Lower Communication Stateless RLWE PIR

Andy Leiserson
Mozilla
andy@leiserson.org

Abstract

We present improved query packing techniques and parameters for RLWE-based PIR, reducing the communication required by stateless single-server protocols. Such protocols present fewer operational challenges than stateful or multi-server protocols, making them attractive for real-world deployments of PIR. Concretely, we achieve a stateless PIR from a 1 GiB database with less than 300 kB of communication, while maintaining reasonable amounts of server computation.

We discuss application of our techniques to a blocklist application. Google's Safe Browsing service provides information about websites and downloads that may be malicious or harmful. The Safe Browsing service currently submits prefixes of hashes of URLs to a server (sometimes via OHTTP), which provides some degree of non-cryptographic privacy for the queried URLs. We study the use of PIR to query the Safe Browsing database without revealing anything about queried URLs, and find that PIR for Safe Browsing can be implemented with queries under 200 kilobytes. Combined with a query rate optimization, we estimate that the Safe Browsing service can be fulfilled over PIR with an increase of less than 40% in the overall cost of providing the service.

Keywords

Private Information Retrieval, Safe Browsing

1 Introduction

A Private Information Retrieval (PIR) protocol is a cryptographic scheme enabling a client to request and obtain a specific record from a database held by a server, without the server learning which record the client requested.

There are numerous applications where this ability is useful. Examples include contact discovery [4], checking for compromised passwords [30], verifying that TLS certificates are properly recorded in transparency logs [23, 30], and screening for malicious or harmful content [9, 24].

Over time, different flavors of PIR have been introduced. Among the important attributes of a protocol are the number of servers required, what setup is required on the client and/or server, and the amount of state either side must maintain. While statefulness or additional servers can reduce the computational burden of a protocol, they also raise operational challenges. In the case of multiple servers, such protocols typically require that the servers be trusted not to collude. Assuring this may require placing the servers in different locations or under control of different organizations, which makes it more difficult to maintain consistent software and data on all the servers. Statefulness introduces additional failure modes and makes load balancing and routing more difficult. For these reasons, we focus on stateless protocols.

Recent work has greatly reduced the amount of server time necessary to handle PIR queries. However, while reducing the compute burden is valuable, it is only part of the picture. The size of queries and answers is also of utmost importance. Unlike compute, which can in many cases be distributed among multiple cores to reduce the user-facing impact, the rate at which data can be transferred to or from a user is largely fixed.

In this paper, we improve upon a series of works [2, 9, 29, 31] on PIR protocols making use of the homomorphic properties of Ring-LWE ciphertexts. The high-level structure of these protocols, and ours, is as follows: First, a packing step generates ciphertext(s) containing the query index. This data is sent to the server, along with some key material. Using homomorphic primitives, the server performs an "unpacking" operation to convert the query into a suitable form for execution. For example, unpacking may convert a single ciphertext containing multiple bits of the query index into a set of ciphertexts each with a single bit. After unpacking, the server executes the query in two stages. The first stage uses a unit vector e_i representing a portion of the query index, and homomorphically computes inner products of that unit vector with blocks of the database. The second stage selects between data blocks by homomorphically computing $(1 - b_i)d_j + b_i d_{j+1}$. Attempting to process the entire database using the approach of only one or the other stage would yield an impractical protocol, but the clever blend of the two approaches is quite efficient, while also managing the noise issues inherent in homomorphic operations using (Ring-)LWE.

We offer two main contributions. First, we aggressively search for parameters that allow the use of smaller rings, which reduces ciphertext size and thus can achieve a lower-communication protocol. Second, we improve upon the packing techniques of WhisPIR [9], enabling more query indices to be unpacked from ciphertext using a single key and without creating an excessive computational burden to extract them.

We provide concrete performance data from an implementation of our protocol. To demonstrate its practical relevance, we discuss how it could be applied to Google's Safe Browsing service.

1.1 Related Work

Private Information Retrieval was introduced by Chor et al. [14], who described schemes utilizing $k \geq 2$ non-colluding servers to provide private access to a database. Kushilevitz and Ostrovsky [25] described how a single server can provide private access to a database.

One strategy for reducing the cost of private information retrieval is moving some of the work to an "offline" stage that occurs ahead of the client deciding which record it wishes to query [15, 23, 24]. Some protocols go as far as streaming the entire database through the client (but not saving it) during the offline stage [33].

Due to the need for a PIR protocol to touch every bit of the database in processing a query [3], a natural limiter in the performance of PIR is the rate at which the algorithm can traverse the database in memory. Protocols requiring multiple servers were the first to

saturate the capacity to read from memory [20], with single-server protocols to follow some time later [23].

In this work, we build directly on the RLWE-based single-server PIR protocols SealPIR [2], OnionPIR [31], Spiral [29], and WhisPIR [9]. Of these, only the last is stateless. [1]

Other recent works proposing stateless, single-server PIR schemes include HintlessPIR [26], Tiptoe [22], and YPIR [30]. Of these, we focus on YPIR as a point of comparison for our protocol, due to its compelling performance.

The recent proposal Respire [8] achieves very low communication, but is not stateless.

Gentry [18] presented the idea of using automorphisms to rearrange data among ciphertext "slots". Angel et al. [2] showed how the $\mathcal{R}_q$-automorphisms $\pi_j : x \mapsto x^j$ can be used to unpack database indices from a PIR query. Many subsequent PIR schemes used or adapted this unpacking technique [9, 27, 29, 31].

Chilotti et al. [12] described a homomorphic multiplication operation between a BFV ciphertext and a GSW ciphertext known as the "external product". The external product was applied to PIR by OnionPIR [31], and is key to the two-stage query evaluation structure.

Chen et al. [11] presented techniques for converting one or multiple LWE ciphertext(s) into an RLWE ciphertext using the π_j automorphisms to evaluate the trace function.

The privacy properties of Safe Browsing were examined in [19]. The papers presenting the PIR schemes Checklist and FrodoPIR analyzed their application to Safe Browsing [16, 24] analyzed their application to Safe Browsing.

2 Preliminaries

We start by clarifying some notation. Then, we briefly recall the BFV and GSW variants of RLWE encryption, along with some homomorphic operations used by our scheme.

