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 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 query sent by the client contains a number of ciphertexts proportional to the size of the database. We show how to compresses this query, achieving size reductions of up to 274×.

The second technique is a new data encoding called *probabilistic batch codes* (PBCs). We use PBCs to build a multi-query PIR scheme that allows the server to amortize its computational cost when processing a batch of requests from the same client. This technique achieves up to 40× speedup over processing queries one at a time, and is significantly more efficient than related encodings. We apply our techniques to the Pung private communication system, which relies on a custom multi-query CPIR protocol for its privacy guarantees. By porting our techniques to Pung, we find that we can simultaneously reduce network costs by 36× and increase throughput by 3×.

1 Introduction

A key cryptographic building block in recent privacy-preserving systems is *private information retrieval* (PIR) [33]. Examples include anonymous and metadata-private communication [11, 58, 63], privacy-preserving media streaming [8, 49], ad delivery [47], friend discovery [18], and subscriptions [31].

PIR allows a client to download an element (e.g., movie, 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. While powerful, PIR is very expensive—and unfortunately this expense is fundamental: PIR schemes force the server to operate on all elements in the database to answer a single query [33]. After all, if the server could omit an element when answering a query it would learn that the omitted element is of no interest to the client.

We are interested in the computational variant of PIR (CPIR) [57], which is desirable since it relies only on cryptographic hardness assumptions and can be used even when the database is operated by a single organization (we discuss alternatives in Section 2.1). Unfortunately, the costs of existing CPIR constructions [8, 21, 24, 39, 56, 57, 60, 72] are so significant that existing CPIR-backed systems must settle with supporting small databases with fewer than 100K entries [8, 11, 47, 49].

In this paper we discuss two orthogonal but complementary contributions that make CPIR more efficient in practice. The first is the introduction of SealPIR, a new CPIR library that extends the most computationally-efficient CPIR protocol, XPIR [8], with a new query compression technique that reduces network costs (§3). Specifically, a query in XPIR (and its base protocol [72]), consists of a vector of n ciphertexts, where n is the number of elements in the server's database. Stern [72] showed that it is possible to reduce the number of ciphertexts to $d\sqrt[d]{n}$ for any positive integer d, thereby making network costs sublinear in the size of the database. The downside of Stern's approach is that it comes at an exponential increase in the size of the response (§3.4). As we show in our evaluation, values of $d > 3$ in XPIR lead to responses that are so large that they outweigh any reduction in query size (§7.1).

SealPIR adopts a fundamentally different approach. Instead of creating a query vector, SealPIR has the client send a single ciphertext containing an encoding of the index of the desired element. 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 (§3.3). 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 queries that are 274× smaller and are 16.4× less expensive for the client to construct. However, SealPIR introduces between 11% and 24% CPU overhead to the server (over XPIR) to obliviously expand queries. We think this is an excellent trade-off since XPIR's protocol is embarrassingly parallel and one can regain the lost throughput by employing additional servers. Furthermore, reducing communication overhead makes PIR usable in settings where clients have limited bandwidth, such as mobile devices or wired connections with data limits [7].

Our second contribution is a new technique to amortize the server's CPU cost when processing multiple queries from the same client. This technique is a relaxation of *batch codes* [52], a data encoding that was originally proposed for this purpose. In practice, most batch code constructions target a different domain—providing load balancing and availability guarantees to distributed storage [67, 69] and network switches [77]; using these constructions to amortize the processing of a batch of PIR queries is not worthwhile since they introduce high network costs while yielding only modest CPU speedups (§4.1).

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, a failure simply means that a client only gets some (not all) of her queries answered in a single interaction. While the implications of a failure depend on the application, we argue that in many cases this is not an issue (§5). Moreover, the failure probability of our constructions is low—about 1 in a trillion multi-queries would be affected.

The key idea behind our PBC construction is a simple new technique called *reverse hashing* (§4.3). This technique flips the way that hashing (e.g., multi-choice [64], cuckoo [66]) 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*. Like batch codes, PBCs amortize CPU costs when processing a batch of queries. Unlike batch codes, PBCs are more network-efficient: they introduce orders of magnitude less network overhead (§7.3).

We demonstrate the benefits of our techniques through an experimental evaluation of several deployments that include well-provisioned, bandwidth-limited, and geo-distributed clients on databases of up to four million entries. We also integrate SealPIR and PBCs into a recent private communication system [11] that uses multi-query CPIR for its privacy guarantees.

In summary, the contributions of this work are:

- SealPIR, a CPIR library that reduces network costs through a new oblivious query expansion procedure (§3).
- The design of PBCs, a new probabilistic data encoding useful for building multi-query PIR protocols that amortize computational costs (§4).
- The implementation and evaluation of SealPIR and PBC on a variety of settings (§7), including porting these techniques to the Pung communication system (§7.4).

Despite these advances, there remains a large performance gap between CPIR implementations and widespread adoption. Nevertheless, we hope that this work can 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. [33] introduce private information retrieval (PIR) to answer the following questions: can a client retrieve an element from a database managed by an untrusted server (or set of servers) without the server learning *which* element was retrieved by the client? And can this be done more efficiently than simply having the client download the entire database? Chor et al.'s affirmative response inspired two lines of work: *information theoretic* PIR (IT-PIR) and *computational* PIR (CPIR).[1]

In IT-PIR schemes [15, 33, 36, 37, 46] the database is replicated across several non-colluding servers. The client issues a query to each server and combines the responses from all of 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 that they hold against computationally-unbounded adversaries and avoid cryptographic hardness assumptions. However, basing systems on IT-PIR poses a significant deployment challenge since it can be difficult to enforce the non-collusion assumption in practice.

On the other hand, CPIR protocols [8, 21, 24, 27, 39, 45, 56, 57, 59, 60, 78] 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 on each database element. Fortunately, there is a long line of work to improve the resource overheads of CPIR (see [8, 56] for the state-of-the-art), and recent work [8] 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. We discuss this protocol in Section 3.

Regardless of which PIR variant a system implements, the concrete costs remain high. As a result, it is hard for systems to support large databases or handle many requests. While supporting large databases remains out of reach—although Section 3 makes progress on this—supporting a batch of queries is the focus of existing proposals. We discuss them next.

2.2 Existing multi-query PIR schemes

Given PIR's high costs, it is desirable to amortize the servicing of many requests. Such scenarios include databases that process a batch of requests from the *same* user (e.g., Email, group chat, bulletin boards). The most general approach to achieve this goal is to use a *batch code* [52, 67, 69]. In a batch code, the database is encoded such that the server (or servers) can respond to *any* k requests (from the same user) more cheaply (computationally) than running k parallel instances of PIR. The trade-off is that batch codes require more network resources (than the k parallel instances). In practice, this network overhead is onerous; we discuss this further in Section 4.1.

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

Amortization for particular protocols. Beimel et al. [16] 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, subcubic matrix multiplication algorithms (e.g., [34, 73]) provide amortization over multiple matrix-vector multiplication instances. This approach is further studied by Lueks and Goldberg [62] in the context of Goldberg's IT-PIR scheme [46].

The second proposal described by Beimel et al. [16] is to preprocess the database in certain IT-PIR schemes to reduce the cost of future queries. Since this works well, recent projects [19, 26] employ an analogous approach in CPIR schemes. However, making the preprocessed database accessible by more than one client under these schemes requires cryptographic primitives that are currently too inefficient to be implemented (*virtual black-box obfuscation* [14] heuristically instantiated from *indistinguishability obfuscation* [43]).

Several works [36, 48, 50, 51] extend specific PIR schemes to achieve CPU or network amortization. Related to CPIR, Groth et al. [48] extend Gentry and Ramzan's [45] 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 low network

[1] Also known as multi-database PIR (IT-PIR) and single-database PIR (CPIR).

```
1:  function SETUP(DB)
2:      Represent DB in an amenable format (see [8, §3.2])
3:
4:  function QUERY(pk, idx, n)
5:      for i = 0 to n − 1 do
6:          c_i ← Enc(pk, i == idx? 1 : 0)
7:      return q ← {c_0, ..., c_{n−1}}
8:
9:  function ANSWER(q = {c_0, ..., c_{n−1}}, DB)
10:     for i = 0 to n − 1 do
11:         a_i ← DB_i · c_i          // plaintext-ciphertext multiplication
12:     return a ← Σ_{i=0}^{n−1} a_i  // homomorphic addition
13:
14: function EXTRACT(sk, a)
15:     return Dec(sk, a)
```

FIGURE 1—CPIR protocol from Stern [72] and XPIR [8] on a database *DB* of n elements. 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 omit the details of all optimizations. The client runs the QUERY and EXTRACT procedures, and the server runs the SETUP and ANSWER procedures. Each element in *DB* is assumed to fit inside a single ciphertext. Otherwise, each element can be split into ℓ smaller chunks, and Lines 11 and 12 can be performed on each chunk individually; in this case ANSWER would return ℓ ciphertexts instead of one.

costs, but Gentry and Ramzan's scheme is computationally expensive (tens of minutes to process a single PIR query, based on our estimates); Groth et al.'s extension compounds this issue.

