published on November 16, 2020

Homomorphic Encryption: a Toy Implementation in Python

Motivation: We made this blog post as self-contained as possible, even though it was initially thought as a follow-up of this tutorial given by OpenMined. The starting point of our Python implementation is this github gist, which follows the Homomorphic Encryption scheme from [FV12]. The motivation behind our implementation was for us to understand in detail the two techniques of [FV12] used for ciphertext multiplication, namely *relinearization* and *modulus-switching*. This essential operation of ciphertext multiplication was missing in the previous implementation. We thought we might share this understanding through a blog post as well since it may be of interest to anyone using the [FV12] scheme in TenSeal or Seal libraries.

> **Disclaimer:** Our toy implementation is not meant to be secure or optimized for efficiency. We did it to better understand the inner workings of the [FV12] scheme, so you can use it as a learning tool. Our full implementation can be found here.

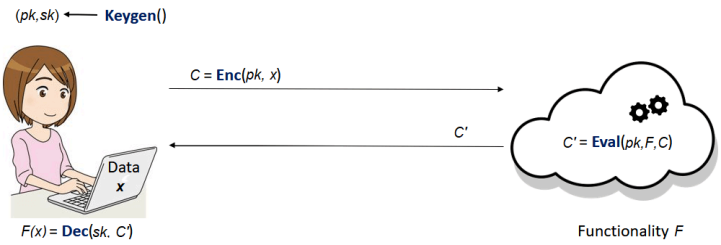
Curious about how to work with data you can't see? In the first part of this blog post we are going to broadly explain what Homomorphic Encryption is and some closely related concepts. In the second part we will follow an example of such a scheme, namely the [FV12] scheme, and discuss some of the details of our implementation.

1. What is Homomorphic Encryption?

Homomorphic Encryption (HE) enables a user to perform meaningful computations on sensitive data **while ensuring the privacy of the data**. This may sound paradoxical to anyone who has ever worked with encrypted data before: if you want to perform useful computations on the encrypted data (e.g. encrypted under classical algorithms like AES), you need to decrypt it first. But once decryption takes place, the privacy of the data is compromised. So how is it possible for HE to overcome this seeming contradiction? 🔮 Well, the solution is highly non-trivial, as it took the cryptographic community more than 30 years to come up with a construction. The first solution was proposed by Craig Gentry in 2009 and was of theoretical interest only. Since then, a lot of research has been done (e.g. [BGV11], [FV12], [CKKS16], [FHEW], [TFHE], [GSW13]) to make these ideas more practical. By the end of this post you should have a basic understanding of how such a construction may work.

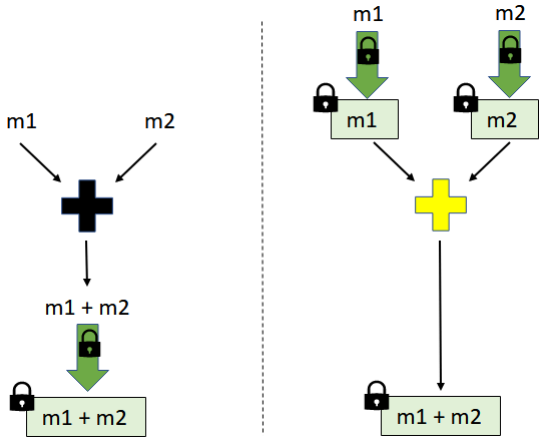
Besides the traditional encryption (Enc), decryption (Dec) and key generation (Keygen) algorithms, an HE scheme also uses *an evaluation algorithm* (Eval). **This is the distinguishing feature that makes computations on encrypted data possible.** 💥 Let's consider the following example: Alice holds some personal information $\boldsymbol{x}$ (e.g. her medical records and her family's medical history). There is also a company that makes very good predictions based on this kind of information, using a refined model, expressed as the functionality $\boldsymbol{F}$ (e.g. a well chosen machine learning model). On one hand, Alice is very interested in these predictions but is also reluctant to trust the company with her sensitive information. On the other hand, the company can't just give their model to Alice to make the predictions herself. A solution using Homomorphic Encryption is given in the picture below. Some important things to notice are:

- Alice sends her data **encrypted**, so the company never learns anything about $\boldsymbol{x}$.
- Computing on the encrypted data $\boldsymbol{C}$ does **not involve** Alice's secret key $\boldsymbol{sk}$. Only her public key $\boldsymbol{pk}$ is used.
- To obtain $\boldsymbol{C'}$ as the encryption of $\boldsymbol{F(x)}$, the evaluation algorithm uses the description of $\boldsymbol{F}$ to do computations on $\boldsymbol{C}$ (which encrypts $\boldsymbol{x}$).
- By using her secret key, $\boldsymbol{sk}$, Alice manages to recover the information that interests her, namely $\boldsymbol{F(x)}$.

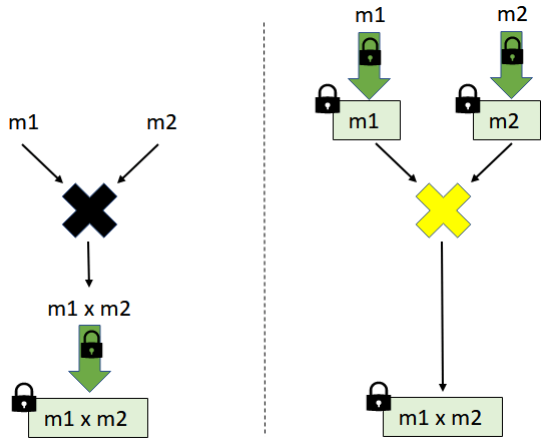


A closer look at the Eval algorithm 🔎

All the existing HE constructions are *homomorphic* with respect to two basic operations: some kind of *addition* and some kind of *multiplication* (e.g. $+$ and $\times$ over the integers or the binary operations XOR and AND, etc.). What we mean is that the scheme allows the efficient computation of c_{add} from the individual ciphertexts $c_1 = \mathsf{Enc}(pk, m_1)$ and $c_2 = \mathsf{Enc}(pk, m_2)$ such that the decryption of c_{add} yields $m_1 + m_2$.

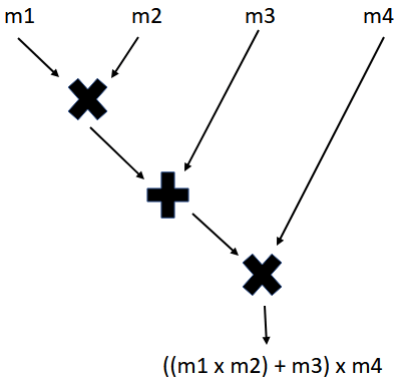


Analogously, the ciphertext c_{mul} corresponding to multiplication, that decrypts to $m_1 \times m_2$, is efficiently computable from the individual ciphertexts $c_1 = \mathsf{Enc}(pk, m_1)$ and $c_2 = \mathsf{Enc}(pk, m_2)$, respectively.



▶ There are classical examples of encryption schemes that are homomorphic with respect to only *one* operation.

All the existing HE schemes support only two types of computations on the encrypted data: some forms of addition and multiplication. This means that the Eval algorithm works only for functionalities F that can be expressed using additions ($+$) and multiplications ($\times$). Another way of saying this is that HE schemes support only arithmetic circuits with addition/multiplication gates. Below we can view as an arithmetic circuit the functionality $F(m_1, m_2, m_3, m_4) = m_1 \times m_2 \times m_4 + m_3 \times m_4$.



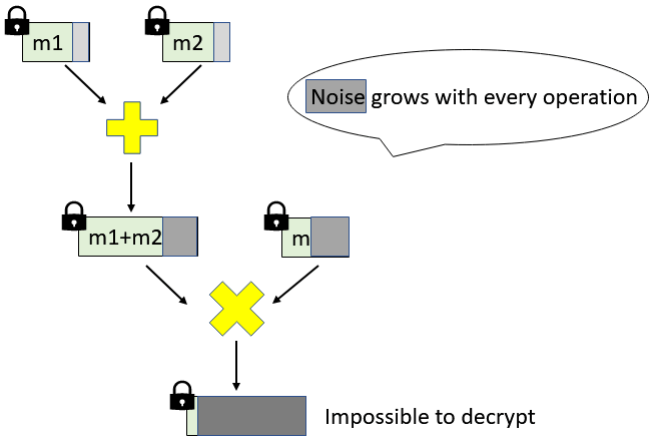
Why focus on homomorphisms with respect to *two* operations?

In principle, any functionality can be expressed using only two basic operations. For example, any binary circuit can be obtained using NAND gates exclusively. In turn, a NAND operation consists of one addition and one multiplication: $\text{NAND}(a, b) = a \times b + 1 \bmod 2$, for any bits $a, b \in \{0, 1\}$.

> 💡 Therefore it is enough to have an HE scheme that supports an unlimited number of additions and multiplications to be able to make any efficient computation we can think of on encrypted data.

Homomorphic Encryption and "noisy" ciphertexts

The most practical HE constructions rely on the hardness of the Ring Learning With Errors (RLWE) problem for their security, as is the case with many lattice-based cryptographic constructions. The inherent **"noise"** of the RLWE problem is inherited by all the schemes that are based on it. In particular, this "noise" element is present in every HE ciphertext and has a great impact on the parameters of the scheme.



The noise grows with every addition or multiplication we do on the ciphertexts. This is very relevant as decryption stops working correctly once the noise exceeds a certain threshold.

Because of this phenomenon, the number of multiplications and additions that can be carried out correctly on the ciphertext is limited. 💥 The parameters of such a scheme can be generated such that it can handle a minimum number of operations. But this minimum number must be decided in advance to set the parameters accordingly. We usually call such a scheme *Somewhat Homomorphic Encryption* (SHE) scheme. When the construction allows an unbounded number of operations, we call such a scheme *Fully Homomorphic Encryption* (FHE). Even though we are not going to discuss it any further, we have to mention that it's possible to obtain FHE from SHE. In fact, Gentry showed how to transform any SHE (that can homomorphically evaluate its own decryption circuit) to FHE, through a computationally expensive process called *bootstrapping*. For applications that don't require many homomorphic evaluations SHE is preferred, as we want to avoid the computational overhead of the bootstrapping.

2. A SHE scheme example

For the remaining of this blog post we will try to make the concepts that we have already presented more concrete, by discussing a *toy implementation* of the SHE scheme construction of [FV12]. Our main goal is to understand how *relinearization* and *modulus-switching* are used to obtain ciphertext multiplication.

> **Notations:** For an integer q, by $\mathbb{Z}_q$ we mean the set $\{0, 1, \ldots, q-1\}$. We denote by $[a]_q$ the remainder of a *modulo* q. For example $[18]_7 = 4$. When rounding to the nearest integer, we use $\lfloor \cdot \rceil$. The basic elements we work with will not be integers, but merely *polynomials with integer coefficients*. We also work with *noisy polynomials*, whose coefficients are sampled according to some error distribution χ. We bound such errors by their largest absolute value of their coefficients, denoted as $\| \cdot \|$.

Quick recap on working with polynomials

The HE scheme we are going to describe deals with *adding and multiplying polynomials*. Here we present a quick example of how to work with polynomials, so that we all have the same starting point. If you already know how to do this, you can skip it.

First thing, let's add and multiply polynomials *modulo some polynomial* f. This "modulo f" thing simply means that we add and multiply the polynomials in the usual way, but we take the remainders of the results when divided by f. When we do these additions and multiplications $\bmod f$, we sometimes say in a fancy way that we are working in the *ring* $\mathbb{Z}[x]/(f)$ of reduced polynomials. 💍

Let's take $f = x^4 + 1$. If we look at $p(x) = x^5$, then $p(x) = x \cdot (x^4 + 1) - x$. Therefore, when taking the reminder we get $p(x) = -x \bmod f$. For faster computations $\bmod f$ you can use this trick: when making $\bmod f$, simply replace x^4 by -1, x^5 by $-x$ and so on.

Let's consider two polynomials $a(x) = x^3 + x^2 + 7$ and $b(x) = x^2 + 11x$. Then:

$$a(x) + b(x) \bmod f = x^3 + 2x^2 + 11x + 7 \bmod f.$$

Here nothing special happened. Let's multiply them:

$$\begin{aligned} a(x) \cdot b(x) \bmod f &= (x^3 + x^2 + 7) \cdot (x^2 + 11x) \bmod f \\ &= x^5 + 11x^4 + x^4 + 11x^3 + 7x^2 + 77x \bmod f \\ &= -x - 11 - 1 + 11x^3 + 7x^2 + 77x \bmod f \\ &= 11x^3 + 7x^2 + 76x - 12 \bmod f. \end{aligned}$$

These operations are implemented here and make use of the cool Numpy library:

```python
import numpy as np
from numpy.polynomial import polynomial as poly

#------Functions for polynomial evaluations mod poly_mod only------
def polymul_wm(x, y, poly_mod):
    """Multiply two polynomials
    Args:
        x, y: two polynomials to be multiplied.
        poly_mod: polynomial modulus.
    Returns:
        A polynomial in Z[X]/(poly_mod).
    """
    return poly.polydiv(poly.polymul(x, y), poly_mod)[1]
def polyadd_wm(x, y, poly_mod):
    """Add two polynomials
    Args:
        x, y: two polynomials to be added.
        poly_mod: polynomial modulus.
    Returns:
        A polynomial in Z[X]/(poly_mod).
    """
    return poly.polydiv(poly.polyadd(x, y), poly_mod)[1]
```

Now let's go one step further and see how we perform these operations of polynomials not just modulo f, *but also modulo some integer* q. As you might

always take integer values from 0 to $q-1$. This time, we say that we are working the ring of reduced polynomials $\mathbb{Z}_q[x]/(f)$. 💍

Let's take the previous example, $f=x^4+1$, $a(x)=x^3+x^2+7$, $b(x)=x^2+11x$ and consider $q=5$. We can think of a and b as already taken $\bmod f$. If we take them further modulo q, then $[a(x)]_q=x^3+x^2+2$ and $[b(x)]_q=x^2+x$. Moreover,

$$[a(x)+b(x)]_q=x^3+x^2+2+x^2+x=x^3+2x^2+x+2$$

and

$$\begin{aligned}[a(x)\cdot b(x)]_q&=(x^3+x^2+2)\cdot(x^2+x)\\&=x^5+x^4+x^4+x^3+2x^2+2x\\&=-x-1-1+x^3+2x^2+2x\\&=x^3+2x^2+x+3\end{aligned}$$

where at the last but one equality we performed modulo $f=x^4+1$ and at the last one, modulo $q=5$.

These operations already mentioned are **`polyadd`** and **`polymul`** implemented here.

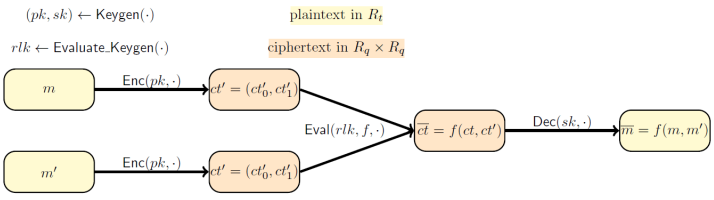
The Fan-Vercauteren ([FV12]) scheme

Next, we recall the basic (**Keygen**, **Enc**, **Dec**) algorithms of the [FV12] scheme. These (almost identical) algorithms have already been described here, but for the sake of completeness, we present them as well. Then we will explain in detail the core of the **Eval** algorithm: the addition and multiplication of the ciphertexts. *Spoiler alert*: We will primarily focus on the two *Relinearization* techniques that enable ciphertext multiplication.

> Let n be power of 2. We call a positive integer t the *plaintext modulus* and a positive integer q as the *ciphertext modulus*. We set $\Delta=\lfloor q/t\rfloor$. The scheme involves adding and multiplying polynomials in $R_t=\mathbb{Z}_t[x]/(x^n+1)$, on the plaintext side, and adding and multiplying polynomials in $R_q=\mathbb{Z}_q[x]/(x^n+1)$, on the ciphertext side. We also denote by R the ring $\mathbb{Z}[x]/(x^n+1)$.

> ! **Disclaimer**: From now on all polynomial operations are assumed to be mod x^n+1, even if we don't mention it every time.

Here is the **high level of the scheme**:



➡️ **Keygen**: The *secret key* sk is a secret binary polynomial s in R, i.e. its coefficients are either 0 or 1. The *public key* pk is created as follows: we sample a uniformly over R_q and an error e according to some error distribution χ over R and output $pk=([-(a\cdot s+e)]_q,a)\in R_q\times R_q$.

> Notice that hardness of the RLWE problem prevents the computation of the secret s from the public key.

The way we generate the uniform polynomials and the binary polynomials is implemented as **`gen_uniform_poly`** and as **`gen_binary_poly`** respectively. The error distribution χ is usually taken as a discretized variant of the *Normal distribution* over $\mathbb{Z}^n$, and is implemented as **`gen_normal_poly`**. They can be found here.

```
1 | def keygen(size, modulus, poly_mod, std1):
```

```
        modulus: coefficient modulus.
        poly_mod: polynomial modulus.
        std1: standard deviation of the error.
    Returns:
        Public and secret key.
    """
    s = gen_binary_poly(size)
    a = gen_uniform_poly(size, modulus)
    e = gen_normal_poly(size, 0, std1)
    b = polyadd(polymul(-a, s, modulus, poly_mod), -e, modulus, poly_mod)
    return (b, a), s
```

➡️ **Enc**: To encrypt a *plaintext* $m \in R_t$, we let $pk = (pk_0, pk_1)$, sample u, e_1, e_2 according to χ over R and output the *ciphertext*

$$\text{Enc}(pk, m) = ([pk_0 \cdot u + e_1 + \Delta \cdot m]_q, [pk_1 \cdot u + e_2]_q) \in R_q \times R_q$$

> Due to the RLWE assumption, the ciphertexts "look" uniformly random to a possible attacker, so they don't reveal any information about the plaintext.

In the piece of code below, the message we want to encrypt, m, is an integer vector of length at most n, with entries in the set $\{0, 1, \ldots, t-1\}$. Before we encode it as a polynomial in $m \in R_t$, we pad it with enough zeros to make it a length n vector.

```
def encrypt(pk, size, q, t, poly_mod, m, std1):
    """Encrypt an integer vector pt.
    Args:
        pk: public-key.
        size: size of polynomials.
        q: ciphertext modulus.
        t: plaintext modulus.
        poly_mod: polynomial modulus.
        m: plaintext message, as an integer vector (of length <= size) with entr
    Returns:
        Tuple representing a ciphertext.
    """
    m = np.array(m + [0] * (size - len(m)), dtype=np.int64) % t
    delta = q // t
    scaled_m = delta * m
    e1 = gen_normal_poly(size, 0, std1)
    e2 = gen_normal_poly(size, 0, std1)
    u = gen_binary_poly(size)
    ct0 = polyadd(
        polyadd(
            polymul(pk[0], u, q, poly_mod),
            e1, q, poly_mod),
        scaled_m, q, poly_mod
    )
    ct1 = polyadd(
        polymul(pk[1], u, q, poly_mod),
        e2, q, poly_mod
    )
    return (ct0, ct1)
```

➡️ **Dec**: Given a ciphertext $ct = \text{Enc}(pk, m) = (ct_0, ct_1)$, we decrypt it using the secret key $sk = s$ as follows:

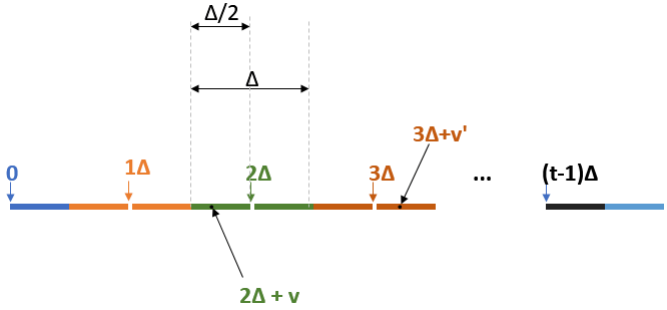
$$\text{Dec}(sk, ct) = \left[\left\lfloor \frac{t \cdot [ct_0 + ct_1 \cdot s]_q}{q} \right\rceil\right]_t \in R_t$$

⛔ Let's stop for a bit and check together how the **Dec** algorithm works. The intuition behind it is that $pk_0 + pk_1 \cdot sk$ is "small". This implies that $ct_0 + ct_1 \cdot sk$ is "close" to the scaled message $\Delta \cdot m$. To recover the message m, we get rid of Δ $(\approx q/t)$ and then apply rounding to shave off the "small" noise. Let's check that this intuition actually works.

First, we set the notation $ct(s) := ct_0 + ct_1 \cdot s$, which we'll frequently use for the rest of the post. If we perform this computation, we will end up getting *a noisy scaled variant of the plaintext*, namely $\Delta \cdot m + v$!

▶ You can click here to see why this happens.

Because we always have $\|\Delta \cdot m + v\| < q$, reducing it $\bmod q$ has no effect (e.g. $[4]_7 = 4$). As long as the noise $\|v\| < \Delta/2$, we can always recover the correct m.



For example, in the picture above, any green point will decrypt to 2 when we scale it by t/q $(\approx 1/\Delta)$ and round it. Analogously, any dark-brown point will decrypt to 3.

We can implement this evaluation, as **`scaled_pt`** below, by performing polynomial operations in R_q. You will see that this equation,

$$[ct(s)]_q = [ct_0 + ct_1 \cdot s]_q = \Delta \cdot m + v,$$

which we will call *the decryption equation*, becomes really useful in deriving ways of computing on ciphertexts. Here we provide the code for the **Dec** algorithm:

```python
def decrypt(sk, q, t, poly_mod, ct):
    """Decrypt a ciphertext.
    Args:
        sk: secret-key.
        size: size of polynomials.
        q: ciphertext modulus.
        t: plaintext modulus.
        poly_mod: polynomial modulus.
        ct: ciphertext.
    Returns:
        Integer vector representing the plaintext.
    """
    scaled_pt = polyadd(
        polymul(ct[1], sk, q, poly_mod),
        ct[0], q, poly_mod
    )
    decrypted_poly = np.round(t * scaled_pt / q) % t
    return np.int64(decrypted_poly)
```

Homomorphic operations of [FV12]

As explained in the first part, the **Eval** algorithm works only for functionalities that can be expressed using addition $(+)$ or multiplication $(\times)$.

Let's take two ciphertexts $ct = \mathsf{Enc}(pk, m)$ and $ct' = \mathsf{Enc}(pk, m')$. We want to see how to construct ciphertexts that decrypt both the addition, $m + m'$, and the multiplication, $m \cdot m'$. Also, keep in mind that when performing operations on m and m', we are actually doing them *modulo* $x^n + 1$ and *modulo* t, since these are the plaintext operations from $R_t = \mathbb{Z}_t[x]/(x^n + 1)$.

Let's write the decryption equations of ct and ct':

$$[ct(s)]_q = \Delta \cdot m + v \text{ and } [ct'(s)]_q = \Delta \cdot m' + v'.$$

➡️ **Addition:** If we simply add the decryption equations, we get

$$[ct(s)]_q + [ct'(s)]_q = \Delta \cdot (m + m') + v + v'.$$

But wait a sec, we need to decrypt to $m_1 + m_2$ modulo t! Notice that $m + m' = t \cdot \epsilon + [m + m']_t$, for some binary polynomial ϵ. Using the notation $r_t(q) := q - \Delta \cdot t$ (this is just the remainder of q divided by t) we get:

$$[ct(s) + ct'(s)]_q = \Delta \cdot [m + m']_t + v_{add} \bmod q,$$

where $v_{add} = v + v' - r_t(q) \cdot \epsilon$.

▶ Click here if you want to see why this follows.

This suggests to set the new ciphertext as $c_{add} = (ct_0 + ct'_0,\ ct_1 + ct'_1)$,

$c_{add} = ct + ct'$ decrypts to the sum, $[m + m']_t$, as long as the new "noise", v_{add}, is smaller than $\Delta/2$.

💡 The **noise growth** for **addition** is quite slow as $\|v_{add}\| < \|v\| + \|v'\| + t < 2B + t$, where B is an upper bound on the "noise" of the ciphertexts that were added. This means we can probably do many additions before decryption stops working.

> To add ciphertexts, it seems like we only need to add polynomials in R_q. So, ciphertext addition is a piece of cake 🍰.

```python
def add_cipher(ct1, ct2, q, poly_mod):
    """Add a ciphertext and a ciphertext.
    Args:
        ct1, ct2: ciphertexts.
        q: ciphertext modulus.
        poly_mod: polynomial modulus.
    Returns:
        Tuple representing a ciphertext.
    """
    new_ct0 = polyadd(ct1[0], ct2[0], q, poly_mod)
    new_ct1 = polyadd(ct1[1], ct2[1], q, poly_mod)
    return (new_ct0, new_ct1)
```

➡️ **Multiplication:** Producing a new ciphertext that decrypts the product of the two messages is not that easy. But we should still try 🤞. The first idea that comes to mind is to simply multiply the decryption equations. It worked for addition, so maybe it works here as well.

$$ct(s) \cdot ct'(s) = \Delta^2 \cdot mm' + \Delta \cdot (mv' + m'v) + vv'. \quad (1)$$

If we scale by t/q to get rid of one Δ, we get something that looks like what we want.

$$\frac{t}{q} \cdot ct(s) \cdot ct'(s) \approx \Delta \cdot mm' + (mv' + m'v)$$

It seems we are on the right track. Let's examine these expressions further. Recall the notations $ct(s) = ct_0 + ct_1 \cdot s$ and $ct'(s) = ct'_0 + ct'_1 \cdot s$. Both of them are *linear* in s, but their multiplication is *quadratic* in s:

$$ct \cdot ct'(s) = ct_0 \cdot ct'_0 + (ct_0 \cdot ct'_1 + ct_1 \cdot ct'_0)s + ct_1 \cdot ct'_1 s^2,$$

which we will write, in short, as $ct \cdot ct'(s) = c_0^{\times} + c_1^{\times} \cdot s + c_2^{\times} \cdot s^2$.

But what about the right hand side? Keep in mind that we work with plaintexts $m, m' \in R_t$, so we should take mm' with *coefficients modulo* t. Therefore, we can apply the same trick as we did for addition: we divide by t and write $mm' = tr_m + [mm']_t$, where r_m is an integer polynomial. Skipping a lot of details, we end up with:

$$t/q \cdot ct \cdot ct'(s) = \Delta \cdot [mm']_t + u_2$$

for u_2 a polynomial with *rational* coefficients. Looks like we're getting closer to obtaining the decryption equation. We can now write the original expression (1) as:

$$t/q \cdot c_0^{\times} + t/q \cdot c_1^{\times} \cdot s + t/q \cdot c_2^{\times} \cdot s^2 = \Delta \cdot [mm']_t + u_2$$

Hm.. these coefficients look like *rational polynomials*. Recall that the ciphertext has *integer polynomials* as elements. So we round each coefficient appearing in the left hand side to their nearest integers and then reduce the whole equation modulo q:

$$[\lfloor t/q \cdot c_0^{\times} \rceil]_q + [\lfloor t/q \cdot c_1^{\times} \rceil]_q \cdot s + [\lfloor t/q \cdot c_2^{\times} \rceil]_q \cdot s^2 = \Delta \cdot [mm']_t + u_3 \quad (2)$$

where u_3 is a "small" integer polynomial, that represents the "noise" after one multiplication.

💡 The **noise growth** for **multiplication** grows a lot faster: $\|u_3\| \leq 2 \cdot n \cdot t \cdot B \cdot (2n+1) \cdot (n+1) + 8t^2 \cdot n^2$, where B is an upper bound for the "noise" of the ciphertexts that were multiplied. We refer the enthusiastic reader for more details to [[FV12] Lem. 2].

> Phew, seems like we are done: we can consider as a ciphertext decrypting to $[mm']_t$ the tuple of scaled and rounded coefficients mod q from left hand side of (2), denoted by (c_0, c_1, c_2). Of course, for a correct decryption, u_3 should have small enough coefficients. You can see below how these coefficients are computed in Python:

```python
def multiplication_coeffs(ct1, ct2, q, t, poly_mod):
    """Multiply two ciphertexts.
        Args:
            ct1: first ciphertext.
            ct2: second ciphertext
            q: ciphertext modulus.
            t: plaintext modulus.
            poly_mod: polynomial modulus.
        Returns:
            Triplet (c0,c1,c2) encoding the multiplied ciphertexts.
        """

    c_0 = np.int64(np.round(polymul_wm(ct1[0], ct2[0], poly_mod) * t / q)) % q
    c_1 = np.int64(np.round(polyadd_wm(polymul_wm(ct1[0], ct2[1], poly_mod), pol
    c_2 = np.int64(np.round(polymul_wm(ct1[1], ct2[1], poly_mod) * t / q)) % q
    return c_0, c_1, c_2
```

But, as a popular movie character would say, "**Houston, we have a problem**". This tuple of coefficients has size 3, **not 2 as the usual ciphertext**. Moreover, the size of such tuple will grow linearly in the number of further multiplications performed on the ciphertexts. In order to restore the size of the ciphertext as 2, we will make use of the so called *relinearization technique*. 💥

Relinearization

> 💡 The idea of **Relinearization** is to reduce the triplet (c_0, c_1, c_2) to a ciphertext pair $(c_0', c_1') \in R_q \times R_q$ that recovers $[mm']_t$ when decrypted with the usual decryption algorithm. We would like to produce a pair (c_0', c_1'), without using the secret s, such that: $[c_0' + s \cdot c_1']_q = [c_0 + c_1 \cdot s + c_2 \cdot s^2 + r]_q$, where r is a "small" error. The correct decryption will be possible, as the "small" error r will vanish because of the rounding in decryption.

As the name suggests it, we transform the degree 2 polynomial, $c_0 + c_1 \cdot s + c_2 \cdot s^2$ into a linear polynomial, i.e. of degree 1. This involves giving extra info about s^2. Using a special public key, called *relinearization key*, we can *linearize* $c_2 \cdot s^2$ (up to some small error) as

$$[c_{20} + c_{21} \cdot s]_q = [c_2 \cdot s^2 + e_{relin}]_q.$$

Therefore, by Equation (2), we can get a standard ciphertext pair as

$$c_{mul} = (c_0 + c_{20}, c_1 + c_{21}),$$

that correctly decrypts to $[mm']_t$, as you can see below:

$$[c_0 + c_{20} + s \cdot (c_1 + c_{21})]_q = [c_0 + c_1 \cdot s + c_2 \cdot s^2 + e_{relin}]_q = \Delta \cdot [mm']_t + v$$

where $v_{mult} = u_3 + e_{relin}$. Therefore, c_{mul} decrypts correctly to $[mm']_t$ if $\|v_{mult}\|$ is less than $\Delta/2$. So yay! we finally know how to get a ciphertext encoding multiplication!

Different versions of relinearization

To complete this discussion, we need to see how to construct the linear approximation of $c_2 \cdot s^2$ and find out what the relinearization key is about. For this, we will go a bit deeper into (technical) details. Don't panic, we'll take you step by step. 😄

We are going to present two versions of linearizing $c_2 \cdot s^2$. First, keep in mind

version of s^2.

Let's think of the following situation: say we include in the public key the following *relinearization* key:

$$rlk = (rlk_0, rlk_1) = ([-(a \cdot s + e) + s^2]_q, a),$$

for some uniform a in R_q and a small error e.

❗ Intuitively, the secret s^2 is hidden by something that looks like an RLWE sample. Notice that,

$$rlk_0 + rlk_1 \cdot s = s^2 + e.$$

To obtain the approximation of $c_2 \cdot s^2$ we should multiply the above expression by c_2. By doing so, we end up with the rather large-norm term $c_2 \cdot e$, due to the size of the coefficients of c_2. We cannot allow such a large "noise" as it will interfere with decryption. To avoid this "noise" blow-up we will employ two techniques described below.

💥 Relinearization: Version 1

One strategy is to use base T decomposition of the coefficients of c_2 to slice c_2 into components of small norm. To do this, we pick a base T and write each coefficient of c_2 in this base. Recall that c_2 is a integer polynomial, modulo $x^n + 1$, so of degree at most $n - 1$. If we write c_2 as a polynomial:

$$c_2(x) = c_2[0] + c_2[1] \cdot x + \ldots + c_2[n-1] \cdot x^{n-1}$$

then we can decompose each coefficient $c_2[i]$ in base T. Notice that since c_2 has coefficients in $[0, q-1]$, the maximum power appearing in these representations is T^ℓ, where $\ell = \lfloor \log_T(q) \rfloor$. For base decomposition we use the function **`int2base`**:

```python
def int2base(n, b):
    """Generates the base decomposition of an integer n.
    Args:
        n: integer to be decomposed.
        b: base.
    Returns:
        array of coefficients from the base decomposition of n
        with the coeff[i] being the coeff of b ^ i.
    """
    if n < b:
        return [n]
    else:
        return [n % b] + int2base(n // b, b)
```

The relinearization key, rlk in this version, consists of masked variants of $T^i \cdot s^2$, instead of s^2. More precisely, for $0 \leq i \leq \ell$, this is defined as follows:

$$(rlk_0[i], rlk_1[i]) = ([-(a_i \cdot s + e_i) + T^i \cdot s^2]_q, a_i)$$

for a_i chosen uniformly in R_q and e_i chosen according to the distribution χ over R (yep, same error distribution as in the description of the scheme). Below you can find the implementation of the function that generates the evaluation (relinearization) key $(rlk_0[i], rlk_1[i])_{0 \leq i \leq \ell}$.

```python
def evaluate_keygen_v1(sk, size, modulus, T, poly_mod, std2):
    """Generate a relinearization key using version 1.
        Args:
            sk: secret key.
            size: size of the polynomials.
            modulus: coefficient modulus.
            T: base.
            poly_mod: polynomial modulus.
            std2: standard deviation for the error distribution.
        Returns:
            rlk: relinearization key.

        """
    n = len(poly_mod) - 1
    l = np.int(np.log(modulus) / np.log(T))
    rlk0 = np.zeros((l + 1, n), dtype=np.int64)
    rlk1 = np.zeros((l + 1, n), dtype=np.int64)
    for i in range(l + 1):
```

```
        secret_part = T ** i * poly.polymul(sk, sk)
        b = np.int64(polyadd(
        polymul_wm(-a, sk, poly_mod),
        polyadd_wm(-e, secret_part, poly_mod), modulus, poly_mod))

        b = np.int64(np.concatenate( (b, [0] * (n - len(b)) ) )) # pad b
        a = np.int64(np.concatenate( (a, [0] * (n - len(a)) ) )) # pad a

        rlk0[i] = b
        rlk1[i] = a
    return rlk0, rlk1
```

Now, given rlk, let's look at how we compute the linear approximation of $c_2 \cdot s^2$. Let the polynomials $c_2(i)$ be the base T decomposition of c_2, such that:

$$c_2 = \sum_{i=0}^{\ell} c_2(i) \cdot T^i.$$

We can get the linear approximation given by (c_{20}, c_{21}), where:

$$c_{20} = \sum_{i=0}^{\ell} rlk_0[i] \cdot c_2(i) \text{ and } c_{21} = \sum_{i=0}^{\ell} rlk_1[i] \cdot c_2(i).$$

Therefore, $c_{20} + c_{21} \cdot s = c_2 \cdot s^2 + e_{\text{relin\_v1}}$ where $e_{\text{relin\_v1}}$ is an error term from R_q.

▶ Click here for more details.

> 💡 By doing base T decomposition we get a "small" **relinearization noise**: $\|e_{\text{relin\_v1}}\| \leq (\ell + 1) \cdot B \cdot T \cdot n/2$ where B is an upper bound on the errors e_i.

❓ Now the question is how to compute the polynomials $c_2(i)$. The coefficients of these polynomials prove to be nothing but the columns of the matrix **`Reps`**.

▶ If you're curious to see why, click here:

So after all this math, we finally got the linear approximation of $c_2 \cdot s^2$ and thus, we can derive the standard ciphertext encoding the multiplication of the plaintexts. Here comes the code:

```
def mul_cipher_v1(ct1, ct2, q, t, T, poly_mod, rlk0, rlk1):
    """Multiply two ciphertexts.
    Args:
        ct1: first ciphertext.
        ct2: second ciphertext
        q: ciphertext modulus.
        t: plaintext modulus.
        T: base
        poly_mod: polynomial modulus.
        rlk0, rlk1: output of the EvaluateKeygen_v1 function.
    Returns:
        Tuple representing a ciphertext.
    """
    n = len(poly_mod) - 1
    l = np.int64(np.log(q) / np.log(T))  #l = log_T(q)

    c_0, c_1, c_2 = multiplication_coeffs(ct1, ct2, q, t, poly_mod)
    c_2 = np.int64(np.concatenate( (c_2, [0] * (n - len(c_2))) )) #pad

    #Next, we decompose c_2 in base T:
    #more precisely, each coefficient of c_2 is decomposed in base T such that c
    Reps = np.zeros((n, l + 1), dtype = np.int64)
    for i in range(n):
        rep = int2base(c_2[i], T)
        rep2 = rep + [0] * (l + 1 - len(rep)) #pad with 0
        Reps[i] = np.array(rep2, dtype=np.int64)
    # Each row Reps[i] is the base T representation of the i-th coefficient c_2
    # The polynomials c_2(j) are given by the columns Reps[:,j].

    c_20 = np.zeros(shape=n)
    c_21 = np.zeros(shape=n)
    # Here we compute the sums: rlk[j][0] * c_2(j) and rlk[j][1] * c_2(j)
    for j in range(l + 1):
        c_20 = polyadd_wm(c_20, polymul_wm(rlk0[j], Reps[:,j], poly_mod), poly_m
        c_21 = polyadd_wm(c_21, polymul_wm(rlk1[j], Reps[:,j], poly_mod), poly_m

    c_20 = np.int64(np.round(c_20)) % q
    c_21 = np.int64(np.round(c_21)) % q

    new_c0 = np.int64(polyadd_wm(c_0, c_20, poly_mod)) % q
    new_c1 = np.int64(polyadd_wm(c_1, c_21, poly_mod)) % q

    return (new_c0, new_c1)
```

This version is much simpler and cleaner than the previous one (yay!) and uses the so-called *modulus switching* technique. Recall that if we try to naively mask s^2 in the reliniarization key, then there is a large blow-up in the noise because of the c_2 multiplication. We want to avoid this to get a correct decryption.

In this version we mask s^2 modulo a a different modulus, $\mathrm{p} \cdot q$, with a much larger $\mathrm{p} \gg q$, as shown below.

$$rlk = (rlk_0, rlk_1) = ([-(a' \cdot s + e') + p \cdot s^2]_{p \cdot q}, a'),$$

for a uniform a' in $R_{p \cdot q}$ and e' drawn according to an error distribution χ' over R. Remember that our goal is to produce an approximation of $[c_2 \cdot s^2]_q$. The idea is that when we scale from $p \cdot q$ back to q, the noise gets divided by the large integer p, significantly reducing its size.

In a safe implementation the distribution χ' should be distinct from χ and its parameters should be carefully chosen for security reasons. Since security is not our main goal in this blog post, you can check the paper for further details.

```python
def evaluate_keygen_v2(sk, size, modulus, poly_mod, extra_modulus, std2):
    """Generate a relinearization key using version 2.
        Args:
            sk: secret key
            size: size of the polynomials.
            modulus: coefficient modulus.
            poly_mod: polynomial modulus.
            extra_modulus: the "p" modulus for modulus switching.
            st2: standard deviation for the error distribution.
        Returns:
            rlk0, rlk1: relinearization key.
        """
    new_modulus = modulus * extra_modulus
    a = gen_uniform_poly(size, new_modulus)
    e = gen_normal_poly(size, 0, std2)
    secret_part = extra_modulus * poly.polymul(sk, sk)

    b = np.int64(polyadd_wm(
        polymul_wm(-a, sk, poly_mod),
        polyadd_wm(-e, secret_part, poly_mod), poly_mod)) % new_modulus
    return b, a
```

The linear approximation of $[c_2 \cdot s^2]_q$ can be computed from the pair:

$$(c_{20}, c_{21}) = \left(\left[\left\lfloor \frac{c_2 \cdot rlk_0}{p} \right\rceil \right]_q, \left[\left\lfloor \frac{c_2 \cdot rlk_1}{p} \right\rceil \right]_q \right).$$

Indeed, $[c_{20} + c_{21} \cdot s]_q = [c_2 \cdot s^2]_q + e_{\text{relin\_v2}}$, for a "small" error $e_{\text{relin\_v2}}$ in R_q .

▶ For an intuition of why this happens, click here.

> 💡 We get a "small" **relinearization noise**, $e_{\text{relin\_v2}} \approx (c_2 \cdot e')/p$, for large p : $\|e_{\text{relin\_v2}}\| \leq \frac{q \cdot B' \cdot n}{p} + \frac{n+1}{2}$.

Next, we provide the easy code implementation for multiplying ciphertexts using this version.

```python
def mul_cipher_v2(ct1, ct2, q, t, p, poly_mod, rlk0, rlk1):
    """Multiply two ciphertexts.
    Args:
        ct1: first ciphertext.
        ct2: second ciphertext.
        q: ciphertext modulus.
        t: plaintext modulus.
        p: modulus-swithcing modulus.
        poly_mod: polynomial modulus.
        rlk0, rlk1: output of the EvaluateKeygen_v2 function.
    Returns:
        Tuple representing a ciphertext.
    """
    c_0, c_1, c_2 = multiplication_coeffs(ct1, ct2, q, t, poly_mod)

    c_20 = np.int64(np.round(polymul_wm(c_2, rlk0, poly_mod) / p)) % q
    c_21 = np.int64(np.round(polymul_wm(c_2, rlk1, poly_mod) / p)) % q

    new_c0 = np.int64(polyadd_wm(c_0, c_20, poly_mod)) % q
    new_c1 = np.int64(polyadd_wm(c_1, c_21, poly_mod)) % q
    return (new_c0, new_c1)
```

Recall that we can derive a fresh standard ciphertext that decrypts to $[mm']_t$ as $c_{mul} = (c_0 + c_{20}, c_1 + c_{21})$, since

$$c_0 + c_{20} + s \cdot (c_1 + c_{21}) = \Delta \cdot [mm']_t + v_{mult},$$

where $v_{mult} = u_3 + e_{\text{relin}}$ and $e_{\text{relin}} \in \{e_{\text{relin\_v1}}, e_{\text{relin\_v2}}\}$.

We need to make sure that c_{mul} decrypts *correctly* to $[mm']_t$. For this, it suffices to choose the parameters such that $\|u_3\| + \|e_{\text{relin}}\| \leq \Delta/2$.

Now that we have **two** versions of relinearization, which help us in deriving c_{mul}, we may wonder which one we can use:

Relinearization	Size of rlk (in bits)	Bound of e_{relin}
Version 1	$2(\ell+1) \cdot n \cdot \log q$	$(\ell+1) \cdot B \cdot T \cdot n/2$
Version 2	$2n \cdot \log pq$	$\frac{q \cdot B' \cdot n}{p} + \frac{n+1}{2}$

➡️ **size of rlk: Version 1** gives a relinearization key as $\ell + 1$ pairs of polynomials in R_q, whereas **Version 2** gives just one such pair. Recall that $\ell = \lfloor \log_T q \rfloor$ and see that this decreases as long as the base T increases.

➡️ **bound of e_{relin}:** The upper bounds of e_{relin} for both versions are according to [FV12], Lem.3. We consider B as a bound taken so that the error distribution χ takes values in $[-\mathrm{B},\ B]$ with high probability. We similarly define $B^{'}$ for the case of the error distribution χ', used in **Version 2**. Notice that for **Version 1**, a larger T leads to more noise (but smaller rlk), whereas in **Version 2**, a larger p leads to smaller noise (but larger rlk). For choosing the parameters in safe implementation, we refer the reader to [FV12] Sec. Realinearisation Version 2.

Setting the parameters

We set the parameters for our toy implementation just to make sure it always correctly decrypts at least one ciphertext multiplication. For that, we made sure that $\|u_3\| + \|e_{\text{relin}}\| < \Delta/2$. In practice, for the current choice it seems that decryption always works for 3 ciphertext multiplications when using V2 relinearization, for example. This is due to the fact that the theoretical bounds are worst-case bounds. For the same parameters we can make hundreds of addition (in practice it seems that decryption works even for 1000 additions 😮). Ciphertext addition is almost for free in terms of noise growth! 💥 We invite you to play around with the parameters and the bounds and try to see how many multiplications you can get (as they are the costly operations).

Let's play! 🎉

Now we can multiply ciphertexts, as we have implemented two versions of this: **`mul_cipher_v1`** and **`mul_cipher_v2`**, using relinearization. We can further add ciphertexts by using **`add_cipher`**. So yay! we can finally perform computations on encrypted data.

Bonus: if you're curious, you can try computing more complex (polynomial) operations on encrypted data, such as *multiplying three ciphertexts*. Here we only wanted to show how to perform one multiplication. For more levels of multiplications, you should set the parameters as in [FV12, Thm1] so that you decrypt correctly 😉.

Bonus2: you can also perform *plain operations*, such as adding or multiplying plaintexts to ciphertexts, by using **`add_plain`** and **`mul_plain`**, (with a *reduced noise growth*) both from here. For further details on how these work, you should check this. 🤓

So what are you waiting for? Go check it out!

```
import rlwe_he_scheme_updated as rlwe_updated
```

```
# Scheme's parameters
# polynomial modulus degree
n = 2 ** 2
# ciphertext modulus
q = 2 ** 14
# plaintext modulus
t = 2
# base for relin_v1
T = int(np.sqrt(q))
#modulusswitching modulus
p = q ** 3

# polynomial modulus
poly_mod = np.array([1] + [0] * (n - 1) + [1])

#standard deviation for the error in the encryption
std1 = 1
#standard deviation for the error in the evaluateKeyGen_v2
std2 = 1

# Keygen
pk, sk = rlwe_updated.keygen(n, q, poly_mod, std1)

#EvaluateKeygen_version1
rlk0_v1, rlk1_v1 = rlwe_updated.evaluate_keygen_v1(sk, n, q, T, poly_mod, st

#EvaluateKeygen_version2
rlk0_v2, rlk1_v2 = rlwe_updated.evaluate_keygen_v2(sk, n, q, poly_mod, p, st

# Encryption
pt1, pt2 = [1, 0, 1, 1], [1, 1, 0, 1]
cst1, cst2 = [0, 1, 1, 0], [0, 1, 0, 0]

ct1 = rlwe_updated.encrypt(pk, n, q, t, poly_mod, pt1, std1)
ct2 = rlwe_updated.encrypt(pk, n, q, t, poly_mod, pt2, std1)

print("[+] Ciphertext ct1({}):".format(pt1))
print("")
print("\t ct1_0:", ct1[0])
print("\t ct1_1:", ct1[1])
print("")
print("[+] Ciphertext ct2({}):".format(pt2))
print("")
print("\t ct1_0:", ct2[0])
print("\t ct1_1:", ct2[1])
print("")

# Evaluation
ct3 = rlwe_updated.add_plain(ct1, cst1, q, t, poly_mod)
ct4 = rlwe_updated.mul_plain(ct2, cst2, q, t, poly_mod)
#ct5 = (ct1 + cst1) + (cst2 * ct2)
ct5 = rlwe_updated.add_cipher(ct3, ct4, q, poly_mod)
# ct6 = ct1 * ct2
ct6 = rlwe_updated.mul_cipher_v1(ct1, ct2, q, t, T, poly_mod, rlk0_v1, rlk1_
ct7 = rlwe_updated.mul_cipher_v2(ct1, ct2, q, t, p, poly_mod, rlk0_v2, rlk1_
# Decryption
decrypted_ct3 = rlwe_updated.decrypt(sk, q, t, poly_mod, ct3)
decrypted_ct4 = rlwe_updated.decrypt(sk, q, t, poly_mod, ct4)
decrypted_ct5 = rlwe_updated.decrypt(sk, q, t, poly_mod, ct5)
decrypted_ct6 = rlwe_updated.decrypt(sk, q, t, poly_mod, ct6)
decrypted_ct7 = rlwe_updated.decrypt(sk, q, t, poly_mod, ct7)

print("[+] Decrypted ct3=(ct1 + {}): {}".format(cst1, decrypted_ct3))
print("[+] Decrypted ct4=(ct2 * {}): {}".format(cst2, decrypted_ct4))
print("[+] Decrypted ct5=(ct1 + {} + {} * ct2): {}".format(cst1, cst2, decry
print("[+] pt1 + {} + {} * pt2): {}".format(cst1, cst2, rlwe_updated.polyadd
                                            rlwe_updated.polyadd(pt1, cst1,
                                            rlwe_updated.polymul(cst2, pt2,
                                            t, poly_mod)))
print("[+] Decrypted ct6=(ct1 * ct2): {}".format(decrypted_ct6))
print("[+] Decrypted ct7=(ct1 * ct2): {}".format(decrypted_ct7))
print("[+] pt1 * pt2: {}".format(rlwe_updated.polymul(pt1, pt2, t, poly_mod)
```

written by Mădălina Bolboceanu, Miruna Roșca, Radu Țițiu