PIR with compressed queries and amortized query processing

Sebastian Angel[⋆†], Hao Chen[‡], Kim Laine[‡], and Srinath Setty[‡]

[⋆]The University of Texas at Austin [†]New York University [‡]Microsoft Research

Abstract

Private information retrieval (PIR) is a key building block in many privacy-preserving systems. Unfortunately, existing constructions remain very expensive. This paper introduces two complementary techniques that make the computational variant of PIR (CPIR) more efficient in practice. The first technique targets a recent class of CPU-efficient CPIR protocols where the PIR query sent by the client is a vector containing a number of ciphertexts proportional to the size of the server's database. We propose a new method to compresses this vector into a single ciphertext, thereby reducing network costs by over 120×.

The second technique is a new data encoding called a *probabilistic batch code* (PBC). We use PBCs to build a multi-query PIR scheme that allows the server to amortize the computational cost of processing a batch of requests. The protocol achieves a 6.7× speedup over processing queries one at a time, and is significantly more efficient than related encodings. We apply our techniques to the Pung unobservable communication system which relies on a custom multi-query CPIR protocol for its privacy guarantees. Replacing Pung's protocol with our schemes, we find that we can simultaneously reduce network costs by 33× and increase throughput by 2×.

1 Introduction

A key cryptographic building block in recent privacy-preserving systems is *private information retrieval* (PIR) [28]. Examples include: anonymous and unobservable communication [7, 54, 60], privacy-preserving media streaming [4, 44], ad delivery [42], friend discovery [14], and notifications [26].

PIR allows a client to download an element (e.g., movie, Web page, friend record) from a database held by an untrusted server (e.g., streaming service, social network) without revealing to the server *which* element was downloaded. PIR is very powerful, but it is also very expensive—and unfortunately this expense is fundamental: PIR schemes force the database server to perform some computation on every element in the database to answer a single client query [28]. After all, if the server were to omit an element when answering a query it would learn that the omitted element is of no interest to the client. Consequently, this lower bound limits the scale at which these systems can operate.

The issue of costs is even more pronounced in the computational variant of PIR (CPIR) [53]. On the one hand, this variant is desirable because it allows PIR to be used by systems under cryptographic hardness assumptions rather than having to assume multiple non-colluding servers (as is the case with the information-theoretic variant of PIR). On the other hand, the computational and network costs of even the most recent CPIR constructions [4, 52, 57] are so significant, that all CPIR-backed systems must settle with supporting small databases with fewer than 100K entries [4, 7, 42, 44].

The goal of this work is to make CPIR more efficient in practice. To this end, this paper discusses two complementary contributions. The first is the introduction of SealPIR, a new PIR library that extends the XPIR protocol [4] by compressing PIR queries (§3). In XPIR (and its base protocol [70]), a client must encode the index of her desired element in the server's database using a query vector containing n ciphertexts (where n is the number of elements in the server's database). Each ciphertext is a different encryption of 0, except for one ciphertext—located at the desired index in the vector—which encrypts 1. Stern [70] showed that it is possible to reduce the number of entries in the vector to $d\sqrt[d]{n}$ ciphertexts for any positive integer d. However, this comes at an exponential increase in the size of the response (§3.3). Indeed, for values of $d > 3$ in XPIR, the larger response far outweighs the reduction in query size for all the databases with which we experiment (§7.1).

Instead of creating a query vector, SealPIR has the client send a single ciphertext containing an encryption of the index. The server then executes a new *oblivious expansion procedure* that extracts the corresponding n-ciphertext vector from the single ciphertext, without leaking any information about the client's index, and without increasing the size of the response (§3.2). The server can then proceed with the XPIR protocol on the extracted vector as before.

In terms of concrete savings over XPIR, SealPIR results in 120× lower network costs and 9× lower CPU cost for the client. However, SealPIR introduces costs to the server. SealPIR adds 81% CPU overhead (over XPIR) to obliviously expand queries, and 1.05× to 4.6× CPU overhead to answer the expanded queries. The latter cost is not fundamental: our implementation simply uses a different cryptographic library than XPIR, which is less optimized (we explain why we chose this library in Section 3.2). Nevertheless, we think this is an excellent tradeoff since XPIR's protocol is embarrassingly parallel and we can regain any lost throughput by employing additional hardware at the server. In contrast, no amount of additional hardware can reduce the client-to-server communication costs.

The second contribution of this paper is a new technique to amortize the server's CPU cost when processing multiple PIR queries from the same client. Our technique is a relaxation of *batch codes* [47], a data encoding that can in principle be used to amortize CPU costs in multi-query PIR. In practice, most batch code constructions target a different domain—providing load balancing and availability guarantees to distributed storage [64, 67] and network switches [75]—and using them to amortize PIR queries is not advisable. In particular, they result in cubic (or even higher) blow-up in network costs over using single-query PIR multiple times, which is an untenable expense in practice. Our encoding, called *probabilistic batch codes (PBC)*, addresses this issue at the expense of introducing a small probability of failure (§4.2). In the context of multi-query PIR, failure simply means that a client only gets some (not all) of her queries answered in a single interaction. While the implications of failure depend on the application, we argue that in many cases this is not an issue in practice (§5). Furthermore, the failure probability of our constructions is very low—about 1 in a trillion multi-queries would be affected.

The key idea behind our efficient PBC construction is a simple new technique called *hashing in the head* (§4.3). This technique flips the way that hashing (e.g., multi-choice [61], cuckoo [63]) is typically used in distributed systems to achieve load balancing: *instead of executing the hashing algorithm during data placement, it is executed during data retrieval*. Our PBC construction amortizes CPU costs when used for multi-query PIR, while simultaneously incurring orders of magnitude lower network costs than existing batch codes (§7.3). Furthermore, PBCs are general and can be used to amortize *any* PIR scheme (both CPIR and the information-theoretic variants).

To demonstrate the benefits of our techniques in practice, we apply them to Pung [7], a recent unobservable communication system that relies on XPIR for its privacy guarantees. Pung enables users to communicate over a fully untrusted communication channel while hiding communication metadata such as the identity of participants, time a message is sent, etc. Pung introduces an application-specific multi-query PIR scheme that allows clients to send and receive multiple messages simultaneously, enabling group communication applications (e.g., Slack) as well as batch retrieval (e.g., downloading several e-mails).

When we replace Pung's custom multi-query PIR scheme (which relies on an existing batch code due to Ishai et al. [47]) with our construction based on PBCs, we achieve 3.4× lower network costs, and 1.98× higher throughput (our construction is actually an improvement on all axes). When we also replace XPIR with SealPIR in Pung, the combination of our techniques achieves 33.2× lower network costs and 2.4× higher throughput. The increase in throughput, despite SealPIR being more expensive on large databases that XPIR, is the result of XPIR having higher fixed costs. In particular, since Pung's multi-retrieval scheme splits up all messages into many small databases, these fixed costs add up quickly.

In summary, the contributions of this work are:

- SealPIR, a new CPIR library that reduces network costs through a novel query compression and oblivious expansion procedure (§3).
- The introduction of PBC, a new probabilistic data encoding suitable to building multi-query PIR protocols that amortize computational costs (§4.2).
- The design of a PBC construction from a simple technique that we call *hashing in the head* (§4.3).
- The implementation and experimental evaluation of the Pung unobservable communication system [7] using SealPIR and our PBC-based multi-query PIR (§6, §7).

Despite the above contributions, there remains a large performance gap between current CPIR implementations and widespread adoption. Nevertheless, we hope that the content of this paper can help usher a way forward.

2 Background and related work

We begin by giving some background on PIR and existing multi-query proposals that relate to our work.

2.1 Private information retrieval (PIR)

Chor et al. [28] introduced private information retrieval (PIR) to address the following problem: how can a client retrieve an element in a remote database managed by an untrusted server (or set of servers) such that the server does not learn *which* element was retrieved by the client? And whether this can be done more efficiently than the trivial solution where the client simply downloads the entire database from the server and locally selects the desired element? This effort served as the catalyst for two lines of work that remain highly active: *information theoretic* PIR (IT-PIR) and *computational* PIR (CPIR).[1]

In IT-PIR schemes [11, 28, 31, 32, 41] the database is replicated across several non-colluding servers. The client then issues a carefully-crafted query to each server and combines the responses from all the servers locally. IT-PIR schemes have two benefits. First, the servers' computation is relatively inexpensive (an XOR for each entry in the database). Second, the privacy guarantees are information-theoretic, meaning they hold against a computationally-unbounded adversary and avoid cryptographic hardness assumptions. However, systems based on IT-PIR encounter deployment challenges owing to the difficulty of enforcing the non-collusion assumption in practice.

On the other hand, CPIR protocols [4, 19, 22, 35, 40, 52, 53, 56, 57, 76] can be used with a database controlled by a single administrative domain (e.g., a company), under cryptographic hardness assumptions. The drawback is that they are more expensive than IT-PIR protocols as they require the database operator to perform costly cryptographic operations (e.g., modular exponentiation) on each database element. Fortunately, there is a long line of work to improve the resource overheads of CPIR (see [4, 52] for the state-of-the-art), and recent work [4] proposes a construction that achieves, for the first time, plausible (although still high) computational costs. Unfortunately, this construction has high network costs that scale unfavorably with the size of the database (e.g., $O(\sqrt{n})$). Figure 1 depicts a simplified version of this protocol (more specifically it depicts Stern's protocol [70]).

Regardless of which flavor of PIR a system implements, the costs remain undeniably high. As a result, it is hard for systems to support large databases or handle many requests. While addressing the former issue (i.e., large databases) remains elusive—although Section 3 makes progress towards this goal—supporting many concurrent queries is the focus of several proposals. We discuss them next.

2.2 Existing Multi-query PIR schemes

Given PIR's high costs, it is desirable to amortize its costs when serving many requests. Such scenarios include private variants of databases that serve many users concurrently (e.g., Netflix, Spotify), or databases that process a batch of request

[1]These two lines of work are sometimes known as multi-database PIR (for IT-PIR) and single-database PIR (for CPIR).

```
1:  function QUERY($pk, idx, n$)
2:      for $i = 0$ to $n - 1$ do
3:          $c_i \leftarrow \text{Enc}(pk, i == idx ? 1 : 0)$
4:      return $q \leftarrow (pk, c_0, \ldots, c_{n-1})$
5:
6:  function ANSWER($q, DB$)
7:      for $i = 0$ to $|DB| - 1$ do
8:          $a_i \leftarrow DB_i \cdot c_i$            // scalar multiplication
9:      return $a \leftarrow \Sigma_{i=0}^{n-1} a_i$          // homomorphic addition
10:
11: function EXTRACT($sk, a$)
12:     return $\text{Dec}(sk, a)$
```

FIGURE 1—CPIR protocol from Stern [70]. This protocol requires an *additively homomorphic* cryptosystem with algorithms (KeyGen, Enc, Dec), where (pk, sk) is the public and secret key pair generated using KeyGen. We give only the basic protocol and omit all optimizations. The client runs the QUERY and EXTRACT procedures, and the server runs the ANSWER procedure. Each element in DB is assumed to fit inside a single ciphertext. Otherwise each element can be split into ℓ smaller chunks, Lines 8 and 9 can be performed on each chunk individually, and ANSWER returns ℓ ciphertexts instead of one.

from the same user (e.g., Gmail, Slack, bulletin boards). The most general approach to accomplish this goal is given by *batch codes* [47]. In a batch code, the database is encoded such that the server(s) can respond to *any k* requests (from the same user) more cheaply than running k parallel instances of PIR. Unfortunately, existing batch codes increase network costs dramatically; using them in practice is ill-advised. We discuss batch codes and revisit this point in Section 4.1.

