FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval

Alex Davidson, Gonçalo Pestana, Sofía Celi

Brave Software

Abstract. We design **FrodoPIR** — a highly configurable, *stateful*, single-server Private Information Retrieval (PIR) scheme that involves an offline phase that is completely *client-independent*. Coupled with small online overheads, it leads to much smaller amortized financial costs on the server-side than previous approaches. In terms of performance for a database of 1 million 1KB elements, FrodoPIR requires < 1 second for responding to a client query, has a server response size blow-up factor of $< 3.6\times$, and financial costs are $\sim \$1$ for answering $100,000$ client queries. Our experimental analysis is built upon a simple, non-optimized Rust implementation, illustrating that FrodoPIR is eminently suitable for large practical deployments.

1 Introduction

A Private Information Retrieval (PIR) scheme provides the ability for clients to retrieve items from an online database, without revealing anything about their queries to the untrusted host server(s). Applications of practical PIR schemes include: anonymous communication [6, 54], anonymous media streaming [43], privacy-preserving ad-delivery [42, 62, 56], private location discovery [34], private contact discovery [14], password-checking [3], and SafeBrowsing [48]. PIR schemes are split into those that are information-theoretically secure, but require the database to be shared between multiple non-colluding servers [4, 24, 8, 10, 28, 9, 11, 67, 33, 32, 26, 37, 65, 52]; and those that are computationally-secure against a single untrusted server [3, 28, 5, 19, 22, 35, 49, 50, 1, 58, 59, 55, 53].

Multi-server PIR constructions are typically much more efficient than single-server schemes. However, finding non-colluding servers to jointly fulfill the PIR functionality is unrealistic in many practical scenarios. To avoid such problems, developing practical single-server PIR schemes is a desirable goal. The most efficient single-server PIR schemes are based on fully homomorphic encryption (FHE), with security derived from the ring learning with errors (RLWE) assumption [1, 5, 55, 3, 58, 53]. Unfortunately, these schemes still incur unmanageable computational, bandwidth, and consequent financial overheads for answering client queries on standard, cloud-based infrastructure.

To drive down online and financial costs, a recent line of work of single-server PIR moves large proportions of the expensive online computation and communication to an offline phase [55, 59, 27] (a technique that also applies in the two-server model [28, 52]). In this model, the client and server prepare an offline internal state to be used for making online queries. Such schemes are referred to as *offline-online* or *stateful*, as opposed to *online-only* or *stateless*. Such works [55, 59, 27] have focused on developing PIR schemes with efficient online phases. The recent work of Corrigan-Gibbs et al. [27], for example, produces a single-server PIR scheme with sublinear efficiency costs.

A key difficulty that has gone unsolved is that either the computation or communication costs induced during the offline phase must scale linearly in the number of clients that will make queries [55, 27, 59]. Moreover, each scheme requires each individual client to make large numbers of queries (e.g. $\sqrt{m}$ for m DB elements) to ensure that the amortized costs remain sublinear. Ultimately, this still results in significant financial costs for any server that plans to run a PIR service in standard cloud-based infrastructure, that will answer queries from large numbers of clients. As a consequence, single-server PIR remains unusable in many real-world applications.

Our results. We build FrodoPIR: a stateful PIR scheme that is built directly upon the learning with errors (LWE) problem only, rather than using RLWE and FHE-based technologies. Similarly to FrodoKEM with respect to lattice-based key exchange [15], we show that — counter to accepted intuition — eschewing ring lattice structures can lead to a flexible, practically efficient PIR scheme. The main benefit of FrodoPIR is that the offline phase of the protocol is performed by the server alone, completely independent of the number of clients or queries that will be made. This results in low amortized computation overheads and an offline client download size that is a tiny fraction of the entire server database.

Our results highlight that the current bottleneck for deploying practical *stateful* PIR schemes is heavily related to the per-client scalability of the offline preprocessing phase. Previous schemes have optimized primarily for per-client asymptotics, which we show do not necessarily translate into financially cheap real-world systems. To this end, FrodoPIR represents an initial exploration in developing stateful PIR schemes that are suitable for large, real-world deployments, where lowering financial costs for server-side operators is of paramount importance. On top of this, FrodoPIR is significantly simpler than previous schemes, making no use of FHE techniques and requiring only modular arithmetic that can be implemented using standard 32-bit unsigned integer instructions.

Our formal contributions are as follows.

1. A stateful single-server PIR scheme, known as FrodoPIR, with security derived from LWE.
2. A simple, open-source Rust implementation — containing only a few hundred lines of code.[1]
3. Experimental analysis that illustrates that FrodoPIR is cheaper to run in large multi-client deployments than all previous single-server PIR schemes.
4. Detailed analysis of various configuration trade-offs and optimizations for FrodoPIR.

[1] https://github.com/brave-experiments/frodo-pir

2 Background

2.1 Overview of Prior Approaches

PIR was first introduced as a cryptographic primitive by Chor, Gilboa, Kushilevitz, and Sudan [26]. Information-theoretic PIR (ITPIR) sees the client interact with multiple non-colluding servers, that each have access to some form of the same database, and the client combines the responses from each server locally [4, 24, 8, 10, 28, 9, 11, 67, 33, 32, 37, 65, 52]. Computationally-secure PIR (cPIR), on the contrary, relies only on a single-server, and provides computational security based on cryptographic assumptions [3, 28, 5, 19, 22, 35, 49, 50, 1, 58, 59, 55]. While ITPIR schemes are more efficient, real-world systems that provide non-collusion guarantees prove very hard to devise in practice. Thus, we focus on cPIR henceforth.

Stateless PIR. Initial constructions of PIR schemes followed the framework of Kushilevitz and Ostrowsky [49], using additively homomorphic encryption (from number-theoretic assumptions) for hiding the client query [19, 22, 35, 50]. Such schemes are known as online-only or stateless, since the client does not have to store any information in order to launch queries. Stateless single-server PIR schemes of this nature have the following underlying structure.

– A client that wishes to learn the i^{th} DB element $\mathsf{DB}[i]$, creates a query vector $\boldsymbol{v}$ of m additively homomorphic ciphertexts, where $\boldsymbol{v}[i]$ encrypts 1 and all others encrypt 0.
– The server responds with a vector $\boldsymbol{w}$, where $\boldsymbol{w}[j] = \boldsymbol{v}[j] * \mathsf{DB}[j]$ ($j \in [m]$, $*$ denotes scalar multiplication).
– The client decrypts $\boldsymbol{w}[i]$ and learns $\mathsf{DB}[i]$.

Sion and Carbunar showed that such schemes actually perform much worse than simply having the client download the entire server database (DB), when the network bandwidth is just a few hundred Kbps [63]. This is a result of performing $O(m)$ expensive arithmetic operations (modular exponentiations or multiplications) for every client query.

The results of [63] stood as a reference point for nearly a decade, until Aguilar-Melchor et al. [1] used lattice-based cryptography (inherently faster than number-theoretic approaches) to construct efficient single-server PIR. In their XPIR scheme server computation time is approximately > 5 seconds for a DB with $m = 2^{20}$ elements, even with the aforementioned asymptotic overheads. Accordingly, bandwidth requirements for the client query are 18MB, and 590KB for the server response. Various schemes both concurrently and since have used RLWE-based FHE to propose similar schemes or optimizations of these methods, such as [31, 5, 58, 3, 55, 53]. In particular, the works of [5, 55, 3, 58] exhibit various optimizations that transform the client query and server database to reduce the size of the query and server response (to around 64KB and 128KB, respectively), whilst maintaining similar computational costs.

Stateful PIR. Unfortunately, stateless cPIR schemes still require computational overheads that are difficult to justify in a large-scale deployment. For example, since it takes around 6.5 seconds to respond to a single client query for a database of 1 million entries [5], such approaches are unlikely to scale for large databases or situations that require timely responses. More recent work has observed that faster online computation can be achieved by moving expensive, query-independent computation to an initial offline phase [59, 55, 28, 27]. This allows reducing the online costs, as well as amortizing the costs of the offline phase across a number of client queries.

The scheme of Patel et al., PSIR [59], enjoys a very fast online phase (less than a second for processing queries on one million-entry database [55]), though the scheme relies on an offline phase that requires the client to download the entire server database — violating the PIR efficiency criterion (Definition 5). The scheme of Mughees et al., Stateful OnionPIR (henceforth SOnionPIR) [55], provides a concretely cheaper approach (from a financial perspective), but at the cost of large computational overheads during the offline phase that is executed as a protocol between each client and the server. More troubling from a financial perspective, these costs scale linearly in the global number of client queries that are launched. While the single-server scheme of Corrigan-Gibbs and Kogan [28] has similar issues as SOnionPIR, the very recent work of Corrigan-Gibbs et al. [27] construct a PIR scheme (henceforth CHKPIR) where all (amortized) asymptotic complexities are sublinear in the number of DB elements, where previous schemes still required $O(m)$ (symmetric) online operations. This reduces further the online costs, but the costs of the offline phase are very similar to the previous works of [59, 55]. In summary, the expensive offline phase in each scheme — that only amortizes per a single client's queries — quickly becomes the main driver of the server-side costs.

The general idea behind each of [59, 55, 27] is that each client and the server cooperatively run a *private batch sum retrieval* protocol that samples c random subsets $S_1, \ldots, S_c$ of elements DB, and computes the sum s_i of all of the elements in each S_i and provides it to the client. During the online phase, the client that wants to query for the element $e_j = \mathsf{DB}[j]$ picks the first $t \in [c]$, where $e_j \notin S_t$. They then construct a partition $\mathcal{P} = (P_1, \ldots, P_k)$ of the indices of DB, where $P_j = S$, and send a succinct description of this partition to the server. The server expands each partition into the set of sums $s_{P_1}, \ldots, s_{P_k}$. The client uses an underlying single-server PIR scheme to learn the sum s_{P_j}, and, finally, outputs $s_{P_j} - s_t$ to learn e_j.

The PSIR scheme implements the private batch sum retrieval protocol by streaming the entire database to the client, while the SOnionPIR and CHKPIR schemes both involve the client specifying their random subsets as FHE ciphertexts, and having the server construct each of the sums using homomorphic properties. When instantiating the underlying single-server PIR scheme during the online phase using FHE-based PIR (such as SealPIR, or *stateless* OnionPIR), it has been shown that these changes result in a much more efficient online phase and significantly smaller server costs, when compared with online-only schemes [59, 55]. Consequently, we focus on these stateful single-server schemes from now on.

Other privacy-preserving data access primitives. Oblivious RAM (ORAM) provides data access pattern privacy for client queries to a server database [39, 40]. This problem is related to PIR, but provides privacy also for the server database: the client learns the queried DB element and nothing more. While recent ORAM schemes enjoy sublinear computation and communication [23, 30, 61, 64], none are inherently multi-client and this leads to very expensive real-world overheads.

Hamlin et al. present Private Anonymous Data Access (PANDA) [44], based on a symmetric-key formulation of PIR known as doubly-efficient PIR [12, 17, 21]. Doubly-efficient PIR schemes are similar to stateful schemes, where there is an initial phase that preprocesses the server database, but the online phase is totally stateless. Unfortunately, symmetric-key doubly-efficient PIR is inherently not multi-client. Public-key instantiations use multiple-servers [12], or are based on expensive cryptographic obfuscation [17]. Batch PIR [46, 45, 7, 51] uses batch codes to achieve sublinear amortized efficiency by allowing clients to retrieve multiple items at once. Unfortunately, such schemes do not provide savings in settings where queries are made *adaptively* — i.e. based on the results of previous queries — which is assumed functionality in standard database and web browsing applications. In this work, we focus on developing PIR schemes that can efficiently handle adaptive client queries.

2.2 Limitations of Existing Stateful PIR Schemes

Expensive preprocessing. The key limitation of SOnionPIR and CHKPIR is the computational cost of the private batch sum retrieval protocol that takes place during the offline phase. This protocol must be invoked per-client, and involves at least $O(m)$ server-side operations and $O(m)$ communication ($m = |\mathsf{DB}|$). These costs are amortized across the number of queries c launched but, even after amortization, the computational costs remain large. For a DB of 2^{20} 1KB elements, the offline phase of SOnionPIR takes 25 seconds *per client query*.[2] For large multi-client systems, the potential for amortization diminishes and these costs quickly become prohibitive.

On the other hand, in PSIR clients must simply download the entire server database before only storing $O(c)$ data. This results in multiple issues. First, as shown in [55], for large numbers of clients the download cost becomes prohibitively large from a financial perspective, and will continue growing for larger databases and items [55]. Second, the PSIR approach is unable to satisfy the fundamental efficiency goal required of PIR schemes (Definition 5).

High online bandwidth consumption. As a result of using FHE-based single-server PIR during the online phase, both SOnionPIR and PSIR have online server response sizes that are relatively very large compared to the size of the

[2] While CHKPIR has not been implemented, the offline phase is very similar and thus will incur similarly large computational overheads.

queried DB element. For example, for 1KB data elements, the response blow-up in SOnionPIR is 128×, while in PSIR it is 320×. The work of CHKPIR uses similar underlying primitives and thus results in similar communication overheads.

Practical security parameters. PSIR and SOnionPIR provide 115 and 111 bits of security, respectively [55, 59] using the primal-USVP cost model for estimating the hardness of cryptographic lattices, as shown in [2]. Achieving 128 bits of security can be important in cases where cryptographic tools must satisfy regulatory compliance checks. Increasing the concrete security parameters of either scheme would require modifying the LWE parameters that are used which, in turn, will significantly impact the efficiency of both schemes.

