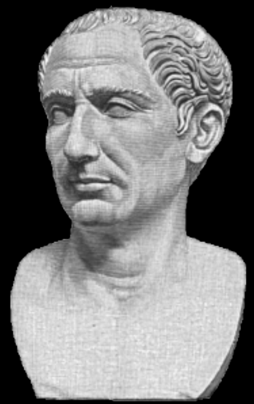
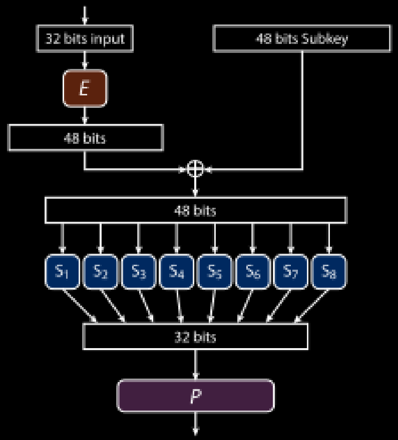
Computing On Encrypted Data with C++

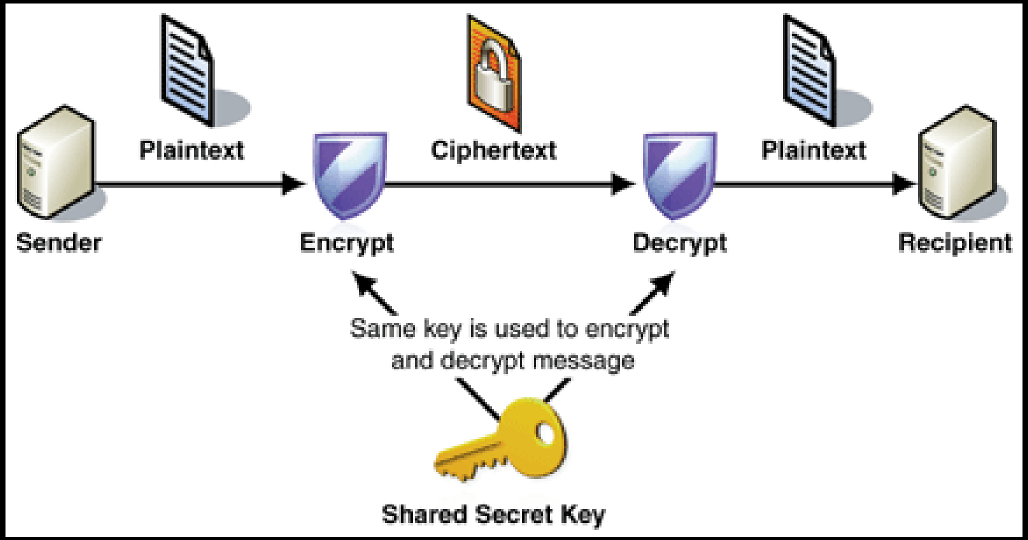
Max Leibovich

Part 0:
Introduction to Cryptography

A Brief History of Cryptography: Symmetric Encryption

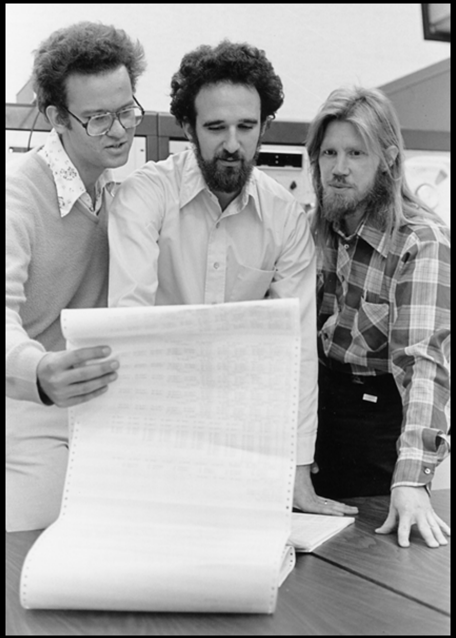


