Private Information Retrieval (PIR)

Does not learn index $i$
Private Information Retrieval (PIR)

Our focus: single-server setting

record $i$

Basic building block in many privacy-preserving protocols

- Metadata-private messaging
- Contact discovery
- Safe browsing
- Private DNS
- Private contact tracing
- Private navigation

[CGKS95]
Efficiency Metrics

1. Query size
   query → response

2. Server Throughput
   \[
   \text{database size} \quad \text{server computation time}
   \]
   “measures how fast the server can respond as a function of database size”
Efficiency Metrics

1. **Query size**
   - query
   - response
   - Without preprocessing, server must perform a linear scan over the database.

2. **Server Throughput**
   - \( \frac{\text{database size}}{\text{server computation time}} \)
   - “measures how fast the server can respond as a function of database size”
Efficiency Metrics

1. Query size
2. Server Throughput
3. Rate
4. Public parameter size

Client generates a reusable set of public parameters.

- Query size measures how fast the server can respond.
- Server Throughput measures communication overhead in responses.
- Rate is calculated as record size divided by response size.
- Public parameter size measures how fast the server can respond as a function of database size.

The diagram illustrates the flow from client to server, with public parameters being generated initially.
The SPIRAL Family of PIR Protocols

Techniques to translate between FHE schemes enables new trade-offs in single-server PIR

Automatic parameter selection based on database configuration

**Base version of SPIRAL**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query size</td>
<td>14 KB</td>
<td>4.5× smaller</td>
</tr>
<tr>
<td>Rate</td>
<td>0.41</td>
<td>2.1× higher</td>
</tr>
<tr>
<td>Throughput</td>
<td>333 MB/s</td>
<td>2.9× higher</td>
</tr>
</tbody>
</table>

(Database with $2^{14}$ records of size 100 KB)

**Streaming versions of SPIRAL**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rate</td>
<td>0.81</td>
<td>3.4× smaller responses</td>
</tr>
<tr>
<td>Throughput</td>
<td>1.9 GB/s</td>
<td>12.3× higher</td>
</tr>
</tbody>
</table>

**Best previous protocol:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rate</td>
<td>0.24</td>
</tr>
<tr>
<td>Throughput</td>
<td>158 MB/s</td>
</tr>
</tbody>
</table>

**Cost:** 3.4× larger public parameters (17 MB)
The SPIRAL Family of PIR Protocols

Techniques to translate between FHE schemes enables new trade-offs in single-server PIR.

Automatic parameter selection based on database configuration.

**Base version of SPIRAL**
- Query size: 14 KB
- Rate: 0.41
- Throughput: 333 MB/s

Throughput: 2.9× higher

(Database with $2^{14}$ records of size 100 KB)

**Cost:** 3.4× larger public parameters (17 MB)

**Streaming versions of SPIRAL**
- Rate: 0.81
- Throughput: 1.9 GB/s

Throughput: 12.3× higher

**Best previous protocol:**
- Rate: 0.24
- Throughput: 158 MB/s

Higher throughput than running software AES over database

(Primary operation: 64-bit integer arithmetic)
The SPIRAL Family of PIR Protocols

Techniques to translate between FHE schemes enable new trade-offs in single-server PIR.

Automatic parameter selection based on database configuration allows for optimal performance.

**Base version of SPIRAL**

| Query size: | 14 KB | 4.5× smaller |
| Rate:       | 0.41  | 2.1× higher  |
| Throughput: | 333 MB/s | 2.9× higher |

(Database with $2^{14}$ records of size 100 KB)

**Cost:** 3.4× larger public parameters (17 MB)

**Streaming versions of SPIRAL**

| Rate:       | 0.81  | 3.4× smaller responses |
| Throughput: | 1.9 GB/s | 12.3× higher |

**Best previous protocol:**

| Rate: | 0.24 |
| Throughput: | 158 MB/s |

Cost of privately streaming a 2 GB movie from database of $2^{14}$ movies estimated to be 1.9× more expensive than no-privacy baseline (based on AWS compute costs).
### PIR from Homomorphic Encryption

**Starting point:** a $\sqrt{N}$ construction ($N = \text{number of records}$)

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_{11}$</td>
<td>$r_{12}$</td>
<td>$r_{13}$</td>
<td>$r_{14}$</td>
</tr>
<tr>
<td>$r_{21}$</td>
<td>$r_{22}$</td>
<td>$r_{23}$</td>
<td>$r_{24}$</td>
</tr>
<tr>
<td>$r_{31}$</td>
<td>$r_{32}$</td>
<td>$r_{33}$</td>
<td>$r_{34}$</td>
</tr>
<tr>
<td>$r_{41}$</td>
<td>$r_{42}$</td>
<td>$r_{43}$</td>
<td>$r_{44}$</td>
</tr>
</tbody>
</table>

Arrange the database as a $\sqrt{N}$-by-$\sqrt{N}$ matrix.
PIR from Homomorphic Encryption

