Improved Constant-weight PIR with an Extension for Multi-query

Jian Liu*
Zhejiang University
liujian2411@zju.edu.cn

Jingyu Li*
Zhejiang University
jingyuli@zju.edu.cn

Di Wu
Zhejiang University
wu.di@zju.edu.cn

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

Abstract

Homomorphic equality operator is essential for many secure computation tasks such as private information retrieval (PIR). However, the folklore homomorphic equality operator is typically considered to be impractical as its multiplicative depth depends on the input bit-length. In Usenix SEC '22, Mahdavi-Kerschbaum propose a homomorphic equality operator with a constant multiplicative depth, based on constant-weight code. On that basis, they propose constant-weight PIR (CwPIR for short); compared with other PIR protocols, CwPIR is more friendly to databases with large payloads and can support keyword query almost for free. Unfortunately, CwPIR cannot support databases with a large number of elements, which limits its real-world impact.

In this paper, we propose a homomorphic constant-weight equality operator that supports batch processing, hence it can perform thousands of equality checks with a much smaller amortized cost. Based on this improved homomorphic equality operator, we propose a novel PIR protocol named PIRANA, which inherits all advantages of CwPIR with a significant improvement in supporting more elements. We further extend PIRANA to support multi-query. To the best of our knowledge, PIRANA is the *first* multi-query PIR that can save both computation and communication. Our experimental results show that our single-query PIRANA is upto 30.8× faster than CwPIR; our multi-query PIRANA saves upto 163.9× communication over the state-of-the-art multi-query PIR (with a similar computational cost).

1 Introduction

Suppose a server S holds a database of n elements, where each element consists of an identifier (index or keyword) and a payload; *private information retrieval* (PIR) allows a client C to retrieve an element from the database without revealing which element was retrieved. It enables a wide range of privacy-preserving applications such as private contact discovery [17], private contact tracing [36], private navigation [37], anonymous messaging [25, 30], and safe browsing [22]. Despite being extensively studied for decades, PIR is still a hot research topic.

The PIR protocols can be roughly categorized into multi-server PIR [13] and single-server PIR [24]. The multi-server protocols are much more efficient in both computation and communication, and can achieve information-theoretic security. However, their reliance of multiple non-colluding servers is an unrealistic assumption in practice. In contrast, the single-server protocols do not have this strong assumption, but introduce a huge performance overhead. It has been shown that such protocols are even slower than trivially having C download the entire database [35]. Furthermore, most of existing PIR protocols only support index queries, and they require extra communication rounds to reduce keyword PIR to index PIR [12]. On the other hand, keyword queries are more commonly used in real-world applications.

Constant-weight PIR. Recently, Mahdavi-Kerschbaum [27] propose a PIR based on *constant-weight code*, the codewords of which have the same Hamming weight. They refer to this PIR as *constant-weight PIR* (CwPIR for short). They design *homomorphic equality operators* for constant-weight codewords with a multiplicative depth that only depends on the Hamming weight k of the code, not the bit-length m of the codewords. CwPIR works as follows: (i) C/S maps its query/identifiers to constant-weight codeword(s); (ii) C homomorphically encrypts the indices that correspond to 1s in its query codeword, and sends the ciphertexts to S; (iii) S obliviously *expands* them into m ciphertexts, which correspond to the m bits of the query codeword; (iv) for each of the n identifiers, S runs the homomorphic equality operator between the identifier codeword and the query codeword, leading to a selection vector of length n; (v) S returns the inner product between the selection vector and the payloads. Compared with prior arts, CwPIR is more friendly to databases with large payloads, and can support keyword query with minor modification, no extra rounds, and minimal overhead.

*Jian Liu and Jingyu Li are co-first authors.

However, CwPIR needs to run ciphertext-ciphertext multiplication for $(k-1)n$ times, hence it cannot support databases with a large number of elements. Their experimental results also confirm this (cf. Table 13 in [27]).

Our contribution. In this paper, we propose a *novel* PIR protocol named PIRANA, which inherits all advantages of CwPIR [27] with a significant improvement in supporting more elements. Our starting point is to replace the homomorphic equality operator in CwPIR with a *novel* SIMD-based one, which supports batch processing: it can perform N (the number of slots in an SIMD ciphertext) equality checks with a single ciphertext-ciphertext multiplication (instead of N). As a result, S in PIRANA only needs to run ciphertext-ciphertext multiplication for $(k-1)\lceil n/N \rceil$ times (instead of $(k-1)n$). We further extend PIRANA to support multi-query by replacing the constant-weight code with a bloom filter. Prior multi-query PIR protocols trade communication for computation: they introduce more communication overhead than processing multiple queries separately. To the best of our knowledge, PIRANA is the *first* multi-query PIR that can save both computation and communication. Its basic idea is to batch multiple queries using bloom filter and batch multiple responses using *oblivious rotation*. In particular, the oblivious rotation scheme, initially proposed by us, allows one to rotate an element from an unknown slot of an SIMD ciphertext to a designated slot. It could be of independent interest. We fully implement PIRANA and systematically evaluate its performance.

We summarize our contribution as follows:

- We propose a *novel* homomorphic equality operator that supports batch processing for constant-weight codewords. (Section 3)
- We design a *novel* constant-weight PIR named PIRANA, which is upto 30.8× faster than CwPIR proposed by Mahdavi-Kerschbaum [27]. (Section 4)
- We propose the *first* muti-query PIR that can save both computation and communication. It saves upto 163.9× communication over the state-of-the-art multi-query PIR [2] (with a similar computational cost). (Section 5)
- We provide a full-fledged implementation and extensive evaluation. (Section 6)

2 Preliminaries

In this section, we describe some building blocks for our protocols.

2.1 Homomorphic encryption

Fully Homomorphic Encryption (FHE) is an encryption scheme that allows arbitrary functions to be performed over encrypted data [19]. In practice, it is usually used in a leveled fashion, i.e., only a predefined number of operations can be performed. In most FHE cryptosystems [5–7, 10, 18], plaintexts are encoded as polynomials from the quotient ring $\mathbb{Z}_p[x]/(x^N+1)$, where N is a power of 2, and p is the plaintext modulus. The plaintext polynomials are then encrypted into ciphertext polynomials $\mathbb{Z}_q[x]/(x^N+1)$, where q is the ciphertext modulus that determines the security level and the number of operations that can be performed.

Notation	Description
C	client
S	server
n	DB size
id	an identifier
pl	a payload
$CW(m,k)$	binary constant-weight code with length m and Hamming weight k
N	polynomial modulus degree in FHE number of slots in a SIMD ciphertext
p	plaintext modulus, SIMD slot size
q	ciphertext modulus
$\mathbf{x}$	a bit array
$\widetilde{\mathbf{x}}$	an SIMD ciphertext
t	$\lceil m/N \rceil$
s	$\lceil n/N \rceil$
%	modulo
L	number of queries
$BC(n,M,L,B)$	a batch code that encodes n elements into M codewords in B buckets, supporting L queries
C	a batch code codeword
$BL(m,k)$	a bloom filter of length m with k hash functions
H	hash function

Table 1: Summary of frequent notations.

Most FHE cryptosystems support single instruction multiple data (SIMD), which encrypts multiple elements into a single ciphertext and processes these encrypted elements in a batch without introducing any extra cost. An SIMD ciphertext has N slots, hence it can be viewed as an encryption of a length-N vector over $\mathbb{Z}_p$. Notice that the SIMD technique also supports cyclic rotations of its slots.

The homomorphic operations used in this paper are summarized as follows:

- $\widetilde{\mathbf{x}} \leftarrow \mathsf{SIMDEnc}(\mathsf{pk}, \mathbf{x})$. The encryption algorithm takes a plaintext vector $\mathbf{x} = [x_1, ..., x_N]$ and outputs an SIMD ciphertext denoted by $\widetilde{\mathbf{x}}$.

- $\mathbf{x} \leftarrow \mathsf{SIMDDec}(\mathsf{pk}, \widetilde{\mathbf{x}})$. The decryption algorithm takes an SIMD ciphertext $\widetilde{\mathbf{x}}$ and outputs a plaintext vector $\mathbf{x}$.
- $\widetilde{\mathbf{z}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{x}}, \widetilde{\mathbf{y}})$. The addition algorithm takes two SIMD ciphertexts $\widetilde{\mathbf{x}}$ and $\widetilde{\mathbf{y}}$; outputs an encryption of $[x_1 + y_1, ..., x_N + y_N]$.
- $\widetilde{\mathbf{z}} \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{x}}, \mathbf{y})$. The ciphertext-plaintext multiplication takes a SIMD ciphertext $\widetilde{\mathbf{x}}$ and a plaintext vector $\mathbf{y}$; outputs an encryption of $[x_1 y_1, ..., x_N y_N]$.
- $\widetilde{\mathbf{z}} \leftarrow \mathsf{SIMDMul}(\widetilde{\mathbf{x}}, \widetilde{\mathbf{y}})$. The ciphertext-ciphertext multiplication takes two SIMD ciphertexts $\widetilde{\mathbf{x}}$ and $\widetilde{\mathbf{y}}$; outputs an encryption of $[x_1 y_1, ..., x_N y_N]$.
- $\widetilde{\mathbf{x}}' \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{x}}, c)$. The rotation algorithm takes an SIMD ciphertext $\widetilde{\mathbf{x}}$ and an integer $c \in [N]$; outputs an encryption of $\left[x_{(1+c)\%N}, ..., x_{(N+c)\%N}\right]$.

2.2 Constant-weight code

Constant-weight code is a form of error detecting code where its codewords are binary strings that share the same Hamming weight k. Its code length m is the bit-length of its codewords and code size n is the number of distinct codewords. Clearly, for a fixed Hamming weight k, to construct a constant-weight code of size n, one must choose its code length m, s.t., $\binom{m}{k} \geq n$. Then, we have

$$m \in O(\sqrt[k]{k!n} + k).$$

We denote the constant-weight code with length m and Hamming weight k by $CW(m,k)$.

