Faster Homomorphic Linear Transformations in HElib

Shai Halevi (IBM)
Victor Shoup (IBM & NYU)

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

. . . more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- ➪ in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- ➪ speed up of up to $\approx 6\times$ for bootstrapping as a whole

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

... more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- speed up of up to $\approx 6\times$ for bootstrapping as a whole

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

. . . more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- speed up of up to $\approx 6\times$ for bootstrapping as a whole

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

. . . more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- speed up of up to $\approx 6\times$ for bootstrapping as a whole

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

. . . more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- speed up of up to $\approx 6\times$ for bootstrapping as a whole

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

...more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- speed up of up to $\approx 6\times$ for bootstrapping as a whole

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

. . . more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- ➪ in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- ➪ speed up of up to $\approx 6\times$ for bootstrapping as a whole

Fully Homomorphic Encryption allows for arbitrary computation on encrypted data

In this talk, the focus is on *linear transformations*

...more specifically, applying a *fixed, public* linear transformation to a vector encrypted in the BGV (Brakerski-Gentry-Vaikuntanathan) cryptosystem

We present *new algorithms and their implementation* in HElib

We get speed ups of up to $\approx 75\times$

One important application: bootstrapping

- ➪ in Chen and Han's new bootstrapping algorithm (Eurocrypt 2018), most of the time is spent performing a change of basis
- ➪ speed up of up to $\approx 6\times$ for bootstrapping as a whole

BGV encryption

$R = \mathbb{Z}[X]/(\Phi_n(X))$

Plaintext space: $R_p := R/pR \quad (p = \text{small prime})$

Ciphertext space: $R_q := R/qR \quad (n, p, q \text{ pairwise coprime})$

Ciphertext: $\bar{c} \in R_q^{2\times 1}$

Secret key: $\bar{s} = (1, s_1) \in R_q^{2\times 1}$, where s_1 has small norm

Decryption: $\langle \bar{s}, \bar{c} \rangle = \underbrace{p\epsilon}_{\text{“noise”}} + m$

BGV encryption

$R = \mathbb{Z}[X]/(\Phi_n(X))$

Plaintext space: $R_p := R/pR \quad (p = \text{small prime})$

Ciphertext space: $R_q := R/qR \quad (n, p, q \text{ pairwise coprime})$

Ciphertext: $\bar{c} \in R_q^{2\times 1}$

Secret key: $\bar{s} = (1, s_1) \in R_q^{2\times 1}$, where s_1 has small norm

Decryption: $$\langle \bar{s}, \bar{c} \rangle = \underbrace{p\epsilon}_{\text{“noise”}} + m$$

BGV encryption

$R = \mathbb{Z}[X]/(\Phi_n(X))$

Plaintext space: $R_p := R/pR \quad (p = \text{small prime})$

Ciphertext space: $R_q := R/qR \quad (n, p, q \text{ pairwise coprime})$

Ciphertext: $\bar{c} \in R_q^{2\times 1}$

Secret key: $\bar{s} = (1, s_1) \in R_q^{2\times 1}$, where s_1 has small norm

Decryption:

$$\langle \bar{s}, \bar{c} \rangle = \underbrace{p\epsilon}_{\text{“noise”}} + m$$

BGV encryption

$R = \mathbb{Z}[X]/(\Phi_n(X))$

Plaintext space: $R_p := R/pR$ ($p =$ small prime)

Ciphertext space: $R_q := R/qR$ (n, p, q pairwise coprime)

Ciphertext: $\bar{c} \in R_q^{2\times 1}$

Secret key: $\bar{s} = (1, s_1) \in R_q^{2\times 1}$, where s_1 has small norm

Decryption:

$$\langle \bar{s}, \bar{c} \rangle = \underbrace{p\epsilon}_{\text{“noise”}} + m$$

BGV encryption

$R = \mathbb{Z}[X]/(\Phi_n(X))$

Plaintext space: $R_p := R/pR \quad (p = \text{small prime})$

Ciphertext space: $R_q := R/qR \quad (n, p, q \text{ pairwise coprime})$

Ciphertext: $\bar{c} \in R_q^{2\times 1}$

Secret key: $\bar{s} = (1, s_1) \in R_q^{2\times 1}$, where s_1 has small norm

Decryption: $\langle \bar{s}, \bar{c} \rangle = \underbrace{p\epsilon}_{\text{“noise”}} + m$

BGV encryption

$R = \mathbb{Z}[X]/(\Phi_n(X))$

Plaintext space: $R_p := R/pR \quad (p = \text{small prime})$

Ciphertext space: $R_q := R/qR \quad (n, p, q \text{ pairwise coprime})$

Ciphertext: $\bar{c} \in R_q^{2\times 1}$

Secret key: $\bar{s} = (1, s_1) \in R_q^{2\times 1}$, where s_1 has small norm

Decryption:

$$\langle \bar{s}, \bar{c} \rangle = \underbrace{p\epsilon}_{\text{“noise”}} + m$$

Representation of ciphertext space R_q

Coefficient representation

DoubleCRT representation

- $q = q_1 \cdots q_\ell$, where each q_i is a small prime such that $\mathbb{Z}_{q_i}$ contains nth roots of unity
- A polynomial in R_q is reduced modulo each q_i, and then evaluated at the primitive nth roots of unity in $\mathbb{Z}_{q_i}$

Addition of ciphertexts in DoubleCRT representation takes linear time

. . . so does multiplication by a constant

Switching between DoubleCRT and coefficient representations:

somewhat expensive (requires CRT and FFT)

Representation of ciphertext space R_q

Coefficient representation

DoubleCRT representation

- $q = q_1 \cdots q_\ell$, where each q_i is a small prime such that $\mathbb{Z}_{q_i}$ contains nth roots of unity
- A polynomial in R_q is reduced modulo each q_i, and then evaluated at the primitive nth roots of unity in $\mathbb{Z}_{q_i}$