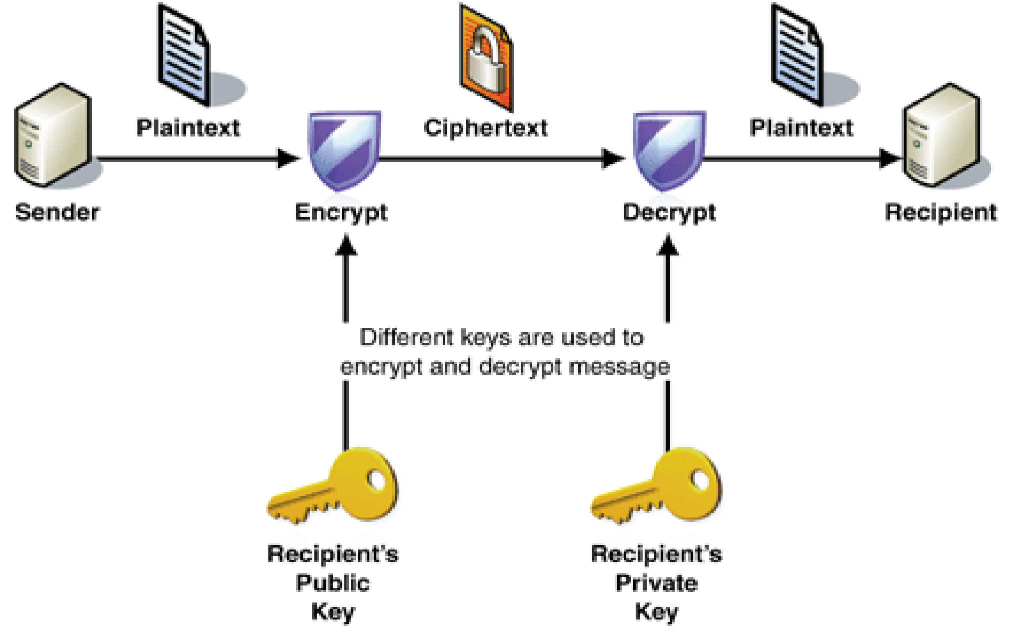


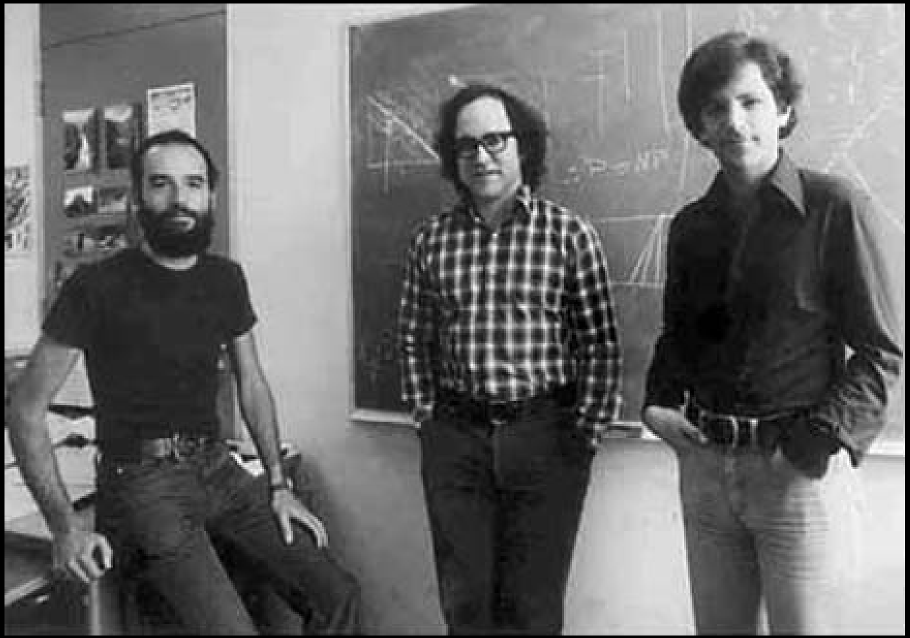


• 100 – 1 A.D.	Caesar cipher (Shift cipher)
• 1553	Vigenère cipher (Poly-alphabetic Substitution Cipher)
• 1920s	Enigma machine
• 1976	DES (Data Encryption Standard) Symmetric-Key Algorithm