Amortization for particular apps. Popcorn [49] pipelines the processing of queries in IT-PIR to amortize disk I/O, which is a bottleneck for databases with very large files such as movies. Pung [11] hybridizes an existing batch code due to Ishai et al. [52] with a probabilistic protocol that exploits the setting of online communication where users can coordinate a priori (e.g., chat, 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 [52], our technique has weaker properties (sufficient for many applications) but is significantly more efficient. Compared to Pung's technique, our approach is more efficient and application-independent (§4.2).

3 SealPIR: An efficient CPIR library

Our starting point for SealPIR is XPIR [8], a recent construction that improves on Stern's CPIR scheme [72]. We give a rough sketch of the protocol in Figure 1. The key idea in XPIR is to perform the encryption and homomorphic operations using a lattice-based cryptosystem (the authors use BV [22]), and preprocess the database in a way that reduces the cost of the operations in Lines 11 and 12 in Figure 1. To our knowledge, this makes XPIR the only CPIR scheme that is usable in practice.

A major drawback of XPIR is network costs. In particular, the 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. Furthermore, lattice-based cryptosystems have a high *expansion factor*, F, which is the size ratio between a ciphertext and the largest plaintext that can be encrypted; for recommended security parameters, $F \geq 6.4$ [10, 28].

operation	**CPU cost (ms)**	**noise growth**
addition	0.002	additive
plaintext multiplication	0.141	multiplicative*
multiplication	1.514	multiplicative
substitution	0.279	additive

FIGURE 2—Cost of operations in SEAL [4]. The parameters used are given in Section 7. Every operation increases the *noise* in a ciphertext. Once the noise passes a threshold, the ciphertext cannot be decrypted. For a given computation, parameters must be chosen 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 [28, §6.2].

To improve network costs, Stern [72] describes a way to represent the query using $d\sqrt[d]{n}$ ciphertexts (instead of n) for any positive integer d. Unfortunately, this increases the response size exponentially from 1 to F^{d-1} ciphertexts (Section 3.4 explains this). If the goal is to minimize network costs, a value of $d = 2$ or 3 is optimal in XPIR for the databases that we evaluate (§7.1). As a result, even with this technique, the query vector is made up of hundreds or thousands of ciphertexts.

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 i to the server, and the server inexpensively evaluates a function EXPAND that outputs n ciphertexts containing an encryption of 0 or 1 (where the i^{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 9).

A straw man approach to construct EXPAND is to create a Boolean circuit that computes the following function: "if the index encrypted by the client is i return 1, else return 0". The server then evaluates this circuit on the client's ciphertext using a fully homomorphic encryption (FHE) scheme (e.g., BV [22], BGV [20], FV [40]) passing in values of $i \in [0, n-1]$ to obtain the n ciphertexts. Unfortunately, this approach is impractical. First, the client must send $\log(n)$ ciphertexts as the query (one for each bit of its index since the server evaluates a Boolean circuit). Second, the Boolean circuit is concretely large (thousands of gates) and expensive to evaluate. Finally, the server must evaluate this circuit for each of the n possible indices.

Instead, we propose a new algorithm to implement EXPAND. It relies on FHE, but perhaps surprisingly, it does not require encrypting each bit of the index individually, working with Boolean gates, or performing any homomorphic multiplications. This last point is critical for performance, since homomorphic multiplications are expensive and require using larger security parameters (Figure 2). We note that the cryptosystem used by XPIR (BV [22]) is an FHE scheme, so we could implement EXPAND using that. However, we choose to implement all of SealPIR using the SEAL homomorphic library [4]—based on the Fan-Vercauteren (FV) [40] cryptosystem—instead. We make this choice for pragmatic reasons: EXPAND requires the implementation of a new homomorphic operation, and SEAL 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. Specifically, 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. In Section 6 we discuss how regular binary data, for example a movie, is encoded in an FV plaintext, and what these polynomials actually look like in practice.

Ciphertexts in FV consist of two polynomials, each of which is 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 plaintext is encrypted, the corresponding ciphertext contains noise. As operations such as addition or multiplication are performed, the noise of the output ciphertext grows based on the noise of the operands and the operation being performed (Figure 2 gives the noise growth of several operations). Once the noise passes a threshold, the ciphertext cannot be decrypted. The noise growth of operations depends heavily on t, so t should be kept small. However, lower t means that more FV plaintexts are needed to represent the binary data (movie, etc.). Larger q supports more noise, but results in lower security [28]. The expansion factor is $F = 2\log(q)/\log(t)$. We discuss concrete parameters in Section 7.

In addition to the standard operations of a cryptosystem (key generation, encryption, decryption), FV also supports homomorphic addition, multiplication, and relinearization (which is performed after multiplications to keep 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 $p_1(x), p_2(x)$, 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 $p_1(x)$, and given a plaintext $p_2(x)$, the operation $p_2(x) \cdot c$ results in a ciphertext that encrypts $p_1(x) \cdot p_2(x)$.
- **Substitution**: Given a ciphertext c that encrypts plaintext $p(x)$ and an odd integer k, the operation $\mathsf{Sub}(c, k)$ returns an encryption of $p(x^k)$. For instance if c encrypts $p(x) = 7 + x^2 + 2x^3$, then $\mathsf{Sub}(c, 3)$ returns an encryption of $p(x^3) = 7 + (x^3)^2 + 2(x^3)^3 = 7 + x^6 + 2x^9$.

Our implementation of the substitution operation is based on the plaintext slot permutation technique discussed by Gentry et al. [44, §4.2]. Fortunately, substitution requires only a subset of the operations needed by the arbitrary permutations that Gentry et al. consider, so we can implement it very efficiently, as shown in the last row of Figure 2. We give a detailed description of substitution in Appendix A.1.

3.2 Encoding the index

A client who wishes to retrieve the i^{th} element from the server's database using SealPIR generates an FV plaintext that encodes this index. The client does so by representing $i \in [0, n-1]$ as the monomial $x^i \in R_t$. The client then encrypts this plaintext to obtain $query = Enc(x^i)$, which is then sent to the server. We later discuss how to handle larger databases for which the index cannot be represented by a single FV plaintext (§3.5).

3.3 Expanding queries obliviously

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

As discussed in the previous section, the server receives $query = Enc(x^i)$, with $i \in \{0, 1\}$ in this case (since $n = 2$) as the client's desired index. The server first expands $query$ into two ciphertexts $c_0 = query$ and $c_1 = query \cdot x^{-1}$:

$$c_0 = \begin{cases} Enc(1) & \text{if } i = 0 \\ Enc(x) & \text{if } i = 1 \end{cases}$$

$$c_1 = \begin{cases} Enc(x^i \cdot x^{-1}) = Enc(x^{-1}) & \text{if } i = 0 \\ Enc(x^i \cdot x^{-1}) = Enc(1) & \text{if } i = 1 \end{cases}$$

The server computes $c'_j = c_j + \mathsf{Sub}(c_j, N+1)$ for $j \in \{0, 1\}$. Since operations in R_t are defined modulo $x^N + 1$, a substitution with $N + 1$ transforms the plaintext encrypted by c_0 and c_1 from $p(x)$ to $p(-x)$.[2] Specifically, we have:

$$c'_0 = \begin{cases} Enc(1) + Enc(1) = Enc(2) & \text{if } i = 0 \\ Enc(x) + Enc(-x) = Enc(0) & \text{if } i = 1 \end{cases}$$

$$c'_1 = \begin{cases} Enc(x^{-1}) + Enc(-x^{-1}) = Enc(0) & \text{if } i = 0 \\ Enc(1) + Enc(1) = Enc(2) & \text{if } i = 1 \end{cases}$$

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 $o_j = \alpha \cdot c'_j$. It is the case that o_0 and o_1 contain the desired output of EXPAND: o_i encrypts 1, and o_{1-i} 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 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.2, and bound its noise growth in Appendix A.3.

3.4 Reducing the cost of expansion

One issue with 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 almost as expensive to the server as computing the answer to a query (see Figure 1, Line 9). We show how to reduce this cost by having the client send multiple ciphertexts.

