Fully Homomorphic Encryption over the Integers with Shorter Public Keys

Jean-Sébastien Coron, Avradip Mandal,
David Naccache and Mehdi Tibouchi

University of Luxembourg & ENS

CRYPTO, 2011-08-17

Outline

Fully homomorphic encryption

- Homomorphic encryption:
  - An encryption scheme is homomorphic when it supports operations on encrypted data.
- Multiplicatively homomorphic: RSA.
  - Given $c_1 = m_1^e \mod N, \quad c_2 = m_2^e \mod N$, we have $(c_1 \cdot c_2) = (m_1 \cdot m_2)^e \mod N$
- Additively homomorphic: Paillier.
  - Paillier: given $c_1 = g^{m_1} r^N \mod N^2, \quad c_2 = g^{m_2} s^N \mod N^2$, we have $c_1 \cdot c_2 = g^{m_1+m_2} \cdot (rs)^N \mod N^2$.
- Fully homomorphic: homomorphic for both addition and multiplication
  - Open problem until Gentry's breakthrough in 2009.

Fully homomorphic encryption

- Homomorphic encryption:
  - An encryption scheme is homomorphic when it supports operations on encrypted data.
- Multiplicatively homomorphic: RSA.
  - Given $c_1 = m_1^e \mod N, \quad c_2 = m_2^e \mod N$, we have $(c_1 \cdot c_2) = (m_1 \cdot m_2)^e \mod N$
- Additively homomorphic: Paillier.
  - Paillier: given $c_1 = g^{m_1} r^N \mod N^2, \quad c_2 = g^{m_2} s^N \mod N^2$, we have $c_1 \cdot c_2 = g^{m_1+m_2} \cdot (rs)^N \mod N^2$.
- Fully homomorphic: homomorphic for both addition and multiplication
  - Open problem until Gentry's breakthrough in 2009.

Fully homomorphic encryption

- Homomorphic encryption:
  - An encryption scheme is homomorphic when it supports operations on encrypted data.
- Multiplicatively homomorphic: RSA.
  - Given $c_1 = m_1^e \mod N$, $c_2 = m_2^e \mod N$, we have $(c_1 \cdot c_2) = (m_1 \cdot m_2)^e \mod N$
- Additively homomorphic: Paillier.
  - Paillier: given $c_1 = g^{m_1} r^N \mod N^2$, $c_2 = g^{m_2} s^N \mod N^2$, we have $c_1 \cdot c_2 = g^{m_1+m_2} \cdot (rs)^N \mod N^2$.
- Fully homomorphic: homomorphic for both addition and multiplication
  - Open problem until Gentry's breakthrough in 2009.

Fully homomorphic encryption

- Homomorphic encryption:
  - An encryption scheme is homomorphic when it supports operations on encrypted data.
- Multiplicatively homomorphic: RSA.
  - Given $c_1 = m_1^e \mod N$, $c_2 = m_2^e \mod N$, we have $(c_1 \cdot c_2) = (m_1 \cdot m_2)^e \mod N$
- Additively homomorphic: Paillier.
  - Paillier: given $c_1 = g^{m_1} r^N \mod N^2$, $c_2 = g^{m_2} s^N \mod N^2$, we have $c_1 \cdot c_2 = g^{m_1+m_2} \cdot (rs)^N \mod N^2$.
- Fully homomorphic: homomorphic for both addition and multiplication
  - Open problem until Gentry's breakthrough in 2009.

Fully homomorphic public-key encryption

- We restrict ourselves to public-key encryption of a single bit:
  - $0 \rightarrow 203ef6124 \ldots 23ab87_{16}$
  - $1 \rightarrow b327653c1 \ldots db3265_{16}$
- Fully homomorphic property
  - Given $E(b_0)$ and $E(b_1)$, one can compute $E(b_0 \oplus b_1)$ and $E(b_0 \cdot b_1)$ without knowing the private-key.
- Computing over a ring:
  - Given a circuit with xors and ands, and encrypted input bits, one can compute the output in encrypted form, without knowing the private key.
  - As a result: publicly compute any function on encrypted data (or at least any function that can be represented as a boolean circuit with polynomially many gates).

Fully homomorphic public-key encryption

- We restrict ourselves to public-key encryption of a single bit:
  - $0 \rightarrow 203ef6124 \ldots 23ab87_{16}$
  - $1 \rightarrow b327653c1 \ldots db3265_{16}$
- Fully homomorphic property
  - Given $E(b_0)$ and $E(b_1)$, one can compute $E(b_0 \oplus b_1)$ and $E(b_0 \cdot b_1)$ without knowing the private-key.
- Computing over a ring:
  - Given a circuit with xors and ands, and encrypted input bits, one can compute the output in encrypted form, without knowing the private key.
  - As a result: publicly compute any function on encrypted data (or at least any function that can be represented as a boolean circuit with polynomially many gates).