A Brief History of Cryptography: Asymmetric Encryption







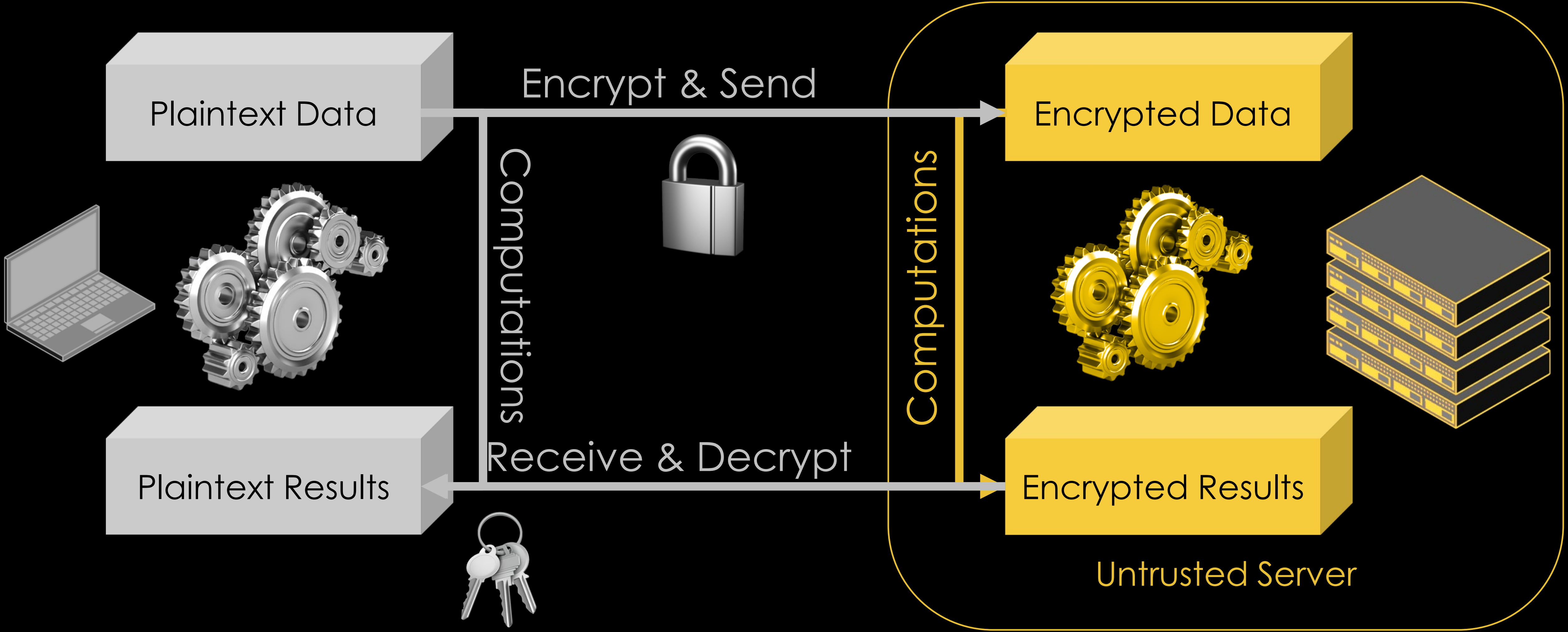
- 1976 Diffie, Hellman and Merkle (DH key Exchange, Merkle's Puzzles)
- 1977 Rivest, Shamir and Adleman (RSA Public-Key Cryptosystem)

What else can we do with Encrypted Data?

1978 Rivest, Adleman and Dertouzos: “On data banks and privacy homomorphisms”

What else can we do with Encrypted Data?

1978 Rivest, Adleman and Dertouzos: “On data banks and privacy homomorphisms”



Plaintext Data

Encrypt & Send

Encrypted Data

Computations

Computations

Plaintext Results

Receive & Decrypt

Encrypted Results

Untrusted Server

Part 1:
Partially Homomorphic Encryption

Where this idea came from?

Lets look at "Textbook" RSA

$\mathrm{Keygen}(k)$: Choose random k-bit primes p, q

Compute $n \coloneqq p \cdot q$ and $\phi(N) = (p-1) \cdot (q-1)$

Choose integer $1 < e < \phi(n)$

which is co-prime to $\phi(n)$, i.e. $\gcd\big(e, \phi(n)\big) = 1$

Compute $d \coloneqq e^{-1} \bmod \phi(n)$, i.e. $d \cdot e \equiv 1 \bmod \phi(n)$

Public Key: $pk = \langle n, e \rangle$ Secret Key: $sk = \langle n, d \rangle$

$\mathrm{Enc}_{pk=\langle n,e \rangle}(m)$: $c \coloneqq m^e \bmod n$

$\mathrm{Dec}_{sk=\langle n,d \rangle}(c)$: $m \coloneqq c^d \bmod n$

Where this idea came from?

Lets look at “Textbook” RSA

$\mathrm{Enc}_{pk=\langle n,e\rangle}(m)$: $c := m^e \bmod n$

$\mathrm{Dec}_{sk=\langle n,d\rangle}(c)$: $m := c^d \bmod n$

RSA has the following property:

$$c_1 = \mathrm{Enc}_{pk}(m_1) = m_1^e \bmod n \qquad c_2 = \mathrm{Enc}_{pk}(m_2) = m_2^e \bmod n$$

$$c_1 \cdot c_2 = [m_1^e \bmod n] \cdot [m_2^e \bmod n] = [(m_1^e \cdot m_2^e) \bmod n] = [(m_1 \cdot m_2)^e \bmod n]$$

$$= \mathrm{Enc}_{pk}(m_1 \cdot m_2) \qquad \Longrightarrow \qquad \mathrm{Dec}_{sk}(c_1 \cdot c_2) = m_1 \cdot m_2$$

