FHE II: Construction

Boaz Barak

**Disclaimer:** The MS-Word version of this text has significant formatting issues compared to the PDF and HTML versions.

# Fully homomorphic encryption: Construction {chapfhetwo }

In the last lecture we defined fully homomorphic encryption, and showed the “bootstrapping theorem” that transforms a partially homomorphic encryption scheme into a fully homomorphic encryption, as long as the original scheme can homomorphically evaluate its own decryption circuit. In this lecture we will show an encryption scheme (due to Gentry, Sahai and Waters, henceforth GSW) meeting the latter property. That is, this lecture is devoted to proving[[1]](#footnote-21) the following theorem:

# 

Assuming the LWE conjecture, there exists a partially homomorphic public key encryption that fits the conditions of the bootstrapping theorem (bootstrapthm). That is, for every two ciphertexts and , the function can be homomorphically evaluated by .

## Prelude: from vectors to matrices

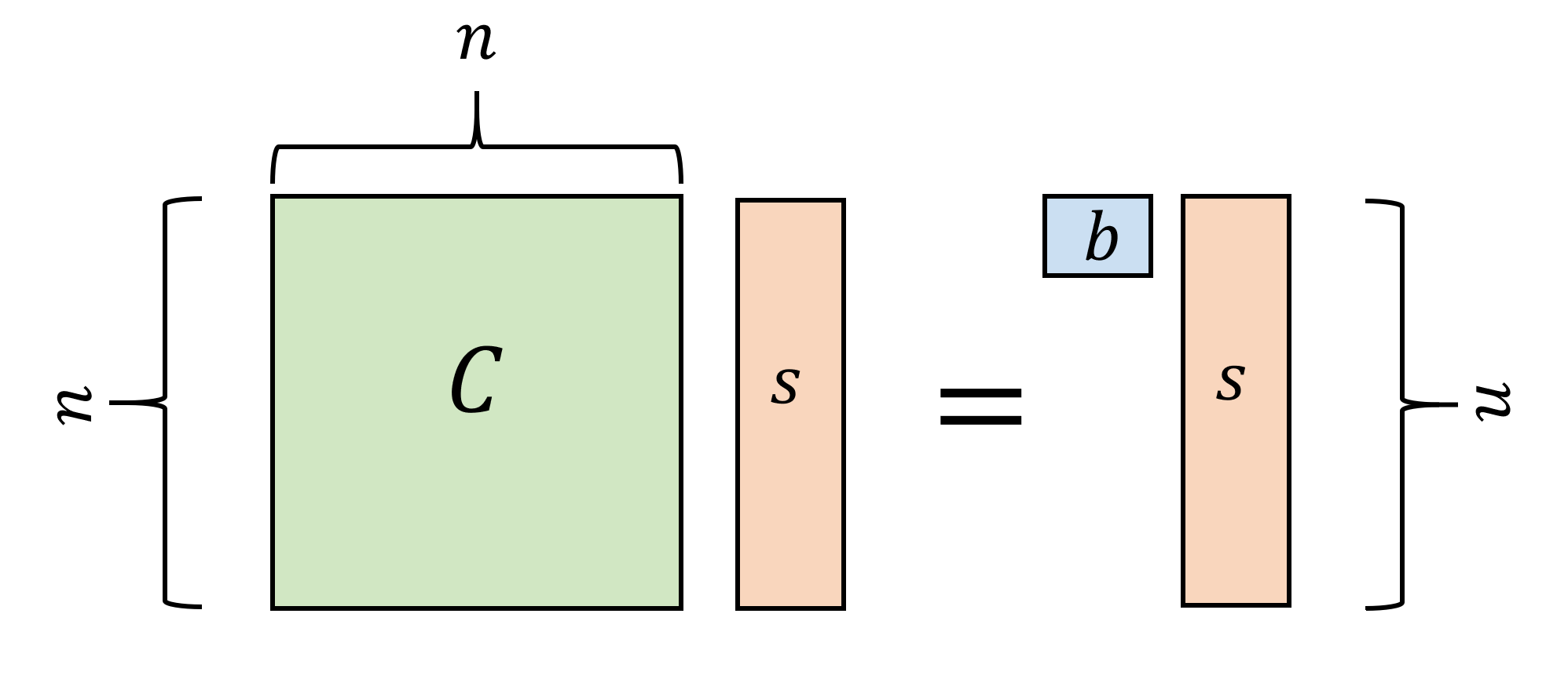
In the linear homomorphic scheme we saw in the last lecture, every ciphertext was a vector such that equals (up to scaling by ) the plaintext bit. We saw that adding two ciphertexts modulo corresponded to XOR’ing (i.e., adding modulo ) the corresponding two plaintexts. That is, if we define as then performing the operation on the ciphertexts corresponds to adding modulo the plaintexts.

