同态加密学习——以 CKKS 为例

- 下面是我在学习同态加密当中之前存在的一些基础问题，以及我的解答。理解如有偏颇还请指正。

噪声是什么？为什么要引入噪声？

- 所谓的同态加密便是利用一个同态的映射，将明文映射到密文域上面进行运算。然而，这种近似一一映射的方式也带来了安全隐患——作为攻击者，我是否可以在固定了密钥的情况下把密文 c 和与之对应的明文 m 之间建立映射呢？如果是这样，这种近似于一一映射的方式显然是不安全的。

- 那么，一个自然的想法，便是让 m 每一次用同一个密钥 k 的情况下的加密结果都不一样。我们的做法，便是在加密的过程当中，在密钥（公钥）当中引入一个 "随机变量" e。这个随机变量就是所谓的 "噪声"。 噪声采样自分布 χ_e （注意，在 CKKS 当中，这个分布是一个关于多项式的分布！），它不能够过大，因为那样会使得我们无法恢复明文。

- 每一次采样的时候，尽管 k, m 固定，但 e 却是随机生成的，因此这也就保证了每一次加密结果会有一些较小的偏差，攻击者也就不能够使用 "建立映射" 的方式攻击密码了。然而，噪声虽然保证了安全性，却也为同态加密在密文域下的加，乘运算带来了麻烦——因为噪声的存在，在密文域的加法，乘法运算当中噪声可能不断增长从而导致经过运算后的密文无法正确解密。

所谓的 Encoding 和 Decoding 的过程是如何实现的？

- 在暑期的项目当中，我阅读了 BFV，CKKS 加密方案在 troy-nova 当中的 example（和 SEAL 的 example是一样的），发现实际上同态加密库的使用其实很简单，只需要知道如何进行多项式的加法，乘法运算就可以使用接口来做到这一点。但是，当我阅读了同态加密介绍的文章后，发现实际上 BFV，CKKS 所使用的明文域居然是一个多项式环，而不是整数/复数向量域！既然如此，我们如果想加密一个自然数 n , 或是一个复数域上的向量 x, 首先我们要先将其映射到多项式环 $R = \mathbb{Z}_Q[x]/x^N + 1$ 上面，然后再在多项式环上面进行加密的操作。

- 既然这样，给定 n 或者 x, 我们是如何将它映射到 R 上的呢？在阅读 BFV 的文献时我并没有发现相关的编码步骤，一开始以为只是将它映射到多项式 x 上面，但后来在阅读 CKKS 的时候发现并非如此。不过在介绍映射的过程之前，我们不妨思考一下，这样的映射都应该满足哪些条件？一个自然的想法是我们希望构建的是一一映射（我们先暂且不考虑 `coeff_modulus` 这些模系数, 因此可以姑且忽略系数 Q 的存在）。我们学过线性代数，知道这一点可以使用线性代数当中的可逆矩阵进行实现。

- CKKS 的设计也的确是这么想的。在这个加密方案当中，我们需要将一个 $\frac{N}{2}$ 维度的复数向量 z 编码到多项式环上面，首先先使用在论文当中定义的 π^{-1} 映射进行 “反射共轭” 操作（这是我为它起的一个比较形象的名字），映射到向量域 H 上面，形成一个 N 分量的向量（请注意，这里我说的是 N 分量而非 N 维度，因为 π 显然是一一映射，所以 H 实际上是 $\frac{N}{2}$ 维，只是在向量表示的时候有 N 个分量）z'，乘以缩放因子 Δ（CKKS 支持的是定点数运算，这一缩放因子可以保证在后续的运算当中，较高精度的位不会被丢掉），然后和 $\mathbb{C}[x]/x^N+1$ 之间进行一一映射，而这是通过 Vandemonde 矩阵做到的：

- $\begin{bmatrix} 1 & \zeta_0 & \zeta_0^2 & \cdots & \zeta_0^{n-1} \ 1 & \zeta_1 & \zeta_1^2 & \cdots & \zeta_1^{n-1} \ \vdots & \vdots & \vdots & \ddots & \vdots \ 1 & \zeta_{n-1} & \zeta_{n-1}^2 & \cdots & \zeta_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \ a_1 \ \vdots \ a_{n-1}

\end{bmatrix}

\begin{bmatrix} z_0 \ z_1 \ \vdots \ z_{N-1} \end{bmatrix}$

- 其中，等式的右边就是 z'，而左边被 Vandemonde 矩阵作用的向量则是 z' 映射到的多项式的系数。

- 最后需要将这个 $\mathbb{C}[x]/x^N+1$ 当中的多项式映射到 R 当中，而这显然无法构成一一映射（注意，我们在上面讲噪声的时候也提到映射是 “近似” 一一映射）。具体的实现方式比较复杂，我在网上查找到了一篇相关文献 1.