2.1 Notation

Bold letters like $\mathbf{g}$ denote vectors. $[n]$ denotes the set of integers $\{1, \ldots, n\}$. Logarithms, unless noted, are to base two. D denotes the size of the database that is the subject of a PIR query.

$\mathbb{Z}_q$ is the ring $\mathbb{Z}/q\mathbb{Z}$. $\mathcal{R}_q$ is the ring of polynomials $\mathbb{Z}_q[x]/(x^n+1)$. For the schemes described herein, n is always a power of two. We denote by π_j the $\mathcal{R}_q$-automorphism that maps $x \mapsto x^j$.

2.2 BFV Encryption

A BFV ciphertext $\mathbf{c} = (c_0, c_1) \in \mathcal{R}_q^2$ encrypting m satisfies $\langle \mathbf{c}, (-s, 1) \rangle = -c_0 \cdot s + c_1 = m + \varepsilon$, where ε is an error term [5, 17].

2.2.1 BFV Ciphertext-Plaintext Multiplication. Given a plaintext polynomial m and a BFV ciphertext (c_0, c_1) encrypting m', the ciphertext $(m \cdot c_0, m \cdot c_1)$ is an encryption of $m \cdot m'$. If the input ciphertext has noise σ, the output ciphertext has noise $m \cdot \sigma$.

[1] In principle it is possible to convert some stateful protocols to stateless ones by sending whatever information would have been provided offline with the query instead, but this tends to be an excessive amount of data.

2.3 GSW Encryption

2.3.1 Gadget Decomposition. Given a decomposition base β and length ℓ, the gadget vector $\mathbf{g}$ for β and ℓ is the vector $\left[1, \beta, \beta^2, \ldots, \beta^{\ell-1}\right]$. The base decomposition $g^{-1}(x)$ satisfies $g^{-1}(x) \cdot \mathbf{g} = x$ for all $x \in \mathcal{R}_q$.

2.4 Homomorphic Operations

2.4.1 BFV Key Switching. Given two secrets s and s', a key-switching key [6] K from s to s' can be obtained as follows:

(1) Sample $\mathbf{k}_0$ randomly from $\mathcal{R}_q^\ell$.
(2) Sample $\mathbf{e}$ from an error distribution.
(3) Compute $\mathbf{k}_1 = s' \cdot \mathbf{k}_0 - s \cdot \mathbf{g} + \mathbf{e}$.

Given the key-switching key $K = [k_0, k_1]$ and a ciphertext (c_0, c_1), the key-switched ciphertext (c_0', c_1') is computed as:

$$[c_0', c_1'] = g^{-1}(c_0) \cdot K + [0, c_1] \tag{1}$$

The output noise $\sigma_{\text{out}}^2 \leq \sigma_{\text{in}}^2 + \ell n(\beta^2/4)\sigma_{\text{ks}}^2$.

2.4.2 Automorphisms. Let τ be an automorphism that maps $p(x) \in \mathcal{R}_q$ to $p(x^k)$ for some k. We have $\langle (\tau(c_0), \tau(c_1)), (-\tau(s), 1) \rangle = \tau(m)$, i.e., the ciphertext $(\tau(c_0), \tau(c_1))$ is an encryption of $\tau(m)$ under the key $\tau(s)$. Therefore, if we perform a key switching operation on $(\tau(c_0), \tau(c_1))$ from $\tau(s)$ to s, we will have an encryption of $\tau(m)$ under the key s, with noise growth given by the key switching operation.

2.4.3 BFV-RGSW External Product. Given a BFV ciphertext $d \in \mathcal{R}_q^2$ encrypting m_d and a RGSW ciphertext $C \in \mathcal{R}_q^{2\ell \times 2}$ encrypting m_C, the external product [12] is defined as:

$$C \boxdot d = \left[g^{-1}(d_0) \quad g^{-1}(d_1)\right] \cdot C \tag{2}$$

$C \boxdot d \in \mathcal{R}_q^2$ is an encryption of $m_d \cdot m_C$. The output noise is the sum $\sigma_d^2 + \ell n(\beta^2/4)\sigma_C^2$.

3 Protocol Components

We begin by discussing some components that will later be incorporated in the overall protocol.

3.1 Splitting the Database

Chor et al. [14] describe a transformation to any PIR scheme that trades query size for answer size by splitting the database into s slices. The client sends one query to the server, which evaluates that query against each of the s slices independently and returns s responses to the client. With modern PIR schemes having $O(\log n)$ query size, this trade-off is relatively unappealing. However, it can be interesting when upload bandwidth is relatively more expensive than download bandwidth. There are also situations where neither packing more query indices into a BFV ciphertext nor processing additional index bits in the GSW folding stage is possible. (The first typically occurs when the noise budget precludes further packing, and the second typically occurs when the GSW stage has become the dominant component of server computation.) In such cases, this transform can still be interesting, and we include it in our parameter search discussed in Section 5.1. This database slicing transformation was also employed by WhisPIR [9].

3.2 Unpacking of BFV ciphertexts

Angel et al. [2] introduced an oblivious expansion procedure for unpacking multiple PIR query indices from a single BFV ciphertext. The expansion procedure takes as input a single BFV ciphertext containing c query indices. It starts by expanding that ciphertext into two ciphertexts, each containing $c/2$ query indices, and then continues recursively expanding each intermediate ciphertext into two ciphertexts each containing half as many query indices, until it reaches c ciphertexts each containing a single query index.

The expansion procedure utilizes the automorphisms over $\mathcal{R}_q$ that map $x \mapsto x^j$. The expansion relies on the following property: for $a \in \mathcal{R}_q$ let $a = \sum_i a_i x^i$. If $j = 2^k + 1$ for some $k \in [\log n]$, then the coefficients in $\pi(a)$ of x^i satisfy:

$$\begin{aligned} \{\pi(a)\}_i &= a_i \quad \text{if } i \text{ is a multiple of } \tfrac{2n}{2^k} \\ \{\pi(a)\}_i &= -a_i \quad \text{if } i \text{ is a multiple of } \tfrac{n}{2^k} \text{ but not of } \tfrac{2n}{2^k} \end{aligned} \tag{3}$$

Thus, computing $a \pm \pi(a)$ zeros certain coefficients of a. Repeated application of these automorphisms replaces all but a single coefficient with zero, achieving the desired unpacking.