Other existing proposals tailor the amortization to particular PIR protocols or particular applications, as we discuss next.

Amortization for particular PIR protocols. Beimel et al. [12] describe two query amortization techniques. The first is based on the observation that queries in many PIR schemes consist of a vector of entries, and answering these queries is equivalent to computing a matrix-vector product (where the product could be over ciphertexts instead of plaintexts, or it could be an XOR operation). By aggregating multiple queries (even from different users), the server's work can be expressed as a product of two matrices. As a result, leveraging sub-cubic matrix multiplication algorithms (e.g., [29, 55, 71]) provides amortization over multiple matrix-vector multiplication instances. This approach is further studied by Lueks and Goldberg [59] in the context of Goldberg's IT-PIR scheme [41].

The second proposal described by Beimel et al. [12] is to perform preprocessing over the database in certain IT-PIR schemes to reduce the cost of future queries. This works well, so recent works [15, 21] employ an analogous preprocessing approach in CPIR schemes. However, making the database accessible by more than a single client under existing CPIR preprocessing schemes requires prohibitively expensive cryptography (a virtual black-box obfuscation primitive [10] instantiated from *indistinguishability obfuscation* [38]).

Groth et al. [43] extend Gentry and Ramzan's [40] CPIR scheme to retrieve k elements at lower amortized network cost by having the client compute k discrete logarithms (with tractable but expensive parameters) on the server's answer. This results in very low network costs, but Gentry and Ramzan's scheme is computationally expensive, and Groth et al.'s extension compounds this issue. With the same goal in mind, Henry et al. [45, 46] and RAID-PIR [31] extend specific PIR protocols. While these techniques result in good amortization, they are only applicable to a certain type of IT-PIR schemes.

Amortization for particular apps. Popcorn [44] pipelines the processing of queries in IT-PIR in order to amortize disk I/O which is a bottleneck for databases with very large files. Pung [7] hybridizes an existing batch code due to Ishai et al. [47] with a probabilistic protocol that exploits the setting of online communication where users that can coordinate a priori (e.g., chat or e-mail). This enables Pung to amortize CPU costs with less network expense than traditional batch codes.

In contrast with the above, our multi-query scheme is agnostic to the particular PIR protocol or application being used. Compared to batch codes [47], our technique has weaker properties (sufficient for most applications) but is significantly more efficient. Compared to Pung's technique, our approach is general, more efficient, and application-independent (§4.2).

3 SealPIR: A new CPIR implementation

XPIR [4] implements Stern's CPIR protocol [70] using the BV lattice-based homomorphic cryptosystem [17]. The basic protocol is given in Figure 1. The key insight in XPIR is that scalar multiplication and homomorphic addition operations can be implemented efficiently in lattice-based cryptosystems using precomputation techniques. To our knowledge, this makes XPIR the only CPIR implementation that is usable in practice.

A major drawback of XPIR (and Stern's protocol) is network costs. In particular, the PIR query sent by the client is large: in the basic scheme, it contains one ciphertext (encrypting 0 or 1) for each entry in an n-element database. This issue is made worse by the underlying lattice cryptosystem: ciphertexts are much larger than plaintexts. Indeed, the *expansion factor* (size ratio of a ciphertext to the largest encodable plaintext) is $F > 5$. While the query size can be reduced to $d\sqrt[d]{n}$ ciphertexts for any positive integer d, it increases the size of the answer exponentially from one ciphertext to F^{d-1} ciphertexts (we explain this in Section 3.3). As a result, regardless of the value of d, the total communication cost will be high. In our experiments, the sweet spot for XPIR is a value of $d = 2$ or 3 (§7.1).

3.1 Compressing queries

At a high level, our goal is to realize the following picture: the client sends one ciphertext containing an encryption of its desired index j to the server, and the server inexpensively evaluates a function EXPAND that outputs n ciphertexts containing an encryption of 0 or 1 (where the j^{th} ciphertext encrypts 1 and others encrypt 0). The server can then use these n ciphertexts as a query and execute the protocol as before (Figure 1, Line 6).

A straw man approach to construct EXPAND is to create a Boolean circuit that computes the following function: "if j matches the encrypted index return 1, else return 0". The server can then evaluate this circuit on the client's ciphertext using a fully homomorphic encryption (FHE) scheme (e.g., BV [17],

operation	CPU cost (ms)	noise growth
addition	0.009	additive
plaintext multiplication	0.285	multiplicative*
multiplication	2.474	multiplicative
substitution (new)	0.229	additive

FIGURE 2—Cost of operations for our parameters ($t \approx 2^{22}$, $q \approx 2^{60}$, $N = 2048$) in SEAL [2]. Every operation increases the *noise* in a ciphertext. Once the noise passes a threshold, the ciphertext cannot be decrypted. For a given computation the security parameters must be large enough to accommodate the expected noise. *While plaintext multiplication yields a multiplicative increase in the noise, the factor is always 1 (i.e., no noise growth) in EXPAND because it is based on the number of non-zero coefficients in the plaintext [23, §6.2].

BGV [16], FV [36]) for values of $j \in [0, n-1]$ to obtain the n ciphertexts. Unfortunately, this approach is very expensive. First, it requires the client to send $\log(n)$ ciphertexts to encrypt each bit of its desired index. Second, it requires n evaluations of a $\log(n)$-depth circuit which contains many homomorphic multiplications—which are the costliest operation in FHE (see Figure 2), and requires the use of larger security parameters.

Instead, we propose a new algorithm to implement EXPAND. It relies on FHE, but it requires the client to send only a single ciphertext. Furthermore, it does not perform any homomorphic multiplications. Note that the underlying cryptosystem used by XPIR (BV [17]) is an FHE scheme, so we could implement our algorithm using that. However, we choose to implement all of SealPIR using the SEAL homomorphic library [2] which implements the Fan-Vercauteren (FV) [36] scheme instead. We make this choice for pragmatic reasons: EXPAND requires the implementation of a new homomorphic group operation, and the SEAL library is a mature code base that is well-documented and already implements many of the necessary building blocks. Below we give some background on FV.

Fan-Vercauteren FHE cryptosystem (FV). In FV, plaintexts are polynomials of degree at most N with integer coefficients modulo t. The polynomials are from the quotient ring $R_t = \mathbb{Z}_t[x]/(x^N + 1)$, where N is a power of 2, and t is the *plaintext modulus* that determines how much data can be packed into a single FV plaintext. Ciphertexts in FV consist of (typically) two polynomials, each in $R_q = \mathbb{Z}_q[x]/(x^N + 1)$. Here q is the *coefficient modulus* that affects how much *noise* a ciphertext can contain, and the security of the cryptosystem. When a ciphertext is created it contains noise that increases as operations are performed on the ciphertext. Once the noise passes a threshold the ciphertext cannot be decrypted. The noise growth of most operations depends heavily on t, so t should be kept small; larger q supports more noise (good) but results in lower security. The expansion factor of this scheme is $F \approx 2\log(q)/\log(t)$. In addition to the standard operations of a cryptosystem (key generation, encryption, decryption), FV also supports homomorphic addition, multiplication, and relinearization (which keeps the number of polynomials in the ciphertext at two); for our purposes we care about the following operations.

- **Addition**: Given ciphertexts c_1 and c_2, which encrypt FV plaintexts $p_1(x), p_2(x) \in R_t$, the operation $c_1 + c_2$ results in a ciphertext that encrypts their sum, $p_1(x) + p_2(x)$.
- **Plaintext multiplication**: Given a ciphertext c that encrypts $p_1(x) \in R_t$, and given a plaintext $p_2(x) \in R_t$, the operation $p_2(x) \cdot c$ results in a ciphertext that encrypts $p_1(x) \cdot p_2(x)$.
- **Substitution (new)**: Given a ciphertext c that encrypts plaintext $p(x) \in R_t$ and an odd integer k, the operation $\mathsf{Sub}(c, k)$ returns an encryption of $p(x^k)$. For instance if c encrypts $p(x) = x^2 + 2x^3$, then $\mathsf{Sub}(c, 3)$ returns an encryption of $p(x^3) = (x^3)^2 + 2(x^3)^3 = x^6 + 2x^9$.

Our implementation of the substitution group operation is related to the plaintext slot permutation technique introduced by Gentry et al. [39, §4.2]. From an algorithmic standpoint, the two are the same. The novelty is in how it interacts with the way in which we encode indices, and how this interaction allows us to design an efficient oblivious query expansion scheme.

Encoding the index. A client that wishes to retrieve the j^{th} element from the server's database using SealPIR generates an FV plaintext that encodes this index. The client does so by representing $j \in [0, n-1]$ as the monomial $x^j \in R_t$. The client then encrypts this plaintext to obtain the SealPIR query $c = Enc(x^j)$, which is then sent to the server.

3.2 Oblivious query expansion

To explain how the server expands $c = Enc(x^j)$ into a vector of n ciphertexts where the j^{th} ciphertext is $Enc(1)$ and all other are $Enc(0)$, we first give a description for $n = 2$.

As described in the previous section, the server receives the query $c = Enc(x^j)$, with $j \in \{0, 1\}$ in this case (since $n = 2$) as the client's desired index. The server first expands c into two ciphertexts $c_0 = c$ and $c_1 = x^{-1} \cdot c$. If $j = 0$, then c_0 is an encryption of 1 (x^0), and c_1 is an encryption of x^{-1}. If on the other hand $j = 1$, then c_0 is an encryption of x, and c_1 is and encryption of 1 ($x^{-1} \cdot x$). The server then computes $d_i = c_i + \mathsf{Sub}(c_i, N+1)$. A useful congruence in the quotient ring R_t is $x^{N+1} \equiv -x \pmod{x^N + 1}$. As a result, a substitution with the integer $N + 1$ transforms the underlying plaintext from $p(x)$ to $p(-x)$. As an example, consider the case for $j = 0$. We have that $d_0 = Enc(1) + \mathsf{Sub}(Enc(1), N+1)$ which is simply $Enc(1) + Enc(1) = Enc(2)$ since there are no instances of x to substitute in the polynomial. d_1 is more interesting. We have that $d_1 = Enc(x^{-1}) + \mathsf{Sub}(Enc(x^{-1}), N+1) = Enc(x^{-1}) + Enc(-x^{-1})$, which is an encryption of 0. We thus have:

$$d_0 = \begin{cases} Enc(2) & \text{if } j = 0 \\ Enc(0) & \text{if } j = 1 \end{cases}$$

$$d_1 = \begin{cases} Enc(0) & \text{if } j = 0 \\ Enc(2) & \text{if } j = 1 \end{cases}$$

Finally, assuming t is odd, we can compute the multiplicative inverse of 2 in $\mathbb{Z}_t$, say α, encode it as the monomial $\alpha \in R_t$, and compute $o_i = \alpha \cdot d_i$. It is the case that o_0 and o_1 contain the desired output of EXPAND: o_j encrypts 1, and o_{1-j} encrypts 0.

