Crust: Verifiable And Efficient Private Information Retrieval with Sublinear Online Time

Yinghao Wang
Zhejiang University

Abstract

Private Information Retrieval (PIR) is a cryptographic primitive that enables a user to retrieve information from a database without revealing the particular information they are seeking, thus preserving their privacy. PIR schemes suffer from high computation overhead. By running an offline preprocessing phase, PIR scheme can achieve sublinear online server computation. On the other hand, although protocols for honest-but-curious servers have been well-studied in both single-server and multi-server scenarios, little work has been done for the case where the server is malicious. In this paper, we propose a simple but efficient sublinear PIR scheme named Crust. The scheme is tailored for verifiability and provide privacy and data integrity against malicious servers. Our scheme can work with two servers or a single server. Aside from verifiability, our scheme is very efficient. Comparing to state-of-the-art two-server and single-server sublinear PIR schemes, our scheme is 22x more efficient in online computation. To the best of our knowledge, this is the first PIR scheme that achieves verifiability, as well as amortized $O(\sqrt{n})$ server computation.

1 Introduction

Private Information Retrieval (PIR) is a noteworthy cryptographic primitive where a server holds a database represented by an array of n records, and a client wishes to query the i-th element ($i \in \{1..n\}$) without disclosing i to the server. PIR has numerous applications in real-world scenarios, including but not limited to private database search [19], private media consumption [9], and credential breach reporting [20].

Verifiability. The advent of cloud computing and the rising trend of outsourced computing have prompted us to reconsider PIR's implementation. By delegating databases to cloud servers with adequate computing power, the existing security model for PIR must be reevaluated. Most PIR schemes[4, 10, 14, 15, 18] assume an honest-but-curious server. However, this assumption is overly idealistic for the cloud setting, where a server may provide incorrect answers intentionally or due to system failures. Therefore, it is essential to consider malicious servers. In such cases, a server may deviate from the scheme and provide arbitrary responses, deceiving the client into accepting an incorrect answer.

In the aforementioned protocols, a server could (at least with non-negligible probability) force the client to retrieve any value it decides. Checklist [12] provides an application of "Safe Browsing" service. If the servers are malicious in the application, they can delibrately provide the client with wrong answers and cheat the client into malicious websites. Another example could be a PIR server for private DNS queries [22]. A malicious server may perform a DNS poisoning attack [5], thus preventing user from accessing certain web service or redirecting the user to a bogus website controlled by malicious party.

In order to secure a PIR scheme in the malicious setting, the client needs to be able to verify the server's response, which is a non-trivial task. In practice, this task is challenging as the verification must not reveal any information about the index being queried.

PIR Schemes with Sublinear Online Time. Corrigan-Gibbs and Kogan introduced an offline-online sublinear PIR construction for a two-server setting using puncturable pseudorandom sets (PPRS) in [4], by running an offline phase aforehead and retrieve hints for online queries. The scheme results in amortized $O(\sqrt{n})$ server computation and communication. They identified that utilizing a better instance of PPRS can further reduce communication costs. Shi et al.[18] presented a meticulous construction of such a PPRS, which they combined with [4] to achieve a PIR scheme with $O(logn)$ communication.

While the asymptotic complexity of the scheme looks promising, their are a few obstacles that prevent it from being practical. First, the scheme needs parallel instances to ensure its correctness, which makes its efficiency worse than expected. Second, if $O(logn)$ communication is to be achieved, either the construction in [18] or [4] is indispensable. The former is known to have a very large constant factor, and the

latter incurs linear client storage. There are a few following works that aims to tackle these obstacles in different ways such as Checklist [12] and TreePIR [13].

At Eurocypt 22', Corrigan-Gibbs et al.[3] applied this offline-online sublinear PIR technique to the single server scenerio. They proposed to retrieve the offline hints with homomorphic encryption so that one server could act as two servers. The work also introduced the concept of main hint and backup hints to allow multiple queries without replenishing existing hints. However, what makes the two-server scheme impractical also applies to the single-server scheme. The scheme needs parallel instances to ensure its correctness, and the communication cost is $O(\sqrt{n})$. The work is extended by Piano[24] which combines the highlight of TreePIR [13] and [3] to construct a more practical single-server scheme.

However, these works introduce extra complications in the scheme and hinder us from introducing verifiability to sublinear schemes. We will discuss the details in Section 3.

Possible constructions. A common approach to achieve data integrity is to compose some data authentication method with the target protocol. Concerning PIR, the database provider could attach a digital signature under a previously published key, or a merkle proof with reference to a certain merkle root to the database records. The integrity of data records are ensured by verifying the attached authentication. There are three major concerns that prevent us from applying these methods to PIR.

1. If the database records are very small in size, the authentication information could enlarge database record size significantly. A typical ECDSA signature is of size at least 320 bits. For a 1-bit record, the authentication information is 320 times larger than the record itself.

2. The authentication information requires extra care when handling database updates, damaging both integrity and efficiency. Without proper version control, the server may conduct a replay attack by replying with outdated database. And the merkle proofs attached to records need to be recalculated after every single update.

3. Common data authentication are vulnerable to selective failure attacks [11]. In such attacks, a malicious server selectively tamper with the database records, and tries to infer the query index i from the result of integrity check. Consider that the database records are protected by digital signatures. The malicious server replaces the first record of the database with a random item that shall not pass the signature verification. If the client accept the answer, the server knows that the first item is not queried. Otherwise, the server knows that the first item is queried.

Therefore, the question of **Whether we can design a malicious-secure PIR scheme with sublinear computation** remains unanswered. We provide a positive answer to this question by offering Crust—two PIR scheme with sublinear computation and verifiability, one for the single server setting, and one for the two-server setting. The core idea of our construction is simple. We retrieve extra information from the server and use it to verify the server's response in the future. Our scheme offers verifiability while incurring minimal overhead.

Our contributions. Our contribution includes:

- We introduce verifiability to sublinear PIR schemes, enabling these schemes to be run in a malicious setting. Our two server scheme can provide integrity with two non-colluding servers, one of them being honest-but-curious and one of them being malicious. Our single server scheme provide integrity against malicious server with a digest from a trusted database provider. To the best of our knowledge, this is the first PIR scheme that achieves both verifiability and $O(\sqrt{n})$ amortized server computation.

- We analyse the exsiting constructions of sublinear PIR schemes and discuss the obstacles that prevent them from being verifiable. We propose a new construction specially designed for verifiability. The construction is efficient and simple. Furthermore, our construction can run in the semi-honest setting by removing verifiability, thus providing better efficiency. Without the overhead of verifiability, our scheme is 4.2x-7.7x more efficient in online computation than state-of-the-art non-verifiable schemes. The verifiability bring only around 2x overhead in online computation, which makes our scheme being verifiable and more efficient than non-verifiable schemes at the same time.

- We propose a detailed evaluation on our scheme comparing to the state-of-the-art schemes, alone with an open-source implementation for our scheme.

The rest of this paper is arranged as follows: We introduce the preliminaries in Section 2. Section 3 presents our proposed construction. In Section 4, we evaluate our scheme's performance. Section 5 introduces related works and Section 6 concludes with a summary.

2 Preliminaries

2.1 Notation

Throughout the rest of this paper, we assume a PIR scheme with a client $\mathcal{C}$ and two servers usually referred to as $\mathcal{S}_{hint}, \mathcal{S}_{query}$. A data owner O has a database $\mathcal{D}$ with n records, where $\mathcal{D} = \mathcal{D}_1, \ldots, \mathcal{D}_n$. We assume n is a perfect square, if not, the database could always be rounded up to the next perfect square. with at most $O(sqrtn)$ dummy records. We use the notation $[k]$ for the set $\{1, 2, \ldots, k\}$. We use λ to denote the security parameter and $negl(\cdot)$ for the negligible function.

Scheme	Servers	Communication	Computation	Verifiable
TreePIR [13]	2	$O(\sqrt{n})$	$O(\sqrt{n})$	✗
DPF-PIR [7]	2	$O(logn)$	$O(n)$	✗
APIR(Merkle) [2]	2	$O(\lambda\sqrt{n}\log n)$	$O(\lambda n\log n)$	✓
Crust (Theorem 1)	2	$O(\sqrt{n})$	$O(\sqrt{n})$	✓
SimplePIR [10]	1	$O(\sqrt{n})$	$O(n)$	✗
Piano [24]	1	$O(\sqrt{n})$	$O(\lambda\sqrt{n})$	✗
APIR(LWE) [2]	1	$O(\sqrt{n})$	$O(n)$	✓
Crust (Theorem 2)	1	$O(\sqrt{n})$	$O(\lambda\sqrt{n})$	✓

Table 1: A comparison of PIR schemes on database size n. The communication and computation are ammortized over $\sqrt{n}$ queries. Some of the schemes has a client-irrelevent database preprocessing phase, which is not included in the table. The database in APIR is rearranged to balance upload and download.

