kerschbaum. Level Up: Private Non-Interactive Decision Tree Evaluation using Levelled Homomorphic Encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, pages 2945–2958, New York, NY, USA, November 2023. Association for Computing Machinery.

[8] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-Source Fully Homomorphic Encryption Library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC'22, pages 53–63, New York, NY, USA, November 2022. Association for Computing Machinery.

[9] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors, 2015.

[10] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, volume 9056, pages 430–454. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[11] Asra Ali, Tancrède Lepoint, S Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-Computation Trade-offs in PIR. *IACR Cryptol. ePrint Arch.*, 2019:1483, 2019.

[12] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication–Computation Trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1811–1828, 2021.

[13] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, San Francisco, CA, May 2018. IEEE.

[14] Sebastian Angel and Srinath Setty. Unobservable Communication over Fully Untrusted Infrastructure. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 551–569, Savannah, GA, November 2016. {USENIX} Association.

[15] Elaine Barker. Recommendation for key management: Part 1 - general. Technical Report NIST SP 800-57pt1r5, National Institute of Standards and Technology, Gaithersburg, MD, May 2020.

[16] Martin Beck. Randomized decryption (RD) mode of operation for homomorphic cryptography - increasing encryption, communication and storage efficiency. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 220–226, April 2015.

[17] Josh Benaloh. Dense Probabilistic Encryption. In *Proceedings of the Workshop on Selected Areas of Cryptography*, pages 120–128, 1994.

[18] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, Lecture Notes in Computer Science, pages 868–886, Berlin, Heidelberg, 2012. Springer.

[19] Zvika Brakerski, Nico Döttling, Sanjam Garg, and Giulio Malavolta. Leveraging Linear Decryption: Rate-1 Fully-Homomorphic Encryption and Time-Lock Puzzles. In Dennis Hofheinz and Alon Rosen, editors, *Theory of Cryptography*, Lecture Notes in Computer Science, pages 407–437, Cham, 2019. Springer International Publishing.

[20] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, January 2012. Association for Computing Machinery.

[21] Zvika Brakerski and Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, Palm Springs, CA, USA, October 2011. IEEE.

[22] Alexander Burton, Samir Jordan Menon, and David J. Wu. Respire: High-rate pir for databases with small records. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*,

CCS '24, page 1463–1477, New York, NY, USA, 2024. Association for Computing Machinery.

[23] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression. *Journal of Cryptology*, 31(3):885–916, July 2018.

[24] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. In Kazue Sako and Nils Ole Tippenhauer, editors, *Applied Cryptography and Network Security*, volume 12726, pages 460–479. Springer International Publishing, Cham, 2021.

[25] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from Fully Homomorphic Encryption with Malicious Security.

[26] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from Fully Homomorphic Encryption with Malicious Security. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1223–1237, Toronto Canada, October 2018. ACM.

[27] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, Lecture Notes in Computer Science, pages 409–437, Cham, 2017. Springer International Publishing.

[28] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology*, 33(1):34–91, January 2020.

[29] Jihoon Cho, Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. Transciphering Framework for Approximate Homomorphic Encryption. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, Lecture Notes in Computer Science, pages 640–669, Cham, 2021. Springer International Publishing.

[30] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[31] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-Server Private Information Retrieval with Sublinear Amortized Time. In *Advances in Cryptology – EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 – June 3, 2022, Proceedings, Part II*, pages 3–33, Berlin, Heidelberg, May 2022. Springer-Verlag.

[32] Henry Corrigan-Gibbs and Dmitry Kogan. Private Information Retrieval with Sublinear Online Time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 44–75, Cham, 2020. Springer International Publishing.

[33] Ivan Damgård and Mads Jurik. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In Kwangjo Kim, editor, *Public Key Cryptography*, Lecture Notes in Computer Science, pages 119–136, Berlin, Heidelberg, 2001. Springer.

[34] Alex Davidson, Gonçalo Pestana, and Sofía Celi. FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval. *Proceedings on Privacy Enhancing Technologies*, 2023(1):365–383, January 2023.

[35] Joe DeBlasio. Opt-out sct auditing in chrome.

[36] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018*, volume 10991, pages 662–692. Springer International Publishing, Cham, 2018.

[37] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Pasta: A Case for Hybrid Homomorphic Encryption, 2021.

[38] Léo Ducas and Daniele Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, Lecture Notes in Computer Science, pages 617–640, Berlin, Heidelberg, 2015. Springer.