However, to get to a fully, or even partially, homomorphic scheme, we need to find a way to perform the NAND operation on the two plaintexts. The challenge is that it seems that to do that we need to find a way to evaluate *multiplications*: find a way to define some operation on ciphertexts that corresponds to multiplying the plaintexts. Alas, a priori, there doesn’t seem to be a natural way to *multiply* two vectors.

The GSW approach to handle this is to move from vectors to *matrices*. As usual, it is instructive to first consider the cryptographer’s dream world where Gaussian elimination doesn’t exist. In this case, the GSW ciphertext encrypting would be an matrix over such that where is the secret key. That is, the encryption of a bit is a matrix such that the secret key is an *eigenvector* (modulo ) of with corresponding eigenvalue . (We defer discussion of how the encrypting party generates such a ciphertext, since this is in any case only a “dream” toy example.)

# 

You should make sure you understand the *types* of all the identifiers we refer to. In particular, above is an *matrix* with entries in , is a *vector* in , and is a *scalar* (i.e., just a number) in . See naivegswfig for a visual representation of the ciphertexts in this “naive” encryption scheme. Keeping track of the dimensions of all objects will become only more important in the rest of this lecture.



In the “naive” version of the GSW encryption, to encrypt a bit we output an matrix such that where is the secret key. In this scheme we can transform encryptions of respectively to an encryption of by letting .

Given and we can recover by just checking if or . The scheme allows homomorphic evaluation of both addition (modulo ) and multiplication, since if and then we can define (where on the righthand side, addition is simply done in ) and (where again this refers to matrix multiplication in ).

Indeed, one can verify that both addition and multiplication succeed since

and

where all these equalities are in .

Addition modulo is not the same as XOR, but given these multiplication and addition operations, we can implement the NAND operation as well. Specifically, for every , . Hence we can take a ciphertext encrypting and a ciphertext encrypting and transform these two ciphertexts to the ciphertext that encrypts (where is the identity matrix). Thus in a world without Gaussian elimination it is not hard to get a fully homomorphic encryption.

# 

We have not shown how to *generate* a ciphertext without knowledge of , and hence strictly speaking we only showed in this world how to get a *private key* fully homomorphic encryption. Our “real world” scheme will be a full fledged *public key* FHE. However we note that private key homomorphic encryption is already very interesting and in fact sufficient for many of the “cloud computing” applications. Moreover, [Rothblum](http://eccc.hpi-web.de/report/2010/146/) gave a generic transformation from a *private key* homomorphic encryption to a *public key* homomorphic encryption.

## Real world partially homomorphic encryption

We now discuss how we can obtain an encryption in the real world where, as much as we’d like to ignore it, there are people who walk among us (not to mention some computer programs) that actually know how to invert matrices. As usual, the idea is to “fool Gaussian elimination with noise” but we will see that we have to be much more careful about “noise management”, otherwise even for the party holding the secret key the noise will overwhelm the signal.[[2]](#footnote-31)

The main idea is that we can expect the following problem to be hard for a random secret : distinguish between samples of random matrices and matrices where for some and “short” satisfying for all . This yields a natural candidate for an encryption scheme where we encrypt by a matrix satisfying where is a “short” vector.[[3]](#footnote-33)

We can now try to check what adding and multiplying two matrices does to the noise. If and then

and

# 

I recommend you pause here and check for yourself whether it will be the case that if encrypts and encrypts up to small noise or not.

We would have loved to say that we can define as above and . For this we would need that equals plus a “short” vector and equals plus a “short” vector. The former statement indeed holds. Looking at eqhommult we see that equals up to the “noise” vector , and if are “short” then is not too long either. That is, if and for every then . So we can at least handle a significant number of additions before the noise gets out of hand.

However, if we consider eqhommult, we see that will be equal to plus the “noise vector” . The first component of this noise vector is “short” (after all and is “short”). However, the second component could be a very large vector. Indeed, since looks like a random matrix in , no matter how small the entries of , many of the entries of are quite likely to be of magnitude at least, say, and so multiplying by takes us “beyond the edge of chaos”.