We can generalize this approach to any power of 2 as long as $n \leq N$. In cases where n is not a power of 2, we can run the algorithm for the next power of 2, and take the first n output

```
function EXPAND($c = Enc(x^j)$)
    find smallest $m = 2^\ell$ such that $m \geq n$
    $ciphertexts \leftarrow [\,c\,]$
    // each outer loop iteration doubles the number of ciphertexts
    // after each round only one entry encrypts non-zero polynomial
    for $i = 0$ to $\ell - 1$ do
        $expansion \leftarrow [\ ]$
        $substitute \leftarrow (N/2^i) + 1$
        for $k = 0$ to $2^i - 1$ do
            $c_0 \leftarrow ciphertexts[k]$
            $c_1 \leftarrow c_0 \cdot x^{-2^i}$
            $expansion[k] \leftarrow c_0 + \mathsf{Sub}(c_0, substitute)$
            $expansion[k + 2^i] \leftarrow c_1 + \mathsf{Sub}(c_1, substitute)$
        $ciphtertexts \leftarrow expansion$
    // ciphertext at position $j$ encrypts $m$ and all others encrypt 0
    $inverse \leftarrow m^{-1} \pmod t$
    for $i = 0$ to $n - 1$ do
        $o_i \leftarrow ciphertexts[i] \cdot inverse$
    return $output \leftarrow (o_0, \ldots, o_{n-1})$
```

FIGURE 3—Procedure that expands a single ciphertext c encoding an index j into a vector of n ciphertexts, where the j^{th} entry is an encryption of 1, and all other entries are encryptions of 0. We extend the FV scheme [36] with a new group operation Sub (see text for details). Plaintexts are in the polynomial quotient ring $R_t = \mathbb{Z}_t[x]/(X^N + 1)$; the following congruence is useful: $x^N \equiv -1 \pmod{x^N + 1}$. $N \geq n$ is a power of 2, and n is the number of elements in the server's database.

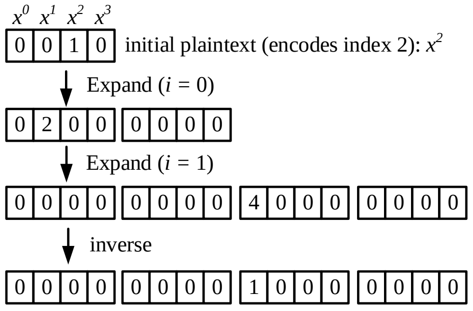


FIGURE 4—Example of EXPAND's effect on the extracted vector's plaintext on each iteration of its outer loop. This example assumes a database with $n = 4$ elements, and an initial query for index 2. Each plaintext is a polynomial represented as an array of coefficients. Note that the server only sees the corresponding ciphertexts (not depicted).

ciphertexts as the client's query. Figure 3 gives the generalized algorithm, and Figure 4 depicts an example for a database of 4 elements. We prove the correctness of EXPAND in Appendix A.1, and bound its noise growth in Appendix A.2.

3.3 Reducing the cost of expansion

One issue of EXPAND is that despite each operation being inexpensive (Figure 2), $O(n)$ operations are needed to extract the n-entry query vector. This is undesirable, since EXPAND could end up being more expensive to the server than computing the answer to a query (see Figure 1, Line 6). We show how to reduce this cost by having the client send multiple ciphertexts.

Stern [70] proposes the following modification to the protocol in Figure 1. Instead of structuring the database DB as an n-entry vector (where each entry is an element), the server structures the database as a $\sqrt{n} \times \sqrt{n}$ matrix M: each cell in M is a different element in DB. The client sends 2 query vectors, v_{row} and v_{col}, each of size $\sqrt{n}$. The vector v_{row} has the encryption of 1 at position r, while v_{col} has the encryption of 1 at position c (where $M[r, c]$ is the client's desired element). The server, upon receiving v_{row} and v_{col}, computes the following matrix-vector product: $A_c = M \cdot v_{col}$, where each multiplication is between a plaintext and a ciphertexts, and additions are on ciphertexts. Observe that A_c is a vector containing the encryption of all entries in column c of M.

The server then performs a similar step using A_c and v_{row}. There is, however, one technical challenge: each entry in A_c is a ciphertext, so it is too big to fit inside another ciphertext (recall that the largest plaintext that can fit in a ciphertext has size $|ciphertext|/F$). To address this, the server splits elements in A_c into F chunks, so A_c can be thought of as a $\sqrt{n}$ by F matrix. The server can now repeat the process as before on the transpose of this matrix: it computes $A_c^T \cdot v_{row}$, to yield a vector of F ciphertexts, which it sends to the client. The client then decrypts all F ciphertexts and combines the result to obtain $Enc(M[r, c])$. The client can then decrypt $Enc(M[r, c])$ to obtain $M[r, c]$—the desired element in DB. This scheme generalizes by structuring the database as a d-dimensional hypercube and having the client send d query vectors of size $\sqrt[d]{n}$. The server then returns F^{d-1} ciphertexts as the response.

We use the above scheme to reduce the computational cost of EXPAND (in contrast, Stern and XPIR use the above technique to reduce network costs by reducing the size of the query vector). Instead of encoding one index, the client encodes d indices (on different ciphertexts), one for each dimension of the database. The server then calls EXPAND on each of the d ciphertexts, and extracts a $\sqrt[d]{n}$-entry vector from each. The server uses the above scheme with the extracted d vectors, which results in the CPU costs of EXPAND being $O(d\sqrt[d]{(n)})$. Of course, this approach has the downside that the PIR response gets larger because of the cryptosystem's expansion factor (F). Specifically, the network cost is d ciphertexts to encode the indices, and F^{d-1} ciphertexts to encode the response. The good news is that for small values of d (2 or 3), this results in major computational savings while still reducing network costs by orders of magnitude over XPIR.

4 New code for amortizing PIR queries

Despite a large body of work on batch codes, we find that most constructions do not focus on PIR amortization. Instead, they target load balancing in distributed storage systems [64, 67] and network switches [75], which have different requirements. If we use these codes to amortize queries in PIR, we would incur prohibitive network costs. Consequently, we depart from this line of work. Our observation is that certain guarantees of batch codes are not necessary for many PIR-backed systems. Relaxing those guarantees leads to constructions that are not only asymptotically better, but also concretely efficient—without compromising the functionality of our target systems. We first provide a more detailed description of batch codes, highlight the sources of overhead, and then introduce our construction.

4.1 Batch codes and their cost

We wish to keep variable naming consistent with the existing batch code literature, so the variables in this section have a different meaning from those of Section 3. We use the shorthand $[n]$ to denote the set of non-negative integers up to (but not including) n. That is, $[n] = \{0, \dots, n-1\}$.

Definition 1 (**batch code**). A (n, N, k, m, t)-batch code C takes as input a collection DB consisting of n elements from a finite *source alphabet* S (e.g., the set of 256-bit strings $\{0,1\}^{256}$, a finite field $\mathbb{F}_p$, the library of Netflix movies) and produces N codewords in a finite *code alphabet* A (potentially different from S) that are distributed among m buckets. Specifically, C is defined as $C : S^n \to (A^{N_0}, \dots, A^{N_{m-1}})$, where $N = \Sigma_{i=0}^{m-1} N_i \geq n$, and $N_i \geq 0$ for all $i \in [m]$. The goal of these codes is to ensure that any k elements from DB can be retrieved from the m buckets by fetching at most t codewords from each of them. In other words, C provides *availability* to the input collection [66].

Availability (to the input elements) in the context of a code C means that each of the n elements in DB can be recovered from multiple disjoint sets of codewords in $C(DB)$. For example, if a system encodes some input collection DB with a code C that provides k-availability, then up to k concurrent users can access *any* one element in DB by reading different codewords in $C(DB)$. This is a useful load balancing property that has significant applications in distributed storage [66]. Batch codes actually provide a stronger version of availability than the above: they provide availability not to individual elements of DB, but to sets (or multisets) of elements in DB. This stronger property is crucial for amortizing the cost of a batch of requests in PIR.

Source of costs. Figure 5 depicts the relationship between the number of codewords (N) and the number of buckets m, as a function of the database size (n) and the batch size (k) for several constructions. All of the codes achieve $t = 1$ (i.e., at most one codeword is retrieved per bucket). A key metric of a code is its *rate*, which is defined as the ratio between the number of input elements and the number of codewords (i.e., n/N). In general, a higher rate leads to increased amortization of CPU costs in multi-query PIR. For instance, a rate of 1 (which is the highest achievable) implies that the server can answer k requests while paying the computational cost of processing a single request. Another important metric is the number of buckets (m). In PIR, a client has to fetch an element from every bucket to maintain privacy (§5). Consequently, a higher m leads to more queries and answers, which in turn results in higher network costs. As Figure 5 depicts, there is a clear tension between achieving a good rate and keeping the number of buckets low.

To relieve some of this tension we describe a different way to build codes that provide weaker (but still useful) guarantees.

4.2 Probabilistic batch codes (PBC)

Batch codes have many excellent properties, but existing constructions have either very low rate or introduce excessive network overheads. We now discuss a variant that strikes a balance.

A *probabilistic batch code* (PBC) differs from a traditional batch code in that it fails to uphold the availability guarantee with small probability p. That is, a collection encoded with a PBC may, with probability p, result in less than k disjoint sets of codewords for some sets of k input elements. Consequently, an application that requires accessing any k elements (e.g., multi-query PIR) might be unable to do so by reading each bucket at most t times. In practice, this is not really an issue: our construction has parameters that result in roughly 1 in a trillion queries failing, which we think is a sufficiently rare occurrence. Furthermore, as we discuss in Section 5, this is an easy failure case to address in multi-query PIR since a client learns whether all of its queries will succeed (or not) *before* issuing them.

In addition to the above relaxation, the specific PBC construction that we describe in this paper introduces another simplification (this is not fundamental to PBCs). Our specific construction provides availability only to sets of elements in the input collection (not to multisets). We emphasize that this is not a limitation for us. While multisets are common in non-PIR applications of batch codes (e.g., distributed storage, network switches), this is not the case for multi-query PIR: if a client truly wishes to retrieve multiple copies of the same element, this can be trivially done without multiset support. Section 5 discusses this further.