Mahdavi-Kerschbaum [27] propose ways to map both indices and keywords to constant-weight codewords. Algorithm 1 shows how they map indices (i.e., $i \in [n]$) to $CW(m,k)$. Intuitively, it maps each i to the i-th valid codeword from a sorted list of codewords, with a complexity $O(m+k)$.

Algorithm 1 Mapping indices to constant-weight codewords

Input $i \in [n], m, k \in \mathbb{N}$ with $\binom{m}{k} \geq n$
Output $\mathbf{x} \in CW(m,k)$

$j := i$
$l := k$
$\mathbf{x} := 0^m$
for $m' := m-1, ..., 0$ **do**
  if $j \geq \binom{m'}{l}$ **then**
    $\mathbf{x}[m'] := 1$
    $j := j - \binom{m'}{l}$
    $l := l - 1$
  end if
  if $l = 0$ **then**
    break
  end if
end for
return $\mathbf{x}$

Algorithm 2 Mapping keywords to constant-weight codewords

Input $kw \in \mathsf{KW}, m, k \in \mathbb{N}$, a set of hash functions $(H_i : |\mathsf{KW}| \rightarrow [m])$
Output $\mathbf{x} \in CW(m,k)$

$i := 1$
$k' := 0$
$\mathbf{x} := 0^m$
while $k' < k$ **do**
  $h := H_i(kw)$
  if $\mathbf{x}[h] = 0$ **then**
    $\mathbf{x}[h] := 1$
    $k' := k' + 1$
  else
    $i := i + 1$
  end if
end while
return $\mathbf{x}$

It becomes tricky to map keywords to constant-weight codewords: keywords could be from a large domain KW; m and k need to be chosen s.t. $\binom{m}{k} \geq |\mathsf{KW}|$, which could lead to prohibitively large m, k. To address this issue, Mahdavi-Kerschbaum [27] propose a lossy mapping (Algorithm 2), which allows a small probability that distinct keywords are mapped to the same codeword.

An equality operator for checking the equality of two values is defined as follows:

Definition 1 (Equality Operator)**.** *A function f is an equality operator over a domain D if $\forall x, y \in D$,*

$$f(x,y) = \begin{cases} 1 & \text{if } x = y \\ 0 & o.w. \end{cases}$$

An equality operator over constant-weight codewords could be:

$$f(\mathbf{x}, \mathbf{y}) = \prod_{\mathbf{y}[i]=1} \mathbf{x}[i] \tag{1}$$

2.3 Batch code

A *batch code* (n, M, L, B)*-BC* encodes a collection of n elements into M codewords distributed among B buckets; any L of the n elements can be recovered by fetching at most one codeword from each bucket. It is an essential building block for *multi-query* PIR: C issues one PIR query to each of the B buckets and receives B responses; to answer these B queries, S computes over all M codewords exactly once. Notice that smaller M leads to lower computation, and smaller B leads to lower communication. Given $M < L \cdot n$, the total computation done by S is lower than running L instances of single-query PIR separately.

Angel et al. [2] introduce the notion of *probabilistic batch code* (PBC) that differs from the traditional batch codes in

that it fails to be complete with probability p, in exchange for a smaller M and B. They provide a PBC construction based on 3-way cuckoo hashing, which encodes n elements into $M = 3n$ codewords distributed among $B = 1.5L$ buckets, with a failure probability of $p = 2^{-40}$.

We summarize the operations of a batch code as follows:

- $[(\mathbf{id}_1, \mathbf{C}_1), ..., (\mathbf{id}_B, \mathbf{C}_B)] \leftarrow \mathsf{Encode}([(id_1, pl_1), ..., (id_n, pl_n)])$, where $(\mathbf{id}_i, \mathbf{C}_i)$ denotes vectors of identifiers and codewords in the i-th bucket.
- $[id'_1, ..., id'_B] \leftarrow \mathsf{GenSchedule}([id_1, ..., id_L])$, which takes a set of L queries and outputs a query for each of the B buckets.
- $[pl_1, ..., pl_L] \leftarrow \mathsf{Decode}([C_1, ..., C_B])$, which takes B codewords and outputs L payloads.

2.4 Bloom filter

A *bloom filter* $BL(m,k)$ is a data structure for efficient membership test [4]. It is a bit array $\mathbf{x}$ of length m initialized with 0s. It is also equipped with k hash functions $\{H_i\}$; the output of each hash function is uniformly distributed in $[m]$. To insert an element x into the bloom filter, one computes k positions: $h_i = H_i(x)\ \forall i \in [k]$, and set each of these k positions in $\mathbf{x}$ as 1 ($\mathbf{x}[h_i] := 1$). To test if an element has been inserted into the bloom filter, k positions are computed in the same way; if any of these positions in $\mathbf{x}$ is 0, the element is for sure not in the bloom filter; otherwise, the element is declared to be in the bloom filter with the following false positive rate:

$$\varepsilon = (1 - e^{-kL/m})^k \tag{2}$$

where L is the number of elements that has been inserted into the bloom filter.

3 SIMD-based Homomorphic Equality Operator

Mahdavi-Kerschbaum [27] propose a *homomorphic equality operator* for constant-weight code based on SealPIR's oblivious expansion [2]. In more detail, to encrypt a codeword $\mathbf{x}$ of length m, they separately encrypt the k indices that correspond to 1s in $\mathbf{x}$ using FHE. To check equality between an encrypted $\mathbf{x}$ and a plain codeword $\mathbf{y}$, they first expand the encrypted indices such that each bit of $\mathbf{x}$ is in a separate ciphertext; then run the equality operator (cf. equation 1) on the encrypted bits. The oblivious expansion requires $\frac{2m(N-1)}{N}$ substitutions[1] and m ciphertext-plaintext multiplications [2]; and the equality operator requires $(k-1)$ ciphertext-ciphertext multiplications.

[1]The runtime of substitution is roughly twice that of ciphertext-plaintext multiplication.

Algorithm 3 SIMD-based homomorphic equality operator

Input $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ and $\mathbf{y}$, with $\mathbf{x}_1||...||\mathbf{x}_t = \mathbf{x}$
Output $\widetilde{\mathbf{v}}$: if $\mathbf{x} = \mathbf{y}$, $\mathbf{v}[1] = 1$ and others are 0s; o.w. $\mathbf{v} = 0^N$

1: Find the k indices $[i_1, ..., i_k]$ in $\mathbf{y}$, where $\mathbf{y}[i] = 1$
2: $a := 1$
3: **for** $i \in [i_1, ..., i_k]$ **do**
4: $\quad j := \lfloor i \,/\, N \rfloor$
5: $\quad c := i \,\%\, N - 1$
6: $\quad \widetilde{\mathbf{u}}_a \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{x}}_j, c)$
7: $\quad a := a + 1$
8: **end for**
9: $\widetilde{\mathbf{v}} := \mathsf{SIMDMul}(\widetilde{\mathbf{u}}_1, ..., \widetilde{\mathbf{u}}_k)$
10: $\mathbf{e} := 0^N$, $\mathbf{e}[1] := 1$
11: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{v}}, \mathbf{e})$

In this section, we propose a faster homomorphic equality operator based on SIMD, where $\mathbf{x}$ is directly encrypted into $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ s.t. $\mathbf{x} = \mathbf{x}_1||...||\mathbf{x}_t$ and $t = \lceil m/N \rceil$.[2] The homomorphic equality operator is described in Algorithm 3. For each index i with $\mathbf{y}[i] = 1$, it rotates the corresponding element in $\mathbf{x}$ to the first slot (Line 4-6), leading to a ciphertext $\widetilde{\mathbf{u}}_a$ with $\mathbf{u}_a[1] = \mathbf{x}[i]$. Then, it multiplies $[\widetilde{\mathbf{u}}_1, ..., \widetilde{\mathbf{u}}_k]$ altogether and gets $\widetilde{\mathbf{v}}$ (Line 9) with

$$\mathbf{v}[1] = \prod_{\mathbf{y}[i]=1} \mathbf{x}[i].$$

Clearly, $\mathbf{v}[1]$ is the result of the equality operator. In the end, it clears out other slots of $\widetilde{\mathbf{v}}$ by multiplying it to a plaintext $\mathbf{e}$ (Line 11).

Algorithm 3 requires k rotations and $(k-1)$ ciphertext-ciphertext multiplications, plus one ciphertext-plaintext multiplication. Given that the runtime of rotation is roughly twice that of ciphertext-plaintext multiplication, our approach saves $(\frac{5mN-4m}{N} - 2k - 1)$ ciphertext-plaintext multiplications compared to that proposed by Mahdavi-Kerschbaum [27]. More importantly, our approach supports batch processing: given that Algorithm 3 only uses the first slot of $\mathbf{u}$ (Line 6), we could in fact make full use of its N slots to run N equality tests in a batch.

Batched homomorphic equality operator. Algorithm 4 describes our batched version. It compares $\mathbf{x}$ with N different $\mathbf{y}$s. For each $\mathbf{y}$, it runs Algorithm 3 with two modifications:

- It rotates $\mathbf{x}[i]$ to the l-th slot, instead of the first slot (Line 7-8).
- It leaves out the ciphertext-ciphertext multiplication among $\widetilde{\mathbf{u}}$s. Instead, it clears out the useless slots of each $\widetilde{\mathbf{u}}$ and adds it to the corresponding $\widetilde{\mathbf{w}}$ (Line 10-11).

The above process leads to k ciphertexts $[\widetilde{\mathbf{w}}_1, ..., \widetilde{\mathbf{w}}_k]$ with $\mathbf{w}_a[j] = \mathbf{y}_j[i_a]$. Then, it multiplies $[\widetilde{\mathbf{w}}_1, ..., \widetilde{\mathbf{w}}_k]$ altogether and

[2]For simplicity, we assume N divides m.