**Starting point:** a $\sqrt{N}$ construction ($N$ = number of records)

Arrange the database as a $\sqrt{N}$-by-$\sqrt{N}$ matrix

Encrypt a 0/1 vector indicating the row containing the desired record

Homomorphically compute product between query vector and database matrix
PIR from Homomorphic Encryption

Starting point: a $\sqrt{N}$ construction ($N =$ number of records)

Arrange the database as a $\sqrt{N}$-by-$\sqrt{N}$ matrix

Encrypt a 0/1 vector indicating the row containing the desired record

Database is in the clear, so additive homomorphism suffices
**PIR from Homomorphic Encryption**

**Starting point:** a $\sqrt{N}$ construction ($N = \text{number of records}$)

Encrypt a 0/1 vector indicating the row containing the desired record

Client decrypts to learn records

Response size: $\sqrt{N} \cdot \text{poly}(\lambda)$

*Homomorphically* compute product between query vector and database matrix
PIR from Homomorphic Encryption

Starting point: a \( \sqrt{N} \) construction (\( N = \) number of records)

Client decrypts to learn records

Encrypt a 0/1 vector indicating the row containing the desired record

\( r_{31}, r_{32}, r_{33}, r_{34} \)

Response size: \( \sqrt{N} \cdot \text{poly}(\lambda) \)

Homomorphically compute product between query vector and database matrix

\( \lambda \) is security parameter
PIR from Homomorphic Encryption

Beyond $\sqrt{N}$ communication: view the database as *hypercube*

**Approach:** Use homomorphic multiplication

Gentry-Halevi [GH19] OnionPIR [MCR21]
SPIRAL: Composing FHE Schemes

Follows Gentry-Halevi blueprint of composing two lattice-based FHE schemes:

FHE ciphertexts are noisy encodings
Homomorphic operations increase noise; more noise = larger parameters = less efficiency

**Scheme 1:** Regev’s encryption scheme [Reg04]

High-rate; only supports additive homomorphism

**Scheme 2:** Gentry-Sahai-Waters encryption scheme [GSW13]

Low rate; supports homomorphic multiplication (with additive noise growth)

Goal: get the best of both worlds
Regev Encodings (over Rings)

Regev encoding of a scalar $m \in R$:

- Secret key allows recovery of noisy version of original message
- To support decryption of “small” values $t \in R_p$, we encode $t$ as $(q/p)t$
- Decryption recovers noisy version of $(q/p)t$ and rounding yields $t$

$$\text{rate} = \frac{\log p}{2 \log q} < \frac{1}{2}$$

OnionPIR: rate = 0.24

All elements are polynomials in the ring $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = 2^k$
Regev encoding of a matrix $M \in R_{q}^{n \times n}$:

$$S^{\top}R_{q}^{n \times (n+1)} \approx R_{q}^{(n+1) \times n}C \approx M$$

Idea: “Reuse” encryption randomness

Rate:

$$\text{rate} = \frac{n^2 \log p}{n(n+1) \log q} = \frac{n^2 \log p}{n^2 + n \log q}$$

Additively homomorphic:

$$S^{\top}C_1 \approx M_1$$

$$S^{\top}C_2 \approx M_2$$

$$S^{\top}(C_1 + C_2) \approx M_1 + M_2$$

All elements are polynomials in the ring $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = 2^k$
Gentry-Sahai-Waters Encodings

**GSW encoding** of a bit $\mu \in \{0,1\}$:

$$R_q^{n \times (n+1)} \quad R_q^{(n+1) \times n} \quad R_q^{n \times (n+1)} \quad R_q^{(n+1) \times m}$$

$$S^T \quad C \quad \approx \quad \mu \quad S^T \quad G$$

$m = (n + 1) \log q$

**Gadget matrix** [MP12]:

$$G = \begin{bmatrix} g^T & \ddots & \end{bmatrix}$$

$$g^T = [1 \quad 2 \quad 2^2 \quad \ldots \quad 2^{\lceil \log_2 q \rceil}]$$

"Powers-of-2" matrix

Construction will use other decomposition bases

All elements are polynomials in the ring $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = 2^k$
Gentry-Sahai-Waters Encodings

GSW encoding of a bit $\mu \in \{0,1\}$:

$$
\begin{align*}
S^T & \approx \mu \\
C & \approx \mu
\end{align*}
$$

Gadget matrix $[MP12]$:

$$
G = \begin{bmatrix}
g^T \\
\vdots \\
g^T
\end{bmatrix}
$$

$$
g^T = [1 \ 2 \ 2^2 \ \ldots \ 2^{\log_2 q}]
$$

"Powers-of-2" matrix

Main property: for every vector $\nu \in \mathbb{Z}_q^{n+1}$, can define $G^{-1}(\nu) \in \{0,1\}^m$ where $GG^{-1}(\nu) = \nu$

"binary decomposition"

All elements are polynomials in the ring $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = 2^k$
Gentry-Sahai-Waters Encodings

