Note discovery proposal [RFP]

Thanks to Khashayar Barooti, Phil Windle, Michael Connor, and Patrick Towa for discussions and feedback

Summary

Below we outline an approach for transaction retrieval that we think is feasible to be implemented without restrictive costs. Specifically, the proposal outlines how users can share a secret via aztec to generate tags. Each transaction is then tagged with a “tag” only derivable by the sender and the receiver and is unique for the transaction. We then rephrase private transaction retrieval as private information retrieval, where a user wants to retrieve transactions corresponding to optimistically generated tags without leaking information of its query. We propose HintlessPIR as the single server PIR scheme that can be used today and further justify our choice with concrete estimations on multiple databases storing transactions collected over different time duration.

Tags

The motivation for tags is straight-forward: “If Alice and Bob know a secret, they can deterministically generate seemingly random values that only either of them can identify as not random”.

The way we intend to use tags in Aztec follows from the following simple example: Let’s say Alice met Bob and handed over a note with value x

on it. Now after few hours Alice wants to send a transaction to Bob. Alice picks a publicly known hash function H(.)

and calculates the tag = H(x + 1)

. Then she publishes the tuple (tag, tx)

. Note that Bob cannot know beforehand how many transaction did Alice send. Thus, to find all possible transactions that Alice could have sent, (1) Bob optimistically generates a set of tags as {H(x+1), H(x+2), ..., H(x+i)}

, where i

is some high value, (2) downloads all tag & transaction tuples (3) Decrypts only the transactions of which the attached tag is in the set Bob generated.

But Bob still had to download ALL tag & transaction tuples, which is no improvement (+worse) over downloading all published transactions and trial decrypting them. Hence, it is important to note that “tags” are only helpful if there exist a way for Bob to retrieve transactions corresponding to tags privately without having to download all tag & transaction tuples. We propose HintlessPIR, a single server PIR scheme, that can be used to achieve this in practice today. But note that Bob can use any other private retrieval mechanism (even the ones with less privacy guarantees) as per convenience.

Before we proceed, we need to solve two more problems with the “tags” based approach outlined above:

1. It is unrealistic to assume Alice and Bob can meet in person. They may want to remain anonymous.
2. In some cases Alice may want to update Bob’s private state with a transaction intended for him but would not want to reveal the transaction itself to Bob.

Assuming reason for (1) is obvious we will motivate (2) with the following example: Let’s say Alice runs a dex and Bob has submitted a request to trade X of token A for Y of token B. Alice can mint herself X of token A if she proves that Y of token B have been minted for Bob. Alice mints Y of token B for Bob in a new note and mints herself X of token A in another note. She sends both the notes to the validators. But she does not send the note that mints Bob Y of token B to Bob. She can then extort Bob to send the note.

To overcome (1) and (2), we will require users to share secrets via Aztec and require users to prove that tags are generated correctly upon transaction creation. Thus, we modify the above procedure to:

Let Bob set a random public nonce N

and threshold constant t

. Alice can only generate a tag as H(x|z)

where z - N < t

. This means Alice can only send as many as t

transactions until Bob updates his public nonce. Bob always searches for all possible values in range [N, N + t)

to find transactions from Alice.

Let there also a be a public nullifier tree T_n

that only stores encryption of shared secrets. Before Alice can send transactions to Bob, she is required to

1. generate a shared secret x

and encrypt it using Bob's public key pk_{bob}.

1. Add enc(x, pk_ {bob})

to T_n

.

Alice needs to add enc(x, pk_{bob})

to T_n

only once. After which she can send as many transaction as she wants to Bob.

For Alice to send a transaction Bob, she:

1. Generates a proof \pi

that enc(x, pk_{bob})

exists in tree. She can request necessary hash path in plain. To remove the trace that she's sending transaction to Bob again after X

minutes, she must request hash paths for all her known contacts at once. This should be cheap in plaintext.

1. She samples a value k \in [N, N + t)

iteratively until she finds a k

that she has not used to send transaction to Bob before.

1. She produces the tag as H(x | k)

. She also generates a proof that tag is formed correctly.

1. She publishes the tag & transaction tuple: (H(x|k), tx)

.

1. Validators accept tx

in (H(x|v), tx)

only if H(x|k)

is unique and \pi

is valid.

Bob trial decrypts all notes in nullifier tree T_n

to find secret x

. Then to find all transactions that Alice sent him, Bob:

1. optimistically generates set of hashes OT = {H(x|v) | v \in [N, N+t)}
2. requests transactions privately from the server corresponding to each hash in the set OT

.

1. If H(x|v)

corresponding to some v

returned a transaction, Bob stores v

in “the used” set just for future convenience.

1. If Bob thinks that unused values in [N, N + t)

are few for anyone of his contacts. Bob updates N

to some other random value.

