Don’t be Dense: Efficient Keyword PIR for Sparse Databases

Sarvar Patel, Joon Young Seo, Kevin Yeo

Michael Rodel, Nir Segal and Hallel Weinberg

This project is submitted as part of completing your duties in the Secure Computation class instructed by Dr. Adi Akavia.

PIR

- Private Information Retrieval (PIR) is a cryptographic protocol that allows a user to retrieve information from a database without revealing which specific information they are interested in.
- The database in the server is represented as an array.

PIR

The protocol divided to four algorithms $(\boldsymbol{Init}, \boldsymbol{Query}, \boldsymbol{Answer}, \boldsymbol{Decrypt})$:

- Init algorithm initializes the keys for communication between the server and the client.
- Query algorithm make a query in the client and sends it encrypted to the server.
- Answer algorithm receives the database and the encrypted query and send a response to the client.
- Decrypt algorithm receives the response and decrypt it.

Techniques overview

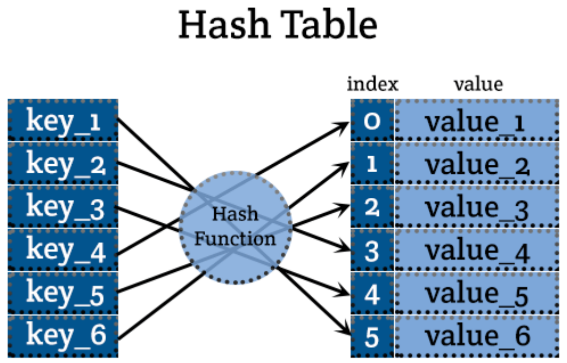
The paper proposes a new keyword private information retrieval (PIR) scheme called SparsePIR.

Key-Value Pairs Representation

- A more efficient representation of a database often involves using a key-value representation with hashing. This approach is preferred because the traditional array representation can be inefficient when dealing with sparse data, where most of the array elements are empty.

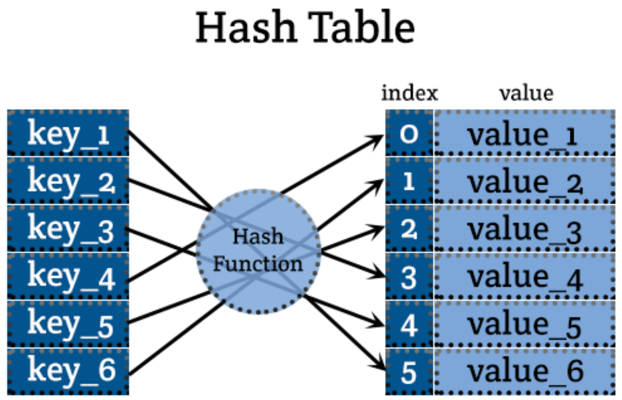
Key-Value Pairs Representation

- By employing a hash table, the data can be organized in a way that reduces sparsity compared to the array representation. Hashing allows for faster access to specific values based on their keys, which can significantly improve data retrieval performance.



Key-Value Pairs Representation

- For some hash key K, we denote $rep(K, k_i, v_i)$ as the hash evaluation of k_i concatenated with v_i (k_i, v_i are the client's key and value).
- $rep(K, k_i, v_i) = F(K, k_i)||v_i$, where $F(K, k_i)$ is the hashed index.



Fully Homomorphic Encryption

- All the computation are under fully homomorphic encryption scheme.
- Given plaintext pt, the encrypted pt is $Enc(pt) = (r, r \cdot s + e + pt)$, where r is a uniformly random element, s is the secret key in FHE scheme and e is a polynomial noise where each coefficient is typically small.
- To decrypt ciphertext (ct_1, ct_2), one will compute $ct_2 - (ct_1 \cdot s) = e + pt$.

Fully Homomorphic Encryption

In the paper, FHE encryption was implemented in the communication and the Answer algorithm on the server side, ensuring that the server will not know the client's query.

Recursion

SparsePIR is compatible with the PIR optimization known as **recursion**.

Recursion allows a client to obtain multiple entries from the database in a single query. This can be done by recursively splitting the query into smaller queries, and then using the results of the smaller queries to reconstruct the results of the original query.

Benefits: Efficiency and Scalability.

Recursion

- Represent the database, n-entry array, as a hypercube of dimensions $d_1 \times d_2 \times \cdots \times d_z$.
- The product of the dimensions is at least n ($d_1 \cdot \ldots \cdot d_z \geq n$).
- To query an entry, the client will generate z indictor vectors $v_1 \in \{0,1\}^{d_1}, \ldots, v_z \in \{0,1\}^{d_z}$, where each v_i has zeros everywhere except for a one indicating the location of the entry with respect to the dimension d_i.
- Denote: $m = d_1$.

Encoding

The process of encoding the database involves generating random binary vectors $v_k \in \{0,1\}^m$ for each entry in the database. These vectors are then encoded using $rep(K_r, k_i, v_i)$.