GSW encoding of a bit $\mu \in \{0,1\}$:

$$R_{q}^{n\times(n+1)} R_{q}^{(n+1)\times n} \approx \mu S^T \quad C \quad S^T \quad R_{q}^{n\times(n+1)} R_{q}^{(n+1)\times m}$$

rate = $\frac{1}{d(n+1)^2 \log q}$

Concretely: $d = 2048, n \geq 1, q = 2^{56}$

All elements are polynomials in the ring $R = \mathbb{Z}[x]/(x^d + 1)$ where $d = 2^k$
Regev-GSW Homomorphism

\[ S^T C_{\text{Reg}} \approx M \]
\[ S^T C_{\text{GSW}} \approx \mu S^T G \]

With noise terms:
\[ S^T C_{\text{GSW}} G^{-1}(C_{\text{Reg}}) = \mu M + E_{\text{GSW}} G^{-1}(C_{\text{Reg}}) + \mu E_{\text{Reg}} \]

**Asymmetric noise growth:** if all GSW ciphertexts are “fresh,” then noise accumulation is additive in the number of multiplications.

\[ C_{\text{GSW}} G^{-1}(C_{\text{Reg}}) \] is a Regev encoding of \( \mu M \)
The Gentry-Halevi Blueprint

Database is represented as $2^{\nu_1} \times 2 \times 2 \times \cdots \times 2$ hypercube.

Query contains $2^{\nu_1}$ matrix Regev ciphertexts

$$\begin{bmatrix} 0 & I_n & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Indicator for index along first dimension

Query contains $\nu_2$ GSW ciphertexts

$$\begin{bmatrix} 0 & 1 & 1 & 0 \end{bmatrix}$$

Indicator for index along subsequent dimensions

Response is a single matrix Regev ciphertext.

Each GSW ciphertext participates in only one multiplication with a Regev ciphertext!
The Gentry-Halevi Blueprint

Database is represented as $2^{\nu_1} \times \frac{2}{2^{\nu_2}} \times 2 \times 2 \times \cdots \times 2$ hypercube.

**Drawback:** large queries

Can compress using polynomial encoding method of Angel et al. [ACLS18]

**Estimated size:** 4 MB/ciphertext

**Estimated query size:** 30 MB

Query contains $2^{\nu_1}$ matrix Regev ciphertexts

```
0  I_n  0  0  0  0  0
```

Indicator for index along first dimension

Query contains $\nu_2$ GSW ciphertexts

```
0  1  1  0
```

Indicator for index along subsequent dimensions
The Gentry-Halevi Blueprint

Database is represented as $2^\nu_1 \times 2 \times 2 \times \cdots \times 2$ hypercube

Drawback: large queries

Query contains $2^{\nu_1}$ matrix Regev ciphertexts

\[
\begin{bmatrix}
0 & I_n & 0 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\]

Indicator for index along first dimension

Query contains $\nu_2$ GSW ciphertexts

\[
\begin{bmatrix}
0 & 1 & 1 & 0 \\
\end{bmatrix}
\]

Indicator for index along subsequent dimensions

Can compress using polynomial encoding method of Angel et al. [ACLS18]

SealPIR query size: 66 KB

Estimated query size: 30 MB

[GH19]
High-level: Gentry-Halevi approach with *scalar* Regev ciphertexts ($n = 1$)

Leverages Chen et al. approach [CCR19] to “assemble” GSW ciphertext using Regev-GSW multiplication

  Regev ciphertexts can be packed using polynomial encoding method [ACLS18, CCR19]

Use of scalar Regev ciphertexts reduces the rate to $\approx 0.24$ (over 4× response overhead)
“Best of both worlds”: Small queries (as in OnionPIR) with the high rate/throughput of the Gentry-Halevi scheme

- **Query size**: 14 KB
- **Rate**: 0.41
- **Throughput**: 333 MB/s

2000× smaller than Gentry-Halevi (4.5× smaller than OnionPIR)
2.1× higher than OnionPIR
2.9× higher than OnionPIR

(Database with $2^{14}$ records of size 100 KB)

**Cost**: 3.4× larger public parameters for extra translation keys

Leverage simple key-switching techniques for query and response compression

- **Scalar Regev → Matrix Regev**
- **Matrix Regev → GSW**

Query compression

Scalar Regev → Matrix Regev

Response compression (for large records)
Scalar Regev → Matrix Regev

**Input:** encoding $c$ where $s_1^T c \approx m$

**Output:** encoding $C$ where $S_2^T C \approx mI_n$

$S_2^T = \begin{bmatrix} -\tilde{s}_0 & \cdots & -\tilde{s}_0 \end{bmatrix}$

$c = \begin{bmatrix} c_0 & \cdots & c_0 \end{bmatrix}$

$S_2^T C = mI_n$

$s_1^T = [-\tilde{s}_0 \mid 1] \in R_q^2$

$c^T = [c_0 \mid c_1] \in R_q^2$

Can replace with $S_2$ with arbitrary secret key using standard key-switching techniques
Goal: use Regev encodings to construct $C$ such that $S^T C \approx \mu S^T G$