Partially Homomorphic Encryptions

- Given groups $(G,*)$ and $(H,\circ)$ a function $f: G \rightarrow H$ is a ***Homomorphism*** if it preserves the operation, i.e. for all $x, y \in G$: $\boxed{f(x * y) = f(x) \circ f(y)}$
- Examples to Homomorphism:

$|x \cdot y| = |x| \cdot |y|$ $(x \cdot y)^c = x^c \cdot y^c$ $e^{x+y} = e^x \cdot e^y$ $\ln(x \cdot y) = \ln(x) + \ln(y)$

- Multiplicatively Homomorphic Encryptions $\mathrm{Enc}(x \cdot y) = \mathrm{Enc}(x) \cdot \mathrm{Enc}(y)$
  - RSA (1977)
  - ElGamal (1985)

Partially Homomorphic Encryptions

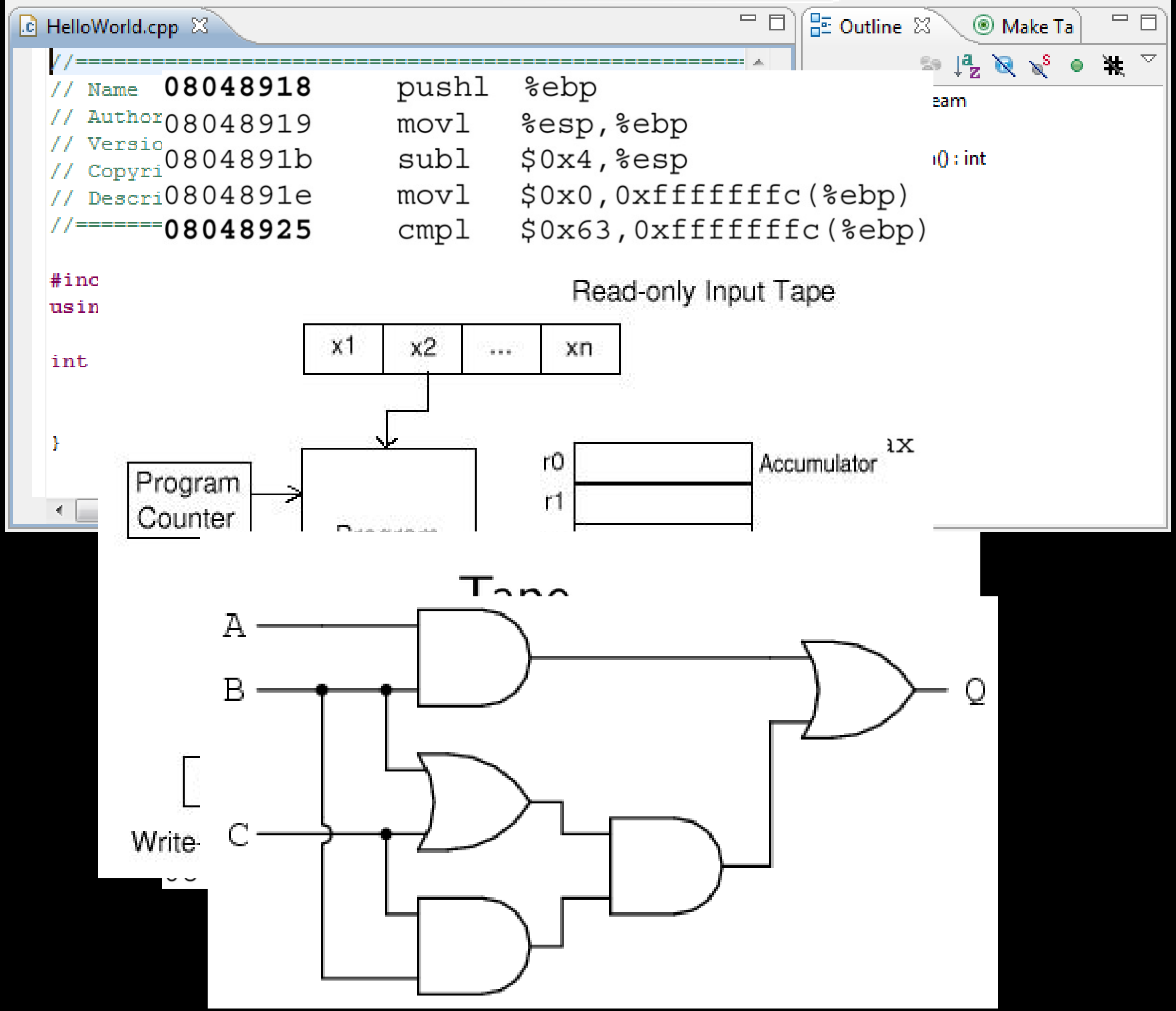
- Additively Homomorphic Encryptions $\mathrm{Enc}(x + y) = \mathrm{Enc}(x) \cdot \mathrm{Enc}(y)$
  - Benaloh(1994)
  - Paillier (1999)
- Homomorphic Encryption with respect to XOR $\mathrm{Enc}(x \oplus y) = \mathrm{Enc}(x) \cdot \mathrm{Enc}(y)$
  - Goldwasser–Micali (1982)
- What can we do when we are restricted to a single operation?