The resulting encoded database, represented as matrix E, will be as follows:

$$\begin{pmatrix} v_{k_1}^T \\ v_{k_2}^T \\ \vdots \\ v_{k_n}^T \end{pmatrix} \cdot E = M \cdot E = \mathrm{y} = \begin{pmatrix} rep(K_r, k_1, v_1) \\ rep(K_r, k_2, v_2) \\ \vdots \\ rep(K_r, k_n, v_n) \end{pmatrix}$$

In this representation:

- $v_{k_\mathrm{i}}^T$ denotes the transpose of the randomly generated binary vector v_{k_i} for the i-th entry.
- y represents the database.

Encoding

The process of encoding the database involves generating random binary vectors $v_k \in \{0,1\}^m$ for each entry in the database. These vectors are then encoded using $rep(K_r, k_i, v_i)$.

The resulting encoded database, represented as matrix E, will be as follows:

$$\begin{pmatrix} v_{k_1}^T \\ v_{k_2}^T \\ \vdots \\ v_{k_n}^T \end{pmatrix} \cdot E = M \cdot E = \mathrm{y} = \begin{pmatrix} rep(K_r, k_1, v_1) \\ rep(K_r, k_2, v_2) \\ \vdots \\ rep(K_r, k_n, v_n) \end{pmatrix}$$

By following this encoding process, the original entries of the database are transformed into an encoded form.

Partition

The paper proposes a novel **partitioning** technique for SparsePIR.

The partitioning technique is based on the idea of dividing the database into a number of partitions, such that each partition contains a small number of database entries. This allows the client to generate a smaller random vector, which can significantly reduce the communication cost of SparsePIR.

Partition

- Create $b = (1+\epsilon)n/d_1$ partitions, where each part will be size d_1.
- Given input of Database $D = \{(k_1, v_1), \dots, (k_n, v_n)\}$ that needs to be encoded, assign the n key-value pairs to the b partitions uniformly at random, ensuring that in each part $\frac{n}{b} = \frac{d_1}{1+\epsilon} < d_1$ pairs will be allocated.

As a result, the problem is effectively reduced to efficiently encoding databases of size $O(d_1)$.

