


**Network Security**  
**Professor Gaurav S. Kasbekar**  
**Department of Electrical Engineering**  
**Indian Institute of Technology, Bombay**  
**Week - 12**  
**Lecture - 70**  
**Cloud Security: Part 4**

Hello, recall that in the previous few lectures, we discussed the basics of cloud computing, and then we discussed cloud security risks and countermeasures. In this lecture, we'll discuss a type of encryption scheme that is suitable for cloud computing, namely homomorphic encryption. With homomorphic encryption, the idea is this: data can be encrypted in a special way, and computation can be done on the encrypted data, and the result is the encrypted answer. As a simple example, consider two variables, A and B, and their encrypted values are  $K(A)$  and  $K(B)$ . Suppose the encrypted values of A and B, namely  $K(A)$  and  $K(B)$ , are stored in the cloud; we want to add A and B together.

Normally, we would have to decrypt  $K(A)$  and  $K(B)$  to get back A and B, then add A and B to get  $A+B$ , and then encrypt the result again to get  $K(A+B)$ . But with homomorphic encryption, we can just perform operations on the encrypted data itself. So, we can find out  $K(A) + K(B)$ , and the result will be the same as  $K(A+B)$ . So, this illustrates the fact that the data can be encrypted—A and B can be encrypted to get  $K(A)$  and  $K(B)$ —and we can perform computation on the encrypted data. So, the computation here is an addition operation. We perform computation on the encrypted quantities, that is,  $K(A)$  and  $K(B)$ .

- With homomorphic encryption, the data can be encrypted (in a special way), computation can be done on the encrypted data, and the result is the encrypted answer
- So, computing on the plaintext data will yield the same answer as computing on the encrypted data and decrypting the result
- An application of homomorphic encryption is to store one's encrypted data in a public cloud and do computations on the encrypted data, without needing to trust the cloud not to leak one's data
- Note that any computation can be done with a circuit consisting of  $\wedge$  (AND) and  $\oplus$  (XOR)

$$\begin{array}{c} A \quad B \\ K(A) + K(B) \\ = K(A+B) \end{array}$$


We add the encrypted quantities together, and since the encryption is done in a special way—namely, homomorphic encryption—the result is the encrypted answer. The result that we get is  $K(A+B)$ . So, it is just as if we had decrypted  $K(A)$  and  $K(B)$  to get  $A$  and  $B$ , then added them, and then re-encrypted the result. So, that's the advantage of homomorphic encryption. Computing on the plaintext data will yield the same answer as computing on the encrypted data and decrypting the result. So, if we had done an addition operation on  $A$  and  $B$ —that is, if we had computed  $A+B$ —then the result is the same as performing a computation on the encrypted data and decrypting the result.

If we find out  $K(A+B)$  and decrypt the result, then we get  $A+B$ , which is the same as computing on the plaintext data. So, what is the relevance of homomorphic encryption to cloud computing? An application of homomorphic encryption is to store one's encrypted data in a public cloud and do computations on the encrypted data without requiring to trust the cloud not to leak one's data. Suppose a user has stored their encrypted data in the cloud. So, the user has stored things like  $K(A)$  and  $K(B)$  in the cloud.

If the user decrypts this data on the cloud itself, then the cloud provider can see the original data, that is, the unencrypted data. But if homomorphic encryption is used, then the user can perform operations on the encrypted data itself and get the result that is desired. There is no need to decrypt it. Hence, the data remains safe from the cloud computing provider. So, the user does not need to trust the cloud. Hence, homomorphic encryption is useful in the context of cloud computing.

Now, we discuss how we can generalize homomorphic encryption. Any computation can be done with a circuit consisting of AND and XOR gates. So, we want to show how we can perform a general computation under homomorphic encryption. My claim is that any computation can be done with a circuit consisting of only AND gates and XOR gates. To see this, recall that a NAND gate is a universal gate.

We know that any logical operation can be done using a set of NAND gates. Now, if we have an AND gate and an XOR gate, notice that we can construct a NAND gate using an AND gate and an XOR gate. Because we have an XOR gate, so, if you tie one of its inputs to one, then if  $X$  is the other input, then notice that the result is  $\bar{X}$ . So, if you have an XOR gate and one of the inputs is tied to one and the other input is  $X$ , then it is easy to see that the output is  $\bar{X}$ . Hence, we can construct an inverter out of an XOR gate.

So, an inverter along with a NAND gate gives us a NAND gate. And we know that NAND gate is a universal gate. We can construct any logical operation using a set of NAND gates.

Hence, this shows that any computation can be done with a circuit consisting of only AND and XOR gates. Now, we'll show an encryption scheme and then we'll show that how to do AND and XOR operations on the data, on the encrypted data.