Algorithm 4 Batched homomorphic equality operator

Input $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ and $[\mathbf{y}_1, ..., \mathbf{y}_N]$, with $\mathbf{x}_1||...||\mathbf{x}_t = \mathbf{x}$
Output $\widetilde{\mathbf{v}}$: for each $l \in [N]$, if $\mathbf{x} = \mathbf{y}_l$, $\mathbf{v}[l] = 1$; o.w. $\mathbf{v}[l] = 0$

1: Init $[\widetilde{\mathbf{w}}_1, ..., \widetilde{\mathbf{w}}_k]$: each $\widetilde{\mathbf{w}}_i$ is an encryption of 0^N
2: **for** $l := 1, ..., N$ **do**
3: Find the k positions $[i_1, ..., i_k]$ in $\mathbf{y}_l$, where $\mathbf{y}_l[i] = 1$
4: $a := 1$
5: **for** $i \in [i_1, ..., i_k]$ **do**
6: $j := \lfloor i \,/\, N \rfloor$
7: $c := i \,\%\, N - l$
8: $\widetilde{\mathbf{u}} \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{x}}_j, c)$
9: $\mathbf{e} := 0^N$, $\mathbf{e}[l] := 1$
10: $\widetilde{\mathbf{u}} \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{u}}, \mathbf{e})$
11: $\widetilde{\mathbf{w}}_a := \mathsf{SIMDAdd}(\widetilde{\mathbf{w}}_a, \widetilde{\mathbf{u}})$
12: $a := a + 1$
13: **end for**
14: **end for**
15: $\widetilde{\mathbf{v}} := \mathsf{SIMDMul}(\widetilde{\mathbf{w}}_1, ..., \widetilde{\mathbf{w}}_k)$
16: return $\widetilde{\mathbf{v}}$

gets $\widetilde{\mathbf{v}}$ (Line 15) with

$$\mathbf{v}[j] = \prod_{\mathbf{y}_j[i]=1} \mathbf{x}[i], \ \forall\ j \in [N].$$

In this way, it accomplishes N homomorphic equality operators with a *single* ciphertext-ciphertext multiplication (which is the most expensive operation).

Remark. Mahdavi-Kerschbaum [27] also use the SIMD technique (but in a different way) when benchmarking their homomorphic equality operators. They encrypt each bit of $\mathbf{x}$ in a separate SIMD ciphertext, which means they use m SIMD ciphertexts to encrypt a single $\mathbf{x}$. To make use of other slots, they encrypt N different $\mathbf{x}$s in these ciphertexts. Then, they measure the amortized time of comparing these $\mathbf{x}$s with a single $\mathbf{y}$. Notice that batching in this way has no relation with PIR; indeed, they did not apply this approach to their CwPIR. In contrast, our proposed homomorphic equality operator naturally implies PIR, as shown in next section.

4 Single-query PIRANA

In this section, we present our constant-weight PIR. It is mostly based on the batched homomorphic equality operator in Algorithm 4. The rough idea is to divide the database into $s := \lceil n/N \rceil$ chunks and run Algorithm 4 separately for each chunk.

4.1 Single-query PIRANA for small payloads

For small payloads, the size of which is smaller than the slot size i.e., $|pl| < p$, the workflow of PIRANA is as follows:

Algorithm 5 Single-query PIRANA for small payloads

Input $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ and $[(\mathbf{y}_1, pl_1), ..., (\mathbf{y}_n, pl_n)]$
Output $\widetilde{\mathbf{v}}$: Suppose $\mathbf{x} = \mathbf{y}_i$, then $\mathbf{v}[i \ \%\ N] = pl_i$ and other elements of $\mathbf{v}$ are 0s

1: $s := \lceil n/N \rceil$ ▷ For simplicity, we assume N divides n
2: **for** $i := 1, .., s$ **do**
3: Run Algorithm 4 with input $([\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t], [\mathbf{y}_{(i-1)N+1}, ..., \mathbf{y}_{(i-1)N+N}])$ and get output $\widetilde{\mathbf{u}}_i$
4: $\widetilde{\mathbf{u}}_i \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{u}}_i, [pl_{(i-1)N+1}, ..., pl_{(i-1)N+N}])$
5: **end for**
6: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{u}}_1, ..., \widetilde{\mathbf{u}}_s)$
7: return $\widetilde{\mathbf{v}}$

- **Setup.** Parameters for the homomorphic encryption are chosen and keys are generated. For each identifier id_i, S maps it to a constant-weight codeword $\mathbf{y}_i$ of length m and weight k.
- **Query.** C maps its query to a constant-weight codeword $\mathbf{x}$ in the same way as S; encrypts $\mathbf{x}$ into $t = \lceil m/N \rceil$ SIMD ciphertexts $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ s.t. $\mathbf{x} = \mathbf{x}_1||...||\mathbf{x}_t$; and sends $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ to S.
- **Answer.** S runs Algorithm 5 to generate a response. It proceeds in two phases:
  1. **Selection vector generation.** For each chunk, S runs Algorithm 4 with $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ and the corresponding $\mathbf{y}$s in that chunk, and gets an encrypted *selection vector* $\widetilde{\mathbf{u}}$ (Line 3).
  2. **Inner product calculation.** S multiplies the payloads to the corresponding slots of $\widetilde{\mathbf{u}}$ (Line 4). Given that $\mathbf{x}$ matches (at most[3]) one $\mathbf{y}$, there will be (at most) one "1" in all slots of all $\mathbf{u}$s, hence only (at most) one payload will be kept and all others will be cleared out. In the end, S adds together all ciphertexts (Line 6), and returns the result $\widetilde{\mathbf{v}}$ to C.
- **Extract.** C decrypts $\widetilde{\mathbf{v}}$ and outputs the non-empty slot (if any).

In this way, S needs to run $s \cdot k \cdot N$ rotations, $(s \cdot k \cdot N + s)$ ciphertext-plaintext multiplications, and $s \cdot (k-1) = \lceil n/N \rceil \cdot (k-1)$ ciphertext-ciphertext multiplications (recall that CwPIR [27] needs to run $(k-1) \cdot n$ ciphertext-ciphertext multiplications, which is its main overhead).

We could further reduce the number of rotations from $s \cdot k \cdot N$ to $t \cdot (N-1)$ (notice that $s \cdot k > n \gg t$). The key observation is that, in Line 8 of Algorithm 4, the same $\widetilde{\mathbf{u}}$ will appear multiple times. Therefore, we could pre-compute all possible $\widetilde{\mathbf{u}}$s. Algorithm 6 describes this optimization. In more detail,

[3] In index PIR, $\mathbf{x}$ matches exactly one $\mathbf{y}$; in keyword PIR, $\mathbf{x}$ matches one $\mathbf{y}$ or there is no match.

Algorithm 6 Single-query PIRANA for small payloads with a reduced number of rotations

Input $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ and $[(\mathbf{y}_1, pl_1), ..., (\mathbf{y}_n, pl_n)]$
Output $\widetilde{\mathbf{v}}$: Suppose $\mathbf{x} = \mathbf{y}_i$, then $\mathbf{v}[i \ \% \ N] = pl_i$ and other elements of $\mathbf{v}$ are 0s

1: Init $\begin{bmatrix} \widetilde{\mathbf{u}}_{1,1} & ... & \widetilde{\mathbf{u}}_{1,N} \\ \vdots & \ddots & \vdots \\ \widetilde{\mathbf{u}}_{t,1} & ... & \widetilde{\mathbf{u}}_{t,N} \end{bmatrix}$: each $\mathbf{u}_{i,j}$ is an encryption of 0^N
2: **for** $i := 1, ..., t$ **do**
3: **for** $j := 1, ..., N-1$ **do**
4: $\widetilde{\mathbf{u}}_{i,j} \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{x}}_t, (j-1))$
5: **end for**
6: **end for**
7: $s := \lceil n/N \rceil$ ▷ For simplicity, we assume N divides n
8: **for** $j := 1, .., s$ **do**
9: Init $[\widetilde{\mathbf{w}}_1, ..., \widetilde{\mathbf{w}}_k]$: each $\widetilde{\mathbf{w}}$ is an encryption of 0^N
10: **for** $l := 1, ..., N$ **do**
11: Find the k positions $[i_1, ..., i_k]$ in $\mathbf{y}_{(j-1)N+l}$, where $\mathbf{y}_{(j-1)N+l}[i] = 1$
12: $a := 1$
13: **for** $i \in [i_1, ..., i_k]$ **do**
14: $\mathbf{e} := 0^N$, $\mathbf{e}[l] := 1$
15: $\widetilde{\mathbf{f}} \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{u}}_{\lfloor i \ / \ N \rfloor, i \ \% \ N-l}, \mathbf{e})$
16: $\widetilde{\mathbf{w}}_a := \mathsf{SIMDAdd}(\widetilde{\mathbf{w}}_a, \widetilde{\mathbf{f}})$
17: $a := a + 1$
18: **end for**
19: **end for**
20: $\widetilde{\mathbf{g}} := \mathsf{SIMDMul}(\widetilde{\mathbf{w}}_1, ..., \widetilde{\mathbf{w}}_k)$
21: $\widetilde{\mathbf{v}}_j \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{g}}, \left[pl_{(j-1)N+1}, ..., pl_{(j-1)N+N}\right])$
22: **end for**
23: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_s)$
24: return $\widetilde{\mathbf{v}}$

for each $\widetilde{\mathbf{x}}_i$, S rotates it $(N-1)$ times to enumerate all possible rotations of $\widetilde{\mathbf{x}}_i$. This leads to $t \times N$ different $\widetilde{\mathbf{u}}$s (Line 1-6 in Algorithm 6). Then, it runs in a similar[4] way as Algorithm 5, with only one modification: instead of rotating $\widetilde{\mathbf{x}}$ to get $\widetilde{\mathbf{u}}$, S directly picks $\widetilde{\mathbf{u}}$ from the pre-computed values (Line 15). We emphasize that this trick also allows us to rotate 2^x slots per rotation ($x = 0$ in our case), which is more efficient than rotating an arbitrary number of slots.