Fully homomorphic public-key encryption

- We restrict ourselves to public-key encryption of a single bit:
  - $0 \rightarrow 203\text{ef}6124\ldots23\text{ab}87_{16}$
  - $1 \rightarrow \text{b}327653\text{c}1\ldots\text{db}3265_{16}$
- Fully homomorphic property
  - Given $E(b_0)$ and $E(b_1)$, one can compute $E(b_0 \oplus b_1)$ and $E(b_0 \cdot b_1)$ without knowing the private-key.
- Computing over a ring:
  - Given a circuit with xors and ands, and encrypted input bits, one can compute the output in encrypted form, without knowing the private key.
  - As a result: publicly compute any function on encrypted data (or at least any function that can be represented as a boolean circuit with polynomially many gates).

What fully homomorphic encryption brings you

- You have a software that given the revenue, past income, headcount, etc., of a company can predict its future stock price.
  - I want to know the future stock price of my company, but I don't want to disclose confidential information.
  - And you don't want to give me your software containing secret formulas.
- Using homomorphic encryption:
  - I encrypt all the inputs using fully homomorphic encryption and send them to you in encrypted form.
  - You process all my inputs, viewing your software as a circuit.
  - You send me the result, still encrypted.
  - I decrypt the result and get the predicted stock price.
  - You didn't learn any information about my company.
- More generally:
  - Cool buzzwords like secure cloud computing.
  - Cool mathematical challenges.

What fully homomorphic encryption brings you

- You have a software that given the revenue, past income, headcount, etc., of a company can predict its future stock price.
  - I want to know the future stock price of my company, but I don't want to disclose confidential information.
  - And you don't want to give me your software containing secret formulas.
- Using homomorphic encryption:
  - I encrypt all the inputs using fully homomorphic encryption and send them to you in encrypted form.
  - You process all my inputs, viewing your software as a circuit.
  - You send me the result, still encrypted.
  - I decrypt the result and get the predicted stock price.
  - You didn't learn any information about my company.
- More generally:
  - Cool buzzwords like secure cloud computing.
  - Cool mathematical challenges.

What fully homomorphic encryption brings you

- You have a software that given the revenue, past income, headcount, etc., of a company can predict its future stock price.
  - I want to know the future stock price of my company, but I don't want to disclose confidential information.
  - And you don't want to give me your software containing secret formulas.
- Using homomorphic encryption:
  - I encrypt all the inputs using fully homomorphic encryption and send them to you in encrypted form.
  - You process all my inputs, viewing your software as a circuit.
  - You send me the result, still encrypted.
  - I decrypt the result and get the predicted stock price.
  - You didn't learn any information about my company.
- More generally:
  - Cool buzzwords like secure cloud computing.
  - Cool mathematical challenges.

Outline

Theory and practice

- Not many FHE schemes have been proposed yet:
  - Breakthrough scheme of Gentry (STOC 2009).
  - Conceptually simpler scheme of van Dijk, Gentry, Halevi and Vaikuntanathan (DGHV) over the integers (Eurocrypt 2010).
  - And that's about it for now (but see the next talk!).
- ...and they are important theoretical constructs, but far from usable in practice.
  - For DGHV: PK size around 2^{60} bits.
  - For Gentry's scheme: hard to suggest parameters at all.
- Ongoing effort to get closer to practicality:
  - For Gentry's scheme: improvement by Smart and Vercauteren (PKC 2010); implementation by Gentry and Halevi (Eurocrypt 2011). PK size: 2.3 GB. Ciphertext refresh: 30 minutes.
  - For DGHV: this work. PK size: 800 MB. Ciphertext refresh: 15 minutes.
  - (And very recently: exciting work by Gentry and others on FHE "without bootstrapping").

Theory and practice

- Not many FHE schemes have been proposed yet:
  - Breakthrough scheme of Gentry (STOC 2009).
  - Conceptually simpler scheme of van Dijk, Gentry, Halevi and Vaikuntanathan (DGHV) over the integers (Eurocrypt 2010).
  - And that's about it for now (but see the next talk!).
- . . . and they are important theoretical constructs, but far from usable in practice.
  - For DGHV: PK size around 2^{60} bits.
  - For Gentry's scheme: hard to suggest parameters at all.
- Ongoing effort to get closer to practicality:
  - For Gentry's scheme: improvement by Smart and Vercauteren (PKC 2010); implementation by Gentry and Halevi (Eurocrypt 2011). PK size: 2.3 GB. Ciphertext refresh: 30 minutes.
  - For DGHV: this work. PK size: 800 MB. Ciphertext refresh: 15 minutes.
  - (And very recently: exciting work by Gentry and others on FHE "without bootstrapping").

Theory and practice

- Not many FHE schemes have been proposed yet:
    - Breakthrough scheme of Gentry (STOC 2009).
    - Conceptually simpler scheme of van Dijk, Gentry, Halevi and Vaikuntanathan (DGHV) over the integers (Eurocrypt 2010).
    - And that's about it for now (but see the next talk!).
- . . . and they are important theoretical constructs, but far from usable in practice.
    - For DGHV: PK size around 2^{60} bits.
    - For Gentry's scheme: hard to suggest parameters at all.
- Ongoing effort to get closer to practicality:
    - For Gentry's scheme: improvement by Smart and Vercauteren (PKC 2010); implementation by Gentry and Halevi (Eurocrypt 2011). PK size: 2.3 GB. Ciphertext refresh: 30 minutes.
    - For DGHV: this work. PK size: 800 MB. Ciphertext refresh: 15 minutes.
    - (And very recently: exciting work by Gentry and others on FHE "without bootstrapping").