Addition of ciphertexts in DoubleCRT representation takes linear time

. . . so does multiplication by a constant

Switching between DoubleCRT and coefficient representations:

somewhat expensive (requires CRT and FFT)

Representation of ciphertext space R_q

Coefficient representation

DoubleCRT representation

- $q = q_1 \cdots q_\ell$, where each q_i is a small prime such that $\mathbb{Z}_{q_i}$ contains nth roots of unity
- A polynomial in R_q is reduced modulo each q_i, and then evaluated at the primitive nth roots of unity in $\mathbb{Z}_{q_i}$

Addition of ciphertexts in DoubleCRT representation takes linear time

. . . so does multiplication by a constant

Switching between DoubleCRT and coefficient representations: somewhat expensive (requires CRT and FFT)

Representation of ciphertext space R_q

Coefficient representation

DoubleCRT representation

- $q = q_1 \cdots q_\ell$, where each q_i is a small prime such that $\mathbb{Z}_{q_i}$ contains nth roots of unity
- A polynomial in R_q is reduced modulo each q_i, and then evaluated at the primitive nth roots of unity in $\mathbb{Z}_{q_i}$

Addition of ciphertexts in DoubleCRT representation takes linear time

. . . so does multiplication by a constant

Switching between DoubleCRT and coefficient representations: somewhat expensive (requires CRT and FFT)

Representation of ciphertext space R_q

Coefficient representation

DoubleCRT representation

- $q = q_1 \cdots q_\ell$, where each q_i is a small prime such that $\mathbb{Z}_{q_i}$ contains nth roots of unity
- A polynomial in R_q is reduced modulo each q_i, and then evaluated at the primitive nth roots of unity in $\mathbb{Z}_{q_i}$

Addition of ciphertexts in DoubleCRT representation takes linear time
   . . . so does multiplication by a constant

Switching between DoubleCRT and coefficient representations:
   somewhat expensive (requires CRT and FFT)

Representation of ciphertext space R_q

Coefficient representation

DoubleCRT representation

- $q = q_1 \cdots q_\ell$, where each q_i is a small prime such that $\mathbb{Z}_{q_i}$ contains nth roots of unity
- A polynomial in R_q is reduced modulo each q_i, and then evaluated at the primitive nth roots of unity in $\mathbb{Z}_{q_i}$

Addition of ciphertexts in DoubleCRT representation takes linear time
 ...so does multiplication by a constant

Switching between DoubleCRT and coefficient representations:
 somewhat expensive (requires CRT and FFT)

Multiplication and Key Switching

Multiplying two ciphertexts in DoubleCRT representation takes linear time

But . . . we get a ciphertext defined with respect to a different secret key

So . . . we include an encryption of this other key under the original key in the public parameters (called a “key switching matrix”)

Using this, we can convert the product ciphertext to an equivalent one under the original key

Key switching is expensive:

- ☞ conversions between coefficient and DoubleCRT representations

Multiplication and Key Switching

Multiplying two ciphertexts in DoubleCRT representation takes linear time

But . . . we get a ciphertext defined with respect to a different secret key

So . . . we include an encryption of this other key under the original key in the public parameters (called a “key switching matrix”)

Using this, we can convert the product ciphertext to an equivalent one under the original key

Key switching is expensive:

- ☞ conversions between coefficient and DoubleCRT representations

Multiplication and Key Switching

Multiplying two ciphertexts in DoubleCRT representation takes linear time

But ... we get a ciphertext defined with respect to a different secret key

So ... we include an encryption of this other key under the original key in the public parameters (called a “key switching matrix”)

Using this, we can convert the product ciphertext to an equivalent one under the original key

Key switching is expensive:

- ☞ conversions between coefficient and DoubleCRT representations

Multiplication and Key Switching

Multiplying two ciphertexts in DoubleCRT representation takes linear time

But . . . we get a ciphertext defined with respect to a different secret key

So . . . we include an encryption of this other key under the original key in the public parameters (called a “key switching matrix”)

Using this, we can convert the product ciphertext to an equivalent one under the original key

Key switching is expensive:

- ☞ conversions between coefficient and DoubleCRT representations

Multiplication and Key Switching

Multiplying two ciphertexts in DoubleCRT representation takes linear time

But . . . we get a ciphertext defined with respect to a different secret key

So . . . we include an encryption of this other key under the original key in the public parameters (called a “key switching matrix”)

Using this, we can convert the product ciphertext to an equivalent one under the original key

Key switching is expensive:

- ☞ conversions between coefficient and DoubleCRT representations

Plaintext space structure

Chinese Remainder Theorem:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \bigoplus_{i=1}^{h} \mathbb{Z}_p[X]/(f_i(X))$$

where $\Phi_n(X) = \prod_{i=1}^{h} f_i(X)$

Each f_i irreducible of degree $d =$ order of p mod n

So we have

$$R_p \cong (\mathrm{GF}(p^d))^h \qquad [\, dh = \phi(n) \,]$$

We can view plaintext space as $\mathrm{GF}(p^d)$, and we can work on vectors of h plaintext "slots" ***in parallel***

Reminiscent of **vectorized** or **SIMD** computation

Plaintext space structure

Chinese Remainder Theorem:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \bigoplus_{i=1}^{h} \mathbb{Z}_p[X]/(f_i(X))$$

where $\Phi_n(X) = \prod_{i=1}^{h} f_i(X)$

Each f_i irreducible of degree d = order of p mod n

So we have

$$R_p \cong (\mathrm{GF}(p^d))^h \qquad [dh = \phi(n)]$$

We can view plaintext space as $\mathrm{GF}(p^d)$, and we can work on vectors of h plaintext "slots" ***in parallel***

Reminiscent of **vectorized** or **SIMD** computation

Plaintext space structure

Chinese Remainder Theorem:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \bigoplus_{i=1}^{h} \mathbb{Z}_p[X]/(f_i(X))$$

where $\Phi_n(X) = \prod_{i=1}^{h} f_i(X)$

Each f_i irreducible of degree $d =$ order of p mod n

So we have

$$R_p \cong (\mathrm{GF}(p^d))^h \qquad [dh = \phi(n)]$$

We can view plaintext space as $\mathrm{GF}(p^d)$, and we can work on vectors of h plaintext "slots" ***in parallel***