[39] Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In George Robert Blakley and David Chaum, editors, *Advances in Cryptology*, Lecture Notes in Computer Science, pages 10–18, Berlin, Heidelberg, 1985. Springer.

[40] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. *Proceedings of the 15th international conference on Practice and Theory in Public Key Cryptography*, 2012:1–16, 2012.

[41] Ben Fisch, Arthur Lazzaretti, Zeyu Liu, and Charalampos Papamanthou. Thorpir: Single server pir via homomorphic thorp shuffles. In *Proceedings of the 2024 on*

ACM SIGSAC Conference on Computer and Communications Security, CCS '24, page 1448–1462, New York, NY, USA, 2024. Association for Computing Machinery.

[42] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford University, Stanford, CA, USA, 2009.

[43] Craig Gentry and Shai Halevi. Compressible FHE with Applications to PIR. In Dennis Hofheinz and Alon Rosen, editors, *Theory of Cryptography*, Lecture Notes in Computer Science, pages 438–464, Cham, 2019. Springer International Publishing.

[44] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One Server for the Price of Two: Simple and Fast {Single-Server} Private Information Retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3889–3905, 2023.

[45] Yin Hu. *Improving the Efficiency of Homomorphic Encryption Schemes*. PhD thesis, Worcester Polytechnic Institute, 2013.

[46] Dmitry Kogan and Henry Corrigan-Gibbs. Private Blocklist Lookups with Checklist.

[47] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless Single-Server Private Information Retrieval, 2023.

[48] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for Checking Compromised Credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1387–1403, New York, NY, USA, November 2019. Association for Computing Machinery.

[49] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the Forty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '12, page 1219–1234, New York, NY, USA, 2012. Association for Computing Machinery.

[50] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, Lecture Notes in Computer Science, pages 1–23, Berlin, Heidelberg, 2010. Springer.

[51] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1723–1740, 2022.

[52] Rasoul Akhavan Mahdavi, Nils Lukas, Faezeh Ebrahimianghazani, Thomas Humphries, Bailey Kacsmar, John Premkumar, Xinda Li, Simon Oya, Ehsan Amjadian, and Florian Kerschbaum. PEPSI: Practically Efficient Private Set Intersection in the Unbalanced Setting, October 2023.

[53] Miti Mazmudar, Shannon Veitch, and Rasoul Akhavan Mahdavi. Peer2PIR: Private Queries for IPFS . In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 4438–4456, Los Alamitos, CA, USA, May 2025. IEEE Computer Society.

[54] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947, San Francisco, CA, USA, May 2022. IEEE.

[55] Samir Jordan Menon and David J. Wu. YPIR: High-Throughput Single-Server PIR with Silent Preprocessing, 2024.

[56] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response Efficient Single-Server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 2292–2306, New York, NY, USA, November 2021. Association for Computing Machinery.

[57] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, October 2011. Association for Computing Machinery.

[58] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, Lecture Notes in Computer Science, pages 223–238, Berlin, Heidelberg, 1999. Springer.

[59] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private Stateful Infor-mation Retrieval. *ACM SIGSAC Conference on Computer & Communications Security*, 18:18, 2018.

[60] Sarvar Patel, Joon Young Seo, and Kevin Yeo. {Don't} be Dense: Efficient Keyword {PIR} for Sparse Databases. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3853–3870, 2023.

[61] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. A framework for efficient and composable oblivious transfer. In *Proceedings of the 28th Annual Conference on Cryptology: Advances in Cryptology*, CRYPTO

2008, page 554–571, Berlin, Heidelberg, 2008. Springer-Verlag.

[62] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. *Journal of the ACM*, 56(6):1–40, September 2009.

[63] Ling Ren, Muhammad Haris Mughees, and I Sun. Simple and practical amortized sublinear private information retrieval using dummy subsets. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, page 1420–1433, New York, NY, USA, 2024. Association for Computing Machinery.

[64] Kurt Thomas, Jennifer Pullman, Kevin Yeo, A Raghunathan, Patrick Gage Kelley, L Invernizzi, B Benko, Tadek Pietraszek, S Patel, D Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security Symposium*, 2019.

[65] Zama. Concrete: TFHE Compiler that converts python programs into FHE equivalent, 2022.

[66] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. Piano: Extremely simple, single-server pir with sublinear server computation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4296–4314, 2024.

