Fully Homomorphic Encryption

Jean-Sébastien Coron

University of Luxembourg

October 6, 2014

Homomorphic Encryption

- Homomorphic encryption: perform operations on plaintexts while manipulating only ciphertexts.
  - Normally, this is not possible.

$$\begin{aligned} AES_K(m_1) &= \texttt{0x3c7317c6bc5634a4ad8479c64714f4f8} \\ AES_K(m_2) &= \texttt{0x7619884e1961b051be1aa407da6cac2c} \\ AES_K(m_1 \oplus m_2) &= \ ? \end{aligned}$$

- For some cryptosystems with algebraic structure, this is possible. For example RSA:

$$\left.\begin{aligned} c_1 &= {m_1}^e \bmod N \\ c_2 &= {m_2}^e \bmod N \end{aligned}\right. \Rightarrow c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

Homomorphic Encryption

- Homomorphic encryption: perform operations on plaintexts while manipulating only ciphertexts.
  - Normally, this is not possible.

$$
\begin{aligned}
\mathsf{AES}_K(m_1) &= \texttt{0x3c7317c6bc5634a4ad8479c64714f4f8} \\
\mathsf{AES}_K(m_2) &= \texttt{0x7619884e1961b051be1aa407da6cac2c} \\
\mathsf{AES}_K(m_1 \oplus m_2) &= \; ?
\end{aligned}
$$

- For some cryptosystems with algebraic structure, this is possible. For example RSA:

$$
\begin{aligned}
c_1 &= {m_1}^e \bmod N \\
c_2 &= {m_2}^e \bmod N
\end{aligned}
\Rightarrow c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N
$$

Homomorphic Encryption

- Homomorphic encryption: perform operations on plaintexts while manipulating only ciphertexts.
  - Normally, this is not possible.

$$\begin{aligned} \mathrm{AES}_K(m_1) &= \texttt{0x3c7317c6bc5634a4ad8479c64714f4f8} \\ \mathrm{AES}_K(m_2) &= \texttt{0x7619884e1961b051be1aa407da6cac2c} \\ \mathrm{AES}_K(m_1 \oplus m_2) &= ? \end{aligned}$$

- For some cryptosystems with algebraic structure, this is possible. For example RSA:

$$\left.\begin{aligned} c_1 &= {m_1}^e \bmod N \\ c_2 &= {m_2}^e \bmod N \end{aligned}\right. \Rightarrow c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

Homomorphic Encryption with RSA

- Multiplicative property of RSA.

$$\left.\begin{array}{l} c_1 = {m_1}^e \bmod N \\ c_2 = {m_2}^e \bmod N \end{array}\right. \Rightarrow c = c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

- Homomorphic encryption: given c_1 and c_2, we can compute the ciphertext c for $m_1 \cdot m_2 \bmod N$
  - using only the public-key
  - without knowing the plaintexts m_1 and m_2.

Homomorphic Encryption with RSA

- Multiplicative property of RSA.

$$\begin{aligned} c_1 &= {m_1}^e \bmod N \\ c_2 &= {m_2}^e \bmod N \end{aligned} \quad \Rightarrow c = c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

- Homomorphic encryption: given c_1 and c_2, we can compute the ciphertext c for $m_1 \cdot m_2 \bmod N$
  - using only the public-key
  - without knowing the plaintexts m_1 and m_2.

Paillier Cryptosystem

- Additively homomorphic: Paillier cryptosystem

$$\begin{array}{l} c_1 = g^{m_1} \bmod N^2 \\ c_2 = g^{m_2} \bmod N^2 \end{array} \Rightarrow c_1 \cdot c_2 = g^{m_1 + m_2\ [N]} \bmod N^2$$

- Application: e-voting.
  - Voter i encrypts his vote $m_i \in \{0, 1\}$ into:

$$c_i = g^{m_i} \cdot z_i^N \bmod N^2$$

  - Votes can be aggregated using only the public-key:

$$c = \prod_i c_i = g^{\sum_i m_i} \cdot z \bmod N^2$$

  - c is enventually decrypted to recover $m = \sum_i m_i$

Paillier Cryptosystem

- Additively homomorphic: Paillier cryptosystem

$$\left.\begin{array}{l} c_1 = g^{m_1} \bmod N^2 \\ c_2 = g^{m_2} \bmod N^2 \end{array}\right. \Rightarrow c_1 \cdot c_2 = g^{m_1+m_2\ [N]} \bmod N^2$$

- Application: e-voting.
  - Voter i encrypts his vote $m_i \in \{0, 1\}$ into:

$$c_i = g^{m_i} \cdot z_i^N \bmod N^2$$

  - Votes can be aggregated using only the public-key:

$$c = \prod_i c_i = g^{\sum_i m_i} \cdot z \bmod N^2$$

  - c is enventually decrypted to recover $m = \sum_i m_i$

Fully homomorphic encryption

- Multiplicatively homomorphic: RSA.

$$\left.\begin{aligned} c_1 &= {m_1}^e \bmod N \\ c_2 &= {m_2}^e \bmod N \end{aligned}\right. \Rightarrow c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

- Additively homomorphic: Paillier

$$\left.\begin{aligned} c_1 &= g^{m_1} \bmod N^2 \\ c_2 &= g^{m_2} \bmod N^2 \end{aligned}\right. \Rightarrow c_1 \cdot c_2 = g^{m_1+m_2\,[N]} \bmod N^2$$

- Fully homomorphic: homomorphic for both addition and multiplication
    - Open problem until Gentry's breakthrough in 2009.

Fully homomorphic encryption

- Multiplicatively homomorphic: RSA.

$$\left.\begin{array}{l} c_1 = {m_1}^e \bmod N \\ c_2 = {m_2}^e \bmod N \end{array}\right. \Rightarrow c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

- Additively homomorphic: Paillier

$$\left.\begin{array}{l} c_1 = g^{m_1} \bmod N^2 \\ c_2 = g^{m_2} \bmod N^2 \end{array}\right. \Rightarrow c_1 \cdot c_2 = g^{m_1+m_2\ [N]} \bmod N^2$$

- Fully homomorphic: homomorphic for both addition and multiplication
  - Open problem until Gentry's breakthrough in 2009.

Fully homomorphic encryption

- Multiplicatively homomorphic: RSA.

$$\left.\begin{aligned} c_1 &= {m_1}^e \bmod N \\ c_2 &= {m_2}^e \bmod N \end{aligned}\right. \Rightarrow c_1 \cdot c_2 = (m_1 \cdot m_2)^e \bmod N$$

- Additively homomorphic: Paillier

$$\left.\begin{aligned} c_1 &= g^{m_1} \bmod N^2 \\ c_2 &= g^{m_2} \bmod N^2 \end{aligned}\right. \Rightarrow c_1 \cdot c_2 = g^{m_1+m_2\ [N]} \bmod N^2$$

- 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:
  - $0 \rightarrow 203ef6124 \ldots 23ab87_{16}$
  - $1 \rightarrow b327653c1 \ldots db3265_{16}$
  - Obviously, encryption must be probabilistic.
- 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.
- Why is it important ?
  - Universality: any Boolean circuit can be written with Xors and Ands.
  - Once you can homomorphically evaluate both a Xor and a And, you can evaluate any Boolean circuit, any computable function.

