CS 798
Privacy in Computation and Communication

Module 5

Privacy in Computation: Homomorphic Encryption

Spring 2024

Homomorphic encryption

Recall the three main ways to achieve privacy in computation:

- Distributed trust
- Trusted hardware
- Homomorphic encryption

Homomorphic encryption

Recall the three main ways to achieve privacy in computation:

- Distributed trust
- Trusted hardware
- Homomorphic encryption

Motivation

- With distributed trust, the privacy and security relied on parties not colluding
- With trusted hardware, the privacy and security relied on the lack of side channels in the software *and hardware* implementations of the programs running in the TEEs, and the TEEs themselves
- With homomorphic encryption, we aim to rely on the hardness of certain mathematical problems
  - We will *not* be talking about the details of these problems in this course, but just the APIs to take advantage of them

Recall: the group API

- Two data types: scalars and points
- A group is a set of points (with certain properties)
- There is *one* operation you can do to combine two points; this operation yields another point
    - $P + Q \to R$
- There are two special points
    - The identity $\mathcal{O}$ and the basepoint B
    - $P + \mathcal{O} = P$ for any point P

The ring API

- In this module, we will go beyond groups, to *rings*
- Just one data type: elements
  - We'll usually write elements with lowercase letters
- *Two* operations that can combine two elements
  - $a + b \to r$
  - $a \cdot b \to s$
- Various (intuitive) rules about the operations
  - e.g., $r \cdot (a + b) = r \cdot a + r \cdot b$ for any elements r, a, b
- Two special elements: 0, 1
  - $r + 0 = r, \quad r \cdot 1 = r, \quad r \cdot 0 = 0$ for any element r

Examples of rings

- The integers ($\mathbb{Z}$)
- The integers mod n ($\mathbb{Z}_n$)
  - Even if n is not prime!
- Polynomials with coefficients in another ring
  - e.g., in $\mathbb{Z}_{10}[x]$:

$$(3x^2 + 7x + 4) + (5x^2 + 4x + 8) = 8x^2 + x + 2$$

$$(3x^2 + 7x + 4) \cdot (5x^2 + 4x + 8) = 5x^4 + 7x^3 + 2x^2 + 2x + 2$$

Examples of rings

- Polynomials of degree less than some fixed N
- If you add two polynomials of degree less than N, you get another such polynomial for sure
- But if you *multiply* two polynomials of degree less than N, you might get a polynomial of degree N or higher
  - So you need a technique for degree reduction
  - Remember we saw something similar in Module 3?

Degree reduction

- The way we went from $\mathbb{Z}$ (an infinite ring) to $\mathbb{Z}_n$ (a finite ring) was to take the remainder when we divide by n
  - $4 \cdot 7 = 28 = 2 \cdot 10 + 8$, so $4 \cdot 7 = 8$ in $\mathbb{Z}_{10}$
- We do the same with polynomials: take the remainder when we do (polynomial) division by a degree-N polynomial
- e.g., in $\mathbb{Z}_{10}[x]$, take the remainder when we divide by the polynomial $x^N + 1$
  - We write the resulting ring as $\frac{\mathbb{Z}_{10}[x]}{(x^N+1)}$

Example

- Use the same polynomials as for the previous example, but now in $\frac{\mathbb{Z}_{10}[x]}{(x^3+1)}$
- $(3x^2 + 7x + 4) \cdot (5x^2 + 4x + 8) = 5x^4 + 7x^3 + 2x^2 + 2x + 2$

  $= (5x + 7)(x^3 + 1) + (2x^2 + 7x + 5) \quad$ in $\mathbb{Z}_{10}[x]$
- So $(3x^2 + 7x + 4) \cdot (5x^2 + 4x + 8) = 2x^2 + 7x + 5$ in $\frac{\mathbb{Z}_{10}[x]}{(x^3+1)}$

What are homomorphisms?

- We said back in Module 2 that *group commitments* are homomorphic
  - $\mathrm{Com}(x) = xB$
  - $\mathrm{Com}(x + y) = (x + y)B = xB + yB = \mathrm{Com}(x) + \mathrm{Com}(y)$
- A *homomorphism* is a function (e.g., "Com") that preserves *structure*
  - e.g., if $x + y = z$, then $\mathrm{Com}(x) + \mathrm{Com}(y) = \mathrm{Com}(z)$
  - Recall that sometimes we write groups additively and sometimes we write groups multiplicatively, so the way we *write* this structure may change; e.g., we may write $\mathrm{Com}(x) = g^x$ and then:

    if $x + y = z$, then $\mathrm{Com}(x) \cdot \mathrm{Com}(y) = \mathrm{Com}(z)$

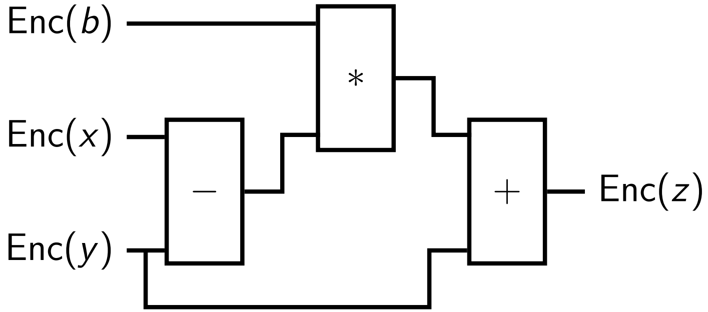
What are homomorphisms?