Letter	Meaning
$\mathcal{D}$	Database
ck	The client key, consisting of PPRS keys
sk	A PPRS key
q	A client query
a	A server answer
h	The hint
λ	Security parameter

Table 2: Summary of frequent notations.

The notation used in this paper is shown in Table 2:

2.2 Private Information Retrieval (PIR)

We adopt the definition from CK20[4].

Definition 2.1 (Offline-Online PIR)**.** An *Offline-Online PIR* scheme allows a client to retrieve a record $\mathcal{D}_i$ from the database $\mathcal{D}$ without revealing the index i to any of the k servers. The scheme consists of algorithm tuple $PIR = (Setup, Hint, Query, Answer, Reconstruct)$:

- $Setup(1^\lambda, n) \to (ck, q_h)$, generates client key ck and hint queries q_h given database size n and security parameter λ.
- $Hint(\mathcal{D}, q_h) \to h$ generates hints h from database $\mathcal{D}$ and hint queries q_h.
- $Query(ck, i) \to q$ generates a query q consisting of k subqueries given the hints h and the index i. The j-th subquery is sent to the j-th server.
- $Answer(q_j) \to a_j$ generates answer a_j in response to subquery q_j.
- $Reconstruct(h, a) \to \mathcal{D}_i$ reconstructs the record $\mathcal{D}_i$ with the help of hint h given answers a.

The scheme should satisfy two essential properties:

Correctness. The client receives the correct record $\mathcal{D}_i$ with overwhelming probability (i.e., $1 - negl(\lambda)$) if both the client and the servers execute the scheme honestly.

Privacy. The servers learns nothing about the index i from the query q_j sent by the client.

In this paper, we primarily explore the schemes related to a two-server setting where $k = 2$, and the single server setting where $k = 1$. In the two-server setting, we assume they do not collude during the execution of the scheme, which is common in sublinear PIR settings. A hint in such schemes usually contains a set and a value. We use the word "parity" or "hint parity" to refer to the value contained in a hint. The word "parity" also refers to the value in an answer. And we use "set" or "hint set" to refer to the set in a hint.

2.3 Puncturable pseudorandom set(PPRS)

A puncturable pseudorandom set (PPRS) is a set that can be represented with a short key and supports an additional "puncture" operation that removes an element from the set. This set is characterized by three algorithms $(Gen, Punc, Eval)$, parameterized by the set size s, range R, key space $\mathcal{K}$, punctured-key space $\mathcal{K}_p$:

- $Gen(1^\lambda, n) \to sk$ probabilistically generates a set key $sk \in \mathcal{K}$, given the set size n and security parameter λ.
- $Punc(sk, i) \to sk_p$ generates a punctured set key $sk_p \in \mathcal{K}_p$, given the key sk and the punctured element index i.
- $Eval(sk) \to S$ generates the corresponding set S, given the set key $sk \in \mathcal{K} \cup \mathcal{K}_p$. Each element in the set is an element in $[R]$.

PPRS must satisfy efficiency, correctness, and privacy. We informally define these as follows:

Efficiency. All three algorithms $(Gen, Punc, Eval)$ should run in time linear in s and polylogarithmic in n.

Correctness. The original set and the punctured set should be correctly generated and evaluated.

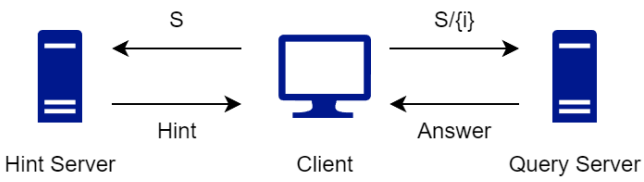

Figure 1: An illustration of the base construction in CK20. The client retrieve hints from hint server, puncturing the item it wishs to query from the hint set, and send the punctured set to the query server. The query server returns the answer to the client. The client can then reconstruct the record from the answer and the hint.

Privacy. The punctured item should be indistinguishable from any other item that is not in the punctured set.

It is noteworthy that here we do not require the PPRS to be non-trivial. That is to say, if the set does not have a succinct expression after an element is punctured. To express the set in plain as a list of values would also satisfied the definition. We shall see that in some cases, we would have to use such a trivial PPRS as the punctured index effectively leaks information. In the following sections, we will use the term "puncture" to describe the operation of removing an element from the set, regardless of whether the PPRS is trivial or not.

In addition, PPRS in our construction may be extended by relaxing the correctness requirement. It can be evaluated to a set with 1 more element than the original set. A detailed explanation is provided in Section 3.

PPRS reduce the space needed to store a list of sets. In the form of PPRS, a set can be effectively expressed with a short key.

2.4 Base construction

To better comprehend our scheme, a basic understanding on the construction 44 in [4] is vital. The scheme is described as follows: two servers, S_{hint} and S_{query}, each holding a copy of the database, are involved. The construction consists of two phases: an offline phase and an online phase.

Offline phase. The client generates m PPRS $\{sk_j\}_{j=1}^{m}$ with size s and range n. The client sends them to S_{hint}. The hint server calculates a parity h_j for each set and returns them to the client. A parity consists of the sum all records in the set sk_j, i.e. $h_j = \Sigma_{i \in Eval(sk_j)} \mathcal{D}_i$. The client stores these sets and their corresponding parities.

Online phase. To make a query, the client finds a PPRS sk_t containing the desired index i and remove i from the set. It also samples a new set sk_{new} containing i and remove i from it. sk_{new} is sent to S_{hint} and sk_t is sent to s_{query}. The servers then calculate the parities of the punctured sets a_{query} and a_{hint} and return them to the client. The client then reconstructs the record $\mathcal{D}_i$ from a_{query} and replenishes the hints from a_{hint}.

In the scheme, the hints is defined as the sum of all the elements in the set. So the record could be easily reconstructed by calculating $\mathcal{D}_i = h_t - a_{query}$. The used set sk_t must be discarded to avoid the server learning the index i. So a new set sk_{new} is sampled to replace the used set. The client then stores the new set sk_{new} and its corresponding parity $h_{new} = h_t - a_{query} + a_{hint}$.

The problem is that, the distribution of the sets sk_t and sk_{new} are twisted. The sets never contains the index i. To fix that, with probability $\frac{s-1}{n}$, the client would not select sk_t. It merely samples a sk_{dummy} containing i and punctures a random element other than i. The set sk_{dummy} is then sent to both servers.

A single instance of the construction fails with non-negligible probability. The client may hold no such set containing i, or a dummy set is sampled and sent to the servers. In both case, the client can learn nothing. To satisfy correctness, λ instances are run in parallel. With proper parameters, it could be shown that in each single instance the scheme fails with probability less than $\frac{1}{2}$, so the final scheme fails with probability negligible in λ, thus correctness holds.

2.5 Verifiability and Selective failure

The selective failure attacks, first proposed in 2006[11], demand a scheme to retain privacy when a malicious server is given access to the verification result of the client. It can be seen as the client sends a bit b indicating whether the answer is accepted to the servers. Thus, the servers may "guess" a index that the client is likely to query, and only corrupt the one record. Privacy can be violated if a scheme does not put that into careful consideration, as we have discussed in the introduction.

The problem is addressed in previous work [2]. They bind one record to the entire database with a digest, and perform linear computaion on the database. However, the problem is subtler in sublinear schemes. The dilemma is clear: Not all records are accessed in one query, thus how can we tell if the database is changed or not, on those untouched records? In addtion to that, some recent works [12, 13, 16] introduce dummy queries to rectify the aforementioned twisted distribution of the query set. The dummy sets are ignored in the final answer. Such dummy queries, if not properly handled, could also be used to perform selective failure attacks.

3 Algorithms

3.1 Two Server Verifiable PIR

We first define our Verifiable PIR model based on the PIR definition.

Definition 3.1 (Verifiable Private Information Retrieval)**.** A *verifiable private information retrieval* scheme is a PIR

scheme $(Setup, Hint, Query, Answer, Reconstruct)$. In addition to correctness and privacy, the scheme also ensures integrity.