Fully homomorphic public-key encryption

- We restrict ourselves to public-key encryption of a single bit:
  - $0 \to 203\text{ef}6124\ldots 23\text{ab}87_{16}$
  - $1 \to \text{b}327653\text{c}1\ldots\text{db}3265_{16}$
  - Obviously, encryption must be probabilistic.
- 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.
- Why is it important ?
  - Universality: any Boolean circuit can be written with Xors and Ands.
  - Once you can homomorphically evaluate both a Xor and a And, you can evaluate any Boolean circuit, any computable function.

Fully homomorphic public-key encryption

- We restrict ourselves to public-key encryption of a single bit:
  - $0 \rightarrow 203ef6124\ldots23ab87_{16}$
  - $1 \rightarrow b327653c1\ldots db3265_{16}$
  - Obviously, encryption must be probabilistic.
- 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.
- Why is it important ?
  - Universality: any Boolean circuit can be written with Xors and Ands.
  - Once you can homomorphically evaluate both a Xor and a And, you can evaluate any Boolean circuit, any computable function.

Outsourcing Computation

- The cloud receives some data m in encrypted form.
  - It receives the ciphertexts c_i corresponding to bits m_i
  - The cloud doesn't know the m_i's
- The cloud performs some computation $f(m)$, but without knowing m
  - The computation of f is written as a Boolean circuit with Xors and Ands
  - Every Xor $z = x \oplus y$ is homomorphically evaluated from the ciphertexts c_x and c_y, to get ciphertext c_z
  - Every And $z' = x \cdot y$ is homomorphically evaluated from the ciphertexts c_x and c_y, to get ciphertext $c_{z'}$
- Eventally the cloud obtains a ciphertext c for $f(m)$
  - The user decrypts c to recover $f(m)$
  - The cloud learns nothing about m

Outsourcing Computation

- The cloud receives some data m in encrypted form.
  - It receives the ciphertexts c_i corresponding to bits m_i
  - The cloud doesn't know the m_i's
- The cloud performs some computation $f(m)$, but without knowing m
  - The computation of f is written as a Boolean circuit with Xors and Ands
  - Every Xor $z = x \oplus y$ is homomorphically evaluated from the ciphertexts c_x and c_y, to get ciphertext c_z
  - Every And $z' = x \cdot y$ is homomorphically evaluated from the ciphertexts c_x and c_y, to get ciphertext $c_{z'}$
- Eventally the cloud obtains a ciphertext c for $f(m)$
  - The user decrypts c to recover $f(m)$
  - The cloud learns nothing about m

Outsourcing Computation

- The cloud receives some data m in encrypted form.
  - It receives the ciphertexts c_i corresponding to bits m_i
  - The cloud doesn't know the m_i's
- The cloud performs some computation $f(m)$, but without knowing m
  - The computation of f is written as a Boolean circuit with Xors and Ands
  - Every Xor $z = x \oplus y$ is homomorphically evaluated from the ciphertexts c_x and c_y, to get ciphertext c_z
  - Every And $z' = x \cdot y$ is homomorphically evaluated from the ciphertexts c_x and c_y, to get ciphertext $c_{z'}$
- Eventally the cloud obtains a ciphertext c for $f(m)$
  - The user decrypts c to recover $f(m)$
  - The cloud learns nothing about m

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.

Cloud Computing

- Goal: cloud computing
  - I encrypt my data before sending it to the cloud
  - The cloud can still search, sort and edit my data on my behalf
  - Data is kept in encrypted form in the cloud.
  - The cloud learns nothing about my data
- The cloud returns encrypted answers
  - that only I can decrypt

Fully Homomorphic Encryption Schemes

- 1. Breakthrough scheme of Gentry [G09], based on ideal lattices. Some optimizations by [SV10].
  - Implementation [GH11]: PK size: 2.3 GB, recrypt: 30 min.
- 2. RLWE schemes [BV11a,BV11b].
  - FHE without bootstrapping (modulus switching) [BGV11]
  - Batch FHE [GHS12]
  - Implementation with homomorphic evaluation of AES [GHS12]
  - And many other papers...
- 3. van Dijk, Gentry, Halevi and Vaikuntanathan's scheme over the integers [DGHV10].
  - Implementation [CMNT11]: PK size: 1 GB, recrypt: 15 min.
  - Public-key compression and modulus switching [CNT12]
  - Batch and homomorphic evaluation of AES [CCKLLTY13].

Fully Homomorphic Encryption Schemes

- 1. Breakthrough scheme of Gentry [G09], based on ideal lattices. Some optimizations by [SV10].
  - Implementation [GH11]: PK size: 2.3 GB, recrypt: 30 min.
- 2. RLWE schemes [BV11a,BV11b].
  - FHE without bootstrapping (modulus switching) [BGV11]
  - Batch FHE [GHS12]
  - Implementation with homomorphic evaluation of AES [GHS12]
  - And many other papers...
- 3. van Dijk, Gentry, Halevi and Vaikuntanathan's scheme over the integers [DGHV10].
  - Implementation [CMNT11]: PK size: 1 GB, recrypt: 15 min.
  - Public-key compression and modulus switching [CNT12]
  - Batch and homomorphic evaluation of AES [CCKLLTY13].

Fully Homomorphic Encryption Schemes

- 1. Breakthrough scheme of Gentry [G09], based on ideal lattices. Some optimizations by [SV10].
  - Implementation [GH11]: PK size: 2.3 GB, recrypt: 30 min.
- 2. RLWE schemes [BV11a,BV11b].
  - FHE without bootstrapping (modulus switching) [BGV11]
  - Batch FHE [GHS12]
  - Implementation with homomorphic evaluation of AES [GHS12]
  - And many other papers...
- 3. van Dijk, Gentry, Halevi and Vaikuntanathan's scheme over the integers [DGHV10].
  - Implementation [CMNT11]: PK size: 1 GB, recrypt: 15 min.
  - Public-key compression and modulus switching [CNT12]
  - Batch and homomorphic evaluation of AES [CCKLLTY13].

The DGHV Scheme

- Ciphertext for $m \in \{0, 1\}$:

$$c = q \cdot p + 2r + m$$

where p is the secret-key, q and r are randoms.

- Decryption:

$$(c \bmod p) \bmod 2 = m$$

- Parameters:

The DGHV Scheme

- Ciphertext for $m \in \{0, 1\}$:

$$c = q \cdot p + 2r + m$$

  where p is the secret-key, q and r are randoms.