- So if we're given $\mathrm{Com}(x)$ and $\mathrm{Com}(y)$, we can compute $\mathrm{Com}(x + y)$ by ourselves, *without knowing* x and y
- In some cases, we may need some additional (public) information like a public key; more on that in a bit

Homomorphic encryption

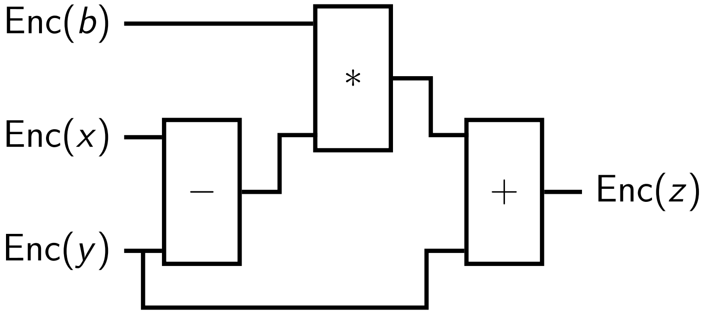
- Homomorphic encryption is then simply where the encryption function is homomorphic:
- For example, given $\text{Enc}(x)$ and $\text{Enc}(y)$, and possibly other public information, you can compute $\text{Enc}(x + y)$ without knowing the decryption key or learning x or y
    - Note that you may not literally add $\text{Enc}(x)$ and $\text{Enc}(y)$ to get $\text{Enc}(x + y)$
    - There's just *some public operation* you can do to get $\text{Enc}(x + y)$

Computation using homomorphic encryption



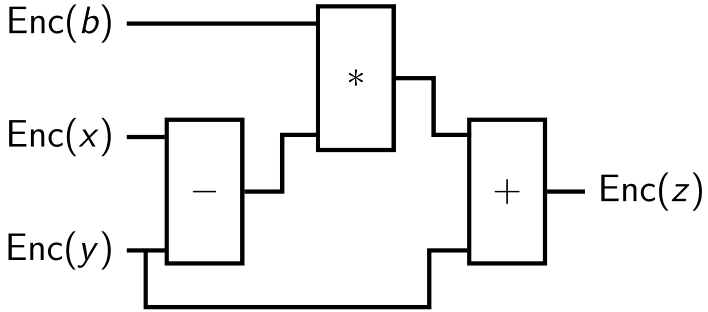
- The strategy is very similar to previous modules
- Write the program as a circuit
- Each gate processes encrypted inputs to yield the encrypted output without being able to decrypt

Computation using homomorphic encryption



- As usual, linear gates ($+$, $-$) are easier than non-linear ($*$)
- Computing the output of gates may require knowledge of the public encryption key and/or other public parameters

Computation using homomorphic encryption



- As was the case in Module 3, the *multiplicative depth* of this circuit plays a crucial role

Types of homomorphic encryption

- Homomorphic encryption has been around since the 1970s, but only in a limited form

- In this most basic form of homomorphic encryption (now called "partially homomorphic encryption"), given $\text{Enc}(x)$ and $\text{Enc}(y)$, you can compute $\text{Enc}(x \cdot y)$ but *not* $\text{Enc}(x + y)$ (multiplicative homomorphic encryption)
  - e.g., Textbook RSA, textbook El Gamal

- In other schemes, it's the other way around: given $\text{Enc}(x)$ and $\text{Enc}(y)$, you can compute $\text{Enc}(x + y)$ but *not* $\text{Enc}(x \cdot y)$ (additive homomorphic encryption)
  - e.g., Exponential El Gamal, Pailler

Types of homomorphic encryption

- In the mid-2000s, we saw homomorphic encryption schemes that could do both addition and multiplication, but were limited to a multiplicative depth of just 1
    - e.g., BGN
    - Dot products
    - Euclidean (squared) distances
- The limit to the multiplicative depth is based on how the math works
    - It's not a parameter to the system
- These schemes (for any fixed multiplicative depth) are now called "somewhat homomorphic encryption"

Types of homomorphic encryption

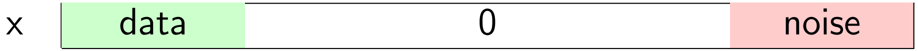
- Breakthrough in 2009: Craig Gentry introduced the first fully homomorphic encryption scheme
    - Can process *any* circuit homomorphically
- We will break the process into two steps

Levelled homomorphic encryption

- The first step is levelled homomorphic encryption
- Here, we can handle circuits of multiplicative depth at most d
  - e.g, BFV, BGV, FHEW, CKKS
- But now d is a *parameter* you can set freely when creating, for example, your private and public keys
  - Larger d will mean larger keys
- This may be all you need, if you know the program you want to evaluate on your encrypted data in advance (and so know its multiplicative depth, and it's not huge)

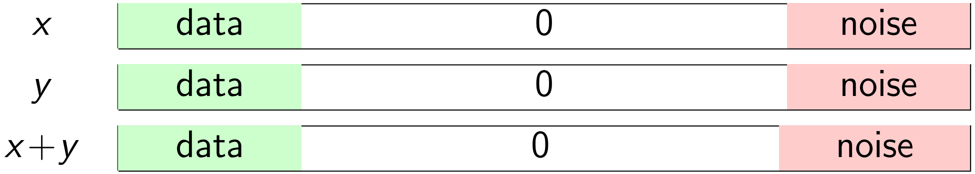
Multiplicative depth

- Why is *d* limited?
- “Noise”
- In systems like these, (extended) plaintexts are large, but only a little bit is “useful” data
  - The useful data is the actual (non-extended) plaintext
- Extended plaintexts start off with only a little bit of noise
  - Some noise is needed for security



Noise

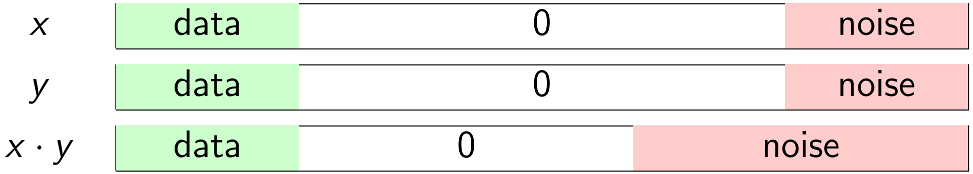
- Adding two ciphertexts:
    - That is, performing the public operation that computes Enc($x + y$) from Enc(x) and Enc(y), not necessarily literally adding the ciphertexts