Definition 2 (**PBC**). A (n, N, k, m, t, p)-PBC is a batch code (Definition 1) with parameters (n, N, k, m, t) that provides k-availability to sets of k input elements from a source alphabet S, except with probability p. A PBC is given by three polynomial-time algorithms (Encode, $\mathsf{GenSchedule}$, Decode), defined as follows:

- *buckets* $\leftarrow \mathsf{Encode}(DB)$: Given an n-element collection $DB \in S^n$, outputs an m-tuple of buckets $\in (A^{N_0}, \dots, A^{N_{m-1}})$, where $m \geq k$, each bucket contains zero or more codewords from the code alphabet A (i.e., $\forall_i, N_i \geq 0$), and the total number of codewords across all buckets is $N = \Sigma_{i=0}^{m-1} N_i \geq n$.
- $\{\sigma, \bot\} \leftarrow \mathsf{GenSchedule}(I)$: Given a set of k indices $I \in [n]^k$ corresponding to the positions of the desired elements in DB, outputs a *schedule* $\sigma : [n]^k \to [N]^+$ that describes, for each index $\in I$, the position of one or more codewords from across the m buckets that should be retrieved.[2] $\mathsf{GenSchedule}$ outputs $\bot$ if it cannot produce a schedule where each index $\in I$ is associated with at least one codeword position $\in [N]$, and where no more than t codewords fall within the same bucket.
- *element* $\leftarrow \mathsf{Decode}(W)$: Given a set of codewords $W \in A^+$, outputs the corresponding element $\in S$.

Definition 3 (**(k, p)-availability**). A probabilistic batch code $C = (\mathsf{Encode}, \mathsf{GenSchedule}, \mathsf{Decode})$ provides (k, p)-availability to the elements of an n-element collection DB, if for any set of k indices $I \in [n]^k$:

$$\begin{aligned}\Pr[\ & \textit{buckets} \leftarrow \mathsf{Encode}(DB)\,; \\ & \sigma \leftarrow \mathsf{GenSchedule}(I)\,; \\ & \sigma \neq \bot \wedge \forall_i \in I, \mathsf{Decode}(\textit{buckets}[\sigma[i]]) = DB[i] \\ ] & = 1 - p.\end{aligned}$$

[2]We let the position of the first codeword in bucket 0 be 0, the position of the last codeword in bucket $m-1$ be $N-1$, and the position of the i^{th} codeword in bucket j be $\Sigma_{\ell=0}^{j-1} N_\ell + i$.

batch code	**codewords (N)**	**buckets (m)**	**probability of failure (p)**
straw man (k-way replication)	kn	k	0
subcube ($\ell \geq 2$) [47, §3.2]	$n \cdot (\ell/\ell+1)^{\log_2(k)}$	$(\ell+1)^{\log_2(k)}$	0
combinatorial ($\binom{\beta}{k-1} \leq n/(k-1)$) [64, §2.2]	$kn - (k-1) \cdot \binom{\beta}{k-1}$	β	0
Balbuena graphs [67, §IV.A]	$2(k^3 - k \cdot \lceil n/(k^3-k) \rceil)$	$2(k^3-k)$	0
Pung hybrid* [7, §4.4]	$4.5n$	$9k$	$\approx 2^{-20}$
probabilistic batch codes (this work, §4.2)			
3-way cuckoo hashing in the head (§4.5)	$3n$	$1.3k$	$\approx 2^{-40}$

FIGURE 5—Cost of existing batch codes and the probabilistic batch code (PBC) construction given in Section 4.5. n indicates the number of elements in the database DB. k gives the number of elements that can be retrieved from DB by querying each codeword in $C(DB)$ at most once, where C is the batch code. Building a multi-query PIR scheme from any of the above constructions leads to computational costs to the server(s) proportional to $O(N)$, and network communication proportional to $O(m)$. We list batch codes that have explicit constructions and can amortize CPU costs for multi-query PIR. Other batch codes have been proposed (e.g., [58, 68, 69, 74]) but they either have no known constructions, or they seek additional properties (e.g., tolerate data erasures, optimize for the case where $n = m$, support multisets) that introduce structure or costs that makes them a poor fit for multi-query PIR. *The scheme in Pung is neither a batch code nor a PBC since it relies on clients replicating the data to buckets (rather than the server). It is, however, straightforward to port Pung's allocation logic via hashing in the head (§4.4) to construct a PBC.

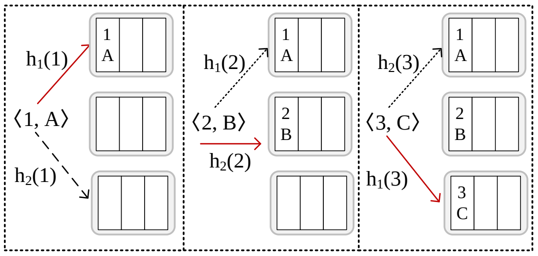


FIGURE 6—Logic for two-choice hashing [9] when allocating three key-value tuples to buckets: $\langle 1, A\rangle$, $\langle 2, B\rangle$, $\langle 3, C\rangle$. Tuples are inserted into the bucket least full. Arrows represent the choices for each tuple (based on different hashes of the tuple's key). The red solid arrow indicates the chosen mapping.

In the next few Sections we give an efficient PBC construction that provides (k, p)-availability while being concretely efficient. The main idea behind our construction is simple. Batch codes are designed to spread out requests to balance the load. Relatedly, many data structures, systems, and networking applications use different variants of hashing—consistent [49], asymmetric [73], weighted [72], multi-choice [9, 61], cuckoo [8, 63], and others [18, 34]—to achieve the same goal. While there is no obvious way to use these hashing schemes to implement multi-query PIR directly, we can do it indirectly: we first build a PBC from a simple technique that we call *hashing in the head*,[3] and then use the PBC to implement multi-query PIR (§5). We detail this process in the next few sections.

4.3 Randomized load balancing

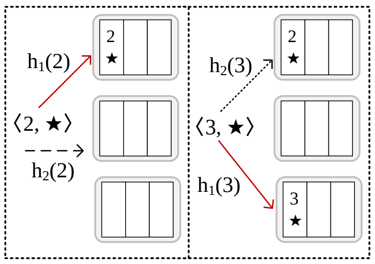
A common use case for (non-cryptographic) hash functions is to build efficient data structures such as hash tables. In a hash table, the insert procedure consists of computing one or more hash functions on the key of the element being inserted. Each application of a hash function returns an index into an array of buckets in the hash table. The item is then placed into one of these buckets following an allocation strategy. For example, in left-hashing [73] the item is placed on the leftmost bucket. In multi-choice hashing [9, 61], the item is placed in the bucket least full among d candidate buckets. In Cuckoo hashing [63], items are moved around following the Cuckoo hashing algorithm (we explain this algorithm in Section 4.5).

An ideal allocation strategy results in each key being assigned to a unique bucket (since this lowers the cost of lookup). However, in practice there are *collisions* where multiple keys map to the same bucket. To look up an item by its key, one computes the different hash functions that were used during insertion on the key to obtain the list of buckets in which the item could have been placed; one then scans each of those buckets for the desired item. The insertion process for 2-choice hashing is depicted in Figure 6.

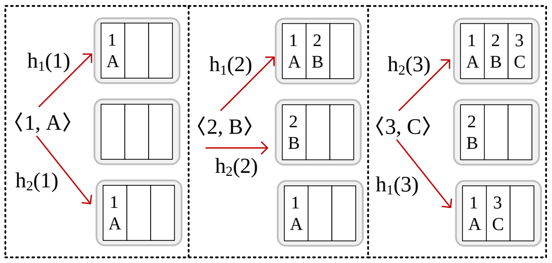
Abstract problem: balls and bins. In the above example, hashing is used to solve an instance of the classic n balls and m bins problem, which arises during insertion. In the above instance, the items to be inserted into a hash table are the n balls, and the buckets in the hash table are the m bins; hashing a key to a bucket approximates an independent and uniform random assignment of a ball to a bin. The number of collisions in a bucket is the load of a bin, and the highest load across all bins is the *max load*. In the worst case, the max load is n/d (all balls map to the same d candidate buckets), but there are more useful bounds that hold with high probability. The goal of the hashing techniques mentioned above is to produce an allocation of balls to bins that reduces this bound (i.e., balance the load across bins).

Interestingly, if we examine other scenarios abstracted by the balls and bins problem, a pattern becomes clear: the allocation strategy is always executed during data placement. In the hash table example, the allocation strategy determines where to insert an element. In the context of a transport protocol [50], the allocation strategy dictates on which path to send a packet. In the context of a job scheduler [62], the allocation strategy selects the server on which to run a task. The result is that the load balancing effect is achieved at the time of "data placement". However, to build a PBC, we must do it at the time of "data retrieval". Hashing in the head achieves this.

[3]The phrase "in the head" was introduced by Ishai et al. [48] to describe the action of an entity who simulates the execution of a protocol (multiparty computation in their case). We borrow this phrase.



(a) consumer's simulation



(b) producer's allocation

FIGURE 7—Example of two-choice hashing in the head. (a) shows the consumer's simulation when inserting two tuples $\langle 2, \star\rangle$, $\langle 3, \star\rangle$. The $\star$ indicates that the value is not known, so an arbitrary value is used. (b) shows a modification to two-choice hashing where the producer stores the tuple in all possible choices. This ensures that the final allocation is always compatible with the consumer's simulation.

4.4 Hashing in the head

We start by introducing two principals: the *producer* and the *consumer*. The producer holds a collection of n elements where each element is a key-value tuple. It is in charge of data placement: taking each of the n elements and placing them into m buckets based on their keys (e.g., insert procedure in a hash table). The consumer holds a set of k keys ($k \leq n$), and is in charge of data retrieval: it fetches items by their key from the buckets that were populated by the producer (e.g., lookup procedure in a hash table). The goal is for the consumer to get all k items by probing each bucket as few times as possible. That is, the consumer has an instance of a k balls and m bins problem, and the goal is to reduce its max load. The challenge is that any clever allocation chosen by the consumer must be compatible with the actions of the producer (who populates the buckets). We describe a solution below.

The consumer starts by imagining in its head that it is a producer with a collection of k elements. In particular, the consumer converts its k keys into k key-value tuples by assigning a dummy value to each key (since it does not know actual values). In this simulation, the consumer follows a specific allocation strategy (e.g., 2-choice hashing, cuckoo hashing) and populates the m buckets accordingly. The result is an allocation that balances the load of the k elements among the m buckets (as we discussed in Section 4.3). The consumer then ends its simulation and uses the resulting allocation to fetch the k elements from the buckets that were populated by the real producer.

Of course, the above only works if the consumer's allocation strategy is compatible with the actions of the producer. Without additional mechanisms, this might not be the case. One reason is that the consumer's simulation is acting on k elements whereas the real producer is acting on n. If the allocation strategy being used (by the consumer and the producer) is randomized or depends on prior choices, the allocations will be different. We illustrate this with an example.

Observe that if a producer generates the allocation in Figure 6 it would not be compatible with the consumer's simulation in Figure 7a despite both entities using the same strategy (since the producers places the item under key "2" in the middle bucket, but the consumer's simulation maps it to the top bucket). To address this, we employ a simple solution: the producer follows the same allocation strategy as the consumer's simulation (e.g., 2-choice hashing) on its n elements but stores the elements in all candidate buckets. That is, whenever the allocation strategy chooses one among d candidate buckets to store an element, the producer stores the element in all d buckets. This ensures that regardless of which k elements are part of the consumer's simulation or which non-deterministic choices the strategy makes, the allocations are always compatible (Figure 7b). Of course this means that the producer is replicating elements, which defeats the point of load balancing. However, this is not an issue, since PBCs only need load balancing during data retrieval.