Tags vs Clues

It is important to note the advantage of clue based approaches (for example,OMR, FMD) is that they guarantee that the receiver will always find transactions meant for them without having to rely on other mechanisms. Whereas in the tag based approach we require to design an additional mechanism, the one outlined above, to assure that receiver can always find transactions intended for them. However, if the additional mechanism isn’t too much of an overhead for the protocol, then tag based approach may fare better for users/protocols. This is because tag based approach is generic over any way of retrieving data corresponding to keys privately (for example, single-server pir, multi-server pir, TEEs, etc.), whereas all clue based approaches are tied to one specific cryptographic design.

Tag Chaining

Special thanks to Khashayar Barooti and Patrick Towa for idea of “Tag chaining” and helpful discussions

In this section we present another way of Alice and Bob deriving a list of tags for the notes Alice sends to Bob. The idea is fairly simple; let us describe it:

1. Alice and Bob keep a value \textsf{curr_tag}_{A,B}

as the current tag for communication between them. This value is initialised as \bot

.

1. When sending a note to Bob, Alice checks if \textsf{curr_tag}_{A,B} = \bot

. Let us handle the case where this is true first: * In case this is true, Alice picks a uniform x

and encrypts it with Bob’s public-key and publishes that on the handshake tree T_n

and sets \textsf{curr_tag}_{A,B}\gets H(x)

(where H

will be modeled as a random oracle)

- Bob checks \textsf{curr_tag}_{A,B} = \bot

and if this is true trial decrypts the ciphertexts in T_n

to recover x

and sets \textsf{curr_tag}_{A,B} \gets H(x)

1. In case this is true, Alice picks a uniform x

and encrypts it with Bob’s public-key and publishes that on the handshake tree T_n

and sets \textsf{curr_tag}_{A,B}\gets H(x)

(where H

will be modeled as a random oracle)

1. Bob checks \textsf{curr_tag}_{A,B} = \bot

and if this is true trial decrypts the ciphertexts in T_n

to recover x

and sets \textsf{curr_tag}_{A,B} \gets H(x)

1. If \textsf{curr_tag}_{A,B} = x

, to send a note to Bob, Alice picks a new uniform tag x'

and publishes (\textsf{curr_tag}_{A,B}, tx, Enc(pk_B, x'))

and updates \textsf{curr_tag}_{A,B}\gets H(x')

1. Bob recovers the note (\textsf{curr_tag}_{A,B},tx,\textsf{ct'})

using \textsf{curr_tag}_{A,B}

, decrypts \textsf{ct}'

to recover x'

. He updates the current tag accordingly \textsf{curr_tag}_{A,B}\gets H(x')

There are a couple of issues that need to be addressed with this approach that we will talk about. But let us first discuss the positives.

- This method removes the need of Bob optimistically generating a set of tags that he might receive notes with.
- Bob will receive the notes in order, as he will always only have the tag for the next note he's supposed to receive.

What are the issues with the approach then?

1. a not very important issue is that Bob can not send batch requests to retrieve several notes at the same time, because he only finds out about the next tag, after decrypting the current note.
2. a more problematic issue is that an adversary can pick a random note (tag, tx, \textsf{ct})

and reissue the note with a different \textsf{ct'}

to break the tag chain. This is resolvable by the validators not allowing tag repetitions or with a signing mechanism. The issue with signing is, the Alice's verification key should be only handed to Bob as otherwise everyone could see which notes Alice has issued.

1. arguebly the most important thing to be addressed is that although Enc(pk_B, x)

does not leak information about x

, it might leak information about Bob. The most severe case is if the ciphertext leaks the identity of Bob, but even if the adversary can figure out which notes "belong to the same person", this is a privacy concern.

[5] addresses the third issue using re-randomized publickeys to achieve a key-indistinguishability property. Let us explain this in a bit more detail.

Key-Randomizable PKE

A key-randomizable PKE is a tuple of algorithms (\textsf{REncKGen,REnc, RDec, Rand})

, which are the key generation, encryption, decryption, and key rerandomization procedures respectively. We say the scheme is (perfectly) correct if:

\textsf{Pr}\left[\textsf{RDec}(esk,ct) = pt \middle\vert \begin{array}{ll} epk,esk\gets \textsf{REncKGen}(1^\lambda)\ epk' \gets \textsf{Rand}(epk) \ct \gets \textsf{REnc}(epk',pt) \ \textsf{}\end{array}\right] = 1

As mentioned before the property we're looking for is key-indistinguishability, meaning given two public key/ secret key pairs (epk_0,esk_1) , (epk_1,esk_1)

, if we pick b\in {0,1}

randomly and rerandomize epk_b

, epk'\gets \textsf{Rand}(epk_b)

, an adversary would not have a significant advantage guessing b

, given epk'

.