A LWECompress Using Packed Compression Keys

The necessary procedures for using a packed compression key is given in Algorithm 6. The GENERATEPACKEDKEY procedure generates the packed key from the LWE secret key. The unpacking procedure computes the compression key from the packed compression key by scaling the packed key with different values. By packing in this particular manner, the compression procedure can be done similar to before, without any changes. The final change is made in the decryption. The response is not necessarily in the lower order bits of the additive ciphertext ciphertext anymore, so a division is required before continuing with the rest of the LWE decryption procedure.

B Compressig RLWE Coefficients

Similar to LWE, we can also construct an encryption scheme based on the Ring Learning with Errors (RLWE) [50] assumption, which we will denote as $\mathcal{E}_{\text{RLWE}}$.

Cryptosystems such as BGV [20], BFV [18, 40], and CKKS [27] have ciphertexts of a similar format. RLWE ciphertexts are useful since they can encrypt a polynomial, i.e. a vector of numbers, instead of just one scalar. For RLWE encryption, we select a dimension N, ciphertext modulus q, plaintext modulus p, and $\Delta = \lfloor q/p \rceil$. Define $R_q = \mathbb{Z}_q[X]/(X^N+1)$ and R_p similarly. Moreover, define a discrete error distribution χ over R_q. For key generation, sample $S(X)$ uniformly from R_q.

Similar to LWE, we can also compress fresh RLWE ciphertexts by sending the seed used to generate $A(X)$ [2, 8, 12]. Using this technique, the size of a ciphertext can be reduced from $2N\log_2 q$ bits to $\lambda + N\log_2 q$.

RLWE ciphertexts also have a linear phase evaluation and hence, can benefit from our compression technique. However, an RLWE ciphertext is only twice as large as the phase so the compression technique, applied naively, would not yield a significant improvement. Our approach is beneficial if the user is only interested in some coefficients of the RLWE plaintext and not all of them.

The main observation is that each coefficient of $\mu'(X)$ in the RLWEDECRYPT procedure can be calculated separately. Specifically, for $0 \le k \le N-1$

Algorithm 6 Procedures for using a packed compression key, including generating the packed key, unpacking it, and the corresponding modified decryption function.

procedure GENERATEPACKEDKEY($\mathsf{sk}_A, \mathsf{sk}$)
  $t = \left\lfloor \frac{0.5\log_2 m}{\log_2 \delta} \right\rfloor$
  for $i \in [\lceil n/t \rceil]$ **do**
    $r \leftarrow \delta^{-(t-1)}(\sum_{j\in[t]} \mathsf{sk}[it+j]\cdot\delta^j) \mod m$
    $\mathsf{pck}_i \leftarrow \mathtt{AEnc}(\mathsf{sk}_A, r)$
  return pck

procedure UNPACKCOMPRESSIONKEY$_q$(pck)
  for $i \in [\lceil n/t \rceil]$ **do**
    for $j \in [t]$ **do**
      $\mathsf{ck}[it+j] \leftarrow \delta^{t-1-j} \otimes \mathsf{pck}[i]$
  return ck

procedure MODIFIEDLWEDECRYPTPACKEDKEY$_{q,p}$(sk_A, x)
  $y \leftarrow \delta^{(t-1)}\mathtt{ADec}(\mathsf{sk}_A, x) \mod m$
  $\mu^{**} = \lfloor y/\delta^{(t-1)} \rceil \mod q$
  $\mu'' = \lfloor \mu^{**}/\Delta \rceil$ ▷ $\Delta = \lfloor q/p \rceil$
  return $\mu'' \in \mathbb{Z}_p$

Algorithm 7 Encryption and Decryption of $\mathcal{E}_{RLWE}$

procedure RLWEENCRYPT($S(X), \mu(X)$)
  Sample $A(X) \xleftarrow{\$} R_q$ and $E(X) \leftarrow \chi$
  $B(X) = A(X)\cdot S(X) + \Delta\cdot\mu(X) + E(X) \mod R_q$
  return $C = (A(X), B(X))$

procedure RLWEDECRYPT($S(X), C$)
  $(A(X), B(X)) \leftarrow C$
  $\mu^*(X) = (B(X) - A(X)\cdot S(X)) \mod R_q$
  $\mu'(X) = \lfloor \mu^*(X)/\Delta \rceil$
  return $\mu'(X) \in R_p$

$$\mu'[k] = \lfloor \frac{\mu^*[k]}{\Delta} \rceil \tag{7}$$

$$= \left\lfloor \frac{B[k] - \sum_{i=0}^{k} A[k-i] \cdot S[i] + \sum_{i=k+1}^{N-1} A[N+k-i] \cdot S[i]}{\Delta} \right\rceil \tag{8}$$

Note that the operations in the numerator are happening modulo q. The numerator of Equation (8) is a linear combination of the coefficients of the secret key, hence it can be computed given the encrypted coefficients of the secret key. The complete procedure to compress the coefficient of X^k in an RLWE plaintext and the corresponding decryption function is shown in Algorithm 8. Compression of RLWE coefficients is fully compatible with the compact compression keys of Section 3.2 and batched compression of Section 3.3.

