Call Me By My Name: Simple, Practical Private Information Retrieval for Keyword Queries

Sofía Celi
Brave Software
Portugal
cherenkov@riseup.net

Alex Davidson
Universidade NOVA de Lisboa & NOVA LINCS
Portugal
a.davidson@fct.unl.pt

ABSTRACT

We introduce ChalametPIR: a single-server Private Information Retrieval (PIR) scheme supporting fast, low-bandwidth *keyword* queries, with a conceptually very simple design. In particular, we develop a generic framework for converting PIR schemes for index queries over flat arrays (based on the Learning With Errors problem) into keyword PIR. This involves representing a key-value map using any probabilistic filter that permits reconstruction of elements from inclusion queries (e.g. Cuckoo filters). In particular, we make use of recently developed *Binary Fuse filters* to construct ChalametPIR, with minimal efficiency blow-up compared with state-of-the-art index-based schemes (all costs bounded by a factor of ≤ 1.08). Furthermore, we show that ChalametPIR achieves runtimes and financial costs that are factors of between 6×-11× and 3.75×-11.4× more efficient, respectively, than state-of-the-art keyword PIR approaches, for varying database configurations. Bandwidth costs are additionally reduced or remain competitive, depending on the configuration. Finally, we believe that our application of Binary Fuse filters in the cryptography setting may bring immediate independent value towards developing efficient variants of other related primitives that benefit from using such filters.

1 INTRODUCTION

Private Information Retrieval (PIR) schemes provide the ability to make private queries on public databases that are hosted by an untrusted (semi-honest) server(s). In the more plausible single-server setting (where there are no trust assumptions over multiple non-colluding servers [46]), the majority of approaches with tolerable costs (e.g. [26, 29, 42, 48, 52, 55, 76]) are limited to querying *indices* over flat arrays. However, this abstraction differs greatly from real-world instantiations of both structured and unstructured databases, as they are often indexed by *keys*.

For a key-value map KV, Chor, Gilboa, and Naor observed that, via a generic transformation, index-based PIR could be used to obtain *PIR-by-keywords* (henceforth KWPIR) [22]. In KWPIR, the client privately queries for a keyword k, and learns $\mathsf{x} = \mathsf{KV}[\mathsf{k}]$. This abstraction remains much simpler than what is expected of today's database systems and requires a logarithmic number of index-based PIR protocols to be run in the size of the database. In turn this last point means that significant running costs are expected, even when using the most recent practical PIR schemes, which typically require hundreds of kB of traffic, and close to a second of server's runtime. In response to these limitations, the last few years have seen the design of single-round constructions from heavily optimised fully-homomorphic encryption [6, 47, 51, 62], as well as approaches that use local client storage to map keyword queries to indices [44]. Even so, considering the most efficient keyword-based SparsePIR scheme of Patel, Seo, and Yeo [62], there is an order of magnitude in the performance deprecation between index- and keyword-based PIR schemes. In particular, where recent work demonstrates very *simple* constructions of PIR guaranteeing state-of-the-art performance, based directly on learning with errors (LWE) [29, 42, 48, 76], similar constructions do not exist in the KWPIR setting.

Our work. We construct KWPIR via a generic transformation that merges LWE-based PIR schemes and *key-value filters* into highly efficient keyword PIR schemes. Key-value filters can be built from well-known Cuckoo filters [35], for example, which map a set into a data structure that allows querying keys and reconstructing corresponding values, with configurable false-positive rates ϵ (see Section 3 for our full abstraction). However, while such techniques have been used in FHE-based PIR design [6, 47] previously, their efficiency appeared to be outperformed by alternative techniques [62]. In constrast, we show that coupling LWE-based PIR schemes with recent innovations in filter-design, known as Binary Fuse filters [40], produces a keyword PIR scheme with state-of-the-art performance across almost all performance metrics, and across a variety of database settings. Our concrete scheme ChalametPIR is built explicitly using this framework, using schemes such as SimplePIR [42] and FrodoPIR [29], while compatible with more recent LWE-based PIR schemes [48].

Regarding concrete performance, for maps containing 1M keys with associated 256 B values (1 GB in total), ChalametPIR based on FrodoPIR achieves online server runtimes of around 100 ms (on a 2021 Macbook) and response sizes of 4 kB. This represents a minimal performance blow-up (1.08×) compared to the original index-based FrodoPIR scheme, and is an order of magnitude more efficient than SparsePIR. By comparing financial costs for standard AWS EC2 infrastructure [7] for various DB settings, we show that the costs ChalametPIR are between 3.75× and 11.4× cheaper than SparsePIR.

Formal contributions. In this work, we achieve the following.

- A formalisation of probabilistic *key-value* filters for the PIR setting (Section 3). We further provide a concrete definition and parameterisation for using Binary Fuse filters [40] in generic cryptographic applications, to store large key-value maps with configurable false-positive rate (Section 4).

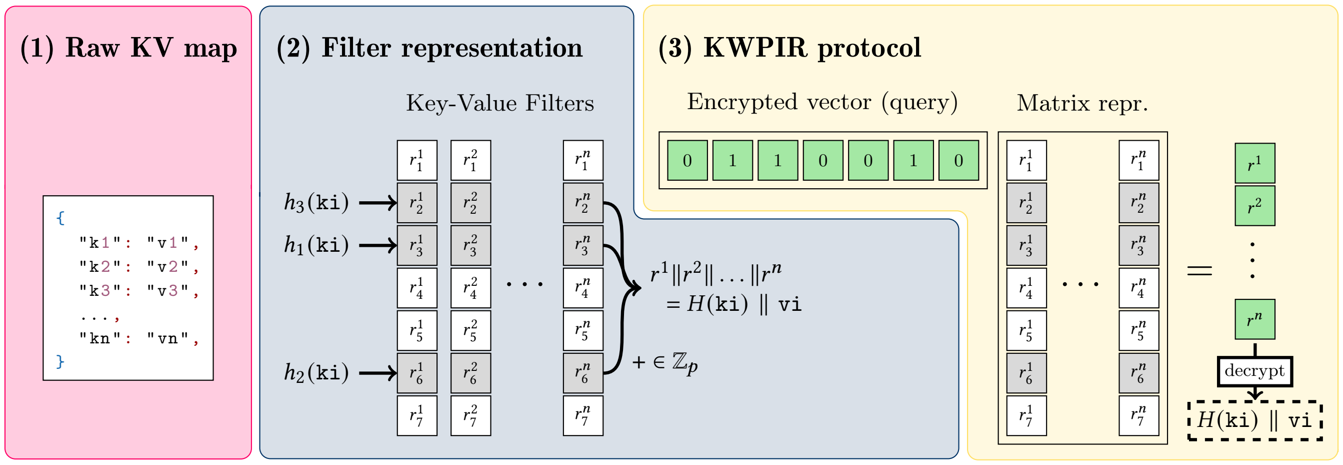

Figure 1: Overview of the generic framework used in ChalametPIR to construct KWPIR protocols over raw key-value maps. The framework uses key-value filters (Section 3.1) and LWE-based PIR (Section 2.4).

- A generic transformation that takes LWE-based PIR schemes and key-value filters, and produces conceptually very simple keyword PIR schemes (Section 5).
- An efficient parametrisation and open-source Rust implementation of the ChalametPIR scheme, based on FrodoPIR (but compatible with general LWEPIR schemes) and Binary Fuse filters.[1] Our experimental analysis shows that ChalametPIR achieves state-of-the-art performance costs (Section 6).

1.1 Technical Overview

Our approach is very simple, and forms of it have been used previously in PIR schemes (for example, see [6, 47, 62]). A high-level visualisation of the methodology is given in Figure 1. In principle, we make use of a key-value map KV of size m, that is indexed by keys $\mathsf{k} \in \mathcal{K}$ with corresponding values $\mathsf{x} \in \mathcal{X}$. We convert this map into a filter structure that permits reconstructing elements over $\mathcal{X}$ using a set of hash functions $\mathsf{H} = \{\mathsf{h}_i\}_{i \in [k]}$, with a configurable false-positive probability, ϵ. In other words, the filter F has a function of the form $\mathsf{fpt}_\epsilon(\mathsf{x}) \leftarrow \mathsf{F.check}(\mathsf{k})$, for some fingerprint function fpt_ϵ that allows deriving x. To avoid, storing huge data elements in each entry of the filter, we break the filter into d "columns", each holding $\log(p)$ bits of a given row of data, and indexed by the same set of hash functions. We interpret these filter columns as a matrix containing N rows, where $N = \varsigma m$ and ς is the natural blow-up introduced by the filter. We can query for an element by using linearly homomorphic encryption (Section 2.2). In principle, this query consists of sending an encrypted vector of all zeroes, except for entries corresponding to $\mathsf{h}_i(\mathsf{k})$ which are set to 1.

Previous work combined this general approach with FHE and Cuckoo hashing [56]. The results, however, have not been shown to be so efficient (notably performing worse than non-filter-based approaches [62]), or leveraged the use of multiple rounds in order to lower communication times via primitives as oblivious transfer [47]. The novelty in our work relies on the merging of non-FHE-based PIR (i.e. LWE-based), and coupling it with concrete and very practical instantiations of filters, such as the novel Binary Fuse filters [40]. Binary Fuse filters set $1.08 \leq \varsigma \leq 1.13$, dependent on the choice of $k \in \{3, 4\}$, which appears to be notably smaller than any other filter design. To the best of our knowledge, our work is the first to explore the usage of Binary Fuse filters in cryptography, as a practical basis for other cryptographic primitives that rely upon filter-based approaches.

Handling false-positives. PIR seems like a natural candidate for using the filter-based approach (one the relies on filter structures with a configurable false-positve probability) as the database is assumed to be *public* and the adversary is assumed to be *semi-honest*. This means that the fact that false-positives can occur does not necessarily lead to security flaws, but they may indeed have real-world impacts. To mitigate false-positives, we use the fact that the false-positives rates of Binary Fuse filters are defined by the length of the key fingerprint in the output, and therefore explicitly structure outputs as $H(\mathsf{k}) \| \mathsf{x}$, where $H : \mathsf{k} \mapsto \{0, 1\}^\mu$, where $\mu \geq 2\epsilon$ is a universal hash function. This means that any query for k' can identify a false-positive by simply checking that the first μ bits are equal to $H(\mathsf{k}')$, and otherwise aborting.

Security and performance. The security of our approach follows naturally from the LWE assumption, as parameterised by the underlying PIR scheme. Performance magnification compared with the corresponding index-based scheme is determined by the factor ς, and thus can be as small as 1.08. Furthermore, using LWE-based PIR schemes that use square-root matrix encodings (such as SimplePIR, see Section 5.1), reduces this magnification further still. In comparison with existing keyword PIR schemes, ChalametPIR (based on either FrodoPIR or SimplePIR) is significantly more efficient across all performance metrics, for almost all database settings (see

[1] https://github.com/claucece/chalamet

Section 6). We provide an open-source Rust implementation of ChalametPIR, based on the FrodoPIR scheme.

1.2 Related Work

Index-based PIR. Private Information Retrieval (PIR) is a decades-old problem first introduced by Chor, Gilboa, Kushilevitz, and Sudan [23], concerned with privately retrieving items from public databases, hosted by untrusted (semi-honest) servers. It is considered that there are two ways to approach this problem. The first, information-theoretic PIR, relies on having at least two non-colluding servers respond to client queries, which are shared across both servers [5, 9–12, 21, 27, 34, 38, 50, 73]. Corrigan-Gibbs and Kogan and follow-up works showed that such protocols can be configured to achieve sublinear asymptotic efficiency in the size of the database [27, 68]. The second approach is single-server PIR, where client queries are handled by a single untrusted server, and security follows from computational assumptions [1, 3, 6, 17, 19, 27, 37, 46, 49, 55, 59, 60]. Such schemes are generally less efficient (though sub-linear approaches exist [26, 76]), but have the advantage of avoiding onerous trust assumptions.

The most efficient single-server index-based PIR schemes are currently based on lattice-based cryptographic assumptions, particularly on the ring learning with errors (RLWE) assumption and FHE [3, 6, 32, 52, 55, 59, 60]. However, a recent line of work that builds PIR from the standard learning with errors (LWE) assumption appears to build PIR with both higher efficiency and a significant degree of simplicity [24, 29, 42, 48]. In particular, the implementations of such schemes require only hundreds of lines of code, and call for almost no optimisations, meaning that they could plausibly be implemented by non-expert developers. FHE-based solutions appear to only remain competitive assuming complex optimisations that seem to be far beyond the capabilities of a non-expert implementer. Various works, such as [60, 76], focus on providing very fast and efficient online PIR protocols, but require streaming the entire database to the client in an offline phase, which violates fundamental PIR efficiency criteria (Definition 2.5).

Keyword PIR. For decades, PIR schemes that could allow querying keywords (and thus maintaining much more realistic functionality, with respect to modern data structures) reduced the problem to running many rounds of index-based PIR. In particular, Chor et al. [22] showed keyword PIR could be achieved by running logarithmically number of rounds of index-based PIR over binary tree structures. Similar constructions based on oblivious PRFs have been given [36], with the advantage of achieving notions of privacy for the database.

More recent lines of work allows for single-round keyword-query functionality via FHE-based PIR. In particular, the work of [3] introduces a solution that leverages a form of cuckoo hashing to probabilistically map keywords into a small table — similar to our proposed approach. Alternatively, one can build keyword-based PIR via equality operators [6]. In this approach, the client's query is encoded into a domain, encrypted and sent to a server. The server computes each bit of the vector using an equality operator — represented as an indicator function that is set to 1 when it returns true, and 0 otherwise — between the client's encrypted query and each database identifier (which can be an index or keyword). Then, the server derives the inner product between the database and the vector, and sends the result to be decrypted by the client. Via a folklore equality operator, they construct a PIR scheme that has the smallest upload cost amongst all non-trivial approaches, but has high computational times due to the high multiplicative depth of equality circuits, as database elements grow. The work of [51] instead uses equality operators for *constant-weight* codewords, that have multiplicative depth that depend only on the Hamming weight of the code, and not on the bit-length of the element. This construction is 10× faster than "regular" equality operators and still facilitates keyword queries. However, computational and communicational times remain very high, when compared with state-of-the-art index-based PIR schemes. The "Checklist" scheme of [44] developed a keyword-based PIR approach based on multi index-based PIR, by having the client locally store a probabilistic mapping between keyword queries and their respective indices using hash prefixes. Unfortunately, their approach requires the client to store $2\epsilon|\mathsf{KV}|$ bits of data to achieve false-positives rates ϵ.

Finally, the work of [62] present a different direction by providing a framework that transforms the database as an encoding of linear combinations, directly utilising the capabilities of an underlying FHE-based PIR scheme. In particular, their approach can be applied to existing schemes such as [52, 55]. This approach results in performance that is an order of magnitude more efficient than [51], and is compatible with recursion [46] and batching [6] techniques.

Filters in cryptography. Bloom [13] and Cuckoo [56] Filters have a long history of applications in cryptography. In particular, such filter descriptions allow efficiently representing and querying sets via k hash function evaluations, with a configurable false-positive probability ϵ. Their application has resulted in various significant advances in achieving efficient designs of protocols for performing private set intersection [28, 33, 64, 65], PIR [3, 6, 47], encrypted search [61], and many others.

Filters in general have seen many advances since the pioneering work of Bloom. Since then, various forms of Bloom filters have been developed that optimise for space and query times [14, 16, 30, 53, 67]. While Bloom filters requires that $k = 1/epsilon$ in the optimal setting, Cuckoo filters [35] set $k = 2$ while still maintaining configurable ϵ. This approach stores larger number of bits (dependent on ϵ) per entry, which further permits entire reconstruction of elements (or fingerprints) from queries, on top of indicating whether elements belong to their set. Further optimisations in filter designs have been introduced since, including XOR filters [39], Ribbon filters [31], and Binary Fuse filters [40]. Binary Fuse filters in particular, provide a constant number of hash functions ($k \in \{\}3, 4$), and appear to represent the state-of-the-art in terms of filter size. We describe Binary Fuse filters in detail in Section 4.

2 PRELIMINARIES

2.1 Notation

We denote by $[n]$ the set $\{1, \ldots, n\}$. For all intents and purposes, we consider sets to be ordered (i.e. as arrays, indexed from 1 onwards) unless stated otherwise. We use $Q = \emptyset$ to denote the initialisation of an empty array. Let $l = |Q|$, we use a function $Q.\mathsf{push}(x)$ to denote the appending of x to the array Q, we use a function $x \leftarrow \mathcal{W}.\mathsf{pop}()$ to denote returning $x = Q[1]$, setting $Q[i-1] = Q[i]$ for $i \in [l]$, and eliminating $Q[l]$, so that $|Q| = l - 1$. Finally, we use a function $Q.\mathsf{rem}(x)$ to denote finding the element x in Q, removing it if found, and then left-shifting all array elements to the right of this element, in the same manner as the $\mathsf{pop}()$ function.

We denote vectors $\boldsymbol{v} = (v_1, \ldots, v_m) \in \mathbb{R}^m$ using bold-face, and similarly (but capitalised) for matrices $\boldsymbol{M} = (\boldsymbol{v}_1|\boldsymbol{v}_2|\ldots|\boldsymbol{v}_n) \in \mathbb{R}^{m \times n}$, where $(\boldsymbol{v}_1|\boldsymbol{v}_2|\ldots|\boldsymbol{v}_n)$ denotes the concatenation of n column vectors into a single matrix. Similarly, we write $\boldsymbol{v} = [\boldsymbol{v}_1\|\ldots\|\boldsymbol{v}_n]$ to denote concatenation of n vectors $\boldsymbol{v}_i \in \mathbb{R}^{m_i}$ into a single vector $\boldsymbol{v} \in \mathbb{R}^{\sum_i m_i}$.