- The noise increases just a tiny bit

Noise

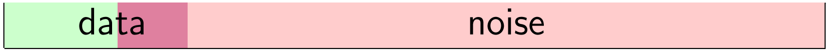
- Multiplying two ciphertexts:
  - That is, performing the public operation that computes $\mathrm{Enc}(x \cdot y)$ from $\mathrm{Enc}(x)$ and $\mathrm{Enc}(y)$, not necessarily literally multiplying the ciphertexts



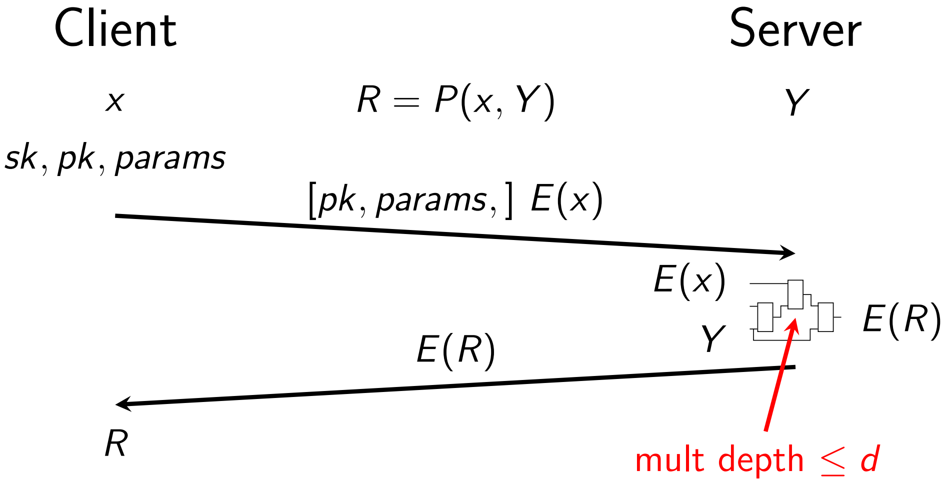
- The noise increases quite a bit

Noise

- So the multiplicative *depth* is what governs how much noise there is
- To allow for a multiplicative depth of d, you make the extended plaintexts large enough to accommodate that much noise
- What if you exceed that depth?



Recap so far



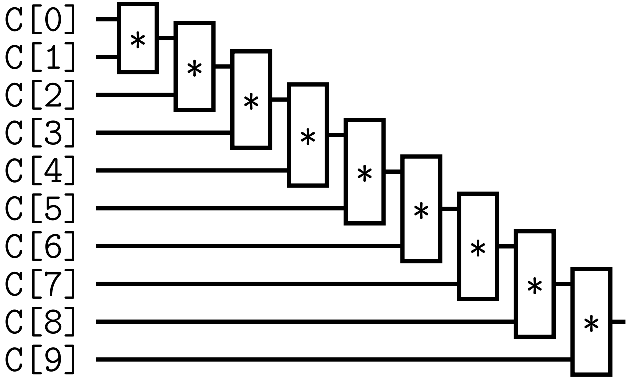
Client
Server
x
$R = P(x, Y)$
Y
$sk, pk, params$
$[pk, params,]\ E(x)$
$E(x)$
Y
$E(R)$
$E(R)$
R
mult depth $\leq d$

Handling circuits with large multiplicative depth

- What can we do if the circuit for the program P has large multiplicative depth?
- The first thing to do is to try to reduce the multiplicative depth of the circuit
- For example (actually quite common in practice), suppose you have k ciphertexts you want to multiply together
    - Again, we technically mean here that you want to end up with a ciphertext whose plaintext is the product of the k plaintexts for the given ciphetexts
    - Not that you're literally multiplying together the ciphertexts

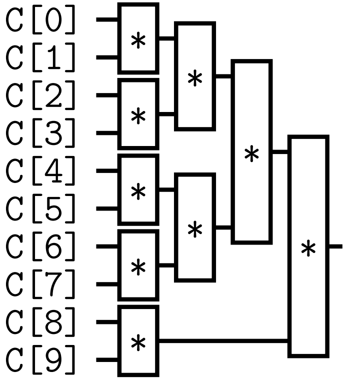
Multiplying k ciphertexts

```
R = C[0]
for i = 1, 2, ..., k-1:
  R = Mult(R, C[i])
return R
```



What is the multiplicative depth?

Multiplying *k* ciphertexts



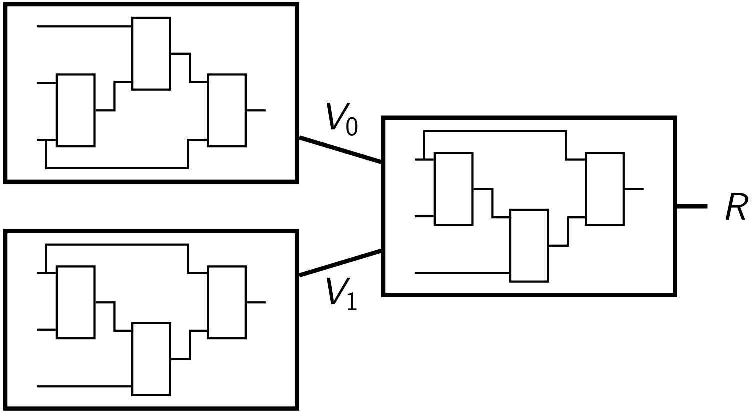
What is the multiplicative depth?

What if it's still large?

