PIRANA: Faster (Multi-query) PIR via Constant-weight Codes

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—**Private information retrieval (PIR) is a crypto-graphic protocol that enables a wide range of privacy-preserving applications. Despite being extensively studied for decades, it is still not efficient enough to be used in practice. In this paper, we propose a novel PIR protocol named PIRANA, based on the recent advances in constant-weight codes. It is upto $188.6\times$ faster than the original constant-weight PIR (presented in Usenix SEC '22). Most importantly, PIRANA naturally supports multi-query. It allows a client to retrieve a batch of elements from the server with a very small extra-cost compared to retrieving a single element. To retrieve 2 730 elements, the amortized cost for retrieving a single element is only 3.9ms in runtime and 3.0KB in bandwidth. We also discuss a way to extend PIRANA to labeled private set intersection (LPSI). Compared to existing LPSI protocols, PIRANA is more friendly to the scenarios where the database updates frequently.**

1. Introduction

Suppose a server S holds a database of n elements, where each element consists of an identifier and a payload; *private information retrieval* (PIR) allows a client C to retrieve an element from S without revealing which element was retrieved. It enables a wide range of privacy-preserving applications such as private contact discovery [24], private contact tracing [48], private navigation [49], anonymous messaging [40], [35], safe browsing [31] and so on. Despite being extensively studied for decades, PIR is still a hot research topic in cryptography.

The PIR protocols can be roughly categorized into multi-server PIR [17] and single-server PIR [34]. 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 need this strong assumption, at the cost of introducing a huge performance overhead. It has been shown that the single-server protocols are even slower than trivially having C download the entire database [47].

Constant-weight PIR. Most of the single-server PIR pro-tocols structure the database as a D-dimensional hyper-cube [33], [38], [4], so that the communication overhead

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

could be $O(D \cdot \sqrt[D]{n})$. However, in this way, most of S's operations need to involve the payloads, which incurs a large computational overhead when the payload size is large. In contrast, Mahdavi-Kerschbaum [37] propose a PIR protocol that can generate a selection vector based on the identifiers only, and it only needs to multiply the payloads to the selection vector in the end; hence the main overhead of this protocol is somehow independent of the payloads. The protocol is 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). To generate a selection vector, CwPIR encodes all identifiers and queries into constant-weight codewords and runs a homomorphic equality operator on them, resulting in $(k-1)n$ ciphertext-ciphertext multiplications. Therefore, CwPIR cannot support databases with a large number of elements. Their experimental results also confirm this (cf. Table 13 in [37]).

Multi-query PIR. In many scenarios, C may want to re-trieve a batch of elements from S at once [24], [35], [32], [42], which motivates *multi-query* PIR [4], [43]. Most of the multi-query PIR protocols use *batch codes* [30] to encode a collection of n elements into M codewords distributed among B buckets; any L of the n elements can be recov-ered by fetching at most one codeword from each bucket. Therefore, C could use a separate single-query PIR to query each bucket; to answer these B queries, S computes over $M < L \cdot n$ codewords exactly once, which significantly reduces the computational overhead. However, given that $B > L$, this may introduce more communication overhead than running L instances of single-query PIR (cf. Figure 12 in [4]). Therefore, existing multi-query PIR protocols offer an unattractive trade-off: they reduce computation but add communication overhead.

Labeled PSI. *Labeled private set intersection* (LPSI) can be considered as multi-query PIR with a stronger security guar-antee: it protects both C's queries and S's database. More specifically, after the protocol, C only learns the elements whose identifiers matches its queries, but nothing else about other elements. Furthermore, the identifiers are keywords instead of indices. To the best of our knowledge, all existing LPSI protocols [13], [18] have S homomorphically evaluate a large degree polynomial that interpolates the payloads. Such protocols require several hours to setup due to the expensive polynomial interpolations. This expensive setup phase makes such protocols undesirable for scenarios where

the database updates frequently.

Our contribution. In this paper, we propose a *novel* PIR protocol named PIRANA, which improves CwPIR [37] to support more elements. Our starting point is to replace the homomorphic equality operator in CwPIR with a SIMD-based one, which supports batch processing: it can perform N (the number of slots in an SIMD ciphertext, typically $N = 4\,096$ or $8\,192$) equality checks with a single ciphertext-ciphertext multiplication. As a result, S in PIRANA only needs to run ciphertext-ciphertext multiplication for $(k-1)\lceil n/N \rceil$ times, instead of $(k-1)n$.

Most importantly, PIRANA naturally supports multi-query. It allows C to retrieve a batch of elements from S with a very small extra-cost in both communication and computation, compared to retrieving a single element. In particular, if we use 3-way cuckoo hashing as the batch code, we are able to retrieve upto $\lfloor N/1.5 \rfloor$ elements with only $3\times$ higher computational cost and almost the same communication cost compared to running the single-query PIRANA once.

Furthermore, PIRANA can be easily extended to support labeled PSI. To protect S's elements, we borrow the idea of using oblivious PRF (OPRF) from [13]: S and C apply OPRF to their identifiers; S uses the OPRF values to mask the corresponding payloads; C retrieves the masked payloads using multi-query PIRANA; C can get a correct payload only if the corresponding query keyword is in the intersection. Compared with the state-of-the-art LPSI protocols, we successfully get rid of the expensive polynomial interpolations, hence we have less expensive setup phase.

We summarize our contribution as follows:

- We design a *novel* constant-weight PIR named PIRANA. It is upto $188.6\times$ faster than CwPIR [37]. (Section 3)

- We extend PIRANA to support multi-query. To retrieve $2\,730$ elements, the amortized cost for retrieving a single element is only 3.9ms in runtime and 3.0KB in bandwidth. (Section 4)

- We further extend PIRANA to support labeled PSI. It is more friendly to the scenarios where the database updates frequently. (Section 5)

- We provide a full-fledged implementation and extensive evaluation. (Section 6)

2. Preliminaries

In this section, we describe some building blocks that will be used in our protocols. A summary of notations is shown in Table 1.

2.1. Homomorphic encryption

Fully Homomorphic Encryption (FHE) is an encryption scheme that allows arbitrary operations to be performed

Notation	Description
C	client
S	server
n	DB size
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
$\mathbf{x}$	a vector
$\widetilde{\mathbf{x}}$	an SIMD ciphertext
t	$\lceil n/N \rceil$
$\%$	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
s	$\lfloor N/B \rfloor$

TABLE 1: Summary of frequent notations.

over encrypted data [27]. In practice, it is usually used in a leveled fashion: the operations can only be performed for a limited times, o.w., the ciphertexts cannot be decrypted. In most FHE cryptosystems [9], [7], [25], [15], [8], 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, as well as how many times the operations can be performed.

Most FHE cryptosystems support the *single instruction multiple data* (SIMD) technique, which allows one to encrypt a vector of elements into a single ciphertext and process 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 $\mathbb{Z}_p^n$. Furthermore, an SIMD ciphertext 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 and CwPIR

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 $CW(m, k)$. Algorithm 1 shows how Mahdavi-Kerschbaum [37] 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 $\mathbf{x} \in CW(m, k)$
1: $j := i$
2: $l := k$
3: $\mathbf{x} := 0^m$
4: **for** $m' := m - 1, ..., 0$ **do**
5:      **if** $j \geq \binom{m'}{l}$ **then**
6:          $\mathbf{x}[m'] := 1$
7:          $j := j - \binom{m'}{l}$
8:          $l := l - 1$
9:      **end if**
10:      **if** $l = 0$ **then**
11:          break
12:      **end if**
13: **end for**
14: return $\mathbf{x}$

Suppose 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 $\forall x, y \in D$,*

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

An equality operator over constant-weight codewords could be:

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

Mahdavi-Kerschbaum [37] propose a *homomorphic equality operator* for constant-weight code based on SealPIR's oblivious expansion [4]. 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 and m ciphertext-plaintext multiplications [4]; and the equality operator requires $(k-1)$ ciphertext-ciphertext multiplications.