Simple, available implementations. No stateful PIR scheme has an implementation. More alarmingly, no previous scheme implements their stateful PIR scheme as part of their experimental analysis. This means that the computational overheads of running existing schemes are either extrapolated from stateless PIR implementations, or remain unavailable. Having simple, small, and available implementations is a significant advantage when it comes to assessing the efficiency and security guarantees that are provided, during security and scientific auditing processes.

2.3 Overview of FrodoPIR

A diagrammatic overview of the FrodoPIR approach is given in Figure 1, and involves the following steps.

1. In the offline phase, the server interprets their own database as a matrix, applies a compression function to said matrix and makes the results available as global public parameters. This compression function shrinks the size of the database by a factor of approximately m/λ, where λ is the security parameter and m is the number of database elements.[3]
2. The client downloads the public parameters, and can compute c sets of preprocessed query parameters.
3. In the online phase, the client uses a single set of preprocessed query parameters to produce an "encrypted" query vector, which is sent to the server.
4. The server responds to the query by multiplying the vector with their database matrix.
5. The client returns the result by "decrypting" the response using their preprocessed query parameters.

The security of the system relies on the decisional LWE problem: each client query is a noisy vector that appears uniformly random to the server. Furthermore, security holds up to a global number of client queries that the server witnesses. When this is reached, the server simply reruns the compression function using newly sampled randomness, and the clients download and process the new parameters.

[3] Thus, the size of the parameters is no longer linear in the size of the database.

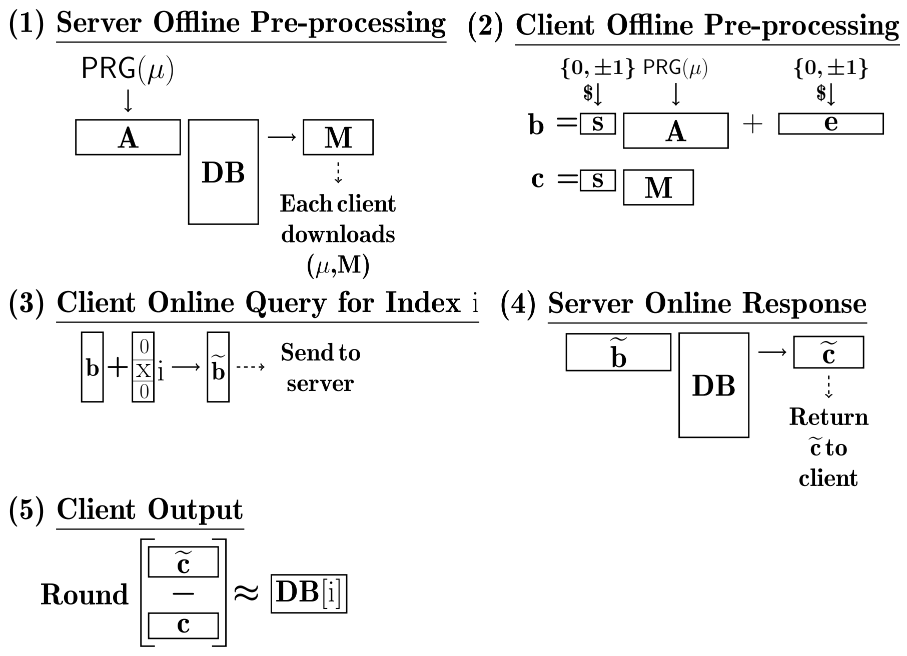
(1) Server Offline Pre-processing
PRG(μ)
A
DB
M
Each client downloads (μ,M)
(2) Client Offline Pre-processing
{0,±1}
$
PRG(μ)
b = s A + e
c = s M
(3) Client Online Query for Index i
b + 0 X 0 i → b̃
Send to server
(4) Server Online Response
b̃
DB
c̃
Return c̃ to client
(5) Client Output
Round [c̃ − c] ≈ DB[i]

Fig. 1. An overview of FrodoPIR. In (1), the server compresses their database **DB** (represented as a matrix) into **M**, via multiplication with the global matrix **A** that is derived randomly from a public seed μ. The client downloads $(\mu, \mathbf{M})$, and in (2) they preprocess a query and store $(\mathbf{b},\mathbf{c})$, note that $\mathbf{b}$ is an LWE sample and is thus randomly distributed. In the online phase, in (3), the client creates their query by adding an indicator value x to the i^{th} vector entry of $\widetilde{\mathbf{b}}$. In (4), the server multiplies the client query vector with their DB matrix and return the result, $\widetilde{\mathbf{c}}$. Finally, in (5), the client subtracts $\mathbf{c}$ from $\widetilde{\mathbf{c}}$ — rounding the result to remove any error terms — and learns the i^{th} row of DB. The full scheme is given in Section 4.

While the ideas behind FrodoPIR are fundamentally similar to previous RLWE-based PIR schemes, the key differentiating factor is that it uses a secure, client-independent preprocessing phase. Moreover, the total client download is much smaller than schemes that involve streaming the entire server database. This trade-off results in a scheme that is significantly cheaper than all previous approaches, including those that achieve sublinear computation and communication complexities such as [27].

The main limitation of the FrodoPIR approach is that the online client queries are linear in the size of the database, whereas previous schemes manage to reduce the size of such queries. Fortunately, we show that FrodoPIR is highly configurable and that we are able to reduce client query sizes (as well as server-side online computation) at the cost of increasing the client download size (see Section 5.4 for more details). We provide a functionality, efficiency, and coarse-level financial comparison between FrodoPIR and previous stateless/stateful PIR schemes

Approach	Security assumptions	Client costs					Server costs				
		Communication		Computation		Storage	Communication		Computation		Financial
		Offline	Online	Offline	Online		Offline	Online	Offline	Online	
Stateless [5, 55, 3]	RLWE	—	m	—	m	—	—	1	—	m	$\$5.2 \times 10^{-3}$
PSIR [59]	RLWE	—	1	m	$\sqrt{m}$	$\sqrt{m}$	$\lvert\mathsf{DB}\rvert/\sqrt{m}$	1	—	m	$\$8.8 \times 10^{-5}$
SOnionPIR [55]	RLWE	$\sqrt{m}$	1	$k \cdot \sqrt{m}$	k	$\sqrt{m}$	$\sqrt{m}$	1	$\sqrt{m}$	m	$\$6.4 \times 10^{-4}$
CHKPIR [27]	RLWE	$\sqrt{m}$	$\sqrt{m}$	$\sqrt{m}$	$\sqrt{m}$	$\sqrt{m}$	$\sqrt{m}$	1	$\sqrt{m}$	$\sqrt{m}$	$\sim \$8.8 \times 10^{-5\dagger}$
FrodoPIR	LWE	—	m	m	1	λ	$\lambda \cdot m^{-1/2}$	1	$\sqrt{m}/C$	m	$\$(1.9/C \times 10^{-2} + 1.3 \times 10^{-5})$

Fig. 2. Asymptotic comparison (ignoring logarithmic factors) of practical approaches for realising single-server PIR with adaptive queries (i.e. not including batch PIR schemes). All costs are amortized according to C clients that launch $c = \sqrt{m}$ queries ($m = \lvert\mathsf{DB}\rvert$ is the total number of elements in the server database). Communication costs relate to the amount of data that is *sent* to the party. The financial costs are given relative to a database containing 2^{20} 1KB elements, are amortized per-query and per-client, and are calculated assuming a server that operates the same AWS EC2 architecture specified in Section 6. †The costs of CHKPIR are assumed to be zero for the online phase, and are thus completely dominated by the offline phase, which can be implemented using techniques from [59, 55, 27].

in Figure 2. We illustrate how amortization of offline computation across *all* client leads to significant efficiency advantages in the experimental analysis of Section 6.

3 Preliminaries

3.1 Notation

We denote vectors and matrices in lower- and upper-case bold-face, respectively. All vectors $\boldsymbol{v}$ are assumed to be in column orientation, and we write $\boldsymbol{v}^T$ to denote the same vector in row orientation. For a set of vectors $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_\ell$, we write $[\boldsymbol{x}_1 \| \cdots \| \boldsymbol{x}_\ell]$ to denote the matrix with the i^{th} column set to equal $\boldsymbol{x}_i$ for $i \in [\ell]$.

Let $\lfloor x \rceil \in \mathbb{Z}$ denote rounding $x \in \mathbb{R}$ to the nearest integer, rounding down in case of a tie. Likewise, we use $\lceil x \rceil$ to indicate explicitly rounding $x \in \mathbb{R}$ to the next highest integer. For $\boldsymbol{x} \in \mathbb{Z}_q^m$, we write $\lfloor \boldsymbol{x} \rceil_p$ to denote the computation of $\lfloor p/q \cdot \boldsymbol{x} \rceil$, where the rounding is applied to each entry of $\boldsymbol{x}$ individually. For some set $\mathcal{X}$, we write $x \leftarrow\!\$\, \mathcal{X}$ to denote that x is sampled from $\mathcal{X}$ uniformly, and we write $\boldsymbol{x} \leftarrow\!\$\, (\mathcal{X})^m$ to denote sampling an m-dimensional vector, with each entry sampled uniformly from $\mathcal{X}$. We write $\log(x)$ to denote taking the base-2 logarithm of x. We use λ to denote the security parameter throughout, and say that an algorithm $\mathcal{A}$ is PPT if it runs in probabilistic polynomial-time with respect to λ.

3.2 Mathematical Preliminaries

We use the learning with errors (LWE) problem in its decisional version, which is equivalent to its search version as proven by Regev [60].

Definition 1. (Decisional LWE problem) *Let χ be some distribution, and let $q, n, m > 0$ depend on λ. The* decision LWE problem ($\mathsf{LWE}_{q,n,m,\chi}$) *is to distinguish between:*

$$(\boldsymbol{A}, \boldsymbol{s}^T \cdot \boldsymbol{A} + \boldsymbol{e}^T) \in \mathbb{Z}_q^{n\times m} \times \mathbb{Z}_q^m, \tag{1}$$
$$(\boldsymbol{A}, \boldsymbol{u}) \in \mathbb{Z}_q^{n\times m} \times \mathbb{Z}_q^m, \tag{2}$$

where $\boldsymbol{A} \leftarrow\!\!\$\, \mathbb{Z}_q^{n\times m}$, $\boldsymbol{s}^T \leftarrow (\chi)^n$, $\boldsymbol{e}^T \leftarrow (\chi)^m$, and $\boldsymbol{u} \leftarrow\!\!\$\, \mathbb{Z}_q^m$.

Evidence that the $\mathsf{LWE}_{q,n,m,\chi}$ problem is hard to solve for appropriate choices of χ — for example, uniform binary or small Gaussian distributions — and for both classical and quantum adversaries follows via reduction from standard lattice problems [60] (as hard as worst case problems on n-dimmesional lattices). The following corollary follows from the work of Brakerski et al. [18], and states that decisional $\mathsf{LWE}_{q,n,m,\chi}$ remains hard when χ is the uniform distribution over ternary values (i.e. $\{0, \pm 1\}$).

Corollary 1. (Ternary LWE [18]) *The $\mathsf{LWE}_{q,n,m,\chi}$ problem is hard to solve when χ is the uniform distribution on $\{-1, 0, 1\}$ (i.e. the uniform ternary distribution).*

In Definition 2 we give a variant of $\mathsf{LWE}_{q,n,m,\chi}$ known as the Matrix LWE problem (denoted by $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$). Corollary 2 shows that $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$ is hard to solve, with only polynomial security loss compared with $\mathsf{LWE}_{q,n,m,\chi}$ [15].

Definition 2. (Decisional Matrix LWE problem [15]) *Let χ be some distribution, and let $q, n, m, \ell > 0$ depend on λ. The* decisional Matrix LWE problem *($\mathsf{MatLWE}_{q,n,m,\chi,\ell}$) is to distinguish between:*

$$(\boldsymbol{A}, \boldsymbol{S} \cdot \boldsymbol{A} + \boldsymbol{E}) \in \mathbb{Z}_q^{n\times m} \times \mathbb{Z}_q^{\ell\times m}, \tag{3}$$
$$(\boldsymbol{A}, \boldsymbol{U}) \in \mathbb{Z}_q^{n\times m} \times \mathbb{Z}_q^{\ell\times m}, \tag{4}$$

where $\boldsymbol{A} \leftarrow\!\!\$\, \mathbb{Z}_q^{n\times m}$, $\boldsymbol{S} \leftarrow (\chi)^{\ell\times n}$, $\boldsymbol{E} \leftarrow (\chi)^{\ell\times m}$, and $\boldsymbol{U} \leftarrow\!\!\$\, \mathbb{Z}_q^{\ell\times m}$.

Corollary 2. (Hardness of $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$ [15]) *Let $\mathcal{A}$ be a PPT adversary against $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$ with advantage ϵ, then we can construct a PPT adversary $\mathcal{B}$ against $\mathsf{LWE}_{q,n,m,\chi}$ with advantage ϵ/ℓ.*

We now state the following as a corollary of the central limit theorem [57], to provide an upper bound on the size of sums of elements sampled from uniform ternary distributions.