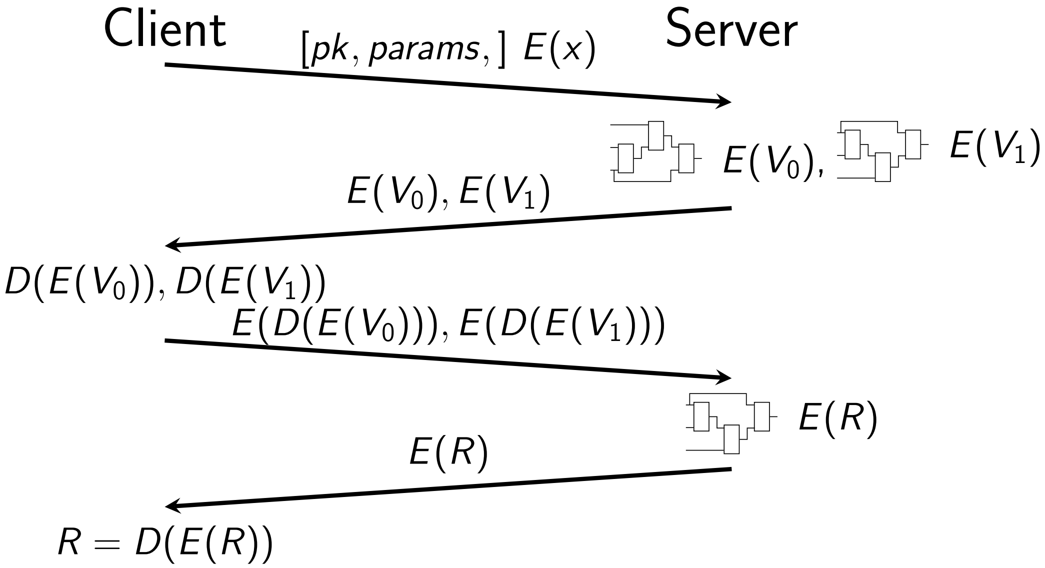
- After you've reduced the multiplicative depth of the circuit as much as you can, the depth might still be quite large
- In *theory*, you can just set the maximum depth for the levelled encryption scheme as high as you like
- But then *each* operation will become extremely slow and expensive!

Breaking the computation into chunks

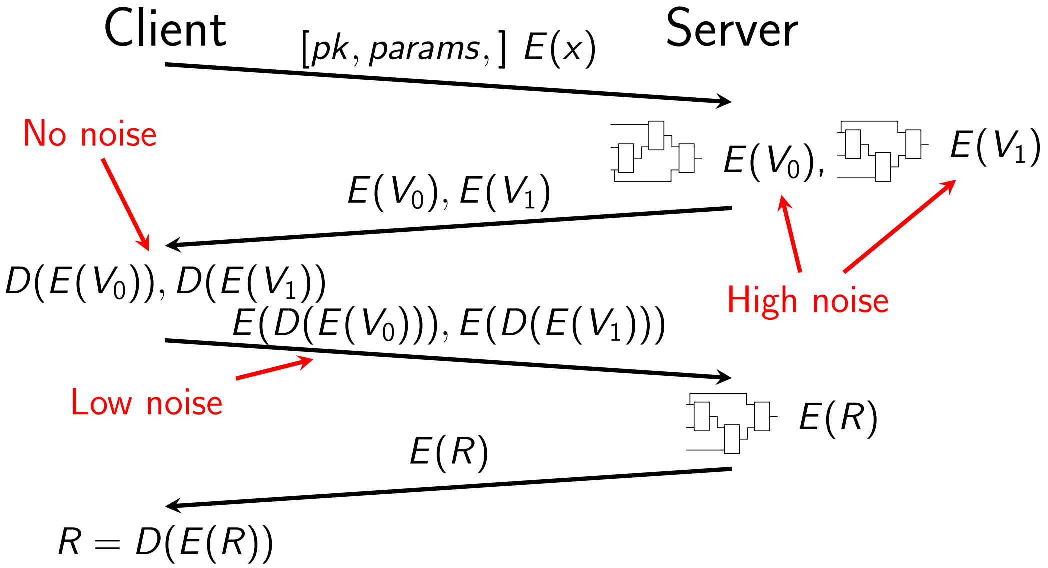
- The next approach is to divide up the program P (or its circuit) into smaller chunks, each of which has small multiplicative depth



Chunked computation



Chunked computation



Fully homomorphic encryption

- The second part of Gentry's work was to find a levelled homomorphic encryption system that was capable of bootstrapping
- Levelled homomorphic encryption $+$ bootstrapping $=$ fully homomorphic encryption (FHE)
- With FHE, you can compute *any* circuit on encrypted data without being able to decrypt it
    - Noninteractively

Removing the interaction

- The purpose of the interaction was because the server had a ciphertext $E(V)$ (of an intermediate value V) with high noise
    - The result of computing a chunk of the circuit with the maximum multiplicative depth d
- The server needs a fresh encryption (of the same value V) with low noise
    - So that it can use it in the next chunk of the circuit
- The way to remove noise is *decryption*
    - But the client has the decryption key (sk) and the server does not

Removing the interaction

- Let's write the secret (private) key and public key explicitly:
- The server has a ciphertext $C = E_{pk}(V)$ with high noise, sends that to the client
- The client has the secret key sk and computes $D_{sk}(C)$, which equals V (removing the noise)
- The client re-encrypts the result with pk to yield $E_{pk}(D_{sk}(C))$, which is $E_{pk}(V)$, but with low noise