Apart from key-indistinguishability we require IND-CPA security. Let us now describe a constrcution.

- \textsf{REncKGen}(1^\lambda)

picks a random x,r

from \mathbb{Z}_q

and computes (y_1 = g^{xr}, y_2 = g^r)

and outputs esk = x

and epk = (y_1,y_2)

- \textsf{Rand}(epk)

on input (y_1,y_2)

picks a random r'

from \mathbb{Z}_q

and outputs epk'\gets (y_1^{r'} , y_2^{r'})

- \textsf{REnc, RDec}

are usual ELGAMAL encryption/decryptions. To encrypt m

with rerandomized public-key (y'_1,y'_2)

we compute ct = ({y'}_1^r, {y'}_2^rm)

for a random r

. And to decrypt a ciphertext ct_0,ct_1

we compute ct_2ct_1^{-1/x}

.

The key-indistinguishability follows from DDH.

How does this solve the issue mentioned? When publishing the new note (tag, tx, ct)

, instead of using pk_B

to encrypt the new uniform tag, Alice first rerandomizes Bob's key pk_B' \gets \textsf{Rand}(pk_B)

and uses pk_B'

to encrypt the new tag.

Retrieving transactions privately

For Bob to find his transactions, he needs a way to retrieve transactions corresponding to tags in optimistically generated set privately. Bob's service provider can choose to offer several solutions that offer different trade offs between bandwidth, latency, and costs. For example, to accommodate for clients with low bandwidth and preference for near-free costs, service provider may choose to serve requests in plain (or ask clients to hide behind mixnets/TOR). Service provider can even choose to use TEEs/Multi-server PIR if their clients are comfortable with it. For clients that require high privacy, have a good internet connection, and can afford to pay some amount of fee for transaction retrieval, server will require a single-server private information retrieval scheme. We believe the service provider will face most difficulty in designing solution for the last set of users, thus we narrow our scope to describing a single-server PIR scheme that we think is cost-effective, sufficiently low bandwidth (compared to other Single-server PIR schemes), and fast.

It will be helpful to dissect Single-server PIR literature into 3 categories:

1. Levelled-HE based approaches
2. Linear HE based approaches
3. Stateful-PIR based approaches

Levelled HE based approaches provide low client-server communication (O(\log{N})

) but at the expense of high server-side compute and request-latency. This is because they use levelled HE (for ex, BGV/BFV) and server runtime complexity is O(N)

. Linear HE based approaches have O(\sqrt{N})

client-server communication, but Linear HE based approaches are faster than levelled HE based approaches (& in some cases approach memory bandwidth of CPU) thus offer low server side compute & request -latency. One can think both (1) & (2) as state-less PIR approaches. Stateless-PIR approaches differ from stateful-PIR approaches because the latter requires client-side compute. Although stateful-PIR approaches have low server-side compute (O{\sqrt{N}})

and low client-server communication (O\log{N})

, they require client to perform a pre-processing step in which the client downloads the entire database.

The transaction database isn't static, thus we can easily rule out (3) stateful-PIR based approaches as suitable for our purposes. Moreover, we would want to avoid high server costs. Thus it will be better to find a approach in the land of Linear HE based approaches (2) than in Levelled-HE based approaches (1).

Note:

Apart from high server-side costs & request latency, another reason why Levelled-HE based approaches are undesirable compared to Linear HE based approaches is that former requires server to store client-specific state. This means before a client makes any query, they should upload some keys (big in size) to the server. Server can then re-use client keys for any no. of client requests. This implies server always learns the timings at which a client makes a query. If we were to eliminate the client-specific state, then the client will have to upload new keys for each query, thus making per query client-server communication very expensive.

SimplePIR[1] is a Linear-HE based PIR scheme that has client-server communication of O(\sqrt{N})

and has fast server-runtime, thus low server-costs and request latency. However, the client needs to download database dependent hint to be able to make queries to the server. One can structure the database in such a manner that elements that change in hint due to any update are linear to the size of the update, but this approach becomes complex and undesirable if the database updates frequently. A recent published paper by Li et al., 2023 introduces HintlessPIR which overcomes the limitation of SimplePIR. Using HintlessPIR[2] one can enjoy low server-runtime & request latency (albeit a bit higher than SimplePIR) without having the client download database dependent hint.

There still remains one issue with HintlessPIR to adopt for our use case. HintlessPIR does not amortises server-runtime and client-server communication (except a big constant) if client were to make batched request. This is somewhat in-appropriate for our use-case since the client optimistically generates a big set of tags and wants to request transactions, if there exists, corresponding to each. We notice that most tags will not have a corresponding transaction in database (given databases are split across appropriate time-spans). Thus we make use of bloom filter that the client downloads and request transactions corresponding to tags for which transactions exist in database with very high probability. Below we first briefly describe HintlessPIR and how bloom filters are used. Then we explore feasibility of using this approach for our use-case based on benchmarks given in the paper.