For $p \in \mathbb{N}$, we let $+_p$ denote the addition operator of elements in $\mathbb{Z}_p$, replacing with $+$ when the modular reduction is obvious. For $x \in \mathbb{Z}_q$ and $q > z > 0$, let $\lfloor x \rceil_{q,z}$ denote the computation of the rounding function $\lfloor (z/q) \cdot x \rceil \mod z$. For a distribution χ, we write $\boldsymbol{x} \leftarrow\!\!\$\, \chi^m$ to denote sampling the vector $\boldsymbol{x}$, where each entry x_i is sampled independently from χ. We let λ denote the concrete security parameter throughout.

2.2 Homomorphic Encryption for Public Inner-Products

A symmetric-key homomorphic encryption scheme for public inner-products (HEIP) allows encrypting vectors $\boldsymbol{v} \in \mathbb{Z}_p^m$ for some $p > 0$, into ciphertext vectors $\boldsymbol{c} \in \mathbb{Z}_q^m$ for $q > p$. With knowledge of $\boldsymbol{c}$ alone, the homomorphic capability of the scheme allows computation of an encryption, c', of the inner product $\langle \boldsymbol{v}, \boldsymbol{w} \rangle \in \mathbb{Z}_p$, for any public vector $\boldsymbol{w} \in \mathbb{Z}_p$. We formally define such an encryption scheme, Σ, in the following way.

- $\Sigma.\mathsf{kgen}(1^\lambda)$: Outputs a key sk and public parameters pp.
- $\Sigma.\mathsf{enc}(\mathsf{pp}, \mathsf{sk}, v \in \mathbb{Z}_p)$: Outputs an encryption c of a value $v \in \mathbb{Z}_p$ corresponding to the secret key sk.
- $\Sigma.\mathsf{eval}(\mathsf{pp}, \boldsymbol{c} \in \mathbb{Z}_q^m, \boldsymbol{w} \in \mathbb{Z}_p^m)$: Let $\boldsymbol{c} = (c_1, \ldots, c_m)$ be a vector of ciphertexts corresponding to sk, and $\boldsymbol{w}$ a plaintext vector. Outputs a ciphertext c'.
- $\Sigma.\mathsf{dec}(\mathsf{pp}, \mathsf{sk}, c)$: Outputs a value $v \in \mathbb{Z}_p$.

In the following, we may also abuse notation and write $\boldsymbol{c} \leftarrow \Sigma.\mathsf{enc}(\mathsf{sk}, \boldsymbol{v} \in \mathbb{Z}_p^m)$ to denote producing a vector of m ciphertexts, where the i^{th} ciphertext c_i encrypts the i^{th} value v_i of $\boldsymbol{v}$. Subsequently, Σ must satisfy the following correctness guarantee, and the standard IND-CPA security guarantee for public-key encryption schemes.

Definition 2.1 (Correctness of evaluation). Let $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{Z}_p^n$, let $\mathsf{sk} \leftarrow \Sigma.\mathsf{setup}(1^\lambda)$, and let $c_{\boldsymbol{v}} \leftarrow \Sigma.\mathsf{enc}(\mathsf{sk}, \boldsymbol{v})$. Then Σ is *correct* if the following guarantees hold.

(1) $\Pr[\boldsymbol{v} \leftarrow \Sigma.\mathsf{dec}(\mathsf{sk}, c_{\boldsymbol{v}})] > 1 - \mathsf{negl}(\lambda)$
(2) $\Pr[\langle \boldsymbol{v}, \boldsymbol{w} \rangle \leftarrow \Sigma.\mathsf{dec}(\mathsf{sk}, \Sigma.\mathsf{eval}(c_{\boldsymbol{v}}, \boldsymbol{w}))] > 1 - \mathsf{negl}(\lambda)$

Encryption scheme from LWE. As described in [29, 42], it is possible to build homomorphic encryption for general linear functions from LWE-based Regev encryption [66]. Let q, p, and n be $\mathsf{poly}(\lambda)$, and let χ_σ be a specific *error* distribution with parameter $\sigma = \mathsf{poly}(\lambda)$. For simplicity, we are going to assume throughout that $q > p$ and $p|q$, and we let $\Delta_{q,p} = q/p$. A description of the Regev-based scheme, Σ_{lwe}, that permits evaluation of public inner-products is as follows.

- $\Sigma_{\mathsf{lwe}}.\mathsf{kgen}(1^\lambda, q, p, n, \sigma)$: Samples $\boldsymbol{s} \leftarrow\!\!\$\, \chi_\sigma^n$, and returns $(\mathsf{pp}, \mathsf{sk}) = ((q, n, p, \chi_\sigma), \boldsymbol{s})$.
- $\Sigma_{\mathsf{lwe}}.\mathsf{enc}(\mathsf{pp}, \mathsf{sk}, v \in \mathbb{Z}_p)$: Samples $\boldsymbol{a} \leftarrow\!\!\$\, \mathbb{Z}_q^n$ and $e \leftarrow\!\!\$\, \chi_\sigma$, and computes $\hat{c} = \mathsf{sk} \cdot \boldsymbol{a} + e + \Delta_{q,p} \cdot v$. Returns $c = (\boldsymbol{a}, \hat{c})$.
- $\Sigma_{\mathsf{lwe}}.\mathsf{eval}(\mathsf{pp}, \boldsymbol{c} \in \mathbb{Z}_q^m, \boldsymbol{w} \in \mathbb{Z}_p^m)$: Parses $\boldsymbol{c}$ as $(\boldsymbol{A}, \hat{\boldsymbol{c}})$, where $\boldsymbol{A} = (\boldsymbol{a}_1|\ldots|\boldsymbol{a}_m) \in \mathbb{Z}_q^{n \times m}$ and $\hat{\boldsymbol{c}} = (\hat{c}_1, \ldots, \hat{c}_m) \in \mathbb{Z}_q^m$, and returns $c' = (\boldsymbol{a}', \hat{c}') = (\boldsymbol{A} \cdot \boldsymbol{w}, \hat{\boldsymbol{c}} \cdot \boldsymbol{w})$.
- $\Sigma_{\mathsf{lwe}}.\mathsf{dec}(\mathsf{pp}, \mathsf{sk}, c)$: It returns $\lfloor \hat{c} - (\mathsf{sk} \cdot \boldsymbol{a}) \rceil_{q/p}$.

Security. It's known from [66] that such an encryption scheme can be proven secure based on the worst-case hardness of known problems over lattices when χ_σ is a discrete Gaussian distribution centred at zero, with standard deviation σ. In [29], an alternative hardness guarantee is given based on the *Ternary LWE* problem, in the case that $\chi = \chi_\sigma$ is chosen to be the uniform ternary distribution that samples elements from $\{0, \pm 1\}$.

Correctness. Regev's encryption is widely known to be additively homomorphic: given two ciphertexts $c_1 = (a_1, \hat{c}_1)$ and $c_2 = (a_2, \hat{c}_2)$, their sum $c_+ = (a_1 + a_2, \hat{c}_1 + \hat{c}_2)$ decrypts to the sum of the plaintexts, as long as the noise does not grow too large. We now highlight parameter settings that have been shown to satisfy correctness with respect to public inner products with vectors in $\mathbb{Z}_p^m$, when considering both Gaussian and uniform ternary error distributions, using the following lemma.

LEMMA 2.2 (CORRECTNESS [29, 42]). *Σ_{lwe} produces correct decryptions with probability $1 - \delta$ (for $\delta > 0$) for public inner products with vectors $\boldsymbol{w} \in \mathbb{Z}_p^m$, if at least one of the following conditions holds.*

- *$\chi = \chi_\sigma$ is a Gaussian error distribution with standard deviation parameter $\sigma = s^2/2\pi$ for $s > 0$, and $q \geq \sqrt{2\ln(2/\delta)} \cdot sigma \cdot p^2 \cdot m^{1/2}$.*
- *χ is a uniform distribution over $\{0, \pm 1\}$, and $q \geq 8 \cdot p^2 \cdot \sqrt{m}$.*

In the second case, $\delta = \mathsf{negl}(\lambda)$ via naive application of the Central Limit Theorem [29].

Clearly, the correctness property can be extended beyond the statistical error distributions considered in this work.

Preprocessing inner-products. In both [29, 42], it is shown that encryptions in Σ_{lwe} can be preprocessed for a global

Construction 2.1: Generic PIR scheme

A PIR protocol between a server holding $\mathsf{DB} \subseteq \mathcal{X}^m$*, and a client that wishes to learn* $\mathsf{DB}[i]$.

(1) The server runs $\mathsf{pp_{DB}} \leftarrow \mathsf{PIR.setup}(1^\lambda)$, and makes $\mathsf{pp_{DB}}$ publicly available.
(2) The client runs $(\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{PIR.query}(\mathsf{pp_{DB}}, i)$, sends q to the server, and stores $(\mathsf{q}, \mathsf{st})$.
(3) The server runs $\mathsf{r} \leftarrow \mathsf{PIR.respond}(\mathsf{pp_{DB}}, \mathsf{DB}, \mathsf{q})$, and returns r to the client.
(4) The client runs $\mathsf{x} \leftarrow \mathsf{PIR.process}(\mathsf{pp_{DB}}, \mathsf{st}, \mathsf{r})$, and outputs x.

matrix $\boldsymbol{A}$. Note that $\boldsymbol{A} \leftarrow\!\!\$\, \mathsf{PRG}(\beta)$ given a uniformly sampled pseudorandom generator seed $\beta \leftarrow\!\!\$\, \{0,1\}^\lambda$, and $\boldsymbol{A}$ is used globally for encrypting vectors of size m.[2] Regev's encryption remains secure with this pseudo-random "global" matrix $\boldsymbol{A}$ when used to encrypt polynomially many messages provided that each ciphertext uses both an independent secret vector $\boldsymbol{s}$ and an error vector $\boldsymbol{e}$ [63]. To denote using such an encryption mechanism, we will write $\Sigma_{\mathsf{lwe}}.\mathsf{enc}_{\boldsymbol{A}}(\mathsf{sk}, \boldsymbol{v})$ for some vector $\boldsymbol{v} \in \mathbb{Z}_p^m$. This modification allows pre-processing inner-products by computing $\boldsymbol{d} = \boldsymbol{A} \cdot \boldsymbol{x} \in \mathbb{Z}_q^n$, for some $\boldsymbol{x}$. Then, when performing evaluations and decryptions, it is enough to only operate on the right-hand side of the ciphertext. Finally, decryption is performed in the normal way, using the secret key $\mathsf{sk} = \boldsymbol{s}$ that is used in the original encryption.

2.3 Private Information Retrieval

Let $\mathsf{DB} \in \mathcal{X}^m$ represent a database containing m elements sampled from some element space $\mathcal{X}$. A (single-server) Private Information Retrieval (PIR) scheme [23], denoted by PIR, consists of the following algorithms.[3]

- $\mathsf{pp_{DB}} \leftarrow \mathsf{PIR.setup}(1^\lambda, \mathsf{DB})$: An algorithm that outputs a set of public parameters.
- $(\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{PIR.query}(\mathsf{pp_{DB}}, i)$: An algorithm that takes some public parameters, and an index $i \in [m]$ as input and outputs a query $\mathsf{q} \in \{0,1\}^*$ and some corresponding state $\mathsf{st} \in \{0,1\}^*$.
- $\mathsf{r} \leftarrow \mathsf{PIR.respond}(\mathsf{pp_{DB}}, \mathsf{DB}, \mathsf{q})$: An algorithm that takes some public parameters, the database, and a query as input. The algorithm outputs a response $\mathsf{r} \in \{0,1\}^*$.
- $\mathsf{x} \leftarrow \mathsf{PIR.process}(\mathsf{pp_{DB}}, \mathsf{st}, \mathsf{r})$: An algorithm that takes public parameters, the corresponding state, and a response as input. The algorithm outputs an element $\mathsf{x} \in \mathcal{X}$.

A generic PIR protocol (using the algorithms defined above) is described in Construction 2.1. All such protocols must satisfy three properties: *correctness*, *security*, and *efficiency*. Broadly speaking, correctness guarantees that a query returns the intended result in the database, security guarantees that the client query hides the index being retrieved, and efficiency guarantees that the solution is more efficient than the trivial solution of downloading the entire database. We provide formal realisations of each property below.

Definition 2.3 (Correctness). Let P_{x} be the probability that the protocol in Construction 2.1 outputs x, where $\mathsf{DB}[i] = \mathsf{x}$. We say that PIR is *correct* if and only if $P_{\mathsf{x}} > 1 - \mathsf{negl}(\lambda)$.

For completeness, we may alternatively allow the query functionality to take an indicator vector as input corresponding to the index i that should be queried. In other words, we may write, $\mathsf{PIR.query}(\mathsf{pp_{DB}}, \boldsymbol{f})$, where $\boldsymbol{f} \in \{0,1\}^m$. Typically, for correctness to hold, we require that $f_i = 1$, if and only if i is the index that should be queried, and 0 elsewhere.

Definition 2.4 (Security). For a PPT algorithm $\mathcal{A}$, let $P^{\mathcal{A}}_{b,b'}$ be the probability that $\mathcal{A}$ outputs $b' = b$ in ℓ-$\mathsf{QIND}^{\mathcal{A}}_{\mathsf{PIR}}$ (Figure 2). We say that PIR is *secure* (and satisfies *ℓ-query indistinguishability*) if $|P^{\mathcal{A}}_b - 1/2| < \mathsf{negl}(\lambda)$ for all such algorithms $\mathcal{A}$.

Definition 2.5 (Efficiency). For a single client launching $O(1)$ queries, PIR is *efficient* if the total communication overhead is smaller than the total bit-length of DB.

PIR for keyword queries. Keyword PIR schemes were first introduced in [22], and consider key-value map databases DB, where elements $\mathsf{x} \in \mathcal{X}$ contained in DB are associated with keys $\mathsf{k} \in \mathcal{K}$, for a key space $\mathcal{K}$. To allow keyword queries to be made against a PIR database, it is necessary to modify the $\mathsf{PIR.query}$ functionality, as seen below.

- $(\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{PIR.query}(\mathsf{pp_{DB}}, \mathsf{k})$: An algorithm that takes public parameters and a key $\mathsf{k} \in \mathcal{K}$ as input, and returns the query q and the state st.

The generic construction in 2.1 can then be modified to have a client that wishes to learn the value associated with k in DB. Correctness, then, is defined as follows.

Definition 2.6 (Correctness for keyword queries). Let $P_{\mathsf{k},\mathsf{x}}$ be the probability that the scheme in Construction 2.1 outputs x, where $\mathsf{DB.read}(\mathsf{k}) = \mathsf{x}$. We say that PIR is *correct* if and only if $P_{\mathsf{k},\mathsf{x}} > 1 - \mathsf{negl}(\lambda)$.

Security is defined in the same way as in Definition 2.4, but using the ℓ-$\mathsf{kwQIND}^{\mathcal{A}}_{\mathsf{PIR}}$ security game defined in Figure 2 (i.e. considering the highlighted lines).

2.4 LWE-based PIR

A recent line of work [24, 29, 42, 48, 75] has focused on creating practical PIR schemes based directly on LWE. We refer to those schemes as "LWE-based PIR" (LWEPIR). In this work, we provide a high-level framework that captures the functionality of each of these approaches. This framework allows us to discuss and implement the functionality of each of these existing schemes, without relying on peculiarities of any individual approach.

[2]Security of the scheme then follows from LWE with polynomial security loss (i.e. Matrix LWE [29]), from a standard hybrid argument.
[3]We only consider single-server (*computationally-secure*) PIR schemes in this work. Section 1.2 discusses multi-server approaches.

```
Experiment $\ell$-QIND$^{\mathcal{A}}_{\mathsf{PIR}}$   ($\ell$-kwQIND$^{\mathcal{A}}_{\mathsf{PIR}}$)
1 :  $b \leftarrow\$ \{0,1\}$
2 :  $\mathsf{pp}_{\mathsf{DB}} \leftarrow \mathsf{PIR.setup}(1^\lambda)$
3 :  $(i_1,\ldots,i_\ell),(j_1,\ldots,j_\ell) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{DB}},\mathsf{DB})$
     $(\mathsf{k}_1,\ldots,\mathsf{k}_\ell),(\mathsf{k}'_1,\ldots,\mathsf{k}'_\ell) \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{DB}},\mathsf{DB})$
4 :  $\mathcal{T} = [(i_\iota + b * (j_\iota - i_\iota))$ for $\iota \in [\ell]]$
     $\mathcal{T} = [(\mathsf{k}_\iota + b * (\mathsf{k}'_\iota - \mathsf{k}_\iota))$ for $\iota \in [\ell]]$
5 :  $Q = \emptyset$
6 :  for $t \in \mathcal{T}$:
7 :    $(\mathsf{q},\mathsf{st}) \leftarrow \mathsf{PIR.query}(\mathsf{pp}_{\mathsf{DB}}, t)$
8 :    $Q.\mathsf{push}(\mathsf{q})$
9 :  $b' \leftarrow \mathcal{A}(\mathsf{pp}_{\mathsf{DB}},\mathsf{DB},Q)$
```

Figure 2: ℓ-query indistinguishability for (keyword) PIR.

Background. The design of each of LWEPIR schemes is similar to established Ring LWE-based PIR ones (e.g. [6, 55]). In essence, the database (with m entries) contains entries in $\mathbb{Z}_p$ and it is organised as a vector, where each entry of the vector is a single entry of the database. The client encrypts a vector of length m with all-zeroes except for a single entry that is equal to 1 in the i^{th} position, using an HEIP scheme for $\mathbb{Z}_p$. The client, then, sends this ciphertext vector to the server. The server, in turn, evaluates the inner-product homomorphically between the received encrypted vector and their database vector, and sends this evaluated vector back to the client. The client decrypts it, and learns the i^{th} element of the database.