Not Much!

Part 2:
Somewhat Homomorphic Encryption

Models Of Computation

- High-Level Programming Language (e.g. C++)
- Low-Level Programming Language (e.g. Assembly)
- Random-Access Machine
- Turing Machine
- Boolean / Arithmetic Circuits

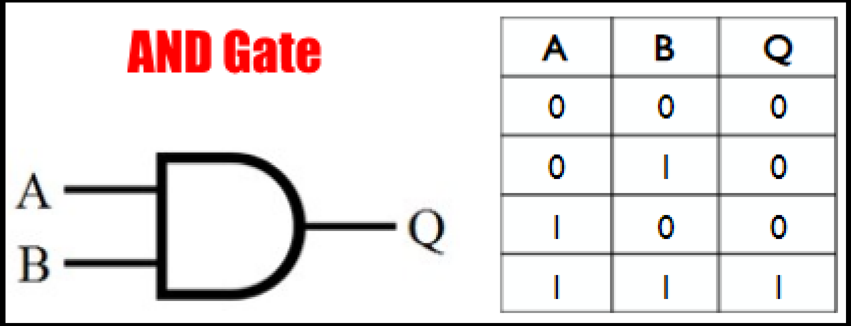


```
08048918      pushl  %ebp
08048919      movl   %esp,%ebp
0804891b      subl   $0x4,%esp
0804891e      movl   $0x0,0xfffffffc(%ebp)
08048925      cmpl   $0x63,0xfffffffc(%ebp)
```

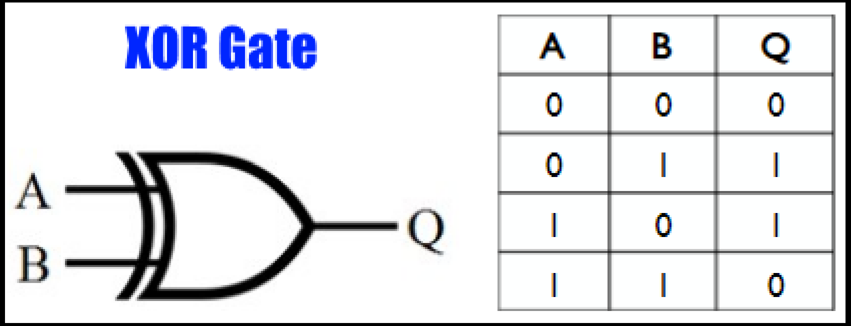
<u>All are **Turing-complete**, but there are **Time** and **Space** complexity tradeoffs</u>

Why Boolean Circuits?

- Because { **XOR**, **AND** } is Turing-complete, **ANY** function can be computed with a Boolean circuit consisting of only { **XOR**, **AND** } gates.



A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1



A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

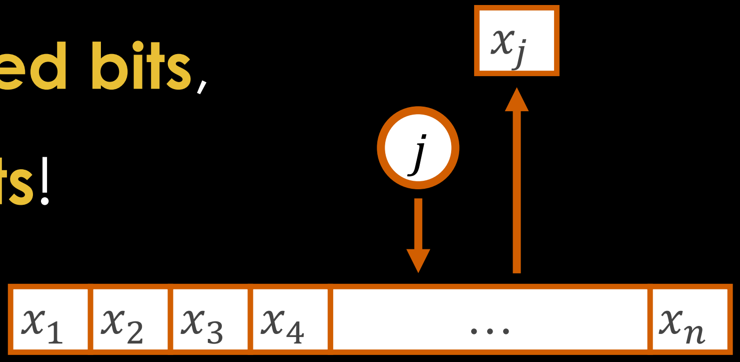
- Over Boolean values we have:

$$\textbf{AND}(a,b) = (a \cdot b) \bmod 2 \qquad \textbf{XOR}(a,b) = (a + b) \bmod 2$$

- Not necessarily the most efficient way to evaluate a function!

Why Boolean Circuits?