HintlessPIR

HintlessPIR uses two different PIR schemes: (1) SimplePIR and (2) NttlessPIR. SimplePIR processes client's encrypted request on the database. NttlessPIR treats hint of SimplePIR as its database and homomorphically performs hint-specific decryption procedure, which was previously done client-side, on server side using encrypted client secrets.

SimplePIR

SimplePIR uses linear homomorphic encryption (based on learning with errors). It structures database as a square matrix and client query is a vector. Server side processing includes a simple matrix vector multiplication where each operation is in Z_Q

where Q

can be set to either 2^{32}

or 2^{64}

for CPU friendliness. Thus server costs & per-query request latency for simplePIR are low. Client query is a vector of \sqrt{m}

elements in Z_Q

, where m

are elements in the database.

However, server response size isn't small unless client maintains a database dependent state, usually referred to as hint. To

understand why, we will have to look inside simplePIR.

Learning with errors

Define parameters, LWE dimension N

, ciphertext modulus Q

, plaintext modulus P

, secret key distribution \chi_{k}

, error distribution \chi_e

. \chi_e

is always a gaussian distribution with standard deviation \sigma_{LWE}

and \chi_k

in most cases is a ternary distribution {-1,0,1} (but may vary).

LWE ciphertext equals

ct = (c_0, c_1) = (a \cdot s + e + \Delta c, a)

where a \in Z^N_Q, s \in \chi_k, e \in \chi_e, \Delta = \frac{Q}{P}, c \in Z_P

is message.

Let simplePIR database be defined as Db \in Z_{P}^{\sqrt{m} \times \sqrt{m}}

. That is, it contains m

elements in Z_P

. Server first samples A \in Z^{\sqrt{m} \times N}_Q

and makes A

public. Server than processes Db

dependent hint matrix H = Db \cdot A

.

Before making any query client must download H \in Z^{\sqrt{m} \times N}_Q

. After which client can make as any queries as client wishes before the Db

updates. Once Db

updates, H

will update as well and the client will need to refresh the hint.

To query a column i

in Db

, client generates encrypted query qu

as

qu = A\cdot s + E + \Delta(col_i)

where s \in \chi^N_s

is query-specific secret, E \in \chi^{\sqrt{m}}_e

, col_i \in {0,1}^{\sqrt{m}}

is a hot vector with all rows set to 0 except i^{th}

row which is set 1.

Server homomorphically generates query response res

as:

res = Db \cdot qu

Client downloads encrypted query response res \in Z^{\sqrt{m}}_Q

and decrypts the response using hint matrix H

as:

res - H \cdot s

Since,

res - H \cdot s = \Delta Db \cdot col_i + Db \cdot e

Assuming Db \cdot e

is small, client can scale down the output by \Delta

to recover i^{th}

column of Db

.

\frac{P(\Delta Db \cdot col_i + Db \cdot e)}{Q} = Db \cdot col_i

Observe that H

has \sqrt{m} \times N

elements in Z_Q

. If database updates frequently then keeping H

in sync on client side with Db

becomes the bottleneck. To use simplePIR for our use-case we need to ask whether it is possible to eliminate the need of client downloading hint matrix H

?

NttlessPIR

NttlessPIR observes that the client uses H

to perform matrix vector product H \cdot s

. What if the client offloads this computation to the server with encrypted vector s

. Now we must answer whether there's an efficient way to do so. In the paper authors answer in affirmation to this question. They use levelled BFV scheme and employ diagonal dominant form for matrix vector product technique described in [3].

To improve the efficiency of homomorphic matrix vector product such that the overhead of levelled BFV while processing client's query is minimized, they make the following observations:

1. Matrix vector product requires rotations, additions, and scalar multiplications. Out of which rotations constitute dominant cost due to NTTs.
2. BFV ciphertexts consist of two parts ~ a pseudorandom part and a random part. The random part is public and can be forced to stay constant across requests without loss to security as long as client refreshes the secret for each request. Hereafter we will only consider the case where random part of ciphertext is derived from a publicly available seed and random part will be referred to as public random part.
3. NTTs are only required for key switching operation in rotations. Key switching operation only touches the public random part of ciphertext. This allows us to pre-process all key switching operations, thus getting rid of all NTT operations at run-time.

4. After (3), to evaluate homomoprhic matrix vector product, at run-time, server only needs to perform cheap scalar multiplications and ciphertext additions.

Thus, in NttlessPIR, sever treats H

as database and pre-processes all operations corresponding to public randomness part of ciphertexts required to homomorphically evaluate H \cdot s

. Then at the time of request, the server, only performs cheap scalar multiplications and additions.