Clearly, this functionality could be implemented using classical additively homomorphic encryption (e.g. Paillier-based [57]). However, it has been shown that using such schemes in the PIR setting results in schemes that are prohibitively expensive to run, in comparison with simply downloading the database in its entirety [69]. The work of [1] showed that such a formulation could be made efficient enough for real-world use-cases, when using Ring LWE-based FHE schemes. A long line of follow-up works [3, 6, 26, 32, 52, 55, 59, 60, 76] have exploited various features of such FHE schemes (for example, by batching ciphertexts and reducing costs using NTT transformations) to make PIR schemes fairly efficient. In the case of [26], it was shown that asymptotic costs can be reduced to $O(\sqrt{m})$, when making $\sqrt{N}$ queries.

Nevertheless, even with such changes, many problems remain for deploying any such PIR scheme at scale. For instance, standard PIR schemes that require no offline setup are costly to run, requiring many seconds and large amounts of bandwidth to run queries on relatively small databases (e.g. $m < 2^{20}$) [3, 6, 52, 55]. So-called *stateful schemes* typically move expensive computation to an offline phase that makes the subsequent phase, the *online* one, cheaper. This offline phase, however, involves heavy computation as, for example, streaming the entire database to every client [60, 76] (which violates the PIR efficiency criterion), or heavy FHE-based computation that is client-dependent, and therefore unlikely to scale for large numbers of clients [52, 55].

Recent work has shown that LWE-based HEIP schemes can be used to develop PIR schemes that are cheap to run in the preprocessing (or "offline") paradigm [24, 29, 42, 48]. In particular, the main advantages include an offline state generation phase that is independent of clients, and with much smaller download footprints than the actual database. Furthermore, such schemes are very simple to implement using Regev-based encryption, relying only on standard unsigned 32-bit integer instructions. Put together, these advantages translate to much smaller concrete costs: costing only tens of milliseconds of computation, and hundreds of kB of amortised costs per query. As a result, such schemes would appear to represent the state-of-the-art for continuing to build more efficient alternatives.

High-level LWEPIR framework. Each LWEPIR scheme relies on a variant of the Regev-based HEIP scheme described in Section 2.2, where a large part of the encryption functionality can be performed *in advance* and be reused over multiple clients. Hence, LWEPIR schemes have two phases: a preprocessing phase that can be amortised over multiple clients, and a per-client online phase. We describe both of these phases in the following.

Pre-processing Phase. Recall that the database (DB) has a size denoted by $m \in \mathbb{N}$. The purpose of the pre-processing phase is to generate a global public state, prior to any individual client query to the server.

Server setup: ($\mathsf{pp}_{\mathsf{DB}} \leftarrow \mathsf{LWEPIR.setup}(1^\lambda, \mathsf{DB})$). The server constructs their database DB containing m elements, each of size w, and samples a short random seed $\beta \in \{0,1\}^\lambda$. Let $m \cdot w = m_1 \cdot m_2$, for $m_1, m_2 \in \mathbb{N}$. The server derives a matrix $\boldsymbol{A} \leftarrow \mathsf{PRG}(\beta, n, m, q) \in \mathbb{Z}_q^{n \times m_1}$, and encodes the DB in a matrix representation as $\boldsymbol{D} \in \mathbb{Z}_p^{m_1 \times m_2}$. It then computes $\boldsymbol{M} \leftarrow \boldsymbol{A} \cdot \boldsymbol{D}$ and publishes the pair $(\beta, \boldsymbol{M})$. It returns the public parameters $\mathsf{pp}_{\mathsf{DB}}$ containing LWE parameters (q, p, n, σ), the seed β, PIR parameters (m_1, m_2), and (optionally) the matrix $\boldsymbol{M}$.

Online Phase. The online phase allows the client to query for the desired database element, after downloading the aforementioned state.

Query. $[(\mathsf{q},\mathsf{st}) \leftarrow \mathsf{LWEPIR.query}(\mathsf{pp}_{\mathsf{DB}}, i)]$: The client downloads $(\beta, \boldsymbol{M})$ and derives $\boldsymbol{A} \leftarrow \mathsf{PRG}(\beta, n, m, q) \in \mathbb{Z}_q^{n \times m_1}$. The client then generates a unit vector $\boldsymbol{f}_i$: an all-zero vector with a single 1 at the index i. The client parses q, p, n, σ from $\mathsf{pp}_{\mathsf{DB}}$, calls $(\mathsf{pp}_{\mathsf{LWE}}, \mathsf{sk}) \leftarrow \Sigma_{\mathsf{lwe}}.\mathsf{kgen}(1^\lambda, q, p, n, \sigma)$, and runs $\boldsymbol{c} \leftarrow \Sigma_{\mathsf{lwe}}.\mathsf{enc}_{\boldsymbol{A}}(\mathsf{pp}_{\mathsf{LWE}}, \mathsf{sk}, \boldsymbol{f}_i)$, where the i^{th} element of $\boldsymbol{c}$, c_i, is an LWE encryption with respect to the i^{th} column, $\boldsymbol{a}_i$, of $\boldsymbol{A}$. The client parses $(\boldsymbol{A}, \hat{\boldsymbol{c}}) = \boldsymbol{c}$, lets $\mathsf{q} = \hat{\boldsymbol{c}}$, and lets $\mathsf{st} = \mathsf{sk} \cdot \boldsymbol{A}$. The client then sends q to the server.

Response. $[\mathsf{r} \leftarrow \mathsf{LWEPIR.respond}(\mathsf{pp}_{\mathsf{DB}}, \boldsymbol{D}, \mathsf{q})]$: The server receives q, and then parses their database matrix as a concatenation of column vectors: $\boldsymbol{D} = (\boldsymbol{db}_1 | \ldots | \boldsymbol{db}_{m_2})$. The server responds to the client with a vector r, where the i^{th} element r_i

of r is the ciphertext computed as $r_i \leftarrow \Sigma_{\mathsf{lwe}}.\mathsf{eval}(\mathsf{pp}_{\mathsf{LWE}}, \mathsf{q}, \boldsymbol{db}_i)$, for each $i \in [m_2]$.

Post-processing. [$\mathsf{x} \leftarrow \mathsf{LWEPIR.process}(\mathsf{pp}_{\mathsf{LWE}}, \mathsf{r}, \mathsf{st})$]: The client receives r and returns $\mathsf{x} \leftarrow \Sigma_{\mathsf{lwe}}.\mathsf{dec}(\mathsf{st}, \mathsf{sk}, \mathsf{r})$.

Correctness. Correctness of LWEPIR follows naturally from the correctness of Σ_{lwe}. First, q is an encryption of the all-zero vector, except in the i^{th} position where it encrypts 1. By the correctness of Σ_{lwe}, the server response is a public inner product of this encrypted vector, and the sequence of vectors in $\mathbb{Z}_p^{m_1}$ that make up the server database. Since the client simply decrypts the server response, correctness of the inner product holds, providing that the conditions in Lemma 2.2 hold for q, χ, m_1. Therefore, the server learns the vector $\mathsf{x} = (\boldsymbol{db}_1[i], \ldots, \boldsymbol{db}_{m_2}[i])$ which is equal to the i^{th} row, $\boldsymbol{D}[i]$, of the database matrix (which trivially decodes to the i^{th} row of the database itself).

Security. The security of the PIR scheme follows from the fact that the client message is simply a vector of Σ_{lwe} encryptions. By a trivial hybrid argument and the IND-CPA security of Σ_{lwe}, the client message hides element that they are querying.

Efficiency. The concrete efficiency of LWEPIR depends on the parameter choices. Intuitively, since the (amortisable) offline cost is $\boldsymbol{M} \in \mathbb{Z}_p^{n \times m_2}$ (where $n \ll m_1$) and the response is a vector $\mathsf{r} \in \mathbb{Z}_p^{m_2}$, then the total bandwidth usage is significantly smaller than the size of the database.

Differences between constructions. In the following, we discuss the differences between available LWEPIR schemes. Without loss of generality, the rest of our work remains agnostic to the specific choice of scheme. However, we describe our eventual PIR scheme in Section 5 using the FrodoPIR matrix formulation [29], as we believe it provides the cleanest interface for building a keyword PIR scheme. We discuss in Section 5.1 the changes that can be made to support alternative formats. Moreover, such modifications primarily relate to the offline phase, while the online phase is almost identical.

Across each of the schemes, q is typically taken to be 2^{32}, which allows performing encrypted operations as high speed native CPU instructions. This implies choosing a power-of-two p, to ensure that $p|q$, though this is not mandatory.

Matrix encoding. The encoding process of DB (that renders $\boldsymbol{D}$) differs between the SimplePIR approach of [24, 42] and the FrodoPIR approach of [29]. The former encodes it with the "square-root" approach [19, 46]: $\boldsymbol{D} \in \mathbb{Z}_p^{\sqrt{m} \times \sqrt{m}}$, and the latter in their own parsing manner: $\boldsymbol{D} \in \mathbb{Z}_p^{m \times d}$ where $d = \lceil w/\log(p) \rceil$[4]. In all schemes, the encrypted vector takes the form of $(0, \ldots, 0, 1, 0, \ldots, 0)$, an all-zero vector except where $\boldsymbol{f}_i[i] = 1$. However, in [24, 42], elements are assumed to correspond to only subsets of columns of the database matrix. In FrodoPIR, the entire row is treated as a single database entry.

The square-root approach results in asymptotic communication cost of $O(\sqrt{m})$, whereas the FrodoPIR approach costs $O(m)$ for the client query, and $O(w/\log(p))$ in the download. However, FrodoPIR appears to make this choice to reduce concrete financial costs in different architectures. In AWS, for example, upload communication is free, and download communication is not. See Section 5.1 for wider discussion.

State download. In SimplePIR [42] and FrodoPIR [29], the matrix $\boldsymbol{M}$ is included in $\mathsf{pp}_{\mathsf{DB}}$ and downloaded. The DoublePIR approach of [42], as well as the very recent the approach of [48], differ in that they highlight that the square-root encoding allows for decoding the response with a smaller portion of the state. The crucial observation is that only $w/\log(p)$ columns of $\boldsymbol{M}$ are needed to decode a PIR query for any given element in $\boldsymbol{D}$. In both approaches, effectively the client runs a separate PIR scheme to retrieve these w columns, and then uses the result to run PIR on the whole database. It is shown in both works that applying this change results in concrete efficiency gains in certain situations.

Query preprocessing. FrodoPIR [29] provides a client preprocessing step for the client, that allows performing a large part of query generation before knowledge of the query is required. In effect, this requires generating $\boldsymbol{b'} = \mathsf{sk} \cdot \boldsymbol{A} + \boldsymbol{e}$ and $\boldsymbol{d'} = \mathsf{sk} \cdot (\boldsymbol{A} \cdot \boldsymbol{D})$ and storing these values securely (i.e. not revealing them).[5] Generating the query in the online phase simply requires adding $(q/p) \cdot \boldsymbol{f}_i$ to $\boldsymbol{b'}$ to generate q, before sending it to the server. On decryption, the client simply subtracts $\boldsymbol{d'}$ directly from the server response, during post-processing.

2.5 Key-value maps

A key-value (KV) map consists of two algorithms, set and read. The $\mathsf{set}(\mathsf{k}, \mathsf{x})$ operation writes the value $\mathsf{x} \in \mathcal{X}$ to the key $\mathsf{k} \in \mathcal{K}$. The $\mathsf{read}(\mathsf{k})$ operation, returns x if (k, x) has been written previously, and $\perp$ otherwise.

Real-or-random key-value (RoRKV) maps are similar except that $\mathsf{read}(\mathsf{k})$ returns some random value $r \in \mathcal{X}$ when (k, x) has not been previously written. It is possible to construct standard KV maps from any RoRKV map, at the cost of increasing the storage of each element by the map key length. The construction is defined as follows.

- $\mathsf{KV.set}(\mathsf{k}, \mathsf{x})$: run $\mathsf{RoRKV.set}(\mathsf{k}, \mathsf{k}\|\mathsf{x})$.
- $\mathsf{KV.read}(\mathsf{k})$: run $y \leftarrow \mathsf{RoRKV.set}(\mathsf{k})$, parse $\mathsf{k}'\|\mathsf{x}' \leftarrow y$, output x' if $\mathsf{k}' = \mathsf{k}$, and $\perp$ otherwise.

To reduce the impact of the length of the key on storage, we can instead store values as $\mathsf{hash}(k)\|\mathsf{x}$, where hash is a universal hash function.

3 PROBABILISTIC KEY-VALUE FILTERS

In this section, we summarise and formalise the concept probabilistic key-value filters, that allow efficiently storing and querying key-value maps., with some false-positive probability $\epsilon > 0$. This structure will form the basis of a keyword querying mechanism, which we will use to build our PIR scheme in Section 5. In Appendix A, we provide additional formalisations for filter designs that focus only on encoding sets, rather than maps, for providing additional context.

[4]Thus, the i^{th} row consists of d $\log(p)$-bit chunks of $\mathsf{DB}[i] \in \mathbb{Z}_p^d$.

[5]Note that this technique is mentioned as "LinPIR" in [48].

3.1 Key-Value Filters

Key-Value filters are a form of storage that allows encoding key-value maps $\mathcal{M}$ of pairs $(\mathsf{k}, \mathsf{x}) \in \mathcal{K} \times \mathcal{X}$, for key and value domains $\mathcal{K}, \mathcal{X} \subseteq \{0,1\}^*$, respectively. Key-value filters consist of four algorithms $\mathsf{setupFilter}$, write, check, and $\mathsf{reconstruct}$, and are parametrised by a fingerprint function, $\mathsf{fpt}_\epsilon : \{0,1\}^* \mapsto \{0,1\}^\mu$. The function fpt_ϵ is, in turn, parametrised by the false-positive probability via the polynomial $\mu = \mu(\epsilon)$. In essence, for every check on a value x to the filter, a fingerprint y is returned, and we say that x is in KV if $y = \mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x})$. The algorithmic structure of key-value filter is provided below.

- $(\mathsf{F}, \mathsf{H}) \leftarrow \text{KeyValue}.\mathsf{setupFilter}(1^m, \epsilon, \circ)$: A static initialisation function that generates a filter F of size $N = O(\log(1/\epsilon)m)$, and a set of hash functions $\mathsf{H} = \{\mathsf{h}_i\}_{i \in [k]}$ for some $k \in \mathbb{N}$, and where $\mathsf{h}_i : \{0,1\}^* \mapsto [N]$. The input $\circ$ defines a mathematical operation that is used for reconstructing data items.
- $b \leftarrow \mathsf{F}.\mathsf{write}(\mathcal{M}, \mathsf{H}, \mathsf{fpt}_\epsilon)$: Writes a map $\mathcal{M}$ to F using the set of hash functions in H, and returns $b = 1$ if successful, and $b = 0$ otherwise. If $b = 0$, it may be necessary to regenerate the filter.
- $\mathsf{F}[\mathsf{H}(\mathsf{k})] \leftarrow \mathsf{F}.\mathsf{check}(\mathsf{k}, \mathsf{H})$: Simply evaluates $\mathsf{H}(\mathsf{k})$ for k, and returns $\mathsf{F}[\mathsf{H}(\mathsf{k})]$.
- $\mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) \leftarrow \mathsf{F}.\mathsf{reconstruct}(\mathsf{k}, \mathsf{H}, \mathsf{fpt}_\epsilon)$: Runs $\mathsf{F}[\mathsf{H}(\mathsf{k})] \leftarrow \mathsf{F}.\mathsf{check}(\mathsf{k}, \mathsf{H})$, and then returns $\bigcirc_{i=1}^{k} \mathsf{F}[\mathsf{h}_i(\mathsf{k})] = \mathsf{F}[\mathsf{h}_1(\mathsf{k})] \circ \ldots \circ \mathsf{F}[\mathsf{h}_k(\mathsf{k})]$.

Correctness. We can express the correctness of a key-value filter with respect to the following two definitions.

Definition 3.1 (Correctness of inclusion). Let $(\mathsf{F}, \mathsf{H}) \leftarrow \text{KeyValue}.\mathsf{setupFilter}(1^m, \epsilon)$, and let $\mathcal{M}$ be any map $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{X}$ such that $(\mathsf{k}, \mathsf{x}) \in \mathcal{K} \times \mathcal{X}$ is contained within $\mathcal{M}$. We say that F correctly indicates inclusion if the following equality holds:

$$\Pr\left[y = \mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) \,\middle|\, \substack{1 \leftarrow \mathsf{F}.\mathsf{write}(\mathcal{M}, \mathsf{H}, \mathsf{fpt}_\epsilon) \\ y \leftarrow \mathsf{F}.\mathsf{reconstruct}(\mathsf{k}, \mathsf{H}, \mathsf{fpt}_\epsilon)}\right] = 1.$$

Definition 3.2 (Correctness of non-inclusion). Let $(\mathsf{F}, \mathsf{H}) \leftarrow \text{KeyValue}.\mathsf{setupFilter}(1^m, \epsilon)$, and let $\mathcal{M}$ be any set $\mathcal{M} \subseteq \{0,1\}^*$ such that $\mathsf{x} \in \{0,1\}^*$ is **not** contained within $\mathcal{M}$. We say that F correctly indicates non-inclusion (with false-positive probability ϵ), if the following inequality holds:

$$\Pr\left[y = \mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) \,\middle|\, \substack{\mathsf{F}.\mathsf{write}(\mathcal{M}, \mathsf{H}, \mathsf{fpt}_\epsilon) \\ y \leftarrow \mathsf{F}.\mathsf{reconstruct}(\mathsf{k}, \mathsf{H}, \mathsf{fpt}_\epsilon)}\right] \leq \epsilon.$$

Constructions. Key-value filters can be built directly from any fingerprint-based filters (see Appendix A.2) that operate only over sets (e.g. Cuckoo, XOR, or Binary Fuse filters). In principle, queries are performed over keys $\mathsf{k} \in \mathcal{K}$, and the $\mathsf{reconstruct}$ algorithm returns $\mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x})$ for $\mathsf{x} \in \mathcal{X}$, if (k, x) is encoded in the filter.[6] More concretely, we can adapt fingerprint-based filters to support key-value functionality by simply modifying the storage procedure to perform $\mathsf{F}[\mathsf{H}(\mathsf{k})] \leftarrow \mathsf{F}.\mathsf{check}(\mathsf{k}, \mathsf{H})$, and then setting $\mathsf{F}[\mathsf{h}_1(\mathsf{k})] = \mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) \circ (-\mathsf{F}[\mathsf{h}_2(\mathsf{k})]) \circ \ldots \circ (-\mathsf{F}[\mathsf{h}_k(\mathsf{k})])$.