## Noise management via encoding

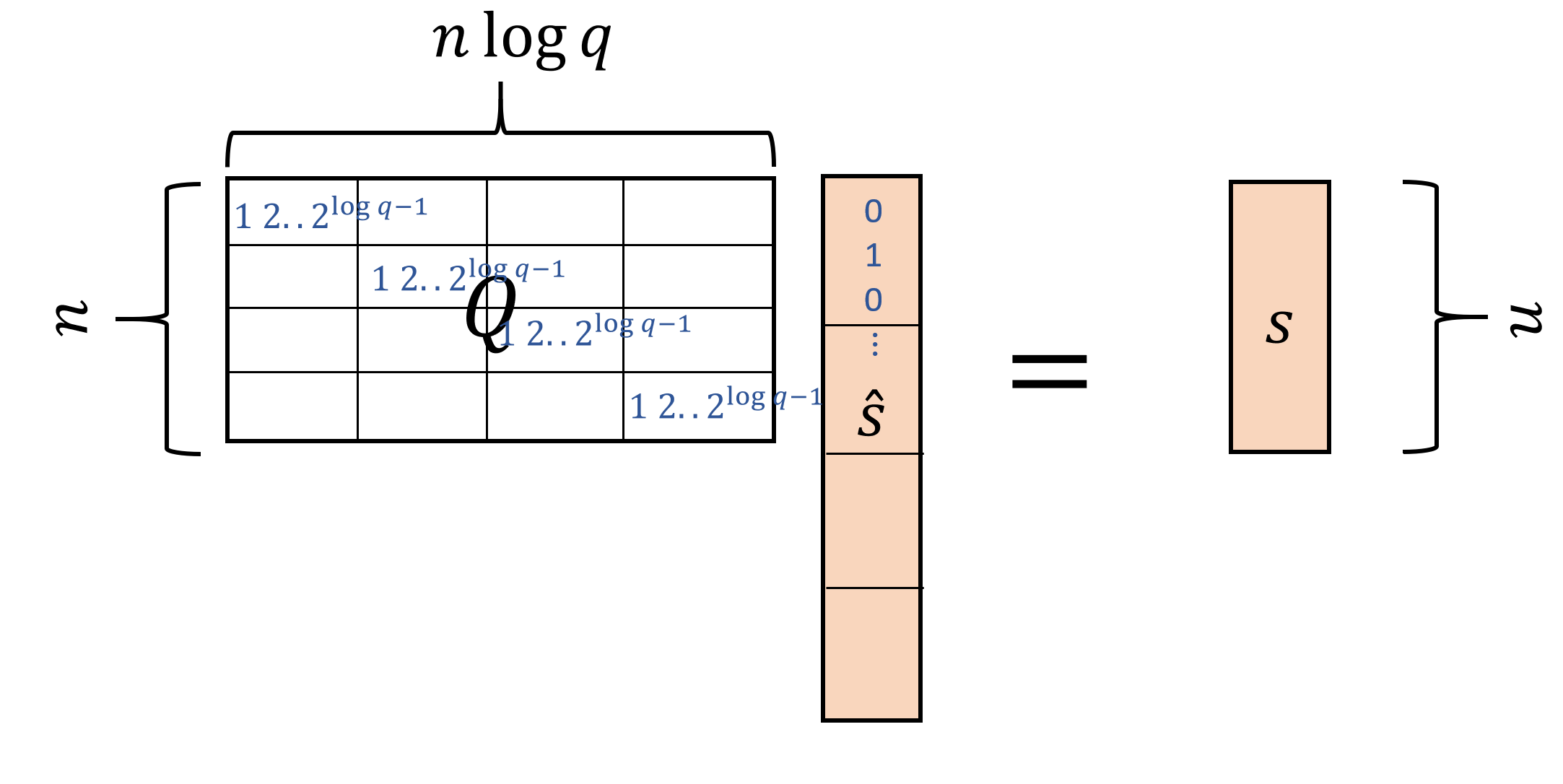
The problem we had above is that the entries of are elements in that can be very large, while we would have loved them to be small numbers such as or . At this point one could say

*“If only there was some way to encode numbers between and using only ’s and ’s”*