Batched HintlessPIR requests

Thanks to Mark Schultz for suggesting that RLWE ciphertexts for NttlessPIR can be amortized across several requests if one seeds and pre-processes hints for as many A

's in simplePIR.

It will be ideal if HintlessPIR amortises costs in batched requests. That is, if client wants to request k

columns from the database then the costs should be sublinear to k

. However, due to the way SimplePIR works, (& it is probably a open problem) costs, except a big constant in client-server communication, cannot be amortised.

To understand what big constant I am referring to, one should realise that a single client query consists of two parts (1) query for SimplePIR (2) pseudo-random part of RLWE ciphertext of s

for NttlessPIR. To request k

columns, client needs to send k

SimplePIR queries. But, since A

is fixed across requests, client will have to use k

different secrets one for each query. Hence, client will be forced to also send k

RLWE ciphertexts, one for each secret, to the server for NttlessPIR. But what if the server fixes k

different A

's for SimplePIR. Then, although client still sends k

simplePIR queries, client will be able to use same secret s

for k

queries. Hence, client needs to only upload RLWE ciphertext of single secret to the server for NttlessPIR.

If the server supports k

A

's for simplePIR, client can amortise a big constant in client-server communication in batched request by only having to upload RLWE ciphertext of a single secret to the server for NttlessPIR. However, this comes at the expense of increased server-runtime. To be specific, in addition to processing k

SimplePIR requests, the server will have k

different hint matrices {H_1, ..., H_k}

and will have to pre-process and run NttlessPIR for each. Thus, to process a batch request of size k

server pre-processing and run-time increases by a factor of k

. This indicates that in addition to having server support batch request of size k

to amortise client-server communication, we will require some other way to limit batch size (i.e. k

) to a small constant.

Bloom filters

We observe that if transaction database is split across time spans, then for an average user only few tags in the set of optimistically generated tags will have a corresponding transaction in database. Thus it seems wasteful if user makes a batch request, requesting corresponding transactions to all tags in optimistically generated tag set.

We require server to construct a bloom filter for the database. Before a client makes a query, they download the bloom filter. Then using the bloom filter, client filters the set size of optimistically generated tags to a small number consisting only of tags that have a corresponding transaction in the database with high probability. Client then sends a batch request, requesting transactions corresponding to tags only in filtered set. On average the size of filtered set will be small, thus allowing us to fix k

for the server to a small constant value.

Client

Before we define complexities and evaluate the procedure for our use-case, it will be helpful to understand what happens on the client's device before making a request.

To make a request, client needs to:

1. Download bloom filter specific to database from the server.
2. Reduce the set of optimistically generated tags using the bloom filter.

Let H(x|\rho)

be the hash function used to generate tag with shared secret x

and random value \rho

. Let client's threshold be t

and current public nonce be W

. Also assume that client has g

contacts. Then the client generates optimistic tags set as:

OT = {H(x_0|W+1), H(x_0|W+2)..., H(x_0|W+(t-1)), H(x_1|W+1), ... H(x_{g-1}|W+(t-1))}

To generate OT

client calls H(.)

(t-1) \cdot g

times.

Client reduces size of OT

using bloom filter B

. Let v

be the no. of hash functions used by the bloom filter. The client reduces OT

to set of active tags AT

that have corresponding transaction with high probability.

AT = {x | B[H_i(x)] = 1 \space \forall i \in [0,v)}

where B[j]

is j^{th}

bit of bloom filter.

Let b

be the batch size supported by the server. To request transactions corresponding to each tag in AT

, client makes \lceil \frac{|AT|}{b} \rceil

batched queries to the server.

Defining Complexities

Let database be a matrix Db \in Z^{n_r \times n_c}_{P}

. Assume that a transaction has y

bits. We store a tag transaction tuple (tag, tx)

in column col = tag \mod{n_c}

of Db

. Moreover, in practice, we expect \log{P} < y

and store the tx

at column col

spanned across stack of multiple rows.

Since a single column will fit \lceil\frac{\log{P} \cdot n_r}{y}\rceil

transactions. Hence, Db

can fit:

T \space txs= \lceil\frac{\log{P} \cdot n_r}{y}\rceil \cdot n_c \space

Define HintlessPIR related parameters:

1. SimplePIR: N

as LWE dimension, Q

as LWE ciphertext modulus (usually set to either 2^{32}

or 2^{64}

), P

as LWE plaintext modulus (same as defined above)

1. NttlessPIR: n

as RLWE ring dimension, q

as RLWE ciphertext modulus, l

is decomposition levels for key switching, p

RLWE plaintext modulus, k = \lceil\frac{\log{Q}}{\log{p}}\rceil

Also let v

be number of hash functions used by bloom filter B

with size m

bits.

Assume that Db

is filled with transactions and server supports batched request of size b

. For simplicity we assume that |AT| = b

. That is set size of active tags equal batch size b

.