Reminiscent of **vectorized** or **SIMD** computation

Plaintext space structure

Chinese Remainder Theorem:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \bigoplus_{i=1}^{h} \mathbb{Z}_p[X]/(f_i(X))$$

where $\Phi_n(X) = \prod_{i=1}^{h} f_i(X)$

Each f_i irreducible of degree $d =$ order of p mod n

So we have

$$R_p \cong (\mathrm{GF}(p^d))^h \qquad [dh = \phi(n)]$$

We can view plaintext space as $\mathrm{GF}(p^d)$, and we can work on vectors of h plaintext “slots” ***in parallel***

Reminiscent of **vectorized** or **SIMD** computation

Plaintext space structure

Chinese Remainder Theorem:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \bigoplus_{i=1}^{h} \mathbb{Z}_p[X]/(f_i(X))$$

where $\Phi_n(X) = \prod_{i=1}^{h} f_i(X)$

Each f_i irreducible of degree $d =$ order of p mod n

So we have

$$R_p \cong (\mathrm{GF}(p^d))^h \qquad\qquad [\, dh = \phi(n)]$$

We can view plaintext space as $\mathrm{GF}(p^d)$, and we can work on vectors of h plaintext "slots" ***in parallel***

Reminiscent of **vectorized** or **SIMD** computation

Some useful automorphisms

Each $j \in \mathbb{Z}_n^*$ defines an automorphism on R_p that sends $X \mapsto X^j$

Homomorphic evaluation: just apply $X \mapsto X^j$ directly to R_q

- ☞ easy . . . but it requires "key switching"

This gives us a set of "rotations" that allow us to move data between "slots"

Some useful automorphisms

Each $j \in \mathbb{Z}_n^*$ defines an automorphism on R_p that sends $X \mapsto X^j$

Homomorphic evaluation: just apply $X \mapsto X^j$ directly to R_q

☞ easy . . . but it requires "key switching"

This gives us a set of "rotations" that allow us to move data between "slots"

Some useful automorphisms

Each $j \in \mathbb{Z}_n^*$ defines an automorphism on R_p that sends $X \mapsto X^j$

Homomorphic evaluation: just apply $X \mapsto X^j$ directly to R_q

- ☞ easy . . . but it requires “key switching”

This gives us a set of “rotations” that allow us to move data between “slots”

A simplified (but not very typical) setting:

$$p \equiv 1 \pmod{n} \implies \Phi_n(X) \text{ splits completely over } \mathbb{Z}_p$$

We have:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \mathrm{GF}(p)^h \quad \text{where } h = \phi(n)$$

via the isomorphism

$$[f(X) \bmod \Phi_n(X)] \mapsto [f(\omega^i)]_{i \in \mathbb{Z}_n^*}$$

where $\omega \in \mathbb{Z}_p^*$ is a primitive nth root of unity

The automorphism $X \mapsto X^j$ sends

$$[f(\omega^i)]_{i \in \mathbb{Z}_n^*} \mapsto [f(\omega^{ij})]_{i \in \mathbb{Z}_n^*}$$

So the data in slot ij moves to slot i

A simplified (but not very typical) setting:

$$p \equiv 1 \pmod{n} \implies \Phi_n(X) \text{ splits completely over } \mathbb{Z}_p$$

We have:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \mathrm{GF}(p)^h \quad \text{where } h = \phi(n)$$

via the isomorphism

$$[f(X) \bmod \Phi_n(X)] \mapsto [f(\omega^i)]_{i \in \mathbb{Z}_n^*}$$

where $\omega \in \mathbb{Z}_p^*$ is a primitive nth root of unity

The automorphism $X \mapsto X^j$ sends

$$[f(\omega^i)]_{i \in \mathbb{Z}_n^*} \mapsto [f(\omega^{ij})]_{i \in \mathbb{Z}_n^*}$$

So the data in slot ij moves to slot i

A simplified (but not very typical) setting:

$$p \equiv 1 \pmod{n} \implies \Phi_n(X) \text{ splits completely over } \mathbb{Z}_p$$

We have:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \mathrm{GF}(p)^h \quad \text{where } h = \phi(n)$$

via the isomorphism

$$[f(X) \bmod \Phi_n(X)] \mapsto [f(\omega^i)]_{i \in \mathbb{Z}_n^*}$$

where $\omega \in \mathbb{Z}_p^*$ is a primitive nth root of unity

The automorphism $X \mapsto X^j$ sends

$$[f(\omega^i)]_{i \in \mathbb{Z}_n^*} \mapsto [f(\omega^{ij})]_{i \in \mathbb{Z}_n^*}$$

So the data in slot ij moves to slot i

A simplified (but not very typical) setting:

$$p \equiv 1 \pmod{n} \implies \Phi_n(X) \text{ splits completely over } \mathbb{Z}_p$$

We have:

$$R_p = \mathbb{Z}_p[X]/(\Phi_n(X)) \cong \mathrm{GF}(p)^h \quad \text{where } h = \phi(n)$$

via the isomorphism

$$[f(X) \bmod \Phi_n(X)] \mapsto [f(\omega^i)]_{i \in \mathbb{Z}_n^*}$$

where $\omega \in \mathbb{Z}_p^*$ is a primitive nth root of unity

The automorphism $X \mapsto X^j$ sends

$$[f(\omega^i)]_{i \in \mathbb{Z}_n^*} \mapsto [f(\omega^{ij})]_{i \in \mathbb{Z}_n^*}$$

So the data in slot ij moves to slot i

General case: the available rotations are determined by the group structure of $\mathbb{Z}_n^*/\langle p\rangle$

Structure theorem:

$$\mathbb{Z}_n^*/\langle p\rangle \cong \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}, \quad \text{where } n_{i+1} \mid n_i \text{ for each } i$$