- $Setup(1^\lambda, n) \to (ck, q_h)$, given the database size n and security parameter λ, generates client key ck and hint queries q_h.
- $Hint(\mathcal{D}, q_h) \to h$, given the database $\mathcal{D}$ and the hint queries q_h, generates hints h.
- $Query(ck, i) \to (q_{hint}, q_{query})$, given the client key ck and the index i, generates two queries for hint server and query server separately.
- $AnswerQuery(D, q_{query}) \to a_{query}$, given the query q_{query}, generates answer a_{query} and replies to the client.
- $AnswerHint(D, q_{hint}) \to a_{hint}$, given the query q_{hint}, generates answer a_{hint} and replies to the client.
- $Reconstruct(h, a_{query}) \to \{\mathcal{D}_i, \perp\}$, given the answer a_{query}, reconstructs the record $\mathcal{D}_i$ with the help of hint h or reject the answers.
- $Replenish(ck, h, a_{hint}) \to (ck, h)$, given the answer a_{hint}, update the hints.

The *Answer* algorithm is delibrately split into two parts. In the scheme the two servers answer the queries in different ways. The scheme should satisfy two additional properties:

Integrity. The client should either reject the answer (output $\perp$) or output the correct record $\mathcal{D}_i$ with overwhelming probability (i.e., $1 - negl(\lambda)$) if the hint server is honest.

Privacy (Concerning selective failure attack). The servers learns nothing about the index i from the query q_j sent by the client with access to a bit b indicating if *Reconstruct* algorithm outputs $\perp$.

Theorem 1. There is a two-server offline/online verifiable private information retrieval scheme with correctness, privacy and strong integrity. On a database of size n, the scheme has:

- Offline communication complexity: $O(\lambda^2\sqrt{n})$
- Offline computation complexity: $O(\lambda n)$
- Online communication complexity: $O(\sqrt{n})$
- Online computation complexity: $O(\sqrt{n})$
- Ammortized computaion complexity: $O(\sqrt{n})$
- Support at least $\sqrt{n}$ queries.

Construction for Query Server. The core of our construction is extremely simple, and is commonly used in secure multi-party computation. In order to compute a function $f(x)$ with malicious participants, the participants calculate two functions $f(x)$ and $g(x) = \alpha f(x)$ with a hidden $\alpha \in F_{2^p}$. The results are then verified by checking if $f(x) \cdot \alpha = g(x)$. If the participants are honest, the verification will always succeed. Otherwise, the verification will fail with a probability 2^{-p}. In our case, we can use the same idea to verify the answer a_{query}.

It is important to note that the two servers in the base construction are not equivalent. We assume that S_{hint} is an honest server.

Instead of calculating one hint parity, we calculate two: the sum parity hs and random parity hr. The two parities are defined by:

For a set of elements, $S = \{x_1, x_2, \ldots, x_s\}$, and a set of random numbers $R = \{r_1, r_2, \ldots, r_s\}$,

$$hs = \Sigma_{i \in [s]} x_i,$$
$$hr = \Sigma_{i \in [s]} r_i \cdot x_i$$

We assume that both hints are both in F_{2^p}, and F_{2^p} is large enough to hold one record in the database.

Recall that in the actual scheme, the set S is represented by a short PPRS. It is natural that we use PPRS to express both the random numbers and the set. A query is also split into two parts: Sum query qs and random query qr. The other part remain the same as the base construction. qs and qr is defined by:

For a PPRS sk containing the query item at index i, a PPRS sr that would evaluate to random numbers,

$$qs = Punc(sk, i),$$
$$qr = Punc(sr, i)$$

The two PPRS sk and sr will have different range. sk evaluates to random indexes in $[n]$ while sr evaluates to random numbers in F_{2^p}.

The servers then compute the answers as and ar the same way as the hints are computed. Except with the punctured index and random number left out. The servers then send the answers to the client. The can client verifies the answers by checking if

$$hr - ar = Eval(sr)_i \cdot (hs - as)$$

If the verification fails, the client outputs $\perp$. An explanation on the integrity is delayed to Section 3.4.

We explicitly point out that this modification to hint calculation and reconstruction is orthogonal to how the hint set is selected. This enables our scheme to run more efficiently in a semi-honest scenario by simply removing the random parity.

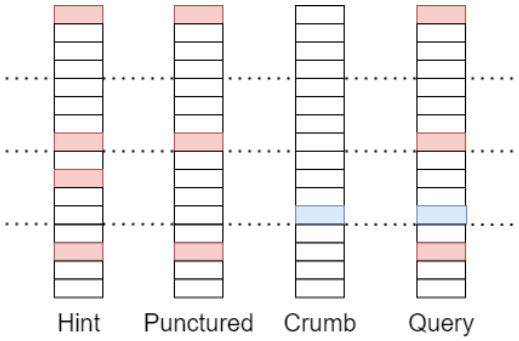

Figure 2: An illustration of the process of generating a query from a hint. The database contains $n = 16$ items and is divided into $\sqrt{n} = 4$ blocks. A hint contains one item in each block. The client wants to query the $i = 10$-th item, which is in the third block. The item is punctured from the hint, and a crumb in the third block is added to the set.

Handling Hint Server. S_{hint} differs from S_{query} in that the client C only uses "existing" hint at S_{query}, but learns "new" hint from S_{hint}. Specifically there are two key points:

1. We run the offline hint retrieving phase with the hint server, where query server is not involved at all.
2. A part of the hint, namely the punctured items, are removed when sent to the query server. These values are not accessed by both server, thus harder to verify by the client.

In such concerns, we choose to limit our scheme to tolerate one honest but curious hint server, and a malicious query server. Of course, the client could always run the single server scheme presented in Section 3.3, if no honest server is available. A detailed explanation on why the verification fails to work with a malicous hint server can be found in the Appendix A

3.2 A new construction of query

There are two main drawbacks in the construction.

1. It needs parallel instances to ensure correctness. The original paper [4] fixes the number of parallel instances to λ. In practice, to achieve a 2^{-128} correctness error on a database with 2^{20} items, around 13 instances are required. This is a significant overhead in both communication and computation, which makes the scheme, though asymptoticly efficient, perform poorly in practice.
2. The original construction of the PPRS does not allow efficient membership test. Which means that, in order for a client to find a PPRS that contains the queried index, the only way is to evaluate the entire set. This makes the client computation linear in the size of the database. An alternative solution is to use a hash table to record the items while evaluating the sets, but this would require linear client storage. Neither of the two solutions is acceptable.

Our solution to these two problems is greatly inspired by recent progression on sublinear PIR [13, 16]. To provide a better grasp of the solution, we first introduce the key contribution of these works—how PPRS is generated, and how queries are organized. As the rest of the scheme, namely the hint calculation, answer generation and reconstruction are almost identical to the original construction.

Instead of using a PPRS with range n to express the set directly, they proposed to first divide the database into $\sqrt{n}$ blocks, express the set with $\sqrt{n}$ ranged sets and use it as an offset in each $\sqrt{n}$ blocks. Concretely speaking, a set with raw elements in range $[\sqrt{n}]$:

$$Eval(S) = \{s_0, s_1, \ldots, s_{\sqrt{n}-1}\}$$

would evaluate to domain $[n]$ with items:

$$EvalSet(S) = \{0 \cdot \sqrt{n} + s_0, 1 \cdot \sqrt{n} + s_1, \ldots, (\sqrt{n}-1) \cdot \sqrt{n} + s_{\sqrt{n}-1}\}$$

To avoid confusion, the evaluation here is denoted as $EvalSet$ to distinguish from the $Eval$ functionality in PPRS definition. We use $EvalSet$ to refer to evaluation with the block offset, and $Eval$ to refer to evaluation without offset. This "divide and sample" trick enables efficient membership test. To check if an index j is in a set, we find the corresponding block by calculating $\lfloor j/\sqrt{n} \rfloor$ and check if the offset is $j - \sqrt{n} \cdot \lfloor j/\sqrt{n} \rfloor$.

Likewise, when generating a query, the client find a set that contains the queried index, and puncture the index in the set. This technique cannot be directly applied to a normal PPRS construction, since the punctured point would narrow the queried item to a block. Theses works propose to send additional random sets to as "dummy sets" to the server. So that the server received set is uniformly distributed. The dummy sets are ignored in the final answer. How the dummy sets are generate differs from scheme to scheme. In these schemes the client always learns the queried record. That release the schemes from running parallel instances, which makes the schemes significantly faster in practice. We omit the details here, and refer readers to the papers for a detailed explanation and proof for its correctness and privacy.