In the original presentations, the client provided $\log c$ automorphism keys to use in unpacking c query indices. Each key contained $O(\ell)$ RLWE ciphertexts. The WhisPIR paper [9] observed that, in light of the isomorphism between the π_j automorphisms in the ring with dimension n and the group $\mathbb{Z}_{2n}^{\times}$, a single automorphism can be iterated to achieve other desired automorphisms. For example, in the case of $n = 1024$, $65^2 \equiv 129 \bmod 2048$. Therefore, $\pi_{129} = \pi_{65}^2$. Unfortunately, for values of $2^k \leq \sqrt{2n}$, the necessary iteration count may be quite large. Table 1 of [9] shows that as the desired packing degree increases, the number of required automorphism evaluations increases from hundreds, to hundreds of thousands.

However, it turns out that the automorphism corresponding to the $(2^k + 1)$-th power is not the only one that will work. For an odd integer c, and any $c \cdot 2^k + 1$:

$$(c \cdot 2^k + 1)^2 = c^2 \cdot 2^{2k} + c \cdot 2^{k+1} + 1 \tag{4}$$

$$= (c^2 \cdot 2^{k-1} + c) \cdot 2^{k+1} + 1 \tag{5}$$

And for any $\frac{2n}{2^k} \mid i$:

$$i \cdot (c \cdot 2^k + 1) = c' \cdot \frac{2n}{2^k} \cdot c \cdot 2^k + i \tag{6}$$

$$\equiv i \pmod{2n} \tag{7}$$

A similar argument shows that if $\frac{n}{2^k} \mid i$ but $\frac{2n}{2^k} \nmid i$, then $i \cdot (c \cdot 2^k + 1) \equiv i + n \pmod{2n}$. Thus, on an a with $a_i = 0$ for $\frac{n}{2^k} \nmid i$, the action of $\pi_{c \cdot 2^k + 1}$ is the same as the action of π_{2^k+1}.

We can unpack $d = 2^k$ values from a ciphertext a using only $\pi_{\frac{2n}{d}+1}$, starting by computing $a \pm \pi_{\frac{2n}{d}+1}^{2^{k-1}}(a)$ to produce two ciphertexts contain 2^{k-1} values, and continuing recursively until using $\pi_{\frac{2n}{d}+1}$ on each of 2^{k-1} intermediate ciphertexts to obtain d fully unpacked ciphertexts.

Compared to the original unpacking procedure, which scales the noise by a factor of d [2], this unpacking has the drawback that it scales the noise by d^2.

Table 1: Query Packing Configurations

	# Keys	Depth	Count	WhisPIR
2^1	1	1	1	
2^2	1	3	4	
2^2	2	2	3	
2^3	1	7	12	
2^3	3	3	7	
2^4	1	15	32	
2^4	4	4	15	
2^5	1	31	80	
2^5	5	5	31	
2^6	1	63	192	
2^6	6	6	63	
2^7	1	127	448	2496
2^7	7	7	127	
2^8	1	255	1024	7168
2^8	8	8	255	

Table 1 summarizes the different unpacking configurations in terms of the depth (relevant to noise growth) and count (relevant to compute burden) of automorphism evaluations.

3.3 Synthesis of RGSW ciphertexts

An RGSW ciphertext for a plaintext m can be viewed as 2ℓ BFV ciphertexts, encrypting the base-ℓ decompositions of m and $-s \cdot m$.

Algorithm 4 of [10] describes how to synthesize an RGSW ciphertext from a set of ℓ BFV ciphertexts encrypting a base decomposition of the plaintext m, and an RGSW encryption of $-s$. The external product operation is invoked using the RGSW encryption of $-s$ and each of the BFV plaintexts in turn, to obtain a base decomposition of $-s \cdot m$. OnionPIR [31] applies this to PIR. The client packs the ℓ components of the base decomposition together with query indices in BFV ciphertexts. The server then unpacks these values using the Angel et al. [2] protocol.

A drawback of this approach is that the noise level in the synthesized RGSW ciphertexts is substantial. In addition to the noise introduced by unpacking, the external product with RGSW$(-s)$ scales the noise by $n\|s\|^2$.

3.4 Modulus Switching

Finally, we adopt "modulus switching" for compression of the response. Modulus switching [7, 18] produces a new ciphertext $\mathbf{c'} = (c'_0, c'_1)$ with respect to a modulus $q' < q$ by rounding $c'_0 = \lfloor c_0 \cdot q'/q \rceil$ and $c'_1 = \lfloor c_1 \cdot q'/q \rceil$.

4 Protocol Details

Fix RLWE parameters n, $\log q$, and $\log p$, and decomposition bases t_{gsw} and t_{ks}.

Let D be the size of the database. Divide the database into s pieces. For simplicity, we assume that D is a multiple of s and that D/s is a power of two. Choose parameters i_{bfv} and i_{gsw} such that $i_{\text{tot}} = i_{\text{bfv}} + i_{\text{gsw}} = \log D$.

Through some mechanism, the client and server agree on a seed to use for PRG-based compression of transmitted ciphertexts.

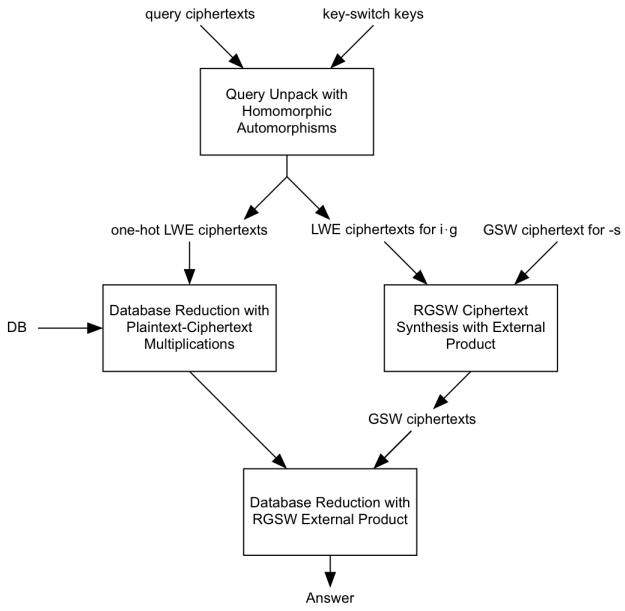

Figure 1: RLWE PIR Protocol Overview

Let $i \in [0, D/s)$ be the client's query index. Divide the query index into two pieces. $(i \bmod 2^{i_{\text{bfv}}})$ is the first-stage query index, and $\lfloor i \,/\, 2^{i_{\text{bfv}}} \rfloor$ is the second-stage query index.