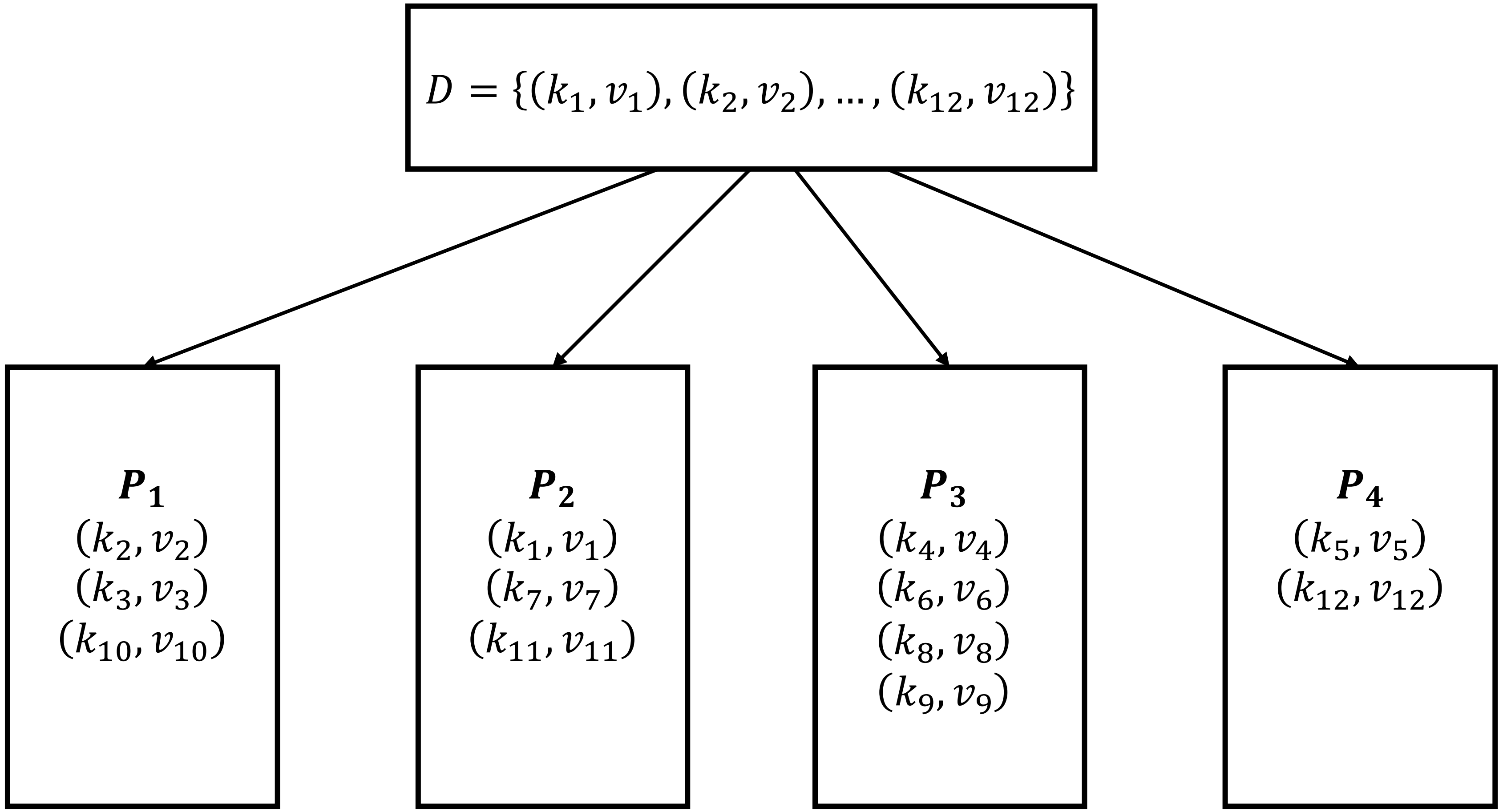
Partition

Consider the i-th part $P_i = \left\{(k_1, v_1),\ \ldots, \left(k_{|P_i|}, v_{|P_i|}\right)\right\}$. Generate M_i that is a $|P_i| \times d_1$ matrix where each entry is uniformly chosen from $\{0,1\}$ (in particular, the i-th row is generated randomly using K_2 and key k_i).

Compute $y_i^t = \left[rep(K_r, k_i, v_i), \ldots, rep\left(K_r, k_{|P_i|}, v_{|P_i|}\right)\right]$.

Finally, Solve the linear system $M_i \cdot e_i = y_i$ and obtain the encoding e_i for part P_i.

Partition


$D = \{(k_1, v_1), (k_2, v_2), \dots, (k_{12}, v_{12})\}$
$\boldsymbol{P_1}$ (k_2, v_2) (k_3, v_3) (k_{10}, v_{10})
$\boldsymbol{P_2}$ (k_1, v_1) (k_7, v_7) (k_{11}, v_{11})
$\boldsymbol{P_3}$ (k_4, v_4) (k_6, v_6) (k_8, v_8) (k_9, v_9)
$\boldsymbol{P_4}$ (k_5, v_5) (k_{12}, v_{12})

Partition

$\boldsymbol{P_3}$
(k_4, v_4)
(k_6, v_6)
(k_8, v_8)
(k_9, v_9)

$$\xrightarrow{\text{Encode } P_3}$$

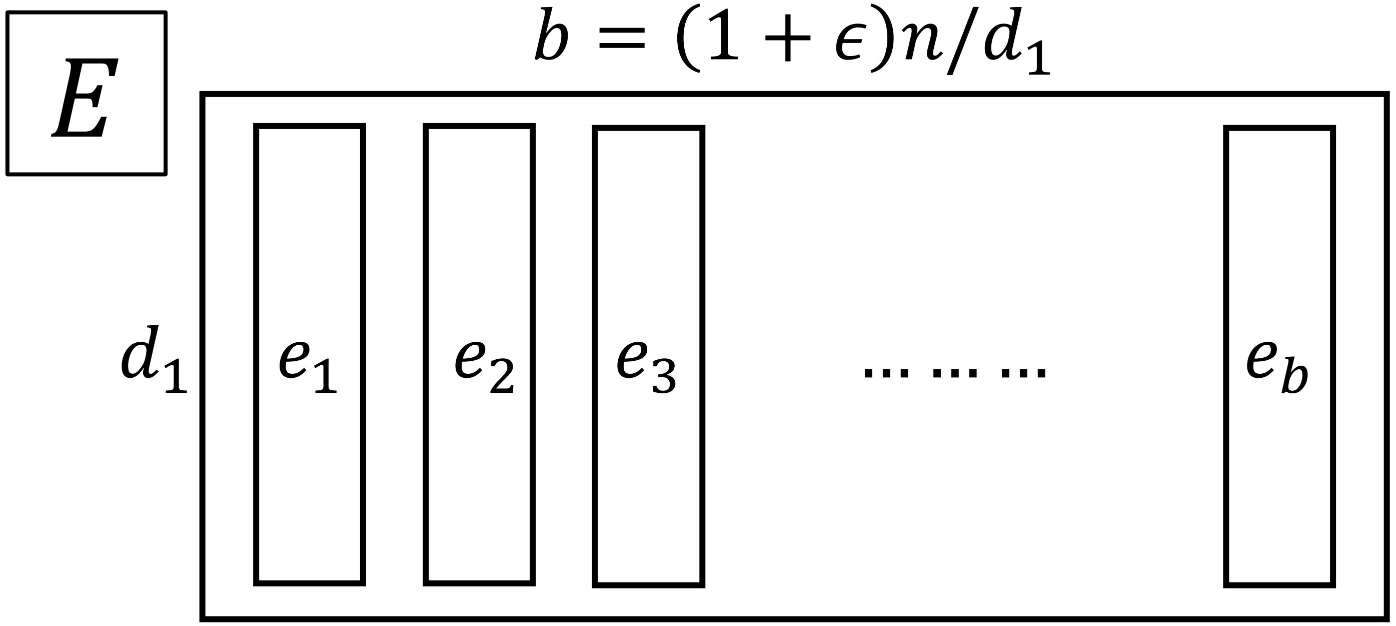
$$e_3 = \begin{bmatrix} F(K_2, k_4)||1 & \cdots & F(K_2, k_4)||5 \\ F(K_2, k_6)||1 & \cdots & F(K_2, k_6)||5 \\ F(K_2, k_8)||1 & \cdots & F(K_2, k_8)||5 \\ F(K_2, k_9)||1 & \cdots & F(K_2, 9)||5 \end{bmatrix}^{-1} \begin{bmatrix} F(K_r, k_4)||v_4 \\ F(K_r, k_6)||v_6 \\ F(K_r, k_8)||v_8 \\ F(K_r, k_9)||v_9 \end{bmatrix}$$