- Decryption:

$$(c \bmod p) \bmod 2 = m$$

- Parameters:

The DGHV Scheme

- Ciphertext for $m \in \{0, 1\}$:

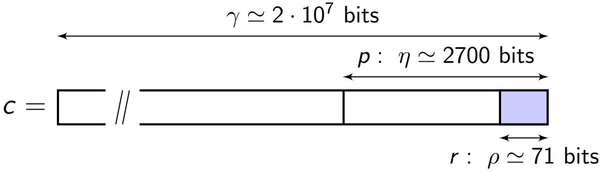
$$c = q \cdot p + 2r + m$$

where p is the secret-key, q and r are randoms.

- Decryption:

$$(c \bmod p) \bmod 2 = m$$

- Parameters:



Homomorphic Properties of DGHV

- Addition:

$$\begin{array}{l} c_1 = q_1 \cdot p + 2r_1 + m_1 \\ c_2 = q_2 \cdot p + 2r_2 + m_2 \end{array} \Rightarrow c_1 + c_2 = q' \cdot p + 2r' + m_1 + m_2$$

  - $c_1 + c_2$ is an encryption of $m_1 + m_2 \bmod 2 = m_1 \oplus m_2$

- Multiplication:

$$\begin{array}{l} c_1 = q_1 \cdot p + 2r_1 + m_1 \\ c_2 = q_2 \cdot p + 2r_2 + m_2 \end{array} \Rightarrow c_1 \cdot c_2 = q'' \cdot p + 2r'' + m_1 \cdot m_2$$

  with

$$r'' = 2r_1 r_2 + r_1 m_2 + r_2 m_1$$

  - $c_1 \cdot c_2$ is an encryption of $m_1 \cdot m_2$
  - Noise becomes twice larger.

Homomorphic Properties of DGHV

- Addition:

$$\begin{array}{l} c_1 = q_1 \cdot p + 2r_1 + m_1 \\ c_2 = q_2 \cdot p + 2r_2 + m_2 \end{array} \Rightarrow c_1 + c_2 = q' \cdot p + 2r' + m_1 + m_2$$

  - $c_1 + c_2$ is an encryption of $m_1 + m_2 \bmod 2 = m_1 \oplus m_2$

- Multiplication:

$$\begin{array}{l} c_1 = q_1 \cdot p + 2r_1 + m_1 \\ c_2 = q_2 \cdot p + 2r_2 + m_2 \end{array} \Rightarrow c_1 \cdot c_2 = q'' \cdot p + 2r'' + m_1 \cdot m_2$$

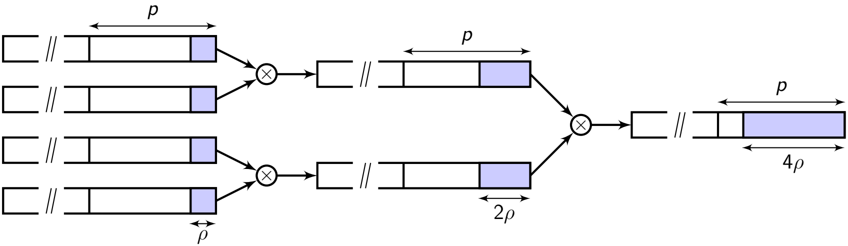
  with

$$r'' = 2r_1 r_2 + r_1 m_2 + r_2 m_1$$

  - $c_1 \cdot c_2$ is an encryption of $m_1 \cdot m_2$
  - Noise becomes twice larger.

Somewhat homomorphic scheme

- The number of multiplications is limited.
  - Noise grows with the number of multiplications.
  - Noise must remain $< p$ for correct decryption.



Gentry's technique

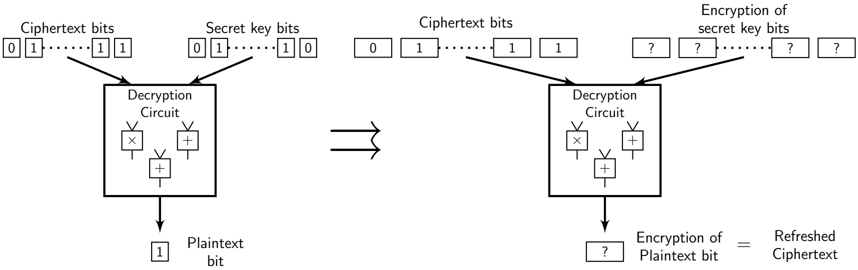
- To build a FHE scheme, start from the somewhat homomorphic scheme, that is:
  - Only a polynomial of small degree can be homomorphically applied on ciphertexts.
  - Otherwise the noise becomes too large and decryption becomes incorrect.
- Then, "squash" the decryption procedure:
  - express the decryption 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: bootstrapping

- 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.

Public-key Encryption with DGHV

- Ciphertext

$$c = q \cdot p + 2r + m$$

- Public-key: a set of τ encryptions of 0's.

$$x_i = q_i \cdot p + 2r_i$$

- Public-key encryption:

$$c = m + 2r + \sum_{i=1}^{\tau} \varepsilon_i \cdot x_i$$

for random $\varepsilon_i \in \{0, 1\}$.

Public-key Encryption with DGHV

- Ciphertext

$$c = q \cdot p + 2r + m$$

- Public-key: a set of τ encryptions of 0's.

$$x_i = q_i \cdot p + 2r_i$$

- Public-key encryption:

$$c = m + 2r + \sum_{i=1}^{\tau} \varepsilon_i \cdot x_i$$

for random $\varepsilon_i \in \{0, 1\}$.

Public-key Encryption with DGHV

- Ciphertext

$$c = q \cdot p + 2r + m$$

- Public-key: a set of τ encryptions of 0's.

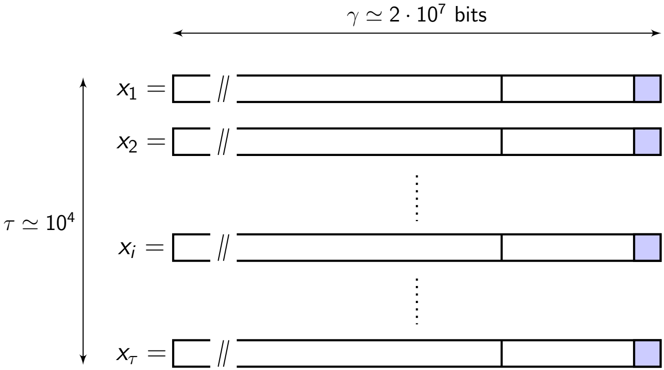
$$x_i = q_i \cdot p + 2r_i$$

- Public-key encryption:

$$c = m + 2r + \sum_{i=1}^{\tau} \varepsilon_i \cdot x_i$$

for random $\varepsilon_i \in \{0, 1\}$.