Algorithm 8 Compressing Extracted RLWE Coefficient, performed by the server and the corresponding modified decryption process, for the client. The compression key is ck such that $\mathsf{ck}[i] = \mathtt{AEnc}_s(S[i])$ and $C \in R_q \times R_q$

1: **procedure** RLWECOMPRESSCOEFFICIENT(ck, C, k)
2: $x = B[k]$
3: **for** $i \in \{0, 1, \cdots, k\}$ **do**
4: $x \leftarrow x \oplus ((q - A[k-i]) \otimes \mathsf{ck}[i])$
5: **for** $i \in \{k+1, \cdots, N-1\}$ **do**
6: $x \leftarrow x \oplus (A[N+k-i] \otimes \mathsf{ck}[i])$
7: **return** x

8: **procedure** MODIFIEDRLWEDECRYPT$_{q,p}(\mathsf{sk}_A, x)$
9: $\mu_k^{**} = \mathtt{ADec}(\mathsf{sk}_A, x) \mod q$
10: $\mu_k' = \lfloor \mu_k^{**} / \Delta \rceil$ ▷ $\Delta = \lfloor q/p \rceil$
11: **return** $\mu_k' \in \mathbb{Z}_p$

Theorem 5 (Correctness). *If $m > q + Nq^2$, Algorithm 8 produces a compressed ciphertext which decrypts to the coefficient of X^k if decrypted using* MODIFIEDRLWEDECRYPT$_{q,p}$. *More formally,*

$$x \leftarrow \text{RLWECOMPRESSCOEFFICIENT}(\mathsf{ck}, c, k)$$
$$\mu_k'' \leftarrow \text{MODIFIEDRLWEDECRYPT}_{q,p}(s, x)$$

then μ_k'' is equal to the coefficient of X^k in

$$\mu'(X) = \text{RLWEDECRYPT}(\mathsf{sk}, c)$$

We provide the proof of Theorem 5 in Appendix C. Similar to the case of LWE ciphertexts, if the coefficients of the RLWE secret key are binary, we can tighten the condition on m in Theorem 5 such that $m > q + Nq$. The following corollary summarizes this fact.

Corollary 3. *If the coefficients of the secret key are binary and $m > q + Nq$, Algorithm 8 produces a compressed ciphertext which decrypts to the coefficient of X^k if decrypted using* MODIFIEDRLWEDECRYPT$_{q,p}$.

Security. A similar argument can be made about the security of compression over RLWE. Let $\mathcal{E}''$ denote the cryptosystem which is the combination of $\mathcal{E}_{RLWE}$ and $\mathcal{E}_A$. The following proposition holds regarding security.

Proposition 2 (Security). *If $\mathcal{E}_{RLWE}$ and $\mathcal{E}_A$ are semantically secure, then $\mathcal{E}''$ is also semantically secure.*

C Proof of RLWE Compression

Proof. Line 1 of Algorithm 8 computes

$$B[k] + \sum_{i=0}^{k} (q - A[k-i]) \cdot S[i] + \sum_{i=k+1}^{N-1} A[N+k-i] \cdot S[i]$$

encrypted under additive encryption, which is possible due to the linear properties. We know that all coefficients of $A(X)$, $B(X)$, and $S(X)$ are elements in $\mathbb{Z}_q$, hence

$$B[k] + \left(\sum_{i=0}^{k} (q - A[k-i]) \cdot S[i] \right) + \left(\sum_{i=k+1}^{N-1} A[N+k-i] \cdot S[i] \right)$$
$$\leq q + \left(\sum_{i=0}^{k} q \cdot q \right) + \left(\sum_{i=k+1}^{N-1} q \cdot q \right) = q + Nq^2 < m$$

so there is no overflow in the plaintext space of the additive cryptosystem.