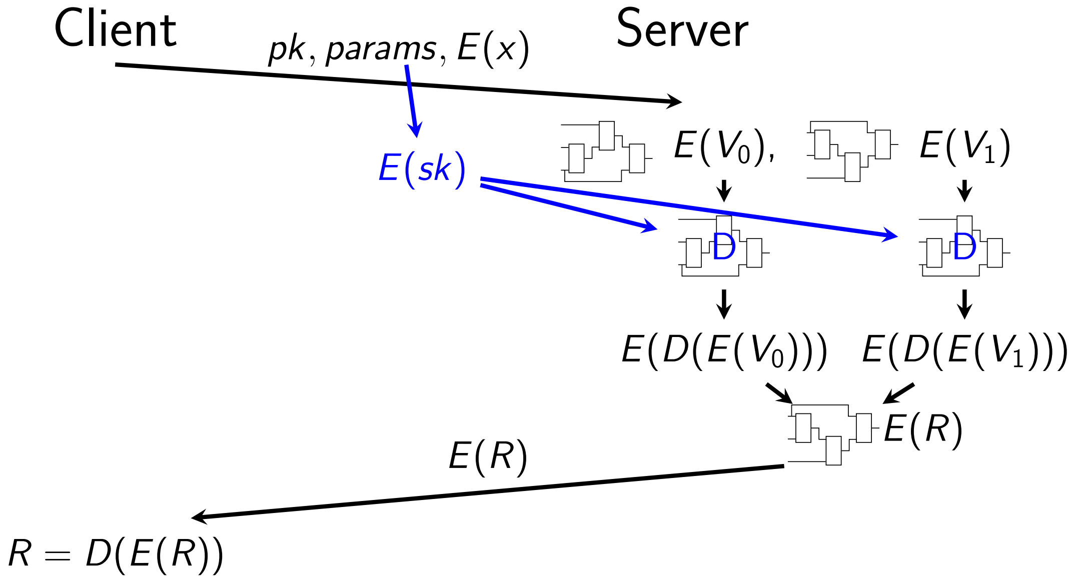
Removing the interaction

- The reason we did the interaction is that the server has C, and wants to compute $E_{pk}(D_{sk}(C))$, but only the client knows sk
- Key observation: $D_{sk}(C)$ is just a program with two inputs: sk is known by the client, and C is known by the server
- Recall: for a program P (of multiplicative depth at most d),
  - If the client has input x and the server has input Y,
  - then the client sends $E_{pk}(x)$ to the server,
  - and the server can compute $E_{pk}(P(x, Y))$

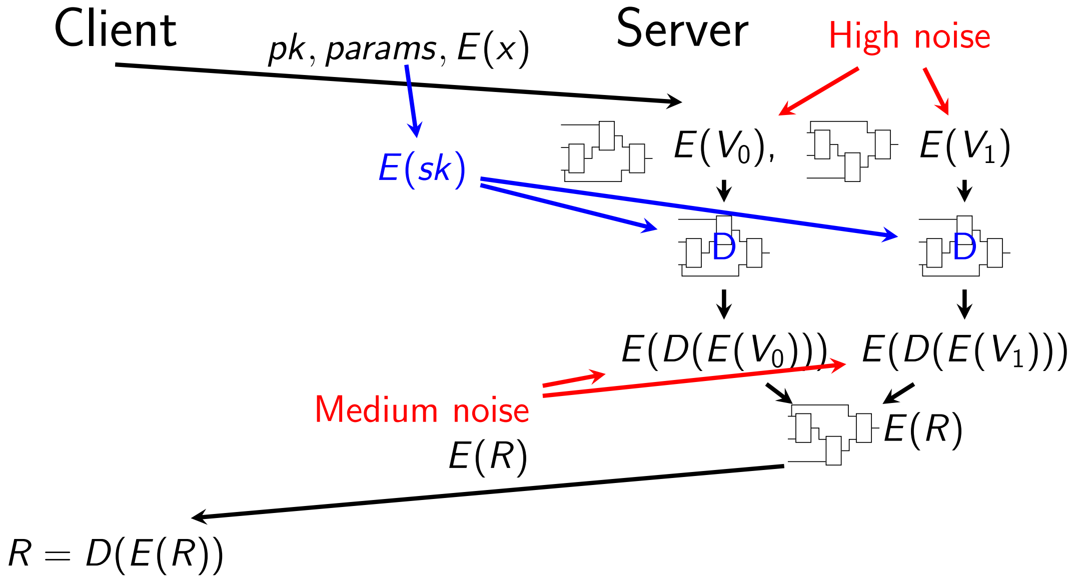
Removing the interaction

- The reason we did the interaction is that the server has C, and wants to compute $E_{pk}(D_{sk}(C))$, but only the client knows sk
- Key observation: $D_{sk}(C)$ is just a program with two inputs: sk is known by the client, and C is known by the server
- Recall: for a program D (of multiplicative depth at most d),
  - If the client has input sk and the server has input C,
  - then the client sends $E_{pk}(sk)$ to the server,
  - and the server can compute $E_{pk}(D_{sk}(C))$

Bootstrapping



Bootstrapping



Bootstrapping

- Bootstrapping works when the decryption circuit for a levelled homomorphic encryption scheme (with maximum multiplicative depth d) itself has multiplicative depth *less than d*

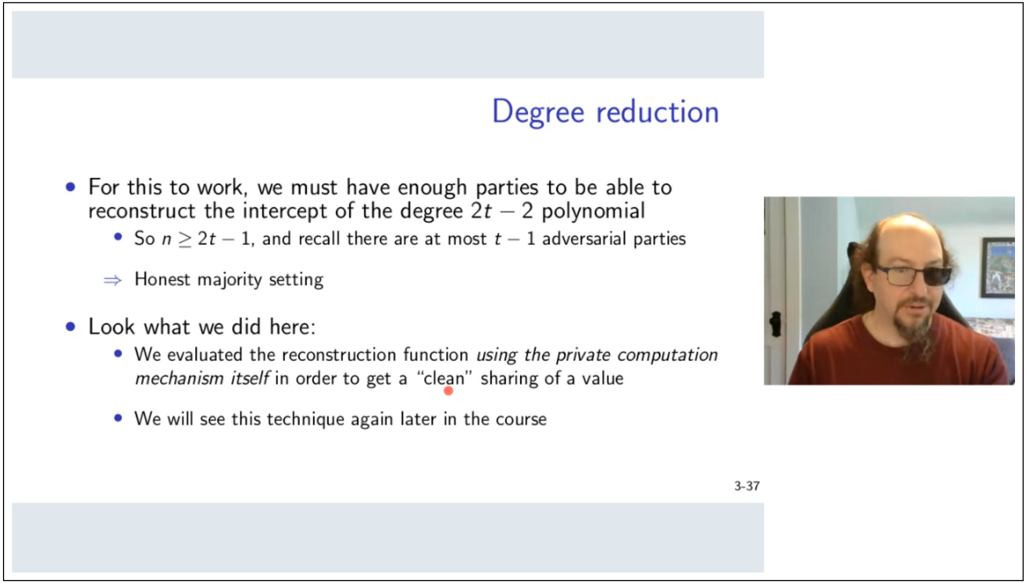
- Say $d = 20$, and the decryption circuit has multiplicative depth 12
  - Then the first chunk can be of multiplicative depth 20
  - The result of that chunk is treated as a *plaintext* input to the decryption circuit
  - The output of the decryption circuit will have had depth 12 multiplications already
  - The next chunk can be of multiplicative depth 8 before you have to decrypt again

  $\Rightarrow$ What happens after that?