Stern [72] 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}

[2] Observe that $x^N + 1 \equiv 0 \pmod{x^N + 1}$ and $x^{N+1} \equiv -x \pmod{x^N + 1}$.

function EXPAND($query = Enc(x^i)$)
  find smallest $m = 2^\ell$ such that $m \geq n$
  $ciphertexts \leftarrow [\, query \,]$
  // each outer loop iteration doubles the number of ciphertexts,
  // and only one ciphertext ever encrypts a non-zero polynomial
  for $j = 0$ to $\ell - 1$ **do**
    for $k = 0$ to $2^j - 1$ **do**
      $c_0 \leftarrow ciphertexts[k]$
      $c_1 \leftarrow c_0 \cdot x^{-2^j}$
      $c'_k \leftarrow c_0 + \mathsf{Sub}(c_0, N/2^j + 1)$
      $c'_{k+2^j} \leftarrow c_1 + \mathsf{Sub}(c_1, N/2^j + 1)$
    $ciphtertexts \leftarrow [c'_0, \ldots, c'_{2^{j+1}-1}]$
  // ciphertext at position j encrypts m and all others encrypt 0
  $inverse \leftarrow m^{-1} \pmod{t}$
  for $j = 0$ to $n - 1$ **do**
    $o_j \leftarrow ciphertexts[j] \cdot inverse$
  return $output \leftarrow [o_0, \ldots, o_{n-1}]$

FIGURE 3—Procedure that expands a single ciphertext *query* that encodes an index i into a vector of n ciphertexts, where the i^{th} entry is an encryption of 1, and all other entries are encryptions of 0. We introduce a new group operation Sub (see text for details). Plaintexts are in the polynomial quotient ring $\mathbb{Z}_t[x]/(X^N + 1)$. $N \geq n$ is a power of 2, and n is the number of elements in the server's database.

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 ciphertexts, and additions are on ciphertexts. Observe that A_c is a vector containing the encryptions of the 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

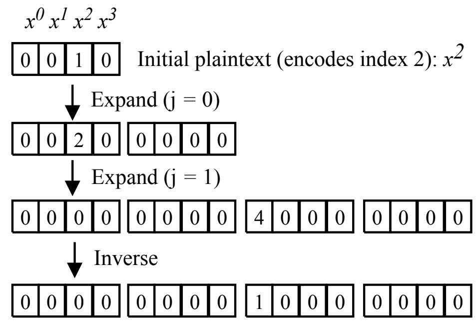


FIGURE 4—Example of EXPAND's effect on the FV plaintexts during each iteration of the outer loop. Each array represents the coefficients of the corresponding polynomial. Note that the server only sees the ciphertexts (not depicted). This example assumes a database with 4 elements, and a query retrieving the third item.

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.

3.5 Handling larger databases

As we discuss in Section 3.3, the size of the query vector that EXPAND can generate is bounded by N. Based on recommended security parameters [10, 28], N is typically 2048 or 4096 (larger N improves security but reduces performance). So how can one index into databases with more than N elements?

We propose two solutions. First, the client sends multiple ciphertexts and the server expands them and concatenates the results. For instance, if N is 2048, the database has 4000 elements, and the client wishes to get the element at index 2050, the client sends 2 ciphertexts: the first encrypts 0 and the second encrypts x^2. The server expands both ciphertexts into 2048-entry vectors and concatenates them to get a 4096-entry vector where the entry at index 2050 encrypts 1, and all others encrypt 0. The server then uses the first 4000 entries as the query vector.

A more efficient solution is to represent the database as a d-dimensional hypercube as we discuss in Section 3.4. This allows the client to send d ciphertexts to index a database of size N^d. For $d = 2$ and $N = 4096$, two ciphertexts are sufficient to index 16.7 million entries. One can also use a combination of these solutions. For example, given a database with 2^{30} entries, SealPIR uses $d = 2$ (so the database is a $2^{15} \times 2^{15}$ matrix), and represents the index for each dimension using $2^{15}/4096 = 8$ ciphertexts. The server expands these 8 ciphertexts and concatenates them to obtain a vector of 2^{15} entries. In total, this approach requires the client to send 16 ciphertexts as the query (8 per dimension), and receive $F \approx 7$ ciphertexts as the response ($d = 3$ would lead to 3 ciphertexts as the query, but F^2 ciphertexts as the response).

In short, the query communication complexity goes from $O(Nd\sqrt[d]{n})$ in XPIR to $O(Nd\lceil\sqrt[d]{n}/N\rceil)$ in SealPIR.

4 Amortizing computational costs in PIR

Answering a PIR query requires computation that is linear in the size of the database, so a promising way to save computational resources is for the server to amortize costs by processing a batch of queries. *Batch codes* [52] are a data encoding that, among other applications, can be used to achieve this goal. In particular, the server can use a batch code to encode its database in a way that it can answer a batch of queries more cheaply (computationally) than answering each query individually. Unfortunately, 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 [61, 67, 69] and network switches [77], which have different requirements. Using these codes to amortize PIR query processing would incur prohibitive network costs.

Our key 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 system. Below we give a description of batch codes, highlight the sources of overhead, and then introduce our construction.

4.1 Batch codes and their cost

A (n, m, k, b)-batch code $\mathcal{B}$ takes as input a collection DB of n elements, and produces a set of m codewords, C, distributed among b buckets.[3] Formally, $\mathcal{B} : DB \to (C_0, \dots, C_{b-1})$, where $|C_i|$ is the number of codewords in bucket i, and the sum of codewords across all buckets is $m = \Sigma_{i=0}^{b-1}|C_i| \geq n$. The goal of these codes is two-fold. First, they ensure that any k elements from DB can be retrieved from the b buckets by fetching at most one codeword from each bucket. Second, they keep the number of total codewords, m, lower than $k \cdot n$.

Example. We describe a $(4, 6, 2, 3)$-batch code, specifically the *subcube batch code* [52]. Let $DB = \{x_1, x_2, x_3, x_4\}$. For the encoding, DB is split in half to produce 2 buckets, and a third bucket is produced by XORing the entries in the first two buckets: $\mathcal{B}(DB) = (\{x_1, x_2\}, \{x_3, x_4\}, \{x_1 \oplus x_3, x_2 \oplus x_4\})$. Observe that one can obtain any 2 elements in DB by querying each bucket at most once. For example, to obtain x_1 and x_2, one can get x_1 from the first bucket, x_4 from the second bucket, and $x_2 \oplus x_4$ from the third bucket; $x_2 = x_4 \oplus (x_2 \oplus x_4)$.

This encoding is helpful for PIR because a client wishing to retrieve 2 elements from DB can, instead of querying DB twice, issue one query to each bucket. The server is in effect computing over 3 "databases" with 2 elements each, which results in 25% fewer operations.

Costs of PIR with batch codes. Figure 5 depicts the relationship between the number of codewords (m) and the number of buckets b, as a function of the database size (n) and the batch size (k) for several constructions. In multi-query PIR, the client issues one query to each of the b buckets, and therefore receives b responses (§5). To answer these b queries, the server computes over all m codewords exactly once; lower values of m lead to less computation, and lower values of b lead to lower network costs. Since $m < k \cdot n$, the total computation done by the server is lower than running k parallel instances of PIR. The drawback is that existing batch codes produce many buckets (see the third column in Figure 5). As a result, they *introduce* significant network overhead over not using a batch code at all. This makes batch codes unappealing in practice.

[3] We use different variable names (e.g., m and b) from the batch code literature to avoid overloading variable names introduced in Section 3.

4.2 Probabilistic batch codes (PBC)

Batch codes have exciting properties, but existing constructions offer an unattractive trade-off: they reduce computation but add network overhead. We make this trade-off more appealing by relaxing batch codes' guarantees.

A *probabilistic batch code* (PBC) differs from a traditional batch code in that it fails to be *complete* with probability p. That is, there might be no way to recover a specific set of k elements from a collection encoded with a PBC by retrieving exactly one codeword from each bucket. The probability of encountering one such set (when elements are uniformly chosen) is p. In the example of Section 4.1, this would mean that under a PBC, a client may be unable to retrieve both x_1 and x_2 by querying buckets at most once (whereas a traditional batch code guarantees that this is always possible). In practice, this is seldom 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 or not it can get all of the elements *before* issuing any queries.

Definition 1 (PBC). A (n, m, k, b, p)-PBC is given by three polynomial-time algorithms (Encode, GenSchedule, Decode):

- $(C_0, \dots, C_{b-1}) \leftarrow \mathsf{Encode}(DB)$: Given an n-element collection DB, output a b-tuple of buckets, where $b \geq k$, each bucket contains zero or more codewords, and the total number of codewords across all buckets is $m = \Sigma_{i=0}^{b-1}|C_i| \geq n$.
- $\{\sigma, \perp\} \leftarrow \mathsf{GenSchedule}(I)$: Given a set of k indices I corresponding to the positions of elements in DB, output a *schedule* $\sigma : I \to \{\{0, \dots, b-1\}^{+}\}^{k}$. The schedule σ gives, for each position $i \in I$, the index of one or more buckets from which to retrieve a codeword that can be used to reconstruct element $DB[i]$. GenSchedule outputs $\perp$ if it cannot produce a schedule where each $i \in I$ is associated with at least one bucket, and where no buckets is used more than once. This failure event occurs with probability p.
- *element* $\leftarrow \mathsf{Decode}(W)$: Given a set of codewords W, output the corresponding element $\in DB$.