We'll show how to do AND and XOR operations on the plaintext data by performing some operations on the encrypted data. Because of this result that we just discussed, to achieve a homomorphic encryption scheme, we need to use some mathematics that allows both the above operations on encrypted data. We now discuss a simple homomorphic encryption scheme. Most fully homomorphic encryption schemes involve complicated mathematics. But to gain some intuition, we discuss one scheme that is easy to understand.

So, this scheme that we discussed is not practical. For example, relative to the plain text data, the encrypted data would expand by a factor of about a billion. Hence, it is not useful in practice. The scheme requires encryption to be done on each bit. So, we'll show how to encrypt a zero and how to encrypt a one.

Then, if you have any binary string, we can separately encrypt each zero and one. So, we will discuss how to encrypt 0 and how to encrypt 1. Now, there is a private key in this scheme. The private key is a large odd integer. Let us call it 'n'. The private key is only known to the party who is going to decrypt the data.

So, this scheme performs ordinary integer arithmetic and not modular arithmetic. So, this is the encryption process to encrypt a bit if we know the private key n. First, assume that we know the private key n; later we'll discuss how to encrypt a bit without knowing the private key n. So, to encrypt a bit if we know n, we choose some very large multiple of n, say  $k \times n$ , where k is a large positive integer. Then, we add or subtract a relatively small even number, which we call noise. So, the value obtained at this point is a noisy multiple of n. It is some multiple of n with a small even number added to it, which is the noise part. Hence, we call this value a noisy multiple of n. If we are encrypting a zero, we are now done.

Notice that the encrypted bit is of this form: ciphertext of the zero that we encrypted is  $kn \pm e$ . So, k is a large positive integer,  $kn$  is some very large multiple of n, and then we either add or subtract a small even number, which is denoted here by e. So, the encrypted bit when the plaintext is zero is of this form:  $c = kn \pm e$ . If the bit to be encrypted is a 1, then we add or subtract 1 to the result that we got here. So, in that case, the encrypted bit is of this form:  $c = kn \pm e \pm 1$ . So, this is the encryption process. If the plaintext is a 0, then we compute  $c = kn \pm e$ , and that is the ciphertext corresponding to the 0, and if the

plaintext is a 1, then we compute  $c = kn \pm e \pm 1$ , and that's the ciphertext corresponding to the 1.

## Simple FHE Scheme

- To encrypt a bit if you know  $n$ :
  - ☐ Choose some very large multiple of  $n$
  - ☐ Add or subtract a relatively small even number (which we will call "noise")
    - We call the value at this point a noisy multiple of  $n$
    - If you are encrypting a 0, you are now done
    - Encrypted bit of form:  $c = kn \pm e$
  - ☐ If the bit to be encrypted is a 1, add or subtract 1
    - Encrypted bit of form:  $c = kn \pm e \pm 1$
- In order to allow someone who does not know  $n$  to encrypt:
  - ☐ We create a public key that is a list of encryptions of 0
  - ☐ To encrypt a 0, one would add together a randomly chosen subset of the encryptions of 0
    - E.g., if  $c_1 = k_1n + e_1$ ,  $c_2 = k_2n + e_2$ , and  $c_3 = k_3n + e_3$ , then  $c_1 + c_2 + c_3 = (k_1 + k_2 + k_3)n + (e_1 + e_2 + e_3)$
  - ☐ To encrypt a 1, they would do the same thing, but add or subtract 1 at the end
- Only someone who knows  $n$  will be able to decrypt:
  - ☐ To decrypt a value  $x$ , find the nearest multiple of  $n$
  - ☐ If the difference between  $x$  and the multiple of  $n$  is even,  $x$  decrypts to a 0
  - ☐ If the difference is odd,  $x$  decrypts to a 1

Now, we have discussed how to encrypt the bit if someone knows the private key. Now, to allow someone who does not know the private key  $n$  to encrypt, we create a public key that is a list of encryptions of 0, and to encrypt the 0, we just select a random subset of the encryptions of 0 and add them together. So, suppose the randomly chosen subset of the encryptions of 0 are these. The first element of the subset is  $c_1 = k_1n + e_1$ . So, this is one encryption of 0.

Then the second element of the subset is  $c_2 = k_2n + e_2$ . This is another encryption of 0. And similarly, we have  $c_3 = k_3n + e_3$ . This is another encryption of 0. In this case, this subset consists of three elements.

We add up these elements to get  $c_1 + c_2 + c_3$  equals this. So, if you add all these, then we get  $(k_1 + k_2 + k_3)n + (e_1 + e_2 + e_3)$ . Now, notice that this is of the same form as an encryption of 0, and hence we have encrypted a 0. So, this is our ciphertext for a 0. And we have encrypted 0 without knowing the private key  $n$ . If you want to encrypt a 1, then we do the same thing, but we add or subtract a 1 at the end.