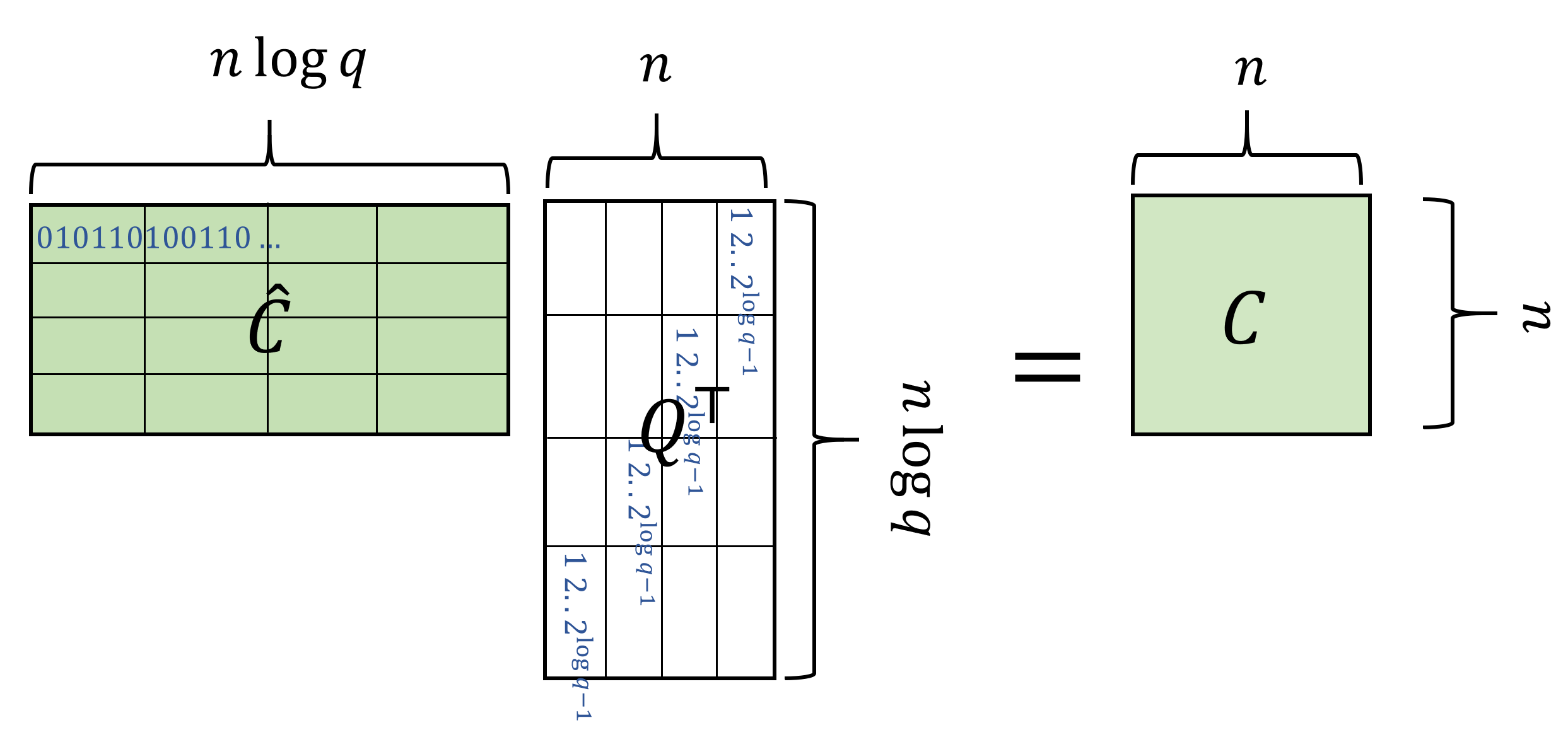
If you think about it hard enough, it turns out that there is something known as the “binary basis” that allows us to encode a number as a vector .[[4]](#footnote-36) What’s even more surprising is that this seemingly trivial trick turns out to be immensely useful. We will define the *binary encoding* of a vector or matrix over by . That is, is obtained by replacing every coordinate with coordinates such that

Specifically, if , then we denote by the -dimensional vector with entries in , such that each -sized block of encodes a coordinate of . Similarly, if is an matrix, then we denote by the matrix with entries in that corresponds to encoding every -dimensional row of by an -dimensional row where each -sized block corresponds to a single entry. (We still think of the entries of these vectors and matrices as elements of and so all calculations are still done modulo .)

While encoding in the binary basis is not a linear operation, the *decoding* operation is linear as one can see in eqbinaryencoding. We let be the “decoding” matrix that maps an encoding vector back to the original vector . Specifically, every row of is composed of blocks each of size, where the -th row has only the -th block nonzero, and equal to the values . It’s a good exercise to verify that for every vector and matrix , and . (See encodevecfig amd encodematrixfig.)