In the subsections ahead we describe an efficient PBC construction. Our key idea is as follows. Batch codes spread out elements such that retrieval requests are load balanced among different buckets. Relatedly, many data structures and networking applications use different variants of hashing—consistent [53], asymmetric [75], weighted [74], multi-choice [13, 64], cuckoo [12, 66], and others [23, 38]—to achieve a similar 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, which we call *reverse hashing*, and then use the PBC to implement multi-query PIR (§5).

batch code	codewords (m)	buckets (b)	probability of failure (p)
subcube ($\ell \geq 2$) [52, §3.2]	$n \cdot ((\ell+1)/\ell)^{\log_2(k)}$	$(\ell+1)^{\log_2(k)}$	0
combinatorial ($\binom{r}{k-1} \leq n/(k-1)$) [67, §2.2]	$kn - (k-1) \cdot \binom{r}{k-1}$	r	0
Balbuena graphs [69, §IV.A]	$2(k^3 - k \cdot \lceil n/(k^3-k) \rceil)$	$2(k^3-k)$	0
Pung hybrid* [11, §4.4]	$4.5n$	$9k$	$\approx 2^{-20}$
3-way reverse cuckoo hashing (**this work**, §4.5)	$3n$	$1.5k$	$\approx 2^{-40}$

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 bucket in $\beta(DB)$ at most once, where β is the batch code. Building a multi-query PIR scheme from any of the above constructions leads to computational costs to the server linear in m, and network communication linear in b. 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., [61, 70, 71, 76]) but they either have no known constructions, or they seek additional properties (e.g., tolerate data erasures, optimize for the case where $n = b$, 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 to construct a PBC.

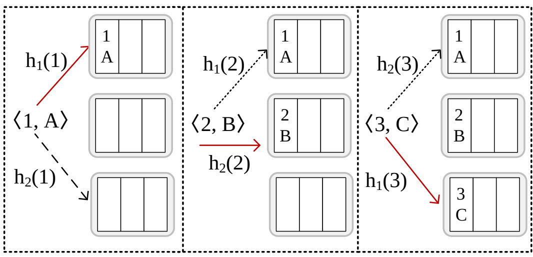


FIGURE 6—Logic for two-choice hashing [13] when allocating three key-value tuples to buckets: $\langle 1, A\rangle$, $\langle 2, B\rangle$, $\langle 3, C\rangle$. Tuples are inserted into the bucket least full. Arrows represent the choices for each tuple based on different hashes of the tuple's key (here we depict an optimistic scenario). The red solid arrow indicates the chosen mapping.

4.3 Randomized load balancing

A common use case for (non-cryptographic) hash functions is to build efficient data structures such as hash tables or dictionaries. In a hash table, the insert procedure consists of computing one or more hash functions on the key of the item 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 algorithm. For example, in multi-choice hashing [13, 64], the item is placed in the bucket least full among several candidate buckets. In Cuckoo hashing [66], items may move around following the Cuckoo hashing algorithm (we explain this algorithm in Section 4.5).

An ideal allocation results in items being assigned to buckets such that all buckets have roughly the same number of items (since this lowers the cost of lookup). In practice, there is load imbalance where some buckets end up having more elements than others; the extent of the imbalance depends on the allocation algorithm and the random choices that it makes. To look up an item by its key, one computes the different hash functions 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. An example of the insertion process for hashing with two choices is given 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 b bins problem, which arises during insertion. The items to be inserted into a hash table are the n balls, and the buckets in the hash table are the b bins; using w hash functions to hash a key to w candidate buckets approximates an independent and uniform random assignment of a ball to w bins. 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/w (all balls map to the same w candidate buckets), but there are much smaller bounds that hold with high probability [13].

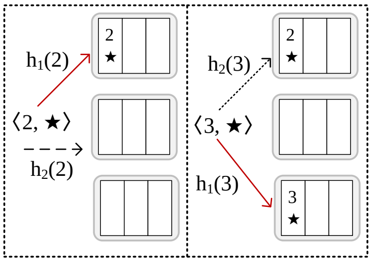
Interestingly, if we examine other scenarios abstracted by the balls and bins problem, a pattern becomes clear: the allocation algorithm is typically executed during data placement. In the hash table example, the allocation algorithm determines where to insert an element. In the context of a transport protocol [54], the allocation algorithm dictates on which path to send a packet. In the context of a job scheduler [65], the allocation algorithm 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". Reverse hashing achieves this.[4]

4.4 Reverse hashing

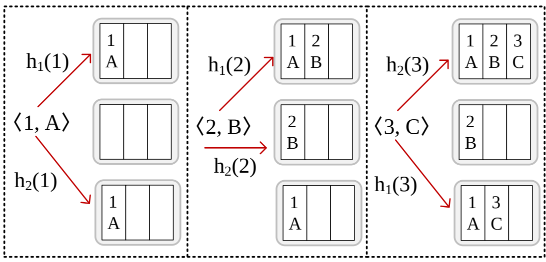
We start by introducing two principals: the *producer* and the *consumer*. The producer holds a collection of n items where each item is a key-value tuple. It is in charge of data placement: taking each of the n elements and placing them into buckets based on their keys following some allocation algorithm. 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. 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 b bins problem, and its goal is to reduce the instance's max load.

Note that the consumer is not inserting elements into buckets (that is the job of the producer). Instead, the consumer is placing "retrieval requests" into the buckets. The challenge is that any clever allocation chosen by the consumer must be *compatible* with the actions of the producer (who populates the buckets). That is, if the consumer, after running its allocation algorithm (e.g., multi-choice hashing) decides to retrieve items x_1, x_2, and x_3, from buckets 2, 3, and 7, it better be the case that

[4]Pung [11, §4.3] makes a similar observation but in a less general setting.



(a) consumer's simulation



(b) producer's allocation

FIGURE 7—Example of two-choice reverse hashing. (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.

the producer previously placed those elements in those exact buckets. We describe how we guarantee compatibility below.

Protocol. The consumer starts by imagining 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 algorithm (e.g., 2-choice hashing, cuckoo hashing) and populates the b buckets accordingly. The result is an allocation that balances the load of the k elements among the b buckets (as we discuss 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.

Guaranteeing that the consumer's allocation is compatible with the producer's actions is challenging. One reason is that the consumer's simulation is acting on k items whereas the real producer is acting on n items. If the allocation algorithm being used (by the consumer and the producer) is randomized or depends on prior choices (this is the case with multi-choice hashing schemes), the allocations will be different. For example, observe that if a producer generates the allocation in Figure 6 it would not be compatible with the consumer's simulation in Figure 7(a) despite both entities using the same algorithm (since the producer places the item under key "2" in the middle bucket, but the consumer's simulation maps it to the top bucket).

To guarantee compatibility we employ a simple solution: the producer follows the same allocation algorithm 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 algorithm chooses one among w candidate buckets to store an element, the producer stores the element in all w buckets. This ensures that regardless of which k elements are part of the consumer's simulation or which non-deterministic choices the algorithm makes, the allocations are always compatible (Figure 7(b)). Of course this means that the producer is replicating elements, which defeats the point of load balancing. However, PBCs only need load balancing during data retrieval.

4.5 A PBC from reverse cuckoo hashing

We give a construction that uses Cuckoo hashing [66] to allocate balls to bins. However, the same method can be used with other algorithms (e.g., multi-choice Greedy [13], LocalSearch [55]) to obtain different parameters. We give a brief summary of Cuckoo hashing's allocation algorithm below.

Cuckoo hashing algorithm. Given n balls, b buckets, and w independent hash functions $h_0, \ldots, h_{w-1}$ that map a ball to a random bucket, compute w candidate buckets for each ball by applying the w hash functions. For each ball x, place x in any empty candidate bucket. If none of the w candidate buckets are empty, select one at random, remove the ball currently in that bucket (x_{old}), place x in the bucket, and re-insert x_{old}. If re-inserting x_{old} causes another ball to be removed, this process continues recursively for a maximum number of iterations.

Construction. Let H be an instance (producer, consumer) of reverse hashing where the allocation algorithm is Cuckoo hashing with w independent hash functions and b bins (we discuss concrete values for w and b later in this section). We construct a (n, m, k, b, 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 b buckets using the indices in DB as keys and the elements as values. This results in $m = wn$ total elements distributed (not necessarily evenly) across the b buckets. Return the buckets.