So, we'll get some number of this form, and that is an encryption of 1 which you have computed without knowing the private key  $n$ . So, now we are shown how to encrypt a bit without knowing the private key. And the public key is a list of encryptions of 0. Now,

only someone who knows  $n$  will be able to decrypt. So, how do we decrypt ciphertext which is either of this form or of this form? To decrypt a value  $x$ , we find the nearest multiple of  $n$ . So, for example, if  $x = kn + e$ , then the nearest multiple of  $n$  will be  $kn$ .

That's because, by choice,  $e$  is a very small number.  $e$  is much smaller than  $n$ . Hence, the nearest multiple of  $n$  to  $kn + e$  will be  $kn$ . Now if the difference between  $x$  and the multiple of  $n$  is even, then  $x$  decrypts to a 0, and if the difference is odd, then  $x$  decrypts to a 1. So, if  $x$  is of this form,  $kn \pm e$ , then the nearest multiple of  $n$  will be  $kn$ , and the difference between  $x$  and the nearest multiple will be just  $\pm e$ , which is even. Hence,  $x$  will decrypt to a 0 as required.

And if the encrypted bit is of this form,  $c = kn \pm e \pm 1$ , then the nearest multiple of  $n$  will be  $kn$ . And if we subtract this nearest multiple from  $x$ , then we get  $e \pm 1$ , which is an odd number. We get  $\pm e \pm 1$ , which is an odd number. Hence,  $x$  decrypts to a 1. So, that's the process for decryption.

Notice that we require the private key  $n$  to decrypt. And the decryption process is very simple. We just find the nearest multiple of  $n$ , remove that from the ciphertext  $x$ , and check whether the result is even or odd. So, this is a simple FHE scheme that we want to discuss. And now we have to show that it is indeed a fully homomorphic encryption scheme.

Let's see how to do different operations using this scheme. Adding two encrypted bits results in the XOR of the two plaintext bits. That's the claim. So, let us prove this claim. We have to discuss how to perform XOR of the plaintext, and AND of the plaintext bits by just performing operations on the ciphertext. First, we claim that adding two encrypted bits results in the XOR of the two plaintext bits.

So, let's consider different cases. The first case is when the first ciphertext is the ciphertext corresponding to 0 and the second ciphertext is corresponding to 0 as well. So,  $c_1 = k_1n + e_1$  and  $c_2 = k_2n + e_2$ . So, this is the ciphertext corresponding to 0 and this is also ciphertext corresponding to 0. Then  $c_1 + c_2 = (k_1 + k_2)n + (e_1 + e_2)$ .

So, this is also ciphertext corresponding to 0. This is consistent with the fact that  $0 \oplus 0 = 0$ . Hence, this is consistent with this property of XOR. Hence, we have successfully XORed the two plaintext bits by just operating on the ciphertext. Now, if  $c_1 = k_1n + e_1$  and  $c_2 = k_2n + e_2 + 1$ , in this case,  $c_1$  is the ciphertext corresponding to 0 and  $c_2$  is the ciphertext corresponding to 1.

In that case,  $c_1 + c_2$  is this. It is an encryption of 1. And this is consistent with this property of XOR,  $0 \oplus 1 = 1$ . Now, let's check the remaining possible case. If  $c_1$  is the encryption of 1 and  $c_2$  is also an encryption of 1, then  $c_1 + c_2 = (k_1 + k_2)n + (e_1 + e_2 + 2)$ .

- Adding two encrypted bits results in the  $\oplus$  (XOR) of the two plaintext bits
  - If  $c_1 = k_1n + e_1$  and  $c_2 = k_2n + e_2$ , then  $c_1 + c_2 = (k_1 + k_2)n + (e_1 + e_2)$   $0 \oplus 0 = 0$
  - If  $c_1 = k_1n + e_1$  and  $c_2 = k_2n + e_2 + 1$ , then  $c_1 + c_2 = (k_1 + k_2)n + (e_1 + e_2) + 1$   $0 \oplus 1 = 1$
  - If  $c_1 = k_1n + e_1 + 1$  and  $c_2 = k_2n + e_2 + 1$ , then  $c_1 + c_2 = (k_1 + k_2)n + (e_1 + e_2 + 2)$   $1 \oplus 1 = 0$
- Multiplying two encrypted bits results in the  $\wedge$  (AND) of the two bits
  - If  $c_1 = k_1n + e_1$  and  $c_2 = k_2n + e_2$ , then  $c_1 \times c_2 = (k_1k_2)n^2 + (k_1e_2 + k_2e_1)n + (e_1e_2)$   $0 \wedge 0 = 0$
  - If  $c_1 = k_1n + e_1$  and  $c_2 = k_2n + e_2 + 1$ , then  $c_1 \times c_2 = (k_1k_2)n^2 + (k_1e_2 + k_2e_1 + k_1)n + (e_1e_2 + e_1)$   $0 \wedge 1 = 0$
  - If  $c_1 = k_1n + e_1 + 1$  and  $c_2 = k_2n + e_2 + 1$ , then  $c_1 \times c_2 = (k_1k_2)n^2 + (k_1e_2 + k_2e_1 + k_1 + k_2)n + (e_1e_2 + e_1 + e_2) + 1$   $1 \wedge 1 = 1$
- Adding or subtracting 1 to an encrypted bit computes the NOT of the encrypted bit