We can encode a vector as a vector that has only entries in by using the binary encoding, replacing every coordinate of with a -sized block in . The decoding operation is *linear* and so we can write for a specific (simple) matrix .



We can encode an matrix over by an matrix using the binary basis. We have the equation where is the same matrix we use to decode a vector.

In our final scheme the ciphertext encrypting will be an matrix with small coefficients such that for a “short” and for . Now given ciphertexts that encrypt respectively, we will define and .

Since we have and we get that

and

But since and for every matrix , the righthand side of fhemultfinaleqfirst equals

but since is a matrix with small coefficients for every and is short, the righthand side of fhemultfinaleqsec equals up to a short vector, and since and is short, we get that equals plus a short vector as desired.

If we keep track of the parameters in the above analysis, we can see that

then if encrypts and encrypts with noise vectors satisfying and then encrypts up to a vector of maximum magnitude at most .

## Putting it all together

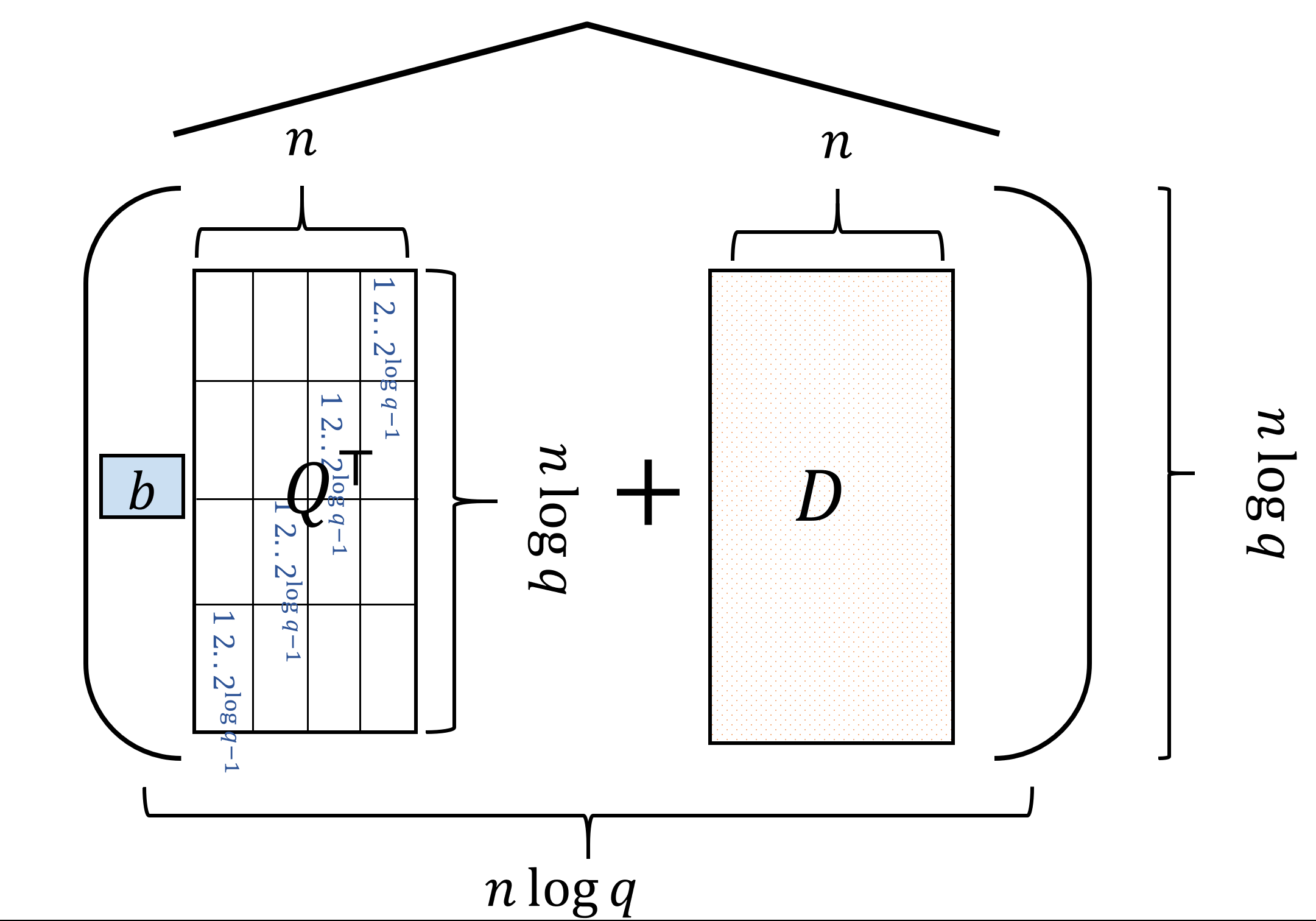
We now describe the full scheme. We are going to use a quantitatively stronger version of LWE. Namely, the -dLWE assumption for . It is not hard to show that we can relax our assumption to -LWE and Brakerski and Vaikuntanathan showed how to relax the assumption to standard (i.e. ) LWE though we will not present this here.