Eliminating dummy set with crumbs. The very problem that hinder us from applying this optimization is the dummy set. As we discussed in the selective failure part, the dummy set also requires verification. Otherwise it will be easy for the server to tell which set is the genuine one. A simple strategy would be the server answers one of the sets with a random value, and the other ones with honest results. If the client accept the answer, the randomly answered set is a dummy one, otherwise it is the genuine one.

We address the problem by taking a different approach to hide the punctured point. The client retrieves extra random hints called **crumbs**. Each crumb is a random record from a block. The client retrieves λ records from each block in the offline phase. There will be $\lambda\sqrt{n}$ crumbs in total.

When making queries, the client now uses a crumb in the queried block to make up the punctured block. Concretely speaking, if the client wants to query the i-th record, it first finds a set that contains i, and puncture i from the set. It then looks for a crumb in the $\lfloor i/\sqrt{n} \rfloor$-th block, and adds it to the set. An illustration is shown in Figure 2. The algorithm is describe in algorithm 3.

Since the crumb is merely one bare record. The "real" answer, which does not contain the crumb, can be easily calculated from the answer and the crumb. The client can then verify and reconstruct the record as is done in the unoptimized scheme.

It is worth mentioning that the punctured set and the crumb cannot be expressed succinctly as previous PPRS does, in the sense that, the punctured point effectively leaks which block the queried index is in. So the set will be expressed in plain in the online queries. Such a construction does not strictly follows the PPRS definition. Specifically, the construction does not satisfy correctness and privacy at the same time. The extra crumb introduced is necessary to retain privacy, but breaks correctness. We choose to relax correctness by allowing one extra element in the evaluated set.

Hint retrieving and replenishment. The offline hint retrieving and online hint replenishment require extra care.

In the offline phase, crumbs can be easily retrieved in plain. Since they are retrieved from a semi-honest hint server with a PRG, the crumbs can be sampled by the server.

In the online phase, the client needs to replenish hints and crumbs. Hint replenishment is done in a different way as querying. The crumbs retrieved from hint server cannot be used at the hint server. Instead, the client samples a new PPRS containing the queried index i. After that i is punctured and a random index in the $\lfloor i/\sqrt{n} \rfloor$-th block is selected to make up the punctured block. The client then sends the set to the hint server. The hint server sends all the $\sqrt{n}$ records in plain to the client. The hint is assembled at the client from the records and $\mathcal{D}_i$. This process is described in algorithm 4.

The client cannot replenish crumbs directly from the hint server, for the crumb will always be in the same block as the queried index, which leaks information to the hint server. Instead, the crumb must be selected randomly. We adopt the techniques from single server sublinear PIR schemes [3]. The crumbs will not be updated after each single queries. Instead, after a collection of $\sqrt{n}$ queries, a new set of $\lambda\sqrt{n}$ crumbs will be fetched from the hint server. In each query 1 crumb will be consumed. The probability that some block would run out of crumbs before $\sqrt{n}$ queries is negligible.

Remark 1 (Queried Index Distribution)**.** As is stated in [3], the database could always be randomly permuted before any of the queries are made. On the other hand, we can always ask the client to save the last $\sqrt{n}$ query result, so that one index is not queried multiple times within a $\sqrt{n}$ queries period. If such a query is made, the client queries a random database element instead, and retrieve the record from the saved results. This will ensure that the queried index is uniformly distributed, which is necessary for the scheme to handle $\sqrt{n}$ queries successfully with overwhelming probability.

Remark 2 (Handling Changing Databases)**.** It is natural for applications to have a changing database. The optimization and technique involved in our scheme is compatible with former techniques handling database changes, such as Checklist [12].

Algorithm 1: Hint generate algorithm in the two server setting, run in the offline phase by the client.

Output : Client-side stored hints $Hint$

Samples $M = \lambda\sqrt{n}$ PPRS $sk_1, sk_2, \ldots, sk_M$ with size $\sqrt{n}$ and range $\sqrt{n}$

Samples $M = \lambda\sqrt{n}$ PPRS $sr_1, sr_2, \ldots, sr_M$ with size $\sqrt{n}$ and range p

Sends $sk_1, sk_2, \ldots, sk_M$ and $sr_1, sr_2, \ldots, sr_M$ to the server.

Receive $H = \{(hs_1, hr_1), (hs_2, hr_2), \ldots, (hs_M, hr_M)\}$ from the server.

Store hint as $Hint = (hs_1, hr_1, sk_1, sr_1), \ldots, (hs_M, hr_M, sk_M, sr_M)$

Algorithm 2: Hint generate algorithm in the two server setting, run in the offline phase by the hint server.

Input : Client query PPRS $sk_1, sk_2, \ldots, sk_M$ and $sr_1, sr_2, \ldots, sr_M$

Output : Hint answer H

Let $M = \lambda\sqrt{n}$

Let $H = []$

foreach $i \in [M]$ **do**

  Let $S = EvalSet(sk_i)$, $R = Eval(sr_i)$

  Let $hr = 0$, $hs = 0$

  foreach $j \in [\sqrt{n}]$ **do**

    $hs = hs + S_j$

    $hr = hr + R_j \cdot S_j$

  end

  Let $h = (hs, hr)$

  Append h to H

end

Send H to the client

Algorithm 3: Query algorithm in the two server setting, run in the online phase by the client.

Input : The queried index i
Output : The queried record $\mathcal{D}_i$ or $\perp$

Let $b = \lfloor i/\sqrt{n} \rfloor + 1$
Find t such that hint $h_t = (hs_t, hr_t, sk_t, sr_t), Eval(sk_t)_b = i$. Remove the hint. If no such sk_t exists, halt and output $\perp$
Find a crumb $c = (id, x)$ in C_b. Remove the crumb. Halt and output $\perp$ if crumbs runs out.
Sample a random number r from $[p]$.
Let $qs = EvalSet(Punc(sk_t, b)) \cup \{id\}$
Let $qr = Eval(Punc(sr_t, b)) \cup \{r\}$ `// The addtional item goes to the punctured point`
Send qs, qr to the query server.
Receive as, ar from the query server.
Let $res = hs_t - as + x$, $res_r = hr_t - ar + r \cdot x$.
if $res \cdot Eval(sr_t)_b = res_r$ **then**
    Output $\mathcal{D}_i = res$
else
    Output $\perp$
end

Algorithm 4: Hint replenish algorithm in the two server setting, run in the online phase by the client.

Input : The queried index i, the retrieved database record $\mathcal{D}_i$

Let $b = \lfloor i/\sqrt{n} \rfloor + 1$
Sample a random index id in the b-th block.
Sample a new set sk_{new} with size $\sqrt{n}$ and range $\sqrt{n}$, containing i.
Sample a new set sr_{new} with size $\sqrt{n}$ and range p.
Let $qs_{new} = EvalSet(Punc(sk_{new}, b)) \cup \{id\}$ `// The addtional item goes to the punctured point`
Send qs_{new} to the query server.
Receive $x_1, x_2, \ldots, x_{\sqrt{n}}$ from the query server.
Let $hs = \Sigma_{j \in [\sqrt{n}] \setminus b} x_j + \mathcal{D}_i$
Let $hr = \Sigma_{j \in [\sqrt{n}] \setminus b} Eval(sr_{new})_j \cdot x_j + Eval(sr_{new})_b \cdot \mathcal{D}_i$
Append $h = (hs, hr, sk_{new}, sr_{new})$ to hint storage.

Algorithm 5: Hint generation algorithm in the single server setting, run in the offline phase by the client.

Input : A database digest d

Invoke the crumb replenish algorithm in two server setting on the streamed database.
Samples $2M = 2\lambda\sqrt{n}$ PPRS $sk_1, sk_2, \ldots, sk_{2M}$ with size $\sqrt{n}$ and range $\sqrt{n}$
Samples $2M = 2\lambda\sqrt{n}$ PPRS $sr_1, sr_2, \ldots, sr_{2M}$ with size $\sqrt{n}$ and range p
foreach $i \in [\sqrt{n}]$ **do**
    foreach $j \in [\lambda]$ **do**
        Let $id = (i-1) \cdot \sqrt{n} + j$
        $sk_{id} = Punc(sk_{id}, i)$
        `/* The puncturing can be expressed by simply recording the punctured index, since the PPRS remains with the client and is never sent to the server before being promoted to a main hint. */`
    end
end
Initialize a total of $4M = 4\lambda\sqrt{n}$ parities $hs_1, hr_1, hs_2, hr_2, \ldots, hs_{2M}, hr_{2M}$ to 0.
Initialize the digest $hash$.
foreach $i \in [\sqrt{n}]$ **do**
    Download the i-th part of the database P_i from the server.
    Update $hash$ with P_i.
    foreach $j \in [2M]$ **do**
        Let $x = EvalSet(sk_j)_i$, $r = Eval(sr_j)_i$
        if $x \neq \perp$ **then**
            `// The block is not punctured`
            $hs_j = hs_j + \mathcal{D}_x$
            $hr_j = hr_j + r \cdot \mathcal{D}_x$
        end
    end
end
if $hash \neq d$ **then**
    Halt and output $\perp$
end
Store hint as $Hint = (hs_1, hr_1, sk_1, sr_1, \perp), \ldots, (hs_{2M}, hr_{2M}, sk_{2M}, sr_{2M}, \perp)$
`/* There's one extra item in the hint for hint replenishment. The item is also added to a query when the hint set is evaluated */`

Algorithm 6: Hint replenish algorithm in the single server setting, run in the online phase by the client.