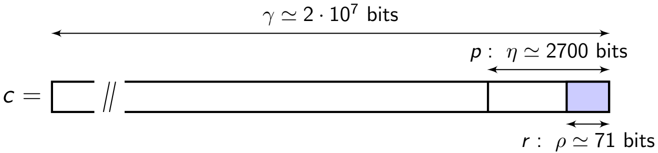
Public Key Size



- Public-key size: $\tau \cdot \gamma = 2 \cdot 10^{11}$ bits $= 25$ GB !
  - In [CMNT11], with quadratic encryption, PK size of 1 GB.

DGHV Ciphertext Compression

- Ciphertext: $c = q \cdot p + 2r + m$

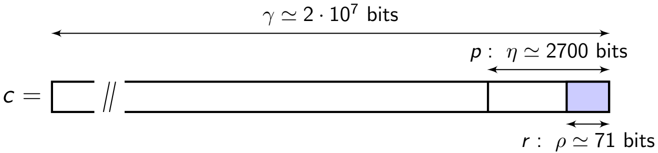


- Compute a pseudo-random $\chi = f(seed)$ of γ bits.

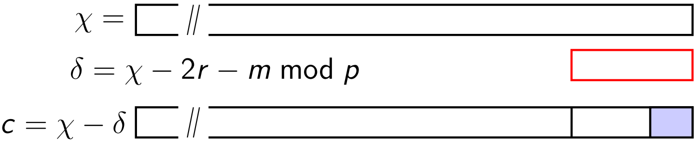
- Only store *seed* and the small correction δ.
- Storage: $\simeq 2700$ bits instead of $2 \cdot 10^7$ bits!

DGHV Ciphertext Compression

- Ciphertext: $c = q \cdot p + 2r + m$



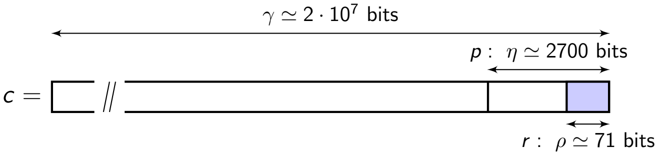
- Compute a pseudo-random $\chi = f(seed)$ of γ bits.



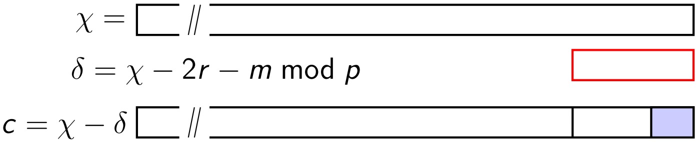
- Only store *seed* and the small correction δ.
- Storage: $\simeq 2\,700$ bits instead of $2 \cdot 10^7$ bits !

DGHV Ciphertext Compression

- Ciphertext: $c = q \cdot p + 2r + m$



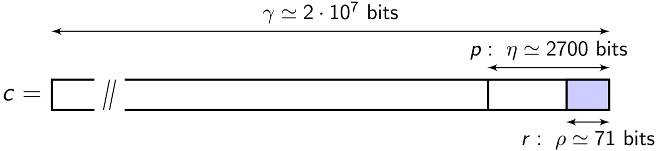
- Compute a pseudo-random $\chi = f(seed)$ of γ bits.



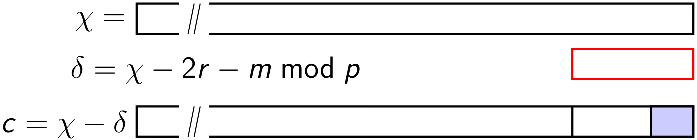
  - Only store *seed* and the small correction δ.
  - Storage: $\simeq 2\,700$ bits instead of $2 \cdot 10^7$ bits !

DGHV Ciphertext Compression

- Ciphertext: $c = q \cdot p + 2r + m$

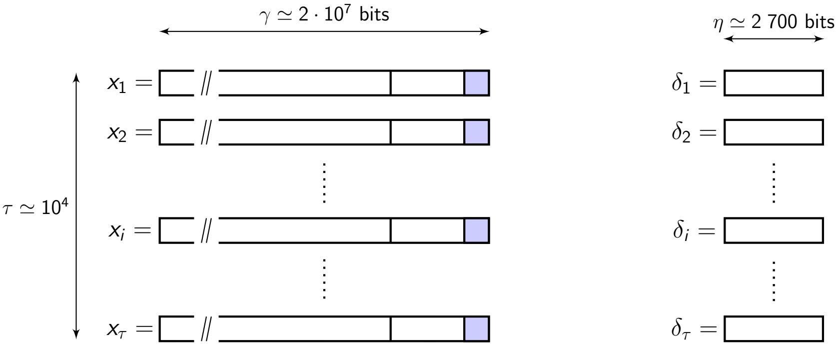


- Compute a pseudo-random $\chi = f(\mathit{seed})$ of γ bits.

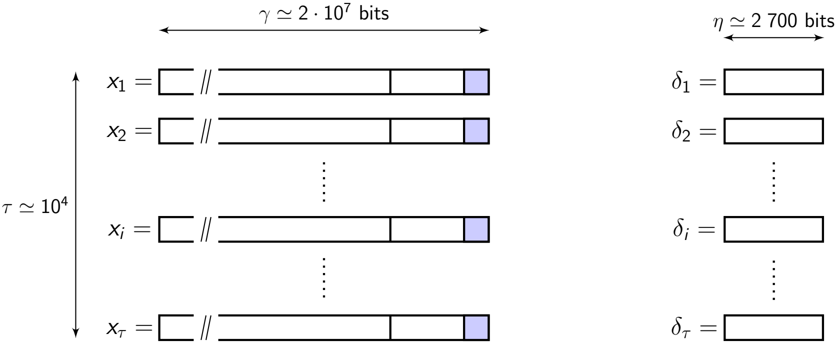


  - Only store *seed* and the small correction δ.
  - Storage: $\simeq 2\,700$ bits instead of $2 \cdot 10^7$ bits !

Compressed Public Key



Compressed Public Key



Old PK: 25 GB

New PK: 3.4 MB !

Security of Compressed PK

- Original DGHV scheme is semantically secure, under the approximate-gcd assumption.
  - Approximate-gcd problem: given a set of $x_i = q_i \cdot p + r_i$, recover p.
- Compressed public key
  - *seed* is part of the public-key, to recover the x_i's, so we cannot argue that $f(seed)$ is pseudo-random.
  - Security in the random oracle model only, still based on approximate-gcd.

Security of Compressed PK

- Original DGHV scheme is semantically secure, under the approximate-gcd assumption.
  - Approximate-gcd problem: given a set of $x_i = q_i \cdot p + r_i$, recover p.
- Compressed public key
  - *seed* is part of the public-key, to recover the x_i's, so we cannot argue that $f(seed)$ is pseudo-random.
  - Security in the random oracle model only, still based on approximate-gcd.

