WhisPIR: Stateless Private Information Retrieval with Low Communication

Leo de Castro [†◇], Kevin Lewi[†], and G. Edward Suh[†§]
[†]Meta, [◇]Massachusetts Institute of Technology, [§]Cornell University

ABSTRACT

Recent constructions of private information retrieval (PIR) have seen significant improvements in computational performance. However, these improvements rely on heavy offline preprocessing that is typically difficult in real-world applications. Motivated by the question of PIR with no offline processing, we introduce WhisPIR, a fully stateless PIR protocol with low per-query communication. WhisPIR clients are all ephemeral, meaning that they appear with only the protocol public parameters and disappear as soon as their query is complete, giving no opportunity for additional "offline" communication that is not counted towards the overall query communication. As such, WhisPIR is highly suited for practical applications that must support many clients and frequent database updates.

We demonstrate that WhisPIR requires significantly less communication than all other lattice-based PIR protocols in a stateless setting. WhisPIR is outperformed in computation only by SimplePIR and HintlessPIR when the database entries are large (several kilobytes). WhisPIR achieves this performance by introducing a number of novel optimizations. These include improvements to the index expansion algorithm of SealPIR & OnionPIR that optimizes the algorithm when only one rotation key is available. WhisPIR also makes novel use of the *non-compact* variant of the BGV homomorphic encryption scheme to further save communication and computation. To demonstrate the practicality of WhisPIR, we apply the protocol to the problem of secure blocklist checking, an important user-safety application in end-to-end encrypted messaging.

1 INTRODUCTION

Private information retrieval (PIR) is a protocol to query a database of records without revealing which record is being retrieved to the machine hosting the database. PIR has a wide variety of applications such as certificate transparency [LG15], metadata-hiding messaging [AS16, ACLS18], password-breach alerting [TPY+19, LPA+19, PIB+22], private contact discovery [KRS+19], and many others.

Recent works on PIR protocols have yielded exciting improvements in computational performance [MW22, HHCG+23, ZPSZ23, HDCGZ23]. However, these recent advancements heavily rely on *offline processing*. This processing incurs substantial communication and computation overhead while also producing a *state* that must be stored and accessed during the online phase. There are two major varieties of PIR state, both of which incur significant penalties.

- **Client-side State.** Many PIR protocols require each client to store a state that depends on the database to assist with query generation and processing. In some protocols, this state is fixed for any number of online queries, such as the database digest in [HHCG+23, ZPSZ23], or it can be consumed in the online phase, such as the per-query state in [HDCGZ23]. A state that is consumed must be replenished in another offline phase. When the database changes, this state must be updated for *each client*, and there is currently no effective solution to update a client-side state in a communication-efficient manner. This precludes applications with frequent database updates. Furthermore, a per-query client-side state assumes consistent opportunities to preprocess client queries as well as persistent client storage between the offline and online phases.
- **Server-side State.** Some PIR protocols have queries where the majority of the communication consist of *reusable* elements that are common across all queries from a specific client. It is common for these protocols to have the client upload these common elements in an offline phase, and during the online phase the server accesses these elements to process the query for a particular client. This includes the per-client state in [MW22], which consists of a client's FHE evaluation keys. In addition to the communication of the upload, this incurs significant storage requirements on the server itself. For applications with many clients, this storage can quickly outgrow the database size.

To illustrate the barriers introduced by these state requirements, we discuss several practical applications of PIR where a stateful PIR protocol will incur too much overhead.

Private contact discovery. Consider the application of private contact discovery. In this application, a server stores a list of registered users, and new users joining a service want to check this list against their contacts without revealing their contacts to the server. Observe that this natural application negates the assumptions of any stateful PIR scheme. The clients in this application have no opportunity for offline preprocessing as they are joining the service for the first time. Any per-client state required by the protocol represents a significant storage overhead, since each database entry corresponds to a client. In addition, each new client joining the service represents an update to the database, meaning that any client-side state must be updated with each new client that joins the service.

Detecting malicious links in E2EE messaging. Consider the setting of device-based end-to-end encrypted (E2EE) messaging, such as in Signal or WhatsApp. The core guarantee of end-to-end encryption in these applications ensures that clients can exchange messages using the server as an intermediary message passer without revealing any message content to the server. However, a common side-effect with E2EE messaging apps is the reduced ability for the intermediary server to detect phishing attempts that involve an adversary sending a message containing a malicious URL.

PIR offers a way for this attack vector to be addressed in a privacy-preserving manner: assuming that the server has access to an up-to-date bank of malicious URLs and fake domains, the client's device

can perform a query on a URL they received through the messaging app using PIR with the server in order to learn whether or not this URL exists in the server's malicious URL bank. This Boolean outcome can then be used to alert the user about the link they received, (i.e. supplementing the link with a warning message within the user's conversation) all without revealing to the server the query or the outcome.

However, note that PIR in this setting must operate under a set of specific constraints: a constantly-updating database, high traffic (on the order of hundreds of millions of users frequently sharing URLs), and relatively relaxed requirements on latency, since the outcome of the PIR query does not necessarily need to block the transmission of the link, but can be processed after the message has already been delivered. Specifically, due to the fact that a large-scale messaging app that deploys this solution would likely be engaging in millions of these queries per second, often attached to much smaller payloads, the network bandwidth incurred by the PIR protocol becomes a bottleneck in the overall practicality of the solution.

1.1 Prior Work

The starting point for this work is the Spiral PIR protocol of Menon and Wu [MW22]. In this protocol, a client's PIR query is encrypted with a fully homomorphic encryption (FHE) scheme [Gen09, BGV12], then the lookup function is evaluated homomorphically on this encrypted index. In all practical FHE schemes, homomorphic operations require additional *evaluation* keys for efficiency and to maintain the compactness of the ciphertext. The Spiral protocol uploads the evaluation keys for each client in an offline phase, then reuses these evaluation keys to process all subsequent client queries. Our work can be viewed as taking the Spiral protocol and compressing the FHE evaluation keys to the point where they can be uploaded along with the actual query, removing any per-client state that must be stored on the server. We introduce a number of optimizations to make using these compressed evaluation keys more efficient.

The Spiral protocol builds on several prior works with a similar pipeline of performing homomorphic evaluation of an encrypted index. In particular, the index expansion algorithm of SealPIR & OnionPIR [ACLS18, MCR21] is a major focus of this work.

A recent line of work beginning with the SimplePIR protocol of Henzinger et al. [HHCG$^+$23] and continuing with the TipToe protocol [HDCGZ23] focuses on optimizing the server computation time of processing a PIR query. While this is motivated by the relatively slow performance of prior PIR constructions, both of these works rely on heavy offline communication and computation that are infeasible in many PIR applications. When the PIR server is deployed by a large company or some other entity with significant resources, server computation time can be heavily parallelized and is typically not the bottleneck, especially for smaller databases. In contrast, the communication between the PIR server and the clients is fixed by the protocol and cannot be optimized with additional local resources. If this communication is beyond what a client network can easily support, the application is simply not usable by these clients. This is a major motivation of our work; enabling PIR for large-scale providers with the resources to optimize server computation by minimizing the per-query communication in settings where an offline phase cannot be supported.

We conclude by mentioning a very recent prior work called HintlessPIR [LMRSW23], which can be viewed as a stateless version of the TipToe protocol. In this protocol, the FHE evaluation keys are transferred along with the query, just like in our protocol. However, the computation to process the query is essentially the same as SimplePIR and TipToe, while our protocol's computation is much more similar to the Spiral protocol. We demonstrate in Section 4 that our protocol outperforms HintlessPIR in communication for all database sizes and strictly outperforms HintlessPIR (better communication and computation) when the database entries are small (less than a few kilobytes).

1.2 Our Contributions

We introduce WhisPIR, a stateless PIR protocol with low per-query communication. WhisPIR outperforms all other PIR protocols in communication in a stateless setting. When database entries are small, WhisPIR also outperforms all prior works in computation except for SimplePIR, which can match the server's memory bandwidth. More details are given in Section 4.

As a stateless PIR protocol, WhisPIR does not require any offline phase to update any client or server parameters, regardless of any database updates (excluding significant changes in the database size). All WhisPIR clients are considered *ephemeral*, meaning that they only interact with the server when they make a query, then disappear once the response has been sent. The only public parameters in WhisPIR are a database size and basic parameters to specify the polynomial ring for the Brakerski-Gentry-Vaikuntanathan (BGV) FHE scheme [BGV12], which can fit into just a few hundred bits.

WhisPIR provides a number of parameter settings to suit a large variety of applications. In particular, WhisPIR can be tuned to adjust the communication-computation tradeoff of the protocol's performance, allowing applications that can handle more communication to benefit from reduced server computation, while applications that have more computational resources to benefit from reduced communication. We discuss these tradeoffs in Section 3.5 and demonstrate the various optima in Section 4.

WhisPIR achieves its performance by relying on several novel optimizations that are likely of independent interest. Firstly, we optimize the index expansion algorithm [ACLS18, MCR21] for when only one rotation key is available. By carefully choosing the rotation supported by this key, we can reduce the number of rotations required to expand the index by over an order of magnitude. Furthermore, we show how dynamically selecting the rotation generator enables further optimizations by splitting the index representation into several ciphertexts, saving an additional 5× in the number of rotations. More details are given in Section 3.2.

An additional optimization introduced by WhisPIR is to adjust the database representation into a hypercube with relatively few dimensions (in practice, at most 4). We then take advantage of the *non-compact* variant of the BGV FHE scheme by allowing the ciphertext to *grow* with each multiplication. This optimization saves in both communication and computation, since we no longer need to transmit the BGV relinearization key and we no longer need to

perform BGV relinearization. Combining this optimization with standard BGV modulus reduction to compress the ciphertext modulus to the smallest decryptable value results in a response ciphertext that is still quite small despite not being theoretically compact. We believe this optimization likely has other applications when low-depth homomorphic evaluation is performed in a stateless setting. More details are given in Section 3.3.

2 BACKGROUND

Notation. For a finite set S, we denote $x \xleftarrow{\$} S$ as sampling an element $x \in S$ uniformly at random. Let $[N]$ denote the index space $\{1, \ldots, N\}$. Unless otherwise specified, the base of a logarithm is always 2.

2.1 PIR: Definitions & API

We begin by giving the API for PIR. The database elements are treated as elements of $\mathbb{Z}_t$, where t is some integer modulus. The parameters for this primitive are a modulus t, a database length N, and a security parameter λ. The algorithms are described below, and the usage of the API is given in Figure 7 in Appendix A.1.

- $\mathsf{pp}, \mathsf{sp} \leftarrow \mathsf{Setup}(1^\lambda, 1^N, t)$
  Takes in a security parameter λ and a database size N and t. Outputs public parameters pp and server parameters sp.
- $\mathsf{qry}, \mathsf{st} \leftarrow \mathsf{Query}(\mathsf{pp}, i \in [N])$
  Takes in an index $i \in [N]$ and produces a query qry with a query state st.
- $\mathsf{ans} \leftarrow \mathsf{Answer}(\mathsf{sp}, \mathbf{D}, \mathsf{qry})$
  Takes in a database $\mathbf{D}$ and a query qry and produces an answer ans.
- $d \leftarrow \mathsf{Recover}(\mathsf{pp}, \mathsf{st}, \mathsf{ans})$
  Takes in a query state st generated from an index $i \in [N]$ and an answer ans. Outputs a database element d. If inputs are honestly generated, the output d should be $\mathbf{D}[i]$.