Outline

Gentry's technique

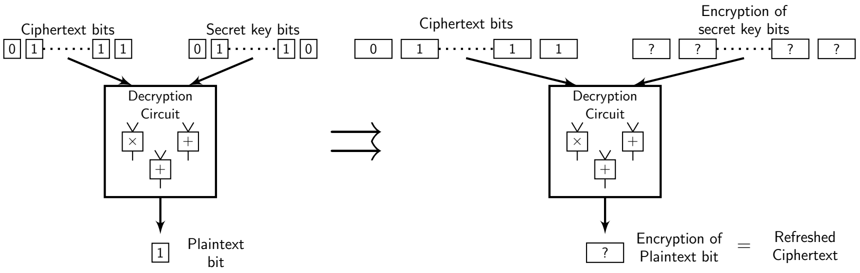
- To build a FHE scheme, start from the somewhat homomorphic scheme, that is:
    - Only a polynomial of small degree can be homomorphically applied on ciphertexts.
    - Otherwise the noise becomes too large and decryption becomes incorrect.
- Then, "squash" the decryption procedure:
    - express the decryption function as a low degree polynomial in the bits of the ciphertext c and the secret key sk (equivalently a boolean circuit of small depth).

Gentry's technique

- To build a FHE scheme, start from the somewhat homomorphic scheme, that is:
    - Only a polynomial of small degree can be homomorphically applied on ciphertexts.
    - Otherwise the noise becomes too large and decryption becomes incorrect.
- Then, "squash" the decryption procedure:
    - express the decryption function as a low degree polynomial in the bits of the ciphertext c and the secret key sk (equivalently a boolean circuit of small depth).

Ciphertext refresh

- Gentry's breakthrough idea: refresh the ciphertext using the decryption circuit homomorphically.
  - Evaluate the decryption polynomial not on the bits of the ciphertext *c* and the secret key *sk*, but homomorphically on the encryption of those bits.
  - Instead of recovering the bit plaintext *m*, one gets an encryption of this bit plaintext, *i.e.* yet another ciphertext for the same plaintext.



Ciphertext refresh

- Refreshed ciphertext:
  - If the degree of the decryption polynomial is small enough, the resulting noise in this new ciphertext can be smaller than in the original ciphertext
- Fully homomorphic encryption:
  - Given two refreshed ciphertexts one can apply again the homomorphic operation (either addition or multiplication), which was not necessarily possible on the original ciphertexts because of the noise threshold.
  - Using this "ciphertext refresh" procedure the number of homomorphic operations becomes unlimited and we get a fully homomorphic encryption scheme.

Ciphertext refresh

- Refreshed ciphertext:
    - If the degree of the decryption polynomial is small enough, the resulting noise in this new ciphertext can be smaller than in the original ciphertext
- Fully homomorphic encryption:
    - Given two refreshed ciphertexts one can apply again the homomorphic operation (either addition or multiplication), which was not necessarily possible on the original ciphertexts because of the noise threshold.
    - Using this "ciphertext refresh" procedure the number of homomorphic operations becomes unlimited and we get a fully homomorphic encryption scheme.

Outline

The DGHV scheme (simplified)

- Key generation:
  - Generate a set of τ public integers:

$$x_i = p \cdot q_i + r_i, \quad 1 \leq i \leq \tau$$

    and $x_0 = p \cdot q_0$, where p is a secret prime.
  - Size of p is η. Size of x_i is γ. Size of r_i is ρ.
- Encryption of a message $m \in \{0, 1\}$:
  - Choose a random subset $S \subset \{1, 2, \dots, \tau\}$ and a random integer r in $(-2^{\rho'}, 2^{\rho'})$, and output the ciphertext:

$$c = m + 2r + 2 \sum_{i \in S} x_i \bmod x_0$$

- Decryption:

$$c \equiv m + 2r + 2 \sum_{i \in S} r_i \pmod{p}$$

  - Output $m \leftarrow (c \mod p) \mod 2$

The DGHV scheme (simplified)

- Key generation:
  - Generate a set of τ public integers:

$$x_i = p \cdot q_i + r_i, \quad 1 \leq i \leq \tau$$

    and $x_0 = p \cdot q_0$, where p is a secret prime.
  - Size of p is η. Size of x_i is γ. Size of r_i is ρ.
- Encryption of a message $m \in \{0,1\}$:
  - Choose a random subset $S \subset \{1, 2, \ldots, \tau\}$ and a random integer r in $(-2^{\rho'}, 2^{\rho'})$, and output the ciphertext:

$$c = m + 2r + 2\sum_{i \in S} x_i \bmod x_0$$

- Decryption:

$$c \equiv m + 2r + 2\sum_{i \in S} r_i \pmod p$$

  - Output $m \leftarrow (c \mod p) \mod 2$

The DGHV scheme (simplified)

- Key generation:
  - Generate a set of τ public integers:

    $$x_i = p \cdot q_i + r_i, \quad 1 \leq i \leq \tau$$

    and $x_0 = p \cdot q_0$, where p is a secret prime.
  - Size of p is η. Size of x_i is γ. Size of r_i is ρ.
- Encryption of a message $m \in \{0, 1\}$:
  - Choose a random subset $S \subset \{1, 2, \ldots, \tau\}$ and a random integer r in $(-2^{\rho'}, 2^{\rho'})$, and output the ciphertext:

    $$c = m + 2r + 2 \sum_{i \in S} x_i \bmod x_0$$

- Decryption:

  $$c \equiv m + 2r + 2 \sum_{i \in S} r_i \pmod{p}$$

  - Output $m \leftarrow (c \mod p) \mod 2$

The DGHV scheme (contd.)