Security of Compressed PK

- Original DGHV scheme is semantically secure, under the approximate-gcd assumption.
  - Approximate-gcd problem: given a set of $x_i = q_i \cdot p + r_i$, recover p.
- Compressed public key
  - *seed* is part of the public-key, to recover the x_i's, so we cannot argue that $f(seed)$ is pseudo-random.
  - Security in the random oracle model only, still based on approximate-gcd.

PK Generation

$$\chi_i = H(seed, i)$$
$$\delta_i = [\chi_i]_p + \lambda_i \cdot p - r_i$$
$$x_i = \chi_i - \delta_i$$

Security of Compressed PK

- Original DGHV scheme is semantically secure, under the approximate-gcd assumption.
  - Approximate-gcd problem: given a set of $x_i = q_i \cdot p + r_i$, recover p.
- Compressed public key
  - *seed* is part of the public-key, to recover the x_i's, so we cannot argue that $f(seed)$ is pseudo-random.
  - Security in the random oracle model only, still based on approximate-gcd.

PK Generation

$$\chi_i = H(seed, i)$$
$$\delta_i = [\chi_i]_p + \lambda_i \cdot p - r_i$$
$$x_i = \chi_i - \delta_i$$

Simulation in ROM

$$H(seed, i) \leftarrow x_i + \delta_i$$
$$\delta_i \leftarrow \{0,1\}^{\eta+\lambda}$$
$$x_i = q_i \cdot p + r_i$$

Hardness assumption for semantic security

- Original DGHV scheme: secure under the General Approximate Common Divisor (GACD) assumption.
  - Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Efficient DGHV variant: secure under the Partial Approximate Common Divisor (PACD) assumption.
  - Given $x_0 = p \cdot q_0$ and polynomially many $x_i = p \cdot q_i + r_i$, find p.
- PACD is clearly easier than GACD.
  - How much easier ?

Hardness assumption for semantic security

- Original DGHV scheme: secure under the General Approximate Common Divisor (GACD) assumption.
  - Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Efficient DGHV variant: secure under the Partial Approximate Common Divisor (PACD) assumption.
  - Given $x_0 = p \cdot q_0$ and polynomially many $x_i = p \cdot q_i + r_i$, find p.
- PACD is clearly easier than GACD.
  - How much easier ?

Hardness assumption for semantic security

- Original DGHV scheme: secure under the General Approximate Common Divisor (GACD) assumption.
  - Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Efficient DGHV variant: secure under the Partial Approximate Common Divisor (PACD) assumption.
  - Given $x_0 = p \cdot q_0$ and polynomially many $x_i = p \cdot q_i + r_i$, find p.
- PACD is clearly easier than GACD.
  - How much easier ?

Brute force attack on the noise

- Brute force attack on the noise.
  - Given $x_0 = q_0 \cdot p$ and $x_1 = q_1 \cdot p + r_1$ with $|r_1| < 2^\rho$, one can guess r_1 and compute $\gcd(x_0, x_1 - r_1)$ to recover p.
  - Requires 2^ρ gcd computation
- Countermeasure:
  - Take a sufficiently large ρ

Solving PACD

- Given $x_0 = p \cdot q_0$ and polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Brute force attack: 2^ρ GCD computations.
  - with $x_0 = q_0 \cdot p$ and $x_1 = q_1 \cdot p + r_1$ and $0 \leq r_1 < 2^\rho$.
- Variant suggested by Phong Nguyen, still in $\mathcal{O}(2^\rho)$:

$$p = \gcd\left(x_0, \prod_{i=0}^{2^\rho - 1} (x_1 - i) \bmod x_0\right)$$

- Improved attack in $\tilde{\mathcal{O}}(2^{\rho/2})$ time and memory by Chen and Nguyen at Eurocrypt 2012.

Solving PACD

- Given $x_0 = p \cdot q_0$ and polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Brute force attack: 2^ρ GCD computations.
  - with $x_0 = q_0 \cdot p$ and $x_1 = q_1 \cdot p + r_1$ and $0 \leq r_1 < 2^\rho$.
- Variant suggested by Phong Nguyen, still in $\mathcal{O}(2^\rho)$:

$$p = \gcd\left(x_0, \prod_{i=0}^{2^\rho-1}(x_1 - i) \bmod x_0\right)$$

- Improved attack in $\tilde{\mathcal{O}}(2^{\rho/2})$ time and memory by Chen and Nguyen at Eurocrypt 2012.

Solving PACD

- Given $x_0 = p \cdot q_0$ and polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Brute force attack: 2^ρ GCD computations.
  - with $x_0 = q_0 \cdot p$ and $x_1 = q_1 \cdot p + r_1$ and $0 \leq r_1 < 2^\rho$.
- Variant suggested by Phong Nguyen, still in $\mathcal{O}(2^\rho)$:

$$p = \gcd\left(x_0, \prod_{i=0}^{2^\rho - 1} (x_1 - i) \bmod x_0\right)$$

- Improved attack in $\tilde{\mathcal{O}}(2^{\rho/2})$ time and memory by Chen and Nguyen at Eurocrypt 2012.

Solving PACD

- Given $x_0 = p \cdot q_0$ and polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Brute force attack: 2^ρ GCD computations.
  - with $x_0 = q_0 \cdot p$ and $x_1 = q_1 \cdot p + r_1$ and $0 \leq r_1 < 2^\rho$.
- Variant suggested by Phong Nguyen, still in $\mathcal{O}(2^\rho)$:

$$p = \gcd\left(x_0, \prod_{i=0}^{2^\rho - 1} (x_1 - i) \bmod x_0\right)$$

- Improved attack in $\tilde{\mathcal{O}}(2^{\rho/2})$ time and memory by Chen and Nguyen at Eurocrypt 2012.

Solving GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
  - Variant without $x_0 = q_0 \cdot p$.
- Brute force attack: $2^{2\rho}$ GCD computations.
  - From $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$
- Using Chen-Nguyen attack: $\tilde{\mathcal{O}}(2^{3\rho/2})$ time.
  - Guess r_1 and apply Chen-Nguyen on r_2
  - $\mathcal{O}(2^\rho) \cdot \tilde{\mathcal{O}}(2^{\rho/2}) = \tilde{\mathcal{O}}(2^{3\rho/2})$ time and $\tilde{\mathcal{O}}(2^{\rho/2})$ memory.
- Better attack [CNT12]: $\tilde{\mathcal{O}}(2^\rho)$ time and memory.

Solving GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
  - Variant without $x_0 = q_0 \cdot p$.
- Brute force attack: $2^{2\rho}$ GCD computations.
  - From $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$
- Using Chen-Nguyen attack: $\tilde{\mathcal{O}}(2^{3\rho/2})$ time.
  - Guess r_1 and apply Chen-Nguyen on r_2
  - $\mathcal{O}(2^\rho) \cdot \tilde{\mathcal{O}}(2^{\rho/2}) = \tilde{\mathcal{O}}(2^{3\rho/2})$ time and $\tilde{\mathcal{O}}(2^{\rho/2})$ memory.
- Better attack [CNT12]: $\tilde{\mathcal{O}}(2^\rho)$ time and memory.