4.2 Single-query PIRANA for large payloads

Notice that the returned ciphertext $\widetilde{\mathbf{v}}$ in Algorithm 5 6 has (at most) one non-empty slot. Algorithm 7 shows how we make full use of other empty slots to return a large payload (assuming a payload is as large as l ciphertexts, i.e., $|pl| = l \cdot N \cdot p$), and Figure 1 visualizes this process. Specifically, S

[4]The description is different because we can no longer use Algorithm 4 as a blackbox.

Algorithm 7 Single-query PIRANA for large payloads

Input $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ and $[(\mathbf{y}_1, pl_1), ..., (\mathbf{y}_n, pl_n)]$
Output $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_l]$: Suppose $\mathbf{x} = \mathbf{y}_i$, then $\mathbf{v}_1[1]||...||\mathbf{v}_1[N]||...||\mathbf{v}_l[1]||...||\mathbf{v}_l[N] = pl_i$ ▷ $|pl| = l \cdot N \cdot p$

1: $s := \lceil n/N \rceil$ ▷ For simplicity, we assume N divides n
2: Init $\begin{bmatrix} \mathbf{d}_{1,1} & ... & \mathbf{d}_{1,l \cdot N} \\ \vdots & \ddots & \vdots \\ \mathbf{d}_{s,1} & ... & \mathbf{d}_{s,l \cdot N} \end{bmatrix}$: each $\mathbf{d}_{i,j}$ is $[pl^j_{(i-1)N+1}, ..., pl^j_{(i-1)N+N}]$, and $pl^1_{(i-1)N+a}||...||pl^{l \cdot N}_{(i-1)N+a} = pl_{(i-1)N+a}$
3: Init $[\widetilde{\mathbf{w}}_1, ..., \widetilde{\mathbf{w}}_{l \cdot N}]$: each $\widetilde{\mathbf{w}}_i$ is an encryption of 0^N
4: **for** $i := 1, .., s$ **do**
5: Run Algorithm 4 with input $([\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t], \left[\mathbf{y}_{(i-1)N+1}, ..., \mathbf{y}_{(i-1)N+N}\right])$ and get output $\widetilde{\mathbf{u}}_i$
6: **for** $j := 1, ..., l \cdot N$ **do**
7: $\widetilde{\mathbf{f}}_j \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{u}}_i, \mathbf{d}_{i,j})$
8: $\widetilde{\mathbf{w}}_j \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{f}}_j, \widetilde{\mathbf{w}}_j)$
9: **end for**
10: **end for**
11: Init $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_l]$: each $\mathbf{v}_i$ is an encryption of 0^N
12: **for** $i := 1, ..., l$ **do**
13: **for** $j := 1, ..., N$ **do**
14: $\widetilde{\mathbf{v}}_i \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{v}}_i, \widetilde{\mathbf{w}}_{(i-1) \cdot N + j})$
15: $\widetilde{\mathbf{v}}_i \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{v}}_i, 1)$
16: **end for**
17: **end for**
18: return $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_l]$

splits each payload into $l \cdot N$ small blocks (Line 2) and runs as follows to answer a query:

❶ For each of the s chunks, it runs Algorithm 4 to get a selection vector (Line 5). Notice that the optimization in Algorithm 6 is also applicable here, but we leave it out for the ease of presentation.

❷ It multiplies the blocks of N payloads to each selection vector, and repeats this for all $l \cdot N$ blocks of each payload (Line 7).

❸ It adds together the ciphertexts from different chunks (Line 8), resulting in $l \cdot N$ ciphertexts.

❹ For every N ciphertexts, it rotates them so that their non-empty slots are interlaced, and adds them together (Line 11-17). By doing this, there will be l ciphertexts left for the whole database. An important optimization here is that we rotate $\widetilde{\mathbf{v}}$ instead of $\widetilde{\mathbf{w}}$, which allows us to rotate one slot per rotation.

Steps 2-4 above involve $s \cdot l \cdot N$ ciphertext-plaintext multiplications and $l \cdot (N-1)$ rotations. We remark that when s is small, S first rotates the selection vectors in a way like Line 2-6 in Algorithm 6 to enumerate all N possible rotations; and

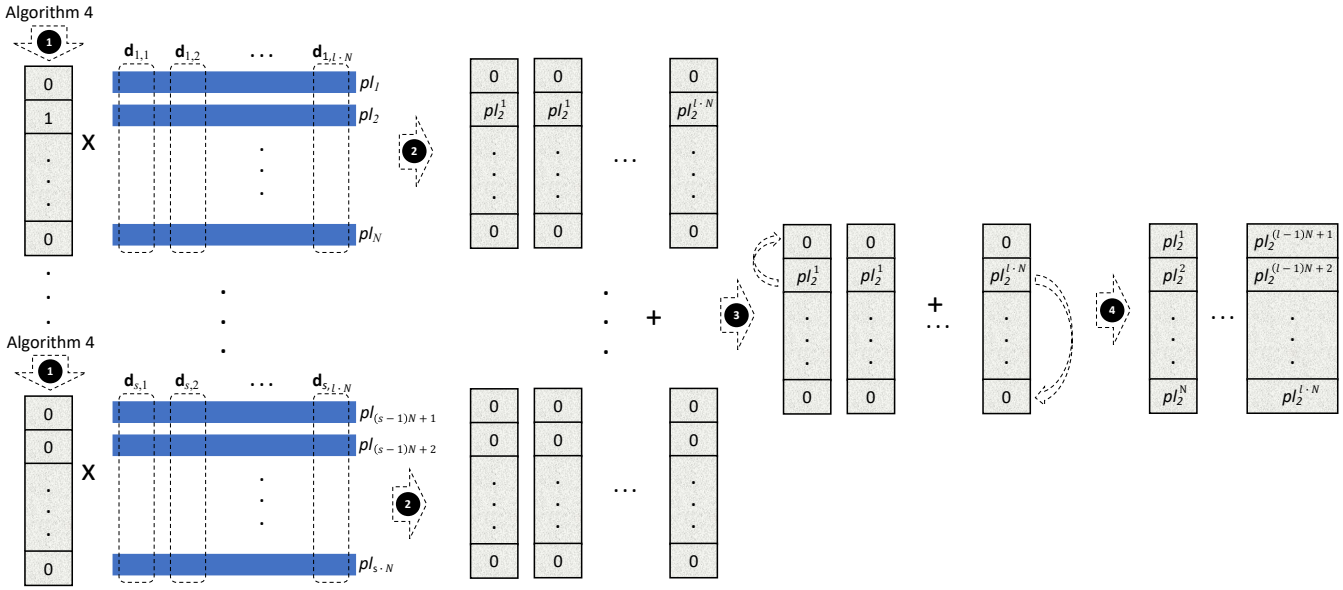

Figure 1: Workflow of single-query PIRANA for large payloads (visualization of Algorithm 7).

Algorithm 8 Single-query PIRANA for large payloads (small n)

Input $[\widetilde{\mathbf{x}}_1,...,\widetilde{\mathbf{x}}_l]$ and $[(\mathbf{y}_1,pl_1),...,(\mathbf{y}_n,pl_n)]$
Output $[\widetilde{\mathbf{v}}_1,...,\widetilde{\mathbf{v}}_l]$: Suppose $\mathbf{x}=\mathbf{y}_i$, then $\mathbf{v}_1[1]||...||\mathbf{v}_1[N]||...||\mathbf{v}_l[1]||...||\mathbf{v}_l[N]=pl_i$ ▷ $|pl|=l\cdot N\cdot p$

1: $s:=\lceil n/N\rceil$ ▷ For simplicity, we assume N divides n
2: Init $\begin{bmatrix}\mathbf{d}_{1,1} & ... & \mathbf{d}_{1,l\cdot N}\\ \vdots & \ddots & \vdots \\ \mathbf{d}_{s,1} & ... & \mathbf{d}_{s,l\cdot N}\end{bmatrix}$: each $\mathbf{d}_{i,j}$ is $[pl^j_{(i-1)N+1},...,pl^j_{(i-1)N+N}]$ with an rotation of $(j-1)$
3: Init $\begin{bmatrix}\widetilde{\mathbf{u}}_{1,1} & ... & \widetilde{\mathbf{u}}_{1,N}\\ \vdots & \ddots & \vdots \\ \widetilde{\mathbf{u}}_{s,1} & ... & \widetilde{\mathbf{u}}_{s,N}\end{bmatrix}$: each $\mathbf{u}_{i,j}$ is an encryption of 0^N
4: Init $[\widetilde{\mathbf{v}}_1,...,\widetilde{\mathbf{v}}_l]$: each $\mathbf{v}_i$ is an encryption of 0^N
5: **for** $i:=1,..,s$ **do**
6: Run Algorithm 4 with input $([\widetilde{\mathbf{x}}_1,...,\widetilde{\mathbf{x}}_l],[\mathbf{y}_{(i-1)N+1},...,\mathbf{y}_{(i-1)N+N}])$ and get output $\widetilde{\mathbf{u}}_i$
7: **for** $j:=1,...,N-1$ **do**
8: $\widetilde{\mathbf{u}}_{i,j}\leftarrow\mathsf{SIMDRotate}(\widetilde{\mathbf{u}}_i,(j-1))$
9: **end for**
10: **for** $c:=1,...,l$ **do**
11: **for** $j:=1,...,N$ **do**
12: $\widetilde{\mathbf{w}}_j\leftarrow\mathsf{SIMDPmul}(\widetilde{\mathbf{u}}_{i,j},\mathbf{d}_{i,(c-1)N+j})$
13: **end for**
14: $\widetilde{\mathbf{v}}_c\leftarrow\mathsf{SIMDAdd}(\widetilde{\mathbf{v}}_c,\widetilde{\mathbf{w}}_1,...,\widetilde{\mathbf{w}}_N)$
15: **end for**
16: **end for**
17: return $[\widetilde{\mathbf{v}}_1,...,\widetilde{\mathbf{v}}_l]$

then multiply the rotated selection vectors to the rotated payloads. In this case, the number of rotations becomes $s\cdot(N-1)$, which is more friendly to databases with a small n but large payloads. Algorithm 8 describes this optimization. In more detail, S first rotates the payloads so that the values to be selected are interlaced (Line 2). Notice that this step only needs to be done once for all queries. After running Algorithm 4, it rotates each of the s selection vectors $(N-1)$ times (Line 7-9). Then, it multiplies the rotated selection vectors with the corresponding rotated payloads (Line 11-13). In the end, it adds every $s\cdot N$ products together, resulting in l inner products.