$S^T = [-s | I_n] \in \mathbb{R}_q^{n \times (n+1)}$

$G = \begin{bmatrix} g^T \\ \vdots \\ g^T \end{bmatrix}$

$rearrange$ $g^T$

$\mu S^T G = \begin{bmatrix} -\mu s g^T \\ \mu I_n \\ 2\mu I_n \\ 2^2 \mu I_n \\ \cdots \\ 2^t \mu I_n \end{bmatrix}$

$A \quad B_0 \quad B_1 \quad B_2 \quad \cdots \quad B_t$

Break $C$ into blocks
Goal: use Regev encodings to construct $C$ such that $S^T C \approx \mu S^T G$

$S^T C = S^T A \begin{bmatrix} S^T B_0 & S^T B_1 & S^T B_2 & \cdots & S^T B_t \end{bmatrix}$

$\mu S^T G = -\mu s g^T \begin{bmatrix} \mu I_n & 2\mu I_n & 2^2 \mu I_n & \cdots & 2^t \mu I_n \end{bmatrix}$

$B_0, \ldots, B_t$ are matrix Regev ciphertexts encrypting $\mu I_n, 2\mu I_n, \ldots, 2^t \mu I_n$

Can derive from scalar Regev encodings of $\mu, 2\mu, \ldots, 2^t \mu$
Goal: use Regev encodings to construct \( C \) such that \( S^T C \approx \mu S^T G \)

\[
S^T C = \begin{bmatrix} S^T A & S^T B_0 & S^T B_1 & \cdots & S^T B_t \end{bmatrix}
\]

\[
\mu S^T G = \begin{bmatrix} -\mu s g^T & \mu I_n & 2\mu I_n & 2^2\mu I_n & \cdots & 2^t\mu I_n \end{bmatrix}
\]

Write \( S^T = [-s \mid I_n] \)

Let \( s_{\text{Reg}} \) be the key for a Regev encoding scheme

Construct key-switching matrix \( W \):

\[
S^T W \approx -s \left( s_{\text{Reg}}^T \otimes g^T \right)
\]

Let \( c_0, \ldots, c_t \) be encodings of \( \mu, \ldots, 2^t \mu \) under \( s_{\text{Reg}} \): \( s_{\text{Reg}}^T c_i \approx 2^i \mu \)

Let \( C = [c_0 \mid \cdots \mid c_t] \)

Then, \( S^T W g^{-1}(C) \approx -s \left( s_{\text{Reg}}^T \otimes g^T \right) g^{-1}(C) \approx -s[\mu \mid \cdots \mid 2^t \mu] = -\mu s g^T \)
Matrix Regev → GSW

**Goal:** use Regev encodings to construct $C$ such that $S^T C \approx \mu S^T G$

\[
S^T C = \begin{bmatrix} S^T A & S^T B_0 & S^T B_1 & \cdots & S^T B_t \end{bmatrix}
\]

\[
\mu S^T G = \begin{bmatrix} -\mu s g^T & \mu I_n & 2\mu I_n & 2^2 \mu I_n & \cdots & 2^t \mu I_n \end{bmatrix}
\]

Write $S^T = [-s | I_n]$

Let $s_{\text{Reg}}$ be the key for a Regev encoding scheme

Construct key-switching matrix $W$:

\[
S^T W \approx -s \left( s_{\text{Reg}}^T \otimes g^T \right)
\]

Let $c_0, \ldots, c_t$ be encoding of $\mu$ under $s_{\text{Reg}}$: $s_{\text{Reg}}^T c_i \approx 2^i \mu$

Let $C = W g^{-1}(C)$

Then, $S^T W g^{-1}(C) \approx -s \left( s_{\text{Reg}}^T \otimes g^T \right) g^{-1}(C) \approx -s [\mu | \cdots | 2^t \mu] = -\mu s g^T$
Matrix Regev $\rightarrow$ GSW

Scalar Regev to Matrix Regev

$Wg^{-1}([c_0| \cdots |c_t])$

$\begin{bmatrix} \mu \\ 2\mu \\ \vdots \\ 2^t\mu \end{bmatrix}$

$\begin{bmatrix} \mu I_n \\ \vdots \\ 2^t\mu I_n \end{bmatrix}$

$-\mu sg^T$

$\begin{bmatrix} -\mu sg^T \\ \mu I_n \\ \cdots \\ 2^t\mu I_n \end{bmatrix}$

Ciphertext contains $(n + 1)^2(t + 1)$ elements of $R_q$

Takeaway: instead of sending $(n + 1)^2(t + 1)$ ring elements per GSW ciphertext, only need to send $2(t + 1)$

scalar Regev encodings: elements of $R_q^2$

matrix Regev encodings: elements of $R_q^{(n+1)\times n}$
Further Compression via Polynomial Encodings

\[ f(x) = \alpha_0 + \alpha_1 x + \cdots + \alpha_t \cdot x^t \] with \( t < d \)

\begin{align*}
c_0 & \quad \mu \\
c_1 & \quad 2\mu \\
\vdots & \\
c_t & \quad 2^t \mu \\
\end{align*}

Expands a Regev encoding of a polynomial into Regev encodings of its coefficients