4.5 A PBC from cuckoo hashing in the head

We give a construction that uses cuckoo hashing [63] to allocate balls to bins. However, the same method can be used with other algorithms (e.g., multi-choice Greedy [9], LocalSearch [51]) to obtain different parameters. We give a brief summary of Cuckoo hashing's allocation strategy below.

Cuckoo hashing algorithm. Given n balls, m buckets, and d independent hash functions $h_0, \ldots, h_{d-1}$, compute d candidate buckets for each ball by applying the d hash functions: $h_i(b) \bmod m$. For each ball b, place b in any empty candidate bucket. If none of the d candidate buckets are empty, select one of the candidate buckets at random, remove the ball currently in that bucket (b_{old}), place b in the bucket, and re-insert b_{old} as before. If re-inserting b_{old} causes another ball to be removed, this process continues recursively for a maximum number of rounds. If this maximum number is reached, the algorithm aborts.

Construction. Let H be an instance (producer, consumer) of hashing in the head where the allocation strategy is Cuckoo hashing with d hash functions and m bins (we discuss concrete values for d and m later in this section). We construct a (n, N, k, m, t, p)-PBC as follows.

Encode(DB). Given a collection DB of n elements, follow H's producer algorithm to allocate the n elements to the m buckets. This results in $N = dn$ total elements distributed across the m buckets (each bucket may contain a different number of elements). Return the buckets.

GenSchedule(I). Given a set of indices $I \in [n]^k$, follow H's consumer algorithm to allocate the k indices to the m buckets. This yields the allocation $\phi : [n]^k \rightarrow [m]^k$, which states, for each index in I, the bucket on which it should be placed. To return the actual schedule for I, σ, we need to go one step further. We need not only the bucket assignment (given by ϕ), but also the index within the bucket (or correspondingly the index in $[N]$). To obtain this mapping we introduce an oracle $\mathcal{O} : [n] \times [m] \rightarrow [N]$, that outputs this mapping for us. There are many ways to obtain or construct this oracle, and we discuss them later in this section. This gives us a straightforward way to construct σ, namely $\forall_i \in I, \sigma[i] = \mathcal{O}(i, \phi(i))$. If more than t indices in I map to the same bucket, return $\perp$ instead.

Decode(W). Since Encode performs only replication, the source and code alphabets (S and A) are the same, and all codewords are input elements (in DB) and require no decoding. Furthermore, σ, which is returned by GenSchedule, has only one entry for each index. As a result, W contains only one codeword. Decode returns W.

Concrete parameters. Analyzing the exact failure probability of Cuckoo hashing remains an open problem (see [33] for the most recent progress). However, several works [24, 65] have estimated this probability empirically for different parameter configurations. Following the analysis in [24, §4.2], we choose $d = 3$ and $m = 1.3k$. In this setting, the failure probability is estimated to be $p \approx 2^{-40}$. This means that, assuming the mapping from indices to buckets is pseudorandom, GenSchedule(I) returns $\perp$ with probability p for any chosen set of indices I. The result is a PBC that provides (k, p)-availability. Figure 5 compares this result with existing batch code constructions and the scheme proposed in Pung [7, §4.4].

Constructing the oracle $\mathcal{O}$. There are several ways that the client can construct $\mathcal{O}$. In the non-private setting, $\mathcal{O}$ could simply return the first element in the bucket (i.e., for bucket j it would return $\Sigma_{\ell=0}^{j-1} N_\ell$); the client could use this index to determine the corresponding bucket, and query this bucket by sending the corresponding index $i \in [n]$. In PIR, the simplest solution is to obtain the mapping from $[n] \rightarrow [N]^d$ directly from the server. While this might sound unreasonable, observe that PIR has an implicit assumption that the client knows the index $\in [n]$ of the desired element. The client could use the same technique to obtain the corresponding d indices in $\in [N]$. In Pung, clients obtain this mapping in a succinct Bloom filter [13].

Another option is for the client to fetch elements using PIR not by index but by their key. As such, $\mathcal{O}$ could return the bucket id (as in the non-private setting) and the client could issue a PIR-by-keywords [27] query to this bucket. One last option is for the clients to construct $\mathcal{O}$ directly. This requires the server to share with clients its source of randomness (e.g., a PRF seed). Clients can then simulate the server's encoding procedure on a database of n dummy elements (replicating each element into d candidate buckets), which yields $\mathcal{O}$. Furthermore, this process is incremental: if a client has $\mathcal{O}$ for an n-element database, it can construct $\mathcal{O}$ for a database with $n + 1$ elements by simulating the insertion of the last element.

5 Multi-query PIR from PBCs

We now discuss how to build a multi-query PIR scheme from any PBC. To simplify our description, we first recall a basic transformation of batch codes and PBCs.

Lemma 1 (Trading time for space)**.** For any value of $t \geq 1$, a (n, N, k, m, t)-batch code implies a $(n, tN, k, tm, 1)$-batch code [47, Lemma 2.4].

```
function SETUP($DB$)
    $buckets \leftarrow$ Encode($DB$)

function MULTIQUERY($pk, I, N$)
    $\sigma \leftarrow$ GenSchedule($I$)
    if $\sigma \neq \perp$ then
        // get an element for each bucket
        // pick a random index if the bucket is not used in $\sigma$
        for $b = 0$ to $m - 1$ do
            $\sigma_b \leftarrow$ index for bucket $b$ (based on $\sigma$)
            $q_b \leftarrow$ QUERY($pk, \sigma_b, N_b$)            // see Fig. 1, Line 1
        return $q \leftarrow (q_0, \ldots, q_{m-1})$
    else
        Deal with failure (see §5)

function MULTIANSWER($q, buckets$)
    for $b = 0$ to $m - 1$ do
        $a_b \leftarrow$ ANSWER($q_b, buckets[b]$)            // see Fig. 1, Line 6
    return $a \leftarrow (a_0, \ldots, a_{m-1})$

function MULTIEXTRACT($sk, a, I, \sigma$)
    // extract the codewords from the provided PIR answers into $cw$
    for $b = 0$ to $m - 1$ do
        $cw_b \leftarrow$ EXTRACT($sk, a_b$)            // see Fig. 1, Line 11
    // select codewords from $cw$ that are relevant to each index in $I$
    for $i = 0$ in $k - 1$ do
        $W \leftarrow$ codewords from $cw$ (based on $\sigma[I_i]$)
        $e_i \leftarrow$ Decode($W$)
    return $(e_0, \ldots, e_{k-1})$
```

FIGURE 8—Multi-query CPIR protocol based on a CPIR protocol and a PBC (Encode, GenSchedule, Decode). I is the set of k desired indices and N is the set of bucket lengths. As in Figure 1, this protocol requires an additively homomorphic cryptosystem with algorithms (KeyGen, Enc, Dec), where (pk, sk) are generated from KeyGen.

The above simply states that instead of using a batch code that requires reading at most t elements from each bucket, one can simply replicate each bucket t times (leading to tm buckets and tN codewords). The result is a batch code that requires reading at most 1 element from each bucket. For the remainder of this section we use the above lemma to convert any PBC. This guarantees that GenSchedule returns a schedule that specifies the position of at most one codeword for any bucket (or returns $\perp$ if no such schedule exist).

Construction. We give the pseudocode for a PBC-based multi-query CPIR scheme in Figure 8. At a high level, the server encodes its database by calling the PBC's Encode procedure. This produces a set of buckets, each of which can be treated as an independent database on which clients can perform PIR. A client who wishes to retrieve elements at indices $I = \{i_0, \ldots, i_{k-1}\}$ can then locally call GenSchedule(I) to obtain a schedule σ. This schedule states, for each index, which element to retrieve from each bucket using PIR. Because of the semantics of GenSchedule (and our application of Lemma 1 to the PBC) it is guaranteed that no bucket is queried more than once. As a result, the client can run one instance of PIR on each bucket, querying for the element indicated in the schedule. If the client has nothing to retrieve from a given bucket (i.e., the schedule does not specify any index for the particular bucket), then the client simply queries a random index for that bucket.

Handling multisets. In Section 4.2 we stated that our PBC construction does not handle multisets. One reason for this is the added costs. Another reason is that multisets can be trivially supported in multi-query PIR. Suppose that a client wishes to retrieve the element at position 15 k times. The client can construct the index set $I = \{15, i_1, i_2, \ldots, i_{k-1}\}$, where all other indices are chosen at random. Once the client retrieves the element at position 15 it can replicate it k times and it can throw away all other elements that were retrieved.

Dealing with failures in the schedule. If the PBC being used provides (k, p)-availability for $p > 0$, then it is possible that for a client's choice of indices, $\sigma = \perp$. In this case, the client is unable to fetch all k elements that it wishes to retrieve privately. However, notice that the client learns of this fact *before* issuing any PIR query (see Figure 8, Line 6). As a result, the client has a few options. First, the client can adjust its set of indices (i.e., choose different elements to retrieve). This is possible in applications where the client needs to retrieve more than a batch of k items. Second, the client can retrieve a subset of the elements. In a messaging application, this would mean that the client would not retrieve all unread messages. In many cases, this is acceptable since messages are not ephemeral so the client can try again at a later time (presumably with a new set of indices). Lastly, the client could fail silently. Which of these strategies is taken depends on the application.

6 Implementation

We build SealPIR by implementing Stern's protocol [70] on top of version 2.3 of the SEAL homomorphic encryption library [2]. This required around 1K lines of C++. We also implement some of the preprocessing optimizations proposed by XPIR [4]. The most difficult component to implement was EXPAND (Figure 3), which required the introduction of the substitution homomorphic operation (§3.1). We implement this group operation in SEAL by porting the plaintext slot permutation algorithm from Gentry et al. [39, §4.2]. This required 400 lines of C++, and we have submitted our changes for others to use.

We also implement a small optimization for EXPAND that is specific to the FV cryptosystem. The optimization is as follows. In FV, an encryption of $2^\ell \pmod{2^y}$, for $y \geq \ell$, is equivalent to an encryption of $1 \pmod{2^{y-\ell}}$. Observe that in Lines 16–18 of Figure 3, EXPAND multiplies the n ciphertexts by the inverse of m where $m = 2^\ell$. Instead, we switch the plaintext modulus of the n ciphertexts from $t = 2^y$ to $2^{y-\ell}$, which allows us to avoid the plaintext multiplications and the inversion, and reduces the noise growth of EXPAND. The result is $n-1$ ciphertexts encoding 0, and one ciphertext encoding 1, as we expect. This also allows us to use any value of t, not just an odd integer (since we no longer need to invert m).

SealPIR exposes the API described in Figure 1 to applications, in addition to a one time setup operation where the server preprocesses the database to encode all elements as FV plaintexts (XPIR exposes a similar API, including its own preprocessing). One difference with XPIR is that the substitution operation in SealPIR requires a special cryptographic key per client. However, a client can use this key across any number of requests and the key is small (about two ciphertexts in size).

We have also implemented *mPIR*, a multi-query PIR library based on PBCs. mPIR implements 5 different PBC constructions: each is a different instance of hashing in the head (§4.4) with a different allocation strategy (e.g., two-choice hashing, Cuckoo hashing, the Hybrid allocation scheme in Pung [7]). This library works transparently on top of both XPIR and SealPIR, and is written in 1,700 lines of Rust. It uses SHA-256 with varying counters to implement the different hash functions.