Final encoding:

$$E = [e_1 \quad e_2 \quad e_3 \quad e_4]$$

Partition

To build the final encoded database E, put each of $e_1, \dots, e_b$ as column vectors in E to construct a $d_1 \times b$ matrix.



Partition

To query for a key k, the client will compute $i = F_1(K_1, k)$ to determine the partition associated with k.

The client randomly generates the associated random vector using hash key K_2 and k, denoted as v_{k_1}. The server applied v_{k_1} to the first dimension of E:

$$[v_{k_1} \cdot e_1 \quad \dots \quad v_{k_1} \cdot e_b]$$

The only entry needed to be retrieved is $v_{k_1} \cdot e_i$ (given k is assigned to the i-th partition).

Partition

Benefits: Reduced communication cost, improved scalability and simplified implementation.

PIR

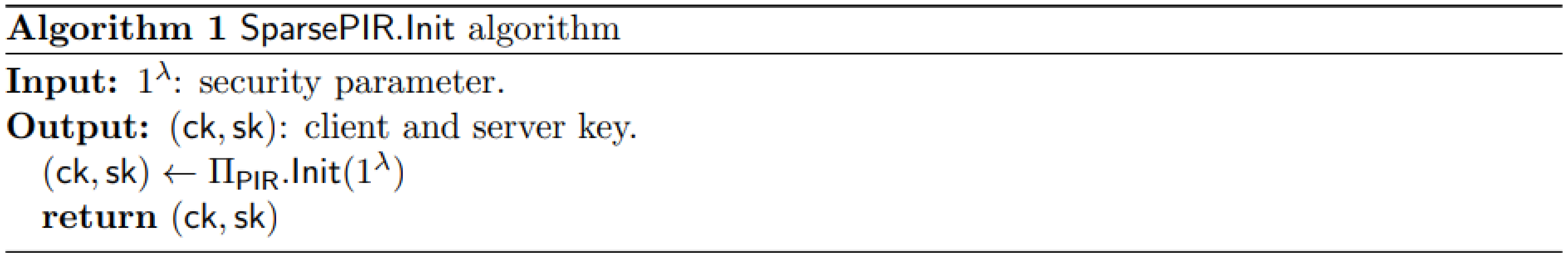
The underlying PIR scheme will represent any m-element database as a hypercube with dimensions $d_1 \times \cdots \times d_z$. PIR is constructed from four algorithm:

- $(\boldsymbol{ck}, \boldsymbol{sk}) \leftarrow \boldsymbol{Init}(\mathbf{1}^{\boldsymbol{\lambda}})$**:** The Init algorithm produces client key ck and server key sk.
- $(\boldsymbol{st}, \boldsymbol{req}) \leftarrow \boldsymbol{Query}(\boldsymbol{ck}, \boldsymbol{k})$**:** The Query algorithm receives z vectors of length $d_1, \ldots, d_z$ that it will homomorphically encrypt and upload to the server. For standard PIR, these z vectors are indicator vectors representing the query index in each dimension.

PIR

- $\boldsymbol{resp} \leftarrow \boldsymbol{Answer}(\boldsymbol{sk}, \boldsymbol{D}, \boldsymbol{req})$: The Answer algorithm receives the database, in a form of two-dimensional matrix of size $d_1 \times \left\lceil \frac{m}{d_1} \right\rceil$, and homomorphic encryptions of vectors $v_1, \dots, v_z$. The Answer algorithm will perform the standard PIR algorithm of applying v_1 to E to obtain a $\left\lceil \frac{m}{d_1} \right\rceil$ vector, arrange the vector into a $d_2 \times \left\lceil \frac{m}{d_1 \cdot d_2} \right\rceil$ matrix and apply v_2 to obtain a vector of size $\left\lceil \frac{m}{d_1 \cdot d_2} \right\rceil$. Repeat this process for all z dimensions.
- $\boldsymbol{v} \leftarrow \boldsymbol{Decrypt}(\boldsymbol{ck}, \boldsymbol{st}, \boldsymbol{resp})$: The Decrypt algorithm receives the server’s response and produces a decrypted answer.