In practice, we could combine Algorithm 7 and Algorithm 8 to minimize the number of rotations for specific l and s. In more detail, we could first run Algorithm 8, but with two modifications:

- It rotates the selection vector for $(\alpha-1)$ times (instead of $(N-1)$ times) and rotates $\frac{N}{\alpha}$ slots per rotation.
- In Line 14, it does not add the N $\widetilde{\mathbf{w}}$s together, instead it only adds the $\widetilde{\mathbf{w}}$s from s chunks together, resulting in $l\cdot N$ $\widetilde{\mathbf{w}}$s.

Then, it runs Algorithm 7 from Line 11, but only rotates for $(\frac{N}{\alpha}-1)$ times in Line 13-16. Now, the total number of rotations becomes:

$$f(\alpha)=(\alpha-1)s+(\frac{N}{\alpha}-1)l.$$

The first-order derivative of $f(\alpha)$ is:

$$\frac{df(\alpha)}{d\alpha}=s-\frac{N\cdot l}{\alpha^2}.$$

The second-order derivative of $f(\alpha)$ is:

$$\frac{d^2 f(\alpha)}{d\alpha^2} = \frac{1}{\alpha^3} > 0.$$

As the second-order derivative is always positive, we have:

1. If the first-order derivative is always negative (i.e. $s - \frac{N \cdot l}{\alpha^2} < 0$), $f(\alpha)$ is monotonically decreasing, hence it is minimal when α is maximal (i.e, $\alpha = N$). In this case, $s < \frac{l}{N}$.
2. If the first-order derivative is always positive (i.e. $s - \frac{N \cdot l}{\alpha^2} > 0$), $f(\alpha)$ is monotonically increasing, hence it is minimal when α is minimal (i.e, $\alpha = 1$). In this case, $s > l \cdot N$.
3. If the first-order derivative could be zero (i.e. $s - \frac{N \cdot l}{\alpha^2} = 0$), $f(\alpha)$ is minimal when $\alpha = \sqrt{Nl/s}$.

Formally, we have:

$$\underset{1 \le \alpha \le N}{\operatorname{argmin}} f(\alpha) = \begin{cases} N & s < \frac{l}{N} \\ \sqrt{Nl/s} & \frac{l}{N} \le s \le l \cdot N \\ 1 & s > l \cdot N \end{cases}$$

which implies that:

- when the payload size is large enough s.t., $l \ge sN$, we should run Algorithm 8;
- when the number of elements is large enough s.t., $s \ge lN$, we should run Algorithm 7;
- in the middle ground case, we should combine Algorithm 7 and Algorithm 8 as aforementioned, with $\alpha = \sqrt{Nl/s}$.

5 Multi-query PIRANA

As we mentioned in Section 2.3, existing multi-query PIR protocols need to query B buckets for L elements, and C has to use a separate single-query PIR to query each bucket. Given that $B > L$, this may introduce more communication overhead than running L instances of single-query PIR. Therefore, existing multi-query PIR protocols offer an unattractive trade-off: they reduce computation but add communication overhead.

In this section, we show that, by combining the idea of our single-query PIRANA with batch code and bloom filter, we come up with the *first* multi-query PIR protocol that can save both computation and communication.

The key observation is that our SIMD-based homomorphic equality operator (Algorithm 3 and 4) still work even if the non-zero bits' indices in $\mathbf{y}$ (denoted $I_{\mathbf{y}} = \{i_1, ..., i_k\}$) is a subset of the non-zero bits' indices in $\mathbf{x}$ (denoted $I_{\mathbf{x}} = \{i_1, ..., i_K\}$), i.e., $I_{\mathbf{y}} \subseteq I_{\mathbf{x}}$. Then, our starting point is to have C insert multiple queries into $\mathbf{x}$. However, this will easily lead to false positives: $I_{\mathbf{y}} \subseteq I_{\mathbf{x}_1} \cup I_{\mathbf{x}_2}$ but $\mathbf{y}$ is neither $\mathbf{x}_1$ nor $\mathbf{x}_2$. To reduce such false positive rate, we have C insert multiple queries into a bloom filter $BL(m,k)$ and encrypt the bloom filter in the same way as before, leading to $\widetilde{\mathbf{x}}$. Then, for an id, S computes the k hash values of id: $I_{\mathbf{y}} = \{i_1, ..., i_k\}$, and runs Line 2-11 of Algorithm 3, resulting in $\widetilde{\mathbf{v}}$ that satisfies:

- if id belongs to the multi-query, $\mathbf{v}[1] = 1$;
- otherwise, $\mathbf{v} = 0^N$.

However, this idea will not work for Algorithm 5-8. For example, in Line 6 of Algorithm 5, multiple non-zero payloads may collide in the same slot of $\widetilde{\mathbf{v}}$. Thanks to the batch code, we could use the bloom filter to query *one* payload from each bucket to avoid collisions, and we could use the same bloom filter to query all buckets.

5.1 Multi-query PIRANA for large payloads

The workflow of our multi-query PIR (for large payloads) is as follows:

Algorithm 9 Multi-query PIR - Query

Input $[id_1, ..., id_L]$
Output $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$
1: $[id'_1, ..., id'_B] \leftarrow \mathsf{GenSchedule}([id_1, ..., id_L])$
2: $\mathbf{x} := 0^m$
3: **for** $i := 1, ..., B$ **do**
4: **for** $j := 1, ..., k$ **do**
5: $\mathbf{x}[H_j(i||id'_i)] := 1$
6: **end for**
7: **end for**
8: $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t] \leftarrow \mathsf{SIMDEnc}(\mathbf{x})$ ▷ $t = \lceil m/N \rceil$ and we assume N divides m for simplicity

- **Setup.** Parameters for the homomorphic encryption are chosen and keys are generated. S encodes its database using a batch code $BC(n, M, L, B)$:

$$[(\mathbf{id}_1, \mathbf{C}_1), ..., (\mathbf{id}_B, \mathbf{C}_B)] \leftarrow \mathsf{Encode}([(id_1, pl_1), ..., (id_n, pl_n)]),$$

  where $(\mathbf{id}_i, \mathbf{C}_i)$ denotes vectors of identifiers and codewords in the i-th bucket.

- **Query.** C runs Algorithm 9 to generate the query ciphertexts. More specifically, given L original queries $[id_1, ..., id_L]$, C first runs GenSchedule of the batch code to generate queries $[id'_1, ..., id'_B]$ for each of the B buckets (Line 1). Then, it inserts these queries into a bloom filter $BL(m,k)$. Notice that the same identifier may appear in different buckets, hence each id'_i needs to be bound with its bucket index i to avoid collisions (Line 5). In the end, S encrypts the bloom filter $\mathbf{x}$ into t ciphertexts $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_t]$ (Line 8) and sends them to S.

- **Answer.** S runs Algorithm 10 to generate a response. In more detail, it first inserts each id_j (bound with its bucket index i) into a separate "bloom filter", denoted by $\mathbf{y}$ (Line 5); and then runs Algorithm 8 for each bucket to retrieve the desired codeword from that bucket (Line 8)[5]. Notice that if we adopt the optimization of Algorithm 6 (i.e., pre-computing all rotations), Line 1-6 of Algorithm 6 only need to be done once and can be reused for all B buckets.

- **Extract.** C decrypts $[\widetilde{\mathbf{v}}_1,...,\widetilde{\mathbf{v}}_B]$ and gets $[C_1,...,C_B]$. Then, it runs

$$[pl_1,...,pl_L] \leftarrow \mathsf{Decode}([C_1,...,C_B]).$$

Algorithm 10 Multi-query PIR - Answer (for large payloads)

Input $[\widetilde{\mathbf{x}}_1,...,\widetilde{\mathbf{x}}_t]$, $[(\mathbf{id}_1,\mathbf{C}_1),...,(\mathbf{id}_B,\mathbf{C}_B)]$
Output $[\widetilde{\mathbf{v}}_1,...,\widetilde{\mathbf{v}}_B]$

1: **for** $i := 1,...,B$ **do**
2: **for** $j := 1,...,|\mathbf{id}_i|$ **do**
3: $\mathbf{y}_{i,j} := 0^m$
4: **for** $l := 1,...,k$ **do**
5: $\mathbf{y}_{i,j}[H_l(i||id_j)] := 1$
6: **end for**
7: **end for**
8: Run Algorithm 8 with input $([\widetilde{\mathbf{x}}_1,...,\widetilde{\mathbf{x}}_t],[(\mathbf{y}_{i,1},C_{i,1}),...,(\mathbf{y}_{i,|\mathbf{id}_i|},C_{i,|\mathbf{id}_i|})])$ and get output $\widetilde{\mathbf{v}}_i$ ▷ Notice that Algorithm 8 will return multiple $\widetilde{\mathbf{v}}$s for a large payload. We leave out this detail for the ease of presentation.
9: **end for**
10: return $[\widetilde{\mathbf{v}}_1,...,\widetilde{\mathbf{v}}_B]$

5.2 Multi-query PIRANA for small payloads

If $|C| > N \cdot p$, the slots of each $\widetilde{\mathbf{v}}$ (returned by Algorithm 10) can be fully utilized; otherwise, some slots will be empty. In the later case, we could add different $\widetilde{\mathbf{v}}$s together to reduce the response size. However, the non-zero payloads from different $\widetilde{\mathbf{v}}$s may collide in the same slot.

Given that there is only one non-zero payload in each ciphertext, we could rotate these non-zero payloads to make them interlaced. The challenge is that the original positions of these non-zero payloads are unknown to S. To this end, we design a scheme for *oblivious rotation* (in Algorithm 11), which allows S to rotate a non-zero element from an unknown slot to a designated slot.

