Implementation of homomorphic LWE-based encryption

Jean-Sébastien Coron

Université du Luxembourg

1 Introduction

In [BV11], the authors described a fully homomorphic encryption scheme based on the learning with errors (LWE) assumption. In particular, they introduced a new relinearization technique for performing the ciphertext multiplication for a “somewhat homomorphic” encryption scheme based on LWE.

1. Install the Sage library, available at `http://www.sagemath.org/`. Alternatively, you can use the Sage Cell Server at `https://sagecell.sagemath.org`.
2. Implement the functions below. The implementation of the functions in Section 4.2 and after is optional.
3. Please submit a `.ipynb` file.

2 Basic LWE encryption

We first recall basic lattice-based encryption, starting from the Regev scheme [Reg05]. Let $q \in \mathbb{Z}$ be a prime number. Let $\vec{s} \in \mathbb{Z}$ be the secret-key. An LWE ciphertext for a message $m \in \{0,1\}$ is $\vec{c} \in \mathbb{F}_q^n$ such that

$$\langle \vec{c}, \vec{s} \rangle = e + m \cdot \lfloor q/2 \rceil \pmod q$$

with $s_1 = 1$. For the error e, we can take the binomial distribution χ with parameter κ, for some small κ. One can let $e = h(u) - h(v)$ where $u, v \leftarrow \{0,1\}^\kappa$, where h is the Hamming weight function.

```
def hw(x):
  return sum(x.digits(base=2))

def binom(kappa=2):
  return hw(ZZ.random_element(2^kappa))-hw(ZZ.random_element(2^kappa))
```

Symmetric-key encryption. For simplicity, we first consider a symmetric scheme. We consider a secret-key $\vec{s} = (s_1, \cdots, s_n)$ with random components modulo q, with $s_1 = 1$.

```
def genKey(n=10,kappa=2,nq=8): # nq is the bitsize of q
  ...
  return n,kappa,q,s
```

An LWE ciphertext is a vector of n elements in $\mathbb{Z}_q$. We have taken $s_1 = 1$, so

$$\langle \vec{c}, \vec{s} \rangle = c_1 + \sum_{i=2}^{n} c_i \cdot s_i = e + m \cdot \lfloor q/2 \rceil \pmod q$$

In the code below, we take as input a scaling factor Δ, so that $\langle \vec{c}, \vec{s} \rangle = e + m \cdot \Delta \pmod q$. Therefore, $\Delta = \lfloor q/2 \rceil$ by default.

```
def LWEencSym(mes,q,kappa,s,Delta=None):
  pass
```

Decryption is performed by computing $m = \mathsf{th}(\langle \vec{c}, \vec{s} \rangle \bmod q)$, where $\mathsf{th}(x) = 1$ if $q/4 \le x \le 3q/4$, and 0 otherwise.

```
def th(x,q):
  pass

def LWEdecrypt(c,s,q):
  pass
```

Public-key encryption. To encrypt, one can use a matrix $\mathbf{A} \in \mathbb{F}_q^{m \times n}$ of row vectors $\vec{a}_i \in \mathbb{F}_q^n$, such that $\langle \vec{a}_i, \vec{s} \rangle = e_i$ for $e_i \leftarrow \chi$, for all $1 \le i \le m$. This can be written $\mathbf{A} \cdot \vec{s} = \vec{e} \pmod q$. Therefore, every row of $\vec{A}$ is an LWE encryption of 0.

```
def vecBinom(n,kappa=2):
  return vector([binom(kappa) for i in range(n)])

def genKeyPK(n=10,ell=40,kappa=2,nq=12):
  return kappa,q,A,s
```

To encrypt a message $m \in \{0, 1\}$, one generates a linear combination of the row vectors $\vec{a}_i$:

$$\vec{c} = \left\lfloor \frac{q}{2} \right\rceil \cdot (m, 0, \ldots, 0) + \vec{u} \cdot \vec{A} \pmod q$$

where $\vec{u} \leftarrow \chi^\ell$.

```
def LWEenc(mes,A,kappa,q):
  pass
```

For decryption, we compute:

$$\langle \vec{c}, \vec{s} \rangle = \left\lfloor \frac{q}{2} \right\rceil \cdot m + \vec{u} \cdot \mathbf{A} \cdot \vec{s} = \left\lfloor \frac{q}{2} \right\rceil \cdot m + \langle \vec{u}, \vec{e} \rangle \pmod q$$