**FHEENC:**

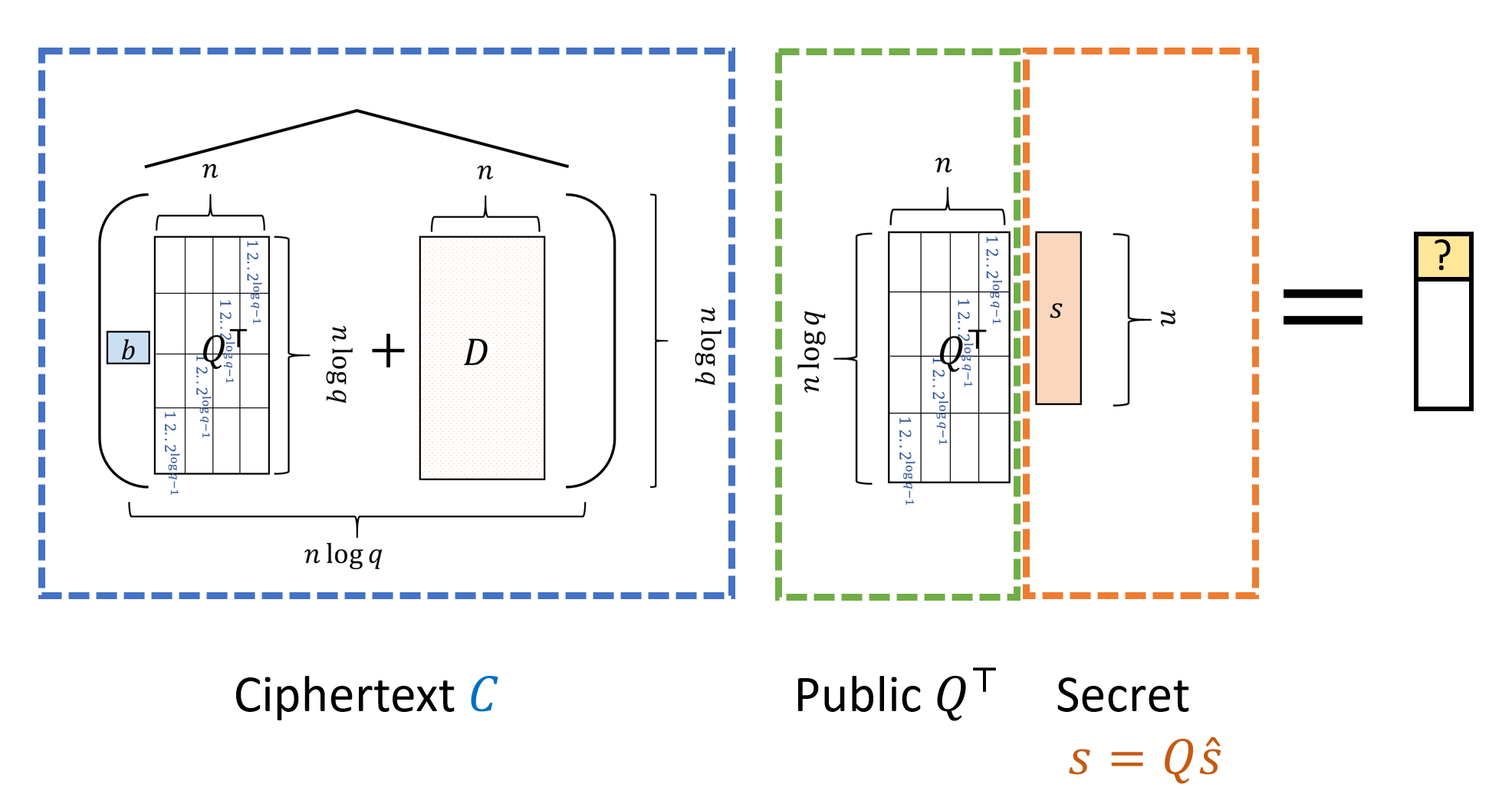
* **Key generation:** As in the scheme of last lecture the secret key is and the public key is a generator such that samples from are indistinguishable from independent random samples from but if is output by then , where the inner product (as all other computations) is done modulo and for every we define . As before, we can assume that which implies that is also since (as can be verified by direct inspection) the first row of is .
* **Encryption:** To encrypt , let output where is the matrix whose rows are generated from . (See fheencfig)
* **Decryption:** To decrypt the ciphertext , we output if and output if , see fhedecfig. (It doesn’t matter what we output on other cases.)
* **NAND evaluation:** Given ciphertexts , we define (sometimes also denoted as ) to equal , where is the identity matrix.