Input : The queried index i, the retrieved database record $\mathcal{D}_i$

Let $b = \lfloor i/\sqrt{n} \rfloor$

Find a hint h from the backup hints for block b (hints with $id \in [M + b \cdot \sqrt{n} + 1, M + (b+1) \cdot \sqrt{n}]$)

Parse hint h as $h = (hs, hr, sk, sr, \perp)$

Let $hs' = hs + \mathcal{D}_i, hr' = hr + Eval(sr)_b \cdot \mathcal{D}_i$

Remove h from backup hints and append (hs', hr', sk, sr, i) it to main hints.

3.3 Single Server Verifiable PIR

Theorem 2. There is a single server offline/online verifiable private information retrieval scheme with correctness, privacy and strong integrity. On a database of size n, the scheme has:

- Offline communication complexity: $O(n)$
- Offline computation complexity: $O(\lambda n)$
- Online communication complexity: $O(\sqrt{n})$
- Online computation complexity: $O(\sqrt{n})$
- Ammortized computaion complexity: $O(\lambda\sqrt{n})$
- Support at least $\sqrt{n}$ queries.

To make the sublinear PIR schemes work in a single server setting, there are two major challenges:

1. There will be no online server for us to replenish hints.
2. Offline hints and oniline queries now goes to the same server, which is prohibited.

To replenish hints in the online phase, we adopt the idea of backup hints[3]. The client retrieve extra hints in the offline phase to replenish the hints consumed in the online phase. The hints are classified into main hints and backup hints. There will be as many main hints as there are in the two-server setting, and λ backup hints for each of the $\sqrt{n}$ blocks. A backup hint for block i is a normal hint with the i-th item punctured. After a query to block i is finished, a backup hint for block i is fetched. The client add the queried item to the backup hint and update the parities. This makes the backup hint an available main hint with exact $\sqrt{n}$ items. The scheme allows a client to make $\sqrt{n}$ queries without rerunning the costly hint retrieving phase. After that, the offline phase must be rerun. The crumbs will be replenished in the offline phase as well. The detail of the hint replenishment is shown in algorithm 6.

The second challenge is handled in two possible ways. [3] proposed to retrieve hints with homomorphic encryption and [17, 24] proposed to stream the entire database to the client.

We adopt the latter one, since it is more efficient in practice. The hint generation could be done in a block-wise streaming way. The client request one block from the server at a time, and update local hints accordingly. For each hint contains exactly zero or one item in each block. After receiving block i, the client iterate through all hints and update their parity with their item in the block. The block can then be discarded and a new block is fetched from the server. This makes the client storage sublinear in the database size. A detailed description is shown in algorithm 5.

In a malicious setting, the client needs to verify the integrity of the database when generating offline hints. The verification could be done in an simpler way since the entire database is accessible to the client. In such single-server situation, a trusted database owner could be obligated to publish a digest of the database, as is assumed in APIR [2]. The digest can be a secure cryptographic hash of the entire database. Details concerning how such digest is designed and implemented is out of the scope of this paper.

3.4 Security Analysis

The arguments here serves as an intuition for the security of our scheme. A formal proof is provided in B.

Two Server Scheme. We omit the unoptimized scheme here, since it merely serves as an introduction to the optimized scheme. The analysis applis only to the optimized scheme. As a reminder, we assume a honest-but-curious hint server and a malicious query server.

Correctness. The two situations in our scheme that the client does not learn the queried record are when the client cannot find a proper set conatining the queried index, or the block runs out of crumbs. By retrieving $M = \lambda\sqrt{n}$ hints and crumbs, both happen with negligible probability. From the algorithm it is clear that an honest answer from the server is always accepted. Thus the correctness error is negligible.

Integrity. We assume the client has already remove the crumb from the answers and hints. This always succeeds since the crumb is known to the client. We denote $r = Eval(sr)_i$ for ease of writing. r is a random element in F_{2^p}, and due to the privacy of PPRS, remains indistinguishable from any other elements in the field. If an adversary $\mathcal{A}_0$ act as S_{query} cheats successfully with non-negligible probability ε, which means it passes the verification with wrong a'_s and a'_r, it could construct two equations as follows:

$$\begin{cases} r \cdot (h_s - a_s) = h_r - a_r \\ r \cdot (h_s - a'_s) = h_r - a'_r \end{cases}$$

Within which a_s, a_r are the honest answer it is expected to answer with, a'_s, a'_r are the maliciously fabricated answer which pass the verification, h_s, h_r are the hint parities and should be unknown to $\mathcal{A}_0$.

Substracting the two equation yields

$$r(a'_s - a_s) = a'_r - a_r$$

Given the equations, $\mathcal{A}_0$ can easily solve r. Which mean there exists an adversary that can break PPRS privacy with advantage at least ε.

To achieve negligible integrity error, we need a large field for the elements. Namely, the field should be at least as large as 2^{λ} where λ is the security parameter.

Privacy. The query server does not affect how the client generates a query, thus it is sufficient to prove the standard privacy (without selective failure) in an honest-but-curious setting. In the offline phase, the hints and sets sampled are not related to the any specifc index. In other words, there're no private inputs from the client that need to be protected yet. In the online phase, each query from the client is indistinguishable from a newly sampled set with one random item in each block. The only difference between the set selected by the client and a random set is that it always contains the queried index. By switching the index to a random one, the set is identical to a random set.

More importantly, we show the scheme is not prone to selective failure attacks. The key observation here is that the malicious query server already learns the client output when it generates its reply. If the query server answers honestly, from correctness, the client will accept the answer and output a record. If the query server cheats, the only thing it can do is to modify the answers. From the integrity analysis, a false answer only passes the verification with probability 2^{-p}. The client output provide no extra information to the server except for probability 2^{-p}.

Single Server Scheme. The argument for our single server scheme is similar to the two server one. The offline hint generate and online query algorithm is almost identical to the single server scheme. The main concern remains is the impact of the hint replenishment.

Correctness. From a similar argument, the chance that the client cannot find a backup hint to replenish the hint is negligible. And since the chance of the client cannot find a proper set conatining the queried index, or the block runs out of crumbs are both negligible, the overall correctness error is negligible.

Integrity. Since the database is verified against a digest in the offline phase, the offline hints are correct except for negligible probability. With the correct hints at hand the online queries are identical to the two server scheme. From the same argument in the two server scheme, the integrity error is negligible.

Privacy. In the offline phase, a server only streams the database to the client and sees nothing. In the online phase, the transcript is identical to the query server of the two server scheme. From the same argument in the two server scheme, the privacy error is negligible.

3.5 Efficiency analysis

Two Server Scheme. In the offline phase, the client sends $M = \lambda\sqrt{n}$ PPRS to the hint server. Each PPRS can be represented by a short key of size λ. The crumbs can be fetched with an extra PPRS or simply using one of the M PPRS. The hint server in return sends $2M$ parities and $\lambda\sqrt{n}$ crumbs to the client. Assuming parities and crumbs are of the same size as one database record, the total offline communication would be $O(\lambda^2\sqrt{n})$ and the offline computation would be $O(\lambda n)$.

In the online phase, the client search for a hint that contains the queried index i, which takes $O(\sqrt{n})$ computation in expectation. The succesive processing of the hint, including puncturing and evaluating the set, also cost $O(\sqrt{n})$ computation. The query with a size of $O(\sqrt{n})$ is then sent to both servers. Each server computes the parities in $O(\sqrt{n})$ time. The hint server answers the query with $O(\sqrt{n})$ records and the query server answers the query with 2 parities. The client then verifies the answer in $O(1)$ time. The total online communication would be $O(\sqrt{n})$ and the online computation would be $O(\sqrt{n})$.