[6]In fingerprint-based filters, fpt_ϵ is defined only over the value space.

Probabilistic Key-Value Maps. To build probabilistic key-value maps (Section 2.5) from key-value filters, we can instantiate the $\mathsf{KV.read}$ function using the $\mathsf{F.reconstruct}$ algorithm. However, this implicitly requires a mechanism for extracting the value x explicitly from the output of $\mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x})$. In concrete constructions, fpt_ϵ is defined as some sort of hash function to ensure that the false-positive probability is lowest. However, such hash functions do not necessarily allow for extraction. Clearly, one could modify fpt_ϵ to be the identity function, such that it returns x directly, but this would effectively violate Definition 3.2 unless we parametrise $\mathcal{X}$ by ϵ. Therefore, to provide extraction with false-positive probability based on ϵ, we can simply use the fingerprint function $\mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) = \mathsf{hash}(\mathsf{k}) \| \mathsf{x}$, where hash is a universal hash function.

In addition, since we work with filters that are *selective* by design — in other words, requiring that the entire set/map of elements is written in one go — we must also impose the same restriction on the key-value map.

Matrix representation and filter concatenation. For a key-value filter, F, we write $\boldsymbol{F} \leftarrow \mathsf{Matrix}(\mathsf{F}) \in \mathcal{X}^{N \times 1}$ to denote the matrix representation of F. In other words, the i^{th} entry of $\boldsymbol{F}$ corresponds to the i^{th} concrete element in F. Clearly, $\boldsymbol{F}$ is a vector, but later we make use of the fact that the concatenation of d filters (each using the same set of hash functions, H) can be expressed generically as a matrix $\boldsymbol{F} \in \mathcal{X}^{N \times d}$, where the $(i,j)^{\text{th}}$ position $F_{i,j}$ corresponds to the i^{th} entry of the j^{th} concatenated filter.

To express a concatenated filter, F, we abuse notation and write $\mathsf{F} = (\mathsf{F}_1, \ldots, \mathsf{F}_d)$, where each F_i is an individual filter. This representation allows expressing $d \cdot \log p$ bits of information per filter-entry. We further abuse notation and write $\mathsf{F}.\mathsf{write}(\mathcal{M}, \mathsf{H}, \mathsf{fpt}_\epsilon)$ and $y \leftarrow \mathsf{F}.\mathsf{reconstruct}(\mathsf{k}, \mathsf{H}, \mathsf{fpt}_\epsilon)$, which allows us to express running $\mathsf{F}_i.\mathsf{write}(\mathcal{M}, \mathsf{H}, \mathsf{fpt}_\epsilon)$ and $y_i \leftarrow \mathsf{F}_i.\mathsf{reconstruct}(\mathsf{k}, \mathsf{H}, \mathsf{fpt}_\epsilon)$ individually, for each $i \in [d]$. In the case of reconstruction, the response y is equal to the bit-concatenation expressed by $y_1 \| \ldots \| y_d$.

4 BINARY FUSE FILTERS FOR $\mathbb{Z}_p$

Binary Fuse filters (BFFs) (as well as their predecessors, XOR filters [39]) were first introduced by Graf and Lemire [40] as an alternative filter-design, focused specifically on minimizing the space and query overheads of key-value filters, while maintaining quick access times. Compared with XOR filters, the constant space overhead can be reduced to $\varsigma \in \{\approx 1.08, \approx 1.13\}$, for the number of hash functions $k \in \{3, 4\}$, respectively. In principle, these savings are achieved by breaking the filter into many, much smaller segments $\mathsf{F} = (\mathsf{F}_1 \| \ldots \| \mathsf{F}_P)$, where P is chosen to be some value that ensures that each segment F_i contains 2^g entries, for some $g \in \mathbb{N}$. Hash function evaluations map uniformly to k contiguous segments (i.e. mapping uniformly to $\{0,1\}^g$), and then reconstruction of $\mathsf{fpt}_\epsilon(\mathsf{x})$ is performed using the XOR operation in $\{0,1\}^\mu$.

Note that Cuckoo and XOR filters can be seen as subclasses of BFFs, where $k = 2$ and $k = 3$, respectively, and segments are chosen to be much larger. However, as well as the reduction in space requirements, hash function evaluations for

BFFs can be chosen to map natively to g-bit domains, making such function accesses cheaper. Concretely, it is shown in [40] that such filters maintain higher performance than Cuckoo filters, even for plausibly negligible ϵ (e.g. 2^{-40}). Even so, while cuckoo filters have been widely used in cryptographic schemes, BFFs are yet to see any significant usage.

Formal description. First, let $\mathcal{K} = \{0,1\}^*$, let $\mathcal{X} = \{0,1\}^l$ for some $l \in \mathbb{N}$, and let $\mathsf{hash} : \{0,1\}^* \mapsto \{0,1\}^\mu$ be a universal hash function, where $\mu \geq 2\epsilon$. We consider key-value maps of the form $\mathcal{M} \subseteq \mathcal{K} \times \mathcal{X}$. Then, we consider the function $\mathsf{fpt}_\epsilon : \mathcal{K} \times \mathcal{X} \mapsto \{0,1\}^{\mu+l}$ as defined in Section 3.1 (see also Algorithm 1), for instantiating probabilistic key-value maps. In other words, $\mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) = \mathsf{hash}(\mathsf{k}) \| \mathsf{x}$. To consider elements in $\mathbb{Z}_p$, we set $p = 2^{\mu+l}$ and let $\circ$ be mod-p addition. All changes so far can be made without altering any of the internal characteristics of the filter itself. We write $\mathsf{BFF_p}$ when referring to such BFFs from this point forth. The formal description of each of the functions is given below.

Formally speaking, BFFs for $\mathbb{Z}_p$ instantiate the algorithms for key-value filters (Section 3.1) in the following way.

- $(\mathsf{F}, \mathsf{H}, \mathsf{fpt}_\epsilon) \leftarrow \text{KeyValue}.\mathsf{setupFilter}(1^m, \epsilon, \circ)$: Runs Algorithm 1 defined in Appendix C.
- $b \leftarrow \mathsf{F}.\mathsf{write}(\mathcal{M}, \mathsf{H})$: Runs Algorithm 2 defined in Appendix C.
- $\mathsf{F}[\mathsf{H}(\mathsf{k})] \leftarrow \mathsf{F}.\mathsf{check}(\mathsf{k}, \mathsf{H})$: Simply evaluates $\mathsf{H}(\mathsf{k})$ for k, and returns $\mathsf{F}[\mathsf{H}(\mathsf{k})]$.
- $\mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) \leftarrow \mathsf{F}.\mathsf{reconstruct}(\mathsf{k}, \mathsf{H})$: First runs $\mathsf{F}[\mathsf{H}(\mathsf{k})] \leftarrow \mathsf{F}.\mathsf{check}(\mathsf{k}, \mathsf{H})$, and returns $\bigcirc_{i=1}^{k} \mathsf{F}[\mathsf{h}_i(\mathsf{k})] = \mathsf{F}[\mathsf{h}_1(\mathsf{k})] \circ \ldots \circ \mathsf{F}[\mathsf{h}_k(\mathsf{k})]$.

Correctness. Since we are building a key-value (key-value) filter, we must show that $\mathsf{BFF_p}$ satisfies correctness, as dictated by Definition 3.1 and Definition 3.2. It is shown in [40] that the write algorithm encodes a set with absolute correctness for reading included elements. The only change in Algorithm 2 is that the fpt_ϵ is computed differently, and so correctness of inclusion follows immediately.

For correctness of non-inclusion, [40] shows, experimentally, that the choice of the parameter ç defined in Algorithm 1 ensures correctness of non-inclusion with false-positive probability ϵ, where 2ϵ bits are stored filter per-entry. However, our analysis differs, in that a false-positive occurs based on the first μ bits of the output of fpt_ϵ. We show in Lemma 4.1 that the probability of a false-positive occurring is indeed bounded above by $2^{-\mu}$ for $\mu = \mu(\epsilon)$, based on the choice of $\mathsf{hash} : \{0,1\}^* \mapsto \{0,1\}^\mu$ being a universal hash function.

LEMMA 4.1 (CORRECTNESS OF NON-INCLUSION). $\mathsf{BFF_p}$ *satisfies correctness of non-inclusion (Definition 3.2), with false-positive probability* $2^{-\mu/2} \leq 2^{-\epsilon} = \mathsf{negl}(\epsilon)$.

PROOF. Consider a key $\mathsf{k} \notin \mathcal{M}$, and a filter description which is sampled as $(\mathsf{F}, \mathsf{H}, \mathsf{fpt}_\epsilon) \leftarrow \text{KeyValue}.\mathsf{setupFilter}(1^m, \epsilon)$, and after writing $\mathsf{F}.\mathsf{write}(\mathcal{M}, \mathsf{H}, \mathsf{fpt}_\epsilon)$. A collision in the algorithm $y \leftarrow \mathsf{F}.\mathsf{reconstruct}(\mathsf{k}^*, \mathcal{M})$ occurs when $y = \mathsf{hash}(\mathsf{k}^*) \| \mathsf{x}'$, for any value x'. To quantify the chance of this event occurring, we consider two possible types of events.

The first type corresponds to when the result of $\mathsf{H}(\mathsf{k}^*) = \mathsf{H}(\mathsf{k})$ for some $\mathsf{k} \in \mathcal{M}$. Then the result $y = \mathsf{hash}(\mathsf{k}) \| \mathcal{M}[\mathsf{k}]$, in which case a collision occurs if $\mathsf{hash}(\mathsf{k}) = \mathsf{hash}(\mathsf{k}^*)$. By the choice of hash as a universal hash function mapping to $\{0,1\}^\mu$, we know that the chance of a collision occurring is $2^{-\mu/2}$.

The second type corresponds to when the result of $\mathsf{H}(\mathsf{k}^*) \neq \mathsf{H}(\mathsf{k})$ for any $\mathsf{k} \in \mathcal{M}$. In this case, the result of running $\mathsf{F}.\mathsf{check}(\mathsf{k}^*, \mathsf{H}(\mathsf{k}^*))$ returns k entries in F, which are independently distributed. Therefore, when summing these entries in $\mathsf{F}.\mathsf{reconstruct}$, we retrieve a sum of independently and uniformly distributed elements in $\mathbb{Z}_p$. As a result, the chance of the first μ bits being equal to $\mathsf{hash}(\mathsf{k}^*)$ is also equal to $2^{-\mu/2}$, by the fact that hash maps uniformly to $\{0,1\}^\mu$. By the choice of $\mu \geq \epsilon$, the rest of the statement follows. □

4.1 Supporting Larger Values

Note that the modifications proposed previously require that the entire value of a RoRKV fit within $\mathbb{Z}_p$, which is likely to incur large costs when it comes to implementing modular arithmetic. Alternatively, as discussed in previously in Section 3.1, filter representations that hold elements in $\mathbb{Z}_p$ can be concatenated, into a vector $\widehat{\mathsf{F}} = (\mathsf{F}_1, \ldots, \mathsf{F}_d)$, where each F_j corresponds to a filter that holds $\log(p)$ bits of data in every element, for $j \in [d]$, leading to concatenated filter that holds $d \log(p)$ bits per-entry. The same set of hash functions H is used to query each filter, meaning that the same locations in each filter are returned when querying a single key. Therefore, to query $\widehat{\mathsf{F}}$, we can abuse notation and write $\widehat{\mathsf{F}}.\mathsf{check}(\mathsf{k}, \mathsf{H}) = \mathsf{F}_1.\mathsf{check}(\mathsf{k}, \mathsf{H}) \| \ldots \| \mathsf{F}_d.\mathsf{check}(\mathsf{k}, \mathsf{H}) \in \mathbb{Z}_p^d$. Likewise, we write $\boldsymbol{F} = \mathsf{Matrix}(\widehat{\mathsf{F}}) \in \mathbb{Z}_p^{N \times d}$ as the matrix representation of this filter. Note that this construction has an identical false-positive rate to a single filter design, as long as we maintain the same fpt_ϵ function used in $\mathsf{BFF_p}$.

4.2 Comparison With Cuckoo Filters

As discussed in in [40], Cuckoo hashing [56] and their filter-variants [35] represent an instantiation of the paradigm introduced by Binary Fuse filters, where $k = 2$ and there are no segments. Cuckoo filters have seen many applications in cryptographic literature, e.g. in private set intersection [64, 65], encrypted search [61], PIR [74], and beyond. For comparing Cuckoo filters wuth Binary Fuse filters, we focus on the size of ç, i.e. the blow-up of the size of the filter compared with the original database, since the number of hash function evaluations is unlikely to make a difference. As is noted in [40], each entry of the cuckoo filter requires an extra 3 bits of representation, and there is a factor of ç = 1.047 to magnify the size by. While ç is concretely smaller in the case of Cuckoo filters, the requirement for holding 3 extra bits per filter entry complicates matters for our PIR scheme, resulting in adding $3d$ bits to the width of the eventual filter matrix. As is shown in [40], in the end Cuckoo filters concretely require more space to represent datasets than Binary Fuse filters. As a result, Binary Fuse filters would appear to represent a non-trivial improvement that could find use-cases in other cryptographic primitives and protocols.

5 KEYWORD PIR CONSTRUCTION

We now describe our Keyword PIR construction, KWPIR, for a generic key-value filter for elements in $\mathbb{Z}_p$. We assume that the server is initialised with a key-value map, KV containing m keys, that clients would like to query. Furthermore, we consider the value space to be $\mathcal{X} = \{0,1\}^w$, and we set $d = (\mu + w)/\log p$, where $\mathsf{hash} : \{0,1\}^* \mapsto \{0,1\}^\mu$, for $\mu = 2\epsilon$. Finally, we use a concatenated filter design $\mathsf{F} = (\mathsf{F}_1, \ldots, \mathsf{F}_d)$, with a $\mathsf{setupFilter}()$ algorithm that returns (F, H), that allows us to encode $(\mu+w)$-bit elements in total. Furthermore, let LWEPIR be an LWE-based PIR scheme (Section 2.4), and let Σ_{lwe} be an LWE-based HEIP scheme (Section 2.2). An algorithmic description of KWPIR is as follows.

- $\mathsf{pp}_{\mathsf{KV}} \leftarrow \mathsf{KWPIR.setup}(1^\lambda, \mathsf{KV})$: Computes $b \leftarrow \mathsf{F.write}(\mathsf{KV}, \mathsf{H})$ and checks that $b = 1$. Otherwise, it retries $\mathsf{F.write}$ a finite number of occasions and aborts if b is never set to 1. Finally, it runs the following setup algorithm $\mathsf{pp}_{\mathsf{LWE}} \leftarrow \mathsf{LWEPIR.setup}(1^\lambda, \mathsf{F})$[7], and returns $\mathsf{pp}_{\mathsf{KV}} = (\mathsf{pp}_{\mathsf{LWE}}, \mathsf{H})$.
- $(\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{KWPIR.query}(\mathsf{pp}_{\mathsf{KV}}, \mathsf{k})$: Runs $(\mathsf{h}_1, \ldots, \mathsf{h}_k) \leftarrow \mathsf{H}(\mathsf{k})$, and then lets $\boldsymbol{f}_{\mathsf{H}(\mathsf{k})} = (f_1, \ldots, f_m)$ be the vector where $f_i = 1$ if and only if $i \in \mathsf{H}(\mathsf{k})$, and is 0 otherwise. Finally, returns $(\mathsf{q}, \mathsf{st}) \leftarrow \mathsf{LWEPIR.query}(\mathsf{pp}_{\mathsf{LWE}}, \boldsymbol{f}_{\mathsf{H}(\mathsf{k})})$.
- $\mathsf{r} \leftarrow \mathsf{KWPIR.respond}(\mathsf{pp}_{\mathsf{KV}}, \mathsf{KV}, \mathsf{q})$: Let $\boldsymbol{F} \leftarrow \mathsf{Matrix}(\mathsf{F}) \in \mathbb{Z}_p^{N \times d}$, and return $\mathsf{r} \leftarrow \mathsf{LWEPIR.respond}(\mathsf{pp}_{\mathsf{LWE}}, \boldsymbol{F}, \mathsf{q})$.
- $\mathsf{x} \leftarrow \mathsf{KWPIR.process}(\mathsf{pp}_{\mathsf{KV}}, \mathsf{st}, \mathsf{r})$: Runs the function $\mathsf{x} \leftarrow \mathsf{LWEPIR.process}(\mathsf{pp}_{\mathsf{LWE}}, \mathsf{st}, \mathsf{r})$, and returns x.

Correctness of KWPIR. The correctness argument for KWPIR follows as a consequence of choosing F to be a Binary Fuse Filter for $\mathbb{Z}_p$, and where LWEPIR is parametrised using the parameters q, p, n, σ, and N.