$$\begin{aligned}
\mu_k^{**} &= \mathtt{ADec}_s(x) \mod q \\
&= \left(\left(B[k] + \sum_{i=0}^{k} (q - A[k-i]) \cdot S[i] \right.\right. \\
&\qquad \left.\left. + \sum_{i=k+1}^{N-1} A[N+k-i] \cdot S[i] \right) \mod m \right) \mod q \\
&= \left(B[k] + \sum_{i=0}^{k} (q - A[k-i]) \cdot S[i] + \sum_{i=k+1}^{N-1} A[N+k-i] \cdot S[i] \right) \mod q \\
&= B[k] - \sum_{i=0}^{k} A[k-i] \cdot S[i] + \sum_{i=k+1}^{N-1} A[N+k-i] \cdot S[i] \mod q
\end{aligned}$$

which is equivalent to the k^{th} coefficient of

$$\mu^*(X) = B(X) - A(X) \cdot S(X) \mod R_q$$

which can be seen by expanding the equation. Given that line 16 of Algorithm 1 performs rounding coefficient-wise, it produces the same result as line 10 of Algorithm 8. □

D Security & Correctness of ZipPIR

In this section, we prove the correctness and security of ZipPIR.

Theorem 3. *(ZipPIR Correctness) For LWE parameters (n, q, χ_e, χ_s) where χ_s is a discrete Gaussian with standard deviation σ, and χ_s is a binary distribution, and plaintext modulus p, and failure rate δ such that*

$$q/p > 2p\sigma\sqrt{2d_0 \ln(2/\delta)} \tag{6}$$

for random $\boldsymbol{A} \in \mathbb{Z}_q^{d_0 \times n}$*, for any database* $\mathsf{db} \in \mathbb{Z}_p^{d_0 \times d_1}$*,* $\boldsymbol{H} = -\mathsf{db}^T \cdot \boldsymbol{A}$*, Paillier modulus m such that* $m > q + nq$*, and any query* $i_0 \in [d_0]$*, if*

$$\begin{aligned}((\mathsf{sk}_P, \mathsf{pt}_r), (m, \mathsf{ck}_r)) &\leftarrow \textsc{Setup}(1^\lambda, (d_0, d_1)) \\ \boldsymbol{k} &\leftarrow \textsc{Hint}(\boldsymbol{H}, (m, \mathsf{ck}_r)) \\ (\mathsf{ck}_o, \mathsf{qu}_0) &\leftarrow \textsc{Query}(i_0, \mathsf{pt}_r) \\ t &\leftarrow \textsc{Response}(\mathsf{db}, \boldsymbol{H}, (\mathsf{ck}_o, \mathsf{qu}_0)) \\ f &\leftarrow \textsc{Extract}(\mathsf{sk}_P, (\boldsymbol{k}, t))\end{aligned}$$

then $\mathbb{P}[\mathsf{db}[i_0] = f] > 1 - \delta - 2d_0/\sqrt{m}$*.*

Proof. We know that ck_r denotes a valid ciphertext, if and only if $\mathsf{ck}_r \in (\mathbb{Z}_{m^2}^*)^{d_0}$. This happens with probability, $(\phi(m^2)/m^2)^{d_0} > 1 - 2d_0/\sqrt{m}$, which is high for secure choices of m. Assuming that $\mathsf{ck}_r \in (\mathbb{Z}_{m^2}^*)^{d_0}$, then assume $r = \textsc{PaillierDecrypt}(\mathsf{sk}_P, \mathsf{ck}_r)$ so

$$\mu = t + \textsc{PaillierDecrypt}(\mathsf{sk}_P, \mathbf{k}) \mod m \tag{9}$$

$$= b + \mathbf{H} \cdot \mathsf{ck}_o + \mathbf{H} \cdot \textsc{PaillierDecrypt}(\mathsf{sk}_P, \mathsf{ck}_r) \mod m \tag{10}$$

$$= b + \mathbf{H} \cdot (\mathsf{ck}_o + r) = b + \mathbf{H} \cdot \mathsf{sk} \mod m = b + \mathbf{H} \cdot \mathsf{sk} \tag{11}$$

Where the last equation comes from the fact that $||b + \mathbf{H} \cdot \mathsf{sk}||_\infty < q + nq < m$, so it does overflow mod m. Hence,

$$\mu \mod q = b + \mathbf{H} \cdot \mathsf{sk} \mod q \tag{12}$$

$$= \mathsf{db}^T \cdot \mathsf{qu}_0 - \mathsf{db}^T \cdot A \cdot \mathsf{sk} \mod q \tag{13}$$

$$= \mathsf{db}^T (\Delta u_0 + e) = \Delta \mathsf{db}[i] + \mathsf{db}^T e \tag{14}$$