Construct a length-$(2^{i_{\text{bfv}}})$ vector v over $\mathbb{Z}_q$ that is all zero except for the position indicated by the first-stage query index. If synthesizing RGSW ciphertexts, compute the base-t_{gsw} decompositions of the second stage query indices and append them to v.

Pack the vector v into a vector u over $\mathcal{R}_q$, placing d elements of v into the lowest-order coefficients of each element of u. Utilizing the PRG to generate the c_0 term, encrypt u using BFV encryption.

Again using the PRG to generate half of the components, generate an automorphism key for $\pi_{\frac{2n}{d}}$.

Finally, if RGSW synthesis is not in use, encrypt the second-stage query indices bit-by-bit using RGSW encryption. If RGSW synthesis is in use, generate an RGSW encryption of $-s$.

The query consists of the non-PRG-derived components of the BFV ciphertexts, the automorphism key, and if applicable, the RGSW ciphertexts.

Possibly in an offline stage, the server may perform the unpacking procedure for the PRG-generated components of the BFV query ciphertexts, and precompute the multiplication with the database.

Upon receiving the query, the server first uses the automorphism key to unpack the BFV ciphertexts as described in Section 3.2. If RGSW synthesis is in use, the server multiplies base decomposition component from the unpacked query with $-s$ and assembles RGSW ciphertexts.

The server then multiplies the first-stage query vector with the database to reduce it by a factor of $2^{i_{\text{bfv}}}$. This requires a polynomial multiplication for each database element, but the database can be transformed to the NTT domain ahead of time, so this step can be accomplished with $O(D)$ pointwise multiplications and $O(D/2^{i_{\text{bfv}}})$ inverse NTTs.

The server then uses each of the RGSW ciphertexts successively to reduce the remaining database by a factor of two for each ciphertext. At the end, one ciphertext (two $\mathcal{R}_q$ elements) remain.

The server then repeats the first- and second-steps of processing against the database for each of the $s-1$ additional slices. Finally, the server converts the ciphertexts to the reduced modulus q'. The answer consists of $2s$ $\mathcal{R}_{q'}$ elements.

5 Parameter Selection

In this section we discuss concrete parameters for instantiation of our scheme. The choice of parameters for the PIR scheme can optimize for one or more of several competing factors:

- The computation necessary to answer a query.
- The communication required by a query. Often, the total communication for both query and answer is considered. However, in many applications, upload bandwidth is more limited than download bandwidth, thus, it may be desirable to consider the query size separately from the answer size.
- The rate of the scheme, meaning the ratio of database bytes retrieved to the answer size.

5.1 RLWE Parameters

A major challenge when using lattice-based homomorphic encryption is managing the "noise growth". A ciphertext in a lattice-based encryption scheme incorporates some amount of noise. Decryption algorithms applied to the ciphertext are probabilistic. When the noise is small, the chance that decryption fails to recover the correct plaintext is negligible. As the noise increases, so does the probability of decryption failure.

Homormorphic operations produce an output ciphertext with more noise than was present in the input ciphertext. The extent of the noise increase depends on the operation being performed. An algorithm for homomorphic computation must account for the noise growth to ensure correctness.

The ciphertext modulus q is the the primary determinant of the amount of noise that an RLWE-based PIR protocol will tolerate. Common values of q are 56 or 108 bits, with larger values allowing for more noise. However, larger q values also increase the size of ring elements and thus of the messages in a PIR protocol. The size of an encoded ring element is $n \log q$, where the ring dimension n is typically 2048 for $q = 56$ and 4096 for $q = 108$, yielding ciphertext sizes of 14.3 kB and 55.3 kB.

While stateful PIR protocols may produce a query consisting of only one such ring element [2, 29, 31], stateless protocols have thus far required more. In this work, in addition to $\log q$ of 56 or 108, we also consider the combination $(n, \log q) = (1024, 30)$, which has a ring element encoding approximately 4x smaller than $\log q = 56$.

For simplicity of implementation we restrict ourselves to the power-of-two ring dimensions 1024 and 2048. We use a discrete gaussian distribution with standard deviation $\sigma = 6.4$ for both the secret and the noise. We use a prime or product of two primes for the modulus, and select the size to provide 128 bits of security as estimated by the Lattice Estimator [1], arriving at $\log q = 30$ for $n = 1024$ and $\log q = 56$ for $n = 2048$. While our search script for the

non-lattice parameters supports $n = 4096$, no such configurations were selected.

5.2 Other Parameters

We have implemented a tool to search for suitable parameter sets in Python. The parameters considered by the tool are:

- D, the database size.
- s, the number of database slices.
- n, the ring degree, which is chosen simultaneously with $\log q$, the size of the ciphertext modulus.
- $\log p$, the size of the plaintext modulus.
- t_{gsw}, the decomposition base for RGSW query ciphertexts.
- t_{ks}, the decomposition base for automorphism keys.
- i_{bfv} and i_{gsw}, the number of query index bits processed in each stage of processing. The total number of query index bits (i.e. the sum $i_{\text{bfv}} + i_{\text{gsw}}$) is determined by D, n, q, and p.
- Whether to pack base decompositions into the BFV query ciphertexts and synthesize RGSW ciphertexts on the server, or to send RGSW ciphertexts directly.
- A query packing scheme degree d.
- The number s of database slices to be processed independently as discussed in Section 3.1.

For each set of candidate parameters, the tool computes the following:

- The noise level in the answer ciphertexts.
- The query and answer size.
- The server compute time to fulfill the query.