The idea of oblivious rotation is borrowed from binary search. Suppose there is a single non-zero element in an SIMD ciphertext $\widetilde{\mathbf{u}}$; that element is either in the first-half or in the second-half of the N slots of $\widetilde{\mathbf{u}}$. If the designated slot is in the first-half, S generates a ciphertext $\widetilde{\mathbf{v}}$, which is a rotation of $\widetilde{\mathbf{u}}$ from the second-half to the first-half (Line 5). Then, it adds $\widetilde{\mathbf{v}}$ and $\widetilde{\mathbf{u}}$ (Line 6). Now, the non-zero element is for sure in the first-half, then it clears out the second-half of $\widetilde{\mathbf{u}}$ (Line 7). Similarly, if the designated slot is in the second-half, S rotates $\widetilde{\mathbf{u}}$ from the first-half to the second-half (Line 10) and clears out the first half (Line 12). S runs this process recursively until reaching the designated slot. Now, the non-zero element is in the i-th slot of the returned ciphertext $\widetilde{\mathbf{v}}$.

Algorithm 11 Oblivious rotation

Input $i, \widetilde{\mathbf{u}}$ ▷ there is only one non-zero element in $\widetilde{\mathbf{u}}$ and its position is unknown.
Output $\widetilde{\mathbf{v}}$ ▷ rotate the non-zero element to the i-th slot.

1: $a := 1, b := N$
2: **for** $j := 1,...,\log N$ **do**
3: $c := a + (b-a)/2$
4: **if** $c > i$ **then**
5: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{u}}, 2^j)$
6: $\widetilde{\mathbf{u}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{u}}, \widetilde{\mathbf{v}})$
7: set $\mathbf{u}[c : (c+2^j)]$ to 0s ▷ multiply $\widetilde{\mathbf{u}}$ by a plaintext
8: $b := c - 1$
9: **else**
10: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{u}}, -2^j)$
11: $\widetilde{\mathbf{u}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{u}}, \widetilde{\mathbf{v}})$
12: set $\mathbf{u}[(c-2^j) : c]$ to 0s
13: $a := c + 1$
14: **end if**
15: **end for**
16: return $\widetilde{\mathbf{v}} := \widetilde{\mathbf{u}}$

Algorithm 12 shows how S answers multi-query for small payloads. It assumes each codeword C can fit into a slot, i.e., $|C| < p$. However, it can trivially support the case where $|C| < \frac{N}{B} \cdot p$.

Algorithm 12 Multi-query PIR - Answer (for small payloads)

Input $[\widetilde{\mathbf{x}}_1,...,\widetilde{\mathbf{x}}_t]$, $[(\mathbf{id}_1,\mathbf{C}_1),...,(\mathbf{id}_B,\mathbf{C}_B)]$
Output $\widetilde{\mathbf{v}}$

1: **for** $i := 1,...,B$ **do**
2: **for** $j := 1,...,|\mathbf{id}_i|$ **do**
3: $\mathbf{y}_{i,j} := 0^m$
4: **for** $l := 1,...,k$ **do**
5: $\mathbf{y}_{i,j}[H_l(i||id_j)] := 1$
6: **end for**
7: **end for**
8: Run Algorithm 6 with input $([\widetilde{\mathbf{x}}_1,...,\widetilde{\mathbf{x}}_t],[(\mathbf{y}_{i,1},C_{i,1}),...,(\mathbf{y}_{i,|\mathbf{id}_i|},C_{i,|\mathbf{id}_i|})])$ and get output $\widetilde{\mathbf{u}}_i$
9: Run Algorithm 11 with input $(i,\widetilde{\mathbf{u}}_i)$ and get output $\widetilde{\mathbf{w}}_i$
10: **end for**
11: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{w}}_1,...,\widetilde{\mathbf{w}}_B)$
12: return $\widetilde{\mathbf{v}}$

[5] We use Algorithm 8 because the bucket size is likely to be small. It it is large, we could use Algorithm 7 instead.

5.3 False positives in a bloom filter

Recall that a bloom filter has a false positive rate:

$$\varepsilon = (1 - e^{-kL/m})^k.$$

Given that we need to run M membership tests for a multi-query, the false positive rate for a multi-query is $M\varepsilon$. For example, if we configure the bloom filter s.t. $\varepsilon = 2^{-42}$, we could get a false positive rate of $\approx 2^{-20}$ for a database of 2^{20} elements.

If a false positive happens in a bucket, the query in that bucket will fail, but will not affect other buckets. Still, C will not get all its desired elements. However, before sending the query, C could run Step 2-7 of Algorithm 10 by itself to learn which bucket will fail (if any). In scenarios where C needs to retrieve more than a batch of L elements, it can adjust its current set of queries, e.g., move the *id* that causes false positives to the next batch. In scenarios where false positives are unacceptable, we could switch back to the constant-weight PIR as we will show next.

5.4 Multi-query PIRANA based on constant-weight code

Notice that the number of elements in each bucket is roughly $\frac{M}{B}$, which could be much smaller than n. If we use our constant-weight PIR to query each bucket, we could use a smaller m, which allows us to batch multiple constant-weight codewords into a single ciphertext. Recall that in Algorithms 5-8, each $\mathbf{x}$ corresponds to a single constant-weight codeword. If $N > m$, we could in fact have C store multiple codewords in $\mathbf{x}$. For example, to query $L = 256$ elements from $n = 2^{20}$ elements, if we use PBC [2] to encode the database, we have $M = 3n = 3\ 145\ 728$, $B = 1.5L = 384$, and there are $\frac{M}{B} = 8\ 192$ elements in each bucket. If we set $k = 2$, we have $m = 129$. For a ciphertext with $N = 8\ 192$, we could in fact have C batch $\lfloor \frac{N}{m} \rfloor = 63$ queries into a single ciphertext. As a result, instead of sending $B = 384$ ciphertexts, C only needs to send $\lceil \frac{B}{63} \rceil = 7$ ciphertexts to S. Algorithm 13 shows how C generates query ciphertexts in this way.

Algorithm 13 Constant-weight multi-query PIR - Query

Input $[id_1, ..., id_L]$
Output $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_d]$ ▷ $d = \lceil \frac{B}{c} \rceil$ where $c = \lfloor \frac{N}{m} \rfloor$
1: $[id'_1, ..., id'_B] \leftarrow \mathsf{GenSchedule}([id_1, ..., id_L])$
2: Map $[id'_1, ..., id'_B]$ to constant-weight code: $[\mathbf{z}_1, ..., \mathbf{z}_B]$
3: **for** $i := 1, ..., d$ **do**
4: $\quad \mathbf{x}_i := \mathbf{z}_{(i-1)c+1} || ... || \mathbf{z}_{(i-1)c+c}$
5: $\quad \widetilde{\mathbf{x}}_i \leftarrow \mathsf{SIMDEnc}(\mathbf{x}_i)$
6: **end for**

Algorithm 14 shows how S generates responses for constant-weight multi-query PIR with large payloads. For small payloads, it can simply replace Algorithm 7 with Algorithm 6 and Algorithm 11. Knowing the split points for different codewords, S can easily extend $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_d]$ into B ciphertexts. However, this is unnecessary; instead, S can use them directly to compute the selection vectors (Line 4). By assuming $|\mathbf{id}_i| = N$, S can make full use of the slots in $\widetilde{\mathbf{x}}_j$ for ciphertext-ciphertext multiplication when running Algorithm 8. When $|\mathbf{id}_i| < N$, S can pad $[(\mathbf{y}_{i,1}, C_{i,1}), ..., (\mathbf{y}_{i,|\mathbf{id}_i|}, C_{i,|\mathbf{id}_i|})]$ with elements from other buckets. Similarly, when $|\mathbf{id}_i| > N$, S can pad $[(\mathbf{y}_{i,|\mathbf{id}_i|-N}, C_{i,|\mathbf{id}_i|-N}), ..., (\mathbf{y}_{i,|\mathbf{id}_i|}, C_{i,|\mathbf{id}_i|})]$ with elements from other buckets. We leave out this detail for the ease of presentation.

Algorithm 14 Constant-weight multi-query PIR - Answer (for large payloads)

Input $[\widetilde{\mathbf{x}}_1, ..., \widetilde{\mathbf{x}}_d]$, $[(\mathbf{id}_1, \mathbf{C}_1), ..., (\mathbf{id}_B, \mathbf{C}_B)]$
Output $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_B]$
1: **for** $i := 1, ..., B$ **do**
2: $\quad$ Map $\mathbf{id}_i$ to constant-weight code: $[\mathbf{y}_{i,1}, ..., \mathbf{y}_{i,|\mathbf{id}_i|}]$
3: $\quad$ Pick $\widetilde{\mathbf{x}}_j$ where the i-th query codeword resides
4: $\quad$ Run Algorithm 7 with input $(\widetilde{\mathbf{x}}_j, [(\mathbf{y}_{i,1}, C_{i,1}), ..., (\mathbf{y}_{i,|\mathbf{id}_i|}, C_{i,|\mathbf{id}_i|})])$ and get output $\widetilde{\mathbf{v}}_i$ ▷ For simplicity, we assume $|\mathbf{id}_i| = N$
5: **end for**
6: return $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_B]$

6 Evaluation

In this section, we provide a full-fledged implementation for PIRANA and systematically evaluate its performance.

6.1 Implementation

We fully implement PIRANA based on the Microsoft SEAL homomorphic encryption library (version 4.0)[6]. We use the Brakerski-Fan-Vercauteren (BFV) [5, 18] scheme with $N = 8192$, 30-bit plaintext modulus (for each SIMD slot) and 218-bit ciphertext modulus, which enables us to have a 128-bit security level.

All ciphertext-plaintext multiplications and ciphertext-ciphertext additions are implemented using number theoretic transform (NTT). To this end, we encode all payloads into NTT forms so that they can be multiplied directly to the NTTed selection vectors. Such payload encoding only needs to be done once and can be used for all queries.