- If you can compute **products** and **sums** on **encrypted bits**, you can compute **ANY** function on **encrypted inputs**!
- Example: Private Information Retrieval (PIR)



Server Input:	array of n bits $x_1, \dots, x_n$	Client Input:	index $1 \leq j \leq n$
Server Output:	nothing	Client Output:	bit x_j

Compare Indices $a, b \in \{0,1\}^{\log_2 n}$: $\mathrm{eq}(a,b) = \prod_{1 \le k \le \log_2 n} [a_k + b_k + 1]$

PIR function:

$$f(x_1, \dots, x_n, j) = \sum_{1 \le i \le n} [\, \mathrm{eq}(i,j) \cdot x_i \,]$$

What objects can we add and multiply?

- Polynomials? $(x^4 + 6x^3 + 2x) + (4x^2 - 3x) = (x^4 + 6x^3 + 4x^2 - x)$

  $(5x^2 + 9x + 8) \cdot (7x + 1) = (35x^3 + 68x^2 + 65x + 8)$

- Matrices? $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$ $\begin{pmatrix} 2 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} -1 & 3 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} -2 & 6 \\ -1 & 4 \end{pmatrix}$

- Why not Integers?!? $2 + 2 = 4$ $7 \cdot 6 = 42$

Example Symmetric Encryption Scheme Over Integers

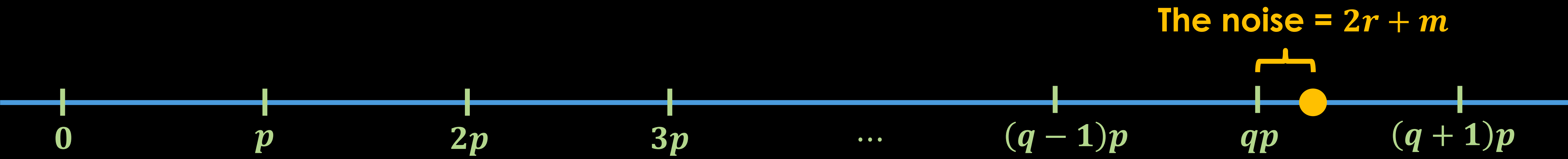
Keygen(k): Pick a random **large** k^2-bit **odd** integer p as the Secret Key

Enc$_p$($m \in \{0,1\}$): Pick a random k^5-bit integer q and compute $q \cdot p$ a **large** multiple of p

Pick a random small k-bit integer $2r + m$,

that is **even** when $m = 0$, and **odd** when $m = 1$

Ciphertext will be $c \coloneqq q \cdot p + 2r + m$



Example Symmetric Encryption Scheme Over Integers

$\mathrm{Dec}_p(c \in \mathbb{Z})$:

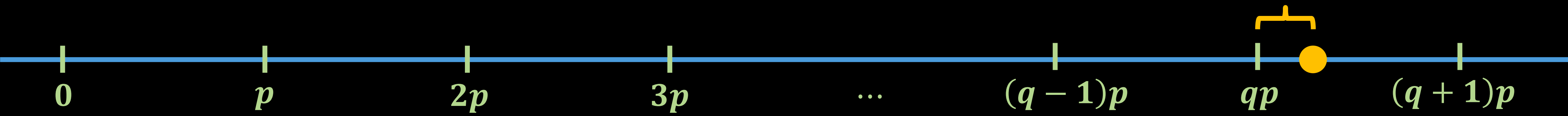
Compute $c' := c \bmod^* p$ to recover the "noise".

Where $c \bmod^* p$ is the "**Centered Modulo**" operation that returns the integer $c' \in (-p/2, p/2)$ such that p divides $c - c'$.

In other words:

1. $c' := c \bmod p$;
2. $if\,(c' > p/2)\ then\ c' := c' - p$;

Return $m := c' \bmod 2$



The noise = $2r + m$

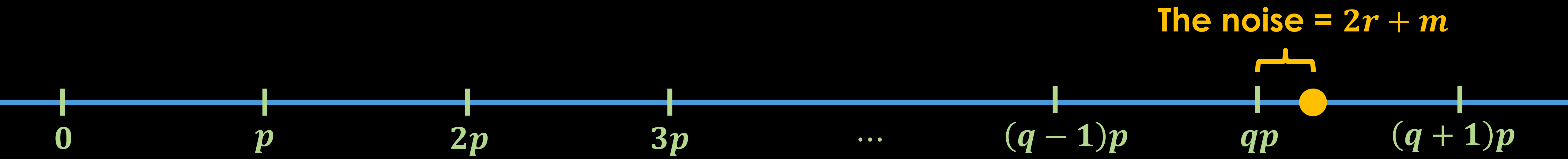
Security

- How secure is this?
  - If noise = 0 and we get two encryptions of 0: $\mathrm{Enc}_p(0) = q_1 p$, $\mathrm{Enc}_p(0) = q_2 p$

    Recovering the Secret Key p is easy: simply calculate $p = \mathrm{GCD}(q_1 p, q_2 p)$
  - But if there is **noise** the GCD attack doesn't work.