GenSchedule(I). Given a set of indices I, follow H's consumer algorithm to allocate the k indices to the b buckets. Return the mapping of indices to buckets. If more than one index maps to the same bucket (i.e., if there are collisions), return $\perp$ instead.

Decode(W). Since Encode performs only replication, all codewords are 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 that codeword.

Concrete parameters. Analyzing the exact failure probability of Cuckoo hashing, and determining the constant factors, remains an open problem (see [42] for recent progress). However, several works [29, 68] have estimated this probability empirically for different parameter configurations. Following the analysis in [29, §4.2], we choose $w = 3$ and $b = 1.5k$. In this setting, the failure probability is estimated to be $p \approx 2^{-40}$ for $k > 200$ (for smaller k it is closer to 2^{-20}). This means that, assuming the mapping from indices to buckets is random, the probability that GenSchedule(I) returns $\perp$ for a set of indices I chosen independently from the hash functions is p. Figure 5 compares this result with existing batch code constructions and the scheme proposed in Pung [11, §4.4].

```
1:  function SETUP(DB)
2:     (C_0, ..., C_{b−1}) ← Encode(DB)
3:     for j = 0 to b − 1 do
4:        SETUP(C_j)                                   // See Fig. 1, Line 1
5:
6:  function MULTIQUERY(pk, I, {|C_0|, ..., |C_{b−1}|})
7:     σ ← GenSchedule(I)
8:     if σ ≠ ⊥ then
9:        // get an element for each bucket
10:       // pick a random index if the bucket is not used in σ
11:       for j = 0 to b − 1 do
12:          idx_j ← index for bucket j (based on σ and 𝒪)
13:          q_j ← QUERY(pk, idx_j, |C_j|)             // see Fig. 1, Line 4
14:       return q ← (q_0, ..., q_{b−1})
15:    else Deal with failure (see §5)
16:
17: function MULTIANSWER(q, (C_0, ..., C_{b−1}))
18:    for j = 0 to b − 1 do
19:       a_j ← ANSWER(q_j, C_j)                       // see Fig. 1, Line 9
20:    return a ← (a_0, ..., a_{b−1})
21:
22: function MULTIEXTRACT(sk, a, I, σ)
23:    // extract the codewords from the provided PIR answers into cw
24:    for j = 0 to b − 1 do
25:       cw_j ← EXTRACT(sk, a_j)                      // see Fig. 1, Line 14
26:    // select codewords from cw that are relevant to each index in I
27:    for i = 0 in k − 1 do
28:       W ← codewords from cw (based on σ[I_i])
29:       e_i ← Decode(W)
30:    return (e_0, ..., 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 $|C_i|$ is the size of bucket i. This protocol assumes a CPIR scheme with the API given in Figure 1.

5 Multi-query PIR from PBCs

We give the pseudocode for a PBC-based multi-query PIR 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, the bucket from which to retrieve an element using PIR. Because of the semantics of GenSchedule it is guaranteed that no bucket is queried more than once (or $\sigma = \perp$). As a result, the client can run one instance of PIR on each bucket. However, a challenge is determining *which* index to retrieve from each bucket: by assumption (of PIR) the client knows the index in *DB*, but this has no relation to the index of that same element in each bucket. To address this, we introduce an oracle $\mathcal{O}$ that provides this information (we discuss it below). If the client has nothing to retrieve from a given bucket, the client simply queries a random index for that bucket.

Constructing the oracle $\mathcal{O}$. There are several ways that the client can construct $\mathcal{O}$. The simplest solution is to obtain the mapping from each index in *DB* to the corresponding indices in each bucket. While this might sound unreasonable, observe that PIR has an implicit assumption that the client knows the index in *DB* of the desired element. The client could use the same technique to obtain the corresponding w indices in $\mathcal{B}(DB)$. For example, in the Pung communication system [11], clients obtain this mapping in a succinct Bloom filter [17].

Another option is for the client to fetch elements in PIR not by index but by a label using PIR-by-keywords [32]. Examples of labels include the name or UUID of a movie, the index in the original *DB* (in this case elements would need to be stored as key-value tuples, where the key is the label). 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 w candidate buckets), which yields $\mathcal{O}$. Furthermore, this process is incremental for many hashing schemes: 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.

Malicious placement of elements. In cases where the server is malicious—rather than semi-honest (or honest but curious)—the server has full control over where to place elements. As a result, the server could place specific elements at indices that can never be retrieved together (i.e., at sets of indices where GenSchedule returns $\perp$). This opens the door to attacks where the server selectively makes certain combinations of elements not retrievable in hopes of observing a client's reaction and breaking the privacy guarantees. Note that a similar attack already exists in the single-query PIR case: the server can selectively place an incorrect element (or garbage) at a particular index and can wait to see if a client complains or not (thereby learning that the index that the client requested was one that contained garbage or not). To address this style of "selective failure" attacks, additional mechanisms are needed. A common solution is to ensure that a client's reaction remains independent of whether or not queries succeed. This guarantees that the attack does not violate privacy. Instead, it violates availability, which a malicious server could violate anyway by not answering queries.

Dealing with failures in the schedule. If the PBC being used has $p > 0$, 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 8). 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 may not be ephemeral so the client can try again at a later time (presumably with a new set of indices). Lastly, the client can fail silently. Which of these strategies is taken by a client depends on the application. In any case, it is imperative that the application's failure-handling logic is designed to not reveal information about a client's indices.

6 Implementation

SealPIR implements XPIR's protocol [8] atop the SEAL homomorphic encryption library [4] (version 2.3.0-4). This is around 2,000 lines of C++ and Rust. The most intricate component is EXPAND (Figure 3) which requires the substitution homomorphic operation (§3.1). We implement this operation in SEAL by porting the Galois group actions algorithm from Gentry et al. [44, §4.2]. We discuss this in detail in Appendix A.1.

SealPIR exposes the API in Figure 1. A difference with XPIR is that substitution requires auxiliary cryptographic material to be generated by the client and be sent to the server (see Appendix A.1). However, a client can reuse this material across all of its requests and it is relatively small (2.9 MB per client).

Encoding elements as FV plaintexts. In SealPIR, an FV plaintext is represented as an array of 64-bit integers, where each integer is mod t. Each element in the array represents a coefficient of the corresponding polynomial. We encode an element $e \in DB$ into an FV plaintexts $p(x)$ by storing $\log(t)$ bits of e into each coefficient of $p(x)$. If elements are small, we store many elements into a single FV plaintext (for example, the first element is stored in the first 20 coefficients, etc.). This reduces the total number of FV plaintexts in the database, and consequently the computational costs of PIR.

Optimization to EXPAND. 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 14–16 of Figure 3, EXPAND multiplies the n ciphertexts by the inverse of m where $m = 2^\ell$ (the goal of this multiplication is to ensure that all ciphertexts encrypt either 0 or 1). Instead, we change the plaintext modulus of the n ciphertexts from $t = 2^y$ to $t' = 2^{y-\ell}$, which allows us to avoid the plaintext multiplications and the inversion, and reduces the noise growth of EXPAND. The result is $n - 1$ ciphertexts encrypting 0, and one ciphertext encrypting 1, as we expect. This optimization requires t to be divisible by m rather than being an odd integer. One drawback is that the server must represent the database using FV plaintexts defined with the plaintext modulus t' (rather than t). As a result, we can pack fewer database elements into a single FV plaintext.

To set t', we find the largest integer value of $\log(t')$ for which the following inequality holds:

$$\log(t') + \lceil \log(\lceil \sqrt[d]{n_{fv}} \rceil) \rceil \leq \log(t) \tag{1}$$

$$n_{fv} = \lceil n/\alpha \rceil$$

$$\alpha = \lfloor N \log(t')/\beta \rfloor$$

Here α is the number of elements of size β bits that can be packed into a single FV plaintext, and n_{fv} is the number of FV plaintexts needed to represent n elements of size β.

Implementation of PBCs. We also implement *mPIR*, a multi-query PIR library based on PBCs. mPIR implements 5 different PBC constructions based on reverse hashing (§4.4) with different allocation algorithms (e.g., two-choice hashing, Cuckoo hashing, the Hybrid allocation scheme in Pung [11]). 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.

7 Evaluation

Our evaluation answers four questions:

1. What are the concrete resource costs of SealPIR, and how do they compare to XPIR?
2. What is the throughput and latency achieved by SealPIR under different deployment scenarios?
3. What are the concrete benefits provided by PBCs, and how do they compare to existing batch codes?
4. What is the impact of using SealPIR and mPIR on a representative system?

Experimental setup. We run our experiments using Microsoft's Azure instances in three data centers: West US, South India, and West Europe. We run the PIR servers on H16 instances (16-core 3.6 GHz Intel Xeon E5-2667 and 112 GB RAM), and clients on F16s instances (16-core, 2.4 GHz Intel Xeon E5-2673 and 32 GB RAM), all running Ubuntu 16.04. We compile all our code with Rust's nightly version 1.25. For XPIR, we use the publicly available source code [9] and integrate it into our testing framework using Rust wrappers. 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.

Parameters. We choose security parameters for FHE following XPIR's latest estimates [5], which are based on the analysis and tools by Albrecht et al. [10]. We set the degree of ciphertexts' polynomials to 2048, and the size of the coefficients to 60 bits (N and q in Section 3). Specifically, SEAL uses a value of $q = 2^{60} - 2^{18} + 1$, whereas XPIR uses $q = 2^{61} - i \cdot 2^{14} + 1$, for various values of i [6].

Each database element is 288 bytes. We choose this size since the Pung communication system uses 288-byte messages (§7.4). Unless otherwise stated, SealPIR uses a plaintext modulus t=2^{23}. A larger t leads to lower network and computational costs, but might cause noise to grow too much, preventing ciphertexts from decrypting successfully (we lower t in some experiments to ensure that we can always decrypt the result). For XPIR, we use $\alpha = 14$, meaning that we pack α elements into a single XPIR plaintext, thereby reducing the number of elements stored in the database by a factor of α. For 288-byte elements and our security parameters, setting $\alpha = 14$ has roughly the same effect as setting $t = 2^{23}$ in SealPIR (although our optimization to EXPAND, which we discuss in Section 6, means that SealPIR ultimately packs fewer elements together than XPIR).

7.1 Cost and performance of SealPIR

To evaluate SealPIR, we run a series of microbenchmarks to measure: (i) the time to generate, expand, and answer a query; (ii) the time to extract the response; and (iii) the time to preprocess the database. We study several database sizes and repeat the same experiment for XPIR using two different dimension parameters d (§3.4). Figure 9 tabulates our results.

CPU costs. We find that the computational costs of query generation are an order of magnitude lower under SealPIR than under XPIR. This is because the client in SealPIR generates d ciphertexts as a query rather than $d\sqrt[d]{n}$ ciphertexts as in

	XPIR ($d = 2$)			XPIR ($d = 3$)			SealPIR ($d = 2$)		
database size (n)	65,536	262,144	1,048,576	65,536	262,144	1,048,576	65,536	262,144	1,048,576
client CPU costs (ms)									
QUERY	13.83	27.57	55.14	4.98	8.03	12.74	3.37	3.37	3.37
EXTRACT	0.34	0.29	0.30	2.47	2.49	2.57	1.37	1.39	1.69
server CPU costs (sec)									
SETUP	0.15	0.57	2.27	0.15	0.58	2.32	0.23	1.04	4.26
EXPAND	N/A	N/A	N/A	N/A	N/A	N/A	0.05	0.11	0.23
ANSWER	0.21	0.63	2.12	0.27	0.78	2.52	0.13	0.5	2.01
network costs (KB)									
query	4,384	8,768	17,536	1,632	2,560	4,064	64	64	64
answer	256	256	256	1,824	1,952	1,952	256	256	256

FIGURE 9—Microbenchmarks of CPU and network costs for XPIR and SealPIR under varying database sizes (n). Elements are of size 288 bytes.

XPIR (§3.4). When it comes to the server, SealPIR's EXPAND procedure introduces CPU overheads of 11% to 38% (over answering a query vector directly). While this is high, it results in significant network savings (which we discuss below). Furthermore, even with the overhead of EXPAND, the cost of answering a query in SealPIR is comparable to XPIR.

We note that larger values of d lead to more computation for the server for two reasons. First, structuring the database as a d-dimensional hyperrectangle often requires padding the database with dummy plaintexts to fit all dimensions. Second, as we discuss in Section 3.4, the ciphertext expansion factor effectively increases the size of the elements by a factor of F after processing each dimension, necessitating more computation.

Network costs. For network costs, SealPIR enjoys a significant reduction owing to its query encoding and EXPAND procedure (§3.3). For larger databases, the query size reductions over XPIR are 274× when $d = 2$, and 63× when $d = 3$.

7.2 SealPIR's response time and throughput

While microbenchmarks are useful for understanding how SealPIR compares to XPIR, another important axis is understanding how these costs affect response time and throughput.

7.2.1 Response times

To measure response time, we run experiments where we deploy a PIR server in Azure's US West data center, and place a PIR client under four deployment scenarios. We then measure the time to retrieve a 288-byte element using SealPIR, XPIR, and `scp` (i.e., secure copy command line tool). We use `scp` to represent a client downloading the entire database (naive PIR).

Deployment scenarios

intra-DC: the client and the server are both in the US West data center. The bandwidth between the two VMs is approximately 3.4 Gbps (measured using the `iperf` measurement tool). This scenario is mostly pedagogical since it makes little sense to use PIR inside two VMs in the same data center controlled by the same operator. It gives an idea of the performance that PIR schemes could achieve if network bandwidth were plentiful.

inter-DC: the client is placed in the South India data center. The bandwidth between the two VMs is approximately 800 Mbps. This scenario represents clients who deploy their applications in a data center (or well-provisioned proxy) that they trust, and access content from an untrusted data center.

home network: the client is placed in the South India data center. However, we use the `tc` traffic control utility to configure the Linux kernel packet scheduler in both VMs to maintain a 20 Mbps send rate. We choose this number as it is slightly over the mean download speed in the U.S. (18.7 Mbps) according to Akamai's latest connectivity report [1, §4]. This scenario is optimistic to XPIR since it ignores the asymmetry present in home networks where the uplink bandwidth is typically much lower (meanwhile in XPIR, the queries are large). Nevertheless it gives a rough estimate of a common PIR use case in which a client accesses an element from their home machine.

mobile network: the client is placed in the South India data center. We use `tc` to configure VMs to maintain a 10 Mbps send rate. We choose this number as it approximates the average data speed achieved by users across all U.S. carriers according to OpenSignal's 2017 State of Mobile Networks report [2] and Akamai [1, §8]. As with the home network, this scenario is optimistic (for XPIR) as it ignores the discrepancy between download and upload speeds. It represents the use of PIR from a mobile or data-limited device, which is a common deployment for applications such as private communication (§7.4).

Results. Figure 10 depicts the results. At very high speeds (intra-DC), naive PIR (`scp`) is currently the best option, which is not surprising given the computational costs introduced by PIR. In this regime, SealPIR is competitive with both instances of XPIR, although our implementation falls behind on the largest database size. The primary issue is that, for a database with $n = 2^{22}$ elements, our optimization of EXPAND makes the plaintext modulus very small ($t' = 2^{12}$, see Equation 1 in Section 6). This causes SealPIR to use many more plaintexts than XPIR. For even larger databases, since we must use a higher dimension anyway (§3.5), the difference in the number of plaintexts between XPIR and SealPIR (for the same d) becomes less prominent until n is large enough that the second operand in Equation 1 approaches $\log(t)$ again.

When it comes to lower network speeds, XPIR and SealPIR significantly outperform `scp`. As bandwidth decreases (home, mobile), SealPIR's lower network consumption and competitive CPU costs yield up to a 42% reduction in response time.

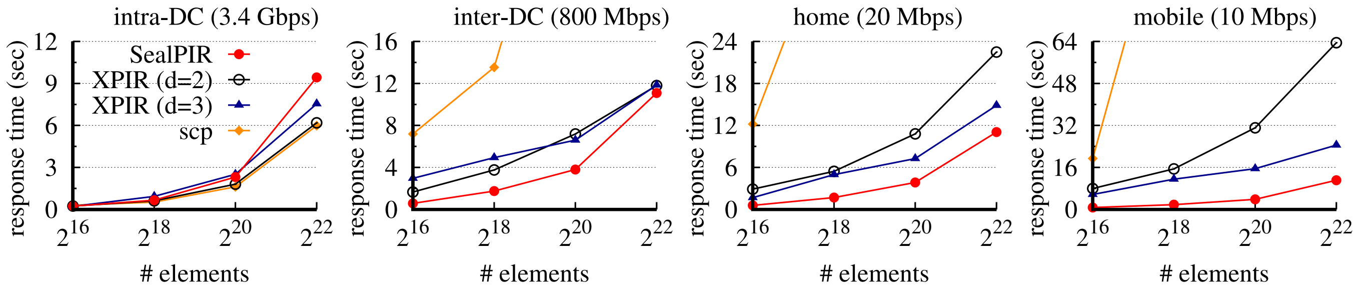


FIGURE 10—Mean response time experienced by a client under different deployments (see text for a description of network conditions) with different PIR schemes. When the network bandwidth is plentiful (intra-DC), downloading the entire database (`scp`) achieves the lowest response time. However, when the network bandwidth is limited (home, mobile), SealPIR achieves the lowest response time.

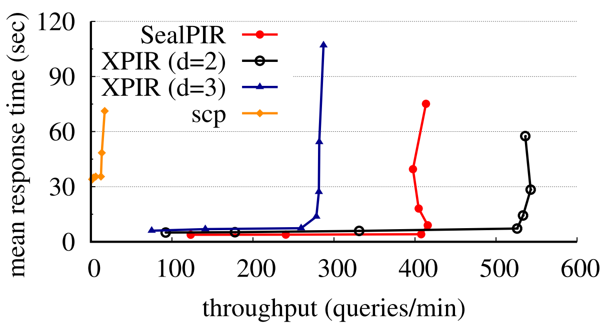


FIGURE 11—Comparing throughput vs. mean response time under SealPIR and XPIR (with $d = 2$ and $d = 3$) when using a database with 2^{20} elements where each element is 288 bytes long. We find that XPIR with $d = 2$ saturates at 9 requests/second whereas SealPIR saturates at 7 requests/second (a 23% reduction in throughput). When XPIR uses $d = 3$, SealPIR achieves about 50% higher throughput.

7.2.2 Throughput