Following are the complexities:

1. Server pre-processing time:
2. 2bn_r n_cN

operations in Z_Q

: This is needed to pre-process Db

b

times with each of A_1, ..., A_b

to output H_1, ..., H_b

hint matrices.

1. bklnN \log{n}

operations in Z_q

: This is need to pre-process “public randomness” b

times in NttlessPIR, one for each hint matrix H_j

.

1. vT

hash operations: This is needed to insert T

transactions in bloom filter B

.

1. 2bn_r n_cN

operations in Z_Q

: This is needed to pre-process Db

b

times with each of A_1, ..., A_b

to output H_1, ..., H_b

hint matrices.

1. bklnN \log{n}

operations in Z_q

: This is need to pre-process “public randomness” b

times in NttlessPIR, one for each hint matrix H_j

.

1. vT

hash operations: This is needed to insert T

transactions in bloom filter B

.

1. Server run-time:
2. bn_rn_c

operations in Z_Q

: This is needed to process b

SimplePIR queries.

1. bkN(n_r + n + (l+2)n)

operations in Z_q

: This is needed to calculate H_j \cdot s_{LWE}

with encrypted s_{LWE}

.

1. bn_rn_c

operations in Z_Q

: This is needed to process b

SimplePIR queries.

1. bkN(n_r + n + (l+2)n)

operations in Z_q

: This is needed to calculate H_j \cdot s_{LWE}

with encrypted s_{LWE}

.

1. Client upload for batched query of size b

: 1. bn_c

elements in Z_Q

: b

SimplePIR query vectors.

1. (k+l)n

elements in Z_q

: Pseudo random parts of RLWE ciphertexts containing s_{LWE}

and rotation keys. This is required for NttlessPIR.

1. bn_c

elements in Z_Q

: b

SimplePIR query vectors.

1. (k+l)n

elements in Z_q

: Pseudo random parts of RLWE ciphertexts containing s_{LWE}

and rotation keys. This is required for NttlessPIR.

1. Client download for batched query of size b

: 1. bn_r

elements in Z_Q

: b

SimplePIR responses.

1. 2bk(n_r + n)

elements in Z_q

: b

NttlessPIR responses. Required to recover H_j \cdot s_{LWE}

by the client.

1. bn_r

elements in Z_Q

: b

SimplePIR responses.

1. 2bk(n_r + n)

elements in Z_q

: b

NttlessPIR responses. Required to recover H_j \cdot s_{LWE}

by the client.

1. Client pre-query download:
2. m

bits: Size of bloom filter.

1. m

bits: Size of bloom filter.

1. Client pre-query processing:
2. |OT|

: hash operations: To generate set of optimistic tags.

1. v|OT|

hash operations: To filter OT

down to set of active tags AT

.

1. |OT|

: hash operations: To generate set of optimistic tags.

1. v|OT|

hash operations: To filter OT

down to set of active tags AT

.

Adding a new transaction

The transaction Db

updates frequently, thus it is important to estimate the cost of inserting new elements. Let’s assume that a new transaction is to be inserted at column i

. Let r = \lceil\frac{y}{\log{P}}\rceil

denote the number of rows a single transaction occupies, thus causes to change. Pre-processing requires:

1. Processing b

hint matrices for SimplePIR (H_j = Db \cdot A_j

for each j \in [0,b)

). It has complexity bNr

.

To understand why, assume that only elements till q^{th}

column (starting from left) in i^{th}

row of Db

are set to 1 and rest are set to 0. Then each element in i^{th}

row of H_j

is inner product of q

elements in Db

and q

elements in corresponding column of A_j

for each column of A_j

. Let us update the element at (i,q+1)

in Db

to 1, which was previosuly set to 0. Recalculating an element in i^{th}

row of H_j

will require a single product (Db[i,q+1] \times A_j[q+1, z]

for the corresponding column z

in A_j

) and an addition. Then updating each element in i^{th}

row will require N

products and additions. Since inserting a single tx affect r

rows, we have complexity Nr

. Thus bNr

for b

hints matrices.

1. in NttlessPIR to,
2. pre-process key rotations for “random” part of RLWE ciphertext of s_{LWE}
3. pre-process each hint matrix H_j

represented in diagonal dominant form as N

SIMD encoded RLWE plaintexts.

1. pre-process public randomness of scalar multiplications and ciphertext additions to calculate H_j \cdot s_{LWE}

homomorphically.

1. pre-process key rotations for “random” part of RLWE ciphertext of s_{LWE}
2. pre-process each hint matrix H_j

represented in diagonal dominant form as N

SIMD encoded RLWE plaintexts.

1. pre-process public randomness of scalar multiplications and ciphertext additions to calculate H_j \cdot s_{LWE}

homomorphically.