Example: suppose $\mathbb{Z}_n^*/\langle p\rangle \cong \mathbb{Z}_3 \times \mathbb{Z}_3$

We have 9 slots arranged in a 3×3 array:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

We can rotate all the rows (simultaneously) by any amount, or all the columns simultaneously by any amount

More generally: we have a k-dimensional hypercube, with rotations in each dimension

General case: the available rotations are determined by the group structure of $\mathbb{Z}_n^*/\langle p\rangle$

Structure theorem:

$$\mathbb{Z}_n^*/\langle p\rangle \cong \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}, \quad \text{where } n_{i+1} \mid n_i \text{ for each } i$$

Example: suppose $\mathbb{Z}_n^*/\langle p\rangle \cong \mathbb{Z}_3 \times \mathbb{Z}_3$

We have 9 slots arranged in a 3×3 array:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

We can rotate all the rows (simultaneously) by any amount, or all the columns simultaneously by any amount

More generally: we have a k-dimensional hypercube, with rotations in each dimension

General case: the available rotations are determined by the group structure of $\mathbb{Z}_n^*/\langle p \rangle$

Structure theorem:

$$\mathbb{Z}_n^*/\langle p \rangle \cong \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}, \quad \text{where } n_{i+1} \mid n_i \text{ for each } i$$

Example: suppose $\mathbb{Z}_n^*/\langle p \rangle \cong \mathbb{Z}_3 \times \mathbb{Z}_3$

We have 9 slots arranged in a 3×3 array:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

We can rotate all the rows (simultaneously) by any amount, or all the columns simultaneously by any amount

More generally: we have a k-dimensional hypercube, with rotations in each dimension

General case: the available rotations are determined by the group structure of $\mathbb{Z}_n^*/\langle p \rangle$

Structure theorem:

$$\mathbb{Z}_n^*/\langle p \rangle \cong \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}, \quad \text{where } n_{i+1} \mid n_i \text{ for each } i$$

Example: suppose $\mathbb{Z}_n^*/\langle p \rangle \cong \mathbb{Z}_3 \times \mathbb{Z}_3$

We have 9 slots arranged in a 3×3 array:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

We can rotate all the rows (simultaneously) by any amount, or all the columns simultaneously by any amount

More generally: we have a k-dimensional hypercube, with rotations in each dimension

General case: the available rotations are determined by the group structure of $\mathbb{Z}_n^*/\langle p\rangle$

Structure theorem:

$$\mathbb{Z}_n^*/\langle p\rangle \cong \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}, \quad \text{where } n_{i+1} \mid n_i \text{ for each } i$$

Example: suppose $\mathbb{Z}_n^*/\langle p\rangle \cong \mathbb{Z}_3 \times \mathbb{Z}_3$

We have 9 slots arranged in a 3×3 array:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

We can rotate all the rows (simultaneously) by any amount, or all the columns simultaneously by any amount

More generally: we have a k-dimensional hypercube, with rotations in each dimension

General case: the available rotations are determined by the group structure of $\mathbb{Z}_n^*/\langle p \rangle$

Structure theorem:

$$\mathbb{Z}_n^*/\langle p \rangle \cong \mathbb{Z}_{n_1} \times \cdots \times \mathbb{Z}_{n_k}, \quad \text{where } n_{i+1} \mid n_i \text{ for each } i$$

Example: suppose $\mathbb{Z}_n^*/\langle p \rangle \cong \mathbb{Z}_3 \times \mathbb{Z}_3$

We have 9 slots arranged in a 3×3 array:

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

We can rotate all the rows (simultaneously) by any amount, or all the columns simultaneously by any amount

More generally: we have a k-dimensional hypercube, with rotations in each dimension

The main topic: computing $\mathrm{GF}(\mathrm{p}^{\mathrm{d}})$-linear maps

Input: an encrypted vector v with h slots in $\mathrm{GF}(p^d)$

Output: $L(v)$, for some fixed, public $\mathrm{GF}(p^d)$-linear map L

- Equivalently: Mv, where $M \in \mathrm{GF}(p^d)^{h \times h}$

The main topic: computing $\mathrm{GF}(\mathrm{p}^{\mathrm{d}})$-linear maps

Input: an encrypted vector v with h slots in $\mathrm{GF}(p^d)$

Output: $L(v)$, for some fixed, public $\mathrm{GF}(p^d)$-linear map L

- Equivalently: Mv, where $M \in \mathrm{GF}(p^d)^{h \times h}$

An obvious approach: Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} v_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix} v_2 + \begin{bmatrix} a_{13} \\ a_{23} \\ a_{33} \end{bmatrix} v_3$$

$$= \begin{bmatrix} a_{11} v_1 \\ a_{21} v_1 \\ a_{31} v_1 \end{bmatrix} + \begin{bmatrix} a_{12} v_2 \\ a_{22} v_2 \\ a_{32} v_2 \end{bmatrix} + \begin{bmatrix} a_{13} v_3 \\ a_{23} v_3 \\ a_{33} v_3 \end{bmatrix}$$

Requires a “multibroadcast”:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \mapsto \left(\begin{bmatrix} v_1 \\ v_1 \\ v_1 \end{bmatrix}, \begin{bmatrix} v_2 \\ v_2 \\ v_2 \end{bmatrix}, \begin{bmatrix} v_3 \\ v_3 \\ v_3 \end{bmatrix} \right)$$

- *can be done using $O(h)$ rotations/mul-by-const*
- *overkill*

An obvious approach: Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} v_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix} v_2 + \begin{bmatrix} a_{13} \\ a_{23} \\ a_{33} \end{bmatrix} v_3$$

$$= \begin{bmatrix} a_{11}v_1 \\ a_{21}v_1 \\ a_{31}v_1 \end{bmatrix} + \begin{bmatrix} a_{12}v_2 \\ a_{22}v_2 \\ a_{32}v_2 \end{bmatrix} + \begin{bmatrix} a_{13}v_3 \\ a_{23}v_3 \\ a_{33}v_3 \end{bmatrix}$$