Theorem 5.1 (Correctness of KWPIR). *Let F be a Binary Fuse Filter for $\mathbb{Z}_p$, and let LWEPIR be a correct PIR for index-based queries for generic databases $\mathsf{DB} \in \mathbb{Z}_p^{N \times w}$, with LWE parameters q, p, n, σ, and N. Then KWPIR is a correct PIR scheme for making keyword queries against KV.*

Proof. The proof of correctness almost follows immediately from the LWEPIR scheme — based on the correct parametrisation of q, p, and σ with respect to the size N of the filter matrix — and the false-positive probability ϵ. The difference is that $\boldsymbol{f}_{\mathsf{H}(\mathsf{k})}$ is an indicator vector where k values are set to 1, rather than only a single element. Following this, the eventual decryption x is equal to a sum of the form:

$$\begin{aligned} &\boldsymbol{F}[\mathsf{h}_1(\mathsf{k})] + \ldots + \boldsymbol{F}[\mathsf{h}_k(\mathsf{k})] \mod p \\ &= \mathsf{F}[\mathsf{h}_1(\mathsf{k})] + \ldots + \mathsf{F}[\mathsf{h}_k(\mathsf{k})] \mod p \\ &= \mathsf{F.reconstruct}(\mathsf{k}, \mathsf{H}) \\ &= \mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}), \end{aligned}$$

which corresponds to the desired result. □

False-positives and larger data elements. Note explicitly that PIR correctness only considers the case where a query is made for an element that belongs to F. By definition, the filter responds to such queries with 100% accuracy, in other words false-negatives cannot occur. We do not prove any property for false-positives occurring in the PIR scheme, and so we do not have to consider the impact of false-positives occurring in the formal definition. However, speaking concretely on the possibility of a false-positive occurring, let us consider the encoding of values as $\mathsf{k} \| \mathsf{x}$. In this regime, it would be necessary to find a key k_0, such that $\mathsf{F.reconstruct}(\mathsf{k}_0, \mathsf{H})$ returns $\mathsf{k}_0 \| \mathsf{x}_0$, for some value x_0, where $\mathsf{k}_0 \notin \mathsf{KV}$. Since fpt_ϵ is a universal hash function in $\mathbb{Z}_p$, this occurs with probability $2^{-\kappa}$.

This concrete estimation relies on encoding the entire data value in a single entry of the filter, which may harm performance. As mentioned in Section 4.1, it is trivial to adapt KWPIR to consider longer data elements without increasing the size of p, by using a concatenated filter regime.

Security of KWPIR. The security argument for KWPIR follows generically as a consequence of LWEPIR being a secure PIR for standard index-based databases.

Theorem 5.2 (Security of KWPIR). *Let LWEPIR be a secure PIR scheme satisfying ℓ-query indistinguishability (ℓ-$\mathsf{QIND}^{\mathcal{A}}_{\mathsf{PIR}}$) for index-based queries, for generic databases $\mathsf{DB} \in \mathbb{Z}_p^{N \times d}$, with LWE parameters q, p, n, σ, and N. Let KV be a key-value map, containing $m = N/k$ elements, represented using a filter, F, and a set of k hashes H. Then KWPIR is (ℓ/k)-$\mathsf{kwQIND}^{\mathcal{B}}_{\mathsf{PIR}}$ secure for making keyword queries against F, based on the hardness of $\mathsf{LWE}_{q,n,p,\sigma}$.*

Proof. Let $\mathcal{A}$ be a PPT adversary in the ℓ-$\mathsf{QIND}^{\mathcal{A}}_{\mathsf{PIR}}$ experiment, where $\mathsf{PIR} = \mathsf{LWEPIR}$, and likewise let $\mathcal{B}$ be a PPT adversary in the (ℓ/k)-$\mathsf{kwQIND}^{\mathcal{B}}_{\mathsf{PIR}}$ security game that $\mathcal{A}$ runs as a subroutine. After initialisation, $\mathcal{B}$ produces their lists of keys $(\mathsf{k}_1, \ldots, \mathsf{k}_{\ell/k})$ and $\mathsf{k}'_1, \ldots, \mathsf{k}'_{\ell/k}$ to be queried, and sends these to $\mathcal{A}$. Then, $\mathcal{A}$ runs:

$$(i_{\iota,1}, \ldots, i_{\iota,k}) \leftarrow \mathsf{F.check}(\iota, \mathsf{H}), \quad (j_{\iota,1}, \ldots, j_{\iota,k}) \leftarrow \mathsf{F.check}(\iota, \mathsf{H}),$$

for each $\iota \in [\ell/k]$, and concatenates these lists into two lists of length ℓ, of the form:

$$(i_{1,1}, \ldots, i_{1,k}, i_{2,1}, \ldots, i_{\ell/k,k}), \quad (j_{1,1}, \ldots, j_{1,k}, j_{2,1}, \ldots, j_{\ell/k,k}).$$

Then, $\mathcal{A}$ submits both of these lists to the challenger in the ℓ-$\mathsf{QIND}^{\mathcal{A}}_{\mathsf{PIR}}$ experiment, and learns a list of ℓ queries $\boldsymbol{Q} = (\mathsf{q}_1, \ldots, \mathsf{q}_\ell)$. By the nature of LWEPIR, each q_l (for $l \in [\ell]$) is an LWE-based ciphertext of the form described in Section 2.2. Therefore, $\mathcal{A}$ breaks $\boldsymbol{Q}$ into ℓ/k contiguous segments of length k, where we write $\boldsymbol{Q}_\iota = (\mathsf{q}_{\iota,1}, \ldots, \mathsf{q}_{\iota,k})$ to denote the segment containing $(\mathsf{q}_{(\iota-1)\cdot(\ell/k)+1}, \ldots, \mathsf{q}_{(\iota-1)\cdot(\ell/k)+k})$, for $\iota \in [k]$. Then, $\mathcal{A}$ runs $\widetilde{\mathsf{q}}_\iota \leftarrow \Sigma_{\mathsf{lwe}}.\mathsf{eval}(\boldsymbol{Q}_\iota, \boldsymbol{1}_k)$, for each $\iota \in [\ell/k]$, where $\boldsymbol{1}_k$ is the k-dimensional all-one vector — in other words, performing a homomorphic sum of each of the ciphertexts. Finally, $\mathcal{A}$ returns $\widetilde{\boldsymbol{Q}} = (\widetilde{\mathsf{q}}_1, \ldots, \widetilde{\mathsf{q}}_{\ell/k})$ to $\mathcal{B}$. When $\mathcal{B}$ returns b' to $\mathcal{A}$, $\mathcal{A}$ simply forwards b' to their challenger.

We now show that $\mathcal{A}$ simulates the (ℓ/k)-$\mathsf{kwQIND}^{\mathcal{B}}_{\mathsf{PIR}}$ game perfectly for $\mathcal{B}$. This amounts to showing that $\widetilde{\boldsymbol{Q}}$ is a list of queries for the keywords submitted by $\mathcal{B}$, corresponding to $(\mathsf{k}_1, \ldots, \mathsf{k}_{\ell/k})$ when $b = 0$, and $(\mathsf{k}'_1, \ldots, \mathsf{k}'_{\ell/k})$ when $b = 1$.

[7]For the purpose of making this function call, F is interpreted as a standard database containing N elements in $\mathbb{Z}_p$.

Without loss of generality, let us consider the case of $b = 0$. Notice that $\widetilde{\mathsf{q}}_l$ is an encryption of the vector $f_{\mathsf{H}(\mathsf{k})}$. This is because $\widetilde{\mathsf{q}}_l$ results from the homomorphic evaluation of the sum of $(\mathsf{q}_{(l-1)\cdot(\ell/k)+1}, \ldots, \mathsf{q}_{(l-1)\cdot(\ell/k)+k})$, where $\mathsf{q}_{(l-1)\cdot(\ell/k)+t}$ is a query that encrypts $f_{\mathsf{h}_t(\mathsf{k})}$ — in other words, the all-zero vector with a 1 in position $\mathsf{h}_t(\mathsf{k})$ — for $t \in [k]$. Furthermore, we know that this homomorphic evaluation is correct, because Σ_{lwe} is parameterised to be correct for databases of size m, while the map KV only contains N/k elements (and thus requires N/k homomorphic operations). Therefore, the extra homomorphic computations performed by $\mathcal{A}$ will not violate correctness when each query is applied to the database associated with KV. Note that, by the definition of KWPIR, $\widetilde{\mathsf{q}}_l$ is a ciphertext encrypting $\mathsf{LWEPIR.query}(\mathsf{pp}_{\mathsf{LWE}}, f_{\mathsf{H}(\mathsf{k})})$, which is equivalent to $\mathsf{KWPIR.query}(\mathsf{pp}_{\mathsf{KV}}, \mathsf{k})$. The argument above holds identically in the case that $b = 1$. Therefore, the simulation produced by $\mathcal{A}$ corresponds exactly to the real game in (ℓ/k)-$\mathsf{kwQIND}^{\mathcal{B}}_{\mathsf{PIR}}$.

Now, consider that the possibility that $\mathcal{B}$ has non-negligible advantage, then this translates directly into a non-negligible advantage for $\mathcal{A}$ in ℓ-$\mathsf{QIND}^{\mathcal{A}}_{\mathsf{PIR}}$, since each of the queries submitted by $\mathcal{A}$ are valid against a database of size N. Therefore, given that LWEPIR is a secure PIR scheme based on the $\mathsf{LWE}_{q,n,p,\sigma}$ assumption, we must conclude that $\mathcal{B}$ has negligible advantage similarly. □

5.1 Square-root Matrix Encoding

By fixing the matrix description $\boldsymbol{F} \in \mathbb{Z}_p^{N\times d}$ of the concatenated filter, we immediately align KWPIR with the usage of the FrodoPIR scheme [29] for implementing the LWEPIR scheme. This is because, we effectively treat each row in the filter "database" as a single element. However, we can easily express KWPIR in terms of an LWEPIR scheme that allows "square-root" database encoding as well. To achieve this, we simply modify the filter matrix representation $\boldsymbol{F}$ to encode multiple *rows* of the filter on a single row. This way we can achieve $\boldsymbol{F} \in \mathbb{Z}_p^{\sqrt{N}\times\sqrt{N}}$ which results in asymptotically smaller communication overheads (which we discuss shortly). Then, when querying against $\boldsymbol{F}$, the task is simply to decode only the elements of the response that correspond to the desired columns of $\boldsymbol{F}$. Thus, the actual filter representation does not have to change at all.

6 PERFORMANCE EVALUATION

In this section, we discuss parameter settings, implementation details and experimental evaluation of the KWPIR protocol.

6.1 Implementation

PIR scheme. As previously discussed, we can use any underlying LWEPIR scheme to implement KWPIR. While the choice of scheme impacts bandwidth costs (due to differences in database encoding), the actual online runtimes are largely equivalent. This is due to the fact that the online phase always needs to run $O(N \cdot d)$ operations regardless of the database format. We emphasise here that we focus on detailing the costs of a simple implementation that will more readily usable by non-expert developers, and we ignore the possibility of using optimised matrix multiplication algorithms.

Due to these reasons, for our implementation and instantiation, we decide to use the FrodoPIR Rust implementation[8] as a base for our own implementation.[9] We call this instantiation **ChalametPIR**. Our changes to the FrodoPIR codebase include adding API support for keyword databases, and incorporating an adapted Rust implementation of Binary Fuse filters.[10] Our adaptations of Binary Fuse filters include handling of plaintext operations in $\mathbb{Z}_p$, and modifying the algorithm to work with key-value maps. For bandwidth costs, we provide cost calculations for ChalametPIR instantiated using both FrodoPIR [29] and SimplePIR [42] as the underlying LWEPIR scheme. We further benchmark the offline and online phases of ChalametPIR, taking into account both underlying approaches. We highlight explicitly how the modification of the database format in FrodoPIR and SimplePIR can result in interesting performance trade-offs. Since the number of offline and online server operations should be equivalent in both cases (modulo modification of LWE parameters), we provide runtimes only using our FrodoPIR-based implementation

Experimental parameters. For our FrodoPIR-based implementation [29] we consider an LWE dimension of $n = 1774$, modulus $q = 2^{32}$ to provide $\lambda = 128$ bits of security [2]. For SimplePIR, we use $q = 2^{32}$ and $n = 1024$. To establish the size of p, the plaintext modulus, in FrodoPIR, we must take into account the database size m. Primarily, we consider key-value map sizes of $2^{16} \le m \le 2^{20}$, where $p = 2^{10}$ for $m \in \{2^{16}, 2^{17}, 2^{18}\}$, and $p = 2^9$ for $m \in \{2^{19}, 2^{20}\}$. For these experiments, we set the size of each value entry (of the form $\mathsf{hash}(\mathsf{k})\|\mathsf{x}$) as simulated to be 1 KB ($w = 2^{13}$ bits) in length. The choice of p when using SimplePIR is simulated using the open-source code of [42].[11] To compare with existing Keyword PIR schemes [51, 62], we additionally experiment with three databases of the form:

- $m = 2^{20}$, $w = 2^{11}$ (256 B), and $p = 2^9$;
- $m = 2^{17}$, $w = 30 \cdot 2^{13}$ (30 kB), and $p = 2^9$;
- $m = 2^{14}$, $w = 100 \cdot 2^{13}$ (100 kB), and $p = 2^9$.

Finally, for the parameters of BFF_p, we consider both cases of $k = 3$ and $k = 4$, where k is the number of hash functions. When calculating the number of entries in the filter for these cases, we set $\varsigma = 1.13$ and $\varsigma = 1.08$, respectively. In some cases we provide only benchmarks for $k = 3$, which provides a lower bound on the efficiency of the approach in terms of bandwidth and server computation.

Computational setup and financial costs. For estimating the runtime of ChalametPIR and maintaining the comparisons as fair as possible with previous work, we use two AWS EC2 instances almost identical to the ones used in [62]: (i) Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz, 32GiB of memory (referred in AWS EC2 as "t2.2xlarge"), and (ii) Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz, 72GiB of memory and

[8]https://github.com/brave-experiments/frodo-pir
[9]https://github.com/claucece/chalamet
[10]Original code: https://github.com/ayazhafiz/xorf
[11]https://github.com/ahenzinger/simplepir

36vCPU (referred in AWS EC2 as "c5.9xlarge").[12] In particular, we use the "t2.2xlarge" machine to compare performance with existing index-based LWE PIR schemes, and we use the "c5.9xlarge" machine to compare with the SparsePIR keyword PIR scheme [62]. In addition, we provide benchmarks using a Macbook M1 Max, to highlight how efficient operations are when they run on commodity hardware. All of our experiments use single-thread execution and results are taken as the average of 100 runs.

In terms of other performance metrics, we use the current AWS financial cost structure for running a server in the "c5.9xlarge" [7]. Therefore, the CPU per-hour cost is estimated as \$1.53/36 = \$0.0425 (since this machine has 36 vCPUs, and we run single-threaded), the download cost is \$0.09 per GB, and the upload cost is zero. Furthermore, following prior works, we define the *rate* as the ratio of the retrieved record size to the response size, and the *throughput* as the ratio of the database size to the server's online computation time.

6.2 Experimental analysis

Here, we describe the online runtime and bandwidth costs of ChalametPIR. Offline costs are not negligible but amortise linearly either globally or per-client, and differ greatly depending on the LWEPIR scheme that is used (we discuss this in Section 7). We describe offline costs for ChalametPIR based on both FrodoPIR and SimplePIR in Appendix B.

Bandwidth. The bandwidth costs for ChalametPIR are given in Table 1. Clearly, the query and response are far more balanced in the case of SimplePIR as opposed to FrodoPIR. As previously alluded, FrodoPIR optimises for the download as this results in reduced financial costs when running the server functionality on standard cloud architectures (since upload costs are typically free). See Table 3 for more details.

Regardless, the total costs are fairly small, requiring data transfer in the order of kilobytes to perform a keyword query. In the case of using FrodoPIR as the underlying PIR scheme, the advantage of having a smaller download is that the *rate* is ≈ 0.3. In other words, the size of the response ciphertext is only $\approx 3\times$ larger than the original record. In Figure 3 we highlight that the bandwidth blow-up introduced by adding the keyword functionality to the underlying index-based LWEPIR scheme is minimal. In other words, performing PIR over the Binary Fuse Filter description results in only a small magnification of both the query and response sizes, when compared with an indexed array (with no keyword query functionality).

Runtimes. The runtimes of ChalametPIR are minimal in all cases, client operations (query and parsing) largely require only a small number of milliseconds of computation in all cases. The server response computation only requires more than a second for the $2^{17} \times 100$ kB database size. Otherwise, it is at most hundreds of milliseconds. Since these times are achieved using single-threaded processing, and given that the computation is a series of independent matrix-vector multiplications, it seems natural that parallelisation would significantly reduce these times further. In addition, these times do not take into account any optimisations that could be introduced with sub-cubic matrix multiplication formulae [25, 71]. Finally, by looking again at Figure 3, we see that the blowup in runtimes introduced by the keyword functionality

[12]Note that [62] used a machine with 64GB of RAM.

$\lvert KV \rvert$	# keys × |value| ($m \times w$)	**Query** (kB) $k=3$	**Query** (kB) $k=4$	**Response** (kB) $k=3$	**Response** (kB) $k=4$
	LWEPIR = FrodoPIR				
$m \uparrow$	$2^{16} \times 1$ kB	287	276	3.2	3.2
	$2^{17} \times 1$ kB	579	553	3.2	3.2
	$2^{18} \times 1$ kB	1157	1106	3.2	3.2
	$2^{19} \times 1$ kB	2314	2212	3.56	3.56
	$2^{20} \times 1$ kB	4628	4424	4.56	4.56
$w \uparrow$	$2^{20} \times 256$ B	4628	4424	0.89	0.89
	$2^{17} \times 30$ kB	579	553	96	96
	$2^{14} \times 100$ kB	72	69	291	291
	LWEPIR = SimplePIR				
$m \uparrow$	$2^{16} \times 1$ kB	31.89	31.17	31.89	31.17
	$2^{17} \times 1$ kB	44.65	43.64	44.65	43.64
	$2^{18} \times 1$ kB	63.78	62.34	63.78	62.34
	$2^{19} \times 1$ kB	90.36	88.32	90.36	88.32
	$2^{20} \times 1$ kB	127.56	124.68	127.56	124.68
$w \uparrow$	$2^{20} \times 256$ B	63.78	62.34	63.78	62.34
	$2^{17} \times 30$ kB	256.18	250.4	256.18	250.4
	$2^{14} \times 100$ kB	180.71	180.71	176.63	176.63

Table 1: Bandwidth costs (kB) for KWPIR.