To get a sense of the end-to-end benefits that SealPIR and mPIR provide to actual applications, we modify the available implementation of the Pung unobservable communication system [1]. Pung is a chat/email service that allows users to exchange messages without leaking any metadata (e.g., who they are talking to, or when they are communicating). We chose Pung because it uses XPIR to achieve its privacy guarantees, and because it also relies on multi-query PIR to allow clients to send or receive multiple messages simultaneously. Consequently, we can switch Pung's PIR engine from XPIR to SealPIR, and we can replace Pung's custom multi-query PIR scheme with mPIR. Since both libraries are orthogonal (though complementary), we can evaluate the impact that each one has on Pung.

7 Evaluation

Our evaluation discusses four questions. First, what is the concrete performance and what are the network costs of SealPIR, and how do they compare to XPIR (§7.1). Second, how efficient are the basic operations (e.g., encode, decode) of the different PBCs that we implement as part of mPIR (§7.2). Third, how does mPIR's Cuckoo hashing PBC variant compare to the multi-query PIR scheme in Pung (§7.3). Last, what are the concrete benefits of using SealPIR and mPIR in Pung (§7.4).

Method and experimental setup. We run all of our experiments on a cluster of Microsoft Azure H16 instances (16-core 3.6 GHz Intel Xeon E5-2667 and 112 GB RAM) running Ubuntu 16.04. We compile all our code with Rust's nightly version 1.22. For XPIR, we use the available source code [5]. We use new security parameters following XPIR's estimator [3]: we set the degree of ciphertexts' polynomials to 2048, and the size of the coefficients to 60 bits (N and q in the terminology of Section 3). Note that the XPIR parameters used for experiments in Pung and other systems (1024 and 60-bit, respectively) no longer provide 80-bit security according to Albrecht et al.'s LWE security estimator [6]. We report all network costs measured at the application layer. We run each experiment 10 times and report averages from those 10 trials. Standard deviations are less than 10% of the reported means.

7.1 Cost and performance of SealPIR

To evaluate SealPIR, we run a series of microbenchmarks to measure the time to generate, expand, and answer a query, the time to extract the response, and the time to preprocess the database. We study several database and element sizes, and repeat the same experiment for XPIR. Figure 9 gives the results.

	XPIR			SealPIR		
database size (n)	8,192	32,768	131,072	8,192	32,768	131,072
client-side CPU costs						
QUERY (288 B elements)	7.36 ms	14.34 ms	33.15 ms	3.42 ms	3.43 ms	3.43 ms
QUERY (1 KB elements)	7.25 ms	15.42 ms	33.77 ms	3.41 ms	3.44 ms	3.44 ms
EXTRACT (288 B elements)	0.43 ms	0.41 ms	0.39 ms	4.92 ms	5.04 ms	5.73 ms
EXTRACT (1 KB elements)	0.80 ms	0.80 ms	0.84 ms	4.97 ms	5.69 ms	6.46 ms
server-side CPU costs						
SETUP (288 B elements)	47.33 ms	205.77 ms	823.74 ms	200.20 ms	288.68 ms	657.01 ms
SETUP (1 KB elements)	58.65 ms	332.38 ms	956.63 ms	275.66 ms	593.14 ms	1,867.03 ms
EXPAND (288 B elements)	———	———	———	35.64 ms	73.39 ms	147.88 ms
EXPAND (1 KB elements)	———	———	———	68.13 ms	138.51 ms	281.27 ms
ANSWER (288 B elements)	73.85 ms	159.54 ms	387.17 ms	77.81 ms	203.2 ms	625.77 ms
ANSWER (1 KB elements)	83.77 ms	170.03 ms	399.57 ms	184.82 ms	561.78 ms	1,845.01 ms
network costs						
query (288 B elements)	2,048 KB	4,096 KB	8,192 KB	64 KB	64 KB	64 KB
query (1 KB elements)	2,048 KB	4,096 KB	8,192 KB	64 KB	64 KB	64 KB
answer (288 B elements)	256 KB	256 KB	256 KB	384 KB	384 KB	448 KB
answer (1 KB elements)	512 KB	512 KB	512 KB	384 KB	448 KB	512 KB

FIGURE 9—Operation cost for XPIR and SealPIR under varying collection sizes (n) and element sizes (288 bytes and 1 KB). We use parameters $N = 2048$, $q = 2^{60} - 2^{18} + 1$, $d = 2$, $\alpha = 8$ or 14 for XPIR and SealPIR (see text for details). SealPIR's plaintext modulus is $t = 2^{22}$.

We find that the computational costs of the client are lower under SealPIR than under XPIR. This is because: with SealPIR, the client only needs to generate two ciphertexts as a query (one ciphertext encodes the row and the other the column of the desired element, see Section 3.3), rather than $2\sqrt{n}$ ciphertexts as in XPIR.

On the other hand, SealPIR introduces significant CPU overheads to the server. These costs stem from the expansion procedure and specifics of our PIR implementation. The expansion procedure's CPU cost ranges between 38.2% to 81.3% that of computing on the query vector directly using XPIR. While this is undeniably high, we think this constitutes an excellent trade-off given the significant network savings (we discuss them below). When it comes to the cost of answering a query (after it has been expanded), SealPIR is $1.05\times$ to $4.6\times$ more expensive than XPIR. The reason for this is that SealPIR is implemented using SEAL, which is a general-purpose FHE library. As such, operations such as multiplying a ciphertext that encrypts a 0 or 1 by some scalar (representing an element), called *Absorb* in XPIR, are not as optimized in SEAL as they are in XPIR—where this is the primary operation. However, unlike the costs introduced by EXPAND, these overheads are not fundamental and can be lowered with further engineering. Finally, as we discuss in Section 3, EXPAND is not specific to SEAL or its underlying FHE scheme, and can be implemented for XPIR.

For network costs, we see a significant benefit owing to SealPIR's EXPAND procedure (§3.2). For the larger databases, the network cost reductions are of $128\times$. In these experiments we use $d = 2$, meaning that we structure the database as a matrix, as described in Section 3.3, for both XPIR and SealPIR. For much larger databases than the ones we study, a larger value of d would yield better results.

We also use $\alpha = 14$ for 288 byte elements and $\alpha = 8$ for 1 KB elements, meaning that we treat α elements as a single logical element (of size $\alpha\times$), thereby reducing the number of elements in the database by a factor of α. A larger α helps reduce the size of the query vector for XPIR, and the CPU costs for both XPIR and SealPIR; increasing α comes at the expense of larger answers (since the database elements are logically "larger"). Taken to the extreme, $\alpha = n$, the client is simply downloading the entire database as a single logical element, which results in negligible query size and CPU costs, but defeats the purpose of using PIR.

PBC scheme	Encode	GenSchedule	Decode
k-way replication	22.5 ms	5.8 μs	0.1 μs
sharding	52.1 ms	112.8 μs	0.3 μs
2-choice hashing	103.6 ms	212.9 μs	0.2 μs
Pung Hybrid	101.8 ms	42.3 μs	1.2 μs
Cuckoo hashing	154.1 ms	319.2 μs	0.15 μs

FIGURE 10—Cost of operations for the five PBCs implemented as part of mPIR. The collection size (n) is 131,072, and the batch size (k) is 64. Each element in the collection is 1 KB. k-way replication simply replicates the n balls into k-bins during the producer's allocation, and picks a different bin for the k balls during the consumer's simulation. Sharding maps every ball to single bin during the producer's allocation, and the consumer uses a hash function during its simulation (this variant has a high failure rate, which we improve by fetching many elements from each bucket and applying Lemma 1, see Section 5).

7.2 Cost of PBC variants

We have implemented five PBCs with different allocation algorithms using hashing in the head. Our goal is to show that all of them admit efficient encoding and decoding procedures. For the purpose of building a multi-query PIR scheme, we wish to select a PBC variant with a high rate and low number of buckets. Our hypothesis is that the higher the rate and lower the number of buckets, the more expensive it is to encode and generate a schedule for that PBC.

	single-query	Pung's multi-retrieval			mPIR (Cuckoo hashing)		
batch size (k)	1	16	64	256	16	64	256
client-side CPU costs							
MultiQuery (288 B elements)	33.15 ms	97.51 ms	46.93 ms	15.99 ms	28.44 ms	15.39 ms	6.02 ms
MultiQuery (1 KB elements)	30.77 ms	114.15 ms	56.10 ms	6.22 ms	159.09 ms	26.68 ms	8.83 ms
MultiExtract (288 B elements)	0.39 ms	3.43 ms	3.12 ms	3.08 ms	0.48 ms	0.51 ms	0.48 ms
MultiExtract (1 KB elements)	0.74 ms	6.40 ms	6.26 ms	21.72 ms	0.94 ms	0.94 ms	0.90 ms
server-side CPU costs							
MultiSetup (288 B elements)	823.74 ms	289.43 ms	99.53 ms	65.44 ms	169.22 ms	49.31 ms	17.50 ms
MultiSetup (1 KB elements)	956.63 ms	630.47 ms	193.61 ms	65.14 ms	438.56 ms	121.12 ms	33.28 ms
MultiAnswer (288 B elements)	387.17 ms	295.80 ms	216.25 ms	114.81 ms	215.04 ms	113.65 ms	57.97 ms
MultiAnswer (1 KB elements)	399.57 ms	359.06 ms	282.51 ms	120.86 ms	297.49 ms	199.52 ms	84.13 ms

FIGURE 11—Per-request (amortized) CPU cost of two multi-query PIR schemes on a database consisting of 131,072 elements, with varying batch and element sizes. The schemes are Pung's multi-retrieval protocol and mPIR, which is based on PBCs (Cuckoo variant). The second column gives the cost of retrieving a single element (no amortization). The underlying PIR library is XPIR, and we use parameters $d = 2$, and $\alpha = 14$ (for 288 B elements) or 8 (for 1 KB elements).

To test this hypothesis we create a collection with 131K elements, each of which is 1 KB, and encode the collection with the different PBCs for a batch size of $k = 64$. We then measure the time to encode, decode, and generate a schedule. We also experiment with other element and collections sizes and find that while the absolute costs vary, they are still small (considering Encode is a one-time operation), and the relative costs are consistent.

Figure 10 lists the CPU time taken by various operations for all the variants we have implemented. Our hypothesis holds to an extent: all the variants that are based on replication (for the producer) and hashing (for the consumer) follow our prediction. The source of costs for schedule generation corresponds to the time taken to find a solution to a k balls, m bins, and d choices problem. The different allocation strategies approximate the optimal solution, and among them, Cuckoo hashing yields the best approximation by recursively relocating elements when there are collisions (§4.5).[4] Encoding performance, on the other hand, is based on the number and the cost of the memory copies, since encoding is a simple repetition code.

Our hypothesis does not hold for the PBC variant that corresponds to a port of Pung's Hybrid multi-retrieval protocol. The reason is that this variant is partially based on the subcube batch code of Ishai et al. [47], for which the final position of each input element is statically determined and does not require computing a hash function (unlike our hashing variants). This allows computing a schedule by consulting a lookup table.