We deploy the PIR server in Azure's US West data center, but access it with an increasing number of concurrent PIR clients deployed across the South India and EU West data centers. We then measure the number of requests serviced per minute at the server, and the request completion times at the clients. Figure 11 depicts the results of running from 4 to 256 clients each requesting one 288-byte element from a database with 2^{20} entries. In our experiments, we ensure that the bottleneck is the server's CPU or WAN network connection, and not the clients or some link between specific data centers.

We find that SealPIR achieves a 50% higher throughput than XPIR with $d = 3$, but a 23% lower throughput than XPIR with $d = 2$. Most of the difference can be attributed to EXPAND, but we believe that with further engineering we can close this gap (since SealPIR is comparable to XPIR according to microbenchmarks). Compared to naive PIR via `scp`, SealPIR and XPIR achieve over 20× higher throughput since the primary bottleneck in naive PIR is network bandwidth and not CPU (which is the bottleneck for both SealPIR and XPIR).

7.3 Benefits of PBCs

To understand how PBCs can improve throughput and what type of network overhead they add, we repeat the microbenchmark experiments of Section 7.1, but this time we use mPIR (with Cuckoo hashing, see Section 4.5). To put the benefits and costs in context, we also evaluate the multi-query PIR scheme of Pung [11]. Pung's protocol, like PBCs, is probabilistic and improves over existing batch codes in terms of costs. In this experiment we use SealPIR with $t = 2^{20}$ as the underlying PIR library and change only the multi-query scheme being used.

Figure 12 gives the results. We find that mPIR does a better job than Pung's scheme at amortizing CPU costs across all batch sizes. This is a direct effect of the Cuckoo PBC producing fewer total codewords (see Figure 5), since computational costs are proportional to the number of elements after encoding (m). At $k = 256$ and 288-byte elements, mPIR achieves a 2.6× reduction in CPU cost for the server when answering queries over Pung's scheme. Over running k parallel instances of PIR, the per-request CPU cost of mPIR is 40.5× lower.

The difference in network costs between Pung's scheme and mPIR is more pronounced. This owes to Pung's scheme building on the subcube batch code of Ishai et al. [52] which creates a large number of buckets (see Figure 5); to preserve privacy, clients must issue a PIR query to each bucket. In terms of concrete savings, mPIR is 6× more network efficient (upload and download) than Pung's scheme. Considering that mPIR also has a lower failure probability (around 2^{-40}, compared to Pung's 2^{-20}), this suggests that mPIR is an attractive replacement to Pung's multi-query protocol, offering improvements on all axes.

Observe that at $k = 256$, mPIR's download costs are the same as running k-parallel instances of PIR. This is counterintuitive since mPIR results in 50% more answers (the extra answers are dummies that hide which buckets are of interest to the client; see Section 5). However, each answer in mPIR contains fewer ciphertexts because of the interaction between SealPIR and mPIR. In particular, mPIR encodes the 2^{20}-entry databases into $1.5k = 384$ buckets, and replicates elements 3 times. Buckets therefore have on average 2^{13} elements. Recall from Section 3 that if $d > 1$, the number of ciphertexts in an answer depends on the expansion factor $F = 2\log(q)/\log(t')$. Furthermore, Equation 1 (Section 6) shows that t' is larger for smaller databases. Indeed, for the original 2^{20}-entry database, $t' = 2^{10}$ ($F = 12$), whereas $t' = 2^{15}$ for the average bucket ($F = 8$). Consequently, for our choice of parameters, the total download communication ends up being the same: $256 \cdot 12 = 384 \cdot 8$ ciphertexts.[5]

[5]Similar benefits apply to Pung's scheme when used with SealPIR: observe in Figure 12 that as k goes from 16 to 64, the amortized answer size goes down.

	single-query	Pung's multi-retrieval			mPIR (Cuckoo hashing)		
batch size (k)	1	16	64	256	16	64	256
client CPU costs (ms)							
MultiQuery	3.07	29.03	28.50	28.58	6.45	5.26	4.92
MultiExtract	2.51	20.00	16.27	16.36	3.26	3.25	2.70
server CPU costs (sec)							
MultiSetup	6.1	2.02	0.64	0.30	1.50	0.38	0.12
MultiAnswer	3.24	1.37	0.49	0.21	0.69	0.23	0.08
network costs (KB)							
query	64	577	577	577	96	96	96
answer	384	2,885	2,308	2,308	480	480	384

FIGURE 12—Per-request (amortized) CPU and network costs of two multi-query PIR schemes on a database consisting of 2^{20} elements, with varying batch 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 SealPIR with $t = 2^{20}$ and elements are 288 bytes.

Note that this parity in download cost is not true in general; it is a result of the particular parameters used in this case. In fact, because of Equation 1 (§6), we can even achieve *lower* amortized download costs. Without EXPAND's optimization, this would not be the case: in some sense, the optimization introduces communication overhead to fetching elements from databases with many entries and mPIR amortizes that overhead. As an aside, Equation 1 does not affect upload costs; these costs increase by 50% since the client is sending 50% more queries.

7.4 Case study: Pung with SealPIR and mPIR

To evaluate the end-to-end benefits that SealPIR and mPIR provide to actual applications, we modify the available implementation of Pung [3]. Pung is a messaging service that allows users to exchange messages in rounds without leaking any metadata (who they are talking to, how often, or when). We choose Pung because it uses XPIR to achieve its privacy guarantees, and because it also relies on multi-query PIR to allow clients to 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.

Experiment. We have clients send one message and retrieve k messages (this models clients engaging in group conversations). We run the system in a close-loop and advance rounds as soon as all clients have sent and retrieved the messages. To experiment with many clients we employ the same simulation technique used in Pung: we have 32 real clients accessing the server, and simulate additional clients by pre-populating the server's database with random messages.

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 code base for all batch sizes greater than 1. There are three reasons for this. First, Pung's multi-retrieval produces 50% more codewords than mPIR, and therefore has to do more processing. Second, Pung's multi-retrieval produces 7× more buckets than mPIR. This forces Pung to run XPIR on many small databases that contain an average of 500 to 8,000 elements (depending on the batch size), which exacerbates XPIR's fixed costs.

Last, even though SealPIR incurs additional CPU costs than

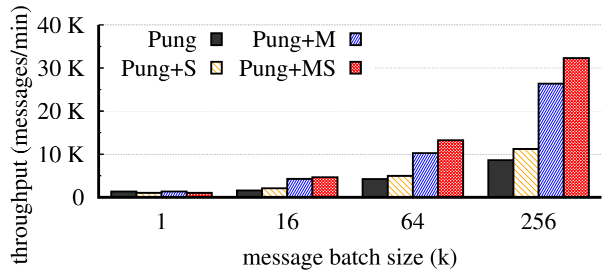


FIGURE 13—Throughput of Pung on one server with 256K users, each retrieving k 288 byte messages per round. The label "Pung" indicates the implementation as given in [3], with updated parameters (§7). "Pung+S" is a version of Pung that uses SealPIR with $t = 2^{20}$; "Pung+M" is a version of Pung that uses mPIR; and "Pung+MS" is a version of Pung that uses both mPIR and SealPIR.

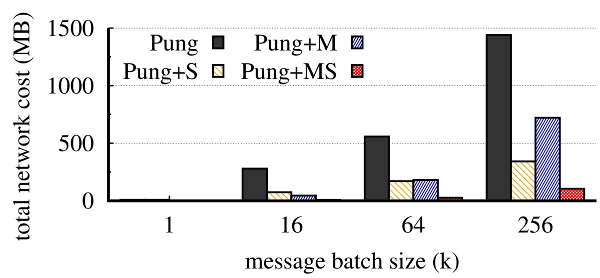


FIGURE 14—Per-user total network cost (upload and download) of a Pung deployment with 256K users. Each user retrieves k 288-byte messages. See Figure 13 for an explanation of the legend.

XPIR ($d = 2$) on large databases as we show in Section 7.1 (this is also why Pung has higher throughput than Pung-MS when the batch size is 1), SealPIR is slightly faster when the database is small (see the column with 65,536 elements in Figure 9). Ultimately, we find that if clients retrieve $k = 64$ messages, the throughput of Pung+MS is 3.1× higher than that of Pung.

When it comes to network costs, the benefits of SealPIR and mPIR are 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 fewer buckets result in savings of over 36×. In particular, the per-client communication costs are cut down to 7.7 MB per round for $k = 16$ (versus 279 MB 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 3.4) the computation consists of several matrix-vector products. We can therefore implement the optimization described by Beimel et al. [16] 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 reverse hashing 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). We could also 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 [25]. Optimal solutions for this problem can be computed in polynomial time [30], and linear time approximations also exist [25, 35, 41].

Changes made on November 16, 2020

We have fixed a typo in Figure 4. In the prior version, the array with the plaintext coefficients after Expand ($j = 0$) had the coefficient 2 in the second entry (representing $2x$) instead of the third entry (representing $2x^2$). The latter is the correct one.

Acknowledgments

We thank Esha Ghosh, Trinabh Gupta, Jay Lorch, Michael Walfish, and Sergey Yekhanin for their helpful feedback. We thank Peter Rindal for sharing his estimates of Cuckoo hashing's failure probability. Sebastian Angel was supported by NSF grant CNS-1514422 and AFOSR grant FA9550-15-1-0302.

References

[1] Akamai state of the internet connectivity report. `https://www.akamai.com/fr/fr/multimedia/documents/state-of-the-internet/q1-2017-state-of-the-internet-connectivity-report.pdf`, May 2017.