    And we believe that neither does any other attack.

    This is called the "*Approximate GCD Assumption*".



Homomorphic Operations

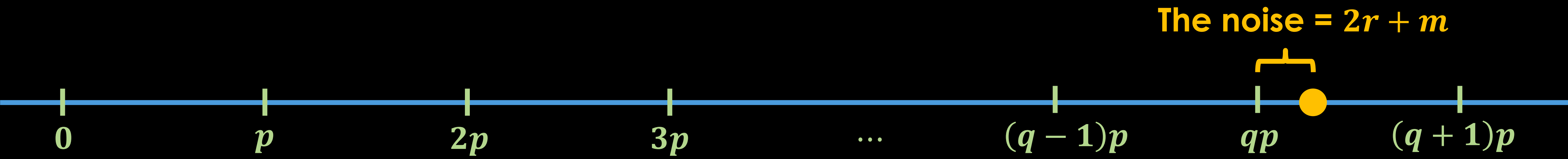
- How do we **XOR** two encrypted bits $c_1 = \mathrm{Enc}_p(m_1)$ and $c_2 = \mathrm{Enc}_p(m_2)$?

$c_1 = q_1 p + 2r_1 + m_1 \qquad c_2 = q_2 p + 2r_2 + m_2$

$c_1 + c_2 = p \cdot (q_1 + q_2) + 2 \cdot (r_1 + r_2) + (m_1 + m_2)$

On decryption, after $\mathrm{mod}^* \, p$: $2 \cdot (r_1 + r_2) + (m_1 + m_2)$

after $\mathrm{mod}\ 2$: $m_1 \oplus m_2$



The noise = $2r + m$

Homomorphic Operations

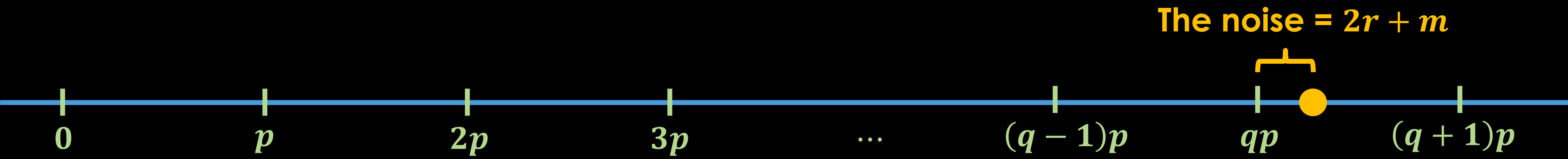
- How do we **AND** two encrypted bits $c_1 = \mathrm{Enc}_p(m_1)$ and $c_2 = \mathrm{Enc}_p(m_2)$?

$c_1 = q_1 p + 2r_1 + m_1 \qquad c_2 = q_2 p + 2r_2 + m_2$

$c_1 \cdot c_2 = p \cdot (q_1 c_2 + q_2 c_1 - q_1 q_2) + 2 \cdot (r_1 r_2 + r_1 m_2 + r_2 m_1) + (m_1 \cdot m_2)$

On decryption, after $\mathrm{mod}^*\, p$: $2 \cdot (r_1 r_2 + r_1 m_2 + r_2 m_1) + (m_1 \cdot m_2)$

after $\mathrm{mod}\ 2$: $m_1 \cdot m_2$

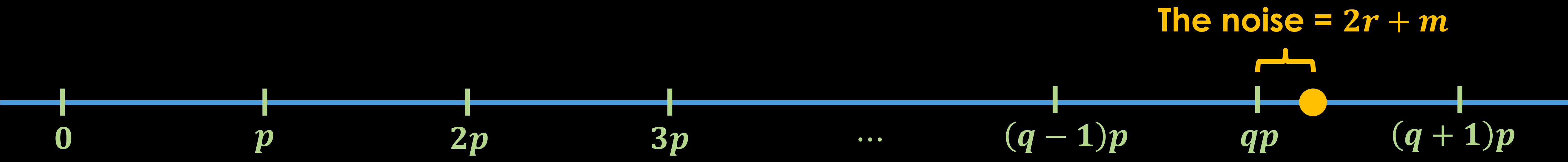


Noise

- What about the noise? The noise grows after each operation!