	DB ($m \times w$)	Query	Response	Parsing
Macbook M1 Max	$2^{16} \times 1024$ B	0.010597	6.5508	0.22001
	$2^{17} \times 1024$ B	0.038866	12.473	0.21894
	$2^{18} \times 1024$ B	0.051996	24.452	0.21658
	$2^{19} \times 1024$ B	0.14442	54.053	0.24204
	$2^{20} \times 1024$ B	0.24049	116.89	0.24384
EC2 "t2.t2xlarge"	$2^{16} \times 1024$ B	0.050048	37.830	0.47251
	$2^{17} \times 1024$ B	0.1787	74.733	0.47046
	$2^{18} \times 1024$ B	0.19739	143.82	0.46782
	$2^{19} \times 1024$ B	0.4219	319.82	0.50735
	$2^{20} \times 1024$ B	0.8471	634.21	0.56381
EC2 "c5.9xlarge"	$2^{20} \times 256$ B	1.3699	133.58	0.090116
	$2^{17} \times 30$ kB	0.055415	1846.6	10.663
	$2^{14} \times 100$ kB	0.0040465	760.64	35.485

Table 2: Online performance (milliseconds) of ChalametPIR (LWEPIR = FrodoPIR, $k=3$) on Macbook M1 Max, and AWS EC2 't2.t2xlarge' and 'c5.9xlarge'. Response is a server operation, while Query and Parsing are run by the client.

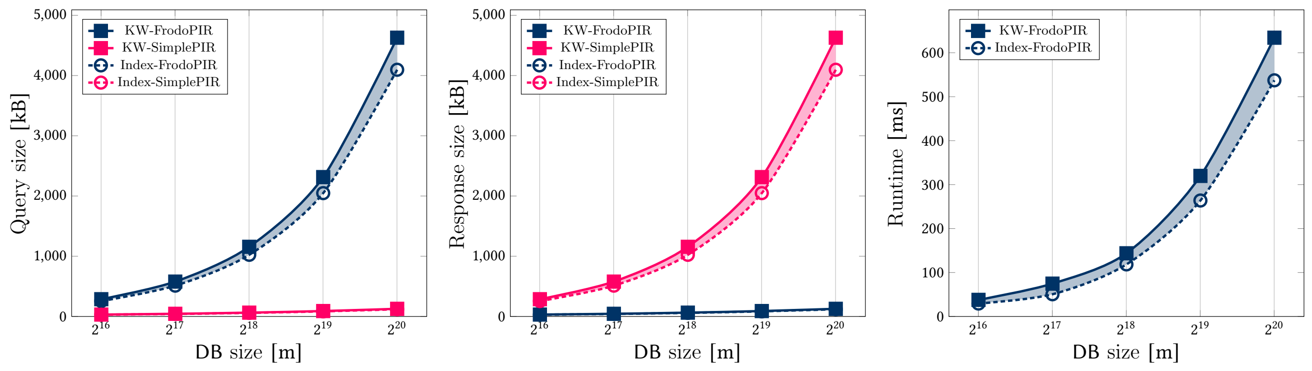

Figure 3: Comparison of online costs (query/response sizes and runtime) for ChalametPIR with index-based LWEPIR $\in$ {FrodoPIR, SimplePIR} schemes. We refer to index-based schemes with "Index" and keyword-based ones with "KW". Note that the values of Index-based and KW-based SimplePIR are so similar that they appear superimposed in the first and second figures.

via Binary Fuse Filter usage is minimal, when compared with the runtimes of the underlying LWEPIR scheme. To ensure that this comparison was accurate, we reran the FrodoPIR experiments and based our comparison on the runtimes given in Table 5 in Appendix B.

6.3 Keyword PIR performance comparison

The schemes of [62] and [51] represent the most efficient single-server keyword PIR protocols to date. In particular, the SparsePIR scheme of [62] represents the state-of-the-art in terms of performance (both communication and runtimes).

In Table 3, we compare ChalametPIR — instantiated with LWEPIR $\in$ {FrodoPIR, SimplePIR} — against SparsePIR — instantiated with both the Onion [55] ("OnionSparsePIR") and Spiral [52] ("SpiralSparsePir") PIR schemes. Since [62] does not provide an open-source implementation of their work, we report the numbers given in their paper. For these comparisons, we use Intel(R) Xeon(R) Platinum 8124M CPU @ 3.00GHz, 72GiB of memory and 36vCPU (referred in AWS EC2 as "c5.9xlarge"), and was the closest that we could find to the machine used in [62].[13] Note that the runtimes of [62] are given with specific AVX2 and AVX-512 instruction sets with SIMD instructions enabled, while we do not use such optimisations. Finally, we provide server runtimes of ChalametPIR using our FrodoPIR-based implementation[14]. We assume equivalent runtimes for a SimplePIR-based implementation, since the number of operations is the same, modulo difference in their choice of LWE security parameters. To simplify the rest of the comparison, we refer to the benchmarked map taking sizes in $(2^{20} \times 256\,\mathrm{B})$, $(2^{17} \times 30\,\mathrm{kB})$, $(2^{14} \times 100\,\mathrm{kB})$ as case I, case II, and case III, respectively.

In terms of runtimes, ChalametPIR is an order of magnitude quicker in case I, with speed-up factors of $6.25\times$ and $7.78\times$ for cases II and III, respectively. This leads to a significant improvement in *throughput*, processing $\sim 2\,\mathrm{GB}$ of data per second, while SparsePIR achieves only hundreds of MB.

In terms of client download, ChalametPIR with FrodoPIR excels in the setting where the elements are smallest, since the client download is dependent only on the parameter w. For case I, this configuration is $> 23\times$ more efficient than SparsePIR, with a download size of $< 1\,\mathrm{kB}$. For case II, we can already see that SpiralSparsePIR is competitive with ChalametPIR with FrodoPIR, achieving similar download costs. Finally, for case III, ChalametPIR with SimplePIR achieves the lowest bandwidth cost across the board, while FrodoPIR has a comparatively larger cost, due to w being larger. We can see that these trends are represented in the rate also, since this is determined by the ratio of the ciphertext download size with respect to the retrieved element. In terms of client upload, since FrodoPIR naturally favours optimising download instead of upload, FrodoPIR-based ChalametPIR performs poorly across all cases. In contrast, SparsePIR provides the best trade-off in terms of upload across all cases.

Finally, in terms of AWS EC2 financial costs, ChalametPIR is by far the cheapest PIR scheme to use across all database sizes. Concretely, a server implementation of ChalametPIR with either LWEPIR scheme results in between $3\times$—$11.4\times$ and cost savings. This cost metric is important for general application providers, that do not have native access to hardware to run server-side software, and must resort to using commercial cloud-computing infrastructure. We show in Appendix B that the offline costs of ChalametPIR have a largely insignificant impact compared with the per-query online costs, when considering amortisation over even moderate client usage, and improving further still for widely-used systems.

Constant-weight PIR. As was shown in [62], the constant-weight PIR scheme of [51] performs much more slowly and with much larger bandwidth constraints than the SparsePIR approach, even for much smaller databases. We attempted to acquire results for each of the larger cases ourselves, but were unable to get full performance figures from the provided implementation[15], due to the experiments failing to terminate. Given that [62] shows SparsePIR is over an order of magnitude

[13] Ubuntu PC, 3.7 GHz Intel Xeon W-2135, 12-core CPU, 64 GB RAM.
[14] https://github.com/claucece/chalamet

[15] https://github.com/RasoulAM/constant-weight-pir

	ChalametPIR		SparsePIR	
	FrodoPIR	SimplePIR	Onion	Spiral
Online costs: $2^{20} \times 256$ B				
Query (kB)	287	63.78	63	14
Response (kB)	0.89	63.78	127	21
Runtime (s)	0.13358	0.13358†	3.04	1.44
Rate	0.28	0.004	0.002	0.012
Throughput (MB/s)	1916	1916	84	178
Cost (USD)	1.65e−6	7.05e−6	4.68e−5	1.88e−5
Online costs: $2^{17} \times 30$ kB				
Query (kB)	579 kB	256.18 kB	63 kB	14 kB
Response (kB)	96 kB	256.18 kB	127 kB	86 kB
Runtime (s)	1.8466	1.8466†	41.91	11.57
Rate	0.313	0.117	0.236	0.349
Throughput (MB/s)	2218	2218	98	354
Cost (USD)	3e−5	4.37e−5	5.05e−4	1.43e−4
Online costs: $2^{14} \times 100$ kB				
Query (kB)	72 kB	180.71 kB	63 kB	14 kB
Response (kB)	291 kB	176.63 kB	508 kB	242 kB
Runtime (s)	0.76064	0.76064†	17.32	5.91
Rate	0.344	0.566	0.197	0.413
Throughput (MB/s)	2692	2692	118	347
Cost (USD)	3.4e−5	2.41e−5	0.25e−4	9.05e−5

Table 3: Comparison of online costs for ChalametPIR (with LWEPIR $\in$ {FrodoPIR, SimplePIR}, and $k = 3$) with SparsePIR based on both the Onion [55] and Spiral [52] PIR schemes. Server runtimes computed on AWS EC2 'c5.9xlarge' running Ubuntu, and financial costs calculated using the same hardware. $\dagger$: Online runtimes for SimplePIR are estimated as equivalent to FrodoPIR, since the number of operations is essentially equivalent. The usage of green indicates the most optimal cases and light-green the second most-optimal case.

more efficient with respect to nearly all of the performance criteria, it is clear that ChalametPIR would achieve a similar (if not more stark) set of contrasts.

7 DISCUSSION

Applications. Information retrieval that allows for a false positive rate has attracted significant interest in recent years, especially in distributed and database systems, where false positives can be tolerated to a degree, and minimal space usage is crucial [16]. While there have been several proposals to efficiently solve this problem [4, 54], privacy has not been deeply considered. Providing efficient KWPIR is a step forward in solving this lack of consideration. In other areas, KW-PIR are fundamental tools for building *credentials-checking* (C3) services, which check if a username, password pair is exposed in order to prevent *credential-stuffing attacks* or *credential-tweaking attacks* [58], as they are one of the most prevalent forms of account compromise [72]. Naively, index-based PIR solutions to this problem allow for only retrieving breached passwords. A keyword-based solution allows for querying for a specific username, password pair, which can better alert a user of a breach of their credentials. Interesting future work can focus on analysing how important the role of database privacy plays in such problem, and how such guarantees can be imported to the KWPIR setting.

Keyword PIR is also a natural fit for *private pattern matching*: privately identifying occurrences of a given string in text. Specifically, for the "exact" version of the problem: retrieve occurrences where the given query exactly matches a substring in the text. The need for privacy in these cases relies on querying on text that can be considered sensitive information [45]. Adapting our scheme for this problem will need to determine how to properly construct the different structures and parameters, and we leave this extension as future work.

Batch PIR. Batch PIR performs Q PIR queries in a single batch, but where processing and communication costs are concretely smaller than the trivial case of launching Q independent PIR queries. As is noted in [42], LWEPIR schemes naturally are amenable to generic batching techniques introduced in [43], to reduce the total server time from far below $O(QN)$, by partitioning the database into Q chunks, and running independent PIR queries on each of these smaller chunks. Since batch PIR is not the main focus of this work, we encourage the reader to see [42] for more details.

Database updates. As noted in previous works [29, 42], LWE-based PIR approaches do not provide native support for handling database updates, beyond re-running the offline state generation procedure. Standard telescopic database update mechanisms can be applied [44], but devising instantiation-specific approaches represents an interesting open problem.

Alternative LWE PIR Schemes. The DoublePIR [42] and HintlessPIR [48] schemes provide alternative LWEPIR protocols that could be considered in the context of ChalametPIR. In both of these approaches, the central idea is that the square-root matrix encoding means that the client does not need the full offline state to decode online queries. In essence, they can use another layer of PIR to retrieve only the elements in the offline hint that are required. In this paradigm, FrodoPIR and SimplePIR simply represent a trivial solution to download the entire *hint* database. In DoublePIR, the idea is to provide another layer of SimplePIR but where the client queries the hint as the intended database. In HintlessPIR, the idea is that using RLWE-based PIR schemes can lead to performance improvements compared with the aforementioned approaches. However, as we discussed in 6.2, since these changes only impact the offline phase, the results that we represent for the online phase would largely be equivalent in each of the cases.

8 CONCLUSION

In this work, we built a simple framework for constructing Keyword PIR based on state-of-the-art index-based PIR schemes. We refer to this framework as KWPIR, and derive

ChalametPIR as a concrete instantiation of it that is compatible with LWEPIR schemes. The framework makes uses of novel key-value filters (Binary Fuse filters) and arrives to computational and communicational times that are essentially competitive with their index-based counterparts. We implemented ChalametPIR in Rust as a proof-of-concept, and with it illustrate that the scheme is more efficient than state-of-the-art keyword-based schemes.

REFERENCES

[1] Carlos Aguilar Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. 2016. XPIR: Private Information Retrieval for Everyone. *PoPETs* 2016, 2 (April 2016), 155–174.

[2] Martin R. Albrecht, Rachel Player, and Sam Scott. 2015. On the concrete hardness of Learning with Errors. *J. Math. Cryptol.* 9, 3 (2015), 169–203. http://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml

[3] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. 2021. Communication-Computation Trade-offs in PIR, See [8], 1811–1828.

[4] Stephen Alstrup, Gerth Brodal, and Theis Rauhe. 2001. Optimal Static Range Reporting in One Dimension. In *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing* (Hersonissos, Greece) *(STOC '01)*. Association for Computing Machinery, New York, NY, USA, 476–482. https://doi.org/10.1145/380752.380842

[5] Andris Ambainis. 1997. Upper Bound on Communication Complexity of Private Information Retrieval. In *ICALP 97 (LNCS, Vol. 1256)*, Pierpaolo Degano, Roberto Gorrieri, and Alberto Marchetti-Spaccamela (Eds.). Springer, Heidelberg, 401–407. https://doi.org/10.1007/3-540-63165-8_196

[6] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. 2018. PIR with Compressed Queries and Amortized Query Processing. In *2018 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 962–979. https://doi.org/10.1109/SP.2018.00062

[7] AWS. [n. d.]. Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/ https://aws.amazon.com/ec2/pricing/on-demand/. Accessed 18th January 2024..

[8] Michael Bailey and Rachel Greenstadt (Eds.). 2021. *USENIX Security 2021*. USENIX Association.

[9] Richard Beigel, Lance Fortnow, and William Gasarch. 2006. A Tight lower bound for restricted pir protocols. *Computational Complexity* 15 (05 2006), 82–91. https://doi.org/10.1007/s00037-006-0208-3

[10] Amos Beimel and Yuval Ishai. 2001. Information-Theoretic Private Information Retrieval: A Unified Construction. In *ICALP 2001 (LNCS, Vol. 2076)*, Fernando Orejas, Paul G. Spirakis, and Jan van Leeuwen (Eds.). Springer, Heidelberg, 912–926. https://doi.org/10.1007/3-540-48224-5_74

[11] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Ilan Orlov. 2012. Share Conversion and Private Information Retrieval. In *2012 IEEE 27th Conference on Computational Complexity*. 258–268. https://doi.org/10.1109/CCC.2012.23

[12] Amos Beimel, Yuval Ishai, Eyal Kushilevitz, and Jean-François Raymond. 2002. Breaking the $O(n^{1/(2k-1)})$ Barrier for Information-Theoretic Private Information Retrieval. In *43rd FOCS*. IEEE Computer Society Press, 261–270. https://doi.org/10.1109/SFCS.2002.1181949

[13] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[14] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. 2006. An Improved Construction for Counting Bloom Filters. In *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 4168)*, Yossi Azar and Thomas Erlebach (Eds.). Springer, 684–695. https://doi.org/10.1007/11841036_61

[15] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. 2008. On the false-positive rate of Bloom filters. *Inform. Process. Lett.* 108, 4 (2008), 210–213. https://doi.org/10.1016/j.ipl.2008.05.018

[16] Andrei Broder and Michael Mitzenmacher. 2003. Network Applications of Bloom Filters: A Survey. *Internet Mathematics* 1, 4 (2003), 485 – 509.

[17] Christian Cachin, Silvio Micali, and Markus Stadler. 1999. Computationally Private Information Retrieval with Polylogarithmic Communication, See [70], 402–414. https://doi.org/10.1007/3-540-48910-X_28

[18] Luís Caires, Giuseppe F. Italiano, Luís Monteiro, Catuscia Palamidessi, and Moti Yung (Eds.). 2005. *ICALP 2005*. LNCS, Vol. 3580. Springer, Heidelberg.

[19] Yan-Cheng Chang. 2004. Single Database Private Information Retrieval with Logarithmic Communication. In *ACISP 04 (LNCS, Vol. 3108)*, Huaxiong Wang, Josef Pieprzyk, and Vijay Varadharajan (Eds.). Springer, Heidelberg, 50–61. https://doi.org/10.1007/978-3-540-27800-9_5

[20] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier filter: an efficient data structure for static support lookup tables. In *15th SODA*, J. Ian Munro (Ed.). ACM-SIAM, 30–39.

[21] Benny Chor and Niv Gilboa. 1997. Computationally Private Information Retrieval (Extended Abstract). In *29th ACM STOC*. ACM Press, 304–313. https://doi.org/10.1145/258533.258609

[22] Benny Chor, Niv Gilboa, and Moni Naor. 1998. Private Information Retrieval by Keywords. Cryptology ePrint Archive, Report 1998/003. https://eprint.iacr.org/1998/003.