The crumbs will have to be refreshed after $\sqrt{n}$ queries, which takes $O(\lambda\sqrt{n})$ communication and computation. Overall the amortized communication would be $O(\sqrt{n})$ and the amortized computation would be $O(\sqrt{n})$.

Single Server Scheme. The one server in the single server scheme acts almost the same as the query server in the two server scheme. The online phase with the server is almost identical. The offline phase involves streaming the database to the client, which takes $O(n)$ communication and $O(\lambda n)$ computation. Unlike the two server scheme, the entire offline phase must be rerun after $\sqrt{n}$ queries. So the amortized communication would be $O(\sqrt{n})$ and the amortized computation be $O(\lambda\sqrt{n})$.

4 Evaluation

Implementation: We evaluated the performance of our construction and compared it with state of art two-server PIR, single server PIR and verifiable PIR schemes. For two-server PIR schemes, we choose DPF-PIR [7],TreePIR[13] and APIR [2]. For single server settings, we choose Simple PIR [10], Piano [24] and the LWE variant of APIR. We developed our implementation. [1].

We selected the parameters to ensure at least 128-bit security and maintained and $m = \lambda\sqrt{n}$ throughout the evaluation. All benchmarks are run single-threaded. For the two server schemes, the two servers are run in serial so that server time is doubled in the evaluation.

We conducted the benchmark on a 32-core AMD EPYC 7K83 CPU with 128GB RAM server. We run the benchmarks on five databases, with different record count and size. The results can be found in table 3 and 4.

[1]The code is available at https://github.com/asternight/vpir

	Database Parameters	Offline		Online	
		Comm.(MB)	Compute(s)	Comm.(KB)	Compute(ms)
DPF-PIR		-	-	0.58	3.3
TreePIR	2^{20} 32-byte entries	2.88	1.4	65.9	0.94
APIR(Merkle)	32 MB in total	-	-	1620	301
Crust		13	6.6	72	0.09
Crust(Without verification)		9	1.6	40	0.04
DPF-PIR		-	-	0.74	54.6
TreePIR	2^{24} 32-byte entries	11.53	25	262.6	4.6
APIR(Merkle)	512 MB in total	-	-	6473	4942
Crust		50	196	288	0.44
Crust(Without verification)		34	38	160	0.18
DPF-PIR		-	-	0.82	249
TreePIR	2^{28} 8-byte entries	11.53	275	262.6	16.6
APIR(Merkle)	2 GB in total	-	-	-	-
Crust		104	1828	640	1.46
Crust(Without verification)		50	475	256	0.61
DPF-PIR		-	-	0.89	842
TreePIR	2^{28} 32-byte entries	46.14	426	1049.6	20.6
APIR(Merkle)	8 GB in total	-	-	-	-
Crust		200	3470	1152	2.46
Crust(Without verification)		136	693	640	0.95
DPF-PIR		-	-	1.05	3372
TreePIR	2^{28} 128-byte entries	184.56	943	4198	29.4
APIR(Merkle)	32 GB in total	-	-	-	-
Crust		776	9411	4224	6.80
Crust(Without verification)		520	1577	2176	1.26

Table 3: Comparison of two-server PIR schemes. The online phase and offline phase are benchmarked separately. DPF-PIR and APIR do not have a client-wise offline phase. We are unable to run APIR on databases holding 2^{28} entries. It consumes more than 128 GB RAM.

4.1 Two server schemes

On the performance of APIR. Similar to our schemes, APIR provide integrity, but at a much larger cost. For two-server schemes, Merkle Tree based APIR is slower and require larger communication than our scheme. The scheme is much slower than DPF-PIR for its database expansion. Merkle Tree based APIR has a considerable database expansion, we are unable to run the the scheme on databases with 2^{28} records due to insufficient RAM.

On the performance of DPF-PIR. DPF-PIR serves as the baseline for a linear PIR scheme. It has a very small communication overhead, but suffers from linear online computation. On large databases, it's efficiency degrades considerably.

On the performance of TreePIR. TreePIR is a sublinear PIR scheme. Our scheme improves upon TreePIR in that, it does not require the server to calculate $\sqrt{n}$ possible results. In addition to that, the instantiating of the PPRS of our scheme is much simpler. In a sense, we do not actually require the PPRS to be effectively "puncturable", since after puncturing a PPRS is always represented in plain. So, the complicated tree-shaped PPRS can be avoided in our scheme. This brings around 7x better performance in the online phase.

4.2 Single server schemes

On the performance of SimplePIR. SimplePIR is similar to a sublinear PIR scheme for that it requires the server to send a "hint" to the client as well. Though the hint can be calculated in advance, a server-client communication overhead cannot be avoided. The rest of SimplePIR is quite straightforward. It serves as a baseline for a linear PIR scheme, just like DPF-PIR. On larger databases, it is outperformed by sublinear PIR schemes.

On the performance of Piano. Piano is a sublinear PIR scheme. Our scheme outperform Piano in the online phase since there's no need to calculate those "dummy answers". We would like to emphasize again that the datas of our scheme are collected with verification enabled and a conservative parameter. Which means in a "fairer" comparison, where both schemes are run under the same parameter and with verification removed, our scheme will be concretely faster.

	Database Parameters	Offline		Online	
		Comm.(MB)	Compute(s)	Comm.(KB)	Compute(ms)
SimplePIR		20	-	40	12.0
Piano	2^{20} 32-byte entries	32	13.6	39	13
APIR(LWE)	32 MB in total	19200	-	1024	5565
Crust		32	11.7	36	0.06
Crust(Without verification)		32	5.0	4	0.03
SimplePIR		84	-	168	82.9
Piano	2^{24} 32-byte entries	512	262	159	65
APIR(LWE)	512 MB in total	76800	-	4096	23071
Crust		512	196	144	0.31
Crust(Without verification)		512	89	16	0.12
SimplePIR		169	-	338	255
Piano	2^{28} 8-byte entries	2048	4881	255	302
APIR(LWE)	2 GB in total	9600	-	4096	24026
Crust		2048	2721	320	0.98
Crust(Without verification)		2048	1235	64	0.39
SimplePIR		344	-	688	899
Piano	2^{28} 32-byte entries	8192	5319	639	329
APIR(LWE)	8 GB in total	38400	-	16384	96104
Crust		8192	3331	416	1.63
Crust(Without verification)		8192	1432	80	0.66
SimplePIR		-	-	-	-
Piano	2^{28} 128-byte entries	32768	6062	2175	377
APIR(LWE)	32 GB in total	153600	-	65536	384416
Crust		32768	8774	2112	4.86
Crust(Without verification)		32768	1804	64	0.66

Table 4: Comparison of single-server PIR schemes. The online phase and offline phase are benchmarked separately. SimplePIR does not have a client-wise offline computation, but a hint is transfered to the client. On the largest database our server runs out of RAM for SimplePIR. The data of the LWE variant of APIR is calculated by the cost of retrieving one bit times the entry width.

On the performance of APIR. The LWE variants of APIR support only 1-bit retrieval. Its performace degrades drastically when the database items are large, since multiple instances of the scheme are needed to retrieve one database item. Similar to SimplePIR, it has a large hint to be transfered to the client in the offline phase. The performance of the scheme is very limited on databases with large records that even if we only run one query after the offline phase, our scheme still outperforms the LWE variant of APIR.

On the performance of Crust. Our schemes outperforms both the state of art verifiable PIR schemes and sublinear PIR schemes. Our schemes are faster than APIR and require less communication. The high offline overhead almost completely comes from the choice of M. We set λ be 128 and fix $M = \lambda\sqrt{n}$ throughout the evaluation. The parameter is rather conservative in implementation. As a comparison, TreePIR chooses $M = \log 2 \cdot \lambda\sqrt{n} \approx 0.3\lambda\sqrt{n}$ and Piano stores $M = 4\log n\sqrt{n}$ main hints and backup hints. The cost of offline phase is almost completely propotional to M. Choosing a smaller M is possible, but it troubles us in proving privacy in a malicious setting. Such a large M is overkill for most cases. In a semi-honest case, one could choose a smaller M, at a cost of increased correctness error.