Mahdavi-Kerschbaum also propose a constant-weight PIR named CwPIR [37] based on this homomorphic equality operator. In a nutshell, it works as follows:

1) C maps its query to a constant-weight codeword $\mathbf{x}$;
2) S maps each of its identifiers to a constant-weight codeword $\mathbf{y}_i$;
3) C homomorphically encrypts the indices that correspond to 1s in $\mathbf{x}$, and sends the ciphertexts to S;
4) S obliviously expands them into m ciphertexts, which correspond to the m bits of $\mathbf{x}$;
5) For each $\mathbf{y}_i$, S runs the homomorphic equality operator between $\mathbf{x}$ and $\mathbf{y}_i$; it results in a selection vector of length n;
6) S returns the inner product between the selection vector and the payloads.

Notice that the payloads get involved only when computing the inner product in the last step, and that step only requires ciphertext-plaintext multiplications. In contrast, the mainstream PIR protocols [4], [41], [39], [3] represent the database as a D-dimensional hypercube; they need to compute inner products (involving the payloads) for each dimension, and require ciphertext-ciphertext multiplications from the second dimension and on (cf. Section 7). Therefore, CwPIR is more friendly to databases with large payloads. However, the selection vector generation (Step 5) in CwPIR requires $(k-1)n$ ciphertext-ciphertext multiplications, which is disastrous when n is large.

2.3. Batch code

A *batch code* [30] $(n, M, L, B)\text{-}BC$ 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. [4] 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 (probabilistic) batch code as follows:

- $[\mathbf{C}_1, ..., \mathbf{C}_B] \leftarrow \mathsf{Encode}([pl_1, ..., pl_n])$, where $\mathbf{C}_i$ denotes vectors of codewords in the i-th bucket.

- $[i'_1, ..., i'_B] \leftarrow \mathsf{GenSchedule}([i^*_1, ..., i^*_L])$, which takes a set of L queries and outputs a query for each of the B buckets.

- $\left[pl_{i^*_1}, ..., pl_{i^*_L}\right] \leftarrow \mathsf{Decode}([C_1[i'_1], ..., C_B[i'_B]])$, which takes B codewords and outputs L payloads.

3. Single-query PIRANA

In this section, we show how we design a faster constant-weight PIR.

3.1. Single-query PIRANA for small payloads

We first consider the case that the payload size is smaller than the slot size, $|pl| \leq p$.

Intuition. We arrange the n elements of the database into a matrix of N rows and $t := \lceil n/N \rceil$ columns (for simplicity, we assume N divides n), s.t., the i-th element is in row $r := i\%N$ and column $c := \lceil i/N \rceil$.

Naively, we can have C send t SIMD ciphertexts, where only the r-th slot of the c-th ciphertext is 1 and all other slots are 0s; S simply multiplies each column to each ciphertext and adds the products together. Then, the payload pl_{i^*} that C wants to retrieve is in the r-th slot of the resulting ciphertext. However, in this way, C needs to send t ciphertexts to S, hence we want to reduce the number of ciphertexts to be sent.

To this end, we encode the column $c \in [1, ..., t]$ as a constant-weight codeword $\mathbf{x} \in CW(m, k)$. Similarly, we encode the column indices $[1, ..., t]$ as t constant-weight codewords $[\mathbf{y}_1, ..., \mathbf{y}_t]$. Then, C only needs to send m SIMD ciphertexts: the r-th slot of each ciphertext corresponds to each bit of $\mathbf{x}$, and other slots are 0s. After running the equality operators for $[\mathbf{y}_1, ..., \mathbf{y}_t]$, S will get the same t SIMD ciphertexts as in the aforementioned naive solution. That is to say, we can achieve the same goal with the naive solution by only sending $m \in O(\sqrt[k]{k!t} + k)$ ciphertexts. Figure 1 visualizes this process.

The protocol. The single-query PIRANA (for small payloads) works as follows:

- **Setup.** Parameters for the homomorphic encryption are chosen and keys are generated. S arranges its database into a matrix of N rows and $t := n/N$ columns (for simplicity, we assume N divides n).
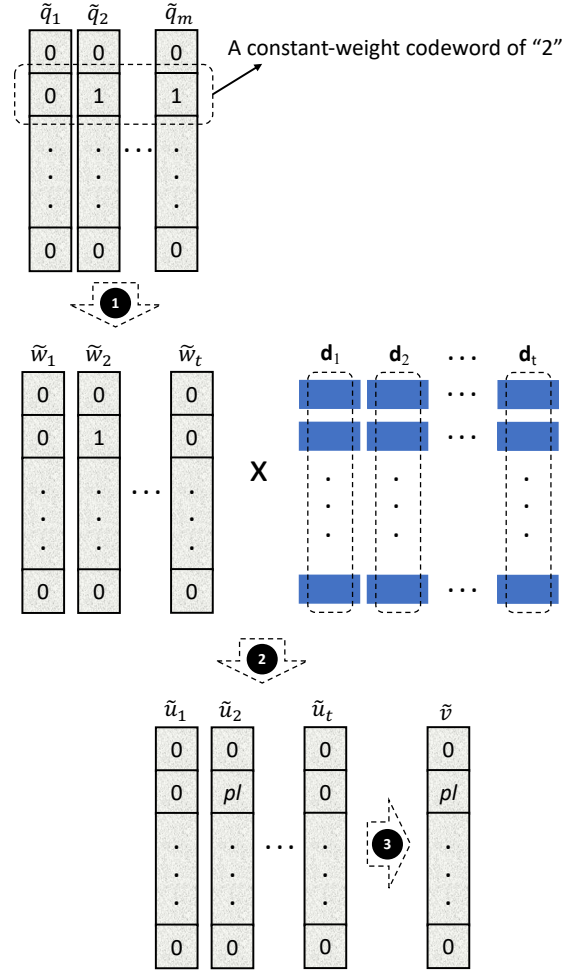

Figure 1: Workflow of single-query PIRANA for small payloads (suppose the element C wants to retrieve is in the second row and second column).

- **Query.** Algorithm 2 shows how C generates a query for retrieving the i^* payload. It first maps i^* to the row index r and column index c of the matrix (Line 2-3). Then, it maps $c \in [1, ..., t]$ to a constant-weight codeword $\mathbf{x} \in CW(m, k)$ (Line 4) and generates the m SIMD ciphertexts based on $\mathbf{x}$ and r (Line 6-15).

- **Answer.** S runs Algorithm 3 to answer a query. For each column, S maps the column index $j \in [1, ..., t]$ to a constant-weight codeword $\mathbf{y}_j \in CW(m, k)$ (Line 3); runs the equality operator between $\mathbf{y}_j$ and $\mathbf{x}$, resulting in a *selection vector* $\widetilde{\mathbf{w}}_j$ (Line 4-5); and multiplies the j-th column of the database to $\widetilde{\mathbf{w}}_j$ (Line 6). Then, it adds the products of all columns together (Line 8) and returns the *inner product* $\widetilde{\mathbf{v}}$ to C. Notice that $\mathbf{v}[r] = pl_{i^*}$ and other elements of $\mathbf{v}$ are 0s.

- **Extract.** C decrypts $\widetilde{\mathbf{v}}$ and outputs its r-th slot.

In this way, S needs to run $\lceil n/N \rceil$ ciphertext-plaintext

Algorithm 2 Single-query PIRANA: Query

Input i^*
Output $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m]$
1: $t := n/N$ $\quad \triangleright$ For simplicity, we assume N divides n
2: $r := i^* \% N$
3: $c := \lceil i^*/N \rceil$
4: $\mathbf{x} \in CW(m, k) \leftarrow$ run Algorithm 1 with $c \in [1, ..., t]$
5: Find the k positions $[i_1, ..., i_k]$ in $\mathbf{x}$, where $\mathbf{x}[i] = 1$
6: **for** $j := 1, ..., m$ **do**
7: $\quad$ **for** $h := 1, ..., N$ **do**
8: $\quad\quad$ **if** $j \in [i_1, ..., i_k]$ and $h = r$ **then**
9: $\quad\quad\quad \mathbf{q}_j[h] := 1$
10: $\quad\quad$ **else**
11: $\quad\quad\quad \mathbf{q}_j[h] := 0$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad \widetilde{\mathbf{q}}_j \leftarrow \mathsf{SIMDEnc}(\mathbf{q}_j)$
15: **end for**

Algorithm 3 Single-query PIRANA: Answer (small payloads)