We distinguish between the public parameters pp and the server parameters sp since in our scheme the server performs non-trivial (but still one-time) precomputation, while the public parameters that the client downloads are quite small (essentially just the database size and a λ-bit PRG seed). We define basic correctness and security requirements for the semi-honest PIR primitive in Appendix A.1.

2.2 BGV Homomorphic Encryption Scheme

At a high level, our PIR protocol will use the homomorphic encryption scheme of Brakerski, Gentry, and Vaikuntanathan [BGV12] to evaluate the lookup function on the database. The exact computation is described in Section 3, and here we describe the basic operations in the BGV scheme used to evaluate this computation.

The BGV scheme is parametrized by a ring $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^n+1)$ and a plaintext modulus p coprime to q. Let $\gamma_{\mathcal{R}}$ define the ring expansion factor (see Appendix A.2 for more information). The basic encryption scheme is defined by the following three algorithms.

- $\mathsf{sk} \leftarrow \mathsf{KeyGen}(1^\lambda)$
  Sample $\mathbf{s} \xleftarrow{\$} \mathcal{R}_q$ uniformly at random over $\mathcal{R}_q$. Return $\mathsf{sk} \leftarrow \mathbf{s}$.
- $\mathsf{ct} \leftarrow \mathsf{Encrypt}(\mathsf{sk}, \mathbf{m} \in \mathcal{R}_p)$
  Sample $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ uniformly at random over $\mathcal{R}_q$. Sample an error polynomial $\mathbf{e} \leftarrow \chi$. Return the ciphertext $\mathsf{ct} \leftarrow (\mathbf{m} + p \cdot \mathbf{e} - \mathbf{a} \cdot \mathbf{s}, \mathbf{a})$, where all operations are over $\mathcal{R}_q$.
- $\mathbf{m} \leftarrow \mathsf{Decrypt}(\mathsf{sk}, \mathsf{ct})$
  Parse $\mathsf{ct} = (\mathbf{c}_0, \mathbf{c}_1)$ and return $(\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} \mod q) \mod p$.

Homomorphic Operations. It is common to view ciphertexts in this scheme as polynomials over $\mathcal{R}_q$ that evaluate to the message on the secret key. This view allows for a natural construction of the encrypted multiplication operation. Consider two ciphertexts $\mathsf{ct} = (\mathbf{c}_0, \mathbf{c}_1)$ and $\mathsf{ct}' = (\mathbf{c}'_0, \mathbf{c}'_1)$ encrypting messages $\mathbf{m}$ and $\mathbf{m}'$, respectively.

$$\begin{aligned} \mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} &= \mathbf{m} + \mathbf{e} \cdot p \\ \mathbf{c}'_0 + \mathbf{c}'_1 \cdot \mathbf{s} &= \mathbf{m}' + \mathbf{e}' \cdot p \\ \mathbf{c}_0\mathbf{c}'_0 + (\mathbf{c}_0\mathbf{c}'_1 + \mathbf{c}'_0\mathbf{c}_1) \cdot \mathbf{s} + \mathbf{c}_1\mathbf{c}'_1 \cdot \mathbf{s}^2 & \\ = \mathbf{m} \cdot \mathbf{m}' + (\mathbf{m}'\mathbf{e} + \mathbf{e}'\mathbf{m}) \cdot p + \mathbf{e}' \cdot \mathbf{e} \cdot p^2 & \end{aligned} \tag{1}$$

This product relation gives a three-term ciphertext

$$\widetilde{\mathsf{ct}} = (\widetilde{\mathbf{c}}_0, \widetilde{\mathbf{c}}_1, \widetilde{\mathbf{c}}_2) = (\mathbf{c}_0 + \mathbf{c}'_0, \mathbf{c}_0\mathbf{c}'_1 + \mathbf{c}'_0\mathbf{c}_1, \mathbf{c}_1\mathbf{c}'_1)$$

that decrypts to $\mathbf{m} \cdot \mathbf{m}' \mod p$. While the full instantiation of BGV includes a *relinearization* operation to reduce the degree of the ciphertext back to a linear function of the secret key, we skip explicit discussion of this operation since it is not used in our PIR protocol. However, mechanically it is almost identical to the automorphism key switching described below.

Note that we have slightly generalized the decryption to view the ciphertext as an element of $\mathcal{R}_q[Y]$ (the polynomial ring where the coefficients are elements of $\mathcal{R}_q$) that is evaluated at the secret key to give the message. We now define the basic homomorphic operations over these general ciphertexts, as well as the more general decryption operation. We refer to [BGV12] for the correctness of these operations.

- $\mathsf{ct}' \leftarrow \mathsf{EvalAdd}(\mathsf{ct}_1 \in \mathcal{R}_q[Y], \mathsf{ct}_2 \in \mathcal{R}_q[Y])$
  This is the homomorphic addition operation. Takes in two polynomials in $\mathcal{R}_q[Y]$ and outputs their sum over $\mathcal{R}_q[Y]$, where the sum is taken coefficient-wise. The noise term in ct' is simply the sum of the noise terms of the input ciphertexts.
- $\mathsf{ct}' \leftarrow \mathsf{EvalMultPlain}(\mathsf{ct} \in \mathcal{R}_q[Y], \mathbf{m} \in \mathcal{R}_p)$
  This is homomorphic multiplication when one operand is in not encrypted. Takes in a ciphertext as an element in $\mathcal{R}_q[Y]$ and a plaintext $\mathbf{m} \in \mathcal{R}_p$ and outputs $\mathsf{ct}' \leftarrow \mathsf{ct} \cdot \mathbf{m}$, where $\mathbf{m}$ is lifted to $\mathcal{R}_q$ and treated as a scalar element of $\mathcal{R}_q[Y]$. If the noise term in the input ciphertext is $\mathbf{e}$, the noise term $\mathbf{e}'$ in the output ciphertext is at most $||\mathbf{e}'|| \leq \gamma_{\mathcal{R}} \cdot ||\mathbf{m}|| \cdot ||\mathbf{e}|| < p\sqrt{n}||\mathbf{e}||$.
- $\mathsf{ct}' \leftarrow \mathsf{EvalMult}(\mathsf{ct}_1 \in \mathcal{R}_q[Y], \mathsf{ct}_2 \in \mathcal{R}_q[Y])$
  This is homomorphic multiplication when both operands are encrypted. Takes in two elements of $\mathcal{R}_q[Y]$ and outputs the polynomial product over $\mathcal{R}_q[Y]$. If the noise terms of the input ciphertexts are $\mathbf{e}$ and $\mathbf{e}'$, then by Equation (1) the noise of the resulting ciphertext will be at most

$$p \cdot (||\mathbf{e}|| + ||\mathbf{e}'|| + \gamma_{\mathcal{R}} \cdot ||\mathbf{e}|| \cdot ||\mathbf{e}'||).$$

- $\mathbf{m} \leftarrow \mathsf{Decrypt}(\mathsf{sk} \in \mathcal{R}_q, \mathsf{ct} \in \mathcal{R}_q[Y])$
  Evaluate $\mathbf{m}' \leftarrow \mathsf{ct}(\mathsf{sk})$ over $\mathcal{R}_q$. Output $\mathbf{m} \leftarrow \mathbf{m}' \mod p$.

The correctness of the Decrypt algorithm holds as long as the noise term of the ciphertext, defined as $\mathsf{ct}(\mathsf{sk}) = \mathbf{m} + p \cdot \mathbf{e}$, satisfies $||\mathbf{e}||_\infty < q/p$.

Modulus Switching. In Appendix A.4, we describe the operation of BGV modulus switching, which is a common technique to compress a ciphertext once homomorphic computation is finished. This is an important optimization to reduce the download size in our PIR protocol.

2.2.1 BGV Automorphisms. The only other homomorphic operation that we use in our PIR protocol is the automorphism operation, which allows us to permute the coefficients of the message while keeping the same secret key. Note that even though we've defined homomorphic operations over general elements of $\mathcal{R}_q[Y]$, we only define the automorphism for ciphertexts that are linear functions of the secret key. These are the only ciphertexts that we permute in WhisPIR.

Let $\pi\colon [n] \to [n]$ be a permutation. Consider an encryption of a message $\mathbf{m}$ of the form $\mathsf{ct} = (\mathbf{c}_0, \mathbf{c}_1)$. We can permute the ciphertext $\mathsf{ct}^{(\pi)} = (\mathbf{c}_0^{(\pi)}, \mathbf{c}_1^{(\pi)})$, which decrypts to the permuted message $\mathbf{m}^{(\pi)}$ under the permuted secret key $\mathbf{s}^{(\pi)}$. In order to decrypt under the original key, we need to switch back to the original key using an "encryption" of the permuted key under the original key. This "encryption" is called a *switching key*. In WhisPIR, we use the following structure for the switching key:

$$\mathsf{swk}_\pi = \left\{(\mathbf{b}_i', \mathbf{a}_i') = \left(B^i \cdot \mathbf{s}^{(\pi)} + p \cdot \mathbf{e} - \mathbf{a}_i' \cdot \mathbf{s}, \mathbf{a}_i'\right)\right\}_{i=0}^{\log_B(q)} \tag{2}$$

where B is a free parameter chosen to tune noise and performance, discussed below. The goal of the key switching operation is to take $\mathsf{ct}^{(\pi)} = (\mathbf{c}_0^{(\pi)}, \mathbf{c}_1^{(\pi)}) \in \mathcal{R}_q^2$ and compute a tuple of the form $(\mathbf{c}_1^{(\pi)} \cdot \mathbf{s}^{(\pi)} + \mathbf{e} \cdot p - \mathbf{a} \cdot \mathbf{s}, \mathbf{a}) = (\mathbf{b}, \mathbf{a})$, where the error $\mathbf{e}$ is relatively small. Once we have this tuple, we can output a new encryption $(\mathbf{b} + \mathbf{c}_0^{(\pi)}, \mathbf{a})$ that decrypts to the permuted message $\mathbf{m}^{(\pi)}$ under the original secret key.

The computation of this tuple is one of the most intensive steps in our protocol, so we describe it in detail here. For a base B, define $w := \lceil \log_B(q) \rceil$. To compute an "encryption" of $\mathbf{c}_1^{(\pi)} \cdot \mathbf{s}^{(\pi)}$, the polynomial $\mathbf{c}_1^{(\pi)}$ is base-decomposed coefficient-wise into w polynomials $\mathbf{d}_0, \ldots, \mathbf{d}_{w-1} \in \mathcal{R}_B$ such that $\mathbf{c}_1^{(\pi)} = \sum_{i=0}^{w-1} B^i \cdot \mathbf{d}_i$, where the sum is performed over $\mathcal{R}_q$. Once the digits $\mathbf{d}_0, \ldots, \mathbf{d}_{w-1}$ of $\mathbf{c}_1^{(\pi)}$ are computed, the full key switching tuple can be computed by taking the inner product of the digits with the switching key, giving the equation

$$(\mathbf{b}, \mathbf{a}) = (\mathbf{c}_1^{(\pi)} \cdot \mathbf{s} + p \cdot \mathbf{e}' - \mathbf{a}\mathbf{s}) = \sum_{i=0}^{w-1} \mathbf{d}_i \cdot (\mathbf{b}_i', \mathbf{a}_i'). \tag{3}$$

To complete the operation, we simply add $\mathbf{b}$ to $\mathbf{c}_0^\pi$ to cancel the $\mathbf{c}_1^{(\pi)} \cdot \mathbf{s}^{(\pi)}$ term and leave only the term linear in the original secret key $\mathbf{s}$. Observe the the noise growth of this operation is the additive term $\mathbf{e}'$ from the switching tuple $(\mathbf{b}, \mathbf{a})$. Let B_χ be the bound on the output of χ, which is the bound on the noise terms in the switching key swk_π. Each tuple in swk_π is multiplied by digits of $\mathbf{c}_1^{(\pi)}$ of size at most B. Therefore, the size of the resulting noise term is $||\mathbf{e}|| \le \gamma_\mathcal{R} \cdot B_\chi \cdot B \cdot w$.