Solving GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
  - Variant without $x_0 = q_0 \cdot p$.
- Brute force attack: $2^{2\rho}$ GCD computations.
  - From $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$
- Using Chen-Nguyen attack: $\tilde{\mathcal{O}}(2^{3\rho/2})$ time.
  - Guess r_1 and apply Chen-Nguyen on r_2
  - $\mathcal{O}(2^\rho) \cdot \tilde{\mathcal{O}}(2^{\rho/2}) = \tilde{\mathcal{O}}(2^{3\rho/2})$ time and $\tilde{\mathcal{O}}(2^{\rho/2})$ memory.
- Better attack [CNT12]: $\tilde{\mathcal{O}}(2^\rho)$ time and memory.

Solving GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
  - Variant without $x_0 = q_0 \cdot p$.
- Brute force attack: $2^{2\rho}$ GCD computations.
  - From $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$
- Using Chen-Nguyen attack: $\tilde{\mathcal{O}}(2^{3\rho/2})$ time.
  - Guess r_1 and apply Chen-Nguyen on r_2
  - $\mathcal{O}(2^{\rho}) \cdot \tilde{\mathcal{O}}(2^{\rho/2}) = \tilde{\mathcal{O}}(2^{3\rho/2})$ time and $\tilde{\mathcal{O}}(2^{\rho/2})$ memory.
- Better attack [CNT12]: $\tilde{\mathcal{O}}(2^{\rho})$ time and memory.

Better Attack against GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Variant of the previous equation with $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$

$$p \,|\, \gcd\left(\prod_{i=0}^{2^\rho - 1}(x_1 - i), \prod_{i=0}^{2^\rho - 1}(x_2 - i)\right)$$

  - Product over $\mathbb{Z}$ can be computed in $\tilde{O}(2^\rho)$ time using a product tree.
  - $\tilde{O}(2^\rho)$ time and memory.
- Problem: many parasitic factors.
  - Can be eliminated by taking the gcd with more products,
  - and by dividing by $B!$ for $B \simeq 2^\rho$.

Better Attack against GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Variant of the previous equation with $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$

$$p \,\Big|\, \gcd\left(\prod_{i=0}^{2^\rho-1}(x_1 - i), \prod_{i=0}^{2^\rho-1}(x_2 - i)\right)$$

  - Product over $\mathbb{Z}$ can be computed in $\tilde{\mathcal{O}}(2^\rho)$ time using a product tree.
  - $\tilde{\mathcal{O}}(2^\rho)$ time and memory
- Problem: many parasitic factors.
  - Can be eliminated by taking the gcd with more products,
  - and by dividing by $B!$ for $B \simeq 2^\rho$.

Better Attack against GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Variant of the previous equation with $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$

$$p \,|\, \gcd\left(\prod_{i=0}^{2^\rho-1}(x_1 - i), \prod_{i=0}^{2^\rho-1}(x_2 - i)\right)$$

  - Product over $\mathbb{Z}$ can be computed in $\tilde{\mathcal{O}}(2^\rho)$ time using a product tree.
  - $\tilde{\mathcal{O}}(2^\rho)$ time and memory
- Problem: many parasitic factors.
  - Can be eliminated by taking the gcd with more products,
  - and by dividing by $B!$ for $B \simeq 2^\rho$.

Better Attack against GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Variant of the previous equation with $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$

$$p \mid \gcd\left(\prod_{i=0}^{2^\rho-1}(x_1 - i), \prod_{i=0}^{2^\rho-1}(x_2 - i)\right)$$

  - Product over $\mathbb{Z}$ can be computed in $\tilde{\mathcal{O}}(2^\rho)$ time using a product tree.
  - $\tilde{\mathcal{O}}(2^\rho)$ time and memory
- Problem: many parasitic factors.
  - Can be eliminated by taking the gcd with more products,
  - and by dividing by $B!$ for $B \simeq 2^\rho$.

Better Attack against GACD

- Given polynomially many $x_i = p \cdot q_i + r_i$, find p.
- Variant of the previous equation with $x_1 = p \cdot q_1 + r_1$ and $x_2 = p \cdot q_2 + r_2$

$$p \,\Big|\, \gcd\left(\prod_{i=0}^{2^\rho-1}(x_1 - i), \prod_{i=0}^{2^\rho-1}(x_2 - i)\right)$$

  - Product over $\mathbb{Z}$ can be computed in $\tilde{\mathcal{O}}(2^\rho)$ time using a product tree.
  - $\tilde{\mathcal{O}}(2^\rho)$ time and memory
- Problem: many parasitic factors.
  - Can be eliminated by taking the gcd with more products,
  - and by dividing by $B!$ for $B \simeq 2^\rho$.

Approximate GCD attack

- Consider t integers: $x_i = p \cdot q_i + r_i$ and $x_0 = p \cdot q_0$.
  - Consider a vector $\vec{u}$ orthogonal to the x_i's:

$$\sum_{i=1}^{t} u_i \cdot x_i = 0 \mod x_0$$

  - This gives $\sum_{i=1}^{t} u_i \cdot r_i = 0 \mod p$.
- If the u_i's are sufficiently small, since the r_i's are small this equality will hold over $\mathbb{Z}$.
  - Such vector $\vec{u}$ can be found using LLL.
- By collecting many orthogonal vectors one can recover $\vec{r}$ and eventually the secret key p
- Countermeasure
  - The size γ of the x_i's must be sufficiently large.

The DGHV scheme (simplified)

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

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

    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 (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.

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.
- Alternative decryption formula for $c = q \cdot p + 2r + m$
  - We have $q = \lfloor c/p \rceil$ and $c = q + m \pmod 2$
  - Therefore

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

- Idea (Gentry, DGHV). Secret-share $1/p$ as a sparse subset sum:

$$1/p = \sum_{i=1}^{\Theta} s_i \cdot y_i + \varepsilon$$

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.
- Alternative decryption formula for $c = q \cdot p + 2r + m$
  - We have $q = \lfloor c/p \rceil$ and $c = q + m \pmod 2$
  - Therefore

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

- Idea (Gentry, DGHV). Secret-share $1/p$ as a sparse subset sum:

$$1/p = \sum_{i=1}^{\Theta} s_i \cdot y_i + \varepsilon$$

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.
- Alternative decryption formula for $c = q \cdot p + 2r + m$
  - We have $q = \lfloor c/p \rceil$ and $c = q + m \pmod 2$
  - Therefore

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

- Idea (Gentry, DGHV). Secret-share $1/p$ as a sparse subset sum:

$$1/p = \sum_{i=1}^{\Theta} s_i \cdot y_i + \varepsilon$$

Squashed decryption

- Alternative equation

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

- Secret-share $1/p$ as a sparse subset sum:

$$1/p = \sum_{i=1}^{\Theta} s_i \cdot y_i + \varepsilon$$

with random public κ-bit numbers y_i, and sparse secret $s_i \in \{0, 1\}$.