Bootstrapping

- What did we do here?
- We evaluated the plaintext reconstruction function *using the private computation mechanism itself* in order to get a “clean” (lower-noise) encryption of a value
- Where have we seen this technique before?

Bootstrapping



Degree reduction

- For this to work, we must have enough parties to be able to reconstruct the intercept of the degree $2t-2$ polynomial
  - So $n \geq 2t-1$, and recall there are at most $t-1$ adversarial parties

  $\Rightarrow$ Honest majority setting
- Look what we did here:
  - We evaluated the reconstruction function *using the private computation mechanism itself* in order to get a "clean" sharing of a value
  - We will see this technique again later in the course

3-37

Using homomorphic encryption

- As usual, we won't be talking about how homomorphic encryption works "on the inside"
- But we will talk about how to use it (the API)
  - Not the API for a specific library, but in general
  - You will be using a specific library (openfhe) on Assignment 3
  - But even there, the exact API differs a bit between the C++, Python, and Rust versions

Setup

- Start by choosing what homomorphic encryption scheme you're going to use
    - e.g, BFV, BGV, FHEW, CKKS
- Libraries like openfhe and others support multiple schemes
- Do you need only levelled homomorphic encryption, or do you need bootstrapping (FHE) as well?
    - As a running example, we will use the BFV levelled homomorphic scheme, without bootstrapping
    - Also sometimes called "B/FV" or just "FV"
    - This is what you'll be using on Assignment 3

Choosing scheme parameters

- The next step is to choose the parameters of the scheme that you want
- You're likely to choose the maximum multiplicative depth d
  - Remember that the larger d is, the larger your ciphertexts are, and the slower your operations are, so keep d as small as your intended circuits permit
  - Count the depth of the multiplication gates in your circuit carefully!
- In some schemes (like BFV), you will also choose the *plaintext modulus n*
  - Plaintexts (in the simple case, but more on this soon) are integers mod n
  - So kind of like scalars in the group API, but here, n is typically pretty small (65537 is a common value)

Key generation

- Then you'll generate your public/private key pair
- As usual, the public key is for encrypting plaintexts to form ciphertexts, and the private key is for decrypting ciphertexts to recover plaintexts

Public parameters

- You will also generate the public parameters we've talked about before
- For example, in order for a server to multiply two ciphertexts, it needs a "multiplication key" given to it as part of the public parameter set
- If you want to enable bootstrapping, then $E_{pk}(sk)$ will be part of the public parameters
- The scheme parameters you chose earlier will also be part of the public parameters

Serialization

- Not specific to homomorphic encryption, but any multi-party protocol will need a way to turn objects in memory (e.g., private keys, public keys, ciphertexts, public parameters) into sequences of bytes so that they can be stored for later use or sent to another party (*serialization*)

- Turning the sequences of bytes back into their corresponding objects is *deserialization*

Client-side operations

- On the client side, there will be operations for:
- Constructing plaintexts from data
    - More on this in a bit
- Using the public key to encrypt plaintexts
- Using the private key to decrypt ciphertexts
- Extracting data from plaintexts

Server-side operations

- On the server side, there will be operations for:
- Add, subtract, multiply ciphertexts by ciphertexts
- Add, subtract, multiply ciphertexts by plaintexts
- (If needed) perform bootstrapping
- Other useful operations can often be built using these basic functions

What are the plaintexts?

- Since there are operations on ciphertexts that result in adding, subtracting, and multiplying the corresponding plaintexts, the plaintexts must form a *ring*
  - A *finite* ring, of course
- So at minimum, it's $\mathbb{Z}_n$ for some n
  - If $n = 2$, then you're just encoding single bits, and the homomorphic operations will be XOR and AND
- As we saw before, for BFV for example, you can choose your values of n when you set up the scheme (but n must have certain properties)
  - 65537 is a common choice

Plaintext packing

- So the simplest thing to do is just have each plaintext be (in this example) an integer in the range $0, 1, \ldots, 65536$
- However, ciphertexts can be quite large (more than 1 MB), and it could be a bit wasteful to only put about 2 bytes of useful data in there
- Homomorphic encryption schemes like BFV allow you to do *plaintext packing*: bundle multiple (typically thousands of) $\mathbb{Z}_n$ values together into a single ciphertext

Plaintext packing

- In BFV, plaintexts are actually not just $\mathbb{Z}_n$
- They are polynomials of degree less than N, whose coefficients are in $\mathbb{Z}_n$
- This is the ring $\frac{\mathbb{Z}_n[x]}{(x^N+1)}$ we saw earlier
  - For example, with $n = 65537$ and $N = 16384$, you can put 16384 values, each in $\mathbb{Z}_{65537}$, into a single plaintext
- But if you don't want to do that, you can just use elements of $\mathbb{Z}_n$, which are really polynomials of degree 0
  - They have a constant term in $\mathbb{Z}_n$, but all the other coefficients are 0

Plaintext packing

- There are actually two different ways to pack multiple $\mathbb{Z}_n$ values into a single plaintext
- They differ in what happens when you multiply the plaintexts together, and what kinds of operations (in addition to addition, subtraction, and multiplication) you can do homomorphically