Precomputing the Decomposition. We now describe a very important optimization to the rotation algorithm that significantly improves the performance of computing rotations on input ciphertexts. In the textbook version of the BGV scheme, the base decomposition described above is the most computationally expensive step in key switching. This is because the decomposition must be performed over the coefficients while the remainder of the ring operations (in particular, polynomial multiplication) are performed in the elements *evaluation* domain. This means that expensive NTTs are required to map the element to it's original coefficients to perform the base decomposition.

In the recent work of Li et al. [LMRSW23], it was observed that if the $\mathbf{a}$ term of a ciphertext is known in advance, all of the NTTs can be *precomputed* in an offline phase. We define this algorithm with the following two functions, one that is computed during the protocol setup and the other that is computed when processing a query.

- $(\mathbf{a}_{out}, \{\mathbf{d}_i\}_{i=0}^{w-1}) \leftarrow \mathsf{PreSwitch}(\{\mathbf{a}_i'\}_{i=0}^{w-1}, \mathbf{c}_1, B, \pi)$
  Takes in as input $\{\mathbf{a}_i'\}_{i=0}^{w-1}$ where each $\mathbf{a}_i' \in \mathcal{R}_q$ as the switching key terms, $\mathbf{c}_1 \in \mathcal{R}_q$ as the ciphertext linear term, the decomposition base B, and the permutation π. Compute the permuted ring element $\mathbf{c}_1^{(\pi)}$. Compute $\mathbf{d}_0, \ldots, \mathbf{d}_{w-1} \in \mathcal{R}_B$ via the coefficient base-decomposition of $\mathbf{c}_1^{(\pi)}$. Compute $\mathbf{a}_{out} := \sum_{i=0}^{w-1} \mathbf{d}_i \cdot \mathbf{a}_i'$ and output $(\mathbf{a}_{out}, \{\mathbf{d}_i\}_{i=0}^{w-1})$.
- $\mathbf{b}_{out} \leftarrow \mathsf{KSNoDecomp}(\{\mathbf{b}_i'\}_{i=0}^{w-1}, \mathbf{c}_0, \{\mathbf{d}_i\}_{i=0}^{w-1}, \pi)$
  Takes in the second elements in the key switching tuple $\{\mathbf{b}_i'\}_{i=0}^{w-1}$, the second element of the ciphertext $\mathbf{b}$, the precomputed digits $\{\mathbf{d}_i\}_{i=0}^{w-1}$, and the permutation π. Compute the permutation $\mathbf{c}_0^{(\pi)}$ of $\mathbf{c}_0$ and output $\mathbf{b}_{out} \leftarrow \mathbf{c}_0^{(\pi)} + \sum_{i=0}^{w-1} \mathbf{b}_i' \cdot \mathbf{d}_i$.

Observe the that the online operation KSNoDecomp consists only of linear operations over $\mathcal{R}_q$. The usage of these algorithms is given in Figure 8 in Appendix A.3, which displays the iterative precomputation where the output of PreSwitch can be used to precompute the next permutation. Note that the usage requires sampling a fresh secret key each time a precomputed output is used, which is necessary for security. The correctness of these operations follows directly from the correctness of the original key switching routine as long as the inputs to PreSwitch matches the elements used to generate the $\mathbf{c}_0$ and $\mathbf{b}_i'$ elements of the ciphertext and switching key.

Iterative Precomputation. An important use of the API described above is that many rotations can be precomputed from a single switching key. In other words, suppose the server wishes to compute the permutations π, π^2, π^3. As shown in Figure 8 in Appendix A.3, the output of one iteration of PreSwitch can be used to precompute the next permutation. In Section 3.2, we discuss more efficient methods of precomputing the rotation when the only desired output is some high power of the input permutation.

3 THE WHISPIR PROTOCOL

In this section, we present WhisPIR, our stateless PIR protocol. We begin with the most basic version of the protocol, which minimizes

total communication. In Section 3.5, we describe the computation optimizations that can be included at only a marginal communication overhead.

3.1 Protocol Overview

We begin with a high-level description of the protocol.

Database Representation. As described in Section 2.1, we begin with the view of the database as an element in $\mathbb{Z}_t^N$. The entries of this database are indexed by $i \in [N]$. Our first step is to digitize this index space with respect to some digit ℓ, such that $\ell^k \geq N$. We can now write an index $i \in [N]$ as k digits $i_1, \ldots, i_k$, where each $i_j \in [\ell]$.

A BGV ciphertext can encrypt n elements of $\mathbb{Z}_p$. We pack database entries into elements in $\mathcal{R}_p$ in order to reduce the effective index space $[N]$. For databases with entries of size less than $n \cdot \lfloor \log_2(p) \rfloor$ bits, this can result in a significant reduction in the total number of indices. For the remainder of this section, we will assume that N is the number of indices after this packing has been performed. Note that if the database entries are larger than $n \cdot \lfloor \log_2(p) \rfloor$, we can simply split elements across multiple $\mathcal{R}_p$ elements that are queried in parallel. As we discuss below, artificially growing the database blocks to further reduce the index space is an effective computation optimization when a larger download can be tolerated.

Query Structure. The structure of a query is stable for all variants of WhisPIR. The query consists of two pieces. The first piece is a BGV rotation key, described in Section 2.2, where the base decomposition is performed over the *full* ciphertext modulus. Note that this differs from most efficient implementations of RLWE-based homomorphic encryption, which performs the base decomposition over some RNS-friendly basis. Decomposing over the RNS basis increases the number of digits in the decomposition, which correspondingly increases the size of the key. We discuss how a hoisting-style optimization [JVC18, LMRSW23] makes this non-standard choice of basis optimal. The second piece is a BGV ciphertext encrypting the index. The representation of an index $i \in [N]$ is in base ℓ, where each digit is represented as a one-hot vector. In total, the representation of this digit is a k-hot binary vector of length $k \cdot \ell$, where there is exactly one 1 in each block of ℓ elements. As long as $k \cdot \ell \leq n$, this digit fits in a single BGV ciphertext. We discuss below why the structure of the automorphism group over $\mathcal{R}_q$ only allows us to use half the elements in the index ciphertext, which in practice necessitates that $k \cdot \ell \leq n/2$. Despite this restriction, the typical choice of $n = 2^{12}$ allows us to represent essentially all practical database sizes with a digit space that fits within a single ciphertext. Some parameter choices require $n = 2^{13}$, but this is not due to the index representation. We discuss below how it is typically optimal to have an index representation with a length far less than $n/2$ for either of these choices of n.

As we discuss below, many encrypted indices can be sent along with a single rotation key if the application allows a client to perform a batch of queries at once. Note that this batch need not be performed interactively, although the upper limit on the batch size must be known in advance so that the server can perform the sufficient precomputation.

Homomorphic Computation. The homomorphic computation proceeds in two phases. The first phase is independent of the database and consists of an index expansion algorithm. This algorithm, first introduced in [ACLS18], has now become relatively standard in many PIR protocols [ACLS18, MCR21, MW22]. At a high level, this algorithm takes in a single BGV ciphertext encrypting a message $\mathbf{m} \in \mathcal{R}_p$ and outputs n ciphertexts where the i^{th} ciphertext encrypts the scalar $m_i \in \mathbb{Z}_p$, the i^{th} coefficient of the original $\mathbf{m}$. This scalar occupies the free term of the full $\mathcal{R}_p$ plaintext element, where the other coefficients are 0. The algorithm consists of a $\log(n)$-depth tree of rotations with n leaves, for a total of roughly $2n$ rotations when all rotation keys are available. We discuss below how we efficiently evaluate this tree with only a single rotation key.

In our protocol, we evaluate this tree until we obtain $k \cdot \ell$ ciphertexts, each encrypting a binary scalar value. The server then takes these ciphertexts and breaks them into k sets of ℓ ciphertexts each, one set for each digit. The server now has encryptions of k one-hot vectors, where each element of the vectors is in its own ciphertext. The server then multiplies each of these one-hot vectors with the database. Recall that we view the database as consisting of $N = \ell^k$ elements of $\mathcal{R}_p$, and each multiplication reduces the dimension of the remaining database by a factor of ℓ. After this depth-k circuit is evaluated, the resulting ciphertext is returned to the client.

3.2 Index Expansion with One Key

We now describe the our optimizations to the coefficient expansion algorithm. The original algorithm [ACLS18, MCR21] is given in Algorithm 1. We define the algorithm over the plaintext ring $\mathcal{R}_p$. This algorithm uses the *substitution* operation over $\mathcal{R}_p$. This operation is indexed by some $r \in \mathbb{Z}_{2n}^*$ and is defined by replacing $x \leftarrow x^r$. These substitutions form a group isomorphic to $\mathbb{Z}_{2n}^*$, and any substitution $r \in \mathbb{Z}_{2n}^*$ of a BGV plaintext can be computed with a single automorphism key [GHS12][1]. The goal of the index expansion phase is to homomorphically evaluate Algorithm 1 on the query, where the number of expanded coefficients is equal to $k \cdot \ell$. Note that the

Algorithm 1 Coefficient Expansion Algorithm. The expansion is over the first c coefficients of $\mathbf{m}$, and correctness requires that $\mathbf{m}$ has zeros for the remaining $n - c$ coefficients.

Input: A ring element $\mathbf{m} \in \mathcal{R}_p$ and scalar $c \leq n$.

$elems \leftarrow [\mathbf{m}]$
Let d be the smallest power of two such that $d \geq c$.
for $i = 0$ to $\log(d) - 1$ **do**
  for $j = 0$ to $2^i - 1$ **do**
    $\mathbf{a}_0 \leftarrow elems[j]$
    $\mathbf{a}_1 \leftarrow \mathbf{a}_0(x^{n/2^i+1})$ ▷ Substitution with $x^{n/2^i+1}$
    $\mathbf{a}'_j \leftarrow \mathbf{a}_0 + \mathbf{a}_1$
    $\mathbf{a}_{j+2^i} \leftarrow (\mathbf{a}_1 - \mathbf{a}_0) \cdot x^{n-2^i}$
  $elems \leftarrow [\mathbf{a}'_0, \ldots, \mathbf{a}'_{2^{i+1}-1}]$
for $i = 0$ to d **do** $\mathbf{b}_i \leftarrow elems[i] \cdot (d^{-1} \mod p)$

Output: $\{\mathbf{b}_i\}_{i=0}^{d-1}$ where $\mathbf{b}_i \in \mathcal{R}_p$ and $\mathbf{b}_i = m_i$.

[1] Computing these substitutions is equivalent to permuting the evaluations of an $\mathcal{R}_p$ element, which is typically the operation of interest. In this work, we never consider the evaluation domain of $\mathcal{R}_p$ since we do not need batched multiplication.

final loop requires the existence of $d^{-1} \mod p$, which requires us to choose an odd p since d is a power of two.