- Noise in ciphertext:
  - $c = m + 2 \cdot r' \mod p$ where $r' = r + \sum_{i \in S} r_i$
  - r' is the noise in the ciphertext.
  - It must remain $< p$ for correct decryption.
- Homomorphic addition: $c_3 \leftarrow c_1 + c_2 \mod x_0$
  - $c_1 + c_2 = m_1 + m_2 + 2(r'_1 + r'_2) \mod p$
  - Works if noise $r'_1 + r'_2$ still less than p.
- Homomorphic multiplication: $c_3 \leftarrow c_1 \cdot c_2 \mod x_0$
  - $c_1 \cdot c_2 = m_1 \cdot m_2 + 2(m_1 \cdot r'_2 + m_2 \cdot r'_1 + 2r'_1 \cdot r'_2) \mod p$
  - Works if noise $r'_1 \cdot r'_2$ remains less than p.
- Somewhat homomorphic scheme
  - Noise grows with every homomorphic addition or multiplication.
  - A limited number of homomorphic operations is supported.
  - This limits the degree of the polynomial that can be applied on ciphertexts.

The DGHV scheme (contd.)

- Noise in ciphertext:
  - $c = m + 2 \cdot r' \mod p$ where $r' = r + \sum_{i \in S} r_i$
  - r' is the noise in the ciphertext.
  - It must remain $< p$ for correct decryption.
- Homomorphic addition: $c_3 \leftarrow c_1 + c_2 \mod x_0$
  - $c_1 + c_2 = m_1 + m_2 + 2(r'_1 + r'_2) \mod p$
  - Works if noise $r'_1 + r'_2$ still less than p.
- Homomorphic multiplication: $c_3 \leftarrow c_1 \cdot c_2 \mod x_0$
  - $c_1 \cdot c_2 = m_1 \cdot m_2 + 2(m_1 \cdot r'_2 + m_2 \cdot r'_1 + 2r'_1 \cdot r'_2) \mod p$
  - Works if noise $r'_1 \cdot r'_2$ remains less than p.
- Somewhat homomorphic scheme
  - Noise grows with every homomorphic addition or multiplication.
  - A limited number of homomorphic operations is supported.
  - This limits the degree of the polynomial that can be applied on ciphertexts.

The DGHV scheme (contd.)

- Noise in ciphertext:
  - $c = m + 2 \cdot r' \mod p$ where $r' = r + \sum_{i \in S} r_i$
  - r' is the noise in the ciphertext.
  - It must remain $< p$ for correct decryption.
- Homomorphic addition: $c_3 \leftarrow c_1 + c_2 \mod x_0$
  - $c_1 + c_2 = m_1 + m_2 + 2(r'_1 + r'_2) \mod p$
  - Works if noise $r'_1 + r'_2$ still less than p.
- Homomorphic multiplication: $c_3 \leftarrow c_1 \cdot c_2 \mod x_0$
  - $c_1 \cdot c_2 = m_1 \cdot m_2 + 2(m_1 \cdot r'_2 + m_2 \cdot r'_1 + 2r'_1 \cdot r'_2) \mod p$
  - Works if noise $r'_1 \cdot r'_2$ remains less than p.
- Somewhat homomorphic scheme
  - Noise grows with every homomorphic addition or multiplication.
  - A limited number of homomorphic operations is supported.
  - This limits the degree of the polynomial that can be applied on ciphertexts.

The DGHV scheme (contd.)

- Noise in ciphertext:
  - $c = m + 2 \cdot r' \mod p$ where $r' = r + \sum_{i \in S} r_i$
  - r' is the noise in the ciphertext.
  - It must remain $< p$ for correct decryption.
- Homomorphic addition: $c_3 \leftarrow c_1 + c_2 \mod x_0$
  - $c_1 + c_2 = m_1 + m_2 + 2(r'_1 + r'_2) \mod p$
  - Works if noise $r'_1 + r'_2$ still less than p.
- Homomorphic multiplication: $c_3 \leftarrow c_1 \cdot c_2 \mod x_0$
  - $c_1 \cdot c_2 = m_1 \cdot m_2 + 2(m_1 \cdot r'_2 + m_2 \cdot r'_1 + 2r'_1 \cdot r'_2) \mod p$
  - Works if noise $r'_1 \cdot r'_2$ remains less than p.
- Somewhat homomorphic scheme
  - Noise grows with every homomorphic addition or multiplication.
  - A limited number of homomorphic operations is supported.
  - This limits the degree of the polynomial that can be applied on ciphertexts.