We run all experiments on an Intel Xeon Cooper Lake (with a base frequency of 3.4 GHz and turbo frequency of 3.8 GHz) server running Ubuntu 20.04. This setup is similar to the setting of CwPIR [27]. All experiments were repeated 5 times and average values (the variances are very small) were reported.

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

	# elements n	2^8	2^9	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}
	codeword length m	24	33	46	65	92	129	182	257	363
	DB Size (MB)	5.2	10	21	42	84	170	340	670	1 300
CwPIR [27]	Selection Vec. (s)	3.9	7.8	15.5	31.0	61.7	123.1	246.2	492.7	983.3
$k=2$	Inner Product (s)	0.2	0.4	0.8	1.6	3.3	6.5	13.1	26.2	52.3
$N=2^{13}$	Total server (s)	4.1	8.2	16.3	32.6	65.0	129.7	259.4	518.9	1 035.6
single-query	Selection Vec. (s)	1.0	1.1	1.3	1.7	2.5	4.1	7.3	13.7	26.8
PIRANA	Inner Product (s)	0.7	0.7	0.7	0.8	1.0	1.3	2.3	3.7	6.9
$k=2$	Total server (s)	1.7	1.8	2.1	2.5	3.5	5.4	9.6	17.5	33.6
$N=2^{13}$	Speedup	2.4×	4.6×	7.8×	13×	18.6×	24×	27×	29.6×	30.8×

Table 2: Microbenchmark of PIRANA and CwPIR.

6.2 Evaluation of single-query PIRANA

We first compare PIRANA (a combination of Algorithm 7 and Algorithm 8 with α depending on the number of elements n and payload sizes) with CwPIR [27] in terms of single-query PIR. To this end, we reproduce the results of CwPIR (Table 7 and Figure 2) reported in their original paper, by running their open-sourced implementation[7]. Our reproduced results are better than their original results, for two reasons: (1) our CPU is more advanced, and (2) we use a newer version of the SEAL library (they use version 3.6). We measure PIRANA by the same metric and list the comparision results in Table 2 and Figure 2.

Microbenchmark. We first microbenchmark the main stages of CwPIR (reproducing Table 7 in [27]) and PIRANA. We measure the runtime for an increasing number of elements n (from 2^8 to 2^{16}) with a somehow constant payload size (20KB). We set $k=2, N=8192$ for both CwPIR and PIRANA. For each n, we choose the minimal m that satisfies $\binom{m}{k} > n$. With such parameter configurations, the client runtime and the upload/download bandwidth will be the same in both CwPIR and PIRANA: in both approaches, C will generate a single ciphertext, send it to S, receive and decrypt the same number of ciphertexts. Furthermore, the client runtime is insignificant compared to the server runtime. To this end, we focus on comparing the server runtime.

Recall that S in CwPIR [27] proceeds in three stages to answer a query: query expansion, selection vector generation, and inner product calculation. Since there is no query expansion in PIRANA, we combine the runtime of query expansion in CwPIR into its selection vector generation for the ease of presentation.

Table 2 lists the microbenchmark results. As expected, PIRANA's advantage in selection vector generation is significant compared to CwPIR. Recall that CwPIR needs to run $(k-1)\cdot n$ ciphertext-ciphertext multiplications to generate a selection vector, whereas PIRANA only needs to run $\lceil n/N \rceil \cdot (k-1)$ times. Surprisingly, inner product calculation in PIRANA is also much faster than that in CwPIR. The reason is that the selection vector generated by CwPIR has n ciphertexts, which need to be transformed to NTT to be multiplied to the payloads, hence they need to run NTT for n times. In contrast, our selection vector only has $\lceil n/N \rceil$ ciphertexts, so we save upto N times of NTT.

To sum up, given our advantages in both selection vector generation and inner product calculation, we achieve upto 30.8× speedup over CwPIR.

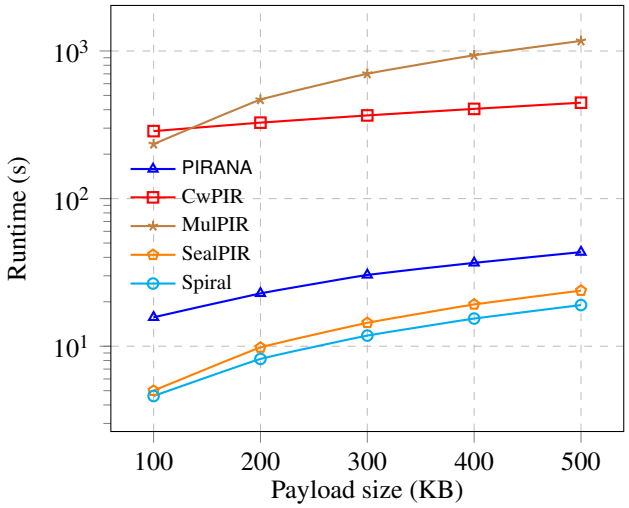

Figure 2: Runtime for different payload sizes (# elements is 2^{14}).

Large payloads. Next, we set $n=2^{14}$ and measure server runtime by increasing the payload size from 100KB to 500KB (reproducing Figure 2 in [27]). The code provided by Mahdavi-Kerschbaum can support at most 100KB payloads for $n=2^{14}$, hence we extrapolate their runtime for larger payloads. Notice that the results (Figure 2 in [27]) reported by their original paper were also extrapolated. In contrast, our PIRANA can indeed support large payloads, hence all results of PIRANA were truly measured (rather than being extrapolated). Figure 2 shows that PIRANA's advantage over CwPIR is also

[7] https://github.com/RasoulAM/constant-weight-pir

significant for large payloads.

We also compare our runtime with SealPIR[8] [2], MulPIR[9] [1] and Spiral[10] [29]. The open-sourced code for these protocols cannot support large payloads either (SealPIR and Spiral can support at most 10KB payloads and MulPIR can support 20KB at most). Therefore, we again extrapolated their runtime. The results show that MulPIR is even much slower than CwPIR, which is consistent with the results reported in [27]. The results also show that the performance of SealPIR, Spiral and PIRANA are in the same level; even though PIRANA is slightly slower, it gains the benefit of better supporting keyword queries.

6.3 Evaluation of multi-query PIRANA

Next, we compare multi-query PIRANA with the state-of-the-art multi-query PIR, i.e., SealPIR with PBC [2]. Recall that PBC is constructed based on 3-way cuckoo hashing, which encodes n elements into $M = 3n$ codewords distributed among $B = 1.5L$ buckets, with a failure probability of $p = 2^{-40}$. To provide a fair comparison, we have to configure the bloom filter to satisfy $M\varepsilon < 2^{-40}$. However, this will lead to a large filter length m. For example, to query $n = 2^{14}$ elements in a database, we need to set $\varepsilon = 2^{-56}$ to achieve $M\varepsilon \approx 2^{-40}$. Given that we want to minimize the multiplicative depth, we set the number of hash functions $k = 5$. For a multi-query of size $L = 256$, we need to set $m = 1\ 485\ 112$ based on Equation 2, hence C needs to send $\lceil \frac{m}{N} \rceil$ ciphertexts to S, which is 91 for $N = 16\ 384$. Even though it is still smaller than multi-SealPIR, which needs to send $B = 384$ ciphertexts, it is less satisfactory.

Given this failure probability, we use our constant-weight version PIRANA for multi-query, i.e., Algorithm 13 and 14. We use PBC as the batch code as well, with the same configuration as multi-SealPIR [2]. For $n = 2^{14}$ elements in the database and $L = 256$ queries, we only need to set the constant-weight codeword length $m = 10$. As a result, we can batch 819 codewords in a ciphertext with $N = 8\ 192$, and C only needs to send one ciphertext to S.

Table 3 lists the microbenchmark results for both multi-SealPIR and multi-PIRANA. It shows that multi-PIRANA is significantly better than multi-SealPIR in terms of query time and query size. This is because multi-PIRANA only requires C to send a single ciphertext to query B buckets, whereas multi-SealPIR needs to send B ciphertexts. In more detail, it takes 4ms for SealPIR to generate a single-query for $L = 1$. However, for multi-query, the amortized generation time for each query increases, as it needs to generate $B > L$ queries. In contrast, the amortized generation time for each query in multi-PIRANA is roughly the single-query generation time divided by L. For example, when $L = 256$, the amortized generation time for each query in multi-PIRANA is only 0.02ms, 225× faster than multi-SealPIR. The advantage of multi-PIRANA is also prominent when considering the query size (upload bandwidth). For $L = 256$, the amortized query size in multi-PIRANA is only 0.83KB; compared with the 136KB amortized query size in multi-SealPIR, we save 163.9× communication.

	# queries L	1	16	64	256
	# buckets B	1	24	96	384
multi-SealPIR [2] $N = 2^{12}$	query (ms)	4	4.5	4.5	4.5
	answer (s)	5	1.4	0.5	0.2
	extract (ms)	10	15	15	15
	query (KB)	90.7	136.1	136.0	136.0
	answer (KB)	1812	2716	2716	2716
multi-PIRANA $k = 2$ $N = 2^{13}$	query (ms)	4.4	0.28	0.07	0.02
	answer (s)	15.7	5.29	1.86	0.76
	extract (ms)	2.07	3.52	3.50	3.36
	query (KB)	211.3	13.2	3.3	0.83
	answer (KB)	402.2	754.9	754.9	754.9

Table 3: Amortized costs for multi-query SealPIR and PIRANA (2^{14} elements, 100KB payloads).

In this experiment, the payload size is 100KB, which fully occupies each ciphertext space, hence S returns $O(B)$ ciphertexts in both multi-SealPIR and multi-PIRANA. Nevertheless, multi-PIRANA still has a significantly better extraction time and answer size than multi-SealPIR, due to the expansion factor in SealPIR (cf. Section 7). The server runtime for answer generation in multi-PIRANA is 3.8× slower than that in multi-SealPIR. Considering our gain in client runtime and bandwidth, this is a real bargain especially in mobile applications.