Picking the Right Generator. The first observation that should be made is that the group $\mathbb{Z}_{2n}^*$ is alternating, so there is no single generator for the entire group. This means that we cannot compute every possible substitution with a single switching key. While Algorithm 1 only requires a small subset of substitutions (namely, substitutions indexed by $n/2^i + 1 \in \mathbb{Z}_{2n}^*$), attempting to expand all n coefficients requires substitutions by both 3 and 5, which are not in the same subgroup. Therefore, we are limited to only expanding half of the coefficients. Luckily, the remainder of the substitutions $n/2^i + 1$ for $i < \log(n)$ are in the subgroup generated by 5. Sending the automorphism key corresponding to substitution $x \leftarrow x^5$ allows $n/2$ coefficients to be expanded with a single key.

However, there are many other choices of generators for this subgroup, and we empirically find that other choices result in far fewer rotations than the naive choice of 5. For a power of two $d \geq k \cdot \ell$, our goal is to select a generator g such that $n/2^i + 1 \in \langle g \rangle \subset \mathbb{Z}_{2n}^*$ for all $0 \leq i < \log(d)$ that minimizes the the following function. Let u_i be the smallest exponent such that $g^{u_i} \equiv n/2^i + 1 \mod 2n$. The i^{th} iteration of the outer loop in Algorithm 1 requires 2^i iterations of the inner loop, each of which requires u_i rotations by g. Therefore, our choice of g should minimize $\sum_{i=0}^{\log(d)-1} u_i \cdot 2^i$, which is the total number of rotations that must be performed during the index expansion. For each choice of k and ℓ, we empirically compute the generator that minimizes the number of rotations as part of the public parameters of the PIR scheme. A survey of optimal generators along with the total number of rotations is given in Table 1. This

Table 1: *Optimal generators $g \in \mathbb{Z}_{2n}^*$ for Algorithm 1 for various choices of d, the number of expanded coefficients.*

$n = 2^{12}$						
d	2048	1024	512	256	128	64
g	2269	5513	5777	7713	65	129
Total Rotations	386048	113664	20736	7168	2496	192
$n = 2^{13}$						
d	2048	1024	512	256	128	64
g	9193	13969	14881	14401	129	257
Total Rotations	506880	91136	22784	5120	448	192

optimization reduces the number of rotations by roughly $2 - 50\times$ below the naive choice of generator. We leave the development of a closed-form expression for this optimal generator, as well as closed-form solutions when multiple generators are available, for future work.

Don't Rotate the Ciphertext, Rotate the Key. In Section 2.2, we describe the optimization to precompute the base decomposition in key switching as well as how this precomputation can be composed to quickly compute iterative rotations. However, observe that in this iterative precomputation (see Figure 8) each online key switching computes the full intermediate result. If only the switching key for π is available, then instead of computing π^3 directly, the input ciphertext is first rotated once to compute π, then a second time to compute π^2, then finally a third time to compute π^3. This is inefficient when only the final permutation π^3 is of interest, as is the case when computing a high power of some generator permutation in this index expansion phase.

To address this, we introduce a variant of the precomputed key switching that is optimized for computing many iterative rotations. In the original version, the permutation π^u was iteratively computed by permuting a ciphertext by π, performing a key switching operation, then repeating this process u times. Our idea is to only rotate the input ciphertext *once* to the target permutation π^u, then perform u key switching operations where each operation rotates the secret key by π^{-1}. The input switching key already performs this rotation by π^{-1}, but correctness only holds when multiplying the switching key by the digits of the term linear in $\mathbf{s}^{(\pi)}$. To rotate $\mathbf{s}^{(\pi^j)}$ back to $\mathbf{s}^{(\pi^{j-1})}$ for $j > 1$, we need to compute rotations of the switching key itself. Observe that applying the permutation π^j to Equation (2) yields the switching key

$$\pi^j(\mathsf{swk}_\pi) = \left\{\left(\pi^j(\mathbf{b}_i'), \pi^j(\mathbf{a}_i')\right)\right\}_{i=0}^{\log_B(q)} \tag{4}$$
$$\pi^j(\mathbf{b}_i') = B^i \cdot \mathbf{s}^{(\pi^{j+1})} + p \cdot \mathbf{e} - \pi^j(\mathbf{a}_i') \cdot \mathbf{s}^{(\pi^j)}$$

that maps an encryption under the secret $\mathbf{s}^{(\pi^{j+1})}$ to an encryption under the secret $\mathbf{s}^{(\pi^j)}$. To complete the key switching for the target ciphertext, we apply the key switching procedure u times using $\pi^{u-1}(\mathsf{swk}_\pi), \pi^{u-2}(\mathsf{swk}_\pi), \ldots, \mathsf{swk}_\pi$ iteratively until the ciphertext decrypts to $\mathbf{m}^{(\pi^u)}$ (the correct permutation of the input plaintext) under the original key.

We define variants of the switching key precomputation and online algorithms in Algorithm 2 and Algorithm 3, respectively.

Algorithm 2 Precomputation of Iterative Rotations. When precomputing many sets of iterative rotations, the permutations of the $\mathbf{a}_i'$ terms are reused.

Input: $\{\mathbf{a}_i'\}_{i=0}^{w-1}$ as the initial switching key terms, $\mathbf{c}_1$ as the input ciphertext linear term, the decomposition base B, the generating permutation π, and the exponent u.

1: Compute the permuted ring element $\mathbf{a} \leftarrow \mathbf{c}_1^{(\pi^u)}$.
2: **for** $j = u - 1$ down to 0 **do**
3: Decompose $\mathbf{a}$ into digits $\vec{\mathbf{d}}^{(j)} \leftarrow \{\mathbf{d}_0^{(j)}, \ldots, \mathbf{d}_{w-1}^{(j)}\}$
4: ▷ Each $\mathbf{d}_i^{(j)} \in \mathcal{R}_B$ and $\mathbf{a} = \sum_{i=0}^{w-1} B^i \mathbf{d}_i^{(j)}$.
5: Set $\mathbf{a} \leftarrow \sum_{i=0}^{w-1} \mathbf{d}_i^{(j)} \cdot \pi^j(\mathbf{a}_i)$.

Output: $\mathbf{a}, \{\vec{\mathbf{d}}_j\}_{j=0}^{u-1}$.

When computing many iterative rotations, this approach reduces the number of permutations (but not the number of inner products with a switching key), since we only rotate the input ciphertext once rather than u times. In addition, observe that the online key switching algorithm (Algorithm 3) is almost entirely a single inner product over $\mathcal{R}_q$. Combined with standard "lazy" modulus-reduction, this allows significant acceleration from vectorized instructions which would not be available if we had to pause and permute the elements

Algorithm 3 Online Computation of Iterative Rotations. We assume the relevant permutations of the $\{\mathbf{b}'_i\}_{i=0}^{w-1}$ terms have already been computed. Define $\mathbf{b}'^{(j)}_i := \pi^j(\mathbf{b}'_i)$.

Input: Takes as input the relevant permutations of the switching key terms $\{\mathbf{b}'^{(j)}_i\}_{0\le i<w, 0\le j<u}$, the ciphertext message term $\mathbf{c}_0$, the u sets of precomputed digits, $\{\vec{\mathbf{d}}_j\}_{j=0}^{u-1}$, and the target permutation π^u.

1: Compute $\mathbf{c}_0^{(\pi^u)} \leftarrow \pi^u(\mathbf{c}_0)$
2: **for** $j = u-1$ to 0 **do**
3: **for** $i = 0$ to $w-1$ **do** $\mathbf{c}_0^{(\pi^u)} \leftarrow \mathbf{c}_0^{(\pi^u)} + \mathbf{d}_i^{(j)} \cdot \mathbf{b}'^{(j)}_i$

Output: $\mathbf{c}_0^{(\pi^u)}$.

after each switching key inner product. Overall, this optimization saves us as much as 4× in computation time during this phase.

3.3 Non-compact Homomorphic Multiplication

We now describe the second phase of the homomorphic computation, which is where the expanded index is multiplied by the database. The output of the expansion is $k \cdot \ell$ ciphertexts, which the server splits into k sets of ℓ ciphertexts each. Each set of ℓ ciphertexts encrypts a one-hot vector, which represents one digit of the index in base ℓ. After the expansion, we perform a depth k homomorphic multiplication over the database, where each multiplication reduces one dimension of the $[\ell] \times [\ell] \times \ldots \times [\ell]$ index space. The first level of the homomorphic multiplication is with the plaintext database, while the remaining $k-1$ levels are ciphertext-ciphertext multiplications. At depth i, the ciphertexts containing the remaining database entries have degree i in the secret key. At each level, these database ciphertexts are multiplied by the corresponding encrypted one-hot vector, representing the current digit of the index.

An important optimization in our protocol is that we do not relinearize the result of multiplications as in most instantiations of BGV. This can be viewed as the non-compact variant of BGV, where the size of the ciphertext grows with the depth of the function. However, since (empirically) we do not require a depth k beyond three or four, this is not an issue in the final output ciphertext. More specifically, the final ciphertext has the form

$$\left(\mathbf{m} + p \cdot \mathbf{e} - \sum_{i=1}^{k} \mathbf{a}_i \cdot \mathbf{s}^i,\ \mathbf{a}_1,\ \ldots,\ \mathbf{a}_k\right).$$

By not relinearizing, we save on communicating the relinearization key, which is as large as the rotation switching key, at the cost of growing the number of ring elements in the resulting ciphertext by roughly a factor of k. We can achieve significant savings by further reducing the ciphertext modulus to the smallest value that remains decryptable, as described in Remark A.2. Overall, the size of this "non-compact" download is often dominated by the size of the upload, as we show in Section 4.

Precomputing the Top Coefficient. We briefly note that we can extend the precomputation described in Section 3.2 to precompute the top coefficient $\mathbf{a}_k$ of the output ciphertext as long as the database remains fixed. If the database changes, this precomputed $\mathbf{a}_k$ term must also be updated, but this takes less computation than even the database scan during a single query. Only in settings where the server is receiving many more updates than queries would maintaining this $\mathbf{a}_k$ term result in significant computational overhead, and for most practical applications this will likely not occur. Therefore, we consider it practical for the server to maintain the precomputed $\mathbf{a}_k$ term, which can save nearly a factor of 2× during the database scan computation. In the benchmarks presented in Section 4, the server will still transmit this $\mathbf{a}_k$ element to the client as part of the response in order to maintain client statelessness.

3.4 Full Protocol Description

We now define the PIR API from Section 2.1 for the WhisPIR protocol in terms of the algorithms described in this section. The usage of this API is identical to Figure 7 in Appendix A.1.

Algorithm 4 WhisPIR Setup. We use the standard technique of transmitting a PRG seed σ rather than sending the truly random terms $\mathbf{c}_1$ and $\{\mathbf{a}'_i\}_{i=0}^{w-1}$.

Input: Security parameter λ and a database size N and t.