Parameter estimates

Security parameter λ.

- ρ : size of noise should be λ bits
- ρ' : size of secondary noise 2λ bits
- η : size of p, $\approx \lambda^2$ bits
- γ : size of x_i, $\approx \lambda^5$
- τ : number of elements (x_i's) in the public key, $\gamma + \lambda$

Public key size $\approx \gamma^2 \approx \lambda^{10}$ ($\approx 2^{62}$ bits for $\lambda = 72$ bits of security).

Outline

Reducing the public key size

- Encrypt using a quadratic form as opposed to a linear form in DGHV:
  - We start with a small numbers of x_i's
  - We combine them multiplicatively to generate the full public key.
- Start with β pairs $x_{i,0}, x_{j,1}$. One can define β^2 integers $x'_{i,j}$ with:

$$x'_{i,j} = x_{i,0} x_{j,1} \mod x_0, \quad 1 \le i, j \le \beta$$

- Encrypt using a linear combination of $x'_{i,j}$ with coefficients $b_{i,j} \in [0, 2^\alpha)$ as oppose to bits.

$$c = m + 2r + 2 \sum_{1 \le i,j \le \beta} b_{i,j} \cdot x_{i,0} \cdot x_{j_1} \mod x_0.$$

- We can take $\beta \approx \lambda^2$, hence PK size shrinks to $2\beta \cdot \gamma \approx \lambda^7$ bits!

Reducing the public key size

- Encrypt using a quadratic form as opposed to a linear form in DGHV:
  - We start with a small numbers of x_i's
  - We combine them multiplicatively to generate the full public key.
- Start with β pairs $x_{i,0}, x_{j,1}$. One can define β^2 integers $x'_{i,j}$ with:

$$x'_{i,j} = x_{i,0} x_{j,1} \mod x_0, \quad 1 \leq i, j \leq \beta$$

- Encrypt using a linear combination of $x'_{i,j}$ with coefficients $b_{i,j} \in [0, 2^\alpha)$ as oppose to bits.

$$c = m + 2r + 2 \sum_{1 \leq i,j \leq \beta} b_{i,j} \cdot x_{i,0} \cdot x_{j_1} \mod x_0.$$

- We can take $\beta \approx \lambda^2$, hence PK size shrinks to $2\beta \cdot \gamma \approx \lambda^7$ bits!

Reducing the public key size

- Encrypt using a quadratic form as opposed to a linear form in DGHV:
  - We start with a small numbers of x_i's
  - We combine them multiplicatively to generate the full public key.
- Start with β pairs $x_{i,0}, x_{j,1}$. One can define β^2 integers $x'_{i,j}$ with:

$$x'_{i,j} = x_{i,0} x_{j,1} \mod x_0, \quad 1 \leq i,j \leq \beta$$

- Encrypt using a linear combination of $x'_{i,j}$ with coefficients $b_{i,j} \in [0, 2^{\alpha})$ as oppose to bits.

$$c = m + 2r + 2 \sum_{1 \leq i,j \leq \beta} b_{i,j} \cdot x_{i,0} \cdot x_{j_1} \mod x_0.$$

- We can take $\beta \approx \lambda^2$, hence PK size shrinks to $2\beta \cdot \gamma \approx \lambda^7$ bits!

Reducing the public key size

- Encrypt using a quadratic form as opposed to a linear form in DGHV:
  - We start with a small numbers of x_i's
  - We combine them multiplicatively to generate the full public key.
- Start with β pairs $x_{i,0}, x_{j,1}$. One can define β^2 integers $x'_{i,j}$ with:

$$x'_{i,j} = x_{i,0} x_{j,1} \mod x_0, \quad 1 \leq i, j \leq \beta$$

- Encrypt using a linear combination of $x'_{i,j}$ with coefficients $b_{i,j} \in [0, 2^\alpha)$ as oppose to bits.

$$c = m + 2r + 2 \sum_{1 \leq i,j \leq \beta} b_{i,j} \cdot x_{i,0} \cdot x_{j_1} \mod x_0.$$

- We can take $\beta \approx \lambda^2$, hence PK size shrinks to $2\beta \cdot \gamma \approx \lambda^7$ bits!

Security of the new scheme

- The proof of semantic security is mostly the same as in the original DGHV paper. Main difficulty: showing that, for a ciphertext c, $\lfloor c/p \rceil$ is statistically close to uniform in $\mathbb{Z}_{q_0}$.
    - In DGHV: use the left-over hash lemma, and the fact that the function family
$$h(\vec{b}) = \sum_{i=1}^{\tau} b_i \cdot q_i$$
is pairwise independent.
    - In our scheme: use a slightly modified left-over hash lemma, and the fact that the function family
$$h'(\vec{b}) = \sum_{1 \le i,j \le \beta} b_{i,j} \cdot q_{i,0} \cdot q_{j,1}$$
is "close enough" to being pairwise independent.
- This fact uses point counting on hyperbolic quadrics in $\mathbb{Z}_{q_0}$, and is the main technical contribution of this paper.