Input $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m]$, $[pl_1, ..., pl_n]$ $\quad \triangleright |pl| \le p$
Output $\widetilde{\mathbf{v}}$: $\mathbf{v}[i^* \% N] = pl_{i^*}$ and other elements of $\mathbf{v}$ are 0s
1: Init $[\mathbf{d}_1, ..., \mathbf{d}_t]$: each $\mathbf{d}_j$ is $\left[pl_{(j-1)N+1}, ..., pl_{jN} \right]$
2: **for** $j := 1, ..., t$ **do**
3: $\quad \mathbf{y}_j \in CW(m, k) \leftarrow$ run Algorithm 1 with $j \in [t]$
4: $\quad$ Find the k positions $[i_1, ..., i_k]$ in $\mathbf{y}_j$, where $\mathbf{y}_j[i] = 1$
5: $\quad \widetilde{\mathbf{w}}_j \leftarrow \mathsf{SIMDMul}(\widetilde{\mathbf{q}}_{i_1}, ..., \widetilde{\mathbf{q}}_{i_k})$
6: $\quad \widetilde{\mathbf{u}}_j \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{w}}_j, \mathbf{d}_j)$
7: **end for**
8: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{u}}_1, ..., \widetilde{\mathbf{u}}_t)$
9: return $\widetilde{\mathbf{v}}$

Algorithm 4 Single-query PIRANA: Answer (large payloads)

Input $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m]$, $[pl_1, ..., pl_n]$ $\quad \triangleright |pl| = l \cdot N \cdot p$
Output $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_l]$: $\mathbf{v}_1[1]||...||\mathbf{v}_l[N] = pl_{i^*}$
1: Split each pl into $pl^1||...||pl^{l \cdot N}$
2: Init $\begin{bmatrix} \mathbf{d}_{1,1} & ... & \mathbf{d}_{1,l \cdot N} \\ \vdots & \ddots & \vdots \\ \mathbf{d}_{t,1} & ... & \mathbf{d}_{t,l \cdot N} \end{bmatrix}$: $\mathbf{d}_{j,i}$ is $\left[pl^i_{(j-1)N+1}, ..., pl^i_{jN} \right]$
3: Init $[\widetilde{\mathbf{u}}_1, ..., \widetilde{\mathbf{u}}_{l \cdot N}]$: each $\widetilde{\mathbf{u}}_i$ is an encryption of 0^N
4: **for** $j := 1, .., t$ **do**
5: $\quad \widetilde{\mathbf{w}}_j \leftarrow$ Run Line 3-5 of Algorithm 3
6: $\quad$ **for** $i := 1, ..., l \cdot N$ **do**
7: $\quad\quad \widetilde{\mathbf{f}}_i \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{w}}_j, \mathbf{d}_{j,i})$
8: $\quad\quad \widetilde{\mathbf{u}}_i \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{f}}_i, \widetilde{\mathbf{u}}_i)$
9: $\quad$ **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: $\quad$ **for** $j := 1, ..., N$ **do**
14: $\quad\quad \widetilde{\mathbf{v}}_i \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{v}}_i, \widetilde{\mathbf{u}}_{(i-1) \cdot N+j})$
15: $\quad\quad \widetilde{\mathbf{v}}_i \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{v}}_i, 1)$
16: $\quad$ **end for**
17: **end for**
18: return $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_l]$

---

multiplications and $\lceil n/N \rceil \cdot (k-1)$ ciphertext-ciphertext multiplications (recall that CwPIR [37] needs to run $n \cdot (k-1)$ ciphertext-ciphertext multiplications, which is its main overhead).

3.2. Single-query PIRANA for large payloads

Notice that the returned ciphertext $\widetilde{\mathbf{v}}$ in Algorithm 3 has only one non-empty slot. We could make full use of other empty slots to return a large payload via a *rotate-and-sum* approach. Suppose a payload is as large as l ciphertexts, i.e., $|pl| = l \cdot N \cdot p$. We could split each payload into $l \cdot N$ small blocks and compute $l \cdot N$ ciphertexts: the combination of their non-empty slots is the payload to be retrieved. Then, for every N ciphertexts, S could rotate them so that their non-empty slots are interlaced, and add them together. As a result, S only needs to return l ciphertexts. Algorithm 4 shows the details and Figure 2 visualizes this process. Specifically, S runs as follows to answer a query:

❶ For each column, it runs Line 3-5 of Algorithm 3 to

get a selection vector (Line 5).

❷ It multiplies the blocks of the N payloads in each column 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 columns (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). There will be l inner products left for the whole database. An important optimization here is that we rotate $\widetilde{\mathbf{v}}$ instead of $\widetilde{\mathbf{u}}$,[1] enabling us to rotate one slot per rotation. Notice that the rotation becomes much faster when the number of rotated slots is a power of two. Therefore, rotating one $(1 = 2^0)$ slot here is more efficient than rotating an arbitrary number of slots.

3.3. Single-query PIRANA for large payloads and small n

Notice that Algorithm 4 requires $l \cdot (N-1)$ rotations. In this subsection, we first show Algorithm 5, which achieves the same goal with Algorithm 4 but requires $t \cdot (N-1)$ rotations, hence it is more friendly to databases with large payloads but a small n. Then, we show how can we find a balance between Algorithm 4 and Algorithm 5.

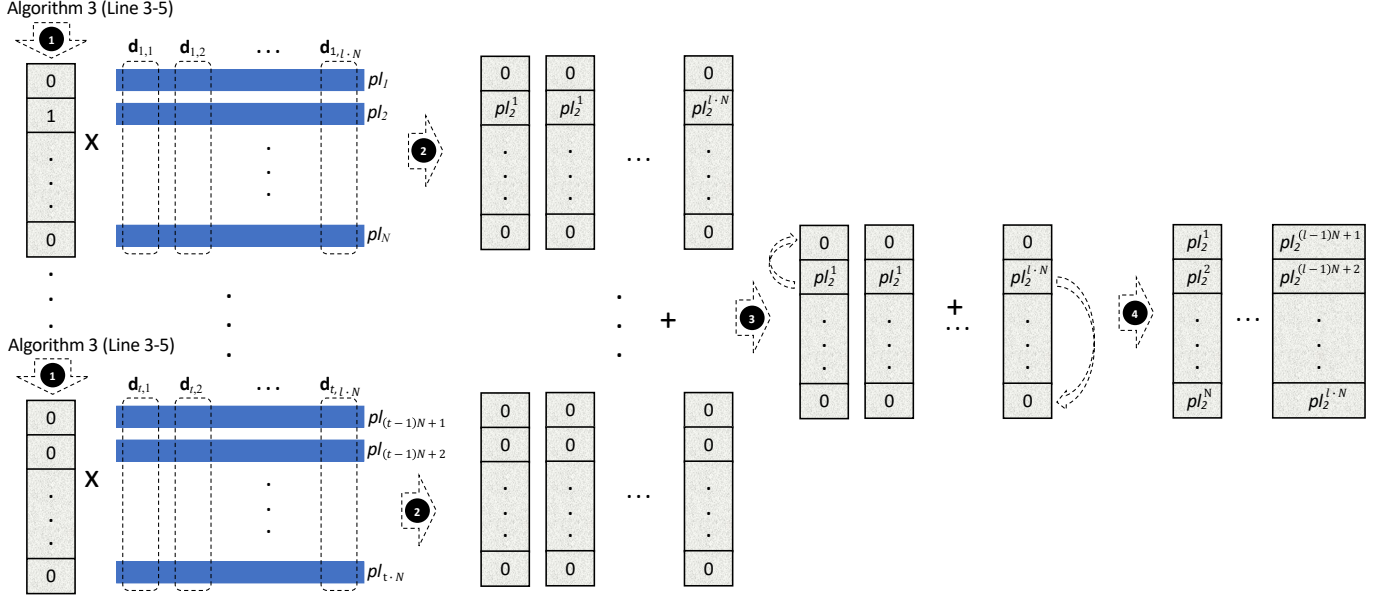1. We rotate $\widetilde{\mathbf{u}}$ in Figure 2 because it is easy to visualize.

Figure 2: Workflow of single-query PIRANA for large payloads (visualization of Algorithm 4).

The intuition is to pre-compute all N possible rotations for each selection vector, then it no longer needs any rotation when computing the inner product. 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 and should be in the setup phase (we simply leave it in Algorithm 5). After running Line 3-5 of Algorithm 3, it rotates each of the t 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 $t \cdot N$ products together, resulting in l inner products (Line 14).