Finally, as mentioned above, our goal with this experiment was not to highlight the schemes' relative performance, but to instead show that all of them have efficient encoding, decoding, and schedule generation procedures. As such, we focus on the Cuckoo variant for the rest of our evaluation since it yields the most efficient parameters (highest rate and fewest buckets), and the second lowest failure probability (k-way replication never fails, but has terrible rate).

[4]We discuss other strategies in Section 8, including the use of more costly allocation strategies that find—rather than approximate—the optimal solution.

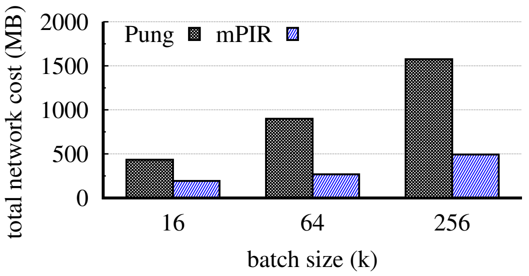


FIGURE 12—Total network cost (download and upload) for two multi-query PIR schemes (Pung and mPIR based on Cuckoo PBC) on a database with 131,072 elements. The cost of retrieval is for 288-byte elements across different batch sizes. The underlying PIR library is XPIR with parameters $\alpha = 14$ and $d = 2$.

7.3 Comparing mPIR to Pung's multi-retrieval

Pung introduces a new probabilistic multi-retrieval PIR protocol to amortize computational costs in exchange for higher communication costs [7, §4.4]. Pung's scheme is orders of magnitude more network efficient than existing batch codes, so we use it as a baseline comparison for mPIR. We set up a database with 131K elements and encode it using Pung's scheme and mPIR's Cuckoo PBC on a single machine. We set up a client on a separate machine and have it retrieve random elements using mPIR and Pung's scheme; we use XPIR as the underlying PIR protocol. Note that we test against both, Pung's actual code and mPIR using the variant that we ported into a PBC (§7.2); we observed comparable performance, so we discuss only the latter. We depict the results for different batch and element sizes, as well as a baseline that retrieves only one element using PIR (so it provides no CPU amortization), in Figure 11.

We find that mPIR does a better job at amortizing CPU costs across all batch sizes compared to Pung's scheme. This is a direct effect of the Cuckoo PBC having a higher rate (i.e., fewer total codewords, see Figure 5), since PIR's computational costs are proportional to the total number of elements after encoding (N). At $k = 256$ and 288-byte elements, mPIR achieves a 1.98× reduction in CPU cost for the server when answering queries (over Pung's scheme). Over the naive approach of retrieving one element at a time, the per-request CPU cost is 6.67× lower.

The difference in network costs, depicted in Figure 12, is more pronounced. The reasons are twofold. First, mPIR's PBC encoding results in fewer codewords (N). Since the query vectors are made up of $2\sqrt{N_b/\alpha}$ ciphertexts each (due to our use of the matrix encoding of the database, $d = 2$, see Section 3.3), mPIR leads to smaller queries. Here, N_b is the number of elements in bucket b. Second, and most significant, is that Pung's scheme builds on the subcube batch code of Ishai et al. [47] which creates a large number of buckets (see the second row of Figure 5); to preserve privacy clients must issue a PIR query to every bucket. These factors lead to each client sending to the server a total of $\Sigma_{b=0}^{m-1}(2\sqrt{N_b/\alpha})$ ciphertexts as the multi-query, while each client receives from the server mF ciphertexts as the multi-answer (F is the cryptosystem's expansion factor, as we discuss in Section 3.3). In terms of concrete savings, mPIR reduces total network costs (upload and download) by up to 3.4× in our experiments. Additionally, mPIR has a lower failure probability of around 2^{-40}, compared to Pung's 2^{-20}. Our results suggest that mPIR is an attractive replacement to Pung's multi-query protocol, offering improvements on all axes.

7.4 Pung with SealPIR and mPIR

We study the effect of both techniques on a deployment of Pung on 4 servers. We first give a very brief summary of Pung.

Communication in Pung. At its core, Pung is an untrusted key-value store where clients deposit and retrieve messages in discrete rounds. A client sends a message by storing it encrypted in the untrusted store under a label known only to the recipient. A client retrieves a message by using PIR; this breaks the link between sender and recipient (Pung also performs other actions needed to hide all communication metadata, but they are not relevant to our discussion). A round in Pung consists of all clients first sending s messages, followed by a phase where clients retrieve k messages with multi-query PIR.

Experiment. In our experiment, we have clients send and retrieve k messages. We perform this operations in a closed-loop, meaning that we advance rounds as soon as all clients have sent and retrieved the messages, rather than waiting for a timeout. To experiment with many clients, we employ the same simulation technique used in Pung: we have 4 real clients (running on their own VMs) who connect to a server each, and then have the servers simulate additional clients by populating their key-value stores with random messages. Pung servers replicate elements in the background to keep a consistent view, and ensure that any client can communicate with any other client regardless of the Pung server to which they connect.

Figure 13 shows the throughput in messages per minute that Pung achieves with mPIR and SealPIR ("Pung+MS"). Pung+MS yields better performance than the existing Pung codebase for many batch sizes. This might seem odd, but there are a few forces at play. First, even though SealPIR incurs additional CPU costs than XPIR, as we show in Section 7.1, the cost disparity is not that high when the database is small (see the columns with 8K elements in Figure 9). Meanwhile, not only does mPIR provide better amortization than Pung's scheme, but

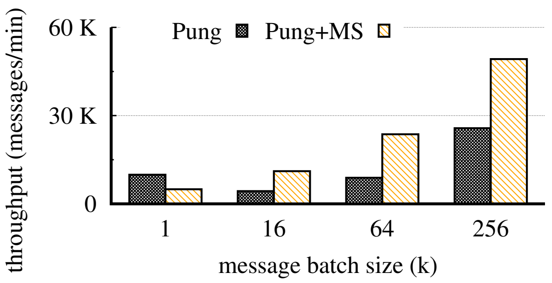


FIGURE 13—Throughput of Pung on a deployment of 4 servers with 131K users, each sending and retrieving k 1 KB messages per round. The label "Pung" indicates the implementation as given in [1], with updated security and XPIR parameters. The label "Pung+MS" corresponds to a version of Pung that uses mPIR and SealPIR.

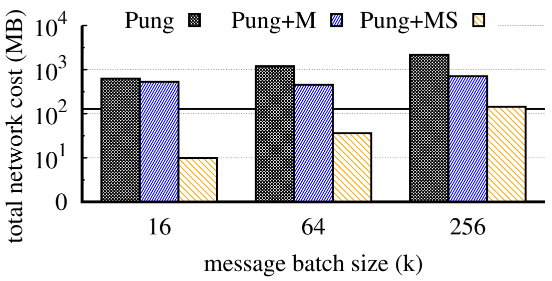


FIGURE 14—Per-user total network cost (upload and download) of a Pung deployment on 4 servers with 131K users. Each user sends and retrieves k 1 KB messages. The label "Pung" indicates the baseline version of Pung, "Pung+M" indicates a version of Pung that uses XPIR and mPIR, and "Pung+MS" indicates a version of Pung that uses mPIR and SealPIR. The black horizontal dashed line indicates the size of the database. A cost higher than this line means that the client would be better off downloading all messages without PIR.

it also breaks up the large database of messages sent by users into buckets that contain an average of 9K elements (which is a number that benefits SealPIR).

Second, as has been documented before [7], XPIR performs poorly when the database size is very small owing to high fixed costs. Since Pung's multi-query scheme generates many more buckets than mPIR ($9k$ buckets in total), the average number of elements in any one bucket is around 2K. This severely handicaps XPIR. Ultimately, we find that when each client is sending $k = 64$ messages each round, Pung+MS with SealPIR and mPIR processes 2.4× as many messages as Pung.

When it comes to network costs, the benefits of SealPIR and mPIR are even more considerable. Figure 14 depicts the total network cost incurred by a single client for one round of the Pung protocol. We find that the compressed queries and the more efficient multi-query PIR protocol result in savings of over 33.2×. In particular, the per-client communication costs are cut down to 36 MB per round for a batch size of 64 (versus 1.2 GB in the original Pung implementation).

8 Discussion

SealPIR significantly reduces the network cost of XPIR, while introducing modest computational overheads. However, there are several opportunities to reduce CPU costs further. Observe that in EXPAND and Stern's protocol, when the database dimension (d) is greater than 1 (see Section 3.3) the computation consists of several matrix-vector products. We can therefore implement the optimization described by Beimel et al. [12] where multiple queries (from potentially different users) are aggregated to form a matrix; the server can then use a subcubic matrix multiplication algorithm to compute the result (§2.2).

Another area of potential improvement is in the design of PBCs. As we show in our evaluation, PBCs built from hashing in the head reduce costs over existing methods, but so far we have only studied allocation strategies that are typically used for *online* load balancing (i.e., balls arrive one at a time). An interesting step going forward is to consider strategies that optimize for the *offline* setting in which all balls are available at the same time (which is the case in PBCs). In this setting, the allocation process can be phrased in terms of orienting the edges of undirected graphs in order to obtain directed graphs with minimum in-degree [20]. Optimal solutions for this problem can be computed in polynomial time [25], and linear time approximations also exist [20, 30, 37].

One last direction that we plan to explore is a different method for ensuring that the consumer and producer allocations in hashing in the head are compatible. In Section 4.4 we only considered solutions where the producer replicates each ball to all candidate buckets. We think that it might be possible to provide the same guarantee at a cost lower than replication.

Appendix

A Query expansion

A.1 Correctness of query expansion

Below we prove that EXPAND (Figure 3) correctly expands one ciphertext into a vector of n ciphertexts with the desired contents. The following theorem makes this formal.

Theorem 1. *Let N be a power of 2, $N \geq n$, and $c = Enc(x^j)$ be the client's encoding of index j. The n output ciphertexts $o_0, \ldots, o_{n-1}$ of EXPAND(c) satisfy, for all $0 \leq k \leq n-1$:*

$$o_k = \begin{cases} Enc(1) & \text{if } j = k \\ Enc(0) & \text{otherwise} \end{cases}$$

Proof. It suffices to prove the case for $n = 2^\ell$. For $i = \{0, 1, \ldots, \ell-1\}$, we claim that after the i^{th} iteration of the outer loop, we have $c = [c_0, \ldots, c_{2^{i+1}-1}]$ such that

$$c[k] = \begin{cases} Enc(2^{i+1}x^{j-k}) & \text{if } j \equiv k \pmod{2^{i+1}} \\ Enc(0) & \text{otherwise} \end{cases}$$

We prove the claim by induction on i. The base case $i = 0$ is explained in the main text of Section 3.2. Suppose the claim is true for some $i \geq 0$. Then in the next iteration, we compute an array c' of ciphertexts.