Corollary 3. (Bounds on uniform ternary sums) *For sufficiently large m, the sum of m samples taken from the uniform distribution over ternary values (i.e. $\{-1, 0, 1\}$) is bounded by $4\sqrt{m}$ with all but negligible probability.*

3.3 Stateful Private Information Retrieval

As discussed, in this work, we will consider *stateful* PIR schemes, where the PIR interactions are split into a query-independent offline phase and a query-dependent online phase [59]. A stateful PIR scheme consists of an offline and an online phase, which are defined as follows.

Offline phase.

– $\mathsf{ssetup}(1^\lambda)$: An algorithm run by the server that outputs some initialization parameters ip.
– $\mathsf{cinit}(\mathsf{ip})$: A client initialization algorithm run on parameters ip. Outputs a message msg to be sent to the server during the offline phase.
– $\mathsf{spreproc}(\mathsf{ip}, \mathsf{DB}, msg)$: A server preprocessing algorithm run on ip, the server database DB, and client message msg. Outputs a set of public parameters pp to be downloaded by the client.
– $\mathsf{cpreproc}(\mathsf{ip}, \mathsf{pp})$: A client preprocessing algorithm run on the server-generated public parameters $(\mathsf{ip}, \mathsf{pp})$, that outputs a state st.

Stateful PIR schemes that omit the cinit algorithm are said to have *client-independent* preprocessing phases.

Online phase.

– $\mathsf{query}(\mathsf{st}, i)$: An algorithm run by the client that generates a PIR query q for the item in the i^{th} index of the server database.
– $\mathsf{respond}(\mathsf{DB}, \mathsf{q})$: A server algorithm that outputs a response r to be returned to the client.
– $\mathsf{process}(\mathsf{st}, \mathsf{r})$: A client algorithm that takes the sever response r, and outputs a database element x.

3.4 PIR requirements

Stateful PIR schemes must guarantee the following.

Correctness. The following correctness definition ensures that clients receive the correct response with overwhelming probability when interacting with an honest server.

Definition 3. (Correctness) *Let* DB *be the server database, let* i *be the index that the client seeks to query during the online phase, and let* $\mathsf{DB}[i]$ *be the* i^{th} *entry of DB. We say a PIR scheme is* correct *if the following inequality is satisfied.*

$$\Pr\left[\mathsf{x} = \mathsf{DB}[i] \left| \begin{array}{l} \mathsf{ip} \leftarrow \mathsf{ssetup}(1^\lambda) \\ \mathsf{pp} \leftarrow \mathsf{spreproc}(\mathsf{ip}, \mathsf{DB}, \mathsf{cinit}(\mathsf{ip})) \\ \mathsf{st} \leftarrow \mathsf{cpreproc}(\mathsf{ip}, \mathsf{pp}) \\ \mathsf{q} \leftarrow \mathsf{query}(\mathsf{st}, i) \\ \mathsf{r} \leftarrow \mathsf{respond}(\mathsf{DB}, \mathsf{q}) \\ \mathsf{x} \leftarrow \mathsf{process}(\mathsf{st}, \mathsf{r}) \end{array} \right.\right] > 1 - \mathsf{negl}(\lambda)$$

Security. We use the standard definition of security in enforcing the indistinguishability of client queries.

Definition 4. (1-query indistinguishability) *Let* DB *be the server database. Generate the server public parameters by running* $\mathsf{ip} \leftarrow \mathsf{ssetup}(1^\lambda)$ *and subsequently* $\mathsf{pp} \leftarrow \mathsf{spreproc}(\mathsf{ip}, \mathsf{DB}, \mathsf{cinit}(\mathsf{ip}))$*, and let* $\mathsf{st} \leftarrow \mathsf{cpreproc}(\mathsf{ip}, \mathsf{pp})$ *be the client state. We say that a PIR scheme is* secure *if, for any PPT adversary* $\mathcal{A}$ *specifying indices* i_0, i_1 *that is given* $\mathsf{q}_b \leftarrow \mathsf{query}(\mathsf{st}, i_b)$ *for* $b \leftarrow\!\!\$\, \{0, 1\}$*, then* $\mathcal{A}$ *has negligible probability in correctly guessing* b*.*

The above definition can be expanded to specify ℓ-query indistinguishability, in other words that two sets of size ℓ of client queries are indistinguishable from each other [59].

Efficiency. PIR schemes require a communication overhead smaller than the solution of having clients download the entire server database. In the stateful PIR case, it should hold when amortizing costs over the number of client queries.

Definition 5. (Efficiency) *For a single client launching* c *queries, a PIR scheme is* efficient *if the total client download communication overhead is smaller than* $|\mathsf{DB}|/c$*.*

For stateful schemes, the total client download cost is calculated using the equation: $\mathit{comms}(\texttt{offline}) + c \cdot \mathit{comms}(\texttt{online})$.

4 Our Scheme

We now describe the $\mathsf{FrodoPIR}$ scheme, writing FPIR for short.

4.1 Cryptographic Setup

Recall that $\mathcal{S}$ is the server holding the database DB that each client attempts to learn entries from. The DB is an array of m elements, each made up of w bits. Each entry is associated with the index i that corresponds to its position in the array. For now, we will assume that the client knows which index it would like to query during the online phase of the protocol.[4] We assume that there are C clients that will each launch a maximum of c queries against DB. Regarding the LWE instantiation that is used: let n and q be the LWE dimension and modulus, respectively; let $0 < \rho < q$; let χ be the uniform distribution over $\{-1, 0, 1\}$; and let λ be the concrete security parameter. Finally, we use $\mathsf{PRG}(\mu, x, y, q)$ to denote a pseudorandom generator that expands a seed $\mu \leftarrow\!\!\$\, \{0, 1\}^\lambda$ into a matrix in $\mathbb{Z}_q^{x \times y}$.

4.2 Preprocessing Phase

We first describe the offline phase which occurs before the client makes any queries to the server. Note that cinit is not required in $\mathsf{FrodoPIR}$, and thus we do not define it.

[4] Section 7 discusses options for mapping string-based queries to indices.

Server setup (FPIR.ssetup)**.** The server constructs their database containing m elements, and samples a seed $\mu \in \{0,1\}^\lambda$.

Server preprocessing (FPIR.spreproc)**.** The server derives a matrix $\boldsymbol{A} \leftarrow \mathsf{PRG}(\mu, n, m, q)$, where $\boldsymbol{A} \in \mathbb{Z}_q^{n\times m}$. It then runs $\boldsymbol{D} \leftarrow \mathsf{parse}(\mathsf{DB}, \rho)$, where parse encodes the DB into a matrix $\boldsymbol{D} \in \mathbb{Z}_\rho^{m\times\omega}$, and where $\omega = \lceil w/\rho \rceil$.[5] The server then stores $\boldsymbol{D}$.

To generate public parameters, the server runs $\boldsymbol{M} \leftarrow \boldsymbol{A} \cdot \boldsymbol{D}$, and then publishes the pair $(\mu, \boldsymbol{M}) \in \{0,1\}^\lambda \times \mathbb{Z}_q^{n\times\omega}$ to a public repository accessible by clients.

Client preprocessing (FPIR.cpreproc)**.** Each client downloads $(\mu, \boldsymbol{M})$ from the public repository, and runs $\boldsymbol{A} \leftarrow \mathsf{PRG}(\mu, n, m, q)$. The client then samples c vectors $\boldsymbol{s}_j \leftarrow (\chi)^n$ and $\boldsymbol{e}_j \leftarrow (\chi)^m$. Finally, it computes $\boldsymbol{b}_j \leftarrow {\boldsymbol{s}_j}^T \cdot \boldsymbol{A} + {\boldsymbol{e}_j}^T \in \mathbb{Z}_q^m$ and $\boldsymbol{c}_j \leftarrow {\boldsymbol{s}_j}^T \cdot \boldsymbol{M} \in \mathbb{Z}_q^\omega$, for each $j \in [c]$, and stores the pairs internally as the set $X = (\boldsymbol{b}_j, \boldsymbol{c}_j)_{j\in[c]}$.

4.3 Online Phase

Client query generation (FPIR.query)**.** For the index i that the client wishes to query, the client generates a vector $\boldsymbol{f}_i = (0, \ldots, 0, q/\rho, 0, \ldots, 0)$, i.e. the all-zero vector except where $\boldsymbol{f}_i[i] = q/\rho$. The client then pops a pair $(\boldsymbol{b}, \boldsymbol{c})$ from the internal storage X that it maintains, and computes $\widetilde{\boldsymbol{b}} \leftarrow \boldsymbol{b} + \boldsymbol{f}_i$, and sends $\widetilde{\boldsymbol{b}}$ to the server.

Server response (FPIR.respond)**.** The server receives $\widetilde{\boldsymbol{b}}$ from the client, and responds with $\widetilde{\boldsymbol{c}} \leftarrow \widetilde{\boldsymbol{b}}^T \cdot \boldsymbol{D} \in \mathbb{Z}_q^\omega$.

Client postprocessing (FPIR.process)**.** The client receives $\widetilde{\boldsymbol{c}}$, and outputs $x \leftarrow \lfloor \widetilde{\boldsymbol{c}} - \boldsymbol{c} \rceil_\rho$.

4.4 Correctness

The output of the client postprocessing phase is $x \leftarrow \lfloor \widetilde{\boldsymbol{c}} - \boldsymbol{c} \rceil_\rho$. Expanding the right-hand side of the equation gives:

$$\begin{aligned} x &\leftarrow \lfloor \widetilde{\boldsymbol{c}} - \boldsymbol{c} \rceil_\rho \\ &= \lfloor (\boldsymbol{s}^T \cdot \boldsymbol{A} + \boldsymbol{e}^T + {\boldsymbol{f}_i}^T)^T \cdot \boldsymbol{D} - (\boldsymbol{s}^T \cdot \boldsymbol{A} \cdot \boldsymbol{D}) \rceil_\rho \\ &= \lfloor (\boldsymbol{e} + \boldsymbol{f}_i)^T \cdot \boldsymbol{D} \rceil_\rho. \end{aligned} \tag{5}$$

Noting that $\lfloor {\boldsymbol{f}_i}^T \cdot \boldsymbol{D} \rceil_\rho = \boldsymbol{D}[i]$ where the i^{th} row $\boldsymbol{D}[i] \in \mathbb{Z}_\rho^\omega$ is interpreted as a column vector, then proving that

$$\lfloor \boldsymbol{e}^T \cdot \boldsymbol{D} + {\boldsymbol{f}_i}^T \cdot \boldsymbol{D} \rceil_\rho = \lfloor {\boldsymbol{f}_i}^T \cdot \boldsymbol{D} \rceil_\rho \tag{6}$$

results in the client learning the correct output. This gives rise to the following theorem.

[5] Thus, the i^{th} row consists of ω $\log(\rho)$-bit chunks of $\mathsf{DB}[i] \in \mathbb{Z}_\rho^\omega$.

Theorem 1. (Correctness) *If $q \geq 8\rho^2\sqrt{m}$, then* FPIR *is* correct *with high probability.*

Proof. See Appendix A.1.

4.5 Security

To prove security of FrodoPIR, we show that any query $\widetilde{\boldsymbol{b}} \leftarrow \mathsf{FPIR.query}(i)$ is distributed uniformly in $\mathbb{Z}_q^m$ from the perspective of $\mathcal{S}$ (Theorem 2). This general result proves that FPIR satisfies 1-query indistinguishability (Definition 4) and we further show that this holds for $\ell = \mathsf{poly}(\lambda)$ client queries in Corollary 4.

Theorem 2. (1-query indistinguishability) FPIR *is secure under observation of* 1 *query, under the assumption that* $\mathsf{LWE}_{q,n,m,\chi}$ *is difficult to solve.*

Proof. See Appendix A.2.

Corollary 4. (ℓ-query indistinguishability) FPIR *is secure under observation of* $\ell = \mathsf{poly}(\lambda)$ *queries, under the assumption that* $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$ *is difficult to solve.*

Proof. See Appendix A.3.

4.6 Efficiency

We give the conditions under which FPIR satisfies the efficiency goal of a PIR scheme, as laid out in Definition 5.

Theorem 3. (Efficiency) *Let c be the upper bound of a single client's* FPIR *queries. Then* FPIR *is* efficient *when:*

$$128 + n\omega \log(q) + c\omega \log(q) < \ |DB|.$$

Proof. This follows by applying Definition 5, considering the communication costs of FrodoPIR (Figure 4).

5 Parameter Settings and Configurations

We now describe parameter settings and potential optimizations that demonstrate the versatility of FrodoPIR. The major parameters of the scheme to be configured are: the concrete security parameter λ; the LWE dimension n; the LWE modulus q; the uniform ternary distribution, χ, used for sampling LWE secret and error vectors; the number of bits, ρ, packed into each entry of the DB matrix, $\boldsymbol{D}$; the number of elements, m, in the server DB; the bit-length, w, of each element in DB; and c, the number of queries that a single client makes.

The number of required computational operations is given in Figure 3, the communication overheads in Figure 4, and the storage overheads in Figure 5.[6] Clearly, the aforementioned parameters all have an impact on the protocol efficiency.

[6] Recall that $\omega = w/\log(\rho)$.

	spreproc	cpreproc	query	respond	process
mod q mults	$nm\omega$	$n(m+\omega)$	—	$m\omega$	—
mod q adds	$n(m-1)\omega$	$(n-1)(m+\omega)$	1	$(m-1)\omega$	ω
PRG	nm	nm	—	—	—

Fig. 3. Number of operations required in FrodoPIR.