7 Related Work

Early single-server PIR. Most of the early single-server PIR protocols follow the blueprint of Kushilevitz and Ostrovsky [23]: representing the database as a D-dimensional hypercube. The original protocol proposed by Kushilevitz and Ostrovsky is based on additively homomorphic encryption, with a query size of $O(\sqrt{N}\log N)$ and a response size of $O(\sqrt{N})$. Cachin et al. [8] instead use the ϕ-Hiding assumption to achieve $O(\log^4 N)$ query size and $O(\log^D N)$ response size. Gentry and Ramazan [20] further reduce the query size of Cachin et al.'s approach to $O(\log^{3-o(1)} N)$. Chang [9] instantiates Kushilevitz-Ostrovsky's approach with Paillier homomorphic encryption to achieve $O(\sqrt{N}\log N)$ query size and $O(\log N)$ response size. This protocol was later generalized by Lipmaa [26] with Damgard-Jurik encryption [16] to achieve $O(\log^2 N)$ query size and $O(\log N)$ response size. As it has been pointed by Sion and Carbunar [35], such protocols

[8] https://github.com/microsoft/SealPIR
[9] https://github.com/OpenMined/PIR
[10] https://github.com/menonsamir/spiral

are even slower than trivially having C download the entire database [35].

Lattice-based PIR. Aguilar-Melchor et al. [28] propose XPIR, which applies a lattice-based homomorphic encryption [7] to the hypercube-based PIR. A major drawback of XPIR is its communication cost: even encoding the database into a 2 or 3-dimensional hypercube, the query vector still consists of hundreds or thousands of ciphertexts; and the high expansion factor of lattice-based cryptosystems makes the matter worse.

SealPIR [2] gets rid of this bottleneck by introducing a query expansion technique. In more detail, (1) C sends S a ciphertext that homomorphically encrypts its desired index i; (2) S obliviously expands it into a selection vector of n ciphertexts where the i-th ciphertext encrypts 1 and others encrypt 0; (3) S returns the inner product between the selection vector and the payloads. Notice that a ciphertext can only be expanded into $O(N)$ ciphertexts. Therefore, C needs to send $O(n/N)$ ciphertexts to S. To reduce this communication overhead, S structures the database as a D-dimensional hyperrectangle so that the above process can be recursively performed for each dimension. As a result, C only needs to send $O(D \cdot \sqrt[D]{n}/N)$ ciphertexts to S.

However, in this way, S needs to compute the inner products between selection vectors and *encrypted* payloads from the second dimension and on. To avoid the expensive ciphertext-ciphertext multiplications, SealPIR [2] treat the encrypted payloads as "plaintexts", and run plaintext-ciphertext multiplications instead. This technique trades one ciphertext-ciphertext multiplication to multiple plaintext-ciphertext multiplications, leading to a large expansion factor for the responses. OnionPIR [31] realizes ciphertext-ciphertext multiplication via *external product* [11], which reduces the response sizes but incurs a large computational overhead. Spiral [29] further improves this idea by composing Regev encryption [33] with GSW encryption [21] to achieve a faster external product.

MulPIR [1] uses alternative ways to reduce the communication of SealPIR. Namely, it uses symmetric key FHE to reduce the upload size and uses modulus switching to reduce the expansion factor. It also introduces a new query expansion scheme to halve the upload size for some specific parameter sets. However, as shown by our benchmarks (cf. Figure 2) and also the benchmarks in [27], MulPIR requires a large server runtime for answer generation.

PIR with preprocessing. Beimel et al. [3] proved that a secure PIR scheme must incur $\Omega(n)$ server-side work. Indeed, if S touches fewer than n elements to answer a query, it will learn that the untouched elements are for sure not to be retrieved. To circumvent this lower bound, Beimel et al. [3] propose the notion of PIR with preprocessing, in which the database is processed in an encoded form. However, the scheme proposed by Beimel et al. is only applicable to multi-server PIR. Patel et al. [32] propose a single-server solution, where C retrieves some helper data during preprocessing and uses them to run online queries in sublinear time. The computation cost of preprocessing is still linear but they are mostly symmetric-key operations. However, the preprocessing stage requires linear communication, which is less desirable. In a recent breakthrough [15], Corrigan-Gibbs and Kogan propose a two-server PIR scheme, which shows promising sublinear efficiency in both theory and practice. They also propose a way to transform their solution to single-server PIR, but it requires running black-box PIR for multiple times during preprocessing. This idea was further explored in [14, 34], but none of them has an overall efficiency that is better than PIR without preprocessing.

Keyword PIR. Most existing solutions for keyword PIR transform the problem to index PIR. For example, Chor et al. [12] propose a solution, where C interactively query S to obtain the index of its desired keyword so that index PIR can be performed. Another example is to use probabilistic hashing to map keywords into a small table and then use index PIR to query the table [1]. However, we believe that equality operators are the most natural solutions for keyword PIR, and we have shown that, by far, PIRANA provides the best support for equality operators.

8 Conclusion

In this paper, we propose a homomorphic constant-weight equality operator that supports batch processing. On that basis, we propose a novel PIR protocol named PIRANA. Compared with its closest competitor, PIRANA can support databases with a larger number of elements. We further extend PIRANA to support multi-query. To the best of our knowledge, it is the first multi-query PIR that can save both computation and communication. We provide a full-fledged implementation and extensive evaluation.

References

[1] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 1811–1828. USENIX Association, 2021.

[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*, pages 962–979. IEEE Computer Society, 2018.

[3] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In Mihir Bellare, editor, *Advances in Cryptology - CRYPTO 2000, 20th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 2000, Proceedings*, volume 1880 of *Lecture Notes in Computer Science*, pages 55–73. Springer, 2000.

[4] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[5] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. *IACR Cryptol. ePrint Arch.*, page 78, 2012.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Trans. Comput. Theory*, 6(3):13:1–13:36, 2014.

[7] Zvika Brakerski and Vinod Vaikuntanathan. Fully homomorphic encryption from ring-lwe and security for key dependent messages. In Phillip Rogaway, editor, *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, volume 6841 of *Lecture Notes in Computer Science*, pages 505–524. Springer, 2011.

[8] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 402–414. Springer, 1999.

[9] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. In Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan, editors, *Information Security and Privacy: 9th Australasian Conference, ACISP 2004, Sydney, Australia, July 13-15, 2004. Proceedings*, volume 3108 of *Lecture Notes in Computer Science*, pages 50–61. Springer, 2004.

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437. Springer, 2017.

[11] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *J. Cryptol.*, 33(1):34–91, 2020.

[12] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. *IACR Cryptol. ePrint Arch.*, page 3, 1998.

[13] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, USA, 23-25 October 1995*, pages 41–50. IEEE Computer Society, 1995.

[14] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In Orr Dunkelman and Stefan Dziembowski, editors, *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*, volume 13276 of *Lecture Notes in Computer Science*, pages 3–33. Springer, 2022.

[15] 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 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 44–75. Springer, 2020.

[16] Ivan Damgård, Mads Jurik, and Jesper Buus Nielsen. A generalization of paillier's public-key system with applications to electronic voting. *Int. J. Inf. Sec.*, 9(6):371–385, 2010.

[17] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. PIR-PSI: scaling private contact discovery. *Proc. Priv. Enhancing Technol.*, 2018(4):159–178, 2018.

[18] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[19] Craig Gentry. *A Fully Homomorphic Encryption Scheme*. PhD thesis, Stanford, CA, USA, 2009. AAI3382729.

[20] Craig Gentry and Zulfikar Ramzan. Single-database private information retrieval with constant communication rate. In Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung, editors, *Automata, Languages and Programming, 32nd International Colloquium, ICALP 2005, Lisbon, Portugal, July*

11-15, 2005, Proceedings, volume 3580 of *Lecture Notes in Computer Science*, pages 803–815. Springer, 2005.

[21] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[22] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, August 2021.

[23] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, FOCS '97, page 364, USA, 1997. IEEE Computer Society.

[24] Eyal Kushilevitz and Rafail Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 364–373. IEEE Computer Society, 1997.

[25] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *Proc. Priv. Enhancing Technol.*, 2016(2):115–134, 2016.

[26] Helger Lipmaa. An oblivious transfer protocol with log-squared communication. In Jianying Zhou, Javier López, Robert H. Deng, and Feng Bao, editors, *Information Security, 8th International Conference, ISC 2005, Singapore, September 20-23, 2005, Proceedings*, volume 3650 of *Lecture Notes in Computer Science*, pages 314–328. Springer, 2005.

[27] 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, Boston, MA, August 2022. USENIX Association.

[28] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR : Private information retrieval for everyone. *Proc. Priv. Enhancing Technol.*, 2016(2):155–174, 2016.

[29] Samir Jordan Menon and David J. Wu. SPIRAL: fast, high-rate single-server PIR via FHE composition. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 930–947. IEEE, 2022.

[30] Prateek Mittal, Femi Olumofin, Carmela Troncoso, Nikita Borisov, and Ian Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *20th USENIX Security Symposium (USENIX Security 11)*, San Francisco, CA, August 2011. USENIX Association.

[31] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onionpir: Response efficient single-server PIR. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2292–2306. ACM, 2021.

[32] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, pages 1002–1019. ACM, 2018.

[33] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.

[34] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV*, volume 12828 of *Lecture Notes in Computer Science*, pages 641–669. Springer, 2021.

[35] Radu Sion and Bogdan Carbunar. On the practicality of private information retrieval. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*. The Internet Society, 2007.

[36] Ni Trieu, Kareem Shehata, Prateek Saxena, Reza Shokri, and Dawn Song. Epione: Lightweight contact tracing with strong privacy. *IEEE Data Eng. Bull.*, 43(2):95–107, 2020.

[37] David J. Wu, Joe Zimmerman, Jérémy Planul, and John C. Mitchell. Privacy-preserving shortest path computation. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, Cali-*

fornia, USA, February 21-24, 2016. The Internet Society, 2016.