Using the same analysis from the proof of SimplePIR [44, Appendix C.2], we can show that $||\mathsf{db}^T e||_\infty < \Delta/2$ with probability $1 - \delta$, assuming Equation (6) holds. Combining this step with the previous step, we can see that with probability $(1 - \delta)(1 - 2d_0/\sqrt{m}) > 1 - \delta - 2d_0/\sqrt{m}$, we will have $f = \lfloor (\mu \mod q) p/q \rceil = \mathsf{db}[i]$, which proves the theorem. □

Theorem 4. *(ZipPIR Security) Assume we have secure LWE parameters* (n, q, χ_e, χ_s) *where* χ_s *is a discrete Gaussian with standard deviation* σ*, and* χ_s *is a binary distribution. Also, assume we have secure Paillier parameters. Then for any PPT adversary* $\mathcal{A}$ *and any* $i, j \in [d_0]$*, we have*

$$\left| \mathbb{P}\left[\mathcal{A}(1^\lambda, (q_h, \mathsf{qu})) = 1 \;\middle|\; \begin{array}{c} ((\_, \mathsf{pt}_r), q_h) \leftarrow \textsc{Setup}(1^\lambda, (d_0, d_1)) \\ \mathsf{qu} \leftarrow \textsc{Query}(i, \mathsf{pt}_r) \end{array} \right] \right.$$
$$\left. - \mathbb{P}\left[\mathcal{A}(1^\lambda, (q_h, \mathsf{qu})) = 1 \;\middle|\; \begin{array}{c} ((\_, \mathsf{pt}_r), q_h) \leftarrow \textsc{Setup}(1^\lambda, (d_0, d_1)) \\ \mathsf{qu} \leftarrow \textsc{Query}(j, \mathsf{pt}_r) \end{array} \right] \right| < \varepsilon$$

where ε *is negligible in the security parameter.*

Proof. We denote $(q_h, \mathsf{qu}) = (\mathsf{pk}_P, \mathsf{ck}_r, \mathsf{ck}_o, \mathsf{qu}_0)$ for simplicity. To prove the theorem, we prove that the tuple, $(\mathsf{pk}_P, \mathsf{ck}_r, \mathsf{ck}_o, \mathsf{qu}_0)$, as generate by ZipPIR is indistinguishable from a random tuple. We will prove this via multiple hybrid arguments. For this, we define 5 distributions to generate the tuple and show that each consecutive pair of distributions are indistinguishable. We define each distribution by the modification to the original definition of the tuple.

Dist. $\mathcal{H}_0$**:** The tuple is generated as described in ZipPIR
Dist. $\mathcal{H}_1$**:** Sample $\mathsf{pt}_r \leftarrow\!\!\$\, \mathbb{Z}_m$ and set $\mathsf{ck}_r = \mathsf{Enc}(\mathsf{sk}_P, \mathsf{pt}_r)$
Dist. $\mathcal{H}_2$**:** Set $\mathsf{ck}_r = \mathsf{Enc}(\mathsf{sk}_P, \mathsf{sk})$ and sample ck_o randomly
Dist. $\mathcal{H}_3$**:** Sample ck_r and ck_o randomly
Dist. $\mathcal{H}_4$**:** Sample qu_0, ck_r, and ck_o

We show that each consecutive pair of distributions are indistinguishable up to a negligible factor.

The only difference between $\mathcal{H}_0$ and $\mathcal{H}_1$ is the probability of producing invalid ciphertexts in $\mathcal{H}_0$, which is $2d_0/\sqrt{m}$ and is negligible in the security parameter for secure choice of m.

$\mathcal{H}_1$ and $\mathcal{H}_2$ can be simulated by a simulator which S which given input from $\mathcal{H}_2$ computes $S(\mathsf{pt}_r, \mathsf{ck}_r, \mathsf{ck}_o) = (\mathsf{pt}_r, \mathsf{pt}_r - \mathsf{ck}_r, \mathsf{ck}_o + \mathsf{pt}_r)$. The output of S is identical to the output of $\mathcal{H}_1$.

Indistinguishability between $\mathcal{H}_2$ and $\mathcal{H}_3$ reduces to the IND-CPA security of the Paillier encryption.

Lastly, indistinguishability between $\mathcal{H}_3$ and $\mathcal{H}_4$ is identical to that of SimplePIR [44, Lemma C.2].

Combining these steps proves that the correctly generated tuple is indistinguishable from random, up to a negligible parameter.

□