Although it is not reported explicitly in the table, note that the expansion factor of the database when converted to NTT representation, $\log q \,/\, \log p$, is an important figure. Assuming that the implementation of the first stage of answering a query is able to saturate the memory bandwidth, the effective throughput in terms of actual database bytes is $(\log p \,/\, \log q)$ times the memory bandwidth. While our analysis considers the cost of increased server time in configurations that process the database at lower rates, another concern is the amount of memory required to hold the NTT-converted database. In low- traffic applications where the CPU is not readily saturated, the high memory load relative to CPU load might be a concern.

Table 2 shows several possible configurations for 256 MiB and 1 GiB databases. Configurations with a name to the left are evaluated in Section 6. (Later, in Section 7, we provide configurations for a 128 MiB Safe Browsing database.) For each combination of database size and $\log q$, the following configurations are shown: (1) the configuration with the minimum cost, under the query size cost metric, (2) the configuration with the minimum query size, (3) the configuration with the minimum cost, under the overall cost metric, (4), the configuration with the minimum total communication, (5) the configuration with the minimum compute. Only configurations with query size less than 1 MiB are considered. The final row of the table is an additional configuration that appears in Figure 2 and Table 3.

The cost size metrics assume charges of \$0.10/GB for data transfer and of \$0.05 per core hour for compute. The overall cost metric charges equally for both query and answer size. The query size cost metric charges only for the query size, but doubles the per-byte cost, and excludes configurations with response size more than 1.25x the query size.

5.3 Discussion

With $\log q = 30$, the noise budget is very limited. It is not possible to pack more than a few tens of indices into a query ciphertext. Synthesizing GSW ciphertexts from a packed base decomposition is out of the question. On the other hand, the small size of ring elements means that making up for the limited packing capacity by sending more query ciphertexts is viable. Although not selected under the the metrics used for the table, there are numerous interesting configurations with $(\log q, \log p) = (30, 2)$.

6 Implementation and Evaluation

We have implemented our PIR scheme in Rust. Polynomial multiplication in $\mathcal{R}_q$ is accelerated using the NTT routines from TFHE-rs [32]. For $\log q = 62$, we use the Chinese Remainer Theorem to represent elements of $\mathbb{Z}_q$ modulo two 31-bit primes. We make use of AVX-512 instructions, which requires the use of the "nightly" (unstable) version of the Rust compiler. We use ChaCha12 to generate pseudorandom ring elements for purposes of compressing transferred ciphertexts.

6.1 Optimizing the First Stage

The first stage of query processing involves multiplying a BFV query ciphertext with each element of the database, and accumulating these products. As is customary, we have optimized this by precomputing the NTT for each database element. In the NTT domain, multiplication of polynomials becomes an pointwise product of the NTT representations. Because each query ciphertext is multiplied with many database elements, and the products are accumulated, this greatly reduces the number of NTTs that need to be performed in the online phase.

The pointwise product between the NTTs of query ciphertext and database element is in $\mathbb{Z}_q$. Computing this product nominally requires a reduction modulo q. Another well-known optimization is to defer this reduction, i.e., calculate an intermediate result $\sum q_i d_i$ in $\mathbb{Z}$ and perform only a single reduction on that intermediate result rather than reducing each $q_i d_i$ product. This optimization is particularly appealing when the modulus q is somewhat less than the machine's word size.

Unfortunately, a 30-bit q in a 32-bit machine word leaves limited room to accumulate products before reducing. Nevertheless, we were able to construct a multiply-accumulate kernel using the AVX-512 instruction set, that reduces after accumulating four products, and still saturates the single-core memory bandwidth on our test machines.

Additionally, we precompute the entire half of the computation that depends only on the PRG-derived term of the ciphertext. Note that this requires evaluating the effect of the query unpacking process to obtain the actual values multiplied with the database. We also precompute the inverse NTT and base-decomposition for the first external product in the second stage. It would be possible to continue and precompute even the inner product between the base decomposition and the PRG-derived term of the GSW ciphertext, but we have not yet done that.

Table 2: Parameters for 256 MiB and 1 GiB databases

	$\log q$	$\log p$	s	ℓ_{gsw}	ℓ_{ks}	i_{tot}	i_{gsw}	d	Req. kiB	Resp. kiB
Configurations for 256 MiB database										
	30	1	64	2	10	15	7	16	203	224
	30	1	64	2	8	15	9	16	180	224
`256mb-a`	30	1	8	2	10	18	10	16	248	28
	30	1	8	2	8	18	12	16	225	28
	30	6	8	3	-	16	9	1	682	40
	56	8	16	2	6	13	4	256	336	168
	56	1	16	4	7	16	12†	32	238	168
	56	4	1	6	8	18	10†	128	322	10
	56	1	1	4	8	20	14†	64	252	10
	56	8	8	2	4	14	5	16	952	84
Configurations for 1 GiB database										
`1gb-b`	30	1	64	2	10	17	9	16	232	224
	30	1	64	2	8	17	11	16	210	224
	30	1	8	2	15	20	11	16	341	28
`1gb-a`	30	1	8	2	8	20	14	16	255	28
	30	6	8	3	-	18	11	1	728	40
	56	4	1	6	8	20	11†	128	350	10
	56	1	1	4	8	22	16†	64	252	10
`1gb-c`	56	6	1	7	8	20	11†	128	378	10
	56	1	1	4	8	22	16†	64	252	10
`1gb-e`	56	12	8	2	8	16	8	16	896	100
`1gb-d`	56	8	8	2	5	16	7	256	560	84

† indicates configurations using RGSW synthesis

6.2 Results

We include performance measurements from the Intel Ice Lake architecture (Azure Esv5 instances) to facilitate comparison with other work. However, we note that the performance of many modern PIR implementations is limited by memory bandwidth. According to [28] [2], the single-core memory bandwidth of the AMD Genoa architecture is 45 GB/s, compared to 17 GB/s for Intel Ice Lake. Prompted by this, we also include performance results from the Genoa architecture (Azure Easv6 instances, which are comparable in cost to Esv5) and find that a significant performance increase is indeed realized.

Our test machines run Ubuntu 24.04. We use build `nightly-2025-02-07` of the Rust compiler. Reported CPU times are for single-thread execution.