- 最终，如果需要解码，只需要将上述步骤进行逆运算即可。但是刚才我们也看到了从 $\mathbb{C}[x]/x^N+1$ 到 R 的映射显然不是一一映射，因此 CKKS 会有一些精度上的损失。在 Microsoft Private AI Bootcamp 当中, Yongsoo Song (也就是 CKKS 的 S) 提到过 [2]。

重线性化的作用和目的是什么?

- CKKS 的密文实际上由两部分组成：(c_0, c_1)，而解密者在持有私钥 s 的情况下可以通过计算 $c_0 + c_1 s$ 得到最终的解密结果。在实际的计算当中，我们需要存储 c_0, c_1, s 的值就可以进行解密了！

- 那如果是密文相乘呢？给定了 (c_0, c_1) 以及 (c_0', c_1')，进行密文相乘之后的解密应该和明文相乘的结果是相同的，也就是说我需要存储的是 $(c_0 + c_1 s)(c_0' + c_1' s)$ 的结果——这是一个关于 s 的二次多项式，也就是说我们需要存储的是常数项，一次项，二次项以及私钥。

- 有没有发现什么不太一样的地方？没错，需要存储的内容实际上增加了！这似乎没有什么，但进一步思考，如果接下来我们对结果再进行乘法，就会造成需要的空间指数级增加——这

显然是我们不想看到的事情。那么，一个自然的想法是 “能不能找到一个一次多项式，使得它的结果和二次多项式是一样的呢”？这样，每一次乘法当中，我们都只需要存储一个一次多项式，然后将结果再用这种方法转化为一个新的一次多项式。

- 这个方法就是我们所谓的重线性化。为了实现这个方法，CKKS 当中引入了辅助密钥的概念：$evk = \left(-a's + e' + Ps^2,\ a'\right)$ 。在这个辅助密钥的帮助下，便可以实现重现性化了：设 $(c_0 + c_1 s)(c_0' + c_1' s) = d_0 + d_1 s + d_2 s^2$, 那么我们只需要考虑 $(d_0, d_1) + \left\lfloor P^{-1} * d_2 * evk \right\rceil$ 即可做到重线性化。

重缩放的作用和目的是什么?

- 重缩放可以说是 CKKS 算法当中最精髓的一个步骤。它能够起到提升精度以及消除噪声的作用。
- 所谓的 “缩放” 就是把密文乘以一个因子 Δ: 我们想一想，如果我们需要编程实现两个小数的乘法，并且希望保持精度。如果小数是 n 位的，那么两个小数 a, b 的乘积在数学上应该是有 $2n$ 位小数。但是，在计算机中又只能存储小数点后面的前 n 位，因此会造成 n 位的精度损失。为了解决这个问题，一个很好的方法便是将 a, b 各乘以一个因子，把它们转化为整数，然后在整数域进行乘法运算就可以避免这个问题了（我们不考虑整数还可能出现往上溢出的情况）。
- 而另外一方面，我们之前也有提到过在密文域内的乘法当中，噪声会随着乘法次数不断增大，最后甚至可能导致无法正常解密。下面是一张直观的图片：
-
-

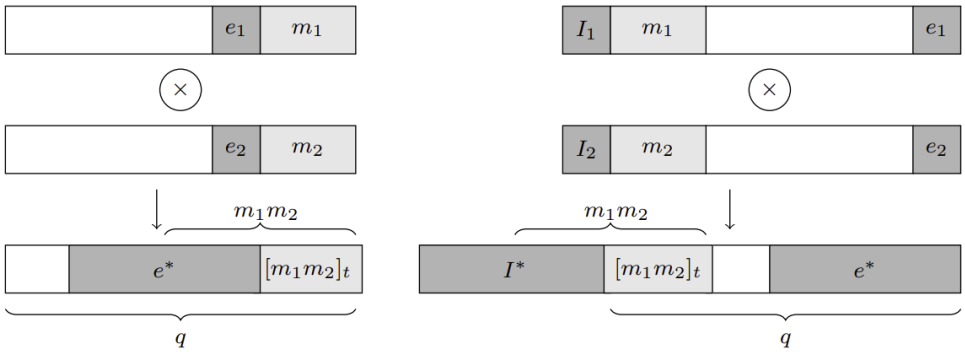

Fig. 1. Homomorphic multiplications of BGV-type HE schemes (left) and FV-type HE schemes (right)

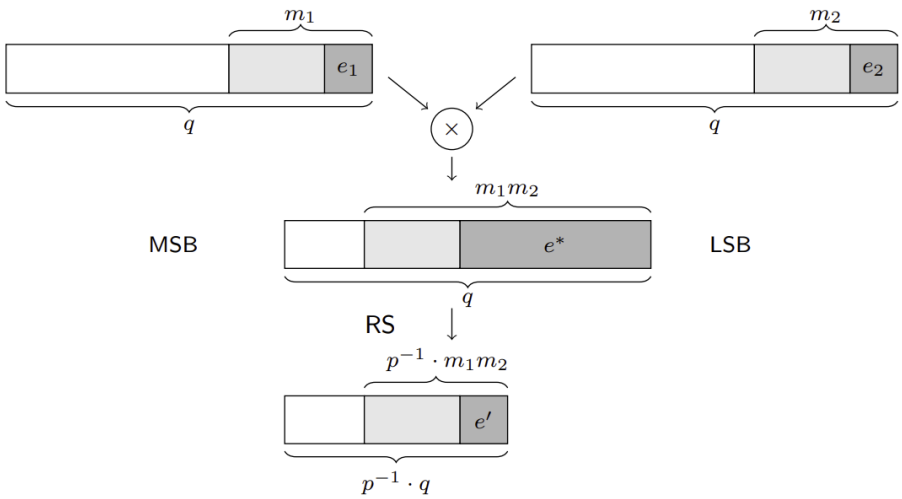

Fig. 2. Homomorphic multiplication and rescaling for approximate arithmetic

-

- 如何消除由于乘法导致增长的噪声呢？实际上我们只需要将密文 “除以” 一个因子就可以做到这一点。这就好比是移位操作——图中可以看到，噪声集中在密文的尾部，如果进行右移位操作，就可以将噪声的一部分去除掉，从而减少噪声的范围。这样一来，我们进行下一次乘法所带来的噪声增长就会减少。

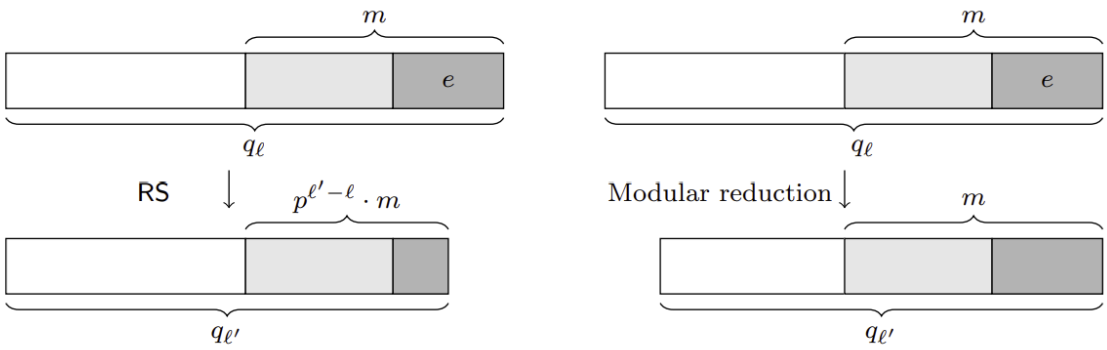

Fig. 3. Rescaling and simple modular reduction

- 具体的，我们可以通过对密文取不同的模数来做到这一点——如果一开始密文是在 $modq$ 的意义下进行计算的，那么我们可以通过取一个更小的模 q' 来做到这一点——这样一来，$modq'$ 就相当于是进行了上面提到的除法和移位操作。

- 根据上面的逻辑，我们可以设想，在每一次密文域乘法运算之后，都需要进行重缩放。而为了让噪声不断变小，我们不难发现，每一次重缩放所需要的模数都需要逐渐变小——这就形成了一条模数链。但是，在我们真实使用接口进行编程的时候，模数链的最后一个数往往是最大的。根据我查到的资料，这一点是为了解密时的安全考虑的，但深层次的原因我还并不是很清楚。

参考文献

[2]: - YouTube

[3]:CKKS explained: Part 1, Vanilla Encoding and Decoding