Requires a "multibroadcast":

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \mapsto \left(\begin{bmatrix} v_1 \\ v_1 \\ v_1 \end{bmatrix}, \begin{bmatrix} v_2 \\ v_2 \\ v_2 \end{bmatrix}, \begin{bmatrix} v_3 \\ v_3 \\ v_3 \end{bmatrix} \right)$$

- *can be done using $O(h)$ rotations/mul-by-const*
- *overkill*

An obvious approach: Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \end{bmatrix} v_1 + \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \end{bmatrix} v_2 + \begin{bmatrix} a_{13} \\ a_{23} \\ a_{33} \end{bmatrix} v_3$$

$$= \begin{bmatrix} a_{11} v_1 \\ a_{21} v_1 \\ a_{31} v_1 \end{bmatrix} + \begin{bmatrix} a_{12} v_2 \\ a_{22} v_2 \\ a_{32} v_2 \end{bmatrix} + \begin{bmatrix} a_{13} v_3 \\ a_{23} v_3 \\ a_{33} v_3 \end{bmatrix}$$

Requires a “multibroadcast”:

$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} \mapsto \left(\begin{bmatrix} v_1 \\ v_1 \\ v_1 \end{bmatrix}, \begin{bmatrix} v_2 \\ v_2 \\ v_2 \end{bmatrix}, \begin{bmatrix} v_3 \\ v_3 \\ v_3 \end{bmatrix} \right)$$

- *can be done using $O(h)$ rotations/mul-by-const*
- *overkill*

A better idea: Cannon [1969], Bernstein [2008]

Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}v_1 \\ a_{22}v_2 \\ a_{33}v_3 \end{bmatrix} + \begin{bmatrix} a_{12}v_2 \\ a_{23}v_3 \\ a_{31}v_1 \end{bmatrix} + \begin{bmatrix} a_{13}v_3 \\ a_{21}v_1 \\ a_{32}v_2 \end{bmatrix}$$

The constants

$$C_0 = (a_{11}, a_{22}, a_{33}), C_1 = (a_{12}, a_{23}, a_{31}), C_2 = (a_{13}, a_{21}, a_{32})$$

constructed using CRT and converted to DoubleCRT

. . . as a pre-computation

Total cost: h rotations (expensive), h mul-by-const (cheap)

A better idea: Cannon [1969], Bernstein [2008]

Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}v_1 \\ a_{22}v_2 \\ a_{33}v_3 \end{bmatrix} + \begin{bmatrix} a_{12}v_2 \\ a_{23}v_3 \\ a_{31}v_1 \end{bmatrix} + \begin{bmatrix} a_{13}v_3 \\ a_{21}v_1 \\ a_{32}v_2 \end{bmatrix}$$

The constants

$$C_0 = (a_{11}, a_{22}, a_{33}), C_1 = (a_{12}, a_{23}, a_{31}), C_2 = (a_{13}, a_{21}, a_{32})$$

constructed using CRT and converted to DoubleCRT

. . . as a pre-computation

Total cost: h rotations (expensive), h mul-by-const (cheap)

A better idea: Cannon [1969], Bernstein [2008]

Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} a_{11}v_1 \\ a_{22}v_2 \\ a_{33}v_3 \end{bmatrix} + \begin{bmatrix} a_{12}v_2 \\ a_{23}v_3 \\ a_{31}v_1 \end{bmatrix} + \begin{bmatrix} a_{13}v_3 \\ a_{21}v_1 \\ a_{32}v_2 \end{bmatrix}$$

The constants

$$C_0 = (a_{11}, a_{22}, a_{33}), C_1 = (a_{12}, a_{23}, a_{31}), C_2 = (a_{13}, a_{21}, a_{32})$$

constructed using CRT and converted to DoubleCRT

... as a pre-computation

Total cost: h rotations (expensive), h mul-by-const (cheap)

A better idea: Cannon [1969], Bernstein [2008]

Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11} v_1 \\ a_{22} v_2 \\ a_{33} v_3 \end{bmatrix} + \begin{bmatrix} a_{12} v_2 \\ a_{23} v_3 \\ a_{31} v_1 \end{bmatrix} + \begin{bmatrix} a_{13} v_3 \\ a_{21} v_1 \\ a_{32} v_2 \end{bmatrix}$$

The constants

$$C_0 = (a_{11}, a_{22}, a_{33}), C_1 = (a_{12}, a_{23}, a_{31}), C_2 = (a_{13}, a_{21}, a_{32})$$

constructed using CRT and converted to DoubleCRT

. . . as a pre-computation

Total cost: h rotations (expensive), h mul-by-const (cheap)

A better idea: Cannon [1969], Bernstein [2008]

Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}v_1 \\ a_{22}v_2 \\ a_{33}v_3 \end{bmatrix} + \begin{bmatrix} a_{12}v_2 \\ a_{23}v_3 \\ a_{31}v_1 \end{bmatrix} + \begin{bmatrix} a_{13}v_3 \\ a_{21}v_1 \\ a_{32}v_2 \end{bmatrix}$$

The constants

$$C_0 = (a_{11}, a_{22}, a_{33}), C_1 = (a_{12}, a_{23}, a_{31}), C_2 = (a_{13}, a_{21}, a_{32})$$

constructed using CRT and converted to DoubleCRT

...as a pre-computation

Total cost: h rotations (expensive), h mul-by-const (cheap)

A better idea: Cannon [1969], Bernstein [2008]

Example: $h = 3$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}v_1 \\ a_{22}v_2 \\ a_{33}v_3 \end{bmatrix} + \begin{bmatrix} a_{12}v_2 \\ a_{23}v_3 \\ a_{31}v_1 \end{bmatrix} + \begin{bmatrix} a_{13}v_3 \\ a_{21}v_1 \\ a_{32}v_2 \end{bmatrix}$$