For correct decryption, we must have $|\langle \vec{u}, \vec{e} \rangle| < q/4$. We can fix the parameters so that this is the case, except with negligible probability. For a constant κ, the distribution of $\langle \vec{u}, \vec{e} \rangle$ looks like a Gaussian with standard deviation $\mathcal{O}(\sqrt{n})$. Hence we can take $q = \mathcal{O}(\sqrt{n})$. We can take $m = \mathcal{O}(n)$, for a proof of security based on the leftover hash lemma.

3 Homomorphic addition

LWE ciphertexts can be added, with a small increase in the noise.

$$\begin{aligned}
\langle \vec{c}_1, \vec{s} \rangle &= e_1 + m_1 \cdot (q+1)/2 \pmod q \\
\langle \vec{c}_2, \vec{s} \rangle &= e_2 + m_2 \cdot (q+1)/2 \pmod q \\
\langle \vec{c}_1 + \vec{c}_2, \vec{s} \rangle &= e_1 + e_2 + (m_1 + m_2) \cdot (q+1)/2 \pmod q
\end{aligned}$$

```
def LWEadd(c1,c2):
  pass
```

```
def testLWEadd(nt=100):
  kappa,q,A,s=genKeyPK()
  for i in range(nt):
    m1=ZZ.random_element(2)
    m2=ZZ.random_element(2)
    c1=LWEenc(m1,A,kappa,q)
    c2=LWEenc(m2,A,kappa,q)
    c3=LWEadd(c1,c2)
    assert LWEdecrypt(c3,s,q)==(m1+m2)%2
```

4 Homomorphic multiplication

Homomorphic multiplication is more complex. It has three steps:

1. Tensor product
2. Binary decomposition
3. Key switching

4.1 Tensor product

LWE ciphertexts can be multiplied by tensor product.

$$2\langle\vec{c}_1,\vec{s}\rangle\cdot\langle\vec{c}_2,\vec{s}\rangle = 2\left(\sum_{i=1}^{n} c_{1,i}s_i\right)\left(\sum_{i=1}^{n} c_{2,i}s_i\right) = 2(e_1+(q+1)/2\cdot m_1)\cdot(e_2+(q+1)/2\cdot m_2) \pmod q$$

which gives:

$$\sum_{i=1}^{n}\sum_{j=1}^{n} 2c_{1,i}c_{2,j}\cdot s_i s_j = e + m_1 m_2\cdot(q+1)/2 \pmod q$$

for $e = 2e_1e_2 + m_1e_2 + m_2e_1$. Hence $\vec{c}' = (2c_{1,i}\cdot c_{2,j})_{i,j} \in \mathbb{Z}_q^{n^2}$ is a new LWE ciphertext for the secret-key $\vec{s}' = (s_i\cdot s_j)_{i,j} \in \mathbb{Z}_q^{n^2}$, with

$$\langle\vec{c}',\vec{s}'\rangle = e + m_1m_2\cdot(q+1)/2 \pmod q$$

Therefore, the bitsize of the noise has roughly doubled. However, we get a ciphertext with n^2 components instead of n.

```
def tensorprod(v1,v2):
  return vector(v1.tensor_product(v2).list())

def LWEmult_tens(c1,c2,q):
  pass

def testLWEmult_tens(nt=100):
  kappa,q,A,s=genKeyPK()
  sp=tensorprod(s,s)
  for i in range(nt):
    m1=ZZ.random_element(2)
    m2=ZZ.random_element(2)
    c1=LWEenc(m1,A,kappa,q)
    c2=LWEenc(m2,A,kappa,q)
    c3=LWEmult_tens(c1,c2,q)
    assert LWEdecrypt(c3,sp,q)==m1*m2
```

The main problem is that the product ciphertext has now n^2 components instead of n. We would like to get back to n components. The first step is to get a ciphertext with binary components only, using an expanded secret-key. In the second step, we apply a key switching.

4.2 Binary decomposition

We want to have a ciphertext with binary components only. This is easy using binary decomposition. For any $0 \leq a, b < q$, we have, using $n_q = \lceil \log_2 q \rceil$:

$$\begin{aligned} a \cdot b &= \sum_{i=0}^{n_q - 1} a_i \cdot 2^i b \pmod q \\ &= \langle \mathsf{BitDecomp}(a), \mathsf{PowerOf2}(b) \rangle \end{aligned}$$

where $\mathsf{BitDecomp}(a) = (a_0, \ldots, a_{n_q-1})$ and $\mathsf{PowerOf2}(b) = (b, 2^1 b, \ldots, 2^{n_q-1} b)$. We can extend BitDecomp and PowerOf2 to vectors by applying it component wise, and flattening the resulting matrix into a vector.

Therefore, given $\vec{c} \in \mathbb{Z}_q^m$ and $\vec{s} \in \mathbb{Z}_q^m$, we can let $\vec{c}' = \mathsf{BitDecomp}(\vec{c})$, and $\vec{s}' = \mathsf{PowerOf2}(\vec{s})$, and we get:

$$\langle \vec{c}', \vec{s}' \rangle = \langle \mathsf{BitDecomp}(\vec{c}), \mathsf{PowerOf2}(\vec{s}) \rangle = \langle \vec{c}, \vec{s} \rangle$$

Therefore, we get a new ciphertext $\vec{c}'$ with binary components only, which encrypts the same message under the new secret key $\vec{s}'$, as the original $\vec{c}$ under $\vec{s}$.

```
def bitdecomp(v,q):
  pass

def powerof2(v,q):
  pass
```

4.3 Key switching

We now explain how to switch keys. Given a binary ciphertext $\vec{c} \in \{0,1\}^m$ under key $\vec{s} \in \mathbb{Z}_q^m$ and another key $\vec{s}' \in \mathbb{Z}_q^n$, we show how to get a new ciphertext $\vec{c}' \in \mathbb{Z}_q^n$ encrypting the same message m under the new key $\vec{s}'$. We start from a ciphertext $\vec{c}$ under $\vec{s}$:

$$u = \langle \vec{c}, \vec{s} \rangle = \sum_{i=1}^{m} c_i \cdot s_i \pmod q$$

We consider LWE pseudo-encryptions $\vec{t}_i$ of each s_i under the new key $\vec{s}'$:

$$\langle \vec{t}_i, \vec{s}' \rangle = f_i + s_i \pmod q$$

for some small errors f_i. This enables to write:

$$u = \sum_{i=1}^{m} c_i \left(\langle \vec{t}_i, \vec{s}' \rangle - f_i \right) = \left\langle \sum_{i=1}^{m} c_i \vec{t}_i, \vec{s}' \right\rangle - \sum_{i=1}^{m} c_i \cdot f_i \pmod q$$

Therefore, we can define a new ciphertext

$$\vec{c}' = \sum_{i=1}^{m} c_i \vec{t}_i \pmod q$$

and we get

$$\langle \vec{c}', \vec{s}' \rangle = \langle \vec{c}, \vec{s} \rangle + f \pmod q$$

for a small additional error f, because the c_i's are binary. Therefore, the two ciphertexts encrypt the same message.

```python
def switchKey(s,sp,q,kappa):
  pass

swenc=switchKey(powerof2(tensorprod(s,s),q),s,q,kappa)

def LWEmult(c1,c2,swenc,q):
    pass
```

References

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. Efficient fully homomorphic encryption from (standard) lwe. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 97–106. IEEE, 2011.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *Proceedings of the 37th Annual ACM Symposium on Theory of Computing, Baltimore, MD, USA, May 22-24, 2005*, pages 84–93. ACM, 2005.