**Takeaway:** We can pack \((\mu, 2\mu, \ldots, 2^t \mu)\) into a single polynomial

As long as \( t + 1 < d \), client and communicate a GSW ciphertext with a single Regev encoding (2 ring elements)

\((n + 1)^2(t + 1)\) ring elements

\(2\) ring elements

Cost: additional (reusable) public parameters needed for Regev-to-GSW translation
Database is represented as $2^{\nu_1} \times 2 \times 2 \times \cdots \times 2$ hypercube

Query contains $2^{\nu_1}$ matrix Regev ciphertexts

Query contains $\nu_2$ GSW ciphertexts

Indicator for index along first dimension

Indicator for index along subsequent dimensions

1. Compress into scalar Regev encodings

2. Pack scalars into single polynomial
Query Expansion in Spiral

Trade-off: larger public parameters, smaller queries

- SealPIR: 3 MB
- OnionPIR: 5 MB
- SPIRAL: 18 MB

- SealPIR: 66 KB
- OnionPIR: 63 KB
- SPIRAL: 14 KB

Moving costs from online to offline phase

SPIRAL also achieves higher rate and throughput
Response Compression via Modulus Switching

PIR response consists of a single matrix Regev encoding

\[
\begin{bmatrix} -s & I_n \end{bmatrix} \approx \frac{q}{p} \cdot M
\]

Modulus \( q \) must be large enough to support target number of homomorphic operations

rate \( \propto \frac{\log p}{\log q} \)

Standard technique in FHE: *modulus reduction*

Rescale ciphertext by \( \frac{q'}{q} \) where \( q' < q \)

rate \( \propto \frac{\log p}{\log q'} \)

Rescaling introduces small amount of noise (from rounding)

This work: Observe that rounding error \( E \) is scaled by \( \begin{bmatrix} -s & I_n \end{bmatrix} \)
PIR response consists of a single matrix Regev encoding

$$\begin{bmatrix} -s | I_n \end{bmatrix} \cong \frac{q}{p} \cdot M$$

Modulus $q$ must be large enough to support target number of homomorphic operations

$$\text{rate } \propto \frac{\log p}{\log q}$$

Standard technique in FHE: *modulus reduction*

Rescale ciphertext by $\frac{q'}{q}$ where $q' < q$

$$\text{rate } \propto \frac{\log p}{\log q'}$$

Rescaling introduces small amount of noise (from rounding)

**This work:** Observe that rounding error $E$ is scaled by $\begin{bmatrix} -s | I_n \end{bmatrix}$

Error scaled by $-s$

Error scaled by $I_n$
Response Compression via Modulus Switching

PIR response consists of a single matrix Regev encoding

\[
\begin{bmatrix}
-s & \mathbf{I}_n
\end{bmatrix}
\approx \frac{q}{p} \cdot M
\]

Observation: At least half of the error components are scaled by identity matrix!

Approach: Use two different moduli to rescale the ciphertext

Standard technique in FHE: \textit{modulus reduction}

Rescale ciphertext by \(\frac{q'}{q}\) where \(q' < q\)

Rate \(\propto \frac{\log p}{\log q'}\)

Rescaling introduces small amount of noise (from rounding)

This work: Observe that rounding error \(E\) is scaled by \(\begin{bmatrix}-s & \mathbf{I}_n\end{bmatrix}\)

Error scaled by \(-s\)

Error scaled by \(\mathbf{I}_n\)
Response Compression via Modulus Switching

PIR response consists of a single matrix Regev encoding

\[
\begin{align*}
C &= \begin{bmatrix} c_0^T & c_1 \end{bmatrix} \\
\end{align*}
\]

Rescale by \(q_2/q\)

Rescale by \(q_1/q\)

Observation: At least half of the error components are scaled by identity matrix!

Approach: Use two different moduli to rescale the ciphertext

Standard technique in FHE: *modulus reduction*

Rescale ciphertext by \(q'/q\) where \(q' < q\)

Rate \(\propto \frac{\log p}{\log q'}\)

Rescaling introduces small amount of noise (from rounding)

This work: Observe that rounding error \(E\) is scaled by \([-s | I_n]\)

Error scaled by \(-s\)

Error scaled by \(I_n\)
Response Compression via Modulus Switching

PIR response consists of a single matrix Regev encoding

\[ C = \begin{pmatrix} c_0^T \\ c_1 \end{pmatrix} \]

Rescale by \( q_2/q \)

Rescale by \( q_1/q \)

\[ \tilde{C}_0 \]

\[ \tilde{C}_1 \]

**Observation:** At least half of the error components are scaled by identity matrix!

**Approach:** Use two different moduli to rescale the ciphertext

\[
\text{rate} = \frac{n^2 \log p}{n^2 \log q_1 + n \log q_2}
\]

- SealPIR 0.01
- Gentry-Halevi (estimated) 0.44
- OnionPIR 0.24

**Overall rate:** 0.34 (with vanilla modulus switching)
0.81 (with split modulus switching)

**This work:** Observe that rounding error \( E \) is scaled by \([-s \mid I_n]\)

Error scaled by \(-s\)

\[
\begin{bmatrix} -s \mid I_n \end{bmatrix}
\]