In the online phase the computation of both servers is minimal—They merely fetch $\sqrt{n}$ entries and calculate their sum and weighted sum. The results are benchmarked with and without verification as a comparison. The overhead of galois field computing is considerably large. If we remove the verification from our scheme (which involves the random parities and the random set), the offline computation is further reduced by $75\% - 83\%$ and online computation reduced by $56\% - 82\%$. Which means in a "fair" comparison to TreePIR when both schemes are in a semi-honest scenario, our scheme will be around 22x faster than TreePIR in the online phase. The verification requires a constant λ bits overhead for each entry, which is not a problem for large entries. But for small entries, the overhead could be considerable. For ease of implementation, in our code both the entries and the random parities are represented with a $max(\lambda, size)$ element. The communication can be further reduced by using a constant λ-bytes random parity.

5 Related work

5.1 Verifiable PIR

The concept of Private Information Retrieval (PIR) in a malicious setting is not very innovative. [8] mentioned the concept, and there have been a body of work, such as [1, 6, 21, 23], focusing on it. These works are based on previous PIR constructions with online computation of $O(n)$ complexity.

Zhang et al.[21] developed their scheme upon the Woodruff-Yekhanin PIR Scheme and provided accountability. Their scheme has little communication overhead compared to previous verifiable PIR schemes, and supports database updates. They also proposed a simple solution for verifiability in a single-server PIR, although the solution is too expensive for a PIR scheme.

Zhao et al.[23] was the first to propose a verifiable single-server PIR with efficiency comparable to some typical single-server PIR schemes. However, their construction focuses on the case where the server remains committed to a predefined database and tries to break the encryption scheme.

Colombo et al.[2] gives a thorough definition of verifiable PIR (referred to as authenticated PIR). They propose a simple construction for multi-server verifiable PIR and an interesting construction for predicate queries utilizing function sharing. They also give constructions for single-server verifiable PIR.

5.2 Sublinear PIR

There are a few improvements cannot be directly applied to our scheme.

The membership testing problem was first addressed in CK20 by instantiating the PPRS with a pseudorandom permutation (PRP) which enables fast membership test. The cost is that, PRP is not puncturable, so in the online phase the client would have to transmit the queries in plain, resulting in $O(\sqrt{n})$ communication.

Checklist. To avoid running parallel instances, a first approach was proposed by Kogan and Corrigan-Gibbs in Checklist[12]. The idea is to retrieve extra information from the servers and ensure even when the desired item is punctured in query to the query server, the client could recover it from the extra information from the hint server. We refer readers to their work for more details. The key point is that, in their construction, the concept of hint server and query server is mixed. The assumption we rely upon, that we only learn new hints from the hint server, is no longer true. In additon, the scheme requires the client to send dummy queries that cannot be verified, which makes the scheme prone to selective failure attacks.

TreePIR. Lazzaretti and Papamathou in TreePIR [13] propose a brand new way of constructing the PPRS, along with a new primitive called weakly puncturable pseudorandom set (WPPRS). They are the first to introduce the "divide and sample" expression used in our scheme.

The scheme is promising except that it yield $\sqrt{n}$ results, within which only one is the real answer and the rest are dummy answers. The client has no way to verify the dummy answers, which makes the scheme prone to selective failure attacks.

6 Conclusion

Verifiable PIR has expanded the scope of PIR applications by enabling clients to protect their privacy and data integrity under malicious settings. In this paper, we present a new sublinear PIR scheme, with both better performance and verifiability. The scheme outperforms state-of-the-art verifiable PIR schemes and sublinear PIR schemes. We believe that our work will further extend the boundaries of PIR and its applications.

References

[1] Amos Beimel and Yoav Stahl. Robust information-theoretic private information retrieval. 20(3):295–321.

[2] Simone Colombo, Kirill Nikitin, Henry Corrigan-Gibbs, David J. Wu, and Bryan Ford. Authenticated private information retrieval. Cryptology ePrint Archive, Paper 2023/297, 2023. https://eprint.iacr.org/2023/297.

[3] Henry Corrigan-Gibbs, Alexandra Henzinger, and Dmitry Kogan. Single-server private information retrieval with sublinear amortized time. In *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part II*, pages 3–33. Springer, 2022.

[4] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, volume 12105, pages 44–75. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[5] David Dagon, Manos Antonakakis, Kevin Day, Xiapu Luo, Christopher P Lee, and Wenke Lee. Recursive dns architectures and vulnerability implications. In *NDSS*, 2009.

[6] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 269–283, Bellevue, WA, August 2012. USENIX Association.

[7] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology – EUROCRYPT 2014*, pages 640–658, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[8] Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. page 18.

[9] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with popcorn. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 91–107, Santa Clara, CA, March 2016. USENIX Association.

[10] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022. https://eprint.iacr.org/2022/949.

[11] Mehmet Sabir Kiraz and Berry Schoenmakers. A protocol issue for the malicious case of yao's garbled circuit construction. 2006.

[12] Dmitry Kogan and Henry Corrigan-Gibbs. Private blocklist lookups with checklist. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 875–892. USENIX Association, August 2021.

[13] Arthur Lazzaretti and Charalampos Papamanthou. Treepir: Sublinear-time and polylog-bandwidth private information retrieval from ddh. Cryptology ePrint Archive, Paper 2023/204, 2023. https://eprint.iacr.org/2023/204.

[14] Rasoul Akhavan Mahdavi and Florian Kerschbaum. Constant-weight PIR: Single-round keyword PIR via constant-weight equality operators. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1723–1740, Boston, MA, August 2022. USENIX Association.

[15] Samir Jordan Menon and David J. Wu. Spiral: Fast, high-rate single-server pir via fhe composition. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 930–947, 2022.

[16] Muhammad Haris Mughees, Sun I, and Ling Ren. Simple and practical amortized sublinear private information retrieval. Cryptology ePrint Archive, Paper 2023/1072, 2023. https://eprint.iacr.org/2023/1072.

[17] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1002–1019, 2018.

[18] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable pseudorandom sets and private information retrieval with near-optimal online bandwidth and time. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology – CRYPTO 2021*, volume 12828, pages 641–669. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[19] Frank Wang, Catherine Yun, Shafi Goldwasser, Vinod Vaikuntanathan, and Matei Zaharia. Splinter: Practical private queries on public data. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 299–313, Boston, MA, March 2017. USENIX Association.

[20] Ke Coby Wang and Michael K. Reiter. Detecting stuffing of a User's credentials at her own accounts. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2201–2218. USENIX Association, August 2020.

[21] Liang Feng Zhang and Reihaneh Safavi-Naini. Verifiable multi-server private information retrieval. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *Applied Cryptography and Network Security*, volume 8479, pages 62–79. Springer International Publishing. Series Title: Lecture Notes in Computer Science.

[22] Fangming Zhao, Yoshiaki Hori, and Kouichi Sakurai. Two-servers pir based dns query scheme with privacy-preserving. pages 299–302, 11 2007.

[23] Liang Zhao, Xingfeng Wang, and Xinyi Huang. Verifiable single-server private information retrieval from LWE with binary errors. 546:897–923.

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

A A Possible Strategy for Hint Server

Suppose we've chosen a set key sk_j to query $\mathcal{D}_i$, which means $i \in Eval(sk_j)$. hs_j, hr_j could be generated with a false $\widehat{\mathcal{D}_i} \neq \mathcal{D}_i$. Notice that i is possibly punctured in the query. The client will reconstruct $\widehat{\mathcal{D}_i}$ unknowingly.

The verification we introduced is not involved here. Since the hints are generated "genuinely", but with a false data source. Although the cheating may be detected in a query that uses sk_j but does not puncture i. Unfortunately, the probability of this happening is not high enough, and surprisingly, cannot

be effectively increased by running parallel instances. We demonstrate the fact with the following adversary strategy:

Consider an adversary $\mathcal{A}$ playing the role of S_{hint}. $\mathcal{A}$ selects an index i and a random value $\widehat{\mathcal{D}_i} \neq \mathcal{D}_i$ before running the scheme. Whenever D_i is accessed, $\mathcal{A}$ use $\widehat{\mathcal{D}_i}$ instead of $\mathcal{D}_i$.

If the client decide to query $\mathcal{D}_i$, it will be deceived in one instance with probability $1 - \frac{s-1}{n}$ (the probability of selecting a set containing i). For one query, the queried index remains the same across all parallel instances. Therefore, if the client queries $\mathcal{D}_i$, it will consistently select poisoned PPRS keys. Since the probability of being deceived for the client in one instance is $1 - \frac{s-1}{n}$, which is close to 1, running parallel instances does not provide significant benefits.

From a similar argument, the scheme is prone to selective failure attack when the hint server is malicious. Although the queries to the hint server are independent from the queried index. The mere fact that the client fails a query is informative enough for the hint server. That means the client just used a "poisoned" hint and that narrows down the queried index to one hint set.