In practice, we could combine Algorithm 4 and Algorithm 5 to minimize the number of rotations for a specific l and t. Specifically, we could first run Algorithm 5, with two modifications:

- In Line 7-9, 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 $\widetilde{\mathbf{f}}$s together, instead it only adds the $\widetilde{\mathbf{f}}$s from t columns together (in a way like in Line 8 of Algorithm 4), resulting in $l \cdot N$ ciphertexts.

Then, it runs Algorithm 4 from Line 11, but only rotates for $(\frac{N}{\alpha} - 1)$ times in Line 13-16. Now, the total number of rotations becomes:

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

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

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

Algorithm 5 Single-query PIRANA: Answer (large payloads and small n)

Input $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m], [pl_1, ..., pl_n]$ ▷ $|pl| = l \cdot N \cdot p$
Output $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_l] : \mathbf{v}_1[1]||...||\mathbf{v}_l[N] = pl_{i^*}$
1: Split each pl into $pl^1||...||pl^{l \cdot N}$
2: Init $\begin{bmatrix} \mathbf{d}_{1,1} & ... & \mathbf{d}_{1,l \cdot N} \\ \vdots & \ddots & \vdots \\ \mathbf{d}_{t,1} & ... & \mathbf{d}_{t,l \cdot N} \end{bmatrix}$: $\mathbf{d}_{j,i}$ is $\left[pl^i_{(j-1)N+1}, ..., pl^i_{jN} \right]$
   with an rotation of i
3: Init $\begin{bmatrix} \widetilde{\mathbf{u}}_{1,1} & ... & \widetilde{\mathbf{u}}_{1,N} \\ \vdots & \ddots & \vdots \\ \widetilde{\mathbf{u}}_{t,1} & ... & \widetilde{\mathbf{u}}_{t,N} \end{bmatrix}$: $\mathbf{u}_{j,i}$ 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** $j := 1, .., t$ **do**
6:    $\widetilde{\mathbf{w}}_j \leftarrow$ Run Line 3-5 of Algorithm 3
7:    **for** $i := 1, ..., N-1$ **do**
8:       $\widetilde{\mathbf{u}}_{j,i} \leftarrow$ SIMDRotate$(\widetilde{\mathbf{w}}_j, i)$
9:    **end for**
10:   **for** $h := 1, ..., l$ **do**
11:     **for** $i := 1, ..., N$ **do**
12:       $\widetilde{\mathbf{f}}_i \leftarrow$ SIMDPmul$(\widetilde{\mathbf{u}}_{j,i}, \mathbf{d}_{j,(h-1)N+i})$
13:     **end for**
14:     $\widetilde{\mathbf{v}}_h \leftarrow$ SIMDAdd$(\widetilde{\mathbf{v}}_h, \widetilde{\mathbf{f}}_1, ..., \widetilde{\mathbf{f}}_N)$
15:   **end for**
16: **end for**
17: **return** $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_l]$

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

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

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

1) If the first-order derivative is always negative (i.e. $t - \frac{N \cdot l}{\alpha^2} < 0$), $f(\alpha)$ is monotonically decreasing, hence it is minimal when α is maximal (i.e, $\alpha = N$). In this case, $t < \frac{l}{N}$.

2) If the first-order derivative is always positive (i.e. $t - \frac{N \cdot l}{\alpha^2} > 0$), $f(\alpha)$ is monotonically increasing, hence it is minimal when α is minimal (i.e, $\alpha = 1$). In this case, $t > l \cdot N$.

3) If the first-order derivative is zero (i.e. $t - \frac{N \cdot l}{\alpha^2} = 0$), $f(\alpha)$ is minimal when $\alpha = \sqrt{Nl/t}$.

Formally, we have:

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

which implies that:

- when the payload size is large enough s.t., $l \ge tN$, we should run Algorithm 5;
- when the number of elements is large enough s.t., $t \ge lN$, we should run Algorithm 4;
- in the middle ground case, we should combine Algorithm 4 and Algorithm 5 as aforementioned, with $\alpha = \sqrt{Nl/t}$.

Remark. To rotate a ciphertext for a specific number of slots, S needs to know the corresponding rotation key. Therefore, C needs to transfer the rotation keys to S. This is not a problem for either Algorithm 4 or Algorithm 5, because both Algorithms only require S to rotate one slot per rotation, hence C only needs to transfer one rotation key. If we combine them, C has to transfer one more rotation key for rotating $\frac{N}{\alpha}$ slots.

4. Multi-query **PIRANA**

In this section, we show how we extend PIRANA to support multi-query with a very small extra-cost in both communication and computation, compared to retrieving a single element.

4.1. Multi-query **PIRANA** for small payloads

Intuition. Recall that in our single-query PIRANA, C sends m SIMD ciphertexts to S; only one slot in each ciphertext is useful and other slots are empty. Then, a natural question to ask is that can we use other slots to batch more queries? The answer is no because there may be multiple desired elements reside on the same row, rendering the query invalid. Thanks to the batch code, we could encode S's database into a $(n, M, L, B) - BC$ and treat each bucket as a "row"; then, the desired elements are for sure in different rows. If $N > B$, we could split a bucket into multiple rows to reduce the bucket size and make full use of the slots.

If we use 3-way cuckoo hashing as the batch code (cf. Section 2.3), we are able to retrieve upto $\lfloor N/1.5 \rfloor$ elements

with only $3\times$ higher computational cost and almost the same communication cost compared to running the single-query PIRANA once (for small payloads with $|pl| \le p$). In more detail, recall that 3-way cuckoo hashing encodes n elements into $M = 3n$ codewords distributed among $B = 1.5L$ buckets; if we set $B = N$, we could maximize the number queries i.e. $L = N/1.5$ and minimize the bucket size i.e. $3n/N$. The number of columns determines both computational and communication costs, and it increases from n/N (in single-query PIRANA) to $3n/N$. Therefore, the computational cost increases exactly 3 times; the communication cost increases from ($\sqrt[k]{k!n/N} + k$) to ($\sqrt[k]{k!3n/N} + k$), which is insignificant.

The protocol. The multi-query PIRANA (for small payloads) works as follows:

Algorithm 6 Multi-query PIRANA: Query

Input $[i_1^*, ..., i_L^*]$
Output $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m]$
1: $[i_1', ..., i_N'] \leftarrow \mathsf{GenSchedule}([i_1^*, ..., i_L^*])$
2: **for** $h := 1, ..., N$ **do**
3:      $\mathbf{x}_h \in CW(m, k) \leftarrow$ run Algorithm 1 with $i_h' \in [M/N]$ ▷ For simplicity, we assume N divides M
4:      Find k positions $[i_1, ..., i_k]$ in $\mathbf{x}_h$, where $\mathbf{x}_h[i] = 1$
5:      **for** $j := 1, ..., m$ **do**
6:          **if** $j \in [i_1, ..., i_k]$ **then**
7:              $\mathbf{q}_j[h] := 1$
8:          **else**
9:              $\mathbf{q}_j[h] := 0$
10:          **end if**
11:      **end for**
12: **end for**
13: **for** $j := 1, ..., m$ **do**
14:      $\widetilde{\mathbf{q}}_j \leftarrow \mathsf{SIMDEnc}(\mathbf{q}_j)$
15: **end for**

- **Setup.** Parameters for the homomorphic encryption are chosen and keys are generated. S encodes its database using a batch code $BC(n, M, L, B)$:
$$[\mathbf{C}_1, ..., \mathbf{C}_B] \leftarrow \mathsf{Encode}([pl_1, ..., pl_n]),$$
where $\mathbf{C}_i$ denotes vectors of codewords in the i-th bucket. If $B < N$, it splits each bucket into $s := N/B$ small buckets (for simplicity, we assume B divides N). That means the database is in fact encoded as:
$$[\mathbf{C}_1, ..., \mathbf{C}_N] \leftarrow \mathsf{Encode}([pl_1, ..., pl_n]),$$
- **Query.** C runs Algorithm 6 to generate the query ciphertexts. More specifically, given L original indices $[i_1^*, ..., i_L^*]$, C first runs GenSchedule of the batch code to generate indices $[i_1', ..., i_B']$ for each of the B buckets. If the buckets has been split into N small buckets, the corresponding indices $[i_1', ..., i_N']$ should be calculated (Line 1). Then, it runs in a similar way as in Algorithm 2, except that it needs to compute a constant-weight codeword $\mathbf{x}_h$ for each of the N slots (Line 4) and uses $\mathbf{x}_h$ to determine the value for that slot.
- **Answer.** S runs Algorithm 7 to answer these L queries in a batch. It is mostly the same as Algorithm 3, except

that $\mathbf{d}_j$ is a combination of the j-th elements in each bucket (Line 2).