Security of the new scheme

- The proof of semantic security is mostly the same as in the original DGHV paper. Main difficulty: showing that, for a ciphertext c, $\lfloor c/p \rceil$ is statistically close to uniform in $\mathbb{Z}_{q_0}$.
  - In DGHV: use the left-over hash lemma, and the fact that the function family
    $$h(\vec{b}) = \sum_{i=1}^{\tau} b_i \cdot q_i$$
    is pairwise independent.
  - In our scheme: use a slightly modified left-over hash lemma, and the fact that the function family
    $$h'(\vec{b}) = \sum_{1 \leq i,j \leq \beta} b_{i,j} \cdot q_{i,0} \cdot q_{j,1}$$
    is "close enough" to being pairwise independent.
- This fact uses point counting on hyperbolic quadrics in $\mathbb{Z}_{q_0}$, and is the main technical contribution of this paper.

Hardness assumption for semantic security

- Actual DGHV scheme: secure under the General Approximate Common Divisor (GACD) assumption.
    - Given polynomially many $p \cdot q_i + r_i$, finding p is hard.
- Our scheme: secure under the Partial Approximate Common Divisor (PACD) assumption.
    - Given $p \cdot q_0$ and polynomially many $p \cdot q_i + r_i$, finding p is hard.
- PACD is a stronger assumption, but Gentry and Halevi suggested that no better attack is known on PACD than on GACD (but more on that later).

Hardness assumption for semantic security

- Actual DGHV scheme: secure under the General Approximate Common Divisor (GACD) assumption.
  - Given polynomially many $p \cdot q_i + r_i$, finding p is hard.
- Our scheme: secure under the Partial Approximate Common Divisor (PACD) assumption.
  - Given $p \cdot q_0$ and polynomially many $p \cdot q_i + r_i$, finding p is hard.
- PACD is a stronger assumption, but Gentry and Halevi suggested that no better attack is known on PACD than on GACD (but more on that later).

Hardness assumption for semantic security

- Actual DGHV scheme: secure under the General Approximate Common Divisor (GACD) assumption.
  - Given polynomially many $p \cdot q_i + r_i$, finding p is hard.
- Our scheme: secure under the Partial Approximate Common Divisor (PACD) assumption.
  - Given $p \cdot q_0$ and polynomially many $p \cdot q_i + r_i$, finding p is hard.
- PACD is a stronger assumption, but Gentry and Halevi suggested that no better attack is known on PACD than on GACD (but more on that later).

Outline

The squashed scheme from DGHV

- The basic decryption $m \leftarrow (c \bmod p) \bmod 2$ cannot be directly expressed as a boolean circuit of low depth.
- But it can be written as:

$$m \leftarrow [c]_2 \oplus [\lfloor c \cdot (1/p) \rceil]_2$$

and this formula can be used for ciphertext refresh if $1/p$ can be put in a compact encrypted form in the public key.

- Idea (Gentry, DGHV): use secret sharing. Represent $1/p$ as a sparse subset sum:

$$\lfloor 2^{\kappa}/p \rceil = \sum_{i=1}^{\Theta} s_i \cdot u_i$$

with random κ-bit integers u_i, and $s_i \in \{0,1\}$. Publish the u_i's and encryptions of the s_i's.

- The decryption function can then be expressed as a polynomial of low degree (30) in the s_i's.

The squashed scheme from DGHV

- The basic decryption $m \leftarrow (c \bmod p) \bmod 2$ cannot be directly expressed as a boolean circuit of low depth.
- But it can be written as:

$$m \leftarrow [c]_2 \oplus [\lfloor c \cdot (1/p) \rceil]_2$$

and this formula can be used for ciphertext refresh if $1/p$ can be put in a compact encrypted form in the public key.

- Idea (Gentry, DGHV): use secret sharing. Represent $1/p$ as a sparse subset sum:

$$\lfloor 2^\kappa / p \rceil = \sum_{i=1}^{\Theta} s_i \cdot u_i$$

with random κ-bit integers u_i, and $s_i \in \{0, 1\}$. Publish the u_i's and encryptions of the s_i's.

- The decryption function can then be expressed as a polynomial of low degree (30) in the s_i's.

The squashed scheme from DGHV

- The basic decryption $m \leftarrow (c \bmod p) \bmod 2$ cannot be directly expressed as a boolean circuit of low depth.
- But it can be written as:

$$m \leftarrow [c]_2 \oplus [\lfloor c \cdot (1/p) \rceil]_2$$

  and this formula can be used for ciphertext refresh if $1/p$ can be put in a compact encrypted form in the public key.
- Idea (Gentry, DGHV): use secret sharing. Represent $1/p$ as a sparse subset sum:

$$\lfloor 2^\kappa / p \rceil = \sum_{i=1}^{\Theta} s_i \cdot u_i$$

  with random κ-bit integers u_i, and $s_i \in \{0,1\}$. Publish the u_i's and encryptions of the s_i's.
- The decryption function can then be expressed as a polynomial of low degree (30) in the s_i's.