# 

Please take your time to read the definition of the scheme, and go over fheencfig and fhedecfig to make sure you understand it.



In our fully homomorphic encryption, the public key is a trapdoor generator . To encrypt a bit , we output where is a matrix whose rows are generated using .



We decrypt a ciphertext by looking at the first coordinate of (or equivalently, ). If then this equals to the first coordinate of , which is at most in magintude. If then we get an extra factor of which we set to be in the interval . We can think of either or as our secret key.

## Analysis of our scheme

To show that that this scheme is a valid partially homomorphic scheme we need to show the following properties:

1. **Correctness:** The decryption of an encryption of equals .
2. **CPA security:** An encryption of is computationally indistinguishable from an encryption of to someone that got the public key.
3. **Homomorphism:** If encrypts and encrypts then encrypts (with a higher amount of noise). The growth of the noise will be the reason that we will not get immediately a fully homomorphic encryption.
4. **Shallow decryption circuit:** To plug this scheme into the bootstrapping theorem we will need to show that its decryption algorithm (or more accurately, the function in the statement of the bootstrapping theorem) can be evaluated in depth (independently of ), and that moreover, the noise grows slowly enough that our scheme is homomorphic with respect to such circuits.

Once we obtain 1-4 above, we can plug FHEENC into the Bootstrapping Theorem (bootstrapthm) and thus complete the proof of existence of a fully homomorphic encryption scheme. We now address those points one by one.

### Correctness

Correctness of the scheme will follow from the following stronger condition:

# 

For every , if is the encryption of then it is an matrix satisfying

where .

# 

For starters, let us see that the dimensions make sense: the encryption of is computed by where is an matrix satisfying for every and is the .

Since is also an matrix, adding (i.e. either or the all-zeroes matrix, depending on whether or not ) to makes sense and applying the operation will transform every row to length and hence is indeed a square matrix.

Let us now see what this matrix does to the vector . Using the fact that for every matrix , we get that

but by construction for every .

fhecorrectlem implies correctness of decryption since by construction we ensured that and hence we get that if then and if then .

### CPA Security

To show CPA security we need to show that an encryption of is indistinguishable from an encryption of . However, by the security of the trapdoor generator, an encryption of computed according to our algorithm will be indistinguishable from an encryption of obtained when the matrix is a random matrix. Now in this case the encryption is obtained by applying the operation to but if is uniformly random then for every choice of , is uniformly random (since a fixed matrix plus a random matrix yields a random matrix) and hence the matrix (and so also the matrix ) contains no information about . This completes the proof of CPA security (can you see why?).

If we want to plug in this scheme in the bootstrapping theorem, then we will also assume that it is *circular secure*. It seems a reasonable assumption though unfortuantely at the moment we do not know how to derive it from LWE. (If we don’t want to make this assumption we can still obtained a *leveled* fully homomorphic encryption as discussed in the previous lecture.)

### Homomorphism

Let , and be a ciphertext such that . We define the *noise* of , denoted as to be the maximum of over all . We make the following lemma, which we’ll call the “noisy homomorphism lemma”:

# 

Let be ciphertexts encrypting respectively with . Then encrypts and satisfies

# 

This follows from the calculations we have done before. As we’ve seen,

But since is a matrix with every row of length , for every . We see that the noise vector in the product has magnitude at most . Adding the identity for the NAND operation adds at most to the noise, and so the total noise magnitude is bounded by the righthand side of eqnoisebound.

### Shallow decryption circuit

Recall that to plug in our homomorphic encryption scheme into the bootstrapping theorem, we needed to show that for every ciphertexts (generated by the encryption algorithm) the function defined as

can be homomorphically evaluated where is the secret key and denotes the decryption algorithm applied to .