SparsePIR



Algorithm 1 SparsePIR.Init algorithm

```
Input: $1^\lambda$: security parameter.
Output: $(\mathsf{ck}, \mathsf{sk})$: client and server key.
  $(\mathsf{ck}, \mathsf{sk}) \leftarrow \Pi_{\mathsf{PIR}}.\mathsf{Init}(1^\lambda)$
  return $(\mathsf{ck}, \mathsf{sk})$
```

SparsePIR

Algorithm 2 SparsePIR.Encode algorithm

```
Input: $D = \{(k_1, v_1), \ldots, (k_n, v_n)\}$: database
Output: $(\mathsf{prms}, \mathbf{E})$: parameters and encoding
  Sample hash keys $\mathsf{K}_1, \mathsf{K}_2$ and $\mathsf{K}_r$
  $b \leftarrow (1 + \epsilon)n/d_1$
  $m \leftarrow (1 + \epsilon)n$
  $P_1 \leftarrow \emptyset, \ldots, P_b \leftarrow \emptyset$
  for $i = 1, \ldots, n$ do                                   ▷ Partition database
      $j \leftarrow F_1(\mathsf{K}_1, k_i)$
      $P_{j+1} \leftarrow P_{j+1} \cup \{(k_i, v_i)\}$
  for $j = 1, \ldots, b$ do
      $\mathbf{e}_i \leftarrow \mathsf{GenerateEncode}(\mathsf{K}_2, \mathsf{K}_r, P_j)$
  $\mathsf{prms} \leftarrow (\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$
  $\mathbf{E} \leftarrow [\mathbf{e}_1, \ldots, \mathbf{e}_b]$
  return $(\mathsf{prms}, \mathbf{E})$
```

SparsePIR

Algorithm 4 GenerateEncode algorithm

```
Input: $(\mathsf{K}_2, \mathsf{K}_r, P)$: hash keys and a part.
Output: $\mathbf{e}$: an encoding of the part.
  $\mathbf{M} \leftarrow []$ as empty array
  $\mathbf{y} \leftarrow []$ as empty array
  for $(k, v) \in P$ do
      Append $\mathsf{RandVector}(\mathsf{K}_2, k)^\intercal$ to $\mathbf{M}$ as row.
      Append $\mathsf{rep}(\mathsf{K}_r, k, v)$ to $\mathbf{y}$.
  $\mathbf{e} \leftarrow \mathsf{SolveLinearSystem}(\mathbf{M}, \mathbf{y})$
  return $\mathbf{e}$
```

Algorithm 3 RandVector algorithm

```
Input: $(\mathsf{K}_2, k)$: the hash key and database key.
Output: $\mathbf{v}_k$: a randomly generated vector.
  $\mathbf{v}_k \leftarrow [0]^{d_1}$
  for $i = 1, \ldots, d_1$ do
      $\mathbf{v}_k[i] \leftarrow F_2(\mathsf{K}_2, k \mid\mid i)$
  return $\mathbf{v}_k$
```

SparsePIR

Algorithm 5 SparsePIR.Query algorithm

Input: $(\mathsf{prms}, \mathsf{ck}, k)$: parameters and the query key.

Output: $(\mathsf{st}, \mathsf{req})$: temporary state and request.

```
  $b \leftarrow (1+\epsilon)n/d_1$
  Parse $\mathsf{prms} = (\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$
  $\mathbf{v}_1 \leftarrow \mathsf{RandVector}(\mathsf{K}_2, k)$
  $j \leftarrow F_1(\mathsf{K}_1, k)$
  for $i = 2, \ldots, z$ do                                  ▷ Generate encoding of $j$
      $j_i \leftarrow \lfloor j / \lceil b/d_i \rceil \rfloor$
      $\mathbf{v}_i \leftarrow [0]^{d_i}$
      $\mathbf{v}_i[j_i + 1] \leftarrow 1$
      $b \leftarrow \lceil b/d_i \rceil$
      $j \leftarrow j \mod b$
  $(\mathsf{st}_{\mathsf{PIR}}, \mathsf{req}) \leftarrow \Pi_{\mathsf{PIR}}.\mathsf{Query}(\mathsf{ck}, \mathbf{v}_1, \ldots, \mathbf{v}_z)$
  return $((\mathsf{st}_{\mathsf{PIR}}, k), \mathsf{req})$
```

SparsePIR

Algorithm 6 SparsePIR.Answer algorithm

Input: $(\mathsf{prms}, \mathsf{sk}, \mathbf{E}, \mathsf{req})$: parameters, server key, encoded databases and the request.
Output: resp: the response to the request.