[23] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. 1995. Private Information Retrieval. In *36th FOCS*. IEEE Computer Society Press, 41–50. https://doi.org/10.1109/SFCS.1995.492461

[24] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. 2023. Authenticated private information retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3835–3851. https://www.usenix.org/conference/usenixsecurity23/presentation/colombo

[25] Don Coppersmith and Shmuel Winograd. 1990. Matrix Multiplication via Arithmetic Progressions. *J. Symb. Comput.* 9, 3 (mar 1990), 251–280. https://doi.org/10.1016/S0747-7171(08)80013-2

[26] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. 2022. Single-Server Private Information Retrieval with Sublinear Amortized Time. In *EUROCRYPT 2022, Part II (LNCS, Vol. 13276)*, Orr Dunkelman and Stefan Dziembowski (Eds.). Springer, Heidelberg, 3–33. https://doi.org/10.1007/978-3-031-07085-3_1

[27] Henry Corrigan-Gibbs and Dmitry Kogan. 2020. Private Information Retrieval with Sublinear Online Time. In *EUROCRYPT 2020, Part I (LNCS, Vol. 12105)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 44–75. https://doi.org/10.1007/978-3-030-45721-1_3

[28] Alex Davidson and Carlos Cid. 2017. An Efficient Toolkit for Computing Private Set Operations. In *ACISP 17, Part II (LNCS, Vol. 10343)*, Josef Pieprzyk and Suriadi Suriadi (Eds.). Springer, Heidelberg, 261–278.

[29] Alex Davidson, Gonçalo Pestana, and Sofía Celi. 2023. FrodoPIR: Simple, Scalable, Single-Server Private Information Retrieval. *PoPETs* 2023, 1 (Jan. 2023), 365–383. https://doi.org/10.56553/popets-2023-0022

[30] Fan Deng and Davood Rafiei. 2006. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis (Eds.). ACM, 25–36. https://doi.org/10.1145/1142473.1142477

[31] Peter C. Dillinger and Stefan Walzer. 2021. Ribbon filter: practically smaller than Bloom and Xor. *CoRR* abs/2103.02515 (2021). arXiv:2103.02515 https://arxiv.org/abs/2103.02515

[32] Changyu Dong and Liqun Chen. 2014. A Fast Single Server Private Information Retrieval Protocol with Low Communication Cost. In *ESORICS 2014, Part I (LNCS, Vol. 8712)*, Miroslaw Kutylowski and Jaideep Vaidya (Eds.). Springer, Heidelberg, 380–399. https://doi.org/10.1007/978-3-319-11203-9_22

[33] Changyu Dong, Liqun Chen, and Zikai Wen. 2013. When private set intersection meets big data: an efficient and scalable protocol. In *ACM CCS 2013*, Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung (Eds.). ACM Press, 789–800. https://doi.org/10.1145/2508859.2516701

[34] Zeev Dvir and Sivakanth Gopi. 2016. 2-Server PIR with Subpolynomial Communication. *J. ACM* 63, 4 (2016), 39:1–39:15. https://doi.org/10.1145/2968443

[35] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies* (Sydney, Australia) *(CoNEXT '14)*. Association for Computing Machinery, New York, NY, USA, 75–88. https://doi.org/10.1145/2674005.2674994

[36] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. 2005. Keyword Search and Oblivious Pseudorandom Functions. In *TCC 2005 (LNCS, Vol. 3378)*, Joe Kilian (Ed.). Springer, Heidelberg, 303–324. https://doi.org/10.1007/978-3-540-30576-7_17

[37] Craig Gentry and Zulfikar Ramzan. 2005. Single-Database Private Information Retrieval with Constant Communication Rate, See [18], 803–815. https://doi.org/10.1007/11523468_65

[38] Niv Gilboa and Yuval Ishai. 2014. Distributed Point Functions and Their Applications. In *EUROCRYPT 2014 (LNCS, Vol. 8441)*, Phong Q. Nguyen and Elisabeth Oswald (Eds.). Springer, Heidelberg, 640–658. https://doi.org/10.1007/978-3-642-55220-5_35

[39] Thomas Mueller Graf and Daniel Lemire. 2020. Xor Filters. *ACM J. Exp. Algorithmics* 25 (2020), 1–16. https://doi.org/10.1145/3376122

[40] Thomas Mueller Graf and Daniel Lemire. 2022. Binary Fuse Filters: Fast and Smaller Than Xor Filters. *ACM J. Exp. Algorithmics* 27 (2022), 1.5:1–1.5:15. https://doi.org/10.1145/3510449

[41] Carmit Hazay and Martijn Stam (Eds.). 2023. *EUROCRYPT 2023, Part I*. LNCS, Vol. 14004. Springer, Heidelberg.

[42] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. 2023. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3889–3905. https://www.usenix.org/conference/usenixsecurity23/presentation/henzinger

[43] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. 2004. Batch codes and their applications. In *36th ACM STOC*, László Babai (Ed.). ACM Press, 262–271. https://doi.org/10.1145/1007352.1007396

[44] Dmitry Kogan and Henry Corrigan-Gibbs. 2021. Private Blocklist Lookups with Checklist, See [8], 875–892.

[45] Vladimir Kolesnikov, Mike Rosulek, and Ni Trieu. 2018. SWiM: Secure Wildcard Pattern Matching from OT Extension. In *FC 2018 (LNCS, Vol. 10957)*, Sarah Meiklejohn and Kazue Sako (Eds.). Springer, Heidelberg, 222–240. https://doi.org/10.1007/978-3-662-58387-6_12

[46] Eyal Kushilevitz and Rafail Ostrovsky. 1997. Replication is NOT Needed: SINGLE Database, Computationally-Private Information Retrieval. In *38th FOCS*. IEEE Computer Society Press, 364–373. https://doi.org/10.1109/SFCS.1997.646125

[47] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. 2021. Private Join and Compute from PIR with Default. In *ASIACRYPT 2021, Part II (LNCS, Vol. 13091)*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer, Heidelberg, 605–634. https://doi.org/10.1007/978-3-030-92075-3_21

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

[49] Helger Lipmaa. 2005. An Oblivious Transfer Protocol with Log-Squared Communication. In *ISC 2005 (LNCS, Vol. 3650)*, Jianying Zhou, Javier Lopez, Robert H. Deng, and Feng Bao (Eds.). Springer, Heidelberg, 314–328.

[50] Yiping Ma, Ke Zhong, Tal Rabin, and Sebastian Angel. 2021. Incremental Offline/Online PIR (extended version). Cryptology ePrint Archive, Report 2021/1438. https://eprint.iacr.org/2021/1438.

[51] Rasoul Akhavan Mahdavi and Florian Kerschbaum. 2022. Constant-weight PIR: Single-round Keyword PIR via Constant-weight Equality Operators. In *USENIX Security 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 1723–1740.

[52] Samir Jordan Menon and David J. Wu. 2022. SPIRAL: Fast, High-Rate Single-Server PIR via FHE Composition. In *2022 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, 930–947. https://doi.org/10.1109/SP46214.2022.9833700

[53] Michael Mitzenmacher. 2002. Compressed bloom filters. *IEEE/ACM Trans. Netw.* 10, 5 (2002), 604–612. https://doi.org/10.1109/TNET.2002.803864

[54] Christian Worm Mortensen, Rasmus Pagh, and Mihai Pundefinedtraçcu. 2005. On Dynamic Range Reporting in One Dimension. In *Proceedings of the Thirty-Seventh Annual ACM Symposium on Theory of Computing* (Baltimore, MD, USA) *(STOC '05)*. Association for Computing Machinery, New York, NY, USA, 104–111. https://doi.org/10.1145/1060590.1060606

[55] Muhammad Haris Mughees, Hao Chen, and Ling Ren. 2021. OnionPIR: Response Efficient Single-Server PIR. In *ACM CCS 2021*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 2292–2306. https://doi.org/10.1145/3460120.3485381

[56] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *J. Algorithms* 51, 2 (2004), 122–144. https://doi.org/10.1016/J.JALGOR.2003.12.002

[57] Pascal Paillier. 1999. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes, See [70], 223–238. https://doi.org/10.1007/3-540-48910-X_16

[58] Bijeeta Pal, Tal Daniel, Rahul Chatterjee, and Thomas Ristenpart. 2019. Beyond Credential Stuffing: Password Similarity Models Using Neural Networks. In *2019 IEEE Symposium on Security and Privacy (SP)*. 417–434. https://doi.org/10.1109/SP.2019.00056

[59] Jeongeun Park and Mehdi Tibouchi. 2020. SHECS-PIR: Somewhat Homomorphic Encryption-Based Compact and Scalable Private Information Retrieval. In *ESORICS 2020, Part II (LNCS, Vol. 12309)*, Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider (Eds.). Springer, Heidelberg, 86–106. https://doi.org/10.1007/978-3-030-59013-0_5

[60] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. 2018. Private Stateful Information Retrieval. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 1002–1019. https://doi.org/10.1145/3243734.3243821

[61] Sarvar Patel, Giuseppe Persiano, Kevin Yeo, and Moti Yung. 2019. Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing. In *ACM CCS 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM Press, 79–93. https://doi.org/10.1145/3319535.3354213

[62] Sarvar Patel, Joon Young Seo, and Kevin Yeo. 2023. Don't be Dense: Efficient Keyword PIR for Sparse Databases. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 3853–3870. https://www.usenix.org/conference/usenixsecurity23/presentation/patel

[63] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. 2008. A Framework for Efficient and Composable Oblivious Transfer. In *CRYPTO 2008 (LNCS, Vol. 5157)*, David Wagner (Ed.). Springer, Heidelberg, 554–571. https://doi.org/10.1007/978-3-540-85174-5_31

[64] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2020. PSI from PaXoS: Fast, Malicious Private Set Intersection. In *EUROCRYPT 2020, Part II (LNCS, Vol. 12106)*, Anne Canteaut and Yuval Ishai (Eds.). Springer, Heidelberg, 739–767. https://doi.org/10.1007/978-3-030-45724-2_25

[65] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. 2018. Efficient Circuit-Based PSI via Cuckoo Hashing. In *EUROCRYPT 2018, Part III (LNCS, Vol. 10822)*, Jesper Buus Nielsen and Vincent Rijmen (Eds.). Springer, Heidelberg, 125–157. https://doi.org/10.1007/978-3-319-78372-7_5

[66] Oded Regev. 2005. On lattices, learning with errors, random linear codes, and cryptography. In *37th ACM STOC*, Harold N. Gabow and Ronald Fagin (Eds.). ACM Press, 84–93. https://doi.org/10.1145/1060590.1060603

[67] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. 2014. The Variable-Increment Counting Bloom Filter. *IEEE/ACM Trans. Netw.* 22, 4 (2014), 1092–1105. https://doi.org/10.1109/TNET.2013.2272604

[68] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce M. Maggs. 2021. Puncturable Pseudorandom Sets and Private Information Retrieval with Near-Optimal Online Bandwidth and Time. In *CRYPTO 2021, Part IV (LNCS, Vol. 12828)*, Tal Malkin and Chris Peikert (Eds.). Springer, Heidelberg, Virtual Event, 641–669. https://doi.org/10.1007/978-3-030-84259-8_22

[69] Radu Sion and Bogdan Carbunar. 2007. On the Practicality of Private Information Retrieval. In *NDSS 2007*. The Internet Society.

[70] Jacques Stern (Ed.). 1999. *EUROCRYPT'99.* LNCS, Vol. 1592. Springer, Heidelberg.

[71] Volker Strassen. 1969. Gaussian Elimination is Not Optimal. *Numer. Math.* 13, 4 (aug 1969), 354–356. https://doi.org/10.1007/BF02165411

[72] Kurt Thomas, Frank Li, Ali Zand, Jacob Barrett, Juri Ranieri, Luca Invernizzi, Yarik Markov, Oxana Comanescu, Vijay Eranti, Angelika Moscicki, Daniel Margolis, Vern Paxson, and Elie Bursztein. 2017. Data Breaches, Phishing, or Malware? Understanding the Risks of Stolen Credentials. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1421–1434. https://doi.org/10.1145/3133956.3134067

[73] Stephanie Wehner and Ronald de Wolf. 2005. Improved Lower Bounds for Locally Decodable Codes and Private Information Retrieval, See [18], 1424–1436. https://doi.org/10.1007/11523468_115

[74] Kevin Yeo. 2023. Lower Bounds for (Batch) PIR with Private Preprocessing, See [41], 518–550. https://doi.org/10.1007/978-3-031-30545-0_18

[75] Mingxun Zhou, Wei-Kai Lin, Yiannis Tselekounis, and Elaine Shi. 2023. Optimal Single-Server Private Information Retrieval, See [41], 395–425. https://doi.org/10.1007/978-3-031-30545-0_14

[76] Mingxun Zhou, Andrew Park, Elaine Shi, and Wenting Zheng. 2023. Piano: Extremely Simple, Single-Server PIR with Sublinear Server Computation. Cryptology ePrint Archive, Paper 2023/452. https://eprint.iacr.org/2023/452 https://eprint.iacr.org/2023/452.

A ADDITIONAL PRELIMINARIES ON FILTERS

Here, we give alternative functionality for filter designs, to help provide context for the key-value filter abstraction that we develop in Section 3.1.

A.1 Data Storage (DS) Filters

Bloom filters [13] were the first construction of a long and fruitful series of works that encode datasets of m elements as a series of digits, and use hash functions to record fast data writing and access. In principle, they are a compact data structure for probabilistic set membership testing. The speed of writing and reading from the database is achieved at the cost of a configurable probability (ϵ) of *false-positive* reads occurring [15] — indicating, erroneously, that an element belongs to the set — while *false negatives* are impossible. The theoretical lower bound for the space requirement of all such data structures is $O(\log(1/\epsilon)m)$ bits [16].

Such filters have an algorithmic description that can be generalised to the following form.

- $(\mathsf{F},\mathsf{H}) \leftarrow \mathrm{DS}.\mathsf{setupFilter}(1^m,\epsilon)$: A static initialisation function that returns a filter F of size $N = O(\log(1/\epsilon)m)$, and a set of uniform independent hash functions $\mathsf{H} = \{\mathsf{h}_i\}_{i\in[k]}$ for some $k \in \mathbb{N}$, and where $\mathsf{h}_i : \{0,1\}^* \mapsto [N]$.
- $b \leftarrow \mathsf{F.write}(\mathcal{S},\mathsf{H})$: Writes a set $\mathcal{S} \subseteq \{0,1\}^*$ to F using the set of hash functions in H, and returns $b = 1$ if successful, and $b = 0$ otherwise. If $b = 0$, it may be necessary to regenerate the filter.
- $b \leftarrow \mathsf{F.check}(\mathsf{x},\mathsf{H})$: Checks if $x \in \{0,1\}^*$ has been previously written to F using H, and returns $b = 1$ if it has and $b = 0$ otherwise. Note that $b = 1$ is returned erroneously with probability up to ϵ.

We write $\mathsf{F}[j]$ to indicate the j^{th} entry of F. Furthermore, we will write $\mathsf{H}(\mathsf{x})$ as shorthand to indicate the set $\{\mathsf{h}_i(\mathsf{x})\}_{i\in[k]}$, and $\mathsf{F}[\mathsf{H}(\mathsf{x})]$ as shorthand to indicate the set $\{\mathsf{F}[\mathsf{h}_i(\mathsf{x})]\}_{i\in[k]}$.

Correctness. We can express the correctness of a data storage filter with respect to the following two definitions.

Definition A.1 (Correctness of inclusion). Let $(\mathsf{F},\mathsf{H}) \leftarrow \mathrm{DS}.\mathsf{setupFilter}(1^m,\epsilon)$, and let $\mathcal{S}$ be any set $\mathcal{S} \subseteq \{0,1\}^*$ such that $\mathsf{x} \in \{0,1\}^*$ is contained within $\mathcal{S}$. We say that F correctly indicates inclusion if the following equality holds:

$$\Pr[1 \leftarrow \mathsf{F.check}(\mathsf{x},\mathsf{H}) \mid 1 \leftarrow \mathsf{F.write}(\mathcal{S},\mathsf{H})] = 1.$$

Definition A.2 (Correctness of non-inclusion). Let $(\mathsf{F},\mathsf{H}) \leftarrow \mathrm{DS}.\mathsf{setupFilter}(1^m,\epsilon)$, and let $\mathcal{S}$ be any set $\mathcal{S} \subseteq \{0,1\}^*$ such that $\mathsf{x} \in \{0,1\}^*$ is **not** contained within $\mathcal{S}$. We say that F correctly indicates non-inclusion (with false-positive probability ϵ), if the following inequality holds:

$$\Pr[1 \leftarrow \mathsf{F.check}(\mathsf{x},\mathsf{H}) \mid 1 \leftarrow \mathsf{F.write}(\mathcal{S},\mathsf{H})] \le \epsilon.$$

Constructions. For Bloom filters, F is constructed as a bitstring of length $N = 1.44km$, where $k = \log(1/\epsilon)$. For each $\mathsf{x} \in \mathcal{S}$, $\mathsf{F.write}(\mathcal{S},\mathsf{H})$ executes each $j \leftarrow \mathsf{h}_i(\mathsf{x})$ and sets $\mathsf{F}[j] = 1$ (ignoring the cases where it is already set to 1). If $\mathsf{F}[j] = 1$ for each j already, then construction of F fails. When $\mathsf{F.check}(\mathsf{x},\mathsf{H})$ is executed, it returns $\bigwedge_{i=1}^{k} \mathsf{F}[\mathsf{h}_i(\mathsf{x})] = 1$. The time complexity of evaluating the Bloom filter is k hash function operations per write/read. The space requirement is a factor of 1.44 larger than the optimal lower bound. Various advances have been made in the area of creating cheaper variants of Bloom filters, with respect to querying and storage constraints [14, 30, 53, 67].