- Decryption becomes:

$$m \leftarrow [c]_2 \oplus \left[\left\lfloor \sum_{i=1}^{\Theta} s_i \cdot (y_i \cdot c) \right\rceil \right]_2$$

Squashed decryption

- Alternative equation

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

- Secret-share $1/p$ as a sparse subset sum:

$$1/p = \sum_{i=1}^{\Theta} s_i \cdot y_i + \varepsilon$$

with random public κ-bit numbers y_i, and sparse secret $s_i \in \{0, 1\}$.

- Decryption becomes:

$$m \leftarrow [c]_2 \oplus \left[\left\lfloor \sum_{i=1}^{\Theta} s_i \cdot (y_i \cdot c) \right\rceil \right]_2$$

Squashed decryption

- Alternative equation

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

- Secret-share $1/p$ as a sparse subset sum:

$$1/p = \sum_{i=1}^{\Theta} s_i \cdot y_i + \varepsilon$$

  with random public κ-bit numbers y_i, and sparse secret $s_i \in \{0, 1\}$.

- Decryption becomes:

$$m \leftarrow [c]_2 \oplus \left[\left\lfloor \sum_{i=1}^{\Theta} s_i \cdot (y_i \cdot c) \right\rceil \right]_2$$

Squashed decryption

- Alternative decryption equation:

$$m \leftarrow [c]_2 \oplus \left[\left\lfloor \sum_{i=1}^{\Theta} s_i \cdot z_i \right\rceil\right]_2$$

  where $z_i = y_i \cdot c$ for public y_i's
- Since s_i is sparse with $H(s_i) = \theta$, only $n = \lceil \log_2(\theta + 1) \rceil$ bits of precision for $z_i = y_i \cdot c$ is required
  - With $\theta = 15$, only $n = 4$ bits of precision for $z_i = y_i \cdot c$
- The decryption function can then be expressed as a polynomial of low degree (30) in the s_i's.

Compressing the public-key

- Size of public-key
  - In the squashed scheme, $\Theta = \tilde{\mathcal{O}}(\lambda^5)$ additional elements y_i in the public key, each of size $\kappa = \tilde{\mathcal{O}}(\lambda^5)$ bits.
  - Therefore this gives again a $\tilde{\mathcal{O}}(\lambda^{10})$-bit public key, instead of $\tilde{\mathcal{O}}(\lambda^5)$ in our variant.
- 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.
  - Take $s_1 = 1$ and generate u_1 so that

$$\sum_{i \in S} u_i = x_p \mod 2^{\kappa+1}$$

The decryption circuit

- We must compute:

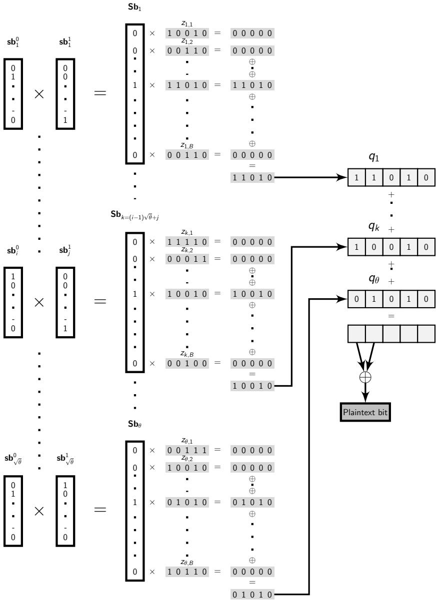
$$m \leftarrow c^* - \left\lfloor \sum_{i=i}^{\Theta} s_i \cdot z_i \right\rceil \mod 2$$

- Trick from Gentry-Halevi:
  - Split the Θ secret key bits into θ boxes of size $B = \Theta/\theta$ each.
  - Then only one secret key bit inside every box is equal to one
- New decryption formula: $m \leftarrow c^* - \left\lfloor \sum_{k=1}^{\theta} \left(\sum_{i=1}^{B} s_{k,i} z_{k,i} \right) \right\rceil_2$
  - The sum:

$$q_k \stackrel{\text{def}}{=} \sum_{i=1}^{B} s_{k,i} z_{k,i}$$

    is obtained by adding B numbers, only one being non-zero.
  - To compute the j-th bit of q_k it suffices to xor all the j-th bits of the numbers $s_{k,i} \cdot z_{k,i}$.

The decryption circuit

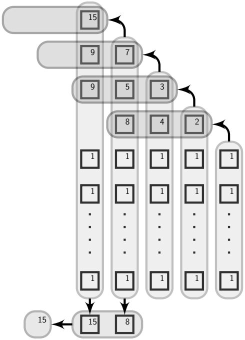


Grade School addition

- The decryption equation is now:

$$m \leftarrow c^* - \left\lfloor \sum_{k=1}^{\theta} q_k \right\rceil \mod 2$$

  - where the q_k's are rational in $[0, 2)$ with n bits of precision after the binary point.



Gentry's Bootstrapping

- The decryption circuit
  - Can now be expressed as a polynomial of small degree d in the secret-key bits s_i, given the $z_i = c \cdot y_i$.

$$m = C_{z_i}(s_1, \ldots, s_\Theta)$$

- To refresh a ciphertext:
  - Publish an encryption of the secret-key bits $\sigma_i = E_{pk}(s_i)$
  - Homomorphically evaluate $m = C_{z_i}(s_1, \ldots, s_\Theta)$, using the encryptions $\sigma_i = E_{pk}(s_i)$
  - We get $E_{pk}(m)$, that is a new ciphertext but possibly with less noise (a "recryption").
  - The new noise has size $\simeq d \cdot \rho$ and is independent of the initial noise.

Constraints on the parameters

- ρ: size of noise
  - $\rho \geq 37$ bits to avoid brute-force attack on the noise
- η: size of p
  - The squashed scheme has a decryption polynomial of degree 30.
  - We must allow for an additional multiplication, so degree $d = 60$
  - $\eta \geq (d+8)\rho = 2516$ bits.
- γ: size of x_i:
  - $\gamma \simeq 12 \cdot 10^6$ bits to avoid lattice attacks
- Public-key size:
  - If we take $\tau = \gamma$, we get a pk size of $\tau \cdot \gamma = \gamma^2 = 1.4 \cdot 10^{14}$ bits. Initial scheme unpractical.
  - We can actually take a much smaller $\tau \simeq 10^4$.