In Figure 2, we show the tradeoff between communication and computation for our scheme and several other stateless, single-server PIR schemes. (HintlessPIR [26] does not appear in the figure because its communication is 2.1 MB.) Our data for this figure is measured on Azure Esv5 (Ice Lake) instances, and is also presented in Table 3. Data for other schemes is taken from the respective publications. Table 4 compares the performance achieved by our implementation with YPIR [30], and also compares the performance on Ice Lake with the performance on Genoa.

[2]Genoa appears in the updated version at https://sites.utexas.edu/jdm4372/2023/12/19/the-evolution-of-single-core-bandwidth-in-multicore-systems-update/

6.3 Discussion

As the database size increases, so too does the cost of processing it as part of a PIR protocol. It becomes relatively more attractive to trade faster traversal of the database for more computationally intensive decoding and encoding of the query and answer. Thus, we expect our scheme to be of interest primarily for databases of several gigabytes or less.

While we report single-thread CPU time for ease of analysis and comparison, note that this is not indicative of the user-facing response time, because an actual implementation could parallelize evaluation of the query across multiple cores. However, the performance scaling to multiple cores may be more complicated than it would initially appear. While the algorithm itself is amenable to parallel execution, the available memory bandwidth when all cores are working in parallel may be less than the product of the core count with the single-core memory bandwidth.

7 Application to Safe Browsing

Google's Safe Browsing service [21] provides information about websites and downloads that may be malicious or harmful, such as phishing or malware distribution sites. The service is used by numerous web browsers, including Chrome, Firefox, and Safari. When a user of a Safe Browsing-enabled client accesses a resource, the client constructs a set of URL fragments including the complete URL, but also various portions of the URL (for example, the fragment `example.com` in addition to the complete URL `subdomain.`

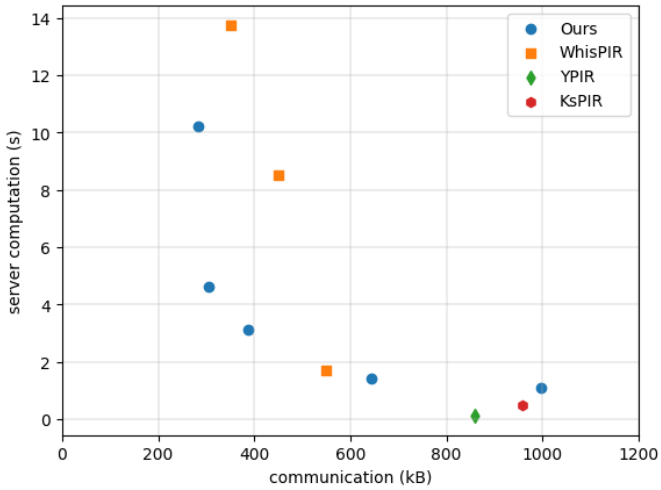

Figure 2: Communication and computation for querying a 1 GiB database with various PIR schemes.

Table 3: Communication and Computation Costs for a 1 GiB Database

Configuration	Req kiB	Resp kiB	ICL Compute	Genoa Compute
1gb-a	255	28	10205	6953
1gb-b	278	28	4645	2599
1gb-c	378	10	3140	1375
1gb-d	560	84	1440	792
1gb-e	896	100	1088	603

Table 4: Comparison to Other PIR Schemes

Database	Metric	YPIR	Ours
256 MiB	Server Time (ms)	70	1178
e4s_v5	Throughput (GB/s)	3.8	0.23
Intel Ice Lake	Upload (kiB)	670	248
	Download (kiB)	12	28
1 GiB	Server Time	140	3140
e8s_v5	Throughput	7.7	0.34
Intel Ice Lake	Upload	866	378
	Download	12	10
256MiB	Server Time	56	623
e4as_v6	Throughput	4.8	0.43
AMD Genoa	% Speedup	25%	89%
1GiB	Server Time	90	1375
e8as_v6	Throughput	11.9	0.78
AMD Genoa	% Speedup	56%	128%

`example.com/resource`). The client then hashes each of the fragments, and proceeds to check whether any of those hashes appear on Safe Browsing's list of questionable resources. To reduce the frequency with which they need to contact the server, clients periodically synchronize a smaller first-level database containing the 32-bit prefixes of each hash appearing in the full database. When querying for a hash, if the prefix does not appear in the small database, no server query is necessary. If a client does find a prefix in the first-level database, it asks the server for full hashes beginning with that prefix. Sending only relatively short hash prefixes to the server provides a degree of privacy. Many URLs likely map to each prefix, making it uncertain whether the client was interested in any particular URL. However, it is difficult to translate this into a

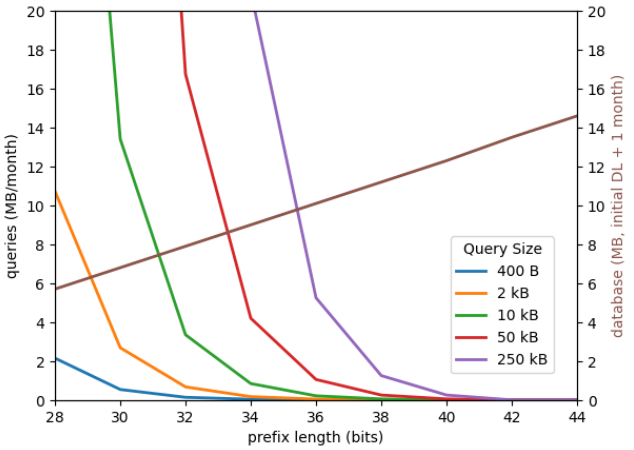

Figure 3: Prefix Length Tradeoff

formal privacy claim, and it is certainly revealing some information, specifically, a vast volume of URLs are known not to be the subject of any particular query.

Periodic synchronization of the first-level database implies a window where clients are not aware of recently-detected malicious sites. Google has recently announced a new version (v5) of the Safe Browsing API to improve the "real-time" alerting capabilities of Safe Browsing. With the new version of the API, the sense of the first-level screening inverts. The client maintains information about sites that are likely benign. If a site does not appear on that list, then a query is made to the server. This increases the possibility of sending sensitive browsing activity to the server. It is possible to query Safe Browsing v5 over OHTTP, but this only masks the origin of the request; it does not protect the information in the request. In other words, the information that somebody is querying for a particular hash prefix is still revealed. Additionally, querying via OHTTP leaves open the possibility of certain correlation or reidentification attacks.