A.2 Fingerprint-Based (FB) Filters

Fingerprint-based filters consider an element space $\mathcal{X} \subseteq \{0,1\}^*$, and are parametrised by a fingerprint function, $\mathsf{fpt}_\epsilon : \{0,1\}^* \mapsto \{0,1\}^\mu$. The function fpt_ϵ is, in turn, parametrised by the false-positive probability via the polynomial $\mu = \mu(\epsilon)$. In essence, for every check on a value x to the filter, a fingerprint y is returned. We say that x is considered as part of the filter if $y = \mathsf{fpt}_\epsilon(\mathsf{x})$. To explicitly differentiate fingerprint-based filters from data storage filters, we add an extra algorithm $\mathsf{reconstruct}$, that explicitly returns the fingerprint values from the filter. The modified algorithmic structure is as follows.

- $(\mathsf{F},\mathsf{H},\mathsf{fpt}_\epsilon) \leftarrow \mathrm{FB}.\mathsf{setupFilter}(1^m,\epsilon,\circ)$: A static initialisation function that generates a filter F of size $N = O(\log(1/\epsilon)m)$, and a set of hash functions $\mathsf{H} = \{\mathsf{h}_i\}_{i\in[k]}$ for some $k \in \mathbb{N}$, and where $\mathsf{h}_i : \{0,1\}^* \mapsto [N]$. Also returns a function $\mathsf{fpt}_\epsilon : \{0,1\}^* \mapsto \{0,1\}^{\mu(\epsilon)}$, parameterised by ϵ. The input $\circ$ defines a mathematical operation that is used for reconstructing data items.
- $b \leftarrow \mathsf{F.write}(\mathcal{S},\mathsf{H},\mathsf{fpt}_\epsilon)$: Writes a map $\mathcal{S}$ to F using the set of hash functions in H, and returns $b = 1$ if successful, and $b = 0$ otherwise. If $b = 0$, it may be necessary to regenerate the filter.

- $\mathsf{F}[\mathsf{H}(\mathsf{x})] \leftarrow \mathsf{F.check}(\mathsf{x}, \mathsf{H})$: Evaluates $\mathsf{H}(\mathsf{x})$ for x, and returns $\mathsf{F}[\mathsf{H}(\mathsf{x})]$.
- $\mathsf{fpt}_\epsilon(\mathsf{x}) \leftarrow \mathsf{F.reconstruct}(\mathsf{x}, \mathsf{H}, \mathsf{fpt}_\epsilon)$: Runs $\mathsf{F}[\mathsf{H}(\mathsf{x})] \leftarrow \mathsf{F.check}(\mathsf{x}, \mathsf{H})$, and then returns $\bigcirc_{i=1}^{k}\mathsf{F}[\mathsf{h}_i(\mathsf{x})] = \mathsf{F}[\mathsf{h}_1(\mathsf{x})] \circ \ldots \circ \mathsf{F}[\mathsf{h}_k(\mathsf{x})]$.

Correctness. For fingerprint-based filters, the check algorithm is only used to return the slots of the filter that should be read when checking the fingerprint of data elements. Therefore, the correctness criteria must be modified to take into account the output of the reconstruct algorithm instead. The modified correctness definitions are given below.

Definition A.3 (Correctness of inclusion). Let $(\mathsf{F}, \mathsf{H}) \leftarrow \mathrm{FB.setupFilter}(1^m, \epsilon)$, and let $\mathcal{S}$ be any set $\mathcal{S} \subseteq \{0,1\}^*$ such that $\mathsf{x} \in \{0,1\}^*$ is contained within $\mathcal{S}$. We say that F correctly indicates inclusion if the following equality holds:

$$\Pr\left[y = \mathsf{fpt}_\epsilon(\mathsf{x}) \,\middle|\, \substack{1 \leftarrow \mathsf{F.write}(\mathcal{S}, \mathsf{H}, \mathsf{fpt}_\epsilon) \\ y \leftarrow \mathsf{F.reconstruct}(\mathsf{x}, \mathsf{H}, \mathsf{fpt}_\epsilon)}\right] = 1.$$

Definition A.4 (Correctness of non-inclusion). Let $(\mathsf{F}, \mathsf{H}) \leftarrow \mathrm{FB.setupFilter}(1^m, \epsilon)$, and let $\mathcal{S}$ be any set $\mathcal{S} \subseteq \{0,1\}^*$ such that $\mathsf{x} \in \{0,1\}^*$ is **not** contained within $\mathcal{S}$. We say that F correctly indicates non-inclusion (with false-positive probability ϵ), if the following inequality holds:

$$\Pr\left[y = \mathsf{fpt}_\epsilon(\mathsf{x}) \,\middle|\, \substack{1 \leftarrow \mathsf{F.write}(\mathcal{S}, \mathsf{H}, \mathsf{fpt}_\epsilon) \\ y \leftarrow \mathsf{F.reconstruct}(\mathsf{x}, \mathsf{H}, \mathsf{fpt}_\epsilon)}\right] \leq \epsilon.$$

Constructions. Garbled Bloom filters were first proposed in [33] as a filter that allows such reconstruction, where $\mathcal{K} = \{0,1\}^*$, $\mathcal{X} = \mathbb{Z}_p$, $\mu(\epsilon) = \epsilon$, and $\circ$ is mod-p addition, for some $p \in \mathbb{N}$. The total space usage of such filters is $\log(p) \cdot N$ bits, where N is the number of entries (which is equivalent to a standard Bloom filter). Such filters are useful for establishing efficient private set intersection protocols, and have also been used in PIR scenarios [33, 47]. Filters are typically first initialised as a series of random values in $\mathcal{X}$. Writing entries to such filters involves computing $\mathsf{F}[\mathsf{H}(\mathsf{k})] \leftarrow \mathsf{F.check}(\mathsf{k}, \mathsf{H})$, and then setting $\mathsf{F}[\mathsf{h}_1(\mathsf{k})] = \mathsf{fpt}_\epsilon(\mathsf{x}) \circ (-\mathsf{F}[\mathsf{h}_2(\mathsf{k})]) \circ \ldots \circ (-\mathsf{F}[\mathsf{h}_k(\mathsf{k})])$. When writing subsequent values, previously modified entries of the filter cannot be modified again; the lexicographically smallest value of $i \in [k]$ where $\mathsf{F}[\mathsf{h}_i(\mathsf{k})]$ can be modified instead. If no such i exists, the construction of F fails.

Cuckoo filters [35] represent a fingerprint-based key-value filter, with smaller space overhead $((\log(1/\epsilon) + 1 + \log b)m/\alpha$, for load factor α), with $k = 2$, $\mathcal{K} = \{0,1\}^*$, $\mathcal{X} = \{0,1\}^l$ for $l = \log(1/\epsilon)$, $\mu = \epsilon$, and $\circ$ is the l-bit XOR operation.

More recently, XOR filters have been proposed by Graf and Lemire [39] as fingerprint-based filters, based on Bloomier filters [20]. XOR filters choose $k = 3$, and divide the filter into three continuous segments, in other words $\mathsf{F} = [\mathsf{F}_1 \| \mathsf{F}_2 \| \mathsf{F}_3]$. The set H is chosen such that h_i maps to indices in the range $N/3$, corresponding to segment F_i. Like cuckoo filters, $\mathcal{X} = \{0,1\}^*$, and $\circ$ is the μ-bit XOR operation. It has been shown that the overall space overhead of XOR filters is $\varsigma\mu \cdot m$, for $\varsigma \approx 1.23$, which is significantly closer to the optimal space overhead than standard Bloom filters. Furthermore, Graf and Lemire showed that XOR filters perform better than Cuckoo filters in cases where ϵ is large (e.g. 1%), but worse when ϵ is chosen to be small (e.g. 2^{-30}) [39].

Offline performance				
Sizes	**Runtime** (sec)	**Storage** (GB)	**Download** (MB) Frodo	Simple
$2^{16} \times 1\,\mathrm{kB}$	25866	0.226	5.54	32.07
$2^{17} \times 1\,\mathrm{kB}$	50772	0.452	5.54	45.35
$2^{18} \times 1\,\mathrm{kB}$	101010	0.904	5.54	64.14
$2^{19} \times 1\,\mathrm{kB}$	225710	1.808	6.16	90.71
$2^{20} \times 1\,\mathrm{kB}$	490110	3.616	6.16	128.28

Table 4: Offline server runtimes (sec)**, storage** (GB)**, and client download costs** (MB) **of offline steps for ChalametPIR, using either** FrodoPIR **or** SimplePIR**, where** $k = 3$**. Runtimes for both** LWEPIR **schemes are estimated to be equivalent, due to identical number of operations.**

B ADDITIONAL BENCHMARKS

Offline costs. Table 4 provides example offline costs for instantiating ChalametPIR with both FrodoPIR and SimplePIR, based on running the computation on a Macbook M1 Max device. The main difference is in the size of the download (the computation and storage only differ depending on the choice of LWE parameters). As we mentioned in Section 7, utilising alternative LWEPIR schemes such as DoublePIR [42] or HintlessPIR [48] will potentially result in smaller costs. As such, our benchmarks here provide an upper-bound that provide a basis for understanding the performance of the offline phase of ChalametPIR. Note that the computational and storage costs of the offline phase amortise over all client queries, meaning that this expensive one-time cost tends to zero for large systems of clients. Furthermore, the one-time download for clients amortises over all their queries.

Comparison with Index-based PIR. For providing additional context for the comparison that we made between Chalamet-PIR and index-based LWE PIR schemes in Section 6.2, we provide concrete runtime and bandwidth numbers for the index-based FrodoPIR scheme in Table 5.

Financial costs. To provide a rough estimate of the financial costs, we use the same costs stated previously for standard AWS usage ($0.0425 per CPU hour, and $0.09 per GB of download [7]). Ultimately, even for moderate numbers of clients, the offline costs become quickly insignificant compared with the per-query online costs accounted for in Table 3.

For the offline computation (which is amortised across all client queries *globally*), the up-front cost ranges between $7.19 and $136. While this initial cost is fairly expensive, even for moderate numbers of clients (e.g. 1M) making moderate numbers of queries (e.g. 100), the costs amortise to lower or of the same order as the per-query online costs (Table 3). For the communication, the download costs per-client range

Online Performance of Index-based FrodoPIR [29]				
	DB ($m \times w$)	Query	Response	Parsing
Macbook M1 Max	$2^{16} \times 1024$ B	0.0076956	5.2735	0.18083
	$2^{17} \times 1024$ B	0.017356	10.545	0.18544
	$2^{18} \times 1024$ B	0.055522	21.101	0.18061
	$2^{19} \times 1024$ B	0.1023	47.675	0.20108
	$2^{20} \times 1024$ B	0.21222	100.63	0.20483
EC2 "t2.t2xlarge"	$2^{16} \times 1024$ B	0.11887	29.482	0.34437
	$2^{17} \times 1024$ B	0.080101	50.585	0.34515
	$2^{18} \times 1024$ B	0.20374	118.54	0.3466
	$2^{19} \times 1024$ B	0.48432	263.83	0.3768
	$2^{20} \times 1024$ B	0.85748	537.28	0.37458
EC2 "c5.9xlarge"	$2^{20} \times 256$ B	1.2324	118.46	0.065281
	$2^{17} \times 30$ kB	0.036396	36.396	8.1519
	$2^{14} \times 100$ kB	0.0033412	637.81	26.599

Table 5: Online performance of Index-based FrodoPIR, based on the implementation provided by [29].

from \$4.8e−5 to \$5.4e−5 for the FrodoPIR configuration, and between \$2.8e − 3 and \$0.01 for SimplePIR. In the case of FrodoPIR, if each client makes at least 100 queries, the costs again become quickly insignificant compared with the per-client online costs. Finally, recent improvements made by the HintlessPIR approach would likely reduce these costs even further. Consequently, we consider the offline phase to have little impact on the total costs of the scheme.

C BINARY FUSE FILTER ALGORITHMS

We define two algorithms that allow us to instantiate the setupFilter (Algorithm 1) and write (Algorithm 2) functionality for a Binary Fuse Filter $\mathsf{BFF_p}$. Intuitively speaking, these definitions differ from their original specification in that they allow us to encode key-value maps in the structure, whereas the work of [40] only allows encoding a set.

First, the setupFilter algorithm returns the filter structure $\mathsf{BFF_p}$, based on specific parameters, and initialises the hash functions for querying. The parameter m defines the maximum number of key-value pairs in the map KV that will be encoded in the filter, defining its eventual size N, in tandem with the false-positive probability ϵ. The operation $+_p$ corresponds to the operation (addition mod$p$) used in reconstruct, and thus each entry of $\mathsf{BFF_p}$ is an element of $\mathbb{Z}_p$.

Second, the write algorithm (Algorithm 2) defines a mechanism for encoding each key-value pair of a given KV in the filter F that is output by the setupFilter algorithm. This algorithm can fail with non-negligible probability (see line 21). When an abort occurs, it is necessary to run an entirely new setupFilter process and repeat the write algorithm (i.e. using a new set of hash functions).

Algorithm 1 $\mathsf{BFF_p}$ $\mathsf{setupFilter}(1^m, \epsilon, +_p)$ algorithm

```
Require: A parameter $k \in \{3,4\}$. A parameter $m \in \mathbb{N}$ denoting the
    number of keys in the maps written to $\mathsf{BFF_p}$.
 1: Sample hash : $\{0,1\}^* \mapsto \{0,1\}^\mu$ as a universal hash function
    for $\mu = \mu(\epsilon)$.
 2: Set $s \in \{2^{\lfloor \log_{3.33}(m)+2.25 \rfloor}, 2^{\lfloor \log_{2.91}(m)-0.5 \rfloor}\}$ for $k \in \{3,4\}$, respectively.
 3: Let $N = çm$, where
    $ç = \begin{cases} \max\left(\left\lfloor\left(0.875 + 0.25 \cdot \max(1, \frac{\log(10^6)}{\log(m)})\right) \cdot m\right\rfloor, \lfloor 1.125m \rfloor\right) \\ \max\left(\left\lfloor\left(0.77 + 0.305 \cdot \max(1, \frac{\log(6 \cdot 10^5)}{\log(m)})\right) \cdot m\right\rfloor, \lfloor 1.075m \rfloor\right) \end{cases}$
    for $k \in \{3,4\}$, respectively.
 4: Sample universal hash functions $\mathsf{h}' : \{0,1\}^* \mapsto [N/s]$. and
    $\mathsf{h}'' : \{0,1\}^* \mapsto [s]$
 5: $\mathsf{F} = \emptyset$
 6: for $i \in [N]$ do $\mathsf{F}[i] \leftarrow\!\$\ \mathbb{Z}_p$ end for
 7: $\mathsf{H} = \emptyset$
 8: for $i \in [k]$ do
 9:     Let $\mathsf{h}_i$ be the function that is evaluated as $\mathsf{h}_i(\cdot) = (N/s \cdot (\mathsf{h}''(\cdot) - 1)) + \mathsf{h}'(\cdot \| i)$
10:     $\mathsf{H}[i] = \mathsf{h}_i$
11: end for
12: Let $\mathsf{fpt}_\epsilon$ be the function that is evaluated as $\mathsf{fpt}_\epsilon(\mathsf{k}, \mathsf{x}) =$
    $\mathsf{hash}(\mathsf{k}) \| \mathsf{x}$
13: return $(\mathsf{F}, \mathsf{H}, \mathsf{fpt}_\epsilon)$
```

Algorithm 2 $\mathsf{BFF_p}$ $\mathsf{F.write}(\mathcal{M}, \mathsf{H}, \mathsf{fpt}_\epsilon)$ algorithm

```
Require: $\mathcal{M}$ is a map containing $m$ distinct keys sampled from $\mathcal{K}$,
    each associated with a data element sampled from $\mathcal{X}$.
 1: Let $S$ be a vector containing the keys $\{\mathsf{k}_i\}_{i \in [m]}$ from $\mathcal{M}$, ordered
    by $\mathsf{h}_1(\mathsf{k})$.
 2: Let $C = \emptyset$
 3: for $j \in [N]$ do $C[j] = \emptyset$ end for
 4: for $i \in [N]$ do
 5:     $\mathsf{k}_i = S[i]$
 6:     for $\iota \in [k]$ do $C[\mathsf{h}_\iota(\mathsf{k}_i)].\mathsf{push}(\mathsf{k}_i)$ end for
 7: end for
 8: $Q = \emptyset$
 9: for $j \in [N]$ do
10:     if $|C[j]| = 1$ then $Q.\mathsf{push}(j)$ end if
11: end for
12: $P = \emptyset$
13: while $|Q| > 0$ do
14:     $j \leftarrow Q.\mathsf{pop}()$
15:     if $|C[j]| = 1$ then
16:         $\mathsf{k}' = C[j]$
17:         $P.\mathsf{push}((\mathsf{k}', j))$
18:         for $\iota \in [k]$ do $C[\mathsf{h}_\iota(\mathsf{k}')].\mathsf{rem}(\mathsf{k}')$ end for
19:     end if
20: end while
21: if $|P| \neq m$ then abort end if
22: while $|P| > 0$ do
23:     $(\mathsf{k}', j) \leftarrow P.\mathsf{pop}()$
24:     $\mathsf{F}[j] = 0 \in \mathbb{Z}_p$
25:     for $z \in \mathsf{H}(\mathsf{k}')$ do
26:         if $z \neq j$ then $\mathsf{F}[j] = \mathsf{F}[j] - \mathsf{F}[z]$ else $\mathsf{F}[j] = \mathsf{F}[j] +$
            $\mathsf{fpt}_\epsilon(\mathsf{k}', \mathcal{M}[\mathsf{k}'])$ end if
27:     end for
28: end while
```