- $c_1 + c_2 = p \cdot (q_1 + q_2) + \underbrace{2 \cdot (r_1 + r_2) + (m_1 + m_2)}_{\text{noise} \approx 2 \cdot (\text{Initial Noise})}$

- $c_1 \cdot c_2 = p \cdot (q_1 c_2 + q_2 c_1 - q_1 q_2) + \underbrace{2 \cdot (r_1 r_2 + r_1 m_2 + r_2 m_1) + (m_1 \cdot m_2)}_{\text{noise} \approx (\text{Initial Noise})^2}$



The noise = $2r + m$

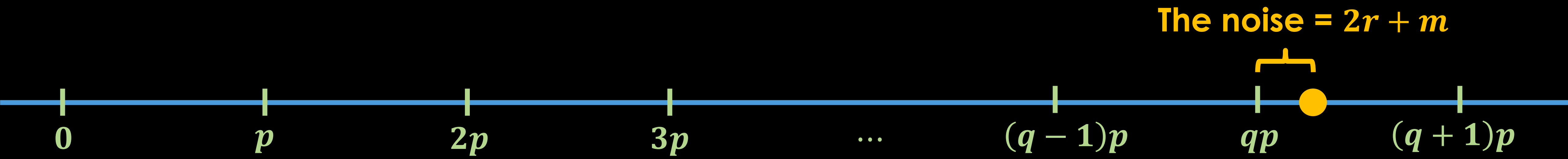
Noise

- Why is this bad?

Initial noise = $2r + m$

noise after computations = r'

- If $\boxed{(2r + m) \bmod 2} \neq \boxed{r' \bmod 2}$ decryption outputs incorrect bit!
- This can happen when $|\text{noise}| > p/2$



The noise = $2r + m$

So, what did we accomplish?

- We can do lots of additions…
- And some multiplications, until we are no longer able to correctly decrypt…
- This is called: **Somewhat** Homomorphic Encryption.
- It Is already enough for some useful applications:
  - PIR over small databases
  - Algorithms implemented as polynomials with logarithmic degree
  - …
- But, we can do much better!

Many C++ Implementations

- This scheme is called DGHV
- There are any many C++ implementations of it in GitHub, e.g.:

  https://github.com/rinon/Simple-Homomorphic-Encryption

  https://github.com/bogdan-kulynych/libshe

  https://github.com/deevashwer/Fully-Homomorphic-DGHV-and-Variants

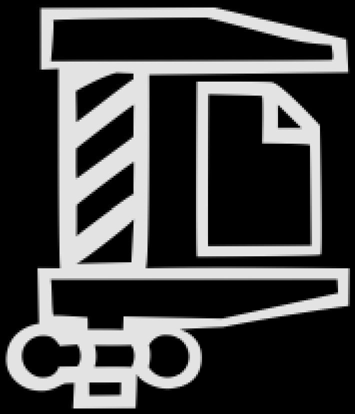
  https://github.com/raduMMR/OMP-DGHV

  https://github.com/andronat/libshe

Part 3:
Leveled Homomorphic Encryption

Noise and Compactness

- Recall that in the scheme over integers we just saw:
  - Noise grows **exponentially** with the multiplicative depth.
  - Scheme is not **compact** as ciphertext size grows with the size of the circuit.
- Let’s see how to tackle these problems…



Noise
Management Techniques

BGV: A scheme over **polynomial rings** $R_q = \mathbb{Z}_q[X]/(X^d + 1)$ based on the hardness of "Ring Learning With Errors" (RLWE) problem.

- How do objects in polynomial rings $R_q = \mathbb{Z}_q[X]/(X^d + 1)$ look like?

  Think of them as a vectors of **size** $\boldsymbol{d}$ where each element is an **integer in** $\boldsymbol{\mathbb{Z}_q}$.

$\mathbf{1}$	$\mathbf{2}$	$\mathbf{3}$		$\boldsymbol{d-1}$	$\boldsymbol{d}$
$\mathbb{Z}_q$	$\mathbb{Z}_q$	$\mathbb{Z}_q$	…	$\mathbb{Z}_q$	$\mathbb{Z}_q$

- In BGV the noise increases **linearly** with the multiplicative depth!

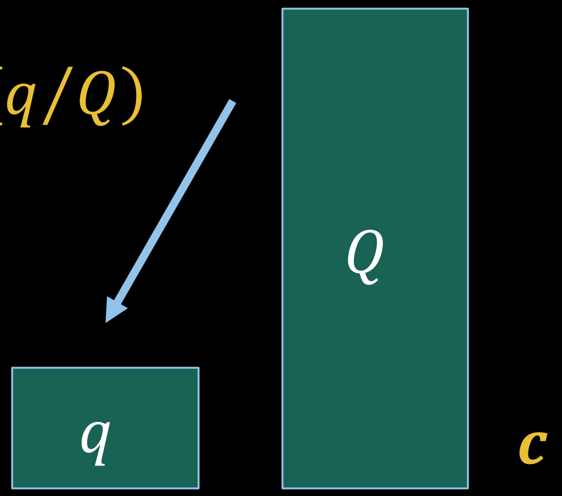
  Lets see how…

Modulus-Switching