The constants

$$C_0 = (a_{11}, a_{22}, a_{33}), C_1 = (a_{12}, a_{23}, a_{31}), C_2 = (a_{13}, a_{21}, a_{32})$$

constructed using CRT and converted to DoubleCRT

...as a pre-computation

Total cost: h rotations (expensive), h mul-by-const (cheap)

An even better idea: baby-step/giant-step

Let $\rho^i(v)$ denote rotation of v by i positions

We want to compute $L(v) = \sum_{i \in [h]} C_i \cdot \rho^i(v)$ for constants $C_0, \ldots, C_{h-1}$

Observation: ρ is an automorphism on the plaintext space R_p

$$\begin{aligned}
L(v) &= \sum_{i \in [h]} C_i \cdot \rho^i(v) \\
&= \sum_{j \in [f]} \sum_{k \in [g]} C_{j+fk} \cdot \rho^{j+fk}(v), \quad \text{where} \quad f, g \approx \sqrt{h} \\
&= \sum_{k \in [g]} \rho^{fk} \Big[\sum_{j \in [f]} C'_{j+fk} \cdot \rho^j(v) \Big], \quad \text{where} \quad C'_{j+fk} := \rho^{-fk}(C_{j+fk})
\end{aligned}$$

An even better idea: baby-step/giant-step

Let $\rho^i(v)$ denote rotation of v by i positions

We want to compute $L(v) = \sum_{i \in [h]} C_i \cdot \rho^i(v)$ for constants $C_0, \ldots, C_{h-1}$

Observation: ρ is an automorphism on the plaintext space R_p

$$
\begin{aligned}
L(v) &= \sum_{i \in [h]} C_i \cdot \rho^i(v) \\
&= \sum_{j \in [f]} \sum_{k \in [g]} C_{j+fk} \cdot \rho^{j+fk}(v), \quad \text{where} \quad f, g \approx \sqrt{h} \\
&= \sum_{k \in [g]} \rho^{fk} \Big[\sum_{j \in [f]} C'_{j+fk} \cdot \rho^{j}(v) \Big], \quad \text{where} \quad C'_{j+fk} := \rho^{-fk}(C_{j+fk})
\end{aligned}
$$

An even better idea: baby-step/giant-step

Let $\rho^i(v)$ denote rotation of v by i positions

We want to compute $L(v) = \sum_{i \in [h]} C_i \cdot \rho^i(v)$ for constants $C_0, \ldots, C_{h-1}$

Observation: ρ is an automorphism on the plaintext space R_p

$$\begin{aligned} L(v) &= \sum_{i \in [h]} C_i \cdot \rho^i(v) \\ &= \sum_{j \in [f]} \sum_{k \in [g]} C_{j+fk} \cdot \rho^{j+fk}(v), \quad \text{where} \quad f, g \approx \sqrt{h} \\ &= \sum_{k \in [g]} \rho^{fk} \Big[\sum_{j \in [f]} C'_{j+fk} \cdot \rho^j(v) \Big], \quad \text{where} \quad C'_{j+fk} := \rho^{-fk}(C_{j+fk}) \end{aligned}$$

An even better idea: baby-step/giant-step

Let $\rho^i(v)$ denote rotation of v by i positions

We want to compute $L(v) = \sum_{i \in [h]} C_i \cdot \rho^i(v)$ for constants $C_0, \ldots, C_{h-1}$

Observation: ρ is an automorphism on the plaintext space R_p

$$L(v) = \sum_{i \in [h]} C_i \cdot \rho^i(v)$$

$$= \sum_{j \in [f]} \sum_{k \in [g]} C_{j+fk} \cdot \rho^{j+fk}(v), \quad \text{where} \quad f, g \approx \sqrt{h}$$

$$= \sum_{k \in [g]} \rho^{fk} \Big[\sum_{j \in [f]} C'_{j+fk} \cdot \rho^j(v) \Big], \quad \text{where} \quad C'_{j+fk} := \rho^{-fk}(C_{j+fk})$$

An even better idea: baby-step/giant-step

Let $\rho^i(v)$ denote rotation of v by i positions

We want to compute $L(v) = \sum_{i \in [h]} C_i \cdot \rho^i(v)$ for constants $C_0, \ldots, C_{h-1}$

Observation: ρ is an automorphism on the plaintext space R_p

$$
\begin{aligned}
L(v) &= \sum_{i \in [h]} C_i \cdot \rho^i(v) \\
&= \sum_{j \in [f]} \sum_{k \in [g]} C_{j+fk} \cdot \rho^{j+fk}(v), \quad \text{where} \quad f, g \approx \sqrt{h} \\
&= \sum_{k \in [g]} \rho^{fk} \Big[\sum_{j \in [f]} C'_{j+fk} \cdot \rho^{j}(v) \Big], \quad \text{where} \quad C'_{j+fk} := \rho^{-fk}(C_{j+fk})
\end{aligned}
$$

An even better idea: baby-step/giant-step

Let $\rho^i(v)$ denote rotation of v by i positions

We want to compute $L(v) = \sum_{i \in [h]} C_i \cdot \rho^i(v)$ for constants $C_0, \ldots, C_{h-1}$

Observation: ρ is an automorphism on the plaintext space R_p

$$
\begin{aligned}
L(v) &= \sum_{i \in [h]} C_i \cdot \rho^i(v) \\
&= \sum_{j \in [f]} \sum_{k \in [g]} C_{j+fk} \cdot \rho^{j+fk}(v), \quad \text{where} \quad f, g \approx \sqrt{h} \\
&= \sum_{k \in [g]} \rho^{fk} \Big[\sum_{j \in [f]} C'_{j+fk} \cdot \rho^j(v) \Big], \quad \text{where} \quad C'_{j+fk} := \rho^{-fk}(C_{j+fk})
\end{aligned}
$$

Baby-step/giant-step algorithm:

1. for each $j \in [f]$: compute $v_j \leftarrow \rho^j(v)$ // baby steps
2. for each $k \in [g]$: compute $w_k \leftarrow \sum_{j\in[f]} C'_{j+fk} \cdot v_j$
3. compute $w \leftarrow \sum_{k\in[g]} \rho^{fk}(w_k)$ // giant steps

Cost:

- Step 1: $\approx \sqrt{h}$ rotations
- Step 2: $\approx h$ mul-by-const
- Step 3: $\approx \sqrt{h}$ rotations

Baby-step/giant-step algorithm:

1. for each $j \in [f]$: compute $v_j \leftarrow \rho^j(v)$ // baby steps
2. for each $k \in [g]$: compute $w_k \leftarrow \sum_{j\in[f]} C'_{j+fk} \cdot v_j$
3. compute $w \leftarrow \sum_{k\in[g]} \rho^{fk}(w_k)$ // giant steps

Cost:

- Step 1: $\approx \sqrt{h}$ rotations
- Step 2: $\approx h$ mul-by-const
- Step 3: $\approx \sqrt{h}$ rotations

Baby-step/giant-step algorithm:

1. for each $j \in [f]$: compute $v_j \leftarrow \rho^j(v)$ // baby steps
2. for each $k \in [g]$: compute $w_k \leftarrow \sum_{j\in[f]} C'_{j+fk} \cdot v_j$
3. compute $w \leftarrow \sum_{k\in[g]} \rho^{fk}(w_k)$ // giant steps

Cost:

- Step 1: $\approx \sqrt{h}$ rotations
- Step 2: $\approx h$ mul-by-const
- Step 3: $\approx \sqrt{h}$ rotations

Baby-step/giant-step algorithm:

1. for each $j \in [f]$: compute $v_j \leftarrow \rho^j(v)$ // baby steps
2. for each $k \in [g]$: compute $w_k \leftarrow \sum_{j \in [f]} C'_{j+fk} \cdot v_j$
3. compute $w \leftarrow \sum_{k \in [g]} \rho^{fk}(w_k)$ // giant steps

Cost:

- Step 1: $\approx \sqrt{h}$ rotations
- Step 2: $\approx h$ mul-by-const
- Step 3: $\approx \sqrt{h}$ rotations

An even more better idea(?)

or . . . "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c'_j \leftarrow \rho^i(c_j)$ for $j = 0, 1$

B) Key Switching, part 1 – break into digits (expensive):
decompose c'_1 as $c'_1 = \sum_k d'_k R_k$, where the R_k's are integer constants and each "digit" d'_k has small norm

☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):
compute the ciphertext $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

An even more better idea(?)

or . . . "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c'_j \leftarrow \rho^i(c_j)$ for $j = 0,1$

B) Key Switching, part 1 – break into digits (expensive):
decompose c'_1 as $c'_1 = \sum_k d'_k R_k$, where the R_k's are integer constants and each "digit" d'_k has small norm

☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):
compute the ciphertext $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

An even more better idea(?)

or ... "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c'_j \leftarrow \rho^i(c_j)$ for $j = 0,1$

B) Key Switching, part 1 – break into digits (expensive):
decompose c'_1 as $c'_1 = \sum_k d'_k R_k$, where the R_k's are integer constants and each "digit" d'_k has small norm

☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):
compute the ciphertext $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

An even more better idea(?)

or ... "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c'_j \leftarrow \rho^i(c_j)$ for $j = 0, 1$

B) Key Switching, part 1 – break into digits (expensive):
decompose c'_1 as $c'_1 = \sum_k d'_k R_k$, where the R_k's are integer constants and each "digit" d'_k has small norm

☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):
compute the ciphertext $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

An even more better idea(?)

or . . . "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c'_j \leftarrow \rho^i(c_j)$ for $j = 0,1$

B) Key Switching, part 1 – break into digits (expensive):
decompose c'_1 as $c'_1 = \sum_k d'_k R_k$, where the R_k's are integer constants and each "digit" d'_k has small norm

☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):
compute the ciphertext $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

An even more better idea(?)

or . . . "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c_j' \leftarrow \rho^i(c_j)$ for $j = 0,1$

B) Key Switching, part 1 – break into digits (expensive):
decompose c_1' as $c_1' = \sum_k d_k' R_k$, where the R_k's are integer constants and each "digit" d_k' has small norm

☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):
compute the ciphertext $(c_0' + c_0'', c_1'')$, where $c_j'' = \sum_k d_k' A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

An even more better idea(?)

or . . . "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c'_j \leftarrow \rho^i(c_j)$ for $j = 0,1$

B) Key Switching, part 1 – break into digits (expensive):
decompose c'_1 as $c'_1 = \sum_k d'_k R_k$, where the R_k's are integer constants and each "digit" d'_k has small norm

- ☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):
compute the ciphertext $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

An even more better idea(?)

or . . . "if $2\sqrt{h}$ rotations are good, then a single rotation is better"

Anatomy of a homomorphic rotation

We want to apply a rotation ρ^i to an encrypted vector v

The ciphertext is a pair $(c_0, c_1) \in R_q^{2\times 1}$

A) Raw automorphism step (cheap): $c'_j \leftarrow \rho^i(c_j)$ for $j = 0, 1$

B) Key Switching, part 1 – break into digits (expensive):

decompose c'_1 as $c'_1 = \sum_k d'_k R_k$, where the R_k's are integer constants and each "digit" d'_k has small norm

- ☞ requires DoubleCRT/coefficient conversion

C) Key Switching, part 2 – apply key switching matrix (cheap):

compute the ciphertext $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$ and the A_{jk}'s are pre-computed DoubleCRT objects

The idea: re-factor this three step process

- Basically, just swap steps (A) and (B), using the fact that ρ^i is an automorphism that does not change the norm by very much

A') Key Switching, part 1 – break into digits (expensive): decompose the *original* c_1 as $c_1 = \sum_k d_k R_k$

B') Raw automorphism step (cheap): $c_0' \leftarrow \rho^i(c_0)$ and $d_k' \leftarrow \rho^i(d_k)$ for each k

C) Key Switching, part 2 – apply key switching matrix (cheap): *exactly* the same as above: compute $(c_0' + c_0'', c_1'')$, where $c_j'' = \sum_k d_k' A_{jk}$

Why is this better? . . . we can perform step (A') just once for many rotations ρ^i

The idea: re-factor this three step process

- Basically, just swap steps (A) and (B), using the fact that ρ^i is an automorphism that does not change the norm by very much

A') Key Switching, part 1 – break into digits (expensive): decompose the *original* c_1 as $c_1 = \sum_k d_k R_k$

B') Raw automorphism step (cheap): $c'_0 \leftarrow \rho^i(c_0)$ and $d'_k \leftarrow \rho^i(d_k)$ for each k

C) Key Switching, part 2 – apply key switching matrix (cheap): *exactly* the same as above: compute $(c'_0 + c''_0, c''_1)$, where $c''_j = \sum_k d'_k A_{jk}$

Why is this better? ... we can perform step (A') just once for many rotations ρ^i

The idea: re-factor this three step process

- Basically, just swap steps (A) and (B), using the fact that ρ^i is an automorphism that does not change the norm by very much

A') Key Switching, part 1 – break into digits (expensive): decompose the *original* c_1 as $c_1 = \sum_k d_k R_k$

B') Raw automorphism step (cheap): $c_0' \leftarrow \rho^i(c_0)$ and $d_k' \leftarrow \rho^i(d_k)$ for each k

C) Key Switching, part 2 – apply key switching matrix (cheap): *exactly* the same as above: compute $(c_0' + c_0'', c_1'')$, where $c_j'' = \sum_k d_k' A_{jk}$

Why is this better? ... we can perform step (A') just once for many rotations ρ^i

The idea: re-factor this three step process

- Basically, just swap steps (A) and (B), using the fact that ρ^i is an automorphism that does not change the norm by very much

A') Key Switching, part 1 – break into digits (expensive): decompose the *original* c_1 as $c_1 = \sum_k d_k R_k$

B') Raw automorphism step (cheap): $c_0' \leftarrow \rho^i(c_0)$ and $d_k' \leftarrow \rho^i(d_k)$ for each k

C) Key Switching, part 2 – apply key switching matrix (cheap): *exactly* the same as above: compute $(c_0' + c_0'', c_1'')$, where $c_j'' = \sum_k d_k' A_{jk}$

Why is this better? . . . we can perform step (A') just once for many rotations ρ^i

The idea: re-factor this three step process

- Basically, just swap steps (A) and (B), using the fact that ρ^i is an automorphism that does not change the norm by very much

A') Key Switching, part 1 – break into digits (expensive): decompose the *original* c_1 as $c_1 = \sum_k d_k R_k$

B') Raw automorphism step (cheap): $c_0' \leftarrow \rho^i(c_0)$ and $d_k' \leftarrow \rho^i(d_k)$ for each k

C) Key Switching, part 2 – apply key switching matrix (cheap): *exactly* the same as above: compute $(c_0' + c_0'', c_1'')$, where $c_j'' = \sum_k d_k' A_{jk}$

Why is this better? . . . we can perform step (A') just once for many rotations ρ^i

The idea: re-factor this three step process

- Basically, just swap steps (A) and (B), using the fact that ρ^i is an automorphism that does not change the norm by very much

A') Key Switching, part 1 – break into digits (expensive): decompose the *original* c_1 as $c_1 = \sum_k d_k R_k$

B') Raw automorphism step (cheap): $c_0' \leftarrow \rho^i(c_0)$ and $d_k' \leftarrow \rho^i(d_k)$ for each k

C) Key Switching, part 2 – apply key switching matrix (cheap): *exactly* the same as above: compute $(c_0' + c_0'', c_1'')$, where $c_j'' = \sum_k d_k' A_{jk}$

Why is this better? ... we can perform step (A') just once for many rotations ρ^i

We call this idea “hoisting” (optimizing compilers are said to “hoist” invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand* . . . may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea "hoisting" (optimizing compilers are said to "hoist" invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand* . . . may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea “hoisting” (optimizing compilers are said to “hoist” invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand* . . . may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea "hoisting" (optimizing compilers are said to "hoist" invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand . . .* may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea “hoisting” (optimizing compilers are said to “hoist” invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand . . .* may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea "hoisting" (optimizing compilers are said to "hoist" invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand . . .* may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea “hoisting” (optimizing compilers are said to “hoist” invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand . . .* may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea "hoisting" (optimizing compilers are said to "hoist" invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand . . .* may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad$ ($2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

We call this idea “hoisting” (optimizing compilers are said to “hoist” invariant computations out of a loop)

So . . . given an encryption of v we can compute an encryption of $\rho^i(v)$ for $i \in [h]$ with just one expensive step and h cheap steps

Application to matrix multiplication:

- on the one hand . . . faster than the basic method (which takes h rotations)
- *on the other hand* . . . may be slower than the BS/GS method for large h
- ***but on the other hand*** . . . we can combine hoisting and BS/GS
  - baby steps: for each $j \in [f]$ compute $v_j \leftarrow \rho^j(v)$
  - hoist out these rotations
  - save a factor of 2 $\quad (2\sqrt{h} \longrightarrow \sqrt{h}$ rotations)

See paper for more details and other improvements:

- More efficient handling of “problematic” dimensions in the hypercube
- Saving space: drastic reduction in the number of “key switching matrices” without too much loss in speed

Questions?

See paper for more details and other improvements:

- ➪ More efficient handling of “problematic” dimensions in the hypercube
- ➪ Saving space: drastic reduction in the number of “key switching matrices” without too much loss in speed

Questions?

See paper for more details and other improvements:

- ➪ More efficient handling of “problematic” dimensions in the hypercube
- ➪ Saving space: drastic reduction in the number of “key switching matrices” without too much loss in speed

Questions?

See paper for more details and other improvements:

- ➪ More efficient handling of “problematic” dimensions in the hypercube
- ➪ Saving space: drastic reduction in the number of “key switching matrices” without too much loss in speed

Questions?