1: Select a polynomial degree n, a plaintext modulus p, and index parameters k and ℓ such that $\ell \cdot k \le n/2$ and $\ell^k \cdot n \cdot \log(p) \ge N \cdot \log(t)$.
2: Select a ciphertext modulus q and a decomposition base B that satisfies correctness & security.
3: Sample a PRG seed $\sigma \leftarrow \{0,1\}^\lambda$.
4: Select a generator permutation π that minimizes the number of key switching operations in Algorithm 1.
5: Set the public parameters
$$\mathsf{pp} \leftarrow (\sigma, \pi, k, \ell, n, q, B, p, N, t).$$
6: Sample a query index polynomial $\mathbf{c}_1$ and $w := \lceil \log_B(q) \rceil$ polynomials $\mathbf{a}'_i$ for $i \in [w]$ as the switching key polynomials. All of these polynomials are outputs of the PRG at σ.
7: Use Algorithm 2 to precompute all rotations in Algorithm 1. Set sp to be pp along with all outputs of all iterations of Algorithm 2 for the precomputed rotations.

Output: Public parameters pp and server parameters sp.

- $\mathsf{pp}, \mathsf{sp} \leftarrow \mathsf{Setup}(1^\lambda, 1^N, t)$
  This is defined in Algorithm 4.
- $\mathsf{qry}, \mathsf{st} \leftarrow \mathsf{Query}(\mathsf{pp}, i \in [N])$
  Parse the public parameters
  $$(\sigma, \pi, k, \ell, n, q, B, p, N, t) \leftarrow \mathsf{pp}.$$
  Using N, t, n, p, k, and ℓ, determine element of $\mathcal{R}_p$ that contains the desired database entry, then determine the digits of this element in base ℓ. Compute $\mathbf{m} \in \mathcal{R}_p$ such that the first $k \cdot \ell$ coefficients are the concatenation of the k digits of this index, where each digit is represented as a length-ℓ one-hot vector. The remaining $n - k \cdot \ell$ coefficients are zeros. Next, compute $\mathbf{c}_1$ and $\{\mathbf{a}'_i\}_{i=0}^{w-1}$ from the PRG seed. Sample a secret $\mathbf{s} \leftarrow \chi$ and compute
  $$\mathbf{c}_0 := \mathbf{m} + p \cdot \mathbf{e} - \mathbf{c}_1 \cdot \mathbf{s}$$
  $$\mathbf{b}'_i := B^i \cdot \mathbf{s}^{(\pi)} + p \cdot \mathbf{e} - \mathbf{a}'_i \cdot \mathbf{s} \quad \text{for} \quad 0 \le i < w$$

where all error terms $\mathbf{e}$ are freshly sampled from χ. Output $\mathsf{qry} \leftarrow (\mathbf{c}_0, \{\mathbf{b}'_i\}_{i=0}^{w-1})$ and $\mathsf{st} \leftarrow (\mathbf{s}, i)$.

- $\mathsf{ans} \leftarrow \mathsf{Answer}(\mathsf{sp}, \mathbf{D}, \mathsf{qry})$
  This algorithm follows the online protocol described in this section. Parse $(\mathbf{c}_0, \{\mathbf{b}'_i\}_{i=0}^{w-1}) \leftarrow \mathsf{qry}$. Homomorphically evaluates Algorithm 1 on the ciphertext element $\mathbf{c}_0$ and the switching key elements $\{\mathbf{b}'_i\}_{i=0}^{w-1}$, using Algorithm 3 to compute the online iterative rotations. Assume the database $\mathbf{D}$ has already been packed into an element of $\left(\mathcal{R}_p^{\ell}\right)^k$. The output of the index expansion is $k \cdot \ell$ encrypted scalars, which are interpreted as encryptions of k length-ℓ one-hot vectors. The server runs the EvalMultPlain algorithm on the first level and the EvalMult algorithm on the subsequent levels, using EvalAdd to sum the results of each level. Each multiplication reduces the dimension of the database by one, and after multiplicative depth k there is only one ciphertext remaining. Return this ciphertext as ans.
- $d \leftarrow \mathsf{Recover}(\mathsf{pp}, \mathsf{st}, \mathsf{ans})$
  Parse $(\mathbf{s}, i) \leftarrow \mathsf{st}$ and compute $\mathbf{m}' \leftarrow \mathsf{Decrypt}(\mathbf{s}, \mathsf{ans})$, where this is the generalized BGV decryption described in Section 2.2. Use the index i to select the correct subset of bits and output $\mathbf{D}[i]$.

3.5 Communication-Computation Trade-offs

The protocol described above is only the basic variant of WhisPIR. Instantiating WhisPIR as described will result in a PIR protocol with small total communication but relatively slow server run times. However, the performance of WhisPIR can be tuned for a target application by choosing a different point in the communication-computation trade-off space. As we show in Section 4, these adjustments can improve the computation times by over an order of magnitude while still maintaining relatively low communication.

Reducing the Index Space by Splitting the Database. The protocol described in Section 3 downloads only a single BGV ciphertext that has been compressed to the smallest possible modulus. However, we can consider a simple variant of the scheme where the database is split into c equal chunks, then the query is evaluated on each chunk in parallel and the responses to all chunks are returned. This increases the download by a factor of c, but it also reduces the index space by a factor of c, since now the client only needs to specify an index within a chunk rather than the entire database. Consider $k = 2$, where a chunk size as small as $c = 4$ will result in the index space $N/c = \ell^k/c = (\ell/2)^2$ that requires an index length of only $k \cdot \ell/2$ rather than $k \cdot \ell$. Observe in Table 1 how reducing the length of an index by a factor of 2 can save anywhere from $3 - 10\times$ the number of rotations in the index expansion phase. Since the downloaded ciphertexts are only a few dozen kilobytes, returning a several more is a small price to pay for what can be a order of magnitude reduction in the index expansion time. This optimization has the added benefit of increasing the maximum record size that can be returned to the client. Furthermore, reducing the number of rotations marginally decreases the upload size, since the ciphertext is required to handle less noise. While total communication may increase, in many commercial applications, download bandwidth is significantly greater than upload bandwidth, resulting in this adjusted communication saving on overall network latency.

Reducing Rotations by Splitting the Index. As Table 1 demonstrates, reducing the length of the index representation by a factor of 2 reduces the rotations required to expand the index by $> 2\times$. This means that if we split the length $k \cdot \ell$ vector into two vectors each of length $k \cdot \ell/2$, encrypt these vectors separately, then expand out the index using two invocations of Algorithm 1, we will save on the overall number of rotations at the cost of increasing the upload. However, as we show in Section 4, the encrypted index is typically around $10 - 20\%$ the size of the switching key, so splitting the index into two ciphertexts represents a relatively small communication increase. We discuss in Section 4 how splitting the index can reduce the number of rotations in the index expansion phase by an order of magnitude with only a marginal communication overhead.

Increasing the Ciphertext Rate. While the above optimizations are effective, they do essentially nothing to improve the runtime of the second phase of query evaluation: the database multiplications. This second phase quickly becomes the bottleneck as the index expansion becomes more efficient. To improve the runtime of this phase, we decrease the number of $\mathcal{R}_p$ elements required to represent the database by increasing the plaintext modulus p. This reduces the number of multiplications that must be performed during the database scan since the same database can be represented with fewer $\mathcal{R}_p$ elements. However, growing the plaintext modulus also increases the noise growth from the homomorphic multiplications (see Section 2.2). Leaving all other parameters constant would quickly require the ciphertext modulus to grow, eventually requiring the ring dimension n to jump to the next power to 2, which should be avoided whenever possible as this results in a significant performance hit.

In order to maintain roughly the same ciphertext modulus, we instead shrink the decomposition base B of the switching key. This has the effect of growing the number of digits in the switching key, although the lazy modular reduction in the iterative preprocessed key switching (see Section 3.2 and Algorithm 3) makes the computational overhead of processing additional digits relatively small. Instead, the most significant effect of this adjustment is growing the size of the query upload, since the switching key must include a $\mathcal{R}_q$ element for each digit in the decomposition.

4 IMPLEMENTATION & EVALUATION

In this section, we present our implementation of WhisPIR, discuss various design points, and compare to prior work.

Experimental Setup. We implement WhisPIR in C++, compile the code with `clang++` version 10, and benchmark on a machine running Ubuntu 20 with an Intel i7 core running at 2.5 GHz and 32 GB of RAM. All computation benchmarks were run on a single thread. While this protocol is "embarrassingly" parallel, available parallelism is highly application dependent, so we leave the examination of the parallel performance for future work. Our implementation makes use of the standard RNS representation of the RLWE modulus [GHS12, KPZ21] to efficiently work over ciphertext moduli larger than 64 bits. All of our parameters achieve 128-bit security level [ACC+18].

Main Focus: Communication and Server Time. We will focus our benchmarks on two measures: the per-query communication

and the server's computation time. The client's computation consists only of key generation, encryption and decryption, which take only a few dozen milliseconds regardless of the database size. We do not expect the client's computation to be a bottleneck in any practical application. Furthermore, while we allow for an application to frequently update the database content, we assume that the database size remains relatively stable throughout the application. In particular, we assume that the server will not have to run the preprocessing of the index expansion algorithm more than once in the application's lifetime, since this output can be reused for all queries. We leave examination of specialized applications that require frequent changes to the database size for future work.

Comparison with Prior Art. Our main points of comparison are

(1) Spiral [MW22], which has the best per-query communication *not* counting the client evaluation keys of state-of-the-art PIR protocols,
(2) SimplePIR [HHCG+23], which has the best server runtime of state-of-the-art PIR protocols,
(3) HintlessPIR [LMRSW23], which represents the current state-of-the-art in stateless PIR protocols.

We benchmark Spiral and SimplePIR by running the implementations[2][3] on the same machine that we benchmark our code. All Spiral and SimplePIR benchmarks were taken with 32-byte database entries. The implementation of HintlessPIR is not available at the time of this writing, so instead we simply take the benchmarks reported in their paper [LMRSW23, Tables 1 & 2][4].

4.1 Isolating Optimizations

WhisPIR provides a number of parameters that can be tuned to adjust performance based on the acceptable communication and computation of an application. To illustrate the impact of each parameter, we analyze the communication and computation of WhisPIR for a 1 GB database. The BGV parameters for a database of this size are $n = 2^{12}$ and a ciphertext modulus requiring two machine words to represent (roughly 110 bits). We begin with exploring the effect of splitting the database into chunks for fixed ciphertext rates, given in Figure 1. We plot all data points in Figure 2, where we also include benchmarks for related works for reference.

In Figure 1, the points where the index representation drops below a given power of two is clearly visible. Observe the drop in the index expansion time as the index representation length shrinks from 2048 to 1024 between the 4 and 8 chunk bars on the left plot. Similarly, in the right plot, the representation length drops from 1024 to 512 between the 1 and 4 chunk bars and from 512 to 256 between the 8 and 16 chunk bars. See Table 1 to compare the reductions in the number of rotations with the reduction in runtime.

It is worth noting that the database scan time is essentially identical for a fixed $\log(p)$ value (and degree n), since this is the only parameter that determines the number of $\mathcal{R}_p$ elements needed to represent the database. For $\log(p) = 1$, it requires about 2.1 million $\mathcal{R}_p$ elements to represent the database, while for $\log(p) = 8$ the rate increases proportionally to only require roughly 262000 $\mathcal{R}_p$ elements. Since the polynomial modulus degree n is not changing and the coefficient modulus is still roughly two machine words, the database scan time scales directly with the plaintext modulus. Observe in Figure 1 that for $\log(p) = 1$, the database scan time takes just under 6 seconds, while for $\log(p) = 8$ the database scan time takes a bit less than 3/4 of a second. This time holds regardless of the parameters of the index expansion phase. In general, when tuning the performance of WhisPIR, we first benchmark the runtime of the database scanning phase, since this is essentially fixed by n, $\log(p)$, and the number of machine words required to represent the ciphertext modulus. Once this runtime is acceptable, the index expansion parameters are then adjusted to meet the desired performance.