Notice that this is an even number. Hence,  $c_1 + c_2$  is of the form  $kn + e$ . So, this is the ciphertext corresponding to 0. And this third result is consistent with this property of XOR, that  $1 \oplus 1 = 0$ . So, these three results together show that if we add two encrypted bits, then that results in the XOR of the two corresponding plaintext bits. So, we can do XOR on the plaintext bits by just performing additions on the encrypted bits. Now, let's consider how to perform AND on the plaintext bits by just operating on the ciphertext.

So, the claim is that if you multiply two encrypted bits, then that results in the AND of the two bits. Let's consider different cases again. Suppose  $c_1$  is the encryption of 0, and  $c_2$  is the encryption of 0 as well. Then,  $c_1 \times c_2$  can be easily computed. It comes out to be this.

So, this is of the form  $kn + e$ , where  $e$  is an even number. So, this is consistent with the property of AND that  $0 \wedge 0 = 0$ . Now, as the next result, suppose  $c_1$  is an encryption of 0 and  $c_2$  is an encryption of 1; then,  $c_1 \times c_2$  comes out to be this. This is again of the form  $kn + e$ . So, that is, this result is consistent with this property of AND, that  $0 \wedge 1 = 0$ .

Now, as a remaining case, suppose  $c_1$  is an encryption of 1 and  $c_2$  is an encryption of 1; then,  $c_1 \times c_2$  comes out to be this. So, this quantity is of the form  $kn + e + 1$ , which is an encryption of 1. Hence, this result is consistent with the property of AND, that  $1 \wedge 1 = 1$ . So, these three results together show that if we multiply two encrypted bits, then that results in the AND of the two corresponding plaintext bits. So, we can perform AND operations on the plaintext without having to decrypt the ciphertext.

We can just do operations on the ciphertext and achieve the AND of the corresponding plaintext. We have shown that our proposed scheme is indeed a fully homomorphic encryption scheme. We can do all logical operations on the plaintext by just doing operations on the ciphertext. We can also easily perform the NOT of a certain plaintext by just doing operations on the ciphertext. Adding or subtracting 1 from an encrypted bit computes the NOT of the encrypted bit.

So, this is left to you as a simple exercise. So, we can also do a NOT operation on the plaintext by just performing operations on the ciphertext. So, this proves that the proposed scheme is a fully homomorphic encryption scheme. Now, some shortcomings of this scheme are as follows. The noise increases each time two encrypted bits are added or multiplied.

So, we can see here that earlier the noise was  $e_1$  or  $e^2$ , then when we added the ciphertext the noise became  $e_1 + e_2$ , which is larger than either of the noise. Similarly, when we multiply the noise, we get a large amount of noise,  $e_1 e_2$  in this case. Here, it is  $e_1 e_2 + e_1 + e_2$ . So, the noise increases each time we perform operations on the ciphertext. It can be shown that if the noise ever gets bigger than half of  $n$ , then decryption will no longer be guaranteed to produce the right answer.

- If the noise ever gets bigger than  $n/2$ , decryption will no longer be guaranteed to produce the right answer

So, we can do additions and multiplications, but we have to avoid letting the noise get bigger than  $n/2$ . So, this is the shortcoming of the scheme. Another limitation of the scheme is that the size of the encrypted bit doubles each time a multiplication is performed. So, we discussed multiplication in the previous slide. You can check that if we multiply two billion bit numbers together, then we get a two billion bit number.

So, after some multiplications the numbers which were large while starting get extremely large. Hence, the computational complexity of this scheme is very large. So, for all these reasons this scheme is not practical. And there are more practical homomorphic schemes proposed in the research literature and this is an area of active research. The design of practical homomorphic schemes.

So, in summary we discussed fully homomorphic encryption, which allows us to do operations on plaintext data by just performing operations on the corresponding ciphertext,

and this is useful in the context of cloud computing, where we can store some encrypted data on the cloud and do operations on the encrypted data without having to decrypt it. So, the data is safe from the cloud provider. So, in the case, where the cloud provider is dishonest, the data remains protected from the cloud provider. This concludes our discussion of cloud computing and its security. Thank you.