[2] Opensignal state of mobile networks: Usa. `https://opensignal.com/reports-data/national/data-2017-08-usa/report.pdf`, Aug. 2017.

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

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

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

[6] XPIR NFLParams. `https://github.com/XPIR-team/XPIR/blob/master/crypto/NFLParams.cpp`, June 2017.

[7] Internet providers with data caps. `https://broadbandnow.com/internet-providers-with-data-caps`, Jan. 2018.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

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

[19] E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can we access a database both locally and privately? In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.

[20] 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.

[21] Z. Brakerski and V. Vaikuntanathan. Efficient fully homomorphic encryption from (standard) LWE. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 2011.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. In *Proceedings of the Theory of Cryptography Conference (TCC)*, Nov. 2017.

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

[28] H. Chen, K. Han, Z. Huang, A. Jalali, and K. Laine. Simple encrypted arithmetic library v2.3.0. `https://https://www.microsoft.com/en-us/research/publication/simple-encrypted-arithmetic-library-v2-3-0/`, Dec. 2017.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

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

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

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

[42] A. Frieze, P. Melsted, and M. Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM Journal on Computing*, 40(2), Mar. 2011.

[43] 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.

[44] 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.

[45] 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.

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

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

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

[56] 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.

[57] 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.

[58] 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.

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

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

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

[62] 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.

[63] 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.

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

[65] 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.

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

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

[68] B. Pinkas, T. Schneider, and M. Zohner. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security*, 21(2), Jan. 2018.

[69] 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.

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

[71] 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.

[72] 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.

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

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

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

[76] 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.

[77] 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.

[78] 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.

Appendix

A Query expansion

A.1 Substitution operator

We now give details on how the substitution operator is implemented. Let Φ_i be the i-th cyclotomic polynomial.[6] As we discuss in Section 3.1, we pick $\Phi_i = x^N + 1$, where N is a power of two (hence $i = 2N$). Recall from that same section that FV plaintexts are polynomials in the ring $R_t = \mathbb{Z}_t[x]/\Phi_i(x)$, and ciphertexts are two polynomials, each in the ring $R_q = \mathbb{Z}_q[x]/\Phi_i(x)$. The secret key sk is a randomly sampled polynomial in R_2.

Let $p(x)$ be the plaintext encrypted by ciphertext $c = (c_0, c_1)$. Our goal is to substitute in $p(x)$ every instance of x with x^k for some integer k, by operating directly on c. Gentry et al. [44, §4.2] show that if $k \in \mathbb{Z}_i^*$ (i.e., k is odd so that it is coprime with i), performing the substitution directly on the ciphertext polynomials (c_0, c_1) and the secret key achieves this goal.

Specifically, let $c^{(k)}$ be the result of replacing every instance of x in the ciphertext polynomials c_0 and c_1 with x^k. Similarly, let $sk^{(k)}$ be the result of replacing every instance of x in the secret key sk with x^k. The result of decrypting $c^{(k)}$ with $sk^{(k)}$ is therefore $p(x^k)$—which is exactly what we want.

One issue with the above is that EXPAND (Figure 3) uses the output ciphertext after substitution, $c^{(k)}$, and adds it to the input ciphertext c in each iteration of the inner loop (see Lines 10 and 11). This operation is not well defined since both ciphertexts are encrypted under different keys (substitution essentially changes the key under which the ciphertext is encrypted). To address this, we perform an operation called *key switching* [20], which allows us to transform an encryption of $c^{(k)}$ under some public key associated with $sk^{(k)}$, to an encryption of $c^{(k)}$ under some public key associated with the original key sk (which is the key under which c is also defined).

Note that the server needs some auxiliary information in order to perform key switching. In particular, the server needs a key-switching matrix showing how to go from $sk^{(k)}$ to sk (see [44, Appendix D] for details), which the client must generate. Since in EXPAND substitution is called for different values of k (notice that in Line 10 and 11 in Figure 3 the value of k depends on j), the client must provide a key-switching matrix for each of them. However, this only needs to be done once and it depends only on the size of the database.

The above allows the server to compute EXPAND: the server first does the substitution followed by the appropriate key switch, and finally performs the addition in the inner loop.

A.2 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 $query = Enc(x^i)$ be the client's encoding of index i. The n output ciphertexts $o_0, \ldots, o_{n-1}$ of EXPAND($query$) satisfy, for all $0 \leq k \leq n-1$:*

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

[6] The i-th cyclotomic polynomial is the unique irreducible polynomial with integer coefficients that is a factor of $x^i - 1$ but not of $x^j - 1$ for any $j < i$.

Proof. It suffices to prove the case for $n = 2^\ell$. For $j = \{0, 1, \ldots, \ell - 1\}$, we claim that after the j^{th} iteration of the outer loop, we have $ciphertexts = [c'_0, \ldots, c'_{2^{j+1}-1}]$ such that

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

We prove the claim by induction on j. The base case $j = 0$ is explained in the main text of Section 3.3. Suppose the claim is true for some $j \geq 0$. Then in the next iteration, we compute an array $ciphertexts'$.

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

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

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

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

However, note that $i < n = 2^\ell$, so $i \equiv k \pmod{2^\ell}$ implies $i = k$. Hence $ciphertexts[k]$ is either an encryption of 0 or an encryption of 2^ℓ. To obtain an encryption of 0 or 1, we multiply $ciphertexts[k]$ by the inverse of 2^ℓ modulo t in the last step (Figure 3, Line 14). ☐

A.3 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 [28] 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 monomial x^j (for some j) with coefficient 1 does not change the noise, and plain multiplication by a constant α 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. ☐

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 15—Cost of operations for five PBCs implemented as part of mPIR. The collection size (n) is 524,288 and the batch size (k) is 64. Each element in the collection is 288 bytes. 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 balls to a single bin during the producer's allocation, and the consumer uses a hash function during simulation (this variant has a high failure rate which we improve by replicating buckets).

B Cost of PBC variants

We have implemented five PBCs with different allocation algorithms using reverse hashing. 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 that reduces the number of codewords (m) and buckets (b). Our hypothesis is that PBCs that produce low values of m and b result in more expensive encoding and schedule generation procedures.

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 15 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, b bins, and w 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). 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. [52], 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 to confirm that all PBCs have reasonably efficient encoding, decoding, and schedule generation procedures. As such, our evaluation (§7) focuses only on the Cuckoo variant since it yields the most efficient parameters, and the second lowest failure probability (k-way replication never fails, but has a very high value of m).