Polynomial packing

- In the first method, the $\mathbb{Z}_n$ values are the *coefficients* of the plaintext polynomial in $\frac{\mathbb{Z}_n[x]}{(x^N+1)}$
- Suppose one ciphertext encrypts the values 2, 5, 4, and a second ciphertext encrypts 3, 1, 8
- Then the corresponding plaintexts are the polynomials $2 + 5x + 4x^2$ and $3 + x + 8x^2$
- If the server adds the ciphertexts, it results in an encryption of $5 + 6x + 12x^2$
  - So just adding the corresponding plaintext entries
  - Similar for subtraction

Polynomial packing

- If the server multiplies the ciphertexts, the result is the multiplication of the plaintexts as polynomials
- So the resulting ciphertext would be an encryption of

$$(2 + 5x + 4x^2)(3 + x + 8x^2) = 6 + 17x + 33x^2 + 44x^3 + 32x^4$$

- If the product has degree N or more, then it also does degree reduction, as discussed earlier

Operations with polynomial packing

- In addition to addition, subtraction, and multiplication, there are a couple of other operations the server can do on ciphertexts when polynomial packing is used
- The first looks weird, and doesn't seem very useful on its own, but is used to construct the second, which is extremely useful

Substitution

- The first additional homomorphic operation (when using polynomial packing) is *substitution*
- This operation takes a ciphertext encrypting a polynomial $f(x)$ and turns it into a ciphertext encrypting the polynomial $f(x^r)$ for some r
  - In the common case of N being a power of 2, r can be any odd integer
- For example, a ciphertext encrypting $2 + 5x + 4x^2$ can be turned into a ciphertext encrypting $2 + 5x^3 + 4x^6$ (by choosing $r = 3$)

Expansion

- As mentioned, this doesn't seem immediately useful. But it's the key ingredient to building *homomorphic expansion*
- This operation takes a ciphertext encrypting a polynomial $f(x)$ of degree $k-1$ and turns it into k ciphertexts, each encrypting one of the coefficients of $f(x)$
- For example, a ciphertext encrypting $2+5x+4x^2$ can be turned into three ciphertexts, encrypting 2, 5, and 4, respectively
  - But: $\lceil \lg k \rceil$ multiplicative depth

Vector packing

- The other way to pack multiple $\mathbb{Z}_n$ values into one plaintext is as a *vector*
- Technically, the elements of the vector are *evaluations* of the plaintext polynomial in $\frac{\mathbb{Z}_n[x]}{(x^N+1)}$, but that's an "inside" detail that's not exposed to the API

Vector packing

- Suppose one ciphertext encrypts the values [2, 5, 4], and a second ciphertext encrypts [3, 1, 8]
- If the server adds the ciphertexts, it results in an encryption of [5, 6, 12]
  - So just adding the corresponding plaintext entries, just like polynomial packing
  - Similar for subtraction
- If the server multiplies the ciphertexts, it results in an encryption of [6, 5, 32]
  - Element-wise multiplication
  - Often easier to deal with than polynomial packing

Vector packing

- The vectors are actually of length $N/2$ (so thousands of entries long), but most of the entries are 0 if you're not using them
- The actual length of the vector becomes important, however, when you do the additional homomorphic operation available with vector packing: *vector rotation*

Vector rotation

- Vector rotation is a homomorphic operation that, given a ciphertext encrypting a certain vector, produces a ciphertext that encrypts that vector, rotated by r positions to the left, for any chosen r

- For example, given a ciphertext that encrypts the vector [2, 5, 4], the server can perform the homomorphic vector rotation operation to yield a ciphertext that is an encryption of the vector:
    - For $r = -1$: $[0, 2, 5, 4]$
    - For $r = -2$: $[0, 0, 2, 5, 4]$
    - For $r = 1$: $[5, 4, 0, \ldots, 0, 2]$
    - For $r = 2$: $[4, 0, \ldots, 0, 2, 5]$

PIR using homomorphic encryption

- We will next look at how to implement PIR in the *single-server* setting, *without trusted hardware*, using homomorphic encryption
- This is a quite active research area!
- Some recent protocols include Spiral, SimplePIR, DoublePIR, FrodoPIR
- We will only look at the details of the simplest HE-based PIR protocols

The database as a matrix

Recall from Module 3:

- Most PIR protocols will model the database D as a matrix
  - For example, a matrix with r rows, each of length s bytes
  - The i^{th} row of the matrix is the i^{th} record of the database

$$D = \begin{bmatrix} \text{Sealing assembly for} \ldots \\ \text{Adjustable-backset} \ldots \\ \text{Conical recreational} \ldots \\ \text{Method of swinging} \ldots \\ \text{Cover for the rails} \ldots \\ \text{Golf ball delivery} \ldots \end{bmatrix}$$

- If you write your query like this: $q = [\,0\ 0\ 0\ 1\ 0\ 0\,]$ then what is $q \cdot D$?

The database as a matrix

- In Module 3, we wrote the database as a matrix of r rows, where each row was one s-byte record
- We will do something very similar here, only instead of the rows being s bytes, they will be some number of *plaintexts*
- Let w be the number of bytes you can fit into one plaintext
    - This will depend on n, N, and whether you use polynomial or vector packing
    - For example, if $n = 65537$, $N = 16384$, and polynomial packing, then $w = 32768$

The database as a matrix

- Then the database matrix will be much as before: r rows, each row being $\lceil \frac{s}{w} \rceil$ plaintexts

$$D = \begin{bmatrix} \text{Sealing assembly for } \ldots \\ \text{Adjustable-backset } \ldots \\ \text{Conical recreational } \ldots \\ \text{Method of swinging } \ldots \\ \text{Cover for the rails } \ldots \\ \text{Golf ball delivery } \ldots \end{bmatrix} = \begin{bmatrix} p_{0,0} & p_{0,1} & \cdots & p_{0,8} \\ p_{1,0} & p_{1,1} & \cdots & p_{1,8} \\ p_{2,0} & p_{2,1} & \cdots & p_{2,8} \\ p_{3,0} & p_{3,1} & \cdots & p_{3,8} \\ p_{4,0} & p_{4,1} & \cdots & p_{4,8} \\ p_{5,0} & p_{5,1} & \cdots & p_{5,8} \end{bmatrix}$$