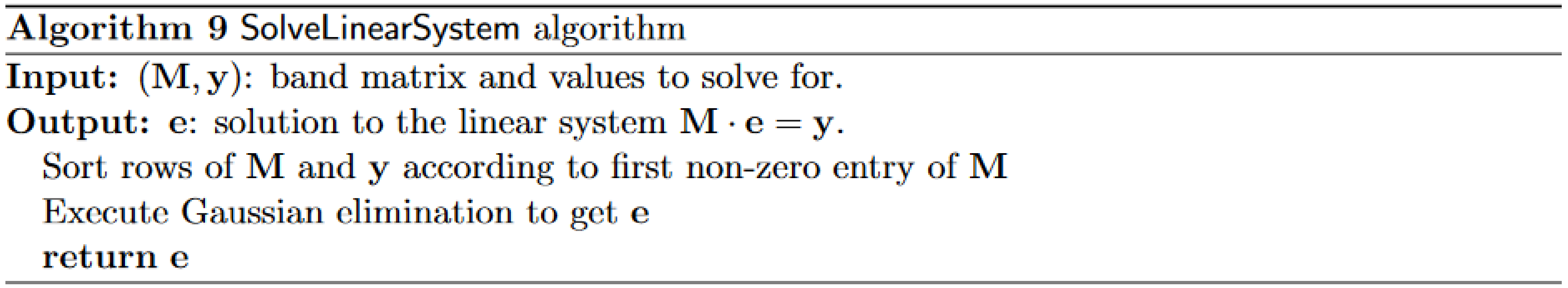
```
  $\mathsf{resp} \leftarrow \Pi_{\mathsf{PIR}}.\mathsf{Answer}(\mathsf{sk}, \mathbf{E}, \mathsf{req})$
  return $\mathsf{resp}$
```

Algorithm 7 SparsePIR.Decrypt algorithm

Input: $(\mathsf{prms}, \mathsf{ck}, \mathsf{st}, \mathsf{resp})$: parameters, client key, temporary state and response.
Output: v: output value

```
  Parse $\mathsf{prms}$ as $(\mathsf{K}_1, \mathsf{K}_2, \mathsf{K}_r)$
  Parse $\mathsf{st}$ as $(\mathsf{st}_{\mathsf{PIR}}, k)$
  $x \leftarrow \Pi_{\mathsf{PIR}}.\mathsf{Decrypt}(\mathsf{ck}, \mathsf{st}_{\mathsf{PIR}}, \mathsf{resp})$
  Parse $x$ as $(\mathsf{id}, v)$
  if $\mathsf{id} = F(\mathsf{K}_r, k)$ then
      return $v$
  else
      return $\perp$
```

SparsePIR



Algorithm 9 SolveLinearSystem algorithm

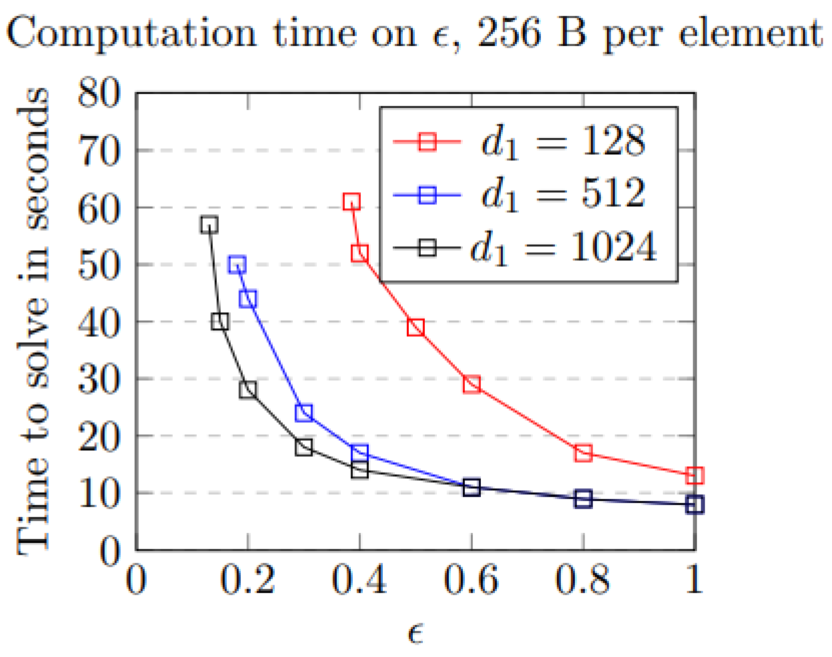
```
Input: $(\mathbf{M}, \mathbf{y})$: band matrix and values to solve for.
Output: $\mathbf{e}$: solution to the linear system $\mathbf{M} \cdot \mathbf{e} = \mathbf{y}$.
  Sort rows of $\mathbf{M}$ and $\mathbf{y}$ according to first non-zero entry of $\mathbf{M}$
  Execute Gaussian elimination to get $\mathbf{e}$
  return $\mathbf{e}$
```