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

$$[pl_{i^*_1}, ..., pl_{i^*_L}] \leftarrow \mathsf{Decode}([C_1[i'_1], ..., C_B[i'_N]]).$$

Algorithm 7 Multi-query PIRANA: Answer (small payloads)

Input $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m], [\mathbf{C}_1, ..., \mathbf{C}_N]$
Output $\widetilde{\mathbf{v}} : \mathbf{v}[j] = \mathbf{C}_j[i'_j]$
1: $t := M/N$ ▷ For simplicity, we assume N divides M
2: Init $[\mathbf{d}_1, ..., \mathbf{d}_t]$: each $\mathbf{d}_j$ is $[\mathbf{C}_1[j], ..., \mathbf{C}_N[j]]$
3: **for** $j := 1, ..., t$ **do**
4:     $\mathbf{y}_j \in CW(m, k) \leftarrow$ run Algorithm 1 with $j \in [t]$
5:     Find the k positions $[i_1, ..., i_k]$ in $\mathbf{y}_j$, where $\mathbf{y}_j[i] = 1$
6:     $\widetilde{\mathbf{w}}_j \leftarrow \mathsf{SIMDMul}(\widetilde{\mathbf{q}}_{i_1}, ..., \widetilde{\mathbf{q}}_{i_k})$
7:     $\widetilde{\mathbf{u}}_j \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{w}}_j, \mathbf{d}_j)$
8: **end for**
9: $\widetilde{\mathbf{v}} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{u}}_1, ..., \widetilde{\mathbf{u}}_t)$
10: return $\widetilde{\mathbf{v}}$

4.2. Multi-query PIRANA for large payloads

Notice that only B out of N slots in $\widetilde{\mathbf{v}}$ (the returned ciphertext of Algorithm 7) are non-zero and other slots are empty. When the payloads are large (i.e., $|pl| = l \cdot N \cdot p$) and B is small, it is desirable to combine multiple ciphertexts into a single one in a similar way as in Algorithm 4. However, this time, there are B (instead of one) non-zero values in $\widetilde{\mathbf{v}}$, hence we cannot directly rotate-and-sum. Fortunately, the positions for the non-zero values are somehow regular: every N/B slots have a single non-zero value. Therefore, we could extract each non-zero value into a separate SIMD ciphertext. For example, to extract the first non-zero value, we could multiply the ciphertext with a bit vector, where the first N/B values are 1s and others are 0s. Then, we could run as in Algorithm 4 separately for the $l \cdot N$ ciphertexts for each bucket, resulting in $l \cdot B$ ciphertexts. Algorithm 8 shows the details.

For each column, S runs in a similar way as in Algorithm 7 to generate a selection vector (Line 9-11). Then, it multiplies the blocks of the N payloads in each column to each selection vector, and repeats this for all $l \cdot N$ blocks of each payload (Line 12-14). Next, it runs as in Algorithm 4 to combine every N ciphertexts (Line 19-25) and this process will be repeated for each bucket. In this way, it requires $B \cdot l \cdot N$ rotations in total. However, we could rotate $\widetilde{\mathbf{u}}$s first to enumerate all rotations, and then extract the non-zero value in each bucket. This optimization reduces the of rotations to $l \cdot N$. The cost is that C has to transfer N rotations keys, but this only needs to be done once during setup. Algorithm 9 shows this optimization. Notice that the vector $\mathbf{e}$ needs to be rotated in corresponding to $\widetilde{\mathbf{u}}$ (Line 8).

Algorithm 8 Multi-query PIRANA: Answer (large payloads)

Input $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m], [\mathbf{C}_1, ..., \mathbf{C}_N]$ ▷ $|pl| = l \cdot N \cdot p$
Output $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_d]$
1: $t := M/N$ ▷ we assume N divides M
2: $s := N/B$ ▷ we assume B divides N
3: Split each $\mathbf{C}[j]$ into $\mathbf{C}[j][1]||...||\mathbf{C}[j][l \cdot N]$
4: Init $\begin{bmatrix} \mathbf{d}_{1,1} & ... & \mathbf{d}_{1,l \cdot N} \\ \vdots & \ddots & \vdots \\ \mathbf{d}_{t,1} & ... & \mathbf{d}_{t,l \cdot N} \end{bmatrix}$: $\mathbf{d}_{j,h}$ is
    $[\mathbf{C}_1[j][h], ..., \mathbf{C}_N[j][h]]$
5: **for** $h := 1, ..., l \cdot N$ **do**
6:     Init $\widetilde{\mathbf{u}}_h$: $\mathbf{u}_h = 0^N$
7: **end for**
8: **for** $j := 1, ..., t$ **do**
9:     $\mathbf{y}_j \in CW(m, k) \leftarrow$ run Algorithm 1 with $j \in [t]$
10:     Find the k positions $[i_1, ..., i_k]$ in $\mathbf{y}_j$, where $\mathbf{y}_j[i] = 1$
11:     $\widetilde{\mathbf{w}}_j \leftarrow \mathsf{SIMDMul}(\widetilde{\mathbf{q}}_{i_1}, ..., \widetilde{\mathbf{q}}_{i_k})$
12:     **for** $h := 1, ..., l \cdot N$ **do**
13:         $\widetilde{\mathbf{u}}_h := \mathsf{SIMDAdd}(\widetilde{\mathbf{u}}_h, \mathsf{SIMDPmul}(\widetilde{\mathbf{w}}_j, \mathbf{d}_{j,h}))$
14:     **end for**
15: **end for**
16: Init $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_{l \cdot B}]$: each $\mathbf{v}_i$ is an encryption of 0^N
17: **for** $h := 1, ..., B$ **do**
18:     Init $\mathbf{e} : \mathbf{e}[(h-1) \cdot s, ..., h \cdot s]=1$ and other slots are 0s
19:     **for** $i := 1, ..., l$ **do**
20:         **for** $j := 1, ..., N$ **do**
21:             $\mathbf{a} \leftarrow \mathsf{SIMDPmul}(\widetilde{\mathbf{u}}_{(i-1) \cdot N+j}, \mathbf{e})$
22:             $\widetilde{\mathbf{v}}_{(h-1)l+i} \leftarrow \mathsf{SIMDAdd}(\widetilde{\mathbf{v}}_{(h-1)l+i}, \mathbf{a})$
23:             $\widetilde{\mathbf{v}}_{(h-1)l+i} \leftarrow \mathsf{SIMDRotate}(\widetilde{\mathbf{v}}_{(h-1)l+i}, 1)$
24:         **end for**
25:     **end for**
26: **end for**
27: return $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_{l \cdot B}]$

4.3. Further optimizations

To be complete, we describe some optimizations used in our protocol. We remark that these optimizations were proposed by prior work.

Using secret-key homomorphic encryption to reduce query size. Notice that in the scenario of PIR, only C needs to do encryption but no one else. To this end, we could use a secret-key version of homomorphic encryption instead of the public-key version. Given that the ciphertext of homomorphic encryption is a tuple (c_0, c_1) in $\mathbb{Z}_q^2[x]$, if we use the secret-key version, c_0 is sampled uniformly at random in $\mathbb{Z}_q[x]$ (whereas in the public-key version, it depends on the public key). Therefore, instead of sending c_0, C can send the seed that has been used to generates c_0, and S can reconstruct c_0 from the seed. This reduces the query size by a factor $2\times$.