Motivated by the increasing privacy concerns arising from real-time Safe Browsing queries, we investigate whether it is feasible to use PIR to query the Safe Browsing database.

7.1 Reducing the Query Rate

At first glance, increasing the size of a Safe Browsing query from a few kilobytes (primarily from the overhead of establishing a TLS connection, rather than the size of the query itself) to hundreds kilobytes for a PIR query, appears that it may significantly increase the cost of the server-side API. However, the introduction of PIR to the equation creates an opportunity to reduce the query rate, thus reducing the effective cost of adopting PIR.

The first-level Safe Browsing database contains several million 32-bit hash prefixes. Given the 32-bit space, this means that a randomly chosen prefix has an approximately 0.1% chance of matching a prefix present in the first-level database. As with any approximate membership query data structure, there is a tradeoff between the size of the database, and the false-positive rate. However, there is an additional factor driving the selection of the false-postive rate for Safe Browsing. As the prefix length increases (and the false-positive rate decreases), the level of privacy also decreases, because fewer URLs map to each bucket.

If PIR is used to query the server, then this leakage is no longer a concern, and we are free to choose a false positive rate based on the tradeoff with the size of the data structure. As the size of a query increases, so does the benefit of reducing the number of queries, even at the cost of increasing the first-level database size. Figure 3 illustrates this tradeoff. We propose to increase the prefix length in the first-level database to 40 bits, which results in a probability around 2^{-18} of a random prefix colliding with an entry in the database.

7.2 Applying PIR to Safe Browsing

Using PIR for Safe Browsing requires resolving a few other difficulties. A PIR scheme that queries a database by index is not actually directly usable, because the client instead desires to query based on hash prefixes. What Safe Browsing actually calls for is a "PIR by keywords" protocol [13]; the hash prefixes are the keywords.

One strategy for adapting a traditional PIR scheme to a keyword-based lookup is to construct some sort of hash table, for example, cuckoo hashing. However, in the case of Safe Browsing, we can take another approach leveraging the fact that the protocol is already distributing a first-level data structure. Since the client already consults this data structure in preparing to make a query, we can augment it with information specifying the index to query in the full database.

One approach (taken by Checklist [24]) is to convey the mapping to the full database implicitly, by transmitting hash prefixes to the client in the order that they appear in the full database. This does make it harder to compress the prefixes in transit (currently, the prefixes are transmitted in sorted order so that they can be compressed with Rice-Golomb coding). There is also an increased storage cost on clients to remember the distribution order while also supporting efficient membership-testing against the set of prefixes.

Part of the challenge faced by Checklist is that it is a PIR scheme with preprocessing, meaning that the client downloads some "hint" information from the server ahead of time. Every time the database changes, previous hints are invalidated and new hints must be downloaded. In PIR schemes without preprocessing like ours, another possibility arises: maintaining the (full) database in sorted order at all times. Then, based only on the location of a particular prefix in the sorted first-level database, a client may determine where to look in the full database.

The natural record size supported by our PIR scheme is $n \log p$, which is sufficient in all candidate configurations to contain multiple Safe Browsing hashes.

7.3 Results

For Safe Browsing, we focus on reducing query size, due to the need to support a variety of web clients, including mobile devices, and because consumer internet connections typically have significantly lower upload speeds than download speeds.

A Safe Browsing database of four million 256-bit hashes is 2^{30} bits or 128 MiB. We show three possible sets of parameters for this database size in Table 5. The first and second parameter sets are

Table 5: Parameters for 128 MB Safe Browsing database

$\log q$	$\log p$	s	ℓ_{gsw}	ℓ_{ks}	i_{tot}	i_{gsw}	d	$\log q'$	Req. kiB	Resp. kiB
30	1	64	2	10	14	6	16	14	188	224
30	1	8	2	10	17	9	16	14	232	28
56	8	16	2	4	12	4	64	21	392	168

Table 6: Cost Estimates for Safe Browsing Service

Parameter	Value
Database additions and removals per hour	600 each
Hourly raw query rate rate (before filtering)	500
New user percentage	15%
Compute cost per core hour	$0.05
Data transfer cost per gigabyte	$0.10

	Existing	Full DB	PIR
First-level database initial transfer, MB	5.2	134	9.4
First-level database updates, MB	2.9	15	3.3
Query transfer, MB	negl	-	0.3
Cost per million users	370	3514	514

quite similar, differing only in the database slicing degree s (and consequences thereof). The third configuration has better database utilization ($\log p / \log q$), and consequently lower estimated compute. Measuring the first parameter set, it takes 531 ms of CPU time to answer a query on an `E2s_v5` Ice Lake machine, and 300 ms on an `E2as_v6` Genoa machine.

Table 6 estimates the cost of providing the existing Safe Browsing service, of distributing the full database, and of providing the Safe Browsing service over PIR. In the case of PIR, a large portion of the cost is due to increasing the size of the first-level data structure to drive the query rate down. Overall, the PIR solution is less than 40% more expensive than the existing solution, and significantly cheaper than distributing the full database. Because major cloud providers do not charge for ingress bandwidth, it is possible to reduce the cost further by trading query size for answer size, however, due to the asymmetry of consumer internet connections, this might compromise the overall quality of service.

8 Conclusion

We have shown how to reduce the communication required for stateless, single-server PIR to less than 300 kB total, or less than 200 kB for the query. This makes practical deployment of PIR more feasible, especially in applications serving bandwidth-constrained clients.

References

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology* 9, 3 (Oct. 2015), 169–203. https://doi.org/10.1515/jmc-2015-0016 Publisher: De Gruyter.

[2] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy (SP)*. 962–979. https://doi.org/10.1109/SP.2018.00062 ISSN: 2375-1207.