The database as a matrix

- Since w is somewhat large, it may be that $s < w$, so each row only has a single plaintext, and the database is actually just a column vector of plaintexts

$$D = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{bmatrix}$$

The most basic HE-based PIR scheme

- Remember that if $q = [\ 0\ 0\ 0\ 1\ 0\ 0\ ]$, then $q \cdot D$ is just the desired row of the database

- So the client constructs this query vector, and encrypts each element: $E(q) = [\ E(0)\ E(0)\ E(0)\ E(1)\ E(0)\ E(0)\ ]$
  - Important: the instances of $E(0)$ aren't just literal copies of each other! They're separately encrypted values

- Send this vector of ciphertexts to the server (along with public parameters)
  - You could also use plaintext packing to pack many of the entries (mostly 0's and maybe the single 1) into single plaintexts before encryption
  - The server would then extract them back into individual ciphertexts before proceeding

The most basic HE-based PIR scheme

- The server now has $E(q) = [\ E(0)\ E(0)\ E(0)\ E(1)\ E(0)\ E(0)\ ]$ and D (and the public parameters that allow it to do homomorphic operations)

- The server now just computes $E(q) \cdot D$, where the multiplications are between ciphertexts and plaintexts, and the additions are of ciphertexts

- Because of the homomorphic property, $E(q) \cdot D = E(q \cdot D)$

- The server sends this back to the client, who decrypts it

The most basic HE-based PIR scheme

$$E(q) = [\ E(0)\ E(0)\ E(0)\ E(1)\ E(0)\ E(0)\ ]$$

$$D = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \\ p_4 \\ p_5 \end{bmatrix}$$

$$\begin{aligned} E(q) \cdot D &= E(0) \cdot p_0 + E(0) \cdot p_1 + E(0) \cdot p_2 + E(1) \cdot p_3 + E(0) \cdot p_4 + E(0) \cdot p_5 \\ &= E(p_3) \end{aligned}$$

The most basic HE-based PIR scheme

- What is the multiplicative depth of this scheme?
- How much data is sent from the client to the server?
- How much data is sent from the server to the client?

PIR from equality tests

- The previous scheme sends *a lot* of data
- We'll next look at a way to send significantly less
- Suppose $Eq(Q, y)$ is a function that takes as input a ciphertext vector Q (encrypting a number x represented in some fashion) and an unencrypted number y
  - Simple option: if x written in binary is 010011, then Q might be $[E(1), E(1), E(0), E(0), E(1), E(0)]$
  - There are other options
- $Eq(Q, y)$ outputs a *ciphertext*: it is an encryption of 1 if $x = y$ and an encryption of 0 otherwise

PIR from equality tests

- To do PIR, the client has the index x it wants to look up, and produces the corresponding encrypted Q
- The client sends Q (and the public parameters, etc.) to the server
- The server computes $R = \sum_{y=0}^{r-1} Eq(Q, y) \cdot D_y$, where D_y is row y of the database matrix, and sends R back to the client
- The client decrypts R to reconstruct the desired row of the database matrix

Keyword PIR from equality tests

- To do PIR, the client has the keyword x it wants to look up, and produces the corresponding encrypted Q
- The client sends Q (and the public parameters, etc.) to the server
- The server computes $R = \sum_{y \in \text{keywords}} Eq(Q, y) \cdot D_y$, where D_y is the value associated with the keyword y, and sends R back to the client
- The client decrypts R to reconstruct the desired row of the database matrix

Implementing the equality test

- How do you implement $Eq(Q, y)$?
- It depends on how you created Q, given x
- For example, you may do it using the "simple option" we talked about before (write x in binary, and entry i of the vector Q is an encryption of bit i of x, where bit 0 is the least significant bit)
- Then $Eq(Q, y) = \prod_{y_i=0}(1 - Q_i) \prod_{y_i=1} Q_i$, where y_i is bit i of y

Implementing the equality test

- What is the multiplicative depth of $Eq(Q, y)$ using this simple option of creating Q?
- What is the resulting multiplicative depth of the PIR scheme?
- How much data is sent from the client to the server?
- How much data is sent from the server to the client?

Constant-weight codewords

- There are better options for turning a value x into a ciphertext vector Q, however
- On Assignment 3, you will explore using *constant-weight codewords*
- General idea: before, we just wrote x in binary, and encrypted each bit to get the elements of Q
- Now: write x using just 0's and 1's, but in a special way: given a parameter k, then for every x, you write it so that there are *exactly* k 1's (and the rest are 0's)

Constant-weight codewords

x	cw	Q
0	1 1 0 0 0	[E(1), E(1), E(0), E(0), E(0)]
1	1 0 1 0 0	[E(1), E(0), E(1), E(0), E(0)]
2	0 1 1 0 0	[E(0), E(1), E(1), E(0), E(0)]
3	1 0 0 1 0	[E(1), E(0), E(0), E(1), E(0)]
4	0 1 0 1 0	[E(0), E(1), E(0), E(1), E(0)]
5	0 0 1 1 0	[E(0), E(0), E(1), E(1), E(0)]
6	1 0 0 0 1	[E(1), E(0), E(0), E(0), E(1)]
7	0 1 0 0 1	[E(0), E(1), E(0), E(0), E(1)]
8	0 0 1 0 1	[E(0), E(0), E(1), E(0), E(1)]
9	0 0 0 1 1	[E(0), E(0), E(0), E(1), E(1)]

Constant-weight codewords

- Since we know Q will have exactly k elements that will be encryptions of 1 (and the rest will be encryptions of 0), the equality test is much simpler: $Eq(Q, y) = \prod_{cw(y)_i = 1} Q_i$
- What is the multiplicative depth of $Eq(Q, y)$ using this simple option of creating Q?
- What is the resulting multiplicative depth of the PIR scheme?
- How much data is sent from the client to the server?
- How much data is sent from the server to the client?