Using modulus switching to reduce answer size. Given

		n	2^{20}
	# elements	n	2^{20}
	payload size	$l \cdot N \cdot p \ (l > 1)$	20KB
	# queries	$L < N$	256
Mughees-Ren [43]	# SIMDPmul	$3l \cdot n$	3 145 728
	# SIMDMul	$\frac{3l \cdot n}{d} + l \cdot B \cdot d$	110 592
	# SIMDRotate	$l \cdot B(d^2 + \frac{3n}{Bd} + d(\log d - 1))$	540 672
	# ciphers (query)	$\frac{Bd}{N}$	2
	# ciphers (answer)	$l \cdot B$	384
multi-PIRANA	# SIMDPmul	$3l \cdot n + \frac{B \cdot l \cdot N}{2}$	4 718 592
	# SIMDMul	$3n/N$	384
	# SIMDRotate	$\frac{l \cdot N}{2}$	4 096
	# ciphers (query)	$\sqrt{6n/N} + 2$	30
	# ciphers (answer)	$l \cdot B$	384

TABLE 2: Comparison between Mughees-Ren [43] and multi-query PIRANA. ($N = 8\ 192$, $p = 20$; the batch code is based on cuckoo hashing, hence $B = 1.5L$; d is a power of two and larger than $(3n/B)^{1/3}$: $d = 32$ for $n = 2^{20}$ and $B = 384$)

Algorithm 9 Multi-query PIRANA: Answer with less rotations (large payloads)

Input $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m]$, $[\mathbf{C}_1, ..., \mathbf{C}_N]$ $\qquad \triangleright |pl| = l \cdot N \cdot p$
Output $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_d]$
1: $s := N/B$ $\qquad\qquad\qquad \triangleright$ we assume B divides N
2: $[\widetilde{\mathbf{u}}_1, ..., \widetilde{\mathbf{u}}_{l \cdot N}] \leftarrow$ Run Line 1-15 of Algorithm 8
3: Init $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_{l \cdot B}]$: each $\mathbf{v}_i$ is an encryption of 0^N
4: **for** $i := 1, ..., l$ **do**
5: $\quad$ **for** $j := 1, ..., N$ **do**
6: $\quad\quad$ $\widetilde{\mathbf{u}}'_{(i-1)N+j} \leftarrow$ SIMDRotate$(\widetilde{\mathbf{u}}_{(i-1)N+j}, j-1)$
7: $\quad\quad$ **for** $h := 1, ..., B$ **do**
8: $\quad\quad\quad$ Init $\mathbf{e} : \mathbf{e}[((h-1) \cdot s + (j-1))\%N, ..., (h \cdot s + (j-1))\%N]$=1 and other slots are 0s
9: $\quad\quad\quad$ $\mathbf{a} \leftarrow$ SIMDPmul$(\widetilde{\mathbf{u}}'_{(i-1) \cdot N+j}, \mathbf{e})$
10: $\quad\quad\quad$ $\widetilde{\mathbf{v}}_{(h-1)l+i} \leftarrow$ SIMDAdd$(\widetilde{\mathbf{v}}_{(h-1)l+i}, \mathbf{a})$
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: **end for**
14: **return** $[\widetilde{\mathbf{v}}_1, ..., \widetilde{\mathbf{v}}_{l \cdot B}]$

---

that the returned ciphertexts will no longer be used for further computations, we could use the *modulus switching* technique [10] to reduce their sizes. Modulus switching is a public operation that can reduce the ciphertext modulus from q to q', without affecting the plaintext. Since the size of a FHE ciphertext is linear in $\log q$, modulus switching reduces it by a factor of $\frac{\log q}{\log q'}$. Therefore, this trick allows S to reduce the response size the the same factor.

More efficient rotations. The rotation operation is in fact not rotating a vector as written in Section 2.1. Instead, it works over a $(N/2) \times 2$ matrix and rotates the columns. This feature benefits Algorithm 9: it allows S to extract two elements with a single rotation, reducing the number of rotations (Line 6) as well as the number of ciphertext-plaintext multiplications (Line 8) by a factor of $2\times$.

4.4. Comparison with Mughees-Ren [43]

A concurrent and independent work was proposed by Mughees-Ren [43] to improve the performance of multi-query PIR. They encode the elements in each bucket (of a batch code) as a 3-dimensional hypercube and batch the queries for different buckets into a single SIMD ciphertext. They further use a rotate-and-sum approach to batch the responses. However, this protocol inherits the drawback from hypercube-based PIRs: most of S's operations need to involve the payloads, which incurs a large computational overhead when the payload size is large. In contrast, the expensive operations (ciphertext-ciphertext multiplications) in PIRANA are independent of the payloads.

Table 2 provides a comparison between Mughees-Ren [43] and multi-query PIRANA. For a large payload size (i.e., $|pl| = l \cdot N \cdot p$ with $l > 1$), PIRANA is basically better than Mughees-Ren's protocol in all aspects except the number of ciphertext-plaintext multiplications. In particular, PIRANA saves $(\frac{3n(l \cdot N - d)}{d \cdot N} + l \cdot B \cdot d)$ ciphertext-plaintext multiplications and $l(B \cdot d^2 + \frac{3n}{d} + B \cdot d(\log d - 1) - \frac{N}{2})$ rotations. Notice that the query size of PIRANA remains the same as long as $N > 1.5L$, whereas the query size of Mughees-Ren increases linearly with L.

The right-most column of Table 2 provides some concrete numbers for a specific database configuration. If we take the experimental computation cost for each homomorphic operation from Mughees-Ren's paper (cf. Table 2 in [43]), we could roughly achieve $8\times$ speedup over Mughees-Ren (444.0s vs. 3 567.7s). The advantage will be more prominent when l is larger.

5. Labeled PSI

Private Set Intersection (PSI) allows two parties, a sender and a receiver, to compute the intersection of their private sets X and Y such that the receiver only learns $X \cap Y$ and the sender learns nothing. Most PSI protocols are *balanced*, i.e., the two sets are of similar size, and the two parties

have similar computation and storage capabilities. In particular, the communication cost of balanced PSI protocols depends on the size of the larger set. On the other hand, *unbalanced* PSI [14], [13], [18] focuses on the case that the receiver's set is much smaller than the sender's and aims to achieve a communication complexity that depends on the size receiver's set.

In some scenarios, the sender holds a label for each item x_i in its set and the receiver wants to learn the labels corresponding to the elements in the intersection. Notice that a labeled and unbalanced PSI (LPSI) protocol can be considered as a multi-query PIR protocol that supports keyword queries and protects S's database. To this end, we denote the sender by S and denote the receiver by C; we use payload instead of label.

At a high level, existing LPSI protocols [13], [18] have C encrypt the elements in Y using FHE and send the ciphertexts to S; for each encrypted y_i, S homomorphically evaluates a polynomial that interpolates the payloads, i.e., $F(x_i) = pl_i\ \forall x_i$, and returns the results to C. In this way, C will get pl_i if $y_i = x_i$. However, naively running in this way will reveal information about the elements that are not in the intersection. To this end, they have S and C apply an OPRF [26] to the elements in X and Y; S uses the OPRF values to mask the corresponding payloads; C retrieves the masked payloads by polynomial evaluation as above and uses its OPRF values to unmask the payloads. In this way, C can only get the elements that are in the intersection. Such protocols require several hours for polynomial interpolations, which makes them undesirable for scenarios where the database updates frequently.

To extend multi-query PIRANA to LPSI (dubbed LPSI-PIRANA), we need to improve it to (1) support keyword queries and (2) protect S's database. Challenge (1) is easy to overcome as the constant-weight code naturally supports keywords. However, mapping keywords instead of indices to constant-weight codewords will increase the codeword bit-length m. An alternative way is to apply cuckoo hashing again to the elements inside each bucket. This time, whenever an element is assigned to a location that is occupied, the old element is evicted and recursively reinserted using a different hash function. In this way, we successfully map keywords to indices. Notice that it is orthogonal to the use of cuckoo hashing to batch multiple PIR queries.

Using cuckoo hashing inside each bucket will expand the bucket for $1.5\times$ and incur more queries. For the latter, when we use 3-way cuckoo hashing, C needs to query three positions for each bucket; whereas in multi-query PIRANA, C only needs to query one position for each bucket. As a result, LPSI-PIRANA is at least $4.5\times$ slower than multi-query PIRANA. Nevertheless, this is still better than existing LPSI protocols. Recall that such protocols need to evaluate a polynomial on the encrypted keyword, which happens in modulo p (20-30 bits), but the (hashed) keyword (80-128 bits) is larger than p. To this end, they splits a (hashed) keyword to occupy several sequential slots, and evaluate polynomials on them separately, which introduces more overhead than our solution.