	Offline	Online
Client upload	—	$m\log(q)$
Client download	$128 + n\omega\log(q)$	$\omega\log(q)$

Fig. 4. Communication overheads (bits) of FrodoPIR.

5.1 Required Invariants

Firstly, FrodoPIR must satisfy Theorem 3:

$$128 + n\omega\log(q) + c\omega\log(q) < mw. \tag{7}$$

This equation is satisfied for very large values of c (e.g. $c > 18,000$ for $m = 2^{16}$).

Secondly, for correctness (Theorem 1), we must have that:

$$q \geq 8\rho^2\sqrt{m}, \tag{8}$$

holds. Finally, for security, $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$ must provide at least 128 bits of concrete classical security. We can estimate the concrete security of LWE instances with the lattice security estimator [2].

5.2 Fixing LWE Parameters

Before configuring FrodoPIR for efficiency, we first fix a set of parameters that provide the necessary concrete security guarantees. We focus on those parameters for $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$, except for m which is the number of DB elements.

Firstly, χ is the uniform ternary distribution. Secondly, we set $q = 2^{32}$, which allows us to use standard 32-bit unsigned integer operations for the implementation of the $\mathbb{Z}_q$ operations. Thirdly, we set $n = 1774$ as the LWE dimension. In order to concretely estimate the security of the $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$ instance, we will use the work of Albrecht et al. [2] to provide the security of $\delta = \mathsf{LWE}_{q,n,m,\chi}$, and then calculate the concrete security parameter as $\lambda = \delta/\log\ell$ (Corollary 2). As ℓ is the total number of queries that the server observes, we set $\ell = 2^{52}$ queries as a suitable upper bound before rotation of $\boldsymbol{A}$ is required. Therefore, $\lambda = \delta - 52$.

5.3 Recommended Database Parameters

Let $\kappa = (\log(\rho)\cdot m)/(n\cdot\log(q))$ denote the improvement factor relative to the offline client download when compared to the original DB size. In Figure 6, we give recommended parameter settings for the FrodoPIR scheme.

	with preprocessing	without
Server storage	$128 + m\omega\log(\rho)$	$128 + m\omega\log(\rho)$
Client storage	$128 + c(m+\omega)\log(q)$	$128 + n\omega\log(q)$

Fig. 5. Storage overheads of FrodoPIR in bits, according to whether client performs any offline preprocessing of queries (where c is the number of preprocessed queries), or not. In the case of no preprocessing being performed, the client storage overhead is logarithmically dependent on the number of elements in DB.

q	2^{32}	2^{32}	2^{32}	2^{32}	2^{32}
n	1774	1774	1774	1774	1774
m	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}
ρ	2^{10}	2^{10}	2^{10}	2^{9}	2^{9}
κ	13.028	26.056	52.112	93.802	187.603
λ	128	128	128	128	128

Fig. 6. Database, query, and security parameter settings for FrodoPIR.

For each parameter set, the concrete security parameter is 128 bits for over 2^{52} client queries. For larger numbers of queries, the concrete security of the instance will decrease until a new LWE matrix $\boldsymbol{A}$ is resampled, and the server updates its public parameters. Security can be increased by increasing the dimension n, though, this reduces κ. We use this in Appendix B, when applying FrodoPIR for online SafeBrowsing API checks.

In Section 6, we consider DB elements of size $w = 1$KB, which leads to $\omega \in \{820, 911\}$, depending on the value of ρ. Changing w has a direct impact on performance.

5.4 Additional Optimizations

Processing larger databases via sharding. As m increases beyond 2^{20}, we see a greater relative saving of bandwidth costs relative to the fixed n that is used (as parameterized by κ). However, this has undesirable impacts on the performance of the scheme. First, all online server-side computation in the online phase is linearly dependent on m, and so increasing m immediately results in higher latency. The offline work scales similarly for client devices, which are typically constrained and unlikely to cope with vast overheads. Second, the client query size rapidly grows as it is also linearly dependent on m.

Overall, we expect that the best approach for operating on larger databases is to *shard* them into s parallel instances, each using a database of size m/s. Each instance can then independently process the *same* single client query. This allows the client to learn the i^{th} index from each of the s shards from only a single query. This allows parallelization of server computation, and careful management of available computing resources. On the client-side, the size of the online query is linear in m/s, rather than m, which can lead to practical communication sizes. Previous work has already highlighted the benefits of performing such sharding on the server database [31] in terms of increasing amortization factors and allowing further degrees of parallelization.

Note that each client must download the public parameters of each of the individual shards. This increases the size of the client download, but with the benefits of reducing the size of their own query and reducing server-side latency. Additionally, noting the independence of each server-side vector-column multiplication in FrodoPIR, we could equally leverage sharding by splitting the server database matrix into smaller subsets of columns for handling larger data elements.

Database updates. Sharding alone does not reduce the client overhead in preprocessing queries, which remain linear in the total database size. This can become expensive if the server database is updated frequently: each time the client has to regenerate their preprocessed query data.

However, coupling sharding with a database updating procedure that touches only few of the shards can reduce database updates to only re-running the ssetup, cpreproc, and spreproc procedures on a small fraction of the database. Specifically, if database updates are confined to a single shard of the database, then these procedures need only be run on that particular shard after every update. Updating a single shard of the database results in only requiring the client to download and process an amount of data that is a $1/(\kappa \cdot s)$ fraction of the entire database. Even for large databases, this fraction is likely to be very small.

Achieving logarithmic client-storage overhead. In Figure 5, it is clear that the storage overheads for the client are dependent on c, the number of preprocessed queries. These costs can be reduced significantly to being logarithmically dependent on m, by simply not performing any preprocessing. The reason that the costs are logarithmic is that the client storage is equal to $(\lambda + n\omega \log(q))$ where, as mentioned in Section 5, q is chosen to be equal to $8\rho^2\sqrt{m}$. This approach requires derivation of the matrix $\boldsymbol{A}$ and query parameters for every online query. Since the derivation of $\boldsymbol{A}$ is fairly costly, computation-constrained clients will benefit from preprocessing client queries.

6 Experimental Analysis

We provide an experimental analysis of the incurred computational runtimes, bandwidth usage, and financial costs when running FrodoPIR. Further, we highlight how such costs amortize over the costs of the offline preprocessing phase. Finally, we compare these costs with the previous stateful PIR schemes — PSIR, SOnionPIR, and CHKPIR — of [59, 55, 27].

Benchmarking equipment. We run all experiments as single-threaded processes on an Amazon `t2.2xlarge` EC2 instance, with 8 CPU cores and 32GB of RAM. At the time of writing, the cost of transferring data from server to clients is \$0.09 per GB, the cost of data transfer from clients to server is free, and the cost of computation is \$0.3712 per hour of usage (or \$0.0464 per CPU hour).[7] This is equivalent to the setup that is used in [55] for comparing SOnionPIR and PSIR.[8]

Database configurations. We provide non-amortized communication and computation benchmarks for a single server database using each of the parameter settings provided in Figure 6. We choose $w = 2^{13}$ bits (or 1KB); and $\omega = 820$ for $m \leq 2^{18}$, and $\omega = 911$ otherwise. These parameters provide 128-bit security for around 2^{52} client queries.

Source code. Our open-source[9] implementation of FrodoPIR is written in Rust, consists of 735 lines of code for the main functionality, and requires no external dependencies relating to cryptographic operations. All modular arithmetic is implemented using instructions associated with the 32-bit unsigned integer type included in the Rust standard library.

Example application. In Appendix B, we further illustrate how FrodoPIR can be applied to real-world use-cases, taking, as an example, the Google SafeBrowsing API [41].

	Number of DB items ($\log(m)$)	16	17	18	19	20
Offline	Client download (KB)	5682.47	5682.47	5682.47	6313.07	6313.07
	Database preprocessing (s)	92.409	185.30	374.56	825.50	1679.8
	Client derive params (s)	0.5208	1.042	2.1	4.29	8.39
	Client query preprocessing (s)	0.134	0.265	0.532	1.058	2.111
Online	Client query (KB)	256	512	1024	2048	4096
	Server response (KB)	3.203	3.203	3.203	3.556	3.556
	Client query (ms)	0.0177	0.0454	0.0813	0.1565	0.3328
	Server response (ms)	45.74	89.57	179.3	397.06	779.75
	Client output (ms)	0.418	0.4182	0.416	0.4559	0.4627

Fig. 7. Non-amortized performance analysis of FrodoPIR. The "Client derive params" cost refers to the cost of deriving the LWE matrix $\boldsymbol{A}$ from the seed μ, while "Client query preprocessing" refers to the cost of query-independent preprocessing required for a single query. The server offline phase costs can be amortized *globally* across the number of queries (C) that are performed, while the client download and parameter derivation costs can be amortized across the number of queries (c) that they individually make.

[7] `https://aws.amazon.com/ec2/pricing/on-demand/`, accessed Jan 2022.
[8] The financial costs that SOnionPIR quote for running this instance are cheaper than we mention here, due to the passage of time between both works.
[9] `https://github.com/brave-experiments/frodo-pir`

6.1 Performance Results

In Figure 7, we provide non-amortized performance results for FrodoPIR. This involves running a single-threaded server instance, and calculating the running times and bandwidth usage when interacting with a single client. Later, we analyze how the offline costs amortize on a per-query basis. Amortization is calculated over the number of clients C, and the number of queries c each client makes (where we set $c = 500$ for all experiments).

Offline phase. The server generates their database matrix DB and public parameters. This is a client-independent operation that scales linearly in m. This process includes pseudorandom derivation of the LWE matrix $\boldsymbol{A} \in \mathbb{Z}_q^{n \times m}$ from a single 128-bit seed, which must also be computed by each client. After downloading the public parameters, the client performs query-independent preprocessing for each query that they will make. The results of preprocessing are used during the online phase. These costs grow roughly linearly in m.

In terms of communication, the server publishes the 128-bit seed, μ, and the matrix $\boldsymbol{M} \in \mathbb{Z}_q^{n \times \omega}$, where $\omega = w/\log(\rho)$. The size of the client download is static for each choice of $\log(\rho)$. As a consequence, the total cost only grows when increasing m dictates that ρ must also decrease.

Online phase. The client computation consists of performing a single addition operation to modify a single portion of preprocessed data. The client also performs a very small amount of postprocessing of servers responses that is almost static across all experiments as it is linear in ω. The dominant computation cost is the server-side processing of the client query that is ≤ 0.8s for all database sizes.

The dominant communication cost relates to the client query, which is equal to $m \log(q)$ bits and scales linearly in the DB size. The server response is significantly smaller — $\omega \log(q)$ bits — resulting in a $< 3.6\times$ overhead in the server response size compared with the original 1KB data element.

Amortization of offline phase. Many of the offline costs in Figure 7 can be amortized significantly over the number of queries that will be launched. In Figure 8, we give an overview of the rate of this amortization for DB preprocessing and parameter generation steps, as well as client download. While the cost of the DB preprocessing is an expensive one-off cost, it is amortized over all queries *globally*, i.e. over all clients. The client preprocessing and download size amortizes over the value of c.

The total amortized computation cost (per-query) for the server and clients are given in Figure 9. The majority of server costs occur during the relatively cheap online phase. The majority of client work is performed during the query-independent offline phase, part of which (derivation of $\boldsymbol{A}$) can be amortized over c. Online costs for clients are very small.

Storage costs. Figure 10 shows that, when the client preprocesses c queries during the offline phase, we see a linear growth in the storage overhead associated with the database size. This overhead becomes fairly large when $|\mathsf{DB}| = 2^{20}$: the client storage is roughly 2GB.

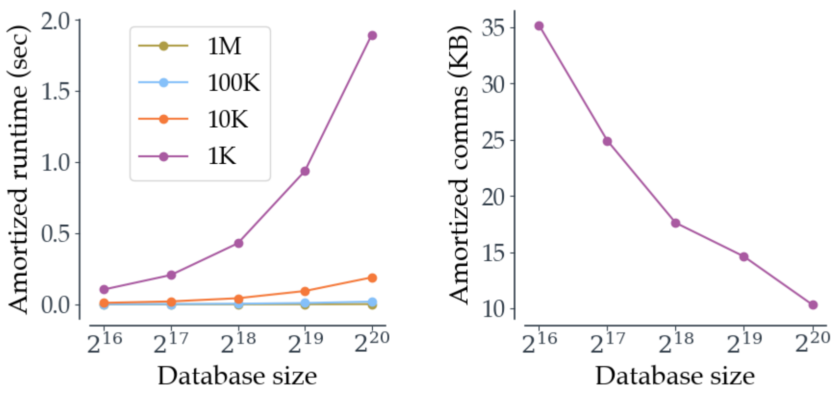

Fig. 8. Amortized (per-query) cost of server preprocessing (**left**) and client offline download size (**right**).