In our case we can think of the secret key as the binary string which describes our vector as a bit string of length . Given a ciphertext , the decryption algorithm takes the dot product modulo of with the first row of (or, equivalently, the dot product of with ) and outputs (respectively ) if the resulting number is small (respectively large).

By repeatedly applying the noisy homomorphism lemma (noisehomolem), we can show that can homorphically evaluate every circuit of NAND gates whose *depth* satisfies . If then (assuming is sufficiently large) then as long as this will be satisfied.

In particular to show that can be homomorphically evaluated it will suffice to show that for every fixed vector there is a depth circuit that on input a string will output if and output if . (We don’t care what does otherwise. The above suffices since given a ciphertext we can use with the vector being the top row of , and hence would correspond to the first entry of . Note that if has depth then the function above has depth at most .)

# 

Please make sure you understand the above argument.

If is a vector then to compute its inner product with a vector we simply need to sum up the numbers where . Summing up numbers can be done via the obvious recursion in depth that is times the depth for a single addition of two numbers. However, the naive way to add two numbers in (each represented by bits) will have depth which is too much for us.

# 

Please stop here and see if you understand why the natural circuit to compute the addition of two numbers modulo (represented as -length binary strings) will require depth . As a hint, one needs to keep track of the “carry”.

Fortunately, because we only care about accuracy up to , if we add numbers, we can drop all but the first most significant digits of our numbers, since including them can change the sum of the numbers by at most . Hence we can easily do this work in depth, which is since .

Let us now show this more formally:

# 

For every there exists some function such that:  
1. For every such that ,   
2. For every such that ,   
3. There is a circuit computing of depth at most .

# 

For every number , write to be the number that is obtained by writing in the binary basis and setting all digits except the most significant ones to zero.  
Note that . We define to equal if and to equal otherwise (where as usual the absolute value of modulo is the minimum of and .) Note that all numbers involved have zeroes in all but the most significant digits and so these less significant digits can be ignored. Hence we can add any pair of such numbers modulo in depth using the standard elementary school algorithm to add two -digit numbers in steps. Now we can add the numbers by adding pairs, and then adding up the results, and this way in a binary tree of depth to get a total depth of . So, all that is left to prove is that this function satisfies the conditions (1) and (2).

Note that so now we want to show that the effect of taking modulo is not much different from taking modulo . Indeed, note that this sum (before a modular reduction) is an integer between and . If is such an integer and we divide by to write for , then since , , and so we can write so the difference between and will be (in our standard modular metric) at most . Overall we get that if is in the interval then will be in the interval which is contained in .

This completes the proof that our scheme can fit into the bootstrapping theorem (i.e., of LWEFHEthm), hence completing the description of the fully homomorphic encryption scheme.

# 

Now would be a good point to go back and see you understand how all the pieces fit together to obtain the complete construction of the fully homomorphic encryption scheme.

## Advanced topics:

* Fully homomorphic encryption for approximate computation over the real numbers: [CKKS](https://eprint.iacr.org/2016/421.pdf)
* Bandwidth efficient fully homomorphic encryption [GH](https://eprint.iacr.org/2019/733.pdf)
* Using fully homomorphic encryption to achieve private information retrieval.

1. This theorem as stated was proven by Brakerski and Vaikuntanathan (ITCS 2014) building a line of work initiated by Gentry’s original STOC 2009 work. We will actually prove a weaker version of this theorem, due to Brakerski and Vaikuntanathan (FOCS 2011), which assumes a quantitative strengthening of LWE. However, we will not follow the proof of Brakerski and Vaikuntanathan but rather a scheme of Gentry, Sahai and Waters (CRYPTO 2013). Also note that, as noted in the previous lecture, all of these results require the extra assumption of *circular security* on top of LWE to achieve a non-leveled fully homomorphic encryption scheme. [↑](#footnote-ref-21)
2. For this reason, Craig Gentry called his highly recommended survey on fully homomorphic encryption and other advanced constructions [computing on the edge of chaos](https://eprint.iacr.org/2014/610). [↑](#footnote-ref-31)
3. We deliberately leave some flexibility in the definition of “short”. While initially “short” might mean that for every , decryption will succeed as long as long as is, say, at most . [↑](#footnote-ref-33)
4. If we were being pedantic the length of the vector (and other constant below) should be the integer but I omit the ceiling symbols for simplicity of notation. [↑](#footnote-ref-36)