Figure 3 illustrates the effect of splitting the index representation (reducing the overall number of rotations). This plot further extends the computational performance of the 16-chunk database by splitting the index into more than one vector. With only one index vector, the index representation length is 256, requiring roughly seven thousand rotations to expand the index (see Table 1). With two index vectors, each vector has only 128 elements, requiring a total of roughly five thousand rotations (2496 for each vector) to expand the index. Observe in Figure 3 that with this parameter setting, we can achieve both sub-second latency in single-threaded server runtimes while also staying under 1 MB of communication. The most dramatic savings comes from splitting each vector one more time. With four vectors of 64 elements each, the total number or rotations is only 768, representing an order of magnitude reduction in the original number of rotations. We demonstrate below how combining these three optimizations results in a wide range of performance profiles for various database sizes.

4.2 Full Benchmarks

In this section, we present benchmarks for WhisPIR for a variety of database sizes to demonstrate how the protocol performance scales.

Observe in Figure 2 the general shape of the performance options for WhisPIR, which is roughly a shape of $1/x$. For all database sizes, the minimum communication point is relatively slow compared to the parameter settings with slightly higher ($< 2\times$) communication overhead. More specifically, we can save over an order of magnitude in the server's computation time while staying within $2\times$ of the minimum communication. On the other side of the trade-off, the server's computation quickly "bottoms-out" at the database scan time, and decreasing this time further requires progressively more communication. At some point, it is necessary to nearly double the communication for a roughly 10% decrease in the server's computation time.

We are interested in understanding the "sweet-spot" of this trade-off, where changes in communication and computation result in proportional changes in the other dimension. It is likely that most applications will be interested in a performance point around this area, since only extremely constrained applications would consider pushing this trade-off further. We plot example points in this region in Figure 4 for various database sizes.

[2] https://github.com/menonsamir/spiral
[3] https://github.com/ahenzinger/simplepir
[4] The computation benchmarks in [LMRSW23, Table 2] were taken on a machine running at 3.0 GHz, while our benchmarking machine runs at 2.5 GHz. Nevertheless, we use the exact run times from [LMRSW23] to avoid any inaccuracies.

Leo de Castro [†◇], Kevin Lewi[†], and G. Edward Suh[†§]
[†]Meta, [◇]Massachusetts Institute of Technology, [§]Cornell University

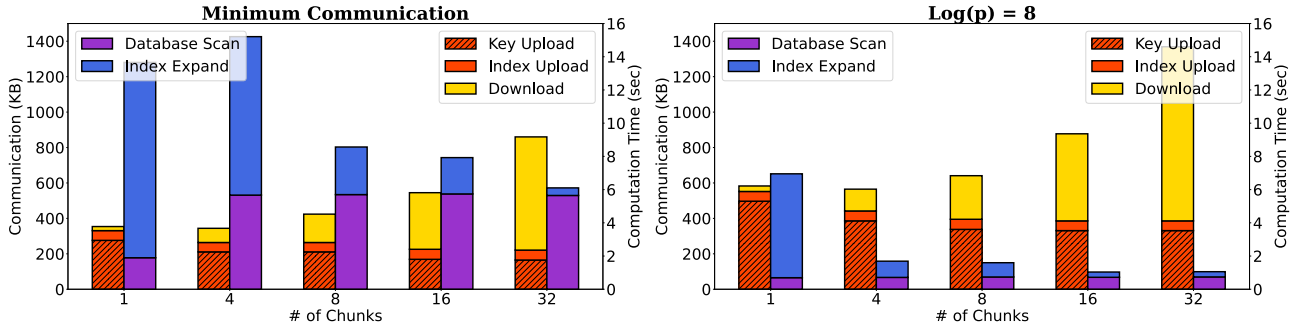

Figure 1: WhisPIR performance with varying parameters for a 1 GB database. All benchmarks use $n = 2^{12}$ and a ciphertext modulus q that fits in two 64-bit machine words. The left plot has no minimum plaintext modulus, and all bars use $\log(p) = 1$ except the first bar that uses $\log(p) = 3$. All benchmarks in the right plot use $\log(p) = 8$.

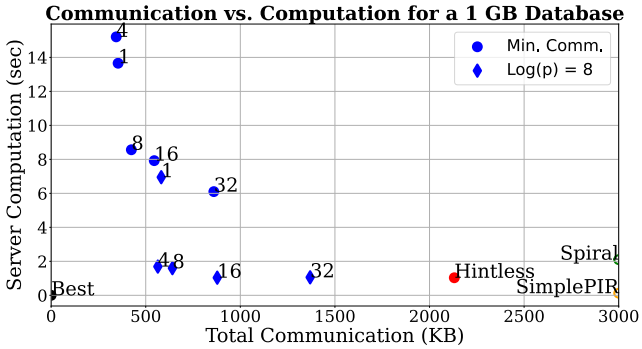

Figure 2: Plot of communication vs. computation trade-off for the parameter options given in Figure 1. The numbers next to the blue points indicate the number of chunks that the database is equally split into. All benchmarks are for a single query running on a single thread. The communication axis is truncated to view our results more clearly. Spiral has communication roughly 15800 KB and a computation time of 2.1 seconds, while SimplePIR has communication of roughly 126000 KB and a computation time of roughly 0.125 seconds.

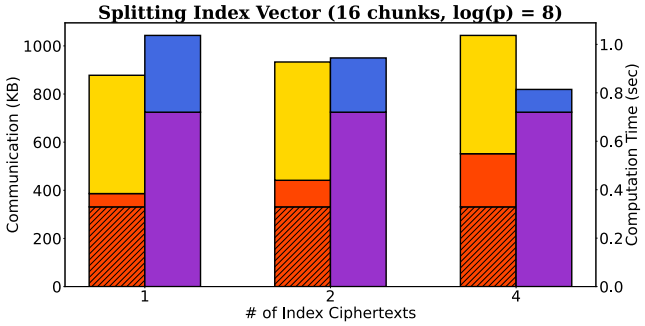

Figure 3: This plot illustrates the effect of splitting the index representation into more digits. The legend is identical to Figure 1. The key upload, the download, and the database scan times are identical throughout the plot. Only the index upload size and the index expansion time change.

Observe in Figure 4 that the shape of the performance tradeoff is essentially the same regardless of the database size. The jump in the communication that occurs within the 8 GiB plot is due to the increase in the polynomial modulus degree from $n = 2^{12}$ to $n = 2^{13}$. This increase continues for the 16 GiB and 32 GiB sizes. Aside from this jump, the communication growth is quite slow. For example, observe the leftmost points in the 16 GiB and 32 GiB plots. The increase in communication is less than 30 KB while the computation increases from about 21 seconds to about 29 seconds. For the same 32 GiB database, WhisPIR can achieve a 20 second computation time while only communicating 1.6× more than the marked 16 GiB point. This wide range of performance options makes WhisPIR well-suited for a large variety of application constraints.

Comparison: Stateless PIR. Figure 4 includes results from HintlessPIR, the current state-of-the-art stateless PIR scheme. The benchmarks from the largest four databases in [LMRSW23] are plotted in Figure 4. The 0.25 GiB database has 256 byte entries, the 0.5 GiB database has 8 byte entries, and the 1 GiB database has 1 byte entries. For all of these databases, WhisPIR outperforms HintlessPIR in both communication and computation. For the 8 GiB database benchmarks, WhisPIR outperforms HintlessPIR in communication but not computation. However, this database has 32 KB entries, which is significantly larger than all other benchmarks in this work. We leave for future work to determine the smallest database entry size for which WhisPIR can no longer strictly outperform HintlessPIR.

Comparison: Stateful PIR. We now compare against prior works that are designed to be stateful. When moving to the stateless setting, these protocols suffer from very high communication, since the protocol state must be transferred along with the query. We do not attempt to optimize the protocol state. The communication for a single query for Spiral, SimplePIR, and WhisPIR is displayed in Figure 5.

Figure 5 displays the communication in terms of the portion that is reusable for any number of queries and the portion that must be resent for each query. The reusable portion corresponds to the protocol state that would be communicated in the offline phase. Observe that nearly all of the communication in both Spiral and SimplePIR is this protocol state. The SimplePIR state is a database digest that must be updated as the database changes. Furthermore,

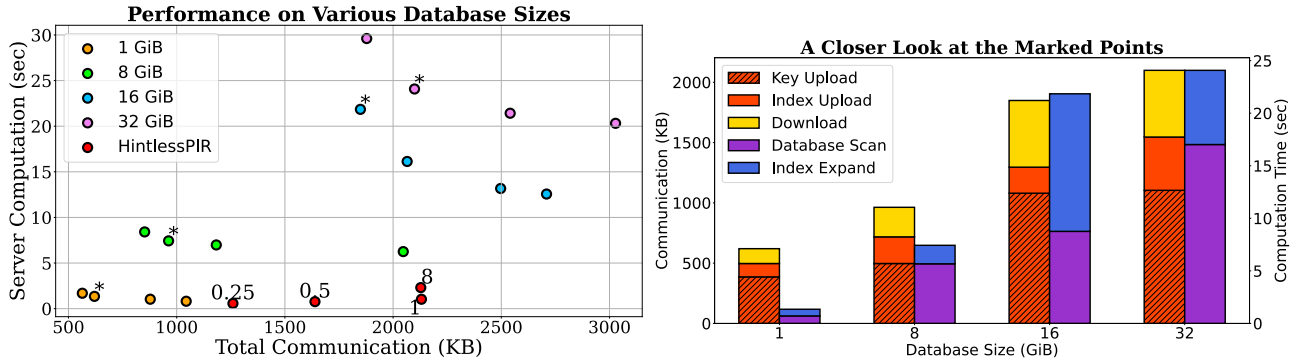

Figure 4: WhisPIR performance for databases of various sizes. These points are examples to illustrate the performance trade-offs; WhisPIR supports more extreme parameter settings in both the communication and computation direction. The plot on the right is a closer look at the marked points on the left, indicated with an *. The numbers next to the HintlessPIR points indicate the database size in GiB. Note that the 8 GiB point is for a database with 32 KB entries, while all other points are for a database with significantly smaller entries ($\leq$ 256 bytes).

the per-query communication is also outperformed by WhisPIR for all database sizes, so the communication of SimplePIR will never outperform WhisPIR for any number of queries. While the computation of SimplePIR is quite fast, roughly 100 milliseconds per gigabyte of database, the high communication precludes this protocol's use in a stateless setting, even when the number of queries is high.

We focus the remainder of the comparison on Spiral, since the per-query communication of Spiral outperforms WhisPIR for all database sizes. This means that for some number of queries, the communication of Spiral will eventually outperform WhisPIR. Note that all parameter values benchmarked for WhisPIR outperform Spiral in computation, so the Spiral computation will never catch up to WhisPIR without further optimization. Solving for the minimum number of queries before Spiral outperforms WhisPIR in communication gives 120 queries for a 1 GiB database, 58 queries for an 8 GiB database, and 26 queries for a 16 GiB database. It is interesting to observe the gradual convergence of the two protocols as the database size grows. Eventually, as the batch size grows, it will likely make sense for WhisPIR to start taking on more features of the Spiral protocol, such as sending more evaluation keys in the initial upload to reduce the per-query communication. We leave these fine-grain trade-offs for future work, as they more closely resemble a stateful PIR protocol setting.