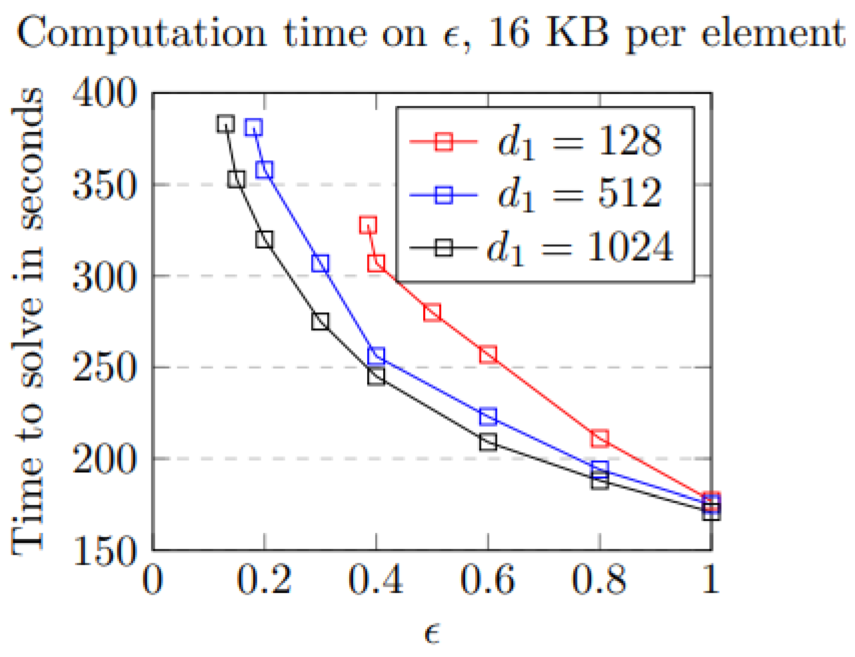
Results summary and significance

	Client Storage	Encoding Size	Response Overhead
Client Mapping	n	n	$1\times$
Cuckoo Hashing	$O(1)$	$(2+\epsilon)n$	$2\times$
Constant- Weight	$O(1)$	n	$2-9\times$
SparsePIR	$O(1)$	$(1+\epsilon)n$	$1\times$
SparsePIR^g	$O(1)$	$(1+\epsilon)n$	$1\times$
SparsePIR^c	$< 11.2KB$	$(1.03)n$	$1\times$

One-round keyword PIR comparison. Response overhead is compared to state-of-the-art standard PIR, that query over dense n-entry arrays.

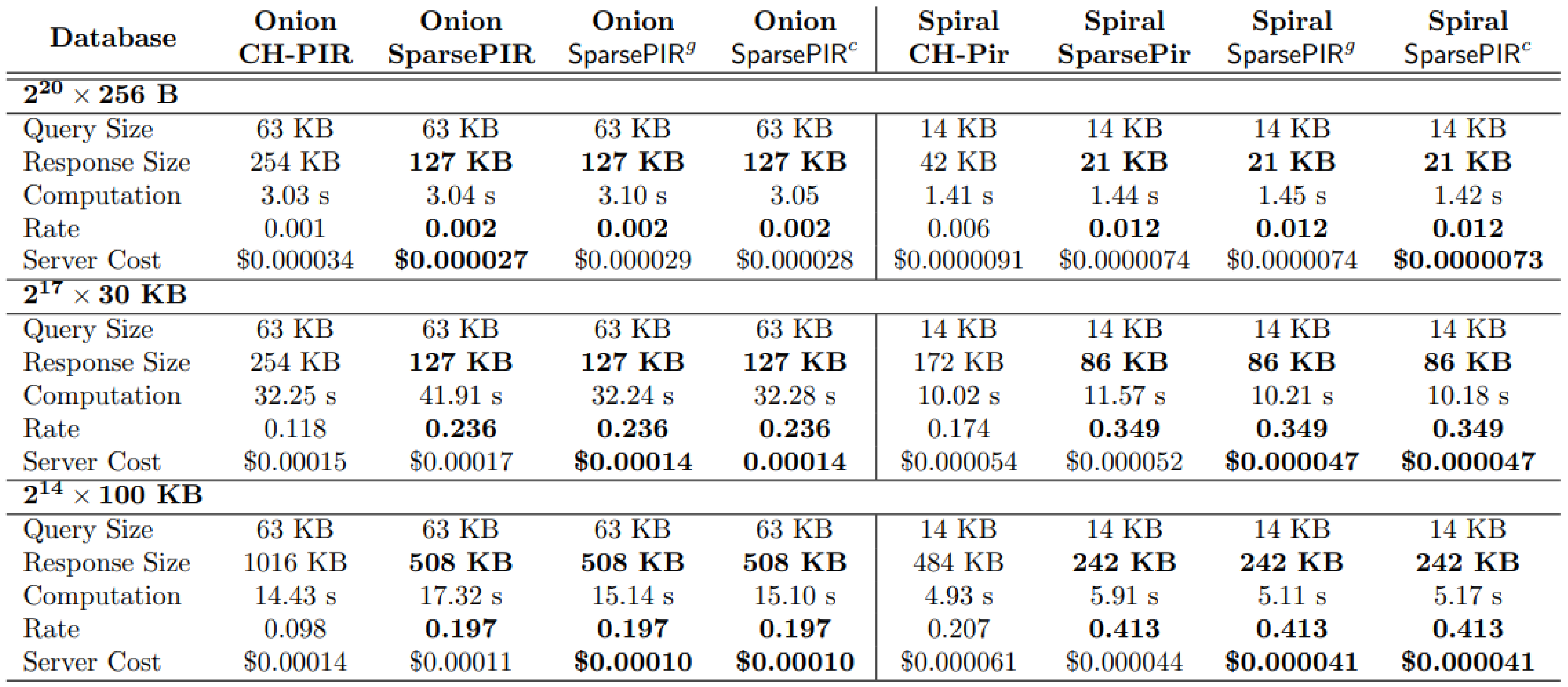
Results summary and significance





Computation time to solve the SparsePIR linear system vs ϵ on 2^{20} elements for dimension sizes $d1 = 128,\ 512, 1024$. The graphs suggest that the ϵ lower bounds for $d1 = 128, 512, 1024$ are roughly $0.38, 0.18, 0.13$ respectively. They chose band parameter $w\ \in\ [50, 60]$.