For challenge (2), we borrow the idea of using OPRF to hide the elements that are not in the intersection. Then, our LPSI-PIRANA works as follows:

- **Setup.**
  1) S computes $(x_i', x_i'') = PRF_k(x_i)$ for all x_i in its set X. The payload pl_i is masked by x_i'': $pl_i' := pl_i \oplus x_i''$.
  2) S uses cuckoo hashing to assign $\{x_1', ..., x_n'\}$ to B buckets.
  3) Inside each buckets, S using cuckoo hashing to map each x to an index and sort the payloads according to their indices.

- **Query.**
  1) For each $y_i \in [y_1, ..., y_L]$, S and C run OPRF so that C gets (y_i', y_i'') without learning the key.
  2) C runs Algorithm 6 with input $[y_1', ..., y_L']$,[2] and sends the output $[\widetilde{\mathbf{q}}_1, ..., \widetilde{\mathbf{q}}_m]$ to S.

- **Answer.** S runs Algorithm 7 (for small payloads) or Algorithm 8 (for large payloads), and returns the results to C.

- **Extract.** C runs the "Extract" of multi-query PIRANA, and unmask the results using y''s.

Compared with the state-of-the-art LPSI protocol [18], we successfully get rid of the expensive polynomial interpolations, and we also save $\lceil M/N \rceil$ ciphertext-plaintext multiplications. Furthermore, recall that when $N > B$, we could use the empty slots to batch more responses (cf. Algorithm 8). This is not the case for [18], because all of their slots are non-empty. Put it in another way, for $x \notin Y$, our protocol results in 0 in the corresponding slot, whereas [18] results in a dummy value.

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 in C++ based on the Microsoft SEAL homomorphic encryption library (version 4.0)[3]. We use the Brakerski-Fan-Vercauteren (BFV) [7], [25] scheme, with $N \in \{4096, 8192\}$ and the default parameters in SEAL for 128-bit security.

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 in the setup phase and can be used for all queries.

We use index queries to evaluate all PIR protocols; and we use keyword queries to evaluate all LPSI protocols. We run all experiments on an Intel Xeon Cooper Lake (with

---

2. For the ease of description, we map keywords to indices using constant-weight code instead of cuckoo hashing.
3. 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}
	DB Size (MB)	5.2	10	21	42	84	170	340	670	1 300
CwPIR [37]	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)	0.001	0.001	0.001	0.001	0.001	0.001	0.027	0.05	0.1
PIRANA	Inner Product (s)	0.22	0.24	0.28	0.36	0.52	0.86	1.57	2.86	5.39
$k = 2$	Total server (s)	0.22	0.24	0.28	0.36	0.52	0.86	1.6	2.9	5.49
$N = 2^{13}$	Speedup	18.6×	34.2×	58.2×	90.6×	125×	151×	162.1×	178.9×	188.6×

TABLE 3: Microbenchmark of PIRANA and CwPIR (payload size is 20KB).

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 [37]. All experiments were repeated 5 times and average values (the variances are very small) were reported.

6.2. Evaluation of single-query PIRANA

We first compare PIRANA (a combination of Algorithm 4 and Algorithm 5 with α depending on the number of elements n and payload sizes) with CwPIR [37] in terms of single-query PIR. To this end, we reproduce the results of CwPIR (i.e., Table 7 in [37]) reported in their original paper, by running their open-sourced implementation[4]. We set ($k = 2$, $N = 8\,192$) for both CwPIR and PIRANA, and choose the minimal m that satisfies $\binom{m}{k} > n$ and $\binom{m}{k} > n/N$ respectively. Our reproduced results of CwPIR 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 comparison results in Table 3. 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. Given our advantages in both selection vector generation and inner product calculation, we achieve upto 188.6× speedup over CwPIR.

On the other hand, the query size of PIRANA is upto 2.5× larger than CwPIR (the response sizes are roughly equal). For example, when $n = 2^{16}$, the query size is 432KB in CwPIR and 1 080KB in PIRANA. Furthermore, C in PIRANA needs to transfer additional 6.2MB rotation keys during setup. However, as we will show next, this communication overhead in PIRANA can be utilized to batch upto $\lfloor N/1.5 \rfloor$ queries for free.

4. https://github.com/RasoulAM/constant-weight-pir

6.3. Evaluation of multi-query PIRANA

We compare multi-query PIRANA with the state-of-the-art[5] multi-query PIR, i.e., SealPIR[6] with PBC [4]. 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 $p = 2^{-40}$. We use PBC as the batch code as well, with the same configuration as multi-SealPIR [4]. To be complete, we also add PBC to the state-of-the-art single-server PIR protocols: Spiral[7] [39] and SimplePIR[8] [1], making them support multi-query. We leave out CwPIR [37] because it is relatively slow.

We use ($n = 2^{20}$, $|pl| = 256$-byte) as the database because most multi-query PIR benchmarks use this configuration. As the payload size is small, we use Algorithm 7 to answer queries, which allows us to set $N = 4\,096$ instead of $8\,192$. Then, the size for each small bucket is $3n/N = 768$. We set $m = 40$ and $k = 2$ to satisfy $\binom{m}{k} > 768$.

Figure 3 shows the comparison results for multi-query PIRs. PIRANA is significantly better than all others in terms of query generation and answer extraction. To generate a query for 1 024 elements, PIRANA only needs 0.13s, 14.4× faster than the best alternative (i.e., SimplePIR). To extract 1 024 elements, PIRANA only needs 38.1ms, 9.5× faster than the best alternative. In terms of answer generation and bandwidth usage, only SimplePIR shows a comparable performance with PIRANA. However, SimplePIR is stateful i.e., clients need to download and maintain a large state. Indeed, during our experiments, we found that C needs to maintain upto 4.8GB state in SimplePIR. This is clearly unfriendly to mobile devices as well as the scenarios where the database updates frequently. On the other hand, C in PIRANA does not need to maintain any state except the encryption keys.

Recall that we could retrieve $\lfloor N/1.5 \rfloor$ elements with the same cost. Specifically, to retrieve 2 730 elements in PIRANA, the amortized cost for retrieving a single element is only 3.9ms runtime and 3.0KB bandwidth.

5. By the time we submitted this paper, the protocol of Mughees-Ren [43] was not accepted and their code was unavailable. Thus, we only provide a theoretical comparison in Section 4.4.
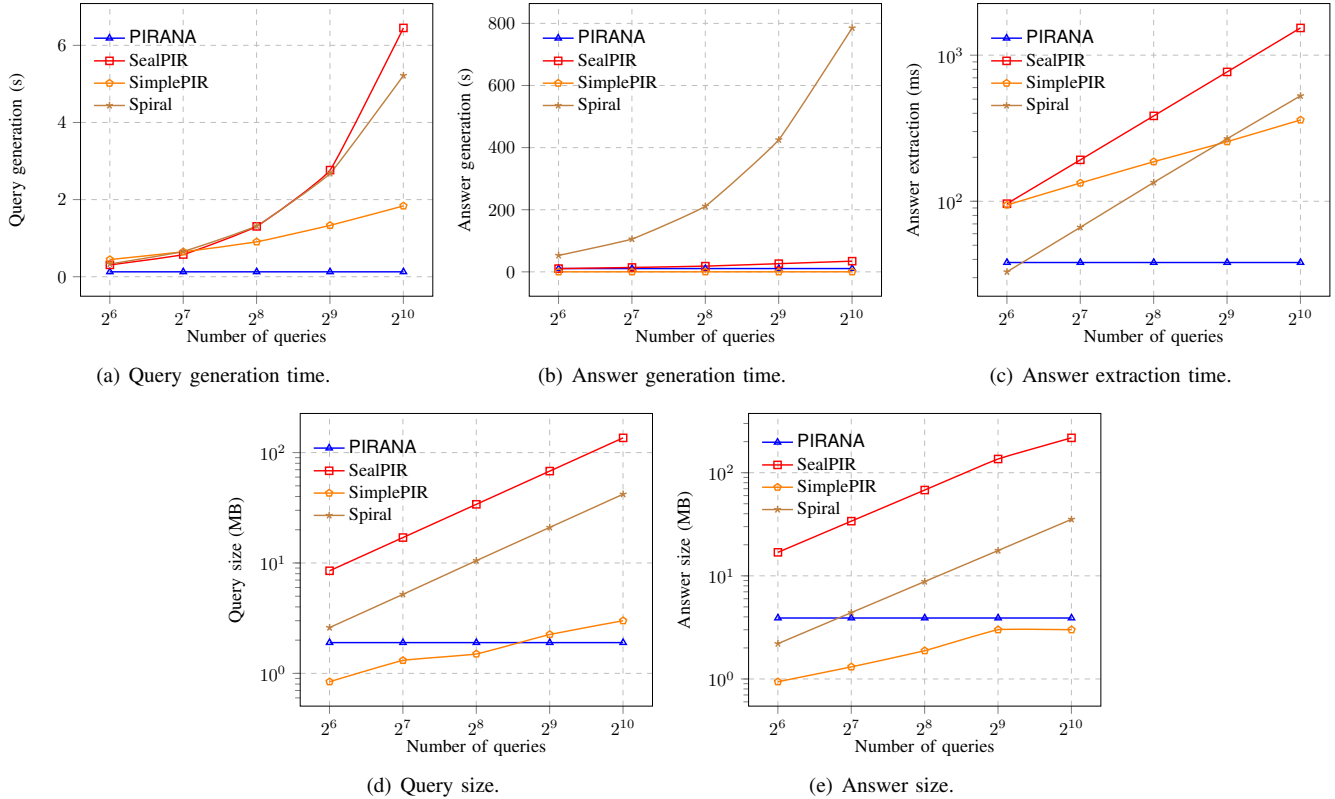6. https://github.com/microsoft/SealPIR
7. https://github.com/menonsamir/spiral
8. https://github.com/ahenzinger/simplepir