Observe that (1) is independent of hint matrix, thus only needs to be processed once. Since updating r

rows of Db

will affect r

rows of H_j

, one will have to reconstruct all N

SIMD encoded RLWE plaintexts. Doing so has complexity of O(bNn\log{n})

. In addition to pre-processing N

SIMD plaintext for each hint matrix H_j

, server will also have to pre-process public randomness corresponding to each H_j

. This will require additional bnN

operations in Z_q

.

Scope of parallelization

In addition to the obvious prallelization opportunity of running pre-processing for b

requests in parallel and processing each request in batched request of size b

in parallel, there are several parallelization opportunities within a single request.

1. Server pre-processing: To pre-process the server -
2. Calculates hint matrices H_1, ..., H_j

each of which require simple matrix multiplications and can be easily parallelized.

1. Pre-computes “public randomness” for key rotations of s_{LWE}

RLWE ciphertexts. This can be easily paralleized by assigning precompute operation for each rotation to a different thread. Note that this needs to run only once.

1. Pre-computes “public randomness” for evaluating multiple matrix-vector multiplications. That is H_j \cdot s

for each hint matrix. For a single H_j

, constructing N

SIMD encoded plaintexts can be assigned to different threads and processing public randomness part of matrix vector product can also be split across threads as well.

1. Calculates hint matrices H_1, ..., H_j

each of which require simple matrix multiplications and can be easily parallelized.

1. Pre-computes “public randomness” for key rotations of s_{LWE}

RLWE ciphertexts. This can be easily paralleized by assigning precompute operation for each rotation to a different thread. Note that this needs to run only once.

1. Pre-computes “public randomness” for evaluating multiple matrix-vector multiplications. That is H_j \cdot s

for each hint matrix. For a single H_j

, constructing N

SIMD encoded plaintexts can be assigned to different threads and processing public randomness part of matrix vector product can also be split across threads as well.

1. Server run-time: At run-time server -
2. Performs b

matrix-vector multiplications, each of which can be easily parallelized.

1. Evaluates b

homomorphic matrix vector products (i.e. H_j \cdot s_{LWE}

). Each matrix vector product requires only scalar multiplications and additions, both of which can be easily split across threads.

1. Performs b

matrix-vector multiplications, each of which can be easily parallelized.

1. Evaluates b

homomorphic matrix vector products (i.e. H_j \cdot s_{LWE}

). Each matrix vector product requires only scalar multiplications and additions, both of which can be easily split across threads.

Concrete values

To check how feasible it is to use the approach in production, let's estimate different costs based on benchmarks given in the paper ([2] section 7). We will consider three different types of databases: small, medium, and big. Small database stores only the transactions published in the past hour. It is suitable for users that are active during the hour. For example, they are actively interacting with a game and want low latency. Medium database stores all transactions of any given day. It is suitable for people that only check their wallets every few hours or within few days. Big database stores all transactions of any given week (i.e. 7 days). It is suitable for users that check their wallet every other week or so.

Throughout we will assume that Aztec will have a TPS of 10 and average transaction size is 128 bytes. Databases are either structured as a square matrix or a rectangle matrix. We will follow the notations defined in "Defining complexities". We will also insert transactions in the database as we did in "Defining complexities" section. That is, given a tag transaction tuple (tag, tx)

we insert tx

stacked across multiple rows in column col = tag \mod{n_c}

where n_c

is no. of columns in Db

. Thus, given that tags are random, it is reasonable to assume that transactions are uniformly distributed across columns in Db

.

Duration

Total transactions

Lower bound of Db size (in Mb)

Size of Db used in HintlessPIR estimations (in Mb)

Small

1 hour

36000

4.4

8

Medium

1 day

864000

105.5

268

Big

1 week

6048000

739

1095

Lower bound of Db

size is calculated as |T| \times 128 \text{bytes}

, where |T|

is upper bound on no. of transaction published in relevant duration (calculated as 10 \times \text{Duration}

). “Lower bound of Db

” is the expected size of transactions database and “Size of Db used in HintelessPIR” estimations is size of database taken from benchmarks provided in the paper. Notice that database used in HintlessPIR estimations is sufficiently bigger than required database to store transactions.

m (no. of bits in Bloom filter)

|T| (maximum capacity of bloom filter)

Small

36000 \times 10

36000

Medium

864000 \times 10

864000

Big

6048000 \times 10

6048000

The table lists bloom filter parameters used for each database. We fix no. of hash functions to v

and limit false positive rate f_p

to 0.0086 [4].