B Security Proof

We will focus on the two server protocol. As is introduced in the brief analysis part, the single server protocol can be easily reduced to the two server protocol, with minimal modification to the proof.

B.1 Correctness

There are two major part concerning correctness, namely

1. The probability that the client cannot find a hint containing the queried index
2. The probability that a certain block runs out of crumbs

We show that both probabilities are negligible. For the first one, we have the following lemma:

Lemma 1. Each hint contains any specific index i with probability $\frac{1}{\sqrt{n}}$.

Proof. Each hint contains exactly one random element in each block of size $\sqrt{n}$. Any specific index falls into one block and is selected with probability $\frac{1}{\sqrt{n}}$.

The first situation happens when the client cannot find a hint containing the queried index. The probability is at most $(1-\frac{1}{\sqrt{n}})^M$. With $M = \lambda\sqrt{n}$, $(1-\frac{1}{\sqrt{n}})^{\lambda\sqrt{n}} < e^{-\lambda}$.

For the second probability, a block runs out of crumbs if more than λ queries falls into it. With $M = \lambda\sqrt{n}$, we prove the following lemma

Lemma 2. The probability that more than λ queries fall into one block is negligible.

Proof. This part of the proof comes from [3]. Consider a list of $Q = \sqrt{n}$ queried indexes $i_1, i_2, \ldots, i_Q$. As we stated in the construction, we assume these index are randomly distributed and distinct. We may also assume that $\lambda < \sqrt{n}$, otherwise the proof would be trivial. Consider a subset $I \subseteq \{i_1, i_2, \ldots, i_Q\}$ of size λ. Let C_I be the event that all queries in I fall into the same block j. We have

$$Pr[C_I] = (\frac{\sqrt{n}}{n})(\frac{\sqrt{n}-1}{n} - 1)\cdots(\frac{\sqrt{n}-\lambda}{n} - \lambda)$$

Each term in the product is at most

$$\frac{\sqrt{n}}{n-\lambda} \leq \frac{\sqrt{n}}{n-\sqrt{n}} = \frac{1}{\sqrt{n}-1}$$

Therefore, $Pr[C_I] \leq (\sqrt{n}-1)^{-\lambda}$. There are at most

$$\binom{Q}{\lambda} \leq (\frac{eQ}{\lambda})^{\lambda}$$

choices of the set I, by the union bound, the probability that there exists such a set I is

$$(\frac{1}{\sqrt{n}-1} \cdot (\frac{eQ}{\lambda}))^{\lambda} \leq (\frac{6}{\lambda})^{\lambda}$$

Which is negligible in λ. Taking a union bound over all $\sqrt{n}$ blocks completes the proof.

The proof for the single server protocol is similar. The only difference is the backup hints. It it easy to see that the backup hints are identical to crumbs, so the probability that a block runs out of backup hints are also negligible.

B.2 Integrity

The general idea of proving integrity has been shown in the security analysis part. We will give a formal proof here. We first prove a useful lemma.

Lemma 3. In the two server version of Crust, for any non-zero offset $\Delta = (\Delta_s, \Delta_r) \in F_{2^p}^2$, the following probability give the chance that the client accepts a false answer $a' = a + \Delta$ from the query server.

$$Pr\left[c \neq \perp : \begin{array}{rcl} (ck, q_h) & \leftarrow & Setup(1^\lambda, n) \\ h & \leftarrow & Hint(\mathcal{D}, q_h) \\ q & \leftarrow & Query(ck, i) \\ a & \leftarrow & Answer(\mathcal{D}, q) \\ c & \leftarrow & Reconstruct(h, a+\Delta) \end{array}\right] \leq \frac{1}{2^p - 1}$$

The statement holds also for the successive queries.

Proof. Let r be the random element in F_{2^p} that is punctured from the random hint. In short, the reconstruction part can be rewritten as

$$Pr[r \cdot (h_s - a_s - \Delta_s) = h_r - a_r - \Delta_r]$$

Within which a_s, a_r are the answers, h_s, h_r are the hint parities. Since the true answers always pass the verification, we have

$$r \cdot (h_s - a_s) = h_r - a_r$$

So the probability can be further simplified as

$$Pr[r\Delta_s - \Delta_r = 0]$$

Noting that r is a random element from PPRS and is indistinguishable from a hidden random value in F_{2^p}. The equation is the evaluation of a non-zero degree-1 polynomial at a random point r. Since a degree-1 polynomial has at most 1 root in F_{2^p}, the probability is at most $\frac{1}{2^p-1}$.

With lemma 3 at hand, integrity follows directly. The single server version shares the same proof, with the only difference being we need to introduce an extra part for the offline setup to ensure the hints are correct. The proof can be done via a basic argument on cryptographic hash functions and we omit it here.

B.3 Privacy

There are two parts in the privacy proof. We first prove that the scheme satisfy privacy under semi-honest setting. Then we move on to the malicious setting and prove the scheme is not prone to selective failure attacks.

We prove the semi-honest privacy by showing that the client's queries are indistinguishable from a random set with one random element in each block. To show that, we prove the following lemma

Lemma 4. For any query q received by the server and any two queried indexs i, i',

$$Pr[q|i] = Pr[q|i']$$

Proof. q can be divided into set qs and qr. Every index except i and i' in queried set qs are generated by a PPRS. Since the PPRS is indistinguishable from a random set, these indexes are indistinguishable from elements in a random set. For the block that contains i or i', the block is resampled with a fresh crumb. A crumb is always a random element in the block. Therefore, the block is also indistinguishable from a random set. The distribution of qs is irrelevent to i and i'. Aside from the queried set qs, the random query qr is merely $\sqrt{n}$ random numbers that has nothing to do with the queried index. The lemma holds.

The scheme allows $\sqrt{n}$ queries. To extend above proofs to subsequent queries, it is necessary to show the distribution of hints remains the same after each query.

Lemma 5. After each query, the client-side hints h follows the same distribution, which is indistinguishable from M random set with one random element in each block.

Proof. The hints are independent from each other. It is sufficient to show that during each query, the consumed hint and the replenished hint follows the same distribution. Recall that in a query to i which falls into the j-th block, the found query set sk_t and the newly sampled set sk_{new} are both uniformly random in every other block than j, and contains i in the j-th block, which directly gives us the lemma.

From lemma 5, we are confident to say correctness, integrity and semi-honest privacy applies to multiple queries. We now move on to the malicious setting.

Discussing privacy in a malicious setting, we first quote from CK20[4], *The client's queries depend neither on the hint nor on any of the servers' past answers. For this reason, the hint and the servers' answers do not play a role in the privacy games we define. Thus, actively malicious behavior by the servers, such as responding incorrectly to a client query, cannot help them break the PIR privacy property.*

This enables us to consider only the selective failure part of the malicious privacy. Namely, the argument shows that the scheme satisfies privacy when the server is not given the query result. Recall that we assume the servers to learn one extra bit indicating whether the answer is accepted.

From lemma 5, we can assume a scenerio during the online phase of the scheme. From a specific point in the online phase (including right after the offline phase finishes), the query server is replaced by an adversary $\mathcal{A}(f)$. The adversary is given access to an algorithm f, which denotes the "next message" function, which enables the adversary to dynamicly choose the answer. For ease of the writing, we assume f would also output a bit b indicating whether the answer is correct. This does not introduce extra information since the adversary can always get the bit by comparing the output of f and the genuine answer calculated on the database $\mathcal{D}$. We prove that

Lemma 6. Let $REAL_{\mathcal{A}}(f)$ be a random variable representing the view of adversary $\mathcal{A}$ in one query. Let $SIM_{\mathcal{A}}(f)$ be the ideal simulation of the protocol where the output of the protocol is selected by the simultor rather than given by the client. There exists a probabilistic polynomial time similator SIM, for all $f, \mathcal{A}$

$$SIM_{\mathcal{A}}(f) \equiv REAL_{\mathcal{A}}(f)$$

Proof. Assume such a simulator, it receive the bit b from f and use exactly the bit as the client result provided to the servers. Now we show that such a simulation is indistinguishable from the real world. Consider the difference between b and the real response b' from the client. The client always accept a true answer. The only situation that $b \neq b'$ is when the client accepts a false answer. From lemma 3, the probability that the client accepts a false answer is negligible. Therefore, the difference between b and b' is negligible. The simulation is indistinguishable from the real world.

The proof naturally extends to the single server scheme, since the server plays the same role as the query server in the two server scheme.