5 APPLICATION: SECURE BLOCKLIST CHECKING

In this section, we discuss a concrete example application of WhisPIR that takes advantage of both the low communication and fine-grain query batching.

Description. Blocklist checking is a common application in messaging services. The task is to reduce spam by screening messages against a "blocklist", which is a list of known spam or malicious content. Typical contents for a blocklist are URLs that are known to be malicious in some way, such as routing to a malicious website. If a user's app detects a message that is in the blocklist, it will not deliver the message to the user or display a warning, preventing the user from encountering spam or phishing attempts.

If this important user-safety feature is to be implemented in an end-to-end encrypted service, a lookup protocol must be implemented that maintains the message privacy while still informing the user if the received message is contained in the blocklist. We propose using WhisPIR to allow users' applications to privately query the blocklist server to check membership of a message in the blocklist without revealing messages to this host server.

There are several features of this application that make WhisPIR a natural choice. This application requires rapid updates; when a new malicious URL is detected, the database must be updated immediately to reflect this new entry. The application cannot rely on users downloading an updated digest for each new malicious URL that is discovered. In addition, this application could be supporting millions or even billions of users, making the storage of large parameters for each client practically infeasible. Finally, the client application is likely a user's cellphone with limited bandwidth, so minimizing the protocol's communication is critical to maintain practicality.

Database Representation. Applying PIR to this application is not immediately straightforward, since the API of PIR performs an index lookup rather than a key-word membership check. Naively performing a lookup over the space of all URLs would be prohibitively slow. Instead, the server first hashes the URLs in the blocklist with a cryptographic hash function (e.g. SHA3) to map the URLs to 32-byte strings. The server then inserts these values into a Cuckoo hash table using two hash functions, resampling the hash functions on insertion failure. As long as the hash table size is at least twice the number of keys in the hash table, insertion is expected to succeed in constant time, even when considering the time to rebuild the table [PR01]. Regardless, this table construction occurs in a setup phase and has no effect on the query runtime. The only state that may change is resampling the hash functions, but the description of these hash function are short ($O(\lambda)$ bits) and can easily be updated if they change.

The result of this Cuckoo hash table construction is a hash table that is twice the size of the original table where the lookup of a hash

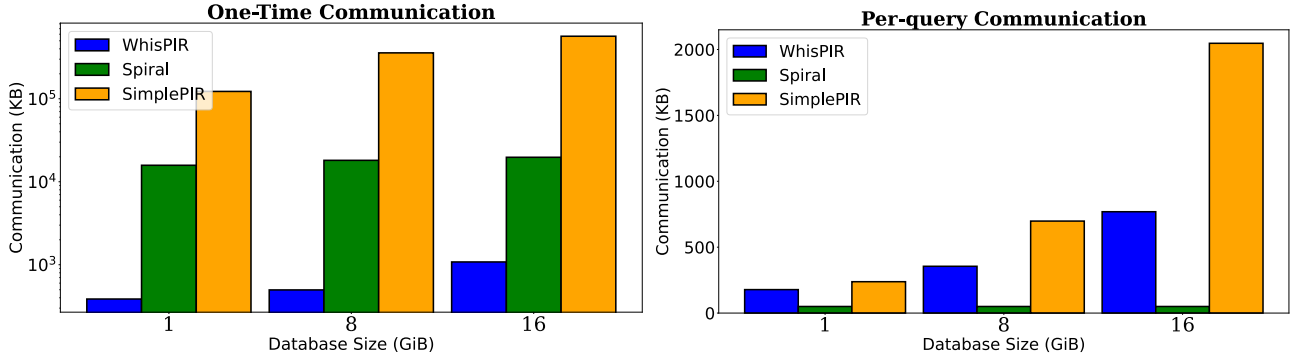

Figure 5: Comparison between WhisPIR and stateful PIR protocols. The left plot indicates the portion of the query that can be reused across multiple queries. Note the log scale on the left plot. The right plot indicates the portion that must be communicated for each query. All WhisPIR values are the minimum communication points from Figure 4.

key requires checking at most two locations in the hash table. To apply WhisPIR, the user's application first hashes the received URL down to a 32-byte string, then uses the Cuckoo hash functions to map this value to two indices in the hash table. The application then acts as the client in WhisPIR to query the hash table at these two indices. The response will be two hash values, which the application checks against the hash of the received URL. If these hashes match, the application determines that the received URL is in the blocklist and proceeds accordingly (e.g. not delivering the message, warning the user, etc.). Otherwise, the application determines that the URL is not in the blocklist and delivers the message normally.

Database Size. Based on conversations with industry experts, it is common for a single malicious URL to map to many entries in a blocklist, since many forms of a URL can route to the same address. For example, the blocked URL `evil.com/a/b/c/` would result in four distinct entries in the blocklist of the form `evil.com/`, `evil.com/a/`, `evil.com/a/b/`, and `evil.com/a/b/c/`. Further regular expression evaluation is performed to normalize URL inputs, but we omit these details as they occur locally on the user's device. Overall, we determine a blocklist consisting of 2^{24} entries is a representative size for practical applications. Each entry is a 32-byte hash, and the hash table is double the size of the input keys. This results in a database of roughly 1 GB that must be queried twice for a URL that the application wishes to test. Note that we only consider pairs of queries to the hash table, which is a query corresponding to a single substring of the URL. A user that receives `evil.com/a/b/c/` will likely want to query all four possibilities, but batching these eight hash table lookups may leak information about the structure of the received URL, potentially compromising privacy.

Performance. To determine the performance WhisPIR on this task at various trade-off points, simply take the benchmarks from Section 4 and double the index upload, the download, and the server computation time. Note that the switching key can be used to process both indices, so this value only needs to be communicated once for a pair of indices. Example values are plotted in Figure 6 along with the estimated performance of the HintlessPIR protocol on this task. Observe that WhisPIR is able to come within 10% of HintlessPIR's runtime while using 25% of the communication, and WhisPIR strictly outperforms HintlessPIR using less than 1/3 the

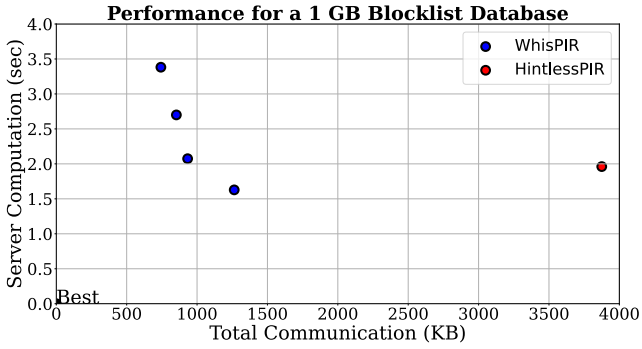

Figure 6: This plot displays the performance of WhisPIR on querying a 1 GB blocklist database. Each query consists of a single pair of PIR indices, one for each Cuckoo hash function. The HintlessPIR performance includes the reuse of the key encryption communication and rotation processing runtime across both indices.

communication. All benchmarks here are taken on a single thread, and practical applications will have significantly more compute resources available for the server's computation. In contrast, there will be essentially no opportunity to change the protocol communication during deployment, so WhisPIR's focus on low communication makes it the clear choice for secure blocklist checking.

REFERENCES

[ACC+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[ACLS18] Sebastian Angel, Hao Chen, Kim Laine, and Srinath Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 962–979, 2018.

[AS16] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, Savannah, GA, November 2016. USENIX Association.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS

'12, page 309–325, New York, NY, USA, 2012. Association for Computing Machinery.

[Gen09] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully homomorphic encryption with polylog overhead. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 465–482, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[HDCGZ23] Alexandra Henzinger, Emma Dauterman, Henry Corrigan-Gibbs, and Nickolai Zeldovich. Private Web Search with Tiptoe. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 396–416, New York, NY, USA, 2023. Association for Computing Machinery.

[HHCG+23] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3889–3905, Anaheim, CA, August 2023. USENIX Association.

[HPS19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An improved rns variant of the bfv homomorphic encryption scheme. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, pages 83–105, Cham, 2019. Springer International Publishing.

[JVC18] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. GAZELLE: A Low Latency Framework for Secure Neural Network Inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, Baltimore, MD, August 2018. USENIX Association.

[KPZ21] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting homomorphic encryption schemes for finite fields. In Mehdi Tibouchi and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 608–639, Cham, 2021. Springer International Publishing.

[KRS+19] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1447–1464, Santa Clara, CA, August 2019. USENIX Association.

[LG15] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In Rainer Böhme and Tatsuaki Okamoto, editors, *Financial Cryptography and Data Security*, pages 168–186, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[LMRSW23] Baiyu Li, Daniele Micciancio, Mariana Raykova, and Mark Schultz-Wu. Hintless Single-Server Private Information Retrieval. Cryptology ePrint Archive, Paper 2023/1733, 2023. https://eprint.iacr.org/2023/1733.

[LPA+19] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1387–1403, New York, NY, USA, 2019. Association for Computing Machinery.

[Lyu12] Vadim Lyubashevsky. Lattice Signatures without Trapdoors. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 738–755, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[MCR21] Muhammad Haris Mughees, Hao Chen, and Ling Ren. OnionPIR: Response Efficient Single-Server PIR. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2292–2306, New York, NY, USA, 2021. Association for Computing Machinery.

[MW22] Samir Jordan Menon and David J. Wu. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947, 2022.

[PIB+22] Bijeeta Pal, Mazharul Islam, Marina Sanusi Bohuk, Nick Sullivan, Luke Valenta, Tara Whalen, Christopher Wood, Thomas Ristenpart, and Rahul Chatterjee. Might I get pwned: A second generation compromised credential checking service. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1831–1848, Boston, MA, August 2022. USENIX Association.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In Friedhelm Meyer auf der Heide, editor, *Algorithms — ESA 2001*, pages 121–133, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[TPY+19] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1556–1571, Santa Clara, CA, August 2019. USENIX Association.

[ZPSZ23] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. Piano: Extremely simple, single-server PIR with sublinear server computation. Cryptology ePrint Archive, Paper 2023/452, 2023. https://eprint.iacr.org/2023/452.

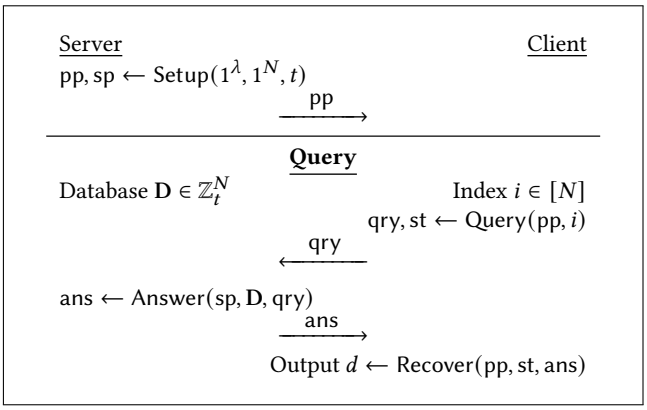

Figure 7: Usage of the PIR API.

A FURTHER BACKGROUND

A.1 PIR Definitions & API Usage

We define basic correctness and security requirements for the semi-honest PIR primitive. Example usage of the PIR API is given in Figure 7.