The squashed scheme from DGHV

- The basic decryption $m \leftarrow (c \bmod p) \bmod 2$ cannot be directly expressed as a boolean circuit of low depth.
- But it can be written as:

$$m \leftarrow [c]_2 \oplus [\lfloor c \cdot (1/p) \rceil]_2$$

  and this formula can be used for ciphertext refresh if $1/p$ can be put in a compact encrypted form in the public key.
- Idea (Gentry, DGHV): use secret sharing. Represent $1/p$ as a sparse subset sum:

$$\lfloor 2^\kappa / p \rceil = \sum_{i=1}^{\Theta} s_i \cdot u_i$$

  with random κ-bit integers u_i, and $s_i \in \{0, 1\}$. Publish the u_i's and encryptions of the s_i's.
- The decryption function can then be expressed as a polynomial of low degree (30) in the s_i's.

Compressing the public key (I)

- Setting parameters, κ should be chosen as $\tilde{\mathcal{O}}(\lambda^5)$ bits.
  - DGHV pick $\Theta = \tilde{\mathcal{O}}(\lambda^5)$ additional elements u_i in the public key, each of size $\kappa = \tilde{\mathcal{O}}(\lambda^5)$ bits.
  - We show that one can actually take $\Theta = \tilde{\mathcal{O}}(\lambda^3)$. But this still gives a $\tilde{\mathcal{O}}(\lambda^8)$-bit public key for the squashed scheme, instead of $\tilde{\mathcal{O}}(\lambda^7)$ for the somewhat homomorphic scheme.
- Using a pseudo-random number generator:
  - Generate $\Theta - 1$ random integers $u_i \in [0, 2^{\kappa+1})$ for $2 \leq i \leq \Theta$, using a pseudo-random generator $f(\mathrm{se})$ where the seed se is generated at random during key generation and made part of the public key.
  - Only u_1 and se need to be stored in the public key.

Compressing the public key (I)

- Setting parameters, κ should be chosen as $\tilde{\mathcal{O}}(\lambda^5)$ bits.
  - DGHV pick $\Theta = \tilde{\mathcal{O}}(\lambda^5)$ additional elements u_i in the public key, each of size $\kappa = \tilde{\mathcal{O}}(\lambda^5)$ bits.
  - We show that one can actually take $\Theta = \tilde{\mathcal{O}}(\lambda^3)$. But this still gives a $\tilde{\mathcal{O}}(\lambda^8)$-bit public key for the squashed scheme, instead of $\tilde{\mathcal{O}}(\lambda^7)$ for the somewhat homomorphic scheme.
- Using a pseudo-random number generator:
  - Generate $\Theta - 1$ random integers $u_i \in [0, 2^{\kappa+1})$ for $2 \leq i \leq \Theta$, using a pseudo-random generator $f(\mathsf{se})$ where the seed se is generated at random during key generation and made part of the public key.
  - Only u_1 and se need to be stored in the public key.

Compressing the public key (II)

- Problem left: there are also Θ other elements of length γ in the public key, namely the encryptions of the s_i's.
- Gentry-Halevi trick:
  - Instead of $\vec{s} = (s_1, \ldots, s_\Theta)$, use two bit vectors $\vec{s}^{(0)}$ and $\vec{s}^{(1)}$ of length $\sqrt{\Theta}$. $\vec{s}$ is then recovered on the fly as:

$$s_{i,j} = s_i^{(0)} \cdot s_j^{(1)}$$

  - The public key only needs to contain encryptions of the bits of $\vec{s}^{(0)}$ and $\vec{s}^{(1)}$.
  - This brings down the size of this part of the public key to about $\sqrt{\Theta} \cdot \gamma = \tilde{\mathcal{O}}(\lambda^{6.5})$. Full public key remains $\approx \lambda^7$ bits.
- We borrow additional optimizations from Gentry-Halevi to further decrease key size and improve efficiency over DGHV:
  - Generate the s_i's in a "boxed" manner to simplify the decryption circuit.
  - Use fewer bits of precision in the decryption process.

Compressing the public key (II)

- Problem left: there are also Θ other elements of length γ in the public key, namely the encryptions of the s_i's.
- Gentry-Halevi trick:
  - Instead of $\vec{s} = (s_1, \dots, s_\Theta)$, use two bit vectors $\vec{s}^{(0)}$ and $\vec{s}^{(1)}$ of length $\sqrt{\Theta}$. $\vec{s}$ is then recovered on the fly as:

    $$s_{i,j} = s_i^{(0)} \cdot s_j^{(1)}$$

  - The public key only needs to contain encryptions of the bits of $\vec{s}^{(0)}$ and $\vec{s}^{(1)}$.
  - This brings down the size of this part of the public key to about $\sqrt{\Theta} \cdot \gamma = \tilde{\mathcal{O}}(\lambda^{6.5})$. Full public key remains $\approx \lambda^7$ bits.
- We borrow additional optimizations from Gentry-Halevi to further decrease key size and improve efficiency over DGHV:
  - Generate the s_i's in a "boxed" manner to simplify the decryption circuit.
  - Use fewer bits of precision in the decryption process.