SMALL

MEDIUM

BIG

HintlessPIR (in ms)

Bloom filter (in hash operations)

HintlessPIR (in ms)

Bloom filter (in hash operations)

HintlessPIR (in ms)

Bloom filter (in hash operations)

Server pre-processing

b \times 3110

(99520

)

36000 \times 8

b \times 51570

(1650240

)

8640000 \times 8

b \times 199150

(6372800

)

60480000 \times 8

Server run-time

a \times 271

(8672

)

0

a \times 575

(18400

)

0

a \times 1033

(33056

)

0

Client pre-query

0

o \times 8

0

o \times 8

0

o \times 8

Client post-query

a \times 4.78

(153

)

0

a \times 25.5

(816

)

0

a \times 52.32

(1675

)

0

HintlessPIR (in kb)

Bloom filter (in kb)

HintlessPIR (in kb)

Bloom filter (in kb)

HintlessPIR (in kb)

Bloom filter (in kb)

Client pre-query download

0

44

0

1055

0

7383

Client query upload

13 \times a + \rho \times 387

(803

)

0

66 \times a + \rho \times 387

(2499

)

0

131 \times a + \rho \times 387

(4579

)

0

Client query download

a \times 151

(4832

)

0

a \times 807

(25824

)

0

a \times 1613

(51616

)

0

Recall that OT

represents set of all optimistically generated tags and AT

represents only the active tags. Client requests transactions from Db

corresponding to tags only in AT

. In a batch request client can request transactions of at-most b

tags. If |AT| \gt b

then client will have to make \rho = \lceil\frac{|AT|}{b}\rceil

batched queries. We denote |AT|

as a

and |OT|

as o

for clarity. All benchmarks are single threaded (Intel i7-1185G CPU running at 3.00GHz and with 32GB RAM, [2] Section 7). Texts in bracket are values are for the case when |AT| = b = 32

(i.e. client requests 32 transactions).

Note: It is safe to ignore the overhead of hash operations both on client side and server side and assume that they wouldn't take more than a few seconds. This is because on modern processors hash functions are quite fast. For example, following are benchmarks for sha256 on M1 with openssl.

```
openssl speed sha256
```

Doing sha256 for 3s on 16 size blocks: 17640021 sha256's in 3.00s Doing sha256 for 3s on 64 size blocks: 14042837 sha256's in 3.00s Doing sha256 for 3s on 256 size blocks: 11193794 sha256's in 3.00s Doing sha256 for 3s on 1024 size blocks: 5016334 sha256's in 3.00s Doing sha256 for 3s on 8192 size blocks: 825002 sha256's in 3.00s

Better than trial decryption?

For client side processing, the answer is a clear yes. With the proposed approach client only needs to run recovery procedure for as many tags as there are in active tag set AT

. Whereas in trial decryption client needs to decrypt all transactions since the last it synced with network.

For bandwidth we may ask, beyond what count of requests would it be better for the client to simply download the database.

[

1

1920×1044 77.4 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/1X/76c8e249bd84f5e6582a0701bdff2762d3f693ff.png)

In the graph above, y-axis represents the threshold which shows the no. of requests beyond which it will be better for client, in terms of bandwidth cost, to download the entire database. That is, when database size is smaller than the total bandwidth to make the batched PIR request. The x-axis represents batch size b

and shows how does the threshold vary across batch sizes. As clear from the graph, the threshold does not vary much across batch sizes. This implies that it may be a better idea to set batch size b

to a small constant, for example 32, depending on the size of an average request. Note that we ignore the case when batch size equals 1 because it skews the graph.

Size of the batch

The graph above indicates that setting the batch size to a higher value may not greatly reduce client's total communication cost as compared to the case when batch size is set to a small constant. So it is natural to ask how does batch size affect client's total communication cost given that client wants to request x

tags.

[

2

1920×931 62.6 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/1X/f69f74cef7ccee4404f071037d0939082a279945.png)

The graph shows how does client's total communication cost (y-axis) vary across batch sizes (b

) (x-axis) for request sizes 10,100,200,300,400

. One can observe that the impact of doubling batch size has degrading impact on reducing client communication cost. This confirms that batch size should be set to a small constant because a high value will increase server's cost unnecessarily without any benefit to the client. Note that we ignore the graph for Medium and Big database because they follow the same pattern.

[

3

1920×931 53.7 KB

](https://europe1.discourse-cdn.com/business20/uploads/aztec/original/1X/a1ef918fb2a8091b36e89a2efdb306d7bdfbec45.png)

This graph displays same things as the one before but ignores smaller batch sizes and adds bigger batch sizes for clarity. One can observe that there's not much difference in client communication when the batch size is 32 vs when batch size is 512.

References

1. One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval - One Server for the Price of Two: Simple and Fast Single-Server Private Information Retrieval
2. Hintless Single-Server Private Information Retrieval - https://eprint.iacr.org/2023/1733.pdf
3. Algorithms in HElib - https://eprint.iacr.org/2014/106.pdf
4. Bloom Filters - Bloom Filters - the math
5. Privacy-Preserving User-Auditable Pseudonym Systems - IEEE Xplore Full-Text PDF: