Introduction to Homomorphic Encryption

Fabian Boemer

March 30, 2020

1 Introduction

Homomorphic Encryption (HE) is a cryptosystem which enables computation on encrypted data. We will be focusing on public-key HE cryptosystems, in which a *public key* is used to encrypt a *plaintext* message into a *ciphertext*. A *secret key*, or *private key*, is used for decryption. In contrast to classical cryptography schemes, HE schemes enable encrypted computation without using the secret key. Computation on the encrypted data remains encrypted. Upon decryption, the plaintext result will be the same as if the computation had been performed on the underlying plaintexts, rather than the ciphertext.

HE schemes are based on *lattice cryptography*, a field of cryptography which forms the basis for many quantum-resistant cryptography schemes.

1.1 Classification

HE schemes are broadly classified by the types of computation they enable on the encrypted data. Typically, HE schemes support either Boolean or arithmetic circuits. Supported arithmetic circuits typically include addition and multiplication.

- *Partially homomorphic encryption (PHE)* schemes support evaluation of only one type of gate, e.g. addition or multiplication. RSA is a PHE scheme.
- *Somewhat homomorphic encryption* schemes support evaluation of two types of gates, e.g. addition and multiplication, but only a subset of circuits.

- *Leveled fully homomorphic encryption (LHE)* schemes support a limited number of evaluations of two types of gates. We will be focusing on LHE schemes, such as BFV, BGV, and CKKS.

- *Fully homomorphic encryption (FHE)* schemes support arbitrary circuits, i.e. of unlimited depth. This is possible with an expensive bootstrapping procedure whose implementations remain largely impractical to date. The FHEW / RLWE schemes are examples of FHE schemes. Note, many LHE schemes can be extended to FHE schemes, so the distinction between LHE and FHE schemes is somewhat arbitrary, based on the practical usage of the schemes.

1.2 How is HE possible?

At first glance, the ability to perform computation on encrypted data seems to contradict the notion of encryption, which obfuscates the underlying data. However, HE is possible by taking advantage of the mathematical structure underlying the encryption, and the relation between the plaintext and ciphertext space.

The idea of HE has been around since 1978, a year after the RSA scheme was published. In fact, RSA was the first HE scheme. RSA is a PHE scheme, supporting an unlimited number of multiplications. Recall, RSA encryption of a message m is given by $Enc(m) = m^e \mod n$, where e, n are parameters of the RSA scheme. Then,

$$\begin{aligned} Enc(m_1) \cdot Enc(m_2) &= m_1^e m_2^e \mod n \\ &= (m_1 m_2)^e \mod n \\ &= Enc(m_1 \cdot m_2) \end{aligned}$$

Thus, $Dec(Enc(m_1) \cdot Enc(m_2)) = Dec(Enc(m_1 \cdot m_2)) = m_1 m_2 \mod n$. So, we can perform multiplication on the encrypted $Enc(m_1)$, $Enc(m_2)$, and, upon decryption, obtain the plaintext modular multiplication $m_1 m_2 \mod n$. Note, however, RSA is *not* additively homomorphic.

2 Mathematical Background

Before introducing an FHE scheme, FV, we cover some mathematics fundamental to HE.

2.1 Galois field

A *finite field*, or *Galois field* is a finite set of elements in a field. A *field*, is a set F along with two operations, addition and multiplication, that satisfy the field axioms. The field axioms are:

- Associativity of addition and multiplication: $a + (b + c) = (a + b) + c$ and $a \cdot (b \cdot c) = (a \cdot b) \cdot c$
- Commutativity of addition ad multiplication: $a+b = b+a$ and $a \cdot b = b \cdot a$
- Additive and multiplicative identity: the existence of two different elements $0, 1 \in F$ such that $a + 0 = a$, $a \cdot 1 = a$.
- Additive inverse: for every $a \in F$, there exists an additive inverse, denoted $-a$, such that $a + (-a) = 0$
- Multiplicative inverse: for every $a \neq 0 \in F$, there exists a multiplicative inverse, denoted a^{-1} or $1/a$, such that $a \cdot a^{-1} = 1$
- Distributivity of multiplication over addition: $a \cdot (b+c) = (a \cdot b) + (a \cdot c)$

The real numbers are an example of a Galois field, with addition and multiplication defined as usual. The integers mod q, denoted

$$GF(q) = \mathbb{Z}/q\mathbb{Z}$$

for q prime, is another example of a Galois field, with addition and multiplication defined as modular addition and multiplication, respectively:

- for $x, y \in GF(q), x + y := (x + y) \mod q$
- for $x, y \in GF(q), x \cdot y := (x \cdot y) \mod q$

Note, q prime is required for existence of the multiplicative inverse for each element.

2.2 Notation

We denote $\mathbb{Z}_q$ as the set of integers in $(-q/2, q/2]$. This is choice for the specific representation of $GF(q)$. We could also choose $\mathbb{Z}_q$ to be the set of integers in $[0, q)$, for instance. For $a \in \mathbb{Z}$, we denote $[a]_q$ the unique integer in $\mathbb{Z}_q$ with $[a]_q = a \mod q$. We let $\lfloor \cdot \rceil$ denote rounding to the nearest integer.

2.3 Polynomials

Let $\mathbb{Z}_q[x]$ be the set of polynomials with coefficients in $\mathbb{Z}_q$. We will use boldface $\mathbf{a}$ or $\mathbf{a}(\mathbf{x})$ to represent polynomials. For $\mathbf{a} \in \mathbb{Z}_q[x]$, we let $[a]_q$ denote the element in $\mathbb{Z}_q[x]$ in which $[\cdot]_q$ has been applied to each element. Now, let $\mathcal{R}_q[x] := \mathbb{Z}_q[x]/\Phi[x]$, where $\Phi[x] = x^N + 1$, where N is a power of two. $\Phi[x]$ is the *cyclotomic polynomial*. The cyclotomic polynomial has several nice properties we don't expand on here. We only note that $\mathcal{R}_q[x]$ satisfies the field axioms, with addition and multiplication defined as addition and multiplication of polynomials, followed by reduction via dividing by $\Phi[x]$. That is $\mathcal{R}_q[x]$ is the set of polynomials of degree up to $N-1$, with integer coefficients in $\mathbb{Z}_q$. For instance, $x^N \equiv -1(\mod x^N + 1)$, so the -1 term can be used to reduce the power of x into the range $[0, N-1]$.

2.3.1 Norms

The *uniform norm* of a polynomial a is the largest coefficient in that polynomial, i.e $\|\mathbf{a}(\mathbf{x})\|_\infty = b$ means b is the largest coefficient in $\mathbf{a}$. The uniform norm is one way to indicate the 'size' of a polynomial.

2.3.2 Sampling

let χ be a distribution on the integers. Then, for $\mathbf{a} \in \mathbb{R}_q[x]$, we denote $\mathbf{a} \sim \chi$ to mean each coefficient of $\mathbf{a}$ is drawn i.i.d from χ. Two common distributions χ are:

- *uniform sampling* with some bound B, i.e. $\mathbf{a} \sim \chi_{U[B]}$ means the coefficients of $\mathbf{a}$ are each drawn uniformly from $\{-B, -B+1, -B+2, \ldots, -2, -1, 0, 1, 2, \ldots, B-2, B-2, B\}$.
- *Discrete Gaussian sampling*. For odd q, the coefficients of $\mathbf{a} \sim \chi_{DG}$ are sampled from $\{-(q-1)/2, \ldots, (q-1)/2\}$, according to a discrete Gaussian distribution with mean 0 and distribution parameter σ. Typically σ is fixed to 3.19, to optimize some properties of the security of the HE scheme.

Note, both uniform sampling and discrete Gaussian sampling ensure the sampled polynomial $\mathbf{a}$ has a small uniform norm, in expectation and with high probability.

3 Security

The security of HE schemes is based on the hardness of the *ring learning with errors (RLWE) problem*, which reduces to the Approximate Shortest Vector Problem (α-SVP). Let $\mathbf{a}, \mathbf{s}$ be sampled uniformly at random from $\mathcal{R}_q$, for some $q \geq 2$. Let $\mathbf{e} \leftarrow \chi$, where χ is some distribution over $\mathbb{R}_q$. Then, the RLWE problem is to recover $\mathbf{s}$ from the pair of polynomials $([\mathbf{a} \cdot \mathbf{s} + \mathbf{e}]_q, \mathbf{a})$. Without loss of security, $\mathbf{s}$ can instead be sampled from χ [1].

3.1 Semantic security

The HE schemes we discuss satisfy semantic security, specifically *Indistinguishability under chosen-plaintext attack (IND-CPA)*. Intuitively, this means knowledge of the ciphertext does not convey any knowledge of the underlying plaintext.

3.2 Encryption parameters

The security of HE schemes is typically determined by (some superset of) the following parameters:

- λ. A security level of λ bits indicates $\sim 2^\lambda$ operations are required to break the decryption. Typical values for λ are 128, 192, 256, with higher λ meaning higher security level.
- N. The *polynomial modulus degree*, N, is power-of-two degree in the RLWE polynomial. Typical values of N are $2^{10}, 2^{11}, \ldots, 2^{15}$. Larger N yields higher security. However, the runtime typically increases as $O(N)$ or $O(N \log N)$. Memory usage typically scales as $O(N)$.
- q. The *ciphertext modulus*, q, is the modulus in the finite field $\mathcal{R}_q[x]$. Larger q yields lower security, but enables more computation on the encrypted data.

Parameter selection is a difficult process, and remains largely hand-tuned for each circuit. It is important to follow the best practices in[2] to ensure the encryption scheme satisfies a reasonable security level.

4 FV Scheme

To show how all these parts fit together, we detail the FV encryption scheme [1]. The FV scheme is an SHE scheme.

The FV scheme samples from R_2, which implies $\|\mathbf{a}\| = 1$ with high probability. In addition to the ciphertext space, $\mathcal{R}_q$, the FV scheme contains has a plaintext space $\mathcal{R}_t$ for $q > t > 1$. Let $\Delta = \lfloor q/t \rfloor$. Some basic operations in the FV scheme are:

- *SecretKeyGen*: Sample the secret key $\mathtt{sk} = \mathbf{s} \leftarrow R_2$.
- *PublicKeyGen*: Sample $\mathbf{a} \leftarrow \mathcal{R}_q, \mathbf{e} \leftarrow \chi$, and output
$$\mathtt{pk} = ([-(\mathbf{a} \cdot \mathbf{s} + \mathbf{e})]_q, \mathbf{a}).$$
- *Encrypt*($\mathtt{pk}$, $\mathbf{m}$): To encrypt a message $\mathbf{m} \in \mathcal{R}_t$, let $\mathbf{p_0} = \mathtt{pk}[0], \mathbf{p_1} = \mathtt{pk}[1]$, sample $\mathbf{u}, \mathbf{e_1}, \mathbf{e_2} \leftarrow \chi$, and return
$$\mathtt{ct} = \left([\mathbf{p_0} \cdot \mathbf{u} + \mathbf{e_1} + \Delta \cdot \mathbf{m}]_q, [\mathbf{p_1} \cdot \mathbf{u} + \mathbf{e_2}]_q\right).$$

We note a few properties of the FV scheme so far.

- The RLWE problem ensures the secret key cannot be recovered from the public key.
- The ciphertext is computed from the secret key $\mathbf{s}$ and the plaintext message $\mathbf{m}$. However, noise is used to prevent the discovery of $\mathbf{m}$ and $\mathbf{s}$. This use of noise an essential component to many HE schemes, and means such HE schemes are non-deterministic. As we will see, HE operations increase the noise of the ciphertext, until a certain bound is reached, past which decryption is no longer accurate. Thus, noise management becomes a critical component to using HE.
- The use of modular arithmetic is essential here to mask $\mathbf{s}$ and $\mathbf{m}$. Since $\mathbf{a}$ is drawn from $\mathcal{R}_q$ uniformly at at random, both $\mathtt{ct}[0]$ and $\mathtt{ct}[1]$ will be uniform at random from $\mathcal{R}_q$. This helps preserve IND-CPA security.
- The use of Δ is to scale the message. This essentially moves the message to the most-significant bits (MSBs) of the ciphertext, safely out of the way of the error polynomials. See Figure 1 in `https://eprint.iacr.org/2016/421.pdf`. Thus, larger Δ will yield more *noise budget*, i.e. more runway for HE operations.

- Figure 1 shows how the secret key, message, mask, and noise contribute to the ciphertext. The relation between `ct`[0] and `ct`[1], hints at the decryption procedure. Multiplying `ct`[1] be $\mathbf{s}$ will allow us to remove the $\mathbf{aus}$ mask from `ct`[0].

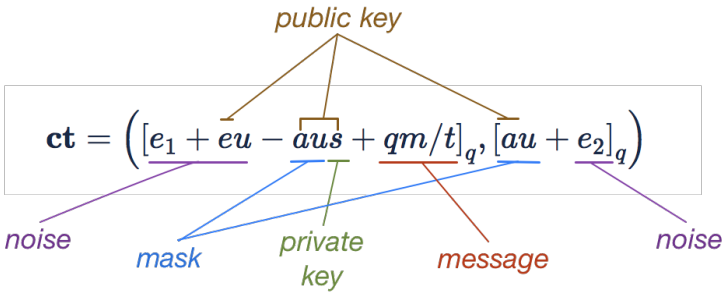

Figure 1: Taken from `https://blog.n1analytics.com/homomorphic-encryption-illustrated-primer/`.

The decryption procedure is given by:

- Decrypt(`sk, ct`). Let $\mathbf{c_0} = \texttt{ct}[0]$ and $\mathbf{c_1} = \texttt{ct}[1]$. Then,

$$\mathbf{m} = \left[\left\lfloor \frac{t \cdot [\mathbf{c_0} + \mathbf{c_1} \cdot \mathbf{s}]_q}{q} \right\rceil\right]_t$$

We note if the error terms $\mathbf{u}, \mathbf{e_1}, \mathbf{e_2}$ are small enough, the decryption procedure is correct, though we recommend verifying this, or following the proof in [1].

4.1 Homomorphic operations

4.1.1 Homomorphic Addition

Suppose we have two ciphertexts a, b encrypting plaintext polynomials $\mathbf{m_1}, \mathbf{m_2}$, respectively, with the same public key:

$$a = ([\mathbf{pk_0} \cdot \mathbf{u_1} + \mathbf{e_1} + \Delta \cdot \mathbf{m_1}]_q, [\mathbf{p_1} \cdot \mathbf{u_1} + \mathbf{e_2}]_q)$$
$$b = ([\mathbf{pk_0} \cdot \mathbf{u_2} + \mathbf{e_3} + \Delta \cdot \mathbf{m_2}]_q, [\mathbf{p_1} \cdot \mathbf{u_2} + \mathbf{e_4}]_q)$$

Note, we used two small polynomials$\mathbf{u_1}, \mathbf{u_2}$, and four error polynomials $\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}, \mathbf{e_4}$. Then,

$$\begin{aligned}\mathbf{a}+\mathbf{b} &= ([\mathbf{pk_0}(\mathbf{u_1}+\mathbf{u_2})+(\mathbf{e_1}+\mathbf{e_3})+\Delta(\mathbf{m_1}+\mathbf{m_2})]_q, [\mathbf{p_1}(\mathbf{u_1}+\mathbf{u_2})+(\mathbf{e_2}+\mathbf{e_4})]_q) \\ &= ([\mathbf{pk_0u_3}+\mathbf{e_5}+\Delta(\mathbf{m_1}+\mathbf{m_2})]_q, [\mathbf{p_1u_3}+\mathbf{e_6}]_q)\end{aligned}$$

Note, this is an encryption of $\mathbf{m_1}+\mathbf{m_2}$ with small polynomial $\mathbf{u_3} := \mathbf{u_1}+\mathbf{u_2}$, and error polynomials $\mathbf{e_5} := \mathbf{e_1}+\mathbf{e_3}$ and $\mathbf{e_6} := \mathbf{e_2}+\mathbf{e_4}$.

Decryption $a+b$ before rounding yields

$$[\Delta(\mathbf{m_1}+\mathbf{m_2})+\mathbf{e_5}+\mathbf{eu_3}+\mathbf{e_6s}]_q$$

As long as $\mathbf{e_5}+\mathbf{eu_3}+\mathbf{e_6s}$ remains small (smaller than $q/(2t)$, in fact), the decryption remains accurate. Note, however, this noise is larger than of the noise in each original ciphertext $\mathbf{a}$ or $\mathbf{b}$. So, we have a limited *noise budget* before HE additions no longer decrypt correctly.

Finally, we note HE addition is simple element-wise polynomial addition.

4.1.2 Homomorphic multiplication

Homomorphic multiplication is more complicated that homomorphic addition, primarily due to the scale factor Δ. Simply multiplying $\mathbf{a}[\mathbf{0}] \cdot \mathbf{b}[\mathbf{0}]$ would yield a $\Delta^2 \cdot \mathbf{m_1m_2}$ term (among other terms), which would turn into $\Delta\mathbf{m_1m_2}$ upon decryption. So, the naive element-wise multiplication does not work.

We recommend reading [**?**] for more discussion, but the long story short is that HE multiplication yields a ciphertext $\texttt{ct} = (\mathbf{c_0}, \mathbf{c_1}, \mathbf{c_2})$, with *three* polynomials, rather than two. The decryption procedure expands to

$$\left[\left\lfloor \frac{t \cdot [\mathbf{c_0}+\mathbf{c_1}\cdot\mathbf{s}+\mathbf{c_2}\cdot\mathbf{s^2}]_q}{q} \right\rceil\right]_t$$

which generalizes the two-polynomial ciphertext decryption.

Although the above approach works, each additional multiplication will increase the size of the ciphertext, resulting in a rapid blowup of the memory and runtime. Therefore, a procedure known as *relinearization* is typically applied to three-polynomial ciphertexts. Relinearization uses a public key known as a *relinearization key* to convert a three-polynomial ciphertext into a

two-polynomial ciphertext. That is, relinearization computes $\texttt{ct'} = [\mathbf{c_0}', \mathbf{c_1}']$ such that

$$[\mathbf{c_0} + \mathbf{c_1} \cdot \mathbf{s} + \mathbf{c_2} \cdot \mathbf{s^2}]_q = [\mathbf{c_0}' + \mathbf{c_1'} \cdot \mathbf{s} + \mathbf{r}]_q$$

for small $\|r\|$.

We refer to [1] for more details on how this is done. With this, we write out the homomorphic multiplication procedure:

- $Mul(\texttt{ct}_1, \texttt{ct}_2, \texttt{rlk})$: compute

$$c_0 = \left[\left\lfloor \frac{t \cdot (\texttt{ct}_1[0] \cdot \texttt{ct}_2[0])}{q} \right\rceil \right]_q$$
$$c_1 = \left[\left\lfloor \frac{t \cdot (\texttt{ct}_1[0] \cdot \texttt{ct}_2[1] + \texttt{ct}_1[1] \cdot \texttt{ct}_2[0])}{q} \right\rceil \right]_q$$
$$c_2 = \left[\left\lfloor \frac{t \cdot (\texttt{ct}_1[1] \cdot \texttt{ct}_2[1])}{q} \right\rceil \right]_q$$

- *Relinearize*: Compute

$$(\mathbf{c}_{2,0}, \mathbf{c}_{2,1}) = \left(\left[\left\lfloor \frac{\mathbf{c}_2 \cdot \texttt{rlk}[0]}{p} \right\rceil \right]_q, \left[\left\lfloor \frac{\mathbf{c}_2 \cdot \texttt{rlk}[1]}{p} \right\rceil \right]_q \right)$$

  where

$$\texttt{rlk} = \left([-\mathbf{a}(\mathbf{a} \cdot \mathbf{s} + \mathbf{e}) + p \cdot \mathbf{s}^2]_{p \cdot q}, \mathbf{a} \right)$$

  for $\mathbf{a} \leftarrow \mathbf{R}_{p \cdot q}, \mathbf{re} \leftarrow \chi'$, where χ' is another distribution, and p is an additional scalar parameter of the FV scheme. Return $(\mathbf{c}_0 + \mathbf{c_{2,0}}]_q, [\mathbf{c}_1 + \mathbf{c}_{2,1}]_q)$.

A few notes on homomorphic multiplication:

- The noise growth is much larger than HE addition. As such, HE circuits are typically parameterized by the number of multiplications in this longest evaluation chain. This number of known as the *multiplicative depth*. For instance, x^4 computed as $x(x(x(x)))$ has a multiplicative depth of 3, whereas $(x \cdot x)(x \cdot x)$ has a multiplicative depth of 2, and is preferable.

Now that we have covered the basic FV scheme, we make a few additional notes:

- *Plaintext operations.* In addition to ciphertext-ciphertext multiplication and addition, FV supports ciphertext-plaintext addition and multiplication. That is, a ciphertext can be added or multiplied with a plaintext, yielding a ciphertext result. Ciphertext-plaintext operations are typically faster than ciphertext-ciphertext operations.
- *Accuracy pitfalls.* There are several reasons the result of HE operations can be inaccurate in practice:
  - The message wraps around the plaintext modulus t. Fix by choosing larger t.
  - The noise corrupts the message. Fix by choosing larger Δ.
  - The ciphertext wraps around the ciphertext modulus q. Fix by choosing larger q.

  Unfortunately, choosing larger t, q, Δ requires larger polynomial modulus N to preserve security, and resulting in larger runtime/memory overhead.

5 Performance

We next make a few notes on HE implementation optimizations, and performance characterization.

5.1 Efficient Polynomial Arithmetic

The primary primitive in the FV scheme is polynomial addition and multiplication, with large polynomials (of degree N, typically at least 4096), and integer coefficients up to q bits (typically at least 100 bits). For efficient implementation, integers are often represented using a *residue number system (RNS)*, and polynomials are often represented in CRT form.

5.1.1 Integer Arithmetic in RNS Form

Let $x, y \in \mathbb{Z}/q\mathbb{Z}$ of q bits. Since q is typically larger than a CPU word size, we must represent x non-natively, e.g. via a list of 64-bit integers $x = n_1 n_2 \dots n_L$, where n_i stores the i'th 64 MSB bits of x. Then, computing $x+y \mod q$ requires $O(L)$ scalar additions. Computing $x \cdot y$ requires $O(L^2)$ scalar operations, when implemented natively with long-hand multiplication.

In the case where $q = \prod_{i=1}^{L} q_i$, the product of L pairwise co-prime factors q_i, the Chinese remainder theorem asserts $x \mod q$ has a unique representation as the list $(x \mod q_1, x \mod q_2, \dots, x \mod q_L)$, known as RNS form, or CRT form. Addition of two numbers in RNS form is performed element-wise, with $O(L)$ runtime. The major advantage of the RNS form is that multiplication is also performed element-wise, in $O(L)$ time.

5.1.2 Polynomial Arithmetic in Canonical Embedding

One way to represent an RLWE polynomial $\mathbf{a}(\mathbf{x}) = \sum_{i=0}^{N-1} a_i x^i$ is through *coefficient embedding*: via a vector of elements

$$\mathbf{a} \mapsto [a_0, a_1, \dots, a_{N-1}].$$

Addition is performed element-wise, in $O(N)$ time (assuming scalar addition and multiplication are constant-time). Multiplication, however, requires $O(N^2)$ time with a naive implementation, though this is improved to $O(N \log N)$, using a FFT-based implementation.

A different way to represent $\mathbf{a}$ is in the *canonical embedding*: a vector of elements whose values are the values of $\mathbf{a}$ evaluated at different values of x:

$$\mathbf{a} \mapsto [a(\sigma_1), a(\sigma_2), \dots, a(\sigma_{N-1})]$$

Then, both addition and multiplication are performed element-wise, in $O(N)$ time.

We note the properties of cyclotomic polynomials ensures correctness and good error propagation bounds, but omit further discussion.

Together, the canonical embedding, and RNS integer representation are known as *DoubleCRT form*. Table 5.1.2 summarizes the previous discussion.

To summarize, polynomials are most efficiently represented as matrix of shape $N \times L$. A ciphertext, therefore, is best represented as a tensor of

	Polynomial		Integer	
Operation	**Coefficient**	**Canonical**	**Native**	**RNS**
Add	$O(N)$	$O(N)$	$O(L)$	$O(L)$
Mult	$O(N \log N)$	$O(N)$	$O(L^2)$	$O(L)$

Table 1: Runtime of addition and multiplication on different polynomial and integer representations. Polynomials have N coefficients, and integers are $\log_2(q)$ bits, where $\log_2(q) \approx L \cdot$ word size. Polynomial operations runtimes assume integer multiplication is $O(1)$.

shape $2 \times N \times L$. HE operations are performed element-wise on elements of the tensor. For optimal performance, therefore, HE operations should traverse the ciphertext elements in order as represented in memory. Note, the $2 \times N \times L$ word-sized scalars required to represent a ciphertext are the primary reason for teh large runtime and memory overhead of HE operations. Typically $N \times L$ is on the order of $10^4 - 10^6$, suggesting a $10^4 - 10^6$ runtime and memory overhead.

6 Homomorphic Encryption for Deep Learning

We focus on the use of HE to protect input data to a DL model. HE can also be used to encrypt the DL model, but we do not discuss this further, beyond noting encrypting the model has additional runtime and memory overhead compared to encrypted the data.

So far, our discussion of HE has been limited to polynomials. However, many applications, such as deep learning (DL) typically operate on floating-point scalars. As a result, a process known as *encoding* is required to map scalars to polynomials. The reverse process, *decoding* transforms the polynomials back to integers. Thus, a typical workflow using HE given a scalar x is:

$$x \xrightarrow{Encode} \mathbf{m} \xrightarrow{Encrypt} ct \xrightarrow{\text{HE add/mult}} ct' \xrightarrow{Decrypt} \mathbf{m}' \xrightarrow{Decode} y$$

6.1 Encoding

There are several different possibilities for encoding a floating-point scalar x into a polynomial $\mathbf{m}$. We use the notation in PALISADE.

- *Integer encoding.* Embed the i'th bit of x to the i'th coefficient of m. This is similar to what is used in FHEW.
- *Scalar encoding.* Multiply and round x by a scale factor s, $x_{int} = \lfloor s \cdot x \rceil$. Let $\mathbf{m} = (x_{int}, \ldots, x_{int})$.

Alternatively, we can also embed a vector of integers $[a_0, \ldots, a_{N-1}]$ to a polynomial:

- *Coefficient Packed encoding.* Let $\mathbf{m} = a_0 + a_1 x + a_2 x^2 + \ldots a_{N-1} x^{N-1}$
- *SIMD encoding.* This is an encoding which enables single-instruction multiple-data arithmetic on the ciphertexts. That is, performing HE operations is equivalent to performing the corresponding plaintext operation on each element in the vector.

SIMD encoding is extremely useful in enabling additonal computation per HE operation. For instance, we can perform the following operations:

- encode the vector $[1, 2, 3, 4] \mapsto \mathbf{m_1}$.
- encode the vector $[5, 6, 7, 8] \mapsto \mathbf{m_2}$.
- encrypt $E[\mathbf{m_1}] \mapsto c_1$
- encrypt $E[\mathbf{m_2}] \mapsto c_2$
- Homomorphically add $c_1 + c_2 \mapsto c_3$,
- Decrypt $c_3 \to m_3$
- Decode $m_3 \mapsto [6, 8, 10, 12]$.

We have performed element-wise addition of a vector, using a single HE addition! In general, the FV scheme enables encoding up to N integers. HE addition and multiplication are performed as-if-element-wise on the plaintext data. Thus, SIMD encoding can increase computation throughput by a factor of N. So, performing a encrypted scalar addition requires the same runtime and memory as encrypted vector addition. Efficient use of SIMD encoding, therefore, requires addition and multiplication on large vectors.

6.2 Use of Encoding in DL

Deep learning is a perfect use case for addition and multiplication on larger vectors. There are two general methods with which to use SIMD encoding, also known as *plaintext packing*, or *SIMD packing* for DL.

- *Batch-axis packing.* When performing inference on a data batch of shape $n \times W \times H \times C$, batch-axis packing encodes the input as $W \times H \times C$ ciphertexts, with ciphertext packing a vector of n values.
- *Inter-axis packing.* An inference data batch of shape $n \times W \times H \times C$ is stored as N ciphertexts, each storing $W \times H \times C$ values using SIMD packing.

We note batch-axis packing has the same runtime for each batch size $n \leq N$. As such, batch-axis packing maximizes throughput, at the cost of higher latency. DL operations such as convolution, GEMM, reshape, and broadcast map naturally to HE operations using batch-axis packing. We refer to [**?**, 3] for more details.

Inter-axis packing requires the use of an additional HE operation, *rotation*, to execute common DL operations. HE rotation uses an additional public key, a *Galois key*, to perform rotation of SIMD-encoded ciphertexts. We refer to GAZELLE [4] for more details on using inter-axis packing for DL. We note inter-axis packing typically has lower latency and lower throughput compared to batch-axis packing. Implementing DL operations such as reshape and broadcast are typically difficult using inter-axis packing.

6.3 Implementing DL for HE

We note HE schemes typically do not support non-polynomial functions. As such, special consideation must be given to non-polynomial activations such as ReLU and MaxPool. For instance, polynomial activations or a decrypt-ReLU-encrypt mechanism may be used. We refer to [5, 3] for more details.

7 Additional Reading

We hope the above serves as an accessible brief introduction to HE. We recommend the following resources to gain further familiarity:

- Microsoft SEAL (`https://github.com/Microsoft/SEAL`). The comments in the examples are very useful.
- `https://blog.n1analytics.com/homomorphic-encryption-illustrated-primer/`
- The CKKS encryption scheme (also known as HEAAN) [6]
- Intel nGraph-HE repository (`ngra.ph/he`) and the corresponding papers [5, 3]
- PALISADE (`https://palisade-crypto.org`)

References

[1] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[2] Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Jeffrey Hoffstein, Kristin Lauter, Satya Lokam, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. Security of homomorphic encryption. Technical report, HomomorphicEncryption.org, Redmond WA, USA, July 2017.

[3] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. ngraph-he2: A high-throughput framework for neural network inference on encrypted data. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 45–56, 2019.

[4] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1651–1669, 2018.

[5] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. ngraph-he: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pages 3–13, 2019.

[6] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.