Compressing the public key (II)

- Problem left: there are also Θ other elements of length γ in the public key, namely the encryptions of the s_i's.
- Gentry-Halevi trick:
  - Instead of $\vec{s} = (s_1, \ldots, s_\Theta)$, use two bit vectors $\vec{s}^{(0)}$ and $\vec{s}^{(1)}$ of length $\sqrt{\Theta}$. $\vec{s}$ is then recovered on the fly as:

$$s_{i,j} = s_i^{(0)} \cdot s_j^{(1)}$$

  - The public key only needs to contain encryptions of the bits of $\vec{s}^{(0)}$ and $\vec{s}^{(1)}$.
  - This brings down the size of this part of the public key to about $\sqrt{\Theta} \cdot \gamma = \tilde{\mathcal{O}}(\lambda^{6.5})$. Full public key remains $\approx \lambda^7$ bits.
- We borrow additional optimizations from Gentry-Halevi to further decrease key size and improve efficiency over DGHV:
  - Generate the s_i's in a “boxed” manner to simplify the decryption circuit.
  - Use fewer bits of precision in the decryption process.

Outline

How we picked concrete parameters

To propose concrete parameters for our schemes, we considered known attacks and estimated their complexity in terms of CPU cycles on a standard PC.

Attacks we considered:

- Brute force attack on the noise (with a refinement due to Nguyen).
- Orthogonal lattice-based attack on the GACD problem.
- Lattice-based attack on the sparse subset-sum problem.

Concrete parameters

Parameters	λ	ρ	η	γ	β	Θ
Toy	42	16	1088	$1.6 \cdot 10^5$	12	144
Small	52	24	1632	$0.86 \cdot 10^6$	23	533
Medium	62	32	2176	$4.2 \cdot 10^6$	44	1972
Large	72	39	2652	$19 \cdot 10^6$	88	7897

Parameters	KeyGen	Encrypt	Expand	Decrypt	Recrypt	PK size
Toy	4.38 s	0.05 s	0.03 s	0.01 s	1.92 s	0.95 MB
Small	36 s	0.79 s	0.46 s	0.01 s	10.5 s	9.6 MB
Medium	5 min 9 s	10 s	8.1 s	0.02 s	1 min 20 s	89 MB
Large	43 min	2 min 57 s	3 min 55 s	0.05 s	14 min 33 s	802 MB

Table: Concrete parameters and corresponding timings — SAGE implementation on a single core of a 3 GHz Intel Core2 CPU.

Concrete parameters

Parameters	λ	ρ	η	γ	β	Θ
Toy	≤ 38	16	1088	$1.6 \cdot 10^5$	12	144
Small	≤ 46	24	1632	$0.86 \cdot 10^6$	23	533
Medium	≤ 55	32	2176	$4.2 \cdot 10^6$	44	1972
Large	≤ 67	39	2652	$19 \cdot 10^6$	88	7897

However: new, more efficient attacks on the PACD and GACD problems put up on eprint by Chen and Nguyen last week! In view of these attacks, more conservative parameters should be picked to reach the Gentry-Halevi security levels.

Another new attack by Cohn and Heninger should also be considered (some work required to assess its bit complexity).

Conclusion

- The conceptually simple DGHV fully homomorphic scheme can be compressed into a scheme implementable on a standard PC.
- But there is still a long way to go to achieve practicality.
- Ongoing progress:
  - Exciting new developments by Brakerski, Gentry and Vaikuntanathan!
    - Can be applied to FHE over the integers (on eprint soon!).
  - Simple trick to compress public keys much further (on eprint now!).
  - Possible to use polynomials of higher degree instead of quadratic forms to achieve better efficiency.
- There is progress on attacking the underlying hard problems as well.

Conclusion

- The conceptually simple DGHV fully homomorphic scheme can be compressed into a scheme implementable on a standard PC.
- But there is still a long way to go to achieve practicality.
- Ongoing progress:
  - Exciting new developments by Brakerski, Gentry and Vaikuntanathan!
    - Can be applied to FHE over the integers (on eprint soon!).
  - Simple trick to compress public keys much further (on eprint now!).
  - Possible to use polynomials of higher degree instead of quadratic forms to achieve better efficiency.
- There is progress on attacking the underlying hard problems as well.

Conclusion

- The conceptually simple DGHV fully homomorphic scheme can be compressed into a scheme implementable on a standard PC.
- But there is still a long way to go to achieve practicality.
- Ongoing progress:
    - Exciting new developments by Brakerski, Gentry and Vaikuntanathan!
        - Can be applied to FHE over the integers (on eprint soon!).
    - Simple trick to compress public keys much further (on eprint now!).
    - Possible to use polynomials of higher degree instead of quadratic forms to achieve better efficiency.
- There is progress on attacking the underlying hard problems as well.

Conclusion

- The conceptually simple DGHV fully homomorphic scheme can be compressed into a scheme implementable on a standard PC.
- But there is still a long way to go to achieve practicality.
- Ongoing progress:
  - Exciting new developments by Brakerski, Gentry and Vaikuntanathan!
    - Can be applied to FHE over the integers (on eprint soon!).
  - Simple trick to compress public keys much further (on eprint now!).
  - Possible to use polynomials of higher degree instead of quadratic forms to achieve better efficiency.
- There is progress on attacking the underlying hard problems as well.

Thank you!