(a) Query generation time.

(b) Answer generation time.

(c) Answer extraction time.

(d) Query size.

(e) Answer size.

Figure 3: Evaluation of multi-query PIRs (2^{20} elements, 256-byte payloads).

6.4. Evaluation of LPSI-PIRANA

We evaluate PIRANA in terms of labeled PSI and compare it with the state-of-the-art LPSI[9] [18]. We took the OPRF implementation directly from the source code of [18]: it uses OMGDH-based OPRF, with the Elligator 2 map [6] for the FourQ elliptic curve [21]. Inside each bucket, we apply cuckoo hashing to map keywords to indices as we discussed in Section 5.

The setup phase of [18] is prohibitively slow due to the expensive polynomial interpolation: it takes 1 256.3s even with 32 threads. The most time-consuming part in the setup phase of LPSI-PIRANA is encoding the payloads into NTT forms, which takes only 157s with a single thread. This is clearly insignificant compared to [18]. Therefore, LPSI-PIRANA is more friendly to the scenarios where the database updates frequently.

Cong et al. [18] introduce several algorithmic optimizations for the polynomial evaluation. For example, they split each bucket into subsets and evaluate a polynomial for each subset separately, which reduces the number of ciphertext-ciphertext multiplications at the cost of increasing the number of ciphertexts to be returned. They also use the windowing technique to reduce the multiplicative depth, but it requires C to send more monomials to S. All these opti-

9. https://github.com/microsoft/APSI/

mizations require fine-tuning the parameters to find the best trade-off between computation and communication, which has been done in the source code of [18]. We simply follow their parameters, which leads to lower performance for us: $3.72\times$ slower in runtime and $2.6\times$ larger in bandwidth, for 1 024 queries. We leave it as future work to optimize our performance for LPSI-PIRANA.

7. Related Work

Early single-server PIR. Most of the early single-server PIR protocols follow the blueprint of Kushilevitz and Ostrovsky [33]: 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. [11] instead use the ϕ-Hiding assumption to achieve $O(\log^4 N)$ query size and $O(\log^D N)$ response size. Gentry and Ramazan [28] further reduce the query size of Cachin et al.'s approach to $O(\log^{3-o(1)} N)$. Chang [12] 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 [36] with Damgard-Jurik encryption [22] to achieve $O(\log^2 N)$ query size and $O(\log N)$ response size. As it has been

pointed by Sion and Carbunar [47], such protocols are even slower than trivially having C download the entire database [47].

Expansion-based PIR. Aguilar-Melchor et al. [38] propose XPIR, which applies a lattice-based homomorphic encryption [9] 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 [4] gets rid of this bottleneck by introducing an oblivious 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 again 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 [4] 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 [41] realizes ciphertext-ciphertext multiplication via *external product* [16], which reduces the response sizes but incurs a large computational overhead. Spiral [39] further improves this idea by composing Regev encryption [45] with GSW encryption [29] to achieve a faster external product. MulPIR [3] 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 the benchmarks in [37], MulPIR requires a large server runtime for answer generation.

SIMD-based PIR. FastPIR [2] uses the SIMD technique the batch the one-hot encoding of the index, which is somehow similar to single-query PIRANA. However, compared to FastPIR, our solution is superior in three aspects: (1) we use constant-weight code to encode the column index, which significantly reduces the query size; (2) we propose a customized rotation technique for large payloads; (3) we use the empty slots to batch more queries. Another recent SIMD-based PIR is from Mughees-Ren [43]. We refer to Section 4.4 for a detailed comparison.

PIR with preprocessing. Beimel et al. [5] proved that a se-cure 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. [5] 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. [44] 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 [20], 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 [19], [46], but none of these protocols has an overall efficiency that is better than PIR without preprocessing.

Another line of work in PIR with preprocessing is by Hengzinger et al. [1] and Davidson et al. [23]. The key observation is that in a LWE-based PIR, the bulk of S's computation is independent of the Cs' queries and can be preprocessed. Compared with [20], [19], [46], this preprocessing only needs to be done once and can be used for all queries. However, each C still needs to download and maintain a large state, e.g., 4.8GB in our experiments, clearly unfriendly to mobile devices and the scenarios where the database updates frequently.

8. Conclusion

In this paper, we propose a novel PIR protocol named PIRANA, based on the recent advances in constant-weight codes. It is significantly faster than the original constant-weight PIR. It allows a client to retrieve a batch of elements from the server with a very small extra-cost in both communication and computation, compared to retrieving a single element. We also discuss a way to extend PIRANA to labeled private set intersection (LPSI). Compared to existing LPSI protocols, PIRANA is more friendly to the scenarios where the database updates frequently.

References

[1] One server for the price of two: Simple and fast Single-Server private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association.

[2] Ishtiyaque Ahmad, Yuntian Yang, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. Addra: Metadata-private voice communication over fully untrusted infrastructure. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*. USENIX Association, 2021.

[3] 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.

[4] 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.

[5] 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.

[6] Daniel J. Bernstein, Mike Hamburg, Anna Krasnova, and Tanja Lange. Elligator: elliptic-curve points indistinguishable from uniform random strings. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 967–980. ACM, 2013.

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

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

[9] 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.

[10] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. *SIAM Journal on computing*, 43(2):831–871, 2014.

[11] 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.

[12] 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.

[13] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. 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 1223–1237. ACM, 2018.

[14] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1243–1255. ACM, 2017.

[15] 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.

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

[17] 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.

[18] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. 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 1135–1150. ACM, 2021.

[19] 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.

[20] 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.

[21] Craig Costello and Patrick Longa. Fourq: four-dimensional decompositions on a q-curve over the mersenne prime. Cryptology ePrint Archive, Paper 2015/565, 2015. https://eprint.iacr.org/2015/565.

[22] 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.

[23] Alex Davidson, Gonçalo Pestana, and Sofía Celi. Frodopir: Simple, scalable, single-server private information retrieval. *IACR Cryptol. ePrint Arch.*, page 981, 2022.

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

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

[26] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 303–324. Springer, 2005.

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

[28] 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.

[29] 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.

[30] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In László Babai, editor, *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, Chicago, IL, USA, June 13-16, 2004*, pages 262–271. ACM, 2004.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Muhammad Haris Mughees, Hao Chen, and Ling Ren. Onion-pir: 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.

[42] Muhammad Haris Mughees, Gonçalo Pestana, Alex Davidson, and Benjamin Livshits. Privatefetch: Scalable catalog delivery in privacy-preserving advertising. *arXiv preprint arXiv:2109.08189*, 2021.

[43] Muhammad Haris Mughees and Ling Ren. Vectorized batch private information retrieval. *Cryptology ePrint Archive*, 2022.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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, California, USA, February 21-24, 2016*. The Internet Society, 2016.