Definition A.1 (Correctness)**.** *We say that a PIR scheme has* correctness error δ *if, for security parameter* λ*, for all databases* $\mathbf{D} \in \mathbb{Z}_t^N$ *and for all indices* $i \in [N]$*,*

$$\Pr\left[\begin{array}{r|l} \mathsf{Recover}(\mathsf{st}, \mathsf{ans}) & \mathsf{pp}, \mathsf{sp} \leftarrow \mathsf{Setup}(1^\lambda, 1^N, t) \\ \neq \mathbf{D}[i] & \mathsf{qry}, \mathsf{st} \leftarrow \mathsf{Query}(i) \\ & \mathsf{ans} \leftarrow \mathsf{Answer}(\mathbf{D}, \mathsf{qry}) \end{array}\right] \leq \delta.$$

The public parameters pp *and server parameters* sp *are implicit inputs in all algorithms following* Setup*.*

Definition A.2 (Query Hiding)**.** *We say that a PIR scheme is* (T, ϵ)-Query Hiding *if, for all adversaries* $\mathcal{A}$ *running in time at most* T*, on database size* N *and for all* $i, j \in [N]$*,*

$$\Big|\Pr[\mathcal{A}(\mathsf{pp}, \mathsf{sp}, \mathsf{qry}) = 1 : (\mathsf{qry}, \mathsf{st}) \leftarrow \mathsf{Query}(\mathsf{pp}, i)] - \Pr[\mathcal{A}(\mathsf{pp}, \mathsf{sp}, \mathsf{qry}) = 1 : (\mathsf{qry}, \mathsf{st}) \leftarrow \mathsf{Query}(\mathsf{pp}, j)]\Big| \leq \epsilon.$$

A.2 Ring Learning with Errors

The security of our PIR scheme is based on the Ring Learning with Errors (RLWE) [Lyu12] problem. This problem is parametrized by a polynomial ring $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^n + 1)$, where $q \in \mathbb{N}$ is the coefficient modulus and n is a power of 2. In addition to the ring itself, the RLWE problem is specified by an *error* distribution χ over $\mathcal{R}_q$ that samples polynomials with small coefficients. In typical instantiations, including this work, χ outputs elements of $\mathcal{R}_q$ where each coefficient is sampled independently from a discrete, zero-centered Gaussian modulo q.

Definition A.3 (The Ring Learning with Errors (RLWE) problem [Lyu12])**.** *For a ring* $\mathcal{R}_q := \mathbb{Z}_q[x]/(x^n + 1)$*, where* q *is a positive integer* q *and* n *is a power of two, let* χ *be an error distribution over* $\mathcal{R}_q$*. The* $\mathsf{RLWE}_{n,q,\chi}$ *problem is to distinguish samples of the following distributions. The first distribution is defined by a secret* $\mathbf{s} \in \mathcal{R}_q$*. Each*

sample of this distribution consists of two elements $(\mathbf{a}, \mathbf{b}) \in \mathcal{R}_q^2$ *where* $\mathbf{a} \xleftarrow{\$} \mathcal{R}_q$ *and* $\mathbf{b} = \mathbf{a} \cdot \mathbf{s} + \mathbf{e}$*, where* $\mathbf{e} \leftarrow \chi$ *is a fresh error sample. The second distribution is simply the uniform distribution over* $\mathcal{R}_q^2$*, where a sample is* $(\mathbf{a}, \mathbf{u}) \xleftarrow{\$} \mathcal{R}_q^2$.

In short, the hardness of RLWE states that samples

$$(\mathbf{a}, \mathbf{a} \cdot \mathbf{s} + \mathbf{e}) \approx_c (\mathbf{a}, \mathbf{u})$$

are computationally indistinguishable, where all operations are over $\mathcal{R}_q$. Note that an instance of $\mathsf{RLWE}_{n,q,\chi}$ uses a fixed $\mathbf{s}$ across all samples while the error $\mathbf{e}$ is freshly sampled from χ each time. It is typical to sample $\mathbf{s} \leftarrow \chi$.

The following definition will be useful in the analysis of the noise growth throughout the homomorphic computations.

Definition A.4 (Ring Expansion Factor)**.** *For a ring* $\mathcal{R} = \mathbb{Z}[x]/(x^n + 1)$*, define* $\gamma_{\mathcal{R}}$ *as the smallest factor such that for any* $\mathbf{a}, \mathbf{b} \in \mathcal{R}$

$$||\mathbf{a} \cdot \mathbf{b}|| \leq \gamma_{\mathcal{R}} \cdot ||\mathbf{a}|| \cdot ||\mathbf{b}||.$$

When $||\cdot||$ *is the Euclidean norm,* $\gamma_{\mathcal{R}} \leq \sqrt{n}$ *by Cauchy-Schwarz. When* $||\cdot||$ *is the* ℓ_∞ *norm,* $\gamma_{\mathcal{R}} \leq n$.

Remark A.1 (Empirical Expansion Factor)**.** *We leverage empirical results [HPS19, Section 6.1] demonstrating that an expansion factor of* $\gamma_{\mathcal{R}} = 2\sqrt{n}$ *is sufficient to bound* $||\mathbf{a} \cdot \mathbf{b}||_\infty \leq \gamma_{\mathcal{R}} \cdot ||\mathbf{a}||_\infty \cdot ||\mathbf{b}||_\infty$*. The discrete Gaussian output bound is* $||\mathbf{e}||_\infty \leq 6\sigma$ *for a fresh sample* $\mathbf{e} \leftarrow \chi$.

Throughout this work, we will write the analysis in terms of a generic ring expansion factor. In Section 4, we instantiate this expansion factor with the empirical analysis of Remark A.1.

A.3 Precomputed Key Switching API Usage

Figure 8 demonstrates the usage of the precomputed key switching API.

A.4 Further BGV Background

Modulus Switching. One of the primary methods used in BGV to manage the noise level is called modulus switching. This operation reduces the noise level in the ciphertext by reducing the modulus by roughly the same amount. By reducing the magnitude of the noise, we reduce the growth of the noise in the resulting ciphertext. We restate the modulus switching definition and noise growth lemma from BGV [BGV12].

Definition A.5 (Modulus Switching [BGV12])**.** *For a vector* $\mathbf{c} \in \mathbb{Z}_{q_1}^n$ *and integers* $q_1 > q_2 > p$*, we define* $\mathbf{c}' \leftarrow \mathsf{ModSwitch}(\mathbf{c}, q_1, q_2, p)$ *as the vector in* $\mathbb{Z}_{q_2}^n$ *closest to* $(q_2/q_1) \cdot \mathbf{c}$ *that satisfies* $\mathbf{c}' \equiv \mathbf{c} \mod p$.

We abuse notation slightly by writing $\mathsf{ModSwitch}(\mathbf{c}, q_1, q_2, p)$ for $\mathbf{c} \in \mathcal{R}_{q_1}[Y]$ as applying ModSwitch to each coefficient of $\mathbf{c}$ in $\mathcal{R}_{q_1}$ seperately. The output is $\mathbf{c}' \in \mathcal{R}_{q_2}[Y]$ of the same degree.

Lemma A.1 (Modulus Switching Noise [BGV12])**.** *Let* $q_1 > q_2 > p$ *be positive integers satisfying* $q_1 \equiv q_2 \equiv 1 \mod p$*. Let* $\mathbf{s} \leftarrow \chi$ *be a secret, where* χ *has standard deviation* σ*. Let* $\mathbf{c} \in \mathcal{R}_{q_1}$ *be such that*

$$||[\langle \mathbf{c}, \mathbf{s} \rangle]_{q_1}|| < q_1 - p(q_1/q_2) \cdot \gamma_{\mathcal{R}} \cdot ||\mathbf{s}||_\infty,$$

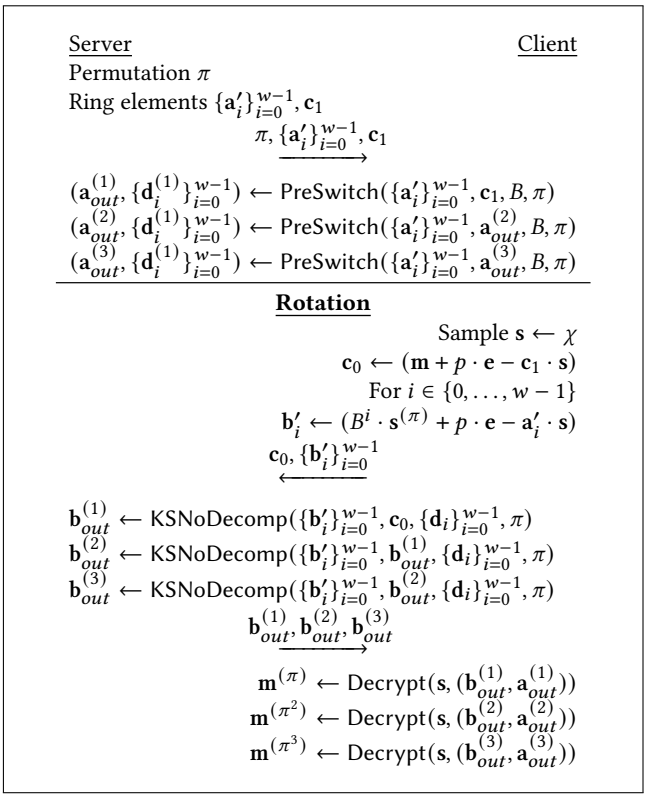

Figure 8: Usage of the precomputed key switching API. The displayed functionality is simply to evaluate three powers of the rotation π. The BGV parameters $\mathcal{R}_q$ and p are implicitly defined. Each error term $\mathbf{e}$ is freshly sampled.

and define $\mathbf{c}' \leftarrow \mathsf{ModSwitch}(\mathbf{c}, q_1, q_2, p)$*. Then, we have*

$$[\langle \mathbf{c}', \mathbf{s} \rangle]_{q_2} = [\langle \mathbf{c}, \mathbf{s} \rangle]_{q_1} \mod p$$
$$||[\langle \mathbf{c}', \mathbf{s} \rangle]_{q_2}|| < \frac{q_2}{q_1} \cdot ||[\langle \mathbf{c}, \mathbf{s} \rangle]_{q_1}|| + p \cdot \gamma_{\mathcal{R}} \cdot ||\mathbf{s}||_\infty. \quad (5)$$

The bound in Equation (5) follows from multiplying the secret with the rounding error from the modulus switch scaling. We will always select moduli such that the scaled down noise $\frac{q_2}{q_1} \cdot ||[\langle \mathbf{c}, \mathbf{s} \rangle]_{q_1}||$ is dominated by the additive $p \cdot \gamma_{\mathcal{R}} \cdot ||\mathbf{s}||_\infty$ term. From Remark A.1, we can bound the overall noise term output by modulus switching by $12\sigma\sqrt{n}$.

Remark A.2 (Compressing Ciphertexts)**.** *Modulus switching enables an important optimization when communicating the results of a homomorphic computation. Regardless of the size of the ciphertext resulting from a homomorphic computation, as long as this ciphertext meets the requirements in Lemma A.1 (i.e. the ciphertext can handle an additive noise of size* $(q_1/q_2) \cdot \gamma_{\mathcal{R}} \cdot ||\mathbf{s}||_\infty$*, which is almost any decryptable ciphertext) then it can be reduced to a minimal modulus before being sent over the network. This minimum modulus is defined by the noise level in Equation* (5)*, meaning that all result ciphertexts have a modulus of size roughly* $p \cdot n$.