Error scaled by \( I_n \)

\[
e_0^T \quad E_1
\]
Vanilla SPIRAL

record $i$

public parameters

Key-switching matrices for ciphertext expansion and translation
Vanilla SPIRAL

Public parameters

Query

Single scalar Regev encoding of a polynomial

Homomorphic expansion

<table>
<thead>
<tr>
<th>0</th>
<th>I_n</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
Vanilla SPIRAL

Public parameters → Query → Homomorphic expansion

First dimension processing

Regev encodings for first dimension

GSW encodings for subsequent dimensions

Regev-GSW folding
Many parameter choices in SPIRAL:
- Plaintext matrix dimension
- Plaintext modulus
- Decomposition bases for key-switching
- Database arrangement

Trade-offs in public parameter size, query size, server throughput, and rate

Use estimated running time + compute cost to choose parameters for an input database configuration

Automatic parameter selection tool
**Basic Comparisons**

<table>
<thead>
<tr>
<th>Database</th>
<th>Metric</th>
<th>SealPIR</th>
<th>FastPIR</th>
<th>OnionPIR</th>
<th>SPIRAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^{18} records</td>
<td>Public Param. Size</td>
<td>3 MB</td>
<td>1 MB</td>
<td>5 MB</td>
<td>18 MB</td>
</tr>
<tr>
<td>30 KB records</td>
<td>Query Size</td>
<td>66 KB</td>
<td>8 MB</td>
<td>63 KB</td>
<td>14 KB</td>
</tr>
<tr>
<td>(7.9 GB database)</td>
<td>Response Size</td>
<td>3 MB</td>
<td>262 KB</td>
<td>127 KB</td>
<td>84 KB</td>
</tr>
<tr>
<td></td>
<td>Server Compute</td>
<td>74.91 s</td>
<td>50.5 s</td>
<td>52.7 s</td>
<td>24.5 s</td>
</tr>
</tbody>
</table>

Database configuration preferred by OnionPIR

**Compared to OnionPIR:**
- reduce query size by 4.5×
- reduce response size by 2×
- reduce compute time by 2×

Throughput:
- Rate: 0.24
- Throughput: 149 MB/s
Basic Comparisons (with Larger Records)

Throughput for 100 GB database ($2^{20}$ records):

- **SPIRAL**: 310 MB/s (322 s)
- **SealPIR**: 102 MB/s (977 s)
- **FastPIR**: 189 MB/s (528 s)
- **OnionPIR**: 122 MB/s (817 s)

**SPIRAL** also has smaller query size and response size, but larger public parameters.

Server cost is *linear* in database size.

All measurements based on single-thread/single-core processing.
Basic Comparisons (with Larger Records)

Client costs:
- Generating reusable public parameters is the most expensive operation, but still < 1 s
- Query generation and response decoding are fast (30 ms and < 1 ms)

Server costs:
- Query expansion typically takes ≈ 1 second (less than 1.5% of overall compute when number of records is large)
- Parameter selection favors configurations that evenly distributes the work between first layer processing and ciphertext folding

(see paper for detailed microbenchmarks)
The Streaming Setting: SPIRALSTREAM

**Streaming setting:** same query reused over multiple databases

- Private video stream (database $D_i$ contains $i^{th}$ block of media) [GCMSAW16]
- Private voice calls (repeated polling of the same “mailbox”) [AS16, AYAAG21]

**Goal:** minimize online costs (i.e., server compute, response size)

Consider larger public parameters or query size (amortized over lifetime of stream)

**Matrix Regev encodings**

```
0 1 0 0
```

```
0 1 0 0
```

**GSW encodings**

```
1 z z^2 z^3
```

```
0 1 1
```

**SPIRAL query expansion**
Removing the initial expansion **significantly** reduces the noise growth from query expansion.

- Decreases size of public parameters (no more automorphism keys)
- Better control of noise growth $\Rightarrow$ higher server throughput and higher rate
- Larger queries (more Regev encodings)

**The Streaming Setting: SPIRALSTREAM**

**SPIRALSTREAM** query

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$z$</td>
<td>$z^2$</td>
<td>$z^3$</td>
</tr>
<tr>
<td>1</td>
<td>$z$</td>
<td>$z^2$</td>
<td>$z^3$</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$\rightarrow$ 0 $I_n$ 0 0

Matrix Regev encodings

$\rightarrow$ 0 1 1

GSW encodings

**SPIRALSTREAM** query expansion
## The Streaming Setting: SPIRALSTREAM

<table>
<thead>
<tr>
<th>Database</th>
<th>Metric</th>
<th>OnionPIR</th>
<th>SPIRAL</th>
<th>SPIRALSTREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>2^{18} records</td>
<td>Public Param. Size</td>
<td>5 MB</td>
<td>18 MB</td>
<td>3 MB</td>
</tr>
<tr>
<td>30 KB records</td>
<td>Query Size</td>
<td>63 KB</td>
<td>14 KB</td>
<td>15 MB</td>
</tr>
<tr>
<td>(7.9 GB database)</td>
<td>Response Size</td>
<td>127 KB</td>
<td>84 KB</td>
<td>62 KB</td>
</tr>
<tr>
<td></td>
<td>Server Compute</td>
<td>52.7 s</td>
<td>24.5 s</td>
<td>9.0 s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rate:</th>
<th>0.23</th>
<th>0.36</th>
<th>0.48</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput:</td>
<td>149 MB/s</td>
<td>322 MB/s</td>
<td>874 MB/s</td>
</tr>
</tbody>
</table>