Results summary and significance



Database	Onion CH-PIR	Onion SparsePIR	Onion SparsePIR^g	Onion SparsePIR^c	Spiral CH-Pir	Spiral SparsePir	Spiral SparsePIR^g	Spiral SparsePIR^c
$2^{20} \times 256$ B								
Query Size	63 KB	63 KB	63 KB	63 KB	14 KB	14 KB	14 KB	14 KB
Response Size	254 KB	**127 KB**	**127 KB**	**127 KB**	42 KB	**21 KB**	**21 KB**	**21 KB**
Computation	3.03 s	3.04 s	3.10 s	3.05	1.41 s	1.44 s	1.45 s	1.42 s
Rate	0.001	**0.002**	**0.002**	**0.002**	0.006	**0.012**	**0.012**	**0.012**
Server Cost	\$0.000034	**\$0.000027**	\$0.000029	\$0.000028	\$0.0000091	\$0.0000074	\$0.0000074	**\$0.0000073**
$2^{17} \times 30$ KB								
Query Size	63 KB	63 KB	63 KB	63 KB	14 KB	14 KB	14 KB	14 KB
Response Size	254 KB	**127 KB**	**127 KB**	**127 KB**	172 KB	**86 KB**	**86 KB**	**86 KB**
Computation	32.25 s	41.91 s	32.24 s	32.28 s	10.02 s	11.57 s	10.21 s	10.18 s
Rate	0.118	**0.236**	**0.236**	**0.236**	0.174	**0.349**	**0.349**	**0.349**
Server Cost	\$0.00015	\$0.00017	**\$0.00014**	**0.00014**	\$0.000054	\$0.000052	**\$0.000047**	**\$0.000047**
$2^{14} \times 100$ KB								
Query Size	63 KB	63 KB	63 KB	63 KB	14 KB	14 KB	14 KB	14 KB
Response Size	1016 KB	**508 KB**	**508 KB**	**508 KB**	484 KB	**242 KB**	**242 KB**	**242 KB**
Computation	14.43 s	17.32 s	15.14 s	15.10 s	4.93 s	5.91 s	5.11 s	5.17 s
Rate	0.098	**0.197**	**0.197**	**0.197**	0.207	**0.413**	**0.413**	**0.413**
Server Cost	\$0.00014	\$0.00011	**\$0.00010**	**\$0.00010**	\$0.000061	\$0.000044	**\$0.000041**	**\$0.000041**

Comparison of cuckoo hashing (CH) and SparsePIR frameworks for various PIR protocols.

SparsePIR halves response size and reduces server costs in exchange for small increase in computation.

Results summary and significance

- The schemes reduce the response size by at least 2x compared to prior keyword PIR constructions.
- Keyword PIR may be built with identical communication and computation costs as standard PIR.

Strengths Of SparsePIR

SparsePIR's main advantage lies in its efficiency. It surpasses state-of-the-art keyword PIR schemes by achieving only half the response overhead, making it highly promising for applications dealing with large and sparse databases.

Strengths Of SparsePIR

Furthermore, the simplicity of SparsePIR is noteworthy. It utilizes a straightforward encoding scheme, which makes it easily comprehensible and implementable. Consequently, SparsePIR becomes an excellent option for applications prioritizing simplicity over the utmost security.

Weaknesses Of SparsePIR

Firstly, SparsePIR introduces a slight increase in server-side computation time due to cryptographic operations and encoding/decoding requirements.

Weaknesses Of SparsePIR

However, while SparsePIR offers simplicity and efficiency, it may provide reduced security guarantees compared to more complex PIR schemes for the following reasons:

- SparsePIR only supports simple queries, limiting its query capabilities. If a user can only issue queries for a single item, an attacker could exploit this to gain insights into the user's data through a series of queries.
- SparsePIR is optimized for smaller databases compared to other PIR schemes. If the database is small, attackers may attempt to brute-force the encryption scheme to access the user's data.

Weaknesses Of SparsePIR

Additionally, SparsePIR is most effective in scenarios with large and sparse databases, but it may not perform as well when dealing with dense databases or databases with frequent updates.

Open Problems

Multi Servers and Complicated Queries: SparsePIR is not suitable for multiple servers, and it also lacks support for complex queries in its protocol.

References

- Sarvar Patel, Joon Young Seo, and Kevin Yeo. Don’t be dense: Efficient keyword pir for sparse databases. USENIX, 2023. https://www.usenix.org/conference/usenixsecurity23/presentation/patel.