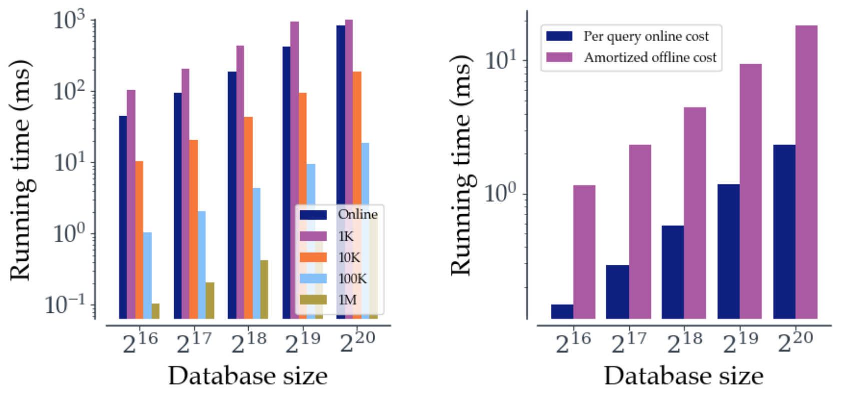

Fig. 9. Total online and amortized (per-query) offline computation costs for the server (**left**) and client (**right**).

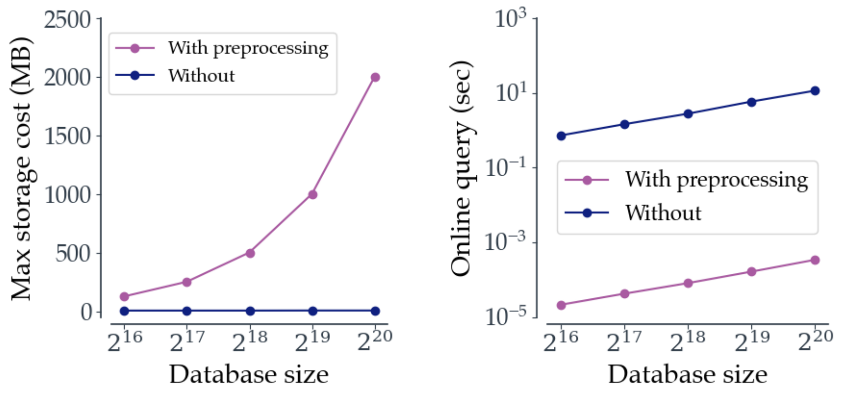

Fig. 10. Left: Storage costs for clients demonstrating the trade-off between amortization of offline preprocessing and ensuring logarithmic storage overhead relative to m. **Right:** Comparison of online query costs when preprocessing, against performing all query-related computation in the online phase.

As mentioned in Section 5.3, it is possible to achieve $\log(m)$ storage overhead on the client-side, which may be valuable for storage-constrained clients. The downside of this approach is that client online query processing grows noticeably, as seen in the right-side of Figure 10. This is due to having to perform all query-related processing in the online phase, including the derivation of $\boldsymbol{A}$ from the public parameters (which can take from between 0.5 to 8.4 seconds, depending on the database size).[10]

Financial costs. The server-side financial costs given in Figure 11 take into account the expenses associated with both bandwidth and single-threaded computation. The initial preprocessing of the server database, and is a little higher than 1 cent for a database of 2^{20}. The online per-query cost is tiny in comparison, and approximately 0.001 of a cent even for the largest DB size. The total per-query cost is calculated as the amortized offline costs, plus the online per-query cost.

6.2 Comparison with Prior Work

In Figure 12, we compare the performance of FrodoPIR with SOnionPIR [55] and PSIR [59], across three performance criteria: computational runtimes, bandwidth usage, and financial cost. As mentioned in Section 2, stateless schemes are much less efficient than stateful schemes, so we do not provide any experimental comparisons with them.

[10] The matrix $\boldsymbol{A}$ must be rederived on usage to achieve $\log(m)$ storage.

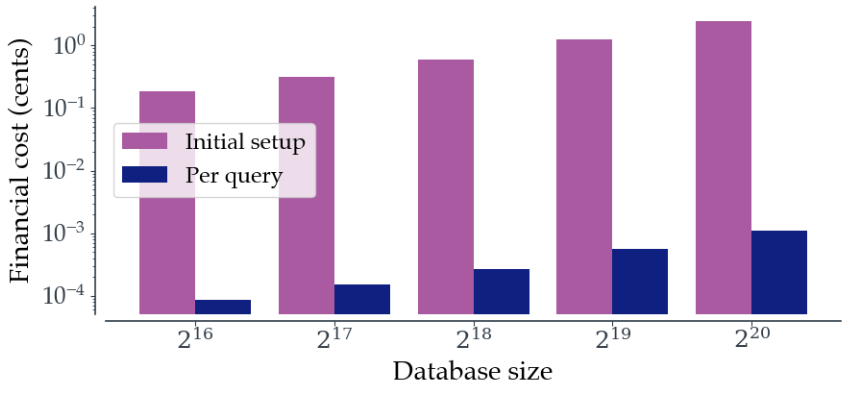

Fig. 11. Financial costs (cents) associated with running the server in FrodoPIR. The initial setup cost can be amortized globally across all client queries.

The comparisons that we present include the cost of answering queries in FrodoPIR against the estimated[11] costs of running both SOnionPIR and PSIR.[12] Note that the costs presented in [55] result from estimating SOnionPIR and PSIR on the same EC2 hardware (`t2.2xlarge`) that we used for implementing FrodoPIR. We also provide details on how these costs amortize as the number of clients grows.

Our comparison considers total database sizes of $|\mathsf{DB}| \in \{2^{16}, 2^{18}, 2^{20}\}$, and element sizes of 1KB. Note that SOnionPIR and PSIR allow packing of 30KB and 3KB of data into each server response [55]. This effectively allows shrinking the server DB by a factor of $30\times$ and $3\times$, respectively, in kind. Since such costs are linear in the size of DB, we reduce the previously estimated runtime costs of both schemes accordingly. Offline costs for SOnionPIR are dependent on the number of queries, c, that are made by each client. For each DB size we set $c = 500$, the same value as used in [55]. For the financial costs, we provide costs per CPU hour of server-side computation. The comparison does not cover storage costs or client computation as neither measurement is explicitly provided by the previous schemes.

Supporting larger databases. Note that [55] provides estimated costs of the SOnionPIR and PSIR schemes for a DB of size 2^{24}, but RAM overheads of FrodoPIR mean that the `t2.2xlarge` EC2 instance is not powerful enough to process a database of this size. This is also likely to be the case for the previous schemes. Building an efficient implementation of FrodoPIR for a database of 2^{24} is possible by sharding, using 16 DB instances with 2^{20} elements. In the interest of maintaining a fair comparison without using parallelization, we do not modify

[11] Neither previous scheme has been fully implemented.

[12] Where PSIR uses SealPIR as the underlying PIR scheme.

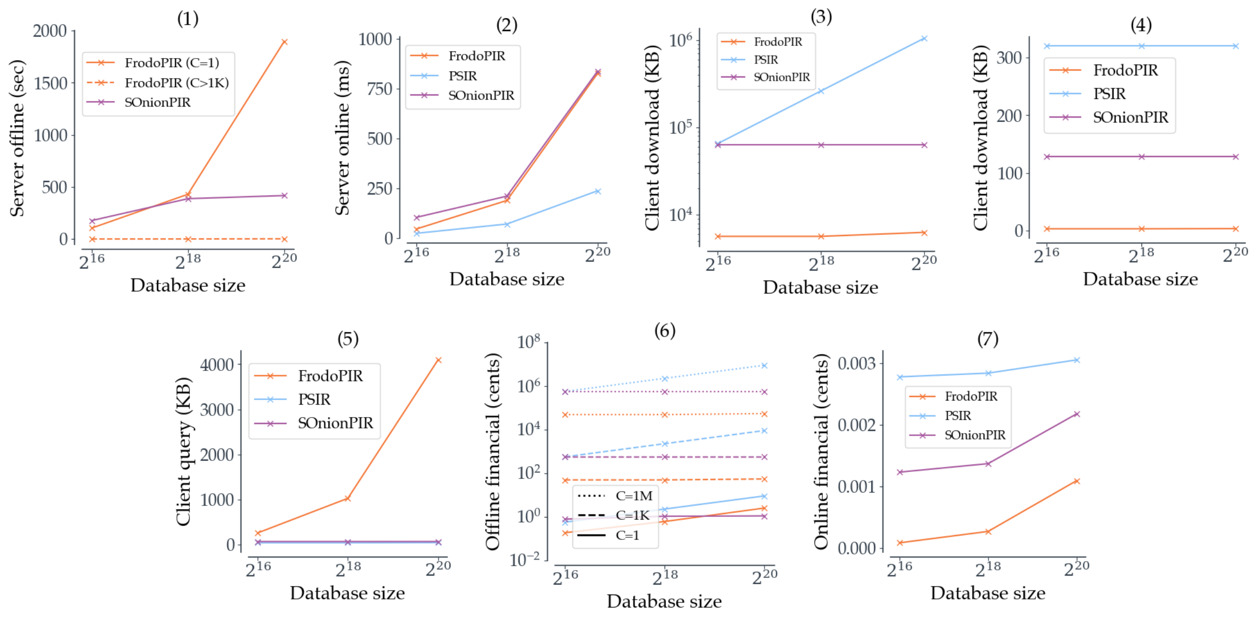

Fig. 12. Comparison of per-client computational, communication, and financial costs for the server when running FrodoPIR, SOnionPIR, and PSIR. We assume that each client makes $c = 500$ queries. We include amortized costs according to various numbers of clients C, to indicate the global amortization potential of FrodoPIR. Individual charts: **(1)** Server offline computation (secs) including amortization potential over C for FrodoPIR; **(2):** Server online computation (ms), amortized according to number of DB entries returned; **(3):** Client offline download (KB); **(4):** Client online download (KB); **(5):** Client online query (KB); **(6):** Server offline financial cost (US cents per CPU hour), compared for different values of C; **(7):** Server online financial cost (US cents per CPU hour).

the hardware used or make use of sharding. Thus, we limit the comparison to database sizes $\leq 2^{20}$.

Security levels. We do not modify the security parameters of either SOnionPIR or PSIR: they both offer ≤ 115 bits of security according to [2]. In contrast, FrodoPIR offers 128-bit security for up to 1 billion client queries and higher security levels for lower numbers. SOnionPIR and PSIR could achieve higher security levels by doubling n,[13] but while computation times would go unchanged, the server online response size would increase dramatically.

Computation. In the offline phase (Figure 12 (1)), the server-side computation for PSIR is zero, since the client simply downloads the entire server DB. The overall cost of computation in FrodoPIR grows linearly in the database size. While

[13] Smaller n would suffice, but n has to be a power-of-two to ensure the efficiency of NTT-related optimizations.

SOnionPIR appears to outperform FrodoPIR for a single client, this cost increases linearly in the number of queries that a client wishes to make. As a consequence, if the number of queries per-client (c) increases, then the cost of SOnionPIR will quickly become greater. More importantly, as the number of clients (C) in the system grows, this cost will continue to increase. In contrast, all preprocessing in FrodoPIR is client-independent, and thus it is fixed regardless of both c and C. Therefore, in a large multi-client deployment, it is clear that FrodoPIR is much cheaper than SOnionPIR.

In the online phase (Figure 12 (2)), PSIR provides the fastest computation times. Both FrodoPIR and SOnionPIR still provide competitive runtimes. FrodoPIR requires ≤ 0.8s for responding to a client query on a DB with 2^{20} elements.

Communication. The offline client download cost (Figure 12 (3)) in SOnionPIR is heavily dependent on the number of queries that will be launched. The cost of PSIR is essentially the cost of downloading the entire server DB. Note that the client download in FrodoPIR grows logarithmically in the size of DB. Overall, since the costs of FrodoPIR amortize across the number of queries launched by the clients, with a much smaller initial cost than PSIR, it is clear that FrodoPIR outperforms the alternatives.

In the online phase, the client download (Figure 12 (4)) in FrodoPIR is smallest for all captured DB sizes. The server response growth rate, even for $|\mathsf{DB}| = 2^{20}$, is $< 3.6\times$, which is significantly smaller than that of SOnionPIR ($128\times$) and PSIR ($320\times$). The major downside of the FrodoPIR approach is that the client query in the online phase (Figure 12 (5)) grows linearly in the size of DB, and is much larger than both SOnionPIR and PSIR — reaching 4MB for client queries when $|\mathsf{DB}| = 2^{20}$. As noted in Section 5.4, this cost can be reduced using database sharding with the additional benefit of reducing server computation times, but at the cost of increasing client download sizes during the offline phase.

Financial costs. In the offline phase (Figure 12 (6)), PSIR provides by far the most expensive option, due to the high network bandwidth usage. The costs of SOnionPIR scale with the number of client queries. The costs of FrodoPIR include a client-independent preprocessing phase, and much lower bandwidth usage than PSIR. Therefore, for large multi-client deployments, the costs of FrodoPIR will clearly be much cheaper than both prior schemes.

The online financial costs (Figure 12 (7)) for all protocols are much smaller than in the offline phase. By far, PSIR is the most expensive protocol to run in the online phase (again, due to the high communication overhead). The costs of FrodoPIR outperform SOnionPIR, demonstrating that the trade-off between computation and communication in FrodoPIR is concretely cheaper to realize on the server-side.

Comparison with online-free PIR. The CHKPIR scheme [27] demonstrates entirely sublinear (amortized) running times and communication costs. However, this depends on each client launching a fairly large number of queries themselves (e.g. $\sqrt{m}$). As is noted in [27], the offline phase can be implemented using

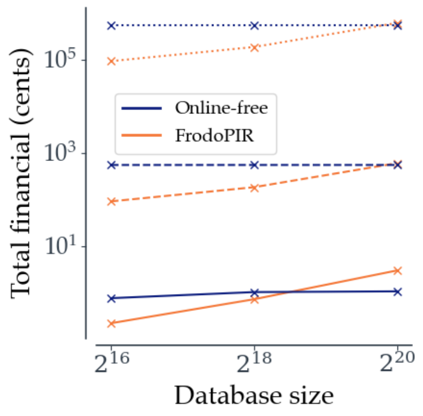

Fig. 13. Comparison of the total financial costs of running a server using FrodoPIR with an online-free PIR scheme that implements the offline phase using SOnionPIR. The costs are compared for: $C = 1$ (solid line), $C = 1000$ (dashed line), and $C = 1$ million (dotted line) clients; where each client makes $c = 500$ queries.

the methods of PSIR or SOnionPIR, and, regardless, it is still non-amortizable across multiple clients.

To illustrate the bottleneck that the offline phase introduces from a financial perspective, we can consider a PIR scheme that has *zero* online costs (which is clearly a significant underestimate for CHKPIR), and has the offline costs of SOnionPIR (sublinear in m and generally lower than PSIR). As shown in Figure 13, FrodoPIR is cheaper to run for databases of size $\leq 2^{18}$, for $c \cdot C$ queries. The costs are almost identical when $|\mathsf{DB}| = 2^{20}$. We can conclude that these results, coupled with the benefits of a simple and available implementation, make FrodoPIR a very attractive option for implementing fast and scalable PIR for large multi-client systems.

7 Discussion

Supporting Keyword Queries. In the interest of supporting more realistic database queries, Chor et al. constructed a PIR-by-keyword framework, where the server DB is a key-value store and client queries are keywords that recover the associated values [25]. Their framework runs multiple instances of index-based PIR as a black-box; FrodoPIR is compatible with this approach. The work of Boyle et al. [16], based upon multi-server distributed point functions, includes direct support for keyword queries directly, but it does not appear to generalize to other PIR schemes.

As well as generic frameworks, FrodoPIR is compatible with external mechanisms for deciding keyword-to-index association. Such mechanisms include the approach detailed by Kogan and Corrigan-Gibbs [48], that furnishes the client with $O(m)$ hash prefixes of each keyword, and associates each with a server DB index. This allows the client to learn the index that they need to query, without running multiple instances of the PIR scheme. It requires sending $O(m)$ data to the client but which, in practice, is a very small fraction of the real database. We discuss practical costs in Appendix B.

Optimizations for server computation. We avoided discussing computational optimizations in this work in favor of building an efficient and usable PIR scheme that non-expert implementers can configure to their particular use-case. However, to decrease runtimes, the work of Beimel et al. [12] demonstrates mechanisms for achieving subcubic overheads for matrix multiplications, and has been used in previous PIR scheme's design to reduce computational workloads [51, 38]. The server offline phase in FrodoPIR involves a large matrix multiplication with dimensions $n \times m$ and $m \times \omega$, which would clearly benefit from such an optimization. The client offline phase, involves preprocessing c queries, each involving a vector-matrix multiplication, which could be batched together into a single matrix multiplication. Furthermore, the server online phase involves a vector-matrix multiplication, for every client query. This optimization can be used by batching a number of queries together. As is observed by Lueks and Goldberg [51], this enables the server's work to scale sublinearly in the number of client queries.

7.1 Other applications

In Appendix B, we illustrate how FrodoPIR can be applied to real-world use-cases like the SafeBrowsing API [41]. We list various applications that could also benefit below. Valuable future work would identify whether FrodoPIR is a practical candidate for solving them.

Certificate auditing. Certificate Transparency (CT) is a system created to increase visibility of issued certificates. This system allows detection of misissued certificates or other forms of Certificate Authorities (CA) misbehavior, via interaction with one or more public logs. Clients should check that certificates are indeed included in these logs, but this leads to a potential privacy violation as it means that, over time, the client presents the browsing history of the user. One can apply FrodoPIR to check whether the promise of inclusion is fulfilled. A full discussion of this can be found in [29].

Certificate revocation checks. Certificate revocation checks typically use the Online Certificate Status Protocol (OCSP). This mechanism allows CAs to inform clients if a certificate is revoked by having them query an endpoint. This mechanism, however, can violate privacy as the certificates are revealed to the CA. An alternative is to have clients download certificate revocation lists (CRLs) from endpoints maintained by CAs. This, though, comes with a huge storage overhead and the need for regular updates. FrodoPIR could be used to perform OCSP queries in a privacy-preserving manner.

PIR for streaming. Previous PIR schemes such as Popcorn [43] use PIR for streaming use-cases, where clients can gradually consume chunks of a data element (as in video streaming applications) while hiding what is consumed. The capability of FrodoPIR for sharding the server database (Section 5.4) could make it a viable candidate in this setting.

8 Conclusion

In this work, we built FrodoPIR. Via a simple proof-of-concept Rust implementation,[14] we illustrated via experimental analysis that FrodoPIR is concretely cheaper than the previous state-of-the-art in stateful PIR schemes, especially in large multi-client deployments. Overall, we believe that FrodoPIR is the first single-server PIR scheme that is both flexible and efficient enough to be deployed at scale, for a variety of applications.

Acknowledgements

The authors would like to thank Benjamin Livshits, Hamed Haddadi, Joe Rowell, Muhammad Haris Mughees, Martin Albrecht, and Ling Ren for providing helpful feedback during various stages of this work.

References

1. C. Aguilar Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, Apr. 2016.
2. M. R. Albrecht, R. Player, and S. Scott. On the concrete hardness of learning with errors. *J. Math. Cryptol.*, 9(3):169–203, 2015. (Estimates calculated Dec 2021).
3. A. Ali, T. Lepoint, S. Patel, M. Raykova, P. Schoppmann, K. Seth, and K. Yeo. Communication–Computation trade-offs in PIR. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1811–1828. USENIX Association, Aug. 2021.
4. A. Ambainis. Upper bound on communication complexity of private information retrieval. In P. Degano, R. Gorrieri, and A. Marchetti-Spaccamela, editors, *ICALP 97*, volume 1256 of *LNCS*, pages 401–407. Springer, Heidelberg, July 1997.
5. S. Angel, H. Chen, K. Laine, and S. T. V. Setty. PIR with compressed queries and amortized query processing. In *2018 IEEE Symposium on Security and Privacy*, pages 962–979. IEEE Computer Society Press, May 2018.
6. S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 551–569, USA, 2016. USENIX Association.
7. S. Angel and S. Setty. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 551–569, Savannah, GA, Nov. 2016. USENIX Association.
8. R. Beigel, L. Fortnow, and W. Gasarch. A tight lower bound for restricted pir protocols. *Computational Complexity*, 15:82–91, 05 2006.
9. A. Beimel and Y. Ishai. Information-theoretic private information retrieval: A unified construction. In F. Orejas, P. G. Spirakis, and J. van Leeuwen, editors, *ICALP 2001*, volume 2076 of *LNCS*, pages 912–926. Springer, Heidelberg, July 2001.

[14] https://github.com/brave-experiments/frodo-pir

10. A. Beimel, Y. Ishai, E. Kushilevitz, and I. Orlov. Share conversion and private information retrieval. In *2012 IEEE 27th Conference on Computational Complexity*, pages 258–268, 2012.
11. A. Beimel, Y. Ishai, E. Kushilevitz, and J.-F. Raymond. Breaking the $O(n^{1/(2k-1)})$ barrier for information-theoretic private information retrieval. In *43rd FOCS*, pages 261–270. IEEE Computer Society Press, Nov. 2002.
12. A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In M. Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 55–73. Springer, Heidelberg, Aug. 2000.
13. S. Bell and P. Komisarczuk. An analysis of phishing blacklists: Google safe browsing, openphish, and phishtank. In *Proceedings of the Australasian Computer Science Week Multiconference*, ACSW '20, New York, NY, USA, 2020. Association for Computing Machinery.
14. N. Borisov, G. Danezis, and I. Goldberg. DP5: A private presence service. *PoPETs*, 2015(2):4–24, Apr. 2015.
15. J. W. Bos, C. Costello, L. Ducas, I. Mironov, M. Naehrig, V. Nikolaenko, A. Raghunathan, and D. Stebila. Frodo: Take off the ring! Practical, quantum-secure key exchange from LWE. In Weippl et al. [66], pages 1006–1018.
16. E. Boyle, N. Gilboa, and Y. Ishai. Function secret sharing: Improvements and extensions. In Weippl et al. [66], pages 1292–1303.
17. E. Boyle, Y. Ishai, R. Pass, and M. Wootters. Can we access a database both locally and privately? In Kalai and Reyzin [47], pages 662–693.
18. Z. Brakerski, A. Langlois, C. Peikert, O. Regev, and D. Stehlé. Classical hardness of learning with errors. In D. Boneh, T. Roughgarden, and J. Feigenbaum, editors, *45th ACM STOC*, pages 575–584. ACM Press, June 2013.
19. C. Cachin, S. Micali, and M. Stadler. Computationally private information retrieval with polylogarithmic communication. In J. Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 402–414. Springer, Heidelberg, May 1999.
20. L. Caires, G. F. Italiano, L. Monteiro, C. Palamidessi, and M. Yung, editors. *ICALP 2005*, volume 3580 of *LNCS*. Springer, Heidelberg, July 2005.
21. R. Canetti, J. Holmgren, and S. Richelson. Towards doubly efficient private information retrieval. In Kalai and Reyzin [47], pages 694–726.
22. Y.-C. Chang. Single database private information retrieval with logarithmic communication. In H. Wang, J. Pieprzyk, and V. Varadharajan, editors, *ACISP 04*, volume 3108 of *LNCS*, pages 50–61. Springer, Heidelberg, July 2004.
23. H. Chen, I. Chillotti, and L. Ren. Onion ring ORAM: Efficient constant bandwidth oblivious RAM from (leveled) TFHE. In L. Cavallaro, J. Kinder, X. Wang, and J. Katz, editors, *ACM CCS 2019*, pages 345–360. ACM Press, Nov. 2019.
24. B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *29th ACM STOC*, pages 304–313. ACM Press, May 1997.
25. B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998. `https://eprint.iacr.org/1998/003`.
26. B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th FOCS*, pages 41–50. IEEE Computer Society Press, Oct. 1995.
27. H. Corrigan-Gibbs, A. Henzinger, and D. Kogan. Single-server private information retrieval with sublinear amortized time. In O. Dunkelman and S. Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022*, pages 3–33, Cham, 2022. Springer International Publishing.

28. H. Corrigan-Gibbs and D. Kogan. Private information retrieval with sublinear online time. In A. Canteaut and Y. Ishai, editors, *EUROCRYPT 2020, Part I*, volume 12105 of *LNCS*, pages 44–75. Springer, Heidelberg, May 2020.
29. J. DeBlasio. Opt-out SCT Auditing in Chrome. URL: `https://docs.google.com/document/d/16G-Q7iN3kB46GSW5bsfH5MO3nKSYyEb77YsM7TMZGE`. Accessed May. 21, 2022.
30. S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In E. Kushilevitz and T. Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174. Springer, Heidelberg, Jan. 2016.
31. C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In M. Kutylowski and J. Vaidya, editors, *ESORICS 2014, Part I*, volume 8712 of *LNCS*, pages 380–399. Springer, Heidelberg, Sept. 2014.
32. Z. Dvir and S. Gopi. 2-server PIR with subpolynomial communication. *J. ACM*, 63(4):39:1–39:15, 2016.
33. K. Efremenko. 3-query locally decodable codes of subexponential length. *SIAM J. Comput.*, 41(6):1694–1703, 2012.
34. E. Fung, G. Kellaris, and D. Papadias. Combining differential privacy and PIR for efficient strong location privacy. In C. Claramunt, M. Schneider, R. C. Wong, L. Xiong, W. Loh, C. Shahabi, and K. Li, editors, *Advances in Spatial and Temporal Databases - 14th International Symposium, SSTD 2015, Hong Kong, China, August 26-28, 2015. Proceedings*, volume 9239 of *Lecture Notes in Computer Science*, pages 295–312. Springer, 2015.
35. C. Gentry and Z. Ramzan. Single-database private information retrieval with constant communication rate. In Caires et al. [20], pages 803–815.
36. T. Gerbet, A. Kumar, and C. Lauradoux. A privacy analysis of google and yandex safe browsing. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 347–358, 2016.
37. N. Gilboa and Y. Ishai. Distributed point functions and their applications. In P. Q. Nguyen and E. Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 640–658. Springer, Heidelberg, May 2014.
38. I. Goldberg. Improving the robustness of private information retrieval. In *2007 IEEE Symposium on Security and Privacy*, pages 131–148. IEEE Computer Society Press, May 2007.
39. O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In A. Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.
40. O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
41. Google. SafeBrowsing API. URL: `https://safebrowsing.google.com/`. Accessed Jan. 25, 2022.
42. M. Green, W. Ladd, and I. Miers. A protocol for privately reporting ad impressions at scale. In Weippl et al. [66], pages 1591–1601.
43. T. Gupta, N. Crooks, W. Mulhern, S. Setty, L. Alvisi, and M. Walfish. Scalable and private media consumption with popcorn. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation*, NSDI'16, page 91–107, USA, 2016. USENIX Association.
44. A. Hamlin, R. Ostrovsky, M. Weiss, and D. Wichs. Private anonymous data access. In Y. Ishai and V. Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 244–273. Springer, Heidelberg, May 2019.

45. R. Henry. Polynomial batch codes for efficient IT-PIR. *PoPETs*, 2016(4):202–218, Oct. 2016.
46. Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In L. Babai, editor, *36th ACM STOC*, pages 262–271. ACM Press, June 2004.
47. Y. Kalai and L. Reyzin, editors. *TCC 2017, Part II*, volume 10678 of *LNCS*. Springer, Heidelberg, Nov. 2017.
48. D. Kogan and H. Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, Aug. 2021.
49. E. Kushilevitz and R. Ostrovsky. Replication is NOT needed: SINGLE database, computationally-private information retrieval. In *38th FOCS*, pages 364–373. IEEE Computer Society Press, Oct. 1997.
50. H. Lipmaa. An oblivious transfer protocol with log-squared communication. In J. Zhou, J. Lopez, R. H. Deng, and F. Bao, editors, *ISC 2005*, volume 3650 of *LNCS*, pages 314–328. Springer, Heidelberg, Sept. 2005.
51. W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In R. Böhme and T. Okamoto, editors, *FC 2015*, volume 8975 of *LNCS*, pages 168–186. Springer, Heidelberg, Jan. 2015.
52. Y. Ma, K. Zhong, T. Rabin, and S. Angel. Incremental offline/online PIR (extended version). *IACR Cryptol. ePrint Arch.*, page 1438, 2021.
53. S. J. Menon and D. J. Wu. SPIRAL: fast, high-rate single-server PIR via FHE composition. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 930–947. IEEE, 2022.
54. P. Mittal, F. G. Olumofin, C. Troncoso, N. Borisov, and I. Goldberg. PIR-tor: Scalable anonymous communication using private information retrieval. In *USENIX Security 2011*. USENIX Association, Aug. 2011.
55. M. H. Mughees, H. Chen, and L. 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.
56. M. H. Mughees, G. Pestana, A. Davidson, and B. Livshits. Privatefetch: Scalable catalog delivery in privacy-preserving advertising. *CoRR*, abs/2109.08189, 2021.
57. T. C. E. of Statistics. *Central Limit Theorem*, pages 66–68. Springer New York, New York, NY, 2008. DOI: https://doi.org/10.1007/978-0-387-32833-1_50.
58. J. Park and M. Tibouchi. SHECS-PIR: Somewhat homomorphic encryption-based compact and scalable private information retrieval. In L. Chen, N. Li, K. Liang, and S. A. Schneider, editors, *ESORICS 2020, Part II*, volume 12309 of *LNCS*, pages 86–106. Springer, Heidelberg, Sept. 2020.
59. S. Patel, G. Persiano, and K. Yeo. Private stateful information retrieval. In D. Lie, M. Mannan, M. Backes, and X. Wang, editors, *ACM CCS 2018*, pages 1002–1019. ACM Press, Oct. 2018.
60. O. Regev. On lattices, learning with errors, random linear codes, and cryptography. In H. N. Gabow and R. Fagin, editors, *37th ACM STOC*, pages 84–93. ACM Press, May 2005.
61. L. Ren, C. W. Fletcher, A. Kwon, E. Stefanov, E. Shi, M. van Dijk, and S. Devadas. Constants count: Practical improvements to oblivious RAM. In J. Jung and T. Holz, editors, *USENIX Security 2015*, pages 415–430. USENIX Association, Aug. 2015.

62. S. Servan-Schreiber, K. Hogan, and S. Devadas. Adveil: A private targeted-advertising ecosystem. Cryptology ePrint Archive, Report 2021/1032, 2021. `https://ia.cr/2021/1032`.
63. R. Sion and B. Carbunar. On the practicality of private information retrieval. In *NDSS 2007*. The Internet Society, Feb. / Mar. 2007.
64. E. Stefanov, M. van Dijk, E. Shi, T. H. Chan, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. *J. ACM*, 65(4):18:1–18:26, 2018.
65. S. Wehner and R. Wolf. Improved lower bounds for locally decodable codes and private information retrieval. In Caires et al. [20], pages 1424–1436.
66. E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors. *ACM CCS 2016*. ACM Press, Oct. 2016.
67. S. Yekhanin. Towards 3-query locally decodable codes of subexponential length. *J. ACM*, 55(1):1:1–1:16, 2008.

A Proofs from Section 4

A.1 Proof of Theorem 1

Let $\widetilde{\boldsymbol{b}} \leftarrow \boldsymbol{b} + \boldsymbol{f}_i$, where i is the requested index of DB by the client. As laid out in Section 4.4, we must show that Equation (6) holds, with all but negligible probability. Firstly, note that since $\boldsymbol{e} \leftarrow (\chi)^m$, then by Corollary 3, we have that $\|\boldsymbol{e} \cdot \boldsymbol{D}\|_\infty \leq 4\rho\sqrt{m}$ with high probability. This follows because the number of samples m is very large and by assuming that each entry in $\boldsymbol{D}$ is equal to $\rho = \|\boldsymbol{D}\|_\infty$. Consequently:

$$\begin{aligned} \lfloor (\boldsymbol{e} + \boldsymbol{f}_i)^T \cdot \boldsymbol{D} \rceil_\rho &= \lfloor \rho/q \cdot (\boldsymbol{e}^T \cdot \boldsymbol{D} + \boldsymbol{f}_i{}^T \cdot \boldsymbol{D}) \rceil \\ &= \lfloor \rho/q \cdot (\boldsymbol{e}^T \cdot \boldsymbol{D}) + \boldsymbol{D}[i] \rceil \\ &= \lfloor \boldsymbol{y} + \boldsymbol{D}[i] \rceil, \end{aligned} \tag{9}$$

where $\boldsymbol{y} = \rho/q \cdot (\boldsymbol{e}^T \cdot \boldsymbol{D})$ and $\boldsymbol{D}[i] \in \mathbb{Z}_q^\omega$ is the i^{th} row of $\boldsymbol{D}$ (interpreted as a column vector). Therefore, $\|\boldsymbol{y}\|_\infty < 4\rho^2\sqrt{m}/q = 1/2$ by the statement of the theorem and, as a consequence:

$$\lfloor (\boldsymbol{e} + \boldsymbol{f}_i)^T \cdot \boldsymbol{D} \rceil_\rho = \boldsymbol{D}[i], \tag{10}$$

which is the correct output of the protocol. □

A.2 Proof of Theorem 2

Let $\mathsf{ip} \leftarrow \mathsf{FPIR.ssetup}(1^\lambda)$, $\mathsf{pp} \leftarrow \mathsf{FPIR.spreproc}(\mathsf{DB})$, $\mathsf{st} \leftarrow \mathsf{FPIR.cpreproc}(\mathsf{pp})$, let $i_0, i_1 \leftarrow \mathcal{A}(\mathsf{ip}, \mathsf{pp})$, let $b \leftarrow\!\$\, \{0,1\}$, and let $\widetilde{\boldsymbol{b}}_b \leftarrow \mathsf{FPIR.query}(\mathsf{st}, i_b)$. In particular, we have that $\widetilde{\boldsymbol{b}}_b = \boldsymbol{s}^T \cdot \boldsymbol{A} + \boldsymbol{e}^T + \boldsymbol{f}_{i_b}{}^T \in \mathbb{Z}_q^m$, for $\boldsymbol{A} \in \mathsf{st}$, $\boldsymbol{s} \leftarrow (\chi)^n$, $\boldsymbol{e} \leftarrow (\chi)^m$, $\boldsymbol{A} \leftarrow \mathsf{PRG}(\mu, n, m, q)$, and $\boldsymbol{f}_{i_b}$ the m-dimensional vector of all zeroes except where $\boldsymbol{f}_{i_b}[i_b] = q/\rho$. Clearly, we can show that FPIR is secure if we can show that the output of $\mathsf{FPIR.query}$ is distributed uniformly.

Firstly, note that $\boldsymbol{A}$ is sampled as the output of a pseudorandom generator, therefore, it is indistinguishable from $\boldsymbol{A} \leftarrow\!\$\, \mathbb{Z}_q^{n\times m}$. Therefore, let $\mathcal{A}$ be an adversary in the $\mathsf{LWE}_{q,n,m,\chi}$ decisional security game (Definition 1), receiving $(\boldsymbol{A}, \boldsymbol{u})$ as the challenge, and let $\mathcal{S}$ be an adversary in the PIR 1-query indistinguishability game (Definition 4). When $\mathcal{A}$ receives the sample in Equation (1), $\boldsymbol{b}$ and $\widetilde{\boldsymbol{b}}$ are distributed identically, and when it receives the sample in Equation (2), then $\boldsymbol{b} \leftarrow\!\$\, \mathbb{Z}_q^m$. Therefore, the adversary $\mathcal{A}$ can simulate the client query to $\mathcal{S}$ by simply sending $\widetilde{\boldsymbol{b}} = \boldsymbol{u} + \boldsymbol{f}_{i_b}$ for $b \leftarrow\!\$\, \{0,1\}$. When $\mathcal{S}$ returns their guess $b' \in \{0,1\}$ to $\mathcal{A}$, $\mathcal{A}$ checks if $b' \overset{?}{=} b$.

Clearly, whatever advantage ϵ that $\mathcal{S}$ has in guessing the correct value of b, immediately translates to $\mathcal{A}$ having advantage ϵ in the decisional $\mathsf{LWE}_{q,n,m,\chi}$ security game. Since we assume that $\mathsf{LWE}_{q,n,m,\chi}$ is difficult to solve, we therefore conclude that $\epsilon \le \mathsf{negl}(\lambda)$.

To conclude, in the case that $\boldsymbol{b}$ is sampled uniformly, then the adversary has no advantage in distinguishing since $\widetilde{\boldsymbol{b}}$ is distributed uniformly. $\square$

A.3 Proof of Corollary 4

We can construct a matrix $\widetilde{\boldsymbol{B}}$ from each query $\widetilde{\boldsymbol{b}_j}$ $(j \in [\ell])$ that $\mathcal{S}$ observes with the following structure:

$$\begin{aligned}
\widetilde{\boldsymbol{B}} &= \left[\widetilde{\boldsymbol{b}_1} \,\middle\|\, \cdots \,\middle\|\, \widetilde{\boldsymbol{b}_\ell}\right] \\
&= \left[\left({\boldsymbol{s}_1}^T \cdot \boldsymbol{A} + {\boldsymbol{e}_1}^T\right)^T + \boldsymbol{f}_{i_1} \,\middle\|\, \cdots \,\middle\|\, \left({\boldsymbol{s}_\ell}^T \cdot \boldsymbol{A} + {\boldsymbol{e}_\ell}^T\right)^T + \boldsymbol{f}_{i_\ell}\right] \\
&= \left([\boldsymbol{s}_1 \,\|\, \cdots \,\|\, \boldsymbol{s}_\ell]^T \cdot \boldsymbol{A} + [\boldsymbol{e}_1 \,\|\, \cdots \,\|\, \boldsymbol{e}_\ell]^T\right)^T + [\boldsymbol{f}_{i_1} \,\|\, \cdots \,\|\, \boldsymbol{f}_{i_\ell}] \\
&= \boldsymbol{S} \cdot \boldsymbol{A} + \boldsymbol{E} + \boldsymbol{F} \in \mathbb{Z}_q^{\ell \times m}.
\end{aligned} \tag{11}$$

Using the same proof strategy as in Theorem 2, we can use $\mathcal{A}$ as an adversary attempting to decide in the decisional $\mathsf{MatLWE}_{q,n,m,\chi,\ell}$ security game (Definition 2). This illustrates that $\mathcal{A}$ has advantage equal to that which $\mathcal{S}$ has in deciding the uniformity of $\widetilde{\boldsymbol{B}}$. Furthermore, by Corollary 2, we know that $\epsilon = \ell \cdot \delta$, where δ is the max advantage of winning in $\mathsf{LWE}_{q,n,m,\chi}$. Since $\ell = \mathsf{poly}(\lambda)$, then $\epsilon = \mathsf{poly}(\lambda) \cdot \mathsf{negl}(\lambda) = \mathsf{negl}(\lambda)$. $\square$

B Example Application: SafeBrowsing

Major browsers such as Google Chrome, Firefox, and Brave integrate a security service run by Google and known as the SafeBrowsing API [41]. SafeBrowsing allows browsers to verify if online resources and webpages that the user requests are "safe". If a resource has been flagged as "unsafe", the user is warned by the browser and asked to explicitly consent visiting the website that contains the unsafe resource. The SafeBrowsing service relies on a list of blocked resources maintained by Google, and it exposes an API that informs the browser if a resource is part of the blocked list. The downside of serving queries to the SafeBrowsing API remotely is that clients would effectively reveal their browsing patterns to Google. It is clear that it will be important to build mechanisms that preserve client privacy from third parties (like Google, in this case), while still being able to inform users if they are about to load malicious content.

B.1 Current SafeBrowsing Implementation

Local storage. In order to avoid calling the remote API for *every* resource, the entire SafeBrowsing blocklist could be shipped with each browser. Unfortunately, due to the size of the full blocklist (> 90MB), it is not optimal to send the full blocklist to the clients. Instead, the SafeBrowsing service ships to every browser a compressed and probabilistic data structure that contains an approximate view of the SafeBrowsing blocklist. This data structure allows performing probabilistic checks of inclusion, with non-negligible chances of *false-positives* occurring and no chance of *false-negatives*. Due to the rate of potential *false positives*, if an inclusion check returns that a resource is part of the data structure, the browser remains uncertain. In order to remove that uncertainty, the browser confirms if the resource is unsafe by calling the remote SafeBrowsing API. Thus, the browser only relies on the remote API call to SafeBrowsing services when the set inclusion against the local data structure returns a potential false positive. This mechanism reduces considerably the number of remote API calls at the expense of storing a compressed, space optimized data structure locally in the browser.

Specifically, the local blocklist is a set of 32-bit hashes of the resource URI, and the full SafeBrowsing blocklist consists of a key-value database mapping a 32-bit hash to a SHA256 hash of a blocked resource URI. The reduction and compression of the local blocklist results in storage and bandwidth savings of about $8\times$ compared to the full SafeBrowsing blocklist. We summarize the two distinct phases of SafeBrowsing checks in the following.

1. **Phase 1: Local check** First, the browser computes the 32-bit hash of the resource URI that has been requested, and checks if the 32-bit hash is part of the local storage. If the set inclusion operation returns 'false' (i.e. the hash of the resource does not exists in the local data structure), then the browser considers the resource safe and proceeds. If the set inclusion operation returns 'true' (i.e. the 32-bit resource hash is part of the local block list), the client proceeds to the next phase.
2. **Phase 2: Remote check** When Phase 1 identifies a *possibly* unsafe resource, the browser needs to confirm whether the resource is a false positive or not. To do so, it requests the full SHA256 hash of the resource's URI by querying the remote SafeBrowsing API for the 32-bit hash of the resource URI computed in Phase 1. If the full SHA256 hash returned by the remote SafeBrowsing API matches with the SHA256 of the resource URI, then the resource is part of the SafeBrowsing blocklist, and the browser considers the resource unsafe.

Privacy considerations. The remote SafeBrowsing resource check (Phase 2) requires the browser to explicitly include the 32-bit hash identifying the resource that is being checked for inclusion on the SafeBrowsing blocklist. As noted by [13, 36], this request leaks information about the browsing history of the user, as the SafeBrowsing API service is able to learn which content a particular user is interested in. Over time, this information can be used by the SafeBrowsing service provider to construct a behavior profiling of web users without their consent.

B.2 SafeBrowsing via FrodoPIR

FrodoPIR can be used to implement the remote SafeBrowsing API service, such that no leakage occurs during the remote SafeBrowsing API check. The intuition is that, once the index that must be queried is known to the client, the remote check can be performed via a PIR query to a remote FrodoPIR database, that stores all the SHA256 hashes of the unsafe URIs. Given the privacy guarantees of FrodoPIR, the client does not leak which resource ID is being queried.

Requirements. Based on the estimates provided by [48], the current SafeBrowsing blocklist contains about 3 million entries. The blocklist grows at a rate of $30,000$ new entries per week. Each of the values in the database consists of a SHA256 hash of the content URI.

Mapping URL hashes to query indices. As in [48], we will assume that local blocklist is augmented to include the index that must be queried in the online database. That is, when the client finds a match in their local blocklist, they use the corresponding index i that is included to make a query for element i in the remote server database.

Database configuration. As shown in Section 5, FrodoPIR provides a high degree of flexibility, allowing developers to choose which trade-offs to make when deploying an instance of the PIR database. We now suggest the following database configuration to implement FrodoPIR for SafeBrowsing:

– We choose $q = 2^{32}$ and $n = 1774$, which should be satisfactory for even the large number of clients using major Internet browsers that integrate with the SafeBrowsing API. According to [2], this provides 128-bit security for 2^{52} client queries. In other words, this allows 4 billion clients to each make 1 million queries, which should be more than enough.
– We require $w = 256$ bits for storing each URI hash in the server database.
– Let $\widetilde{m}$ be the total number of elements in the SafeBrowsing database. We require that $\widetilde{m} \geq 2^{21}$ to accommodate all the 3 million entries and subsequent updates [48]. However, we leverage sharding to break down the databases into smaller sub-databases, as explained in Section 5.4. Assuming that FrodoPIR is running on a machine with 16 cores, we can split the blocklist into $s = 16$ sub-databases, resulting in setting $m = 2^{18}$ per shard. This provides a database with total size 2^{22}, which is enough to store the entire blocklist.
– Given w, m and q, we set $\rho = 2^{10}$ so that the correctness guarantee from Theorem 1 holds true.

Offline	Client download (KB)	180
	Database preprocessing (s)	28.555
	Client derive params (s)	2.2281
	Client query preprocessing (s)	0.573
Online	Client query (KB)	1024
	Server response (KB)	0.1
	Client query (ms)	0.097
	Server response (ms)	5.223
	Client output (ms)	0.012

Fig. 14. Performance analysis of the FrodoPIR scheme when communicating with a single database shard, using the parameters defined in Section B.2.

– We calculate $\omega = m/\log(\rho) = 26$ as described in Section 5.
– The local blocklist that each client must download contains $32 \cdot (m+1)$ bits to include each 32-bit hash prefix plus the corresponding 32-bit index.

We leverage sharding in two different ways. On one hand, to decrease the size of the database by splitting it into sub-databases, allowing us to reduce the size m of each sub-database, and to optimize both user and server performance and bandwidth. In addition, sharding is used to implement a low-cost database update mechanism. Updates to the blocklist happen by adding elements to one sub-database only, in turn requiring clients to derive new parameters only for a single shard at every update, as explained in Section 5.4. This is possible in SafeBrowsing because DB updates are typically only additions, and thus deletion of old content in previous shards is rarely required [48].

B.3 Implementation and Raw Costs

We set up the experimental environment, and report results in Figure 14, corresponding to the raw costs of using the FrodoPIR scheme on the aforementioned parameters. We run all experiments as single-threaded processes on the same Amazon `t2.2xlarge` EC2 instance, with 8 CPU cores and 32GB of RAM, as was used in Section 6.

B.4 Performance Analysis

From Figure 14, we estimate the performance of instantiating the SafeBrowsing API for a single database shard using FrodoPIR, using the parameter set defined in Section B.2. Our extrapolations are based on the following set of usage model assumptions that are taken from the previous work of Kogan and Corrigan-Gibbs [48] on exploring usage of PIR for satisfying the demands of SafeBrowsing.

– On average, clients launch a query every 44 minutes. Assuming 12 hours of daily usage, this leads to approximately 16 queries per day.

- On average, the server database is updated every 94 minutes. This leads to around 16 DB updates per day, with a weekly addition of around $30,000$ records.
- The server is a collection of Z replicas that are distributed globally, that each independently possess and process queries on the same database. Any client query can be fulfilled by a single server.
- Client storage must be, at least, a constant factor smaller than the entire SafeBrowsing database size.

Database initialization and updates. The main server initializes the sub-database, public parameters, and local blocklist for each individual shard. Each of these remain static for a monthly period and are downloaded by each server replica. When the main server initializes, or rotates the matrix $\boldsymbol{A}$, it posts the public parameters $\mathsf{pp} = (\mu, M = \{\boldsymbol{M}_i = \boldsymbol{A} \cdot \boldsymbol{D}_i\}_{i \in [16]})$ and local blocklists to a public location that clients can access and download from. Note that $\boldsymbol{M}_i \in \mathbb{Z}_q^{m \times \omega}$ corresponds to the public parameters made available for each sub-database.

Based on our usage model, we will assume that there are 16 database updates made by the server, each containing 268 records. We assume that clients each download and process 8 updates per day. Each database update touches a single shard DB_i, and results in uploading a new value of $\boldsymbol{M}_i$.

Client processing. Client preprocessing amounts to preprocessing 16 queries per day, using the server provided parameters pp. After every update, the client needs to regenerate the remaining preprocessed state that is associated with the sub-database that was updated. Recall that the client stores:

$$X = (\boldsymbol{b}_j = \boldsymbol{s}^T \cdot \boldsymbol{A} + \boldsymbol{e}^T, C_j = \{\boldsymbol{c}_i = \boldsymbol{s}_j{}^T \cdot \boldsymbol{M}_i\}_{i \in [16]})_{j \in [16]}$$

for each of the $j \in [16]$ queries that the client will launch, and for each of the $i \in [16]$ database shards. The client must also store each $\boldsymbol{s}_j$ that it samples, for responding to server updates as well as the local blocklist.

Overall, at the start of each day, the client rederives $\boldsymbol{A} \leftarrow \mathsf{PRG}(\mu, n, m, q)$, and computes the set X. Every time that the client makes a remote query it removes a pair $(\boldsymbol{b}_j, C_j)$ from storage, and sends $\widetilde{\boldsymbol{b}_j} = \boldsymbol{b} + \boldsymbol{f}_\iota$ to the server, for query index ι computed during the local blocklist check. Whenever the server issues a database update for shard i, the client redownloads $\boldsymbol{M}_i$ and the local blocklist, and uses $\boldsymbol{s}_j$ to update $\boldsymbol{c}_i = \boldsymbol{s}_j{}^T \cdot \boldsymbol{M}_i \in C_j$, for each remaining j (i.e. unused preprocessed query data). According to Figure 14, we have the following (per-day) client computational costs.

- A single derivation of $\boldsymbol{A}$.
- preprocessing of 16 queries for each of the 16 shards.
- Updating of $2\sum_{a=1}^{7} a = 56$ queries per day.
- 16 individual online queries.

We ignore the cost of running queries on the 32-bit hashes in the local blocklist, since these are negligible by comparison. Furthermore, the per shard cost of updating preprocessed query data is almost zero. Therefore, we calculate the total CPU costs of each client to amount to $32.96+16\cdot 0.47+16*0.00025 = 40.48$ seconds per day.

Client download. The initial client download of public parameters is equal to $128 + 16 \cdot (n\omega \log(q)) = 23,615,616$ bits, which corresponds to around 2.82MB. The total size of the local blocklist is approximately $32 \cdot 3$million bits, which is equal to 11.44MB. The running download cost per-day is calculated as $16\omega \log(q) + 8n\omega \log(q)) + 32 * 268 = 11,829,632$ bits, which is roughly 1.41MB.

Client query. The client query is linear in the size of a single shard, which has a maximum of 2^{18} elements. Therefore, each query is around 1MB in size, based on the costs from Figure 7. As a consequence, this results in roughly 16MB of additional communication per-day.

Client storage. The client needs $\sim$ 1MB to store each preprocessed query, and each secret vector $\boldsymbol{s}_j$, for $j \in [16]$. In total, this represents about 16MB of required storage. Secondly, the client must store the local prefix table for the SafeBrowsing API which amounts to storing a further 11.44MB of data. Thirdly, the client stores the public parameters made available by the server, which totals 2.82MB. Overall, the maximum client storage overhead is $\sim$ 30.69MB, which is a $91.55/30.26 = 3.0\times$ saving compared with storing the original database. As the client makes queries, it deletes used preprocessed data, and so this storage overhead will decrease as the day progresses.

Server processing. The non-private SafeBrowsing API has an average latency of around 90ms per client query [48]. This is achieved using $Z = 143$ servers answering client queries. Note that a single FrodoPIR server can answer a single client query in $\sim$ 16ms (Figure 14). We assume that 1 billion queries are received uniformly in 90ms windows over a 44 minute period.[15] Therefore, in each 90ms window around 29334 client queries are received. Further, we assume that each server can answer 3 client queries in 90ms (including time taken to receive and respond to the client HTTP request). To achieve this, we would need at least 9778 individual servers each answering queries on the same FrodoPIR database for servicing 1 billion clients. Clearly, this is much more expensive than running the non-private version of SafeBrowsing, but such a number of servers is still within the realms of practicality, whilst preserving client privacy.

Comparison with [48]. The work of Kogan and Corrigan-Gibbs presents two PIR-based constructions for running the SafeBrowsing API, one based on PIR from distributed point functions (dpfPIR), and the other based on offline-online PIR (ooPIR). Both schemes require two non-colluding servers. We compare the performance of running the SafeBrowsing API using FrodoPIR against both dpfPIR and ooPIR in Figure 15.

Clearly, FrodoPIR involves heavier usage costs compared to all known solutions, either non-private or using multi-server PIR. As previously highlighted, a limitation of the FrodoPIR scheme is the client request size, which makes up a large proportion of the total communication (496MB per month, as opposed to 43.7MB of download). The client computation is also much heavier than in

[15] In other words, simulating 1 query from every client every 44 minutes.

Performance indicators	Non-private	dpfPIR	ooPIR	FrodoPIR
Servers per 1B users	143	9047	1348	9778*
Latency (ms)	90	122	91	90*
Client init (sec)	3.1	2.6	13.3	32.96*
Client running (sec/month)	0.5	0.8	8.0	1272.0*
Initial communication (MB)	5.0	5.0	10.3	2.82
Online communication (MB/month)	3.0	3.6	9.0	539.7
Max storage (MB)	4.5	4.5	26.1	30.69*

Fig. 15. Comparison of instantiating the SafeBrowsing API using either FrodoPIR, or via the two-server PIR schemes of [48]. Costs of FrodoPIR that are estimated are marked with asterisks.

multi-server PIR, due to the requirement for computing high-dimensional cryptographic operations when preprocessing queries.

Otherwise, our estimates suggest that FrodoPIR can provide adequate performance for operators where non-colluding PIR servers are impossible to set up. However, it is worth noting that the experimental analysis of [48] provides significantly more detail than we do here. Our goal is to give a broad understanding of the increased overheads of using FrodoPIR.