PK size and timings

Instance	λ	ρ	η	γ	pk size	Recrypt
Toy	42	27	1026	$150 \cdot 10^3$	77 KB	0.41 s
Small	52	41	1558	$830 \cdot 10^3$	437 KB	4.5 s
Medium	62	56	2128	$4.2 \cdot 10^6$	2.2 MB	51 s
Large	72	71	2698	$19 \cdot 10^6$	10.3 MB	11 min

RLWE-based Schemes

- Parameters
  - The polynomial ring $R_q = \mathbb{Z}_q[x]/ < x^n + 1 >$, where n is a power of 2.
  - Addition and multiplication of polynomials are performed modulo $x^n + 1$ and prime q.
- Ciphertext $\vec{c} = (c_0, c_1)$ such that

$$c_0 + s \cdot c_1 = 2e + m$$

  - $e \leftarrow \chi$, where χ is a narrow Gaussian noise distribution
  - $c_1 \leftarrow R_q$
  - $s \leftarrow \chi$ is the secret key
  - The message m is in $\mathbb{Z}_2/ < x^n + 1 >$
- Decryption:
  - Compute $m = c_0 + s \cdot c_1 \pmod{x^n + 1, 2}$

RLWE-based Schemes

- Parameters
  - The polynomial ring $R_q = \mathbb{Z}_q[x]/ < x^n + 1 >$, where n is a power of 2.
  - Addition and multiplication of polynomials are performed modulo $x^n + 1$ and prime q.
- Ciphertext $\vec{c} = (c_0, c_1)$ such that

$$c_0 + s \cdot c_1 = 2e + m$$

  - $e \leftarrow \chi$, where χ is a narrow Gaussian noise distribution
  - $c_1 \leftarrow R_q$
  - $s \leftarrow \chi$ is the secret key
  - The message m is in $\mathbb{Z}_2/ < x^n + 1 >$
- Decryption:
  - Compute $m = c_0 + s \cdot c_1 \pmod{x^n + 1, 2}$

RLWE-based Schemes

- Parameters
  - The polynomial ring $R_q = \mathbb{Z}_q[x]/ < x^n + 1 >$, where n is a power of 2.
  - Addition and multiplication of polynomials are performed modulo $x^n + 1$ and prime q.
- Ciphertext $\vec{c} = (c_0, c_1)$ such that

$$c_0 + s \cdot c_1 = 2e + m$$

  - $e \leftarrow \chi$, where χ is a narrow Gaussian noise distribution
  - $c_1 \leftarrow R_q$
  - $s \leftarrow \chi$ is the secret key
  - The message m is in $\mathbb{Z}_2/ < x^n + 1 >$
- Decryption:
  - Compute $m = c_0 + s \cdot c_1 \pmod{x^n + 1, 2}$

Somewhat homomorphic scheme

- Addition of ciphertexts:
  - $\vec{c} = (c_0, c_1)$ with $c_0 + s \cdot c_1 = 2e + m$
  - $\vec{c'} = (c'_0, c'_1)$ with $c'_0 + s \cdot c'_1 = 2e' + m'$
  - $(c_0 + c'_0) + s \cdot (c_1 + c'_1) = 2(e + e') + m + m'$
- Multiplication of ciphertexts $\vec{c}$ and $\vec{c'}$:
  - $(c_0 + s \cdot c_1) \cdot (c'_0 + s \cdot c'_1) = (2e + m) \cdot (2e' + m') = 2e'' + m \cdot m'$
  - $(c_0 + s \cdot c_1) \cdot (c'_0 + s \cdot c'_1) = c_0 \cdot c'_0 + s \cdot (c_1 \cdot c'_0 + c_0 \cdot c'_1) + s^2 \cdot c_1 \cdot c'_1$
  - Define $\vec{c''} = (c''_0, c''_1, c''_2) = (c_0 \cdot c'_0, c_1 \cdot c'_0 + c_0 \cdot c'_1, c_1 \cdot c'_1)$

$$c''_0 + c''_1 \cdot s + c''_2 \cdot s^2 = 2e'' + m \cdot m'$$

  - The ciphertext now has 3 elements
  - The ciphertext size grows exponentially with the multiplicative depth

Somewhat homomorphic scheme

- Addition of ciphertexts:
  - $\vec{c} = (c_0, c_1)$ with $c_0 + s \cdot c_1 = 2e + m$
  - $\vec{c'} = (c'_0, c'_1)$ with $c'_0 + s \cdot c'_1 = 2e' + m'$
  - $(c_0 + c'_0) + s \cdot (c_1 + c'_1) = 2(e + e') + m + m'$
- Multiplication of ciphertexts $\vec{c}$ and $\vec{c'}$:
  - $(c_0 + s \cdot c_1) \cdot (c'_0 + s \cdot c'_1) = (2e + m) \cdot (2e' + m') = 2e'' + m \cdot m'$
  - $(c_0 + s \cdot c_1) \cdot (c'_0 + s \cdot c'_1) = c_0 \cdot c'_0 + s \cdot (c_1 \cdot c'_0 + c_0 \cdot c'_1) + s^2 \cdot c_1 \cdot c'_1$
  - Define $\vec{c''} = (c''_0, c''_1, c''_2) = (c_0 \cdot c'_0, c_1 \cdot c'_0 + c_0 \cdot c'_1, c_1 \cdot c'_1)$

$$c''_0 + c''_1 \cdot s + c''_2 \cdot s^2 = 2e'' + m \cdot m'$$

  - The ciphertext now has 3 elements
  - The ciphertext size grows exponentially with the multiplicative depth

Public-key encryption with RLWE

- To encrypt m
  - One needs a fresh pair $(a, a \cdot s + 2e)$
  - where $a \leftarrow R_q$ and $e \leftarrow \chi$
- Idea from [BV11a]:
  - Given one such pair $(a, b) = (a, a \cdot s + 2e)$, easy to re-randomize and generate as many as needeed.
  - $(a', b') = (av + 2e', bv + 2e'')$ where $v, e' \leftarrow \chi$, $e'' \leftarrow \chi'$
  - $b' = (as+2e)v+2e'' = asv+2(ev+e'') = a's+2(ev+e''-e's)$

RLWE Assumption

- RLWE Assumption
  - Let $(a_i, a_i \cdot s + e_i)$ for $1 \le i \le \ell$ where $\ell = \text{poly}(\lambda)$, $a_i \leftarrow R_a$, $s \leftarrow \chi$, $e_i \leftarrow \chi$.
  - The sequence $(a_i, a_i \cdot s + e_i)$ for $1 \le i \le \ell$ is computationally indistinguishable from (a_i, u_i) where $u_i \leftarrow R_q$.
- Semantic security of $\vec{c} = (c_0, c_1)$ where $c_0 + s \cdot c_1 = 2e + m$
  - $\vec{c} = (-s \cdot c_1 - 2e - m, c_1)$ is computationally indistinguishable from $(u - m, c_1)$, where $u \leftarrow R_q$
  - This implies semantic security.

Conclusion

- Fully homomorphic encryption is a very active research area.
- Main challenge: make FHE pratical !
- Recent developments
  - FHE without bootstrapping (modulus switching) [BGV11]
  - Batch FHE [GHS12]
  - Implementation with homomorphic evaluation of AES [GHS12]