- 25% reduction in response size
- 2.7× increase in throughput
The Streaming Setting: SPIRALSTREAM

Streaming throughput: ignoring query expansion costs, assuming optimal record size for each system

Peaks at ≈1.5 GB/s (over 7× faster than previous constructions)
Higher Rate via Response Packing: SPIRAL PACK

Can we further reduce response size?

$$\text{rate} = \frac{n^2 \log p}{n \log q_2 + n^2 \log q_1}$$

$q_1 = 4p$

Increasing the plaintext dimension $n$ increases the rate

SPIRAL and SPIRAL STREAM use $n = 2$

Higher values of $n$ increases computational cost

Each Regev encoding is a $(n + 1) \times n$ matrix, so number of ring operations per homomorphic operation scale with $O(n^3)$

[Not using fast matrix multiplications here]

SPIRAL PACK: Perform homomorphic operations with $n = 1$ and pack responses
Higher Rate via Response Packing: SPIRALPACK

Plaintext space: $R_p^{\times n}

Each record is $n \times n$ matrix

Response consists of $n^2$ Regev encodings

Split database into $n^2$ databases

$i^{th}$ database contains $i^{th}$ entry of record
(elements of $R_p$)

Better throughput
Worse rate
Higher Rate via Response Packing: SPIRALPACK

Variant of scalar Regev to matrix Regev transformation

Requires publishing $n$ key-switching matrices

Packing done only at the very end (cost does not scale with number of records)

$n^2$ Regev ciphertexts with dimension 1

Consists of $2n^2$ ring elements

1 Regev ciphertext with dimension $n$

Consists of $n(n + 1)$ ring elements

SPIRALPACK: higher throughput and rate (for sufficiently large records), larger public parameters
Higher Rate via Response Packing: SPIRALPACK

<table>
<thead>
<tr>
<th>Database Metric</th>
<th>OnionPIR</th>
<th>SPIRAL</th>
<th>SPIRALSTREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Public Param. Size</td>
<td>5 MB</td>
<td>18 MB → 18 MB</td>
<td>3 MB → 16 MB</td>
</tr>
<tr>
<td>Query Size</td>
<td>63 KB</td>
<td>14 KB → 14 KB</td>
<td>15 MB → 30 MB</td>
</tr>
<tr>
<td>Response Size</td>
<td>127 KB</td>
<td>84 KB → 86 KB</td>
<td>62 KB → 96 KB</td>
</tr>
<tr>
<td>Server Compute</td>
<td>52.7 s</td>
<td>24.5 s → 17.7 s</td>
<td>9.0 s → 5.3 s</td>
</tr>
</tbody>
</table>

- Small records ⇒ can only take advantage of low packing dimension
- Higher throughputs since homomorphic operations cheaper
- Responses larger due to extra noise from response packing
## Higher Rate via Response Packing: SPIRALPACK

<table>
<thead>
<tr>
<th>Database Size</th>
<th>Metric</th>
<th>OnionPIR</th>
<th>SPIRAL</th>
<th>SPIRALSTREAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{18}$ records</td>
<td>Public Param. Size</td>
<td>5 MB</td>
<td>18 MB</td>
<td>3 MB</td>
</tr>
<tr>
<td></td>
<td>Query Size</td>
<td>63 KB</td>
<td>14 KB</td>
<td>15 MB</td>
</tr>
<tr>
<td></td>
<td>Response Size</td>
<td>127 KB</td>
<td>84 KB</td>
<td>62 KB</td>
</tr>
<tr>
<td></td>
<td>Server Compute</td>
<td>52.7 s</td>
<td>24.5 s</td>
<td>9.0 s</td>
</tr>
<tr>
<td>30 KB records</td>
<td></td>
<td></td>
<td>$\rightarrow$ 18 MB</td>
<td>$\rightarrow$ 16 MB</td>
</tr>
<tr>
<td>(7.9 GB database)</td>
<td></td>
<td></td>
<td>$\rightarrow$ 14 KB</td>
<td>$\rightarrow$ 30 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\rightarrow$ 86 KB</td>
<td>$\rightarrow$ 96 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\rightarrow$ 17.7 s</td>
<td>$\rightarrow$ 5.3 s</td>
</tr>
<tr>
<td>$2^{14}$ records</td>
<td>Public Param. Size</td>
<td>5 MB</td>
<td>17 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td></td>
<td>Query Size</td>
<td>63 KB</td>
<td>14 KB</td>
<td>8 MB</td>
</tr>
<tr>
<td></td>
<td>Response Size</td>
<td>508 KB</td>
<td>242 KB</td>
<td>208 KB</td>
</tr>
<tr>
<td></td>
<td>Server Compute</td>
<td>14.4 s</td>
<td>4.92 s</td>
<td>2.4 s</td>
</tr>
<tr>
<td>100 KB records</td>
<td></td>
<td></td>
<td>$\rightarrow$ 47 MB</td>
<td>$\rightarrow$ 24 MB</td>
</tr>
<tr>
<td>(1.6 GB database)</td>
<td></td>
<td></td>
<td>$\rightarrow$ 14 KB</td>
<td>$\rightarrow$ 30 MB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\rightarrow$ 188 KB</td>
<td>$\rightarrow$ 150 KB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$\rightarrow$ 4.58 s</td>
<td>$\rightarrow$ 1.2 s</td>
</tr>
</tbody>
</table>