For the first half of the array, i.e., $0 \leq k < 2^{i+1}$, we have $c'[k] = c[k] + \mathsf{Sub}(c[k], N/2^{i+1}+1)$. If $j \not\equiv k \pmod{2^{i+1}}$, then $c'[k]$ is an encryption of 0; otherwise, there is an integer r such that $j - k = 2^{i+1} \cdot r$, and $\mathsf{Sub}(c[k], N/2^{i+1}+1) = Enc(2^{i+1}x^{(N/2^{i+1}+1)(2^{i+1}r)}) = Enc(2^{i+1}(-1)^r x^{j-k})$. Hence, if r is odd, then $c'[k]$ is an encryption of 0; otherwise, $c'[k]$ is an encryption of $2^{i+2}x^{j-k}$. So the claim follows because r is even if and only if $j \equiv k \pmod{2^{i+2}}$.

We now prove the claim for the second half of the array c'. The only interesting case is $j \equiv k - 2^{i+1} \pmod{2^{i+1}}$. In this case, it is easy to see that $c'[k]$ is again $Enc(2^{i+1}(-1)^{(j-k)/2^{i+1}}x^{j-k})$. So the same argument applies.

Finally, using the claim we can show that after the outer loop in EXPAND, we have an array of 2^ℓ ciphertexts such that:

$$c[k] = \begin{cases} Enc(2^\ell x^{j-k}) & \text{if } j \equiv k \pmod{2^\ell} \\ Enc(0) & \text{otherwise} \end{cases}$$

However, note that $j < n = 2^\ell$, so $j \equiv k \pmod{2^\ell}$ implies $j = k$. Hence c_k is either an encryption of 0 or an encryption of 2^ℓ. To obtain an encryption of 0 or 1, we multiply c_k by the inverse of 2^ℓ modulo t in the last step (Figure 3, Line 16). □

A.2 Noise growth of query expansion

One advantage of our query expansion technique over the straw man FHE solution given in Section 3.1 (besides the one mentioned in that section) is that our approach has *much* smaller noise growth. We bound the noise growth of EXPAND (Figure 3) in the theorem below. Before stating the theorem, we give some background on noise. See the SEAL manual [23] for a more detailed explanation. We have that the noise of the addition of two ciphertexts is the sum of their individual noises. Plain multiplication by a power of X does not change the noise, and plain multiplication by a constant multiplies the noise by α. Substitution adds a constant additive term B_{sub} to the noise, which depends on the FV parameters.

Theorem 2. *Let v_{out} be the output noise of EXPAND, and v_{in} be the input noise. Let t denote the plaintext modulus in EXPAND, and let $k = \lceil \log(n) \rceil$. We have that*

$$v_{out} \leq t \cdot (2^k(v_{in} + 2B_{sub}))$$

Proof. Let v_i be the noise after the i^{th} iteration in EXPAND (setting $v_0 = v_{in}$). Then $v_i = 2(v_{i-1} + B_{sub})$. Carrying out the sum, we get

$$v_k = 2^k v_0 + 2(2^k - 1)B_{sub} < 2^k(v_0 + 2B_{sub})$$

Since inverse $\leq t$, the final plain multiplication results in $v_{out} \leq tv_k$. This completes the proof. □

Acknowledgments

We thank Jay Lorch, Trinabh Gupta, and Michael Walfish for their helpful feedback. Sebastian Angel was supported by NSF grant CNS-1514422 and AFOSR grant FA9550-15-1-0302.

References

[1] Pung: Unobservable communication over fully untrusted infrastructure. `https://github.com/pung-project/pung`, Sept. 2017.

[2] Simple encrypted arithmetic library — SEAL. `http://sealcrypto.org`, 2017.

[3] XPIR: NFLLWE security estimator. `https://github.com/XPIR-team/XPIR/blob/master/crypto/NFLLWESecurityEstimator/NFLLWESecurityEstimator-README`, June 2017.

[4] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[5] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. `https://github.com/xpir-team/xpir/`, 2016.

[6] M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3), Oct. 2015.

[7] S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2016.

[8] Y. Arbitman, M. Naor, and G. Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 2010.

[9] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal. Balanced allocations. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1994.

[10] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang. On the (im)possibility of obfuscating programs. *Journal of the ACM*, 59(2), 2012.

[11] A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Nov. 2002.

[12] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2000.

[13] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7), July 1970.

[14] N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, June 2015.

[15] E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can we access a database both locally and privately? Cryptology ePrint Archive, Report 2017/567, Sept. 2017. `http://eprint.iacr.org/2017/567.pdf`.

[16] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the Innovations in Theoretical Computer Science (ITCS) Conference*, Jan. 2012.

[17] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from Ring-LWE and security for key dependent messages. In *Proceedings of the International Cryptology Conference (CRYPTO)*, Aug. 2011.

[18] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen. Horton tables: Fast hash tables for in-memory data-intensive computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, June 2016.

[19] C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, May 1999.

[20] J. A. Cain, P. Sanders, and N. Wormald. The random graph threshold for k-orientability and a fast algorithm for optimal multiple-choice allocation. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Jan. 2007.

[21] R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. Cryptology ePrint Archive, Report 2017/568, Sept. 2017. `http://eprint.iacr.org/2017/568.pdf`.

[22] Y.-C. Chang. Single database private information retrieval with logarithmic communication. In *Proceedings of the Australasian Conference on Information Security and Privacy*, July 2004.

[23] H. Chen, K. Laine, and R. Player. Simple encrypted arithmetic library - SEAL v2.2. Technical Report MSR-TR-2017-22, Microsoft, June 2017.

[24] H. Chen, K. Laine, and P. Rindal. Fast private set intersection from homomorphic encryption. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2017.

[25] L. T. Chen and D. Rotem. Optimal reponse time retrieval of replicated data. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, May 1994.

[26] R. Cheng, W. Scott, B. Parno, I. Zhang, A. Krishnamurthy, and T. Anderson. Talek: a private publish-subscribe protocol. Technical Report UW-CSE-16-11-01, University of Washington Computer Science and Engineering, Nov. 2016.

[27] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, Feb. 1998. `http://eprint.iacr.org/1998/003`.

[28] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1995.

[29] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1987.

[30] A. Czumaj and V. Stemann. Randomized allocation processes. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[31] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *Proceedings of the ACM Cloud Computing Security Workshop (CCSW)*, Nov. 2014.

[32] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2012.

[33] M. Dietzfelbinger, A. Goerdt, M. Mitzenmacher, A. Montanari, R. Pagh, and M. Rink. Tight thresholds for Cuckoo hashing via XORSAT. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 2010.

[34] M. Dietzfelbinger and F. Meyer auf der Heide. A new universal class of hash functions and dynamic hashing in real time. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 1990.

[35] C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, Sept. 2014.

[36] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, Mar. 2012. `http://eprint.iacr.org/2012/144.pdf`.

[37] D. Fernholz and V. Ramachandran. The k-orientability thresholds for $G_{n,p}$. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Jan. 2007.

[38] S. Garg, E. Miles, P. Mukherjee, A. Sahai, A. Srinivasan, and M. Zhandry. Secure obfuscation in a weak multilinear map model. In *Proceedings of the Theory of Cryptography Conference (TCC)*, Oct. 2016.

[39] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, Apr. 2012.

[40] C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, July 2005.

[41] I. Goldberg. Improving the robustness of private information retrieval. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, May 2007.

[42] M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Oct. 2016.

[43] J. Groth, A. Kiayias, and H. Lipmaa. Multi-query computationally-private information retrieval with constant communication rate. In *Proceedings of the International Conference on Practice and Theory in Public Key Cryptography (PKC)*, May 2010.

[44] T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with Popcorn. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Mar. 2016.

[45] R. Henry. Polynomial batch codes for efficient IT-PIR. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[46] R. Henry, Y. Huang, and I. Goldberg. One (block) size fits all: PIR and SPIR with variable-length records via multi-block queries. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb. 2013.

[47] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, June 2004.

[48] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Zero-knowledge from secure multiparty computation. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 21–30, 2007.

[49] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, May 1997.

[50] P. Key, L. Massoulié, and D. Towsley. Path selection and multipath congestion control. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, May 2007.

[51] M. Khosla. Balls into bins made faster. In *Proceedings of the European Symposium on Algorithms (ESA)*, Sept. 2013.

[52] A. Kiayias, N. Leonardos, H. Lipmaa, K. Pavlyk, and Q. Tang. Optimal rate private information retrieval from homomorphic encryption. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2015.

[53] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1997.

[54] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, July 2016.

[55] F. Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the International Symposium on Symbolic and Algebraic Computation (ISSAC)*, July 2014.

[56] H. Lipmaa. First CPIR protocol with data-dependent computation. In *Proceedings of the International Conference on Information, Security and Cryptology (ICISC)*, Dec. 2009.

[57] H. Lipmaa and K. Pavlyk. A simpler rate-optimal CPIR protocol. In *Proceedings of the International Financial Cryptography Conference*, Apr. 2017.

[58] H. Lipmaa and V. Skachek. Linear batch codes. In *Proceedings of the International Castle Meeting on Coding Theory and Applications*, Sept. 2014.

[59] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *Proceedings of the International Financial Cryptography and Data Security Conference*, Jan. 2015.

[60] P. Mittal, F. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-Tor: Scalable anonymous communication using private information retrieval. In *Proceedings of the USENIX Security Symposium*, Aug. 2011.

[61] M. Mitzenmacher. The power of two choices in randomized load balancing. *IEEE Transactions on Parallel and Distributed Systems*, 12(10), Oct. 2001.

[62] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Nov. 2013.

[63] R. Pagh and F. F. Rodler. Cuckoo hashing. In *Proceedings of the European Symposium on Algorithms (ESA)*, Aug. 2001.

[64] M. B. Paterson, D. R. Stinson, and R. Wei. Combinatorial batch codes. *Advances in Mathematics of Communications (AMC)*, 3(1), Feb. 2009.

[65] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. Cryptology ePrint Archive, Report 2016/930, Sept. 2016. `http://eprint.iacr.org/2016/930.pdf`.

[66] A. S. Rawat, D. S. Papailiopoulos, A. G. Dimakis, and S. Vishwanath. Locality and availability in distributed storage. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, June 2014.

[67] A. S. Rawat, Z. Song, A. G. Dimakis, and A. Gál. Batch codes through dense graphs without short cycles. *IEEE Transactions on Information Theory*, 62(4), Apr. 2016.

[68] N. Silberstein. Fractional repetition and erasure batch codes. In *Proceedings of the International Castle Meeting on Coding Theory and Applications*, Sept. 2014.

[69] N. Silberstein and T. Etzion. Optimal fractional repetittion codes and fractional repetition batch codes. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, June 2015.

[70] J. P. Stern. A new and efficient all-or-nothing disclosure of secrets protocol. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Oct. 1998.

[71] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4), Aug. 1969.

[72] K. Talwar and U. Wieder. Balanced allocations: the weighted case. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, June 2007.

[73] B. Vöcking. How asymmetry helps load balancing. *Journal of the ACM*, 50(4), 2003.

[74] Z. Wang, H. M. Kiah, and Y. Cassuto. Optimal binary switch codes with small query size. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, June 2015.

[75] Z. Wang, O. Shaked, Y. Cassuto, and J. Bruck. Codes for network switches. In *Proceedings of the IEEE International Symposium on Information Theory (ISIT)*, July 2013.

[76] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5), May 2013.