[3] Amos Beimel, Yuval Ishai, and Tal Malkin. 2000. Reducing the Servers' Computation in Private Information Retrieval: PIR with Preprocessing *(Lecture Notes in Computer Science)*, Mihir Bellare (Ed.). Springer, Berlin, Heidelberg, 55–73. https://doi.org/10.1007/3-540-44598-6_4

[4] Nikita Borisov, George Danezis, and Ian Goldberg. 2015. DP5: A Private Presence Service. *Proceedings on Privacy Enhancing Technologies* (2015). https://petsymposium.org/popets/2015/popets-2015-0008.php

[5] Zvika Brakerski. 2012. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In *Advances in Cryptology – CRYPTO 2012*, Reihaneh Safavi-Naini and Ran Canetti (Eds.). Springer, Berlin, Heidelberg, 868–886. https://doi.org/10.1007/978-3-642-32009-5_50

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Trans. Comput. Theory* 6, 3 (July 2014), 13:1–13:36. https://doi.org/10.1145/2633600

[7] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE, Palm Springs, CA, USA, 97–106. https://doi.org/10.1109/FOCS.2011.12

[8] Alexander Burton, Samir Jordan Menon, and David J. Wu. 2024. Respire: High-Rate PIR for Databases with Small Records. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1463–1477. https://doi.org/10.1145/3658644.3690328

[9] Leo de Castro, Kevin Lewi, and Edward Suh. 2024. WhisPIR: Stateless Private Information Retrieval with Low Communication. https://eprint.iacr.org/2024/266 Publication info: Preprint..

[10] Hao Chen, Ilaria Chillotti, and Ling Ren. 2019. Onion Ring ORAM: Efficient Constant Bandwidth Oblivious RAM from (Leveled) TFHE. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. Association for Computing Machinery, New York, NY, USA, 345–360. https://doi.org/10.1145/3319535.3354226 Chen19OnionRingORAM.

[11] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2021. Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. In *Applied Cryptography and Network Security*, Kazue Sako and Nils Ole Tippenhauer (Eds.). Springer International Publishing, Cham, 460–479. https://doi.org/10.1007/978-3-030-78372-3_18

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology* 33, 1 (Jan. 2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[13] Benny Chor, Niv Gilboa, and Moni Naor. 1998. Private Information Retrieval by Keywords. https://eprint.iacr.org/1998/003 Publication info: Published elsewhere. Appeared in the THEORY OF CRYPTOGRAPHY LIBRARY and has been included in the ePrint Archive..

[14] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. 1998. Private information retrieval. *J. ACM* 45, 6 (Nov. 1998), 965–981. https://doi.org/10.1145/293347.293350

[15] Henry Corrigan-Gibbs and Dmitry Kogan. 2019. Private Information Retrieval with Sublinear Online Time. https://eprint.iacr.org/2019/1075

[16] Alex Davidson, Gonçalo Pestana, and Sofia Celi. 2023. FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval. *Proceedings on Privacy Enhancing Technologies* 2023, 1 (Jan. 2023), 365–383. https://doi.org/10.56553/popets-2023-0022

[17] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. https://eprint.iacr.org/2012/144

[18] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Fully Homomorphic Encryption with Polylog Overhead. In *Advances in Cryptology – EUROCRYPT 2012*, David Pointcheval and Thomas Johansson (Eds.). Springer, Berlin, Heidelberg, 465–482. https://doi.org/10.1007/978-3-642-29011-4_28

[19] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. 2016. A Privacy Analysis of Google and Yandex Safe Browsing. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 347–358. https://doi.org/10.1109/DSN.2016.39 ISSN: 2158-3927.

[20] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications *(Lecture Notes in Computer Science)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.). Springer, Berlin, Heidelberg, 640–658. https://doi.org/10.1007/978-3-642-55220-5_35

[21] Google. 2025. Safe Browsing. Retrieved February 26, 2025 from https://safebrowsing.google.com/

[22] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. 2023. Private Web Search with Tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 396–416. https://doi.org/10.1145/3600006.3613134

[23] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One server for the price of two: simple and fast single-server private information retrieval. In *Proceedings of the 32nd USENIX Conference on Security Symposium (SEC '23)*. USENIX Association, USA, 3889–3905.

[24] Dmitry Kogan and Henry Corrigan-Gibbs. 2021. Private Blocklist Lookups with Checklist. 875–892. https://www.usenix.org/conference/usenixsecurity21/presentation/kogan

[25] E. Kushilevitz and R. Ostrovsky. 1997. Replication is not needed: single database, computationally-private information retrieval. In *Proceedings 38th Annual Symposium on Foundations of Computer Science*. 364–373. https://doi.org/10.1109/SFCS.1997.646125 ISSN: 0272-5428.

[26] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. 2024. Hintless Single-Server Private Information Retrieval. In *Advances in Cryptology – CRYPTO 2024*, Leonid Reyzin and Douglas Stebila (Eds.). Springer Nature Switzerland, Cham, 183–217. https://doi.org/10.1007/978-3-031-68400-5_6

[27] Ming Luo, Feng-Hao Liu, and Han Wang. 2024. Faster FHE-Based Single-Server Private Information Retrieval. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. Association for Computing Machinery, New York, NY, USA, 1405–1419. https://doi.org/10.1145/3658644.3690233

[28] John McCalpin. 2023. The evolution of single-core bandwidth in multicore processors. Retrieved February 19, 2025 from https://sites.utexas.edu/jdm4372/2023/04/25/the-evolution-of-single-core-bandwidth-in-multicore-processors/

[29] Samir Jordan Menon and David J. Wu. 2022. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *2022 IEEE Symposium on Security and Privacy (SP)*. 930–947. https://doi.org/10.1109/SP46214.2022.9833700 ISSN: 2375-1207.

[30] Samir Jordan Menon and David J. Wu. 2024. YPIR: High-Throughput Single-Server PIR with Silent Preprocessing. In *Proceedings of the 33rd USENIX Conference on Security Symposium (SEC '24)*. USENIX Association, USA, 5985–6002.

[31] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2292–2306. https://doi.org/10.1145/3460120.3485381

[32] Zama. 2022. TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data. https://github.com/zama-ai/tfhe-rs

[33] Mingxun Zhou, Andrew Park, Wenting Zheng, and Elaine Shi. 2024. Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation. In *2024 IEEE Symposium on Security and Privacy (SP)*. 4296–4314. https://doi.org/10.1109/SP54263.2024.00055 ISSN: 2375-1207.