**Rate:**
- OnionPIR: 0.20
- SPIRAL: 0.41 → 0.53
- SPIRALSTREAM: 0.48 → 0.67

**Throughput:**
- OnionPIR: 114 MB/s
- SPIRAL: 333 MB/s → 358 MB/s
- SPIRALSTREAM: 683 MB/s → 1.4 GB/s

With 100 KB records, higher rate **and** throughput in exchange for larger public parameters.
Streaming throughput: ignoring query expansion costs, assuming optimal record size for each system

Packing outperforms non-packed protocol for streaming settings
Packing in the Streaming Setting

**Streaming throughput:** ignoring query expansion costs, assuming optimal record size for each system

Packing outperforms non-packed protocol for streaming settings

1.94 GB/s and a rate of 0.81 (125 MB public parameter and 30 MB query)
Packing in the Streaming Setting

**Streaming throughput:** ignoring query expansion costs, assuming optimal record size for each system

Packing outperforms non-packed protocol for streaming settings

Cost of privately streaming a 2 GB movie from database of $2^{14}$ movies estimated to be $1.9 \times$ more expensive than no-privacy baseline (based on AWS compute costs)

Previously: $\approx 17 \times$ more expensive
A Systematic Way to Explore PIR Trade-Offs

Parameter selection tool can be used to minimize online cost with constraints on public parameter and query sizes

(Database configuration: $2^{14} \times 100$ KB database)
The SPIRAL Family of PIR

Techniques to translate between FHE schemes enables new trade-offs in single-server PIR

Scalar Regev $\rightarrow$ Matrix Regev
Regev $\rightarrow$ GSW

Query compression

Scalar Regev $\rightarrow$ Matrix Regev

Response compression (for large records)

Automatic parameter selection to choose lattice parameters based on database configuration

**Base version of SPIRAL**

<table>
<thead>
<tr>
<th></th>
<th>Query size</th>
<th>Rate</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Query size</strong></td>
<td>14 KB</td>
<td>0.41</td>
<td>333 MB/s</td>
</tr>
<tr>
<td><strong>Rate</strong></td>
<td></td>
<td>4.5×</td>
<td>2.1×</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td></td>
<td></td>
<td>2.9×</td>
</tr>
</tbody>
</table>

(Database with $2^{14}$ records of size 100 KB)

**Streaming versions of SPIRAL**

<table>
<thead>
<tr>
<th></th>
<th>Rate</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Rate</strong></td>
<td>0.81</td>
<td>1.9 GB/s</td>
</tr>
<tr>
<td><strong>Throughput</strong></td>
<td>3.4× smaller</td>
<td>12.3× higher</td>
</tr>
</tbody>
</table>

Improvements primarily due to query and response compression
The SPIRAL Family of PIR

Techniques to translate between FHE schemes enables new trade-offs in single-server PIR

Scalar Regev $\rightarrow$ Matrix Regev
Regev $\rightarrow$ GSW

Query compression

Scalar Regev $\rightarrow$ Matrix Regev

Response compression
(for large records)

Automatic parameter selection to choose lattice parameters based on database configuration

**Base version of SPIRAL**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query size</td>
<td>14 KB</td>
<td>4.5× smaller</td>
</tr>
<tr>
<td>Rate</td>
<td>0.41</td>
<td>2.1× higher</td>
</tr>
<tr>
<td>Throughput</td>
<td>333 MB/s</td>
<td>2.9× higher</td>
</tr>
</tbody>
</table>

(Database with $2^{14}$ records of size 100 KB)

**Streaming versions of SPIRAL**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rate</td>
<td>0.81</td>
<td>3.4× smaller</td>
</tr>
<tr>
<td>Throughput</td>
<td>1.9 GB/s</td>
<td>12.3× higher</td>
</tr>
</tbody>
</table>

Improvements primarily due to fine-tuning scheme parameters for database configuration
Future Directions

Leveraging FHE composition in other privacy-preserving systems
  - Private set intersection (PSI)
  - Oblivious RAM (ORAM)

Hardware acceleration for higher throughput
Leveraging preprocessing to achieve sublinear server computation

Paper: https://eprint.iacr.org/2022/368
Code: https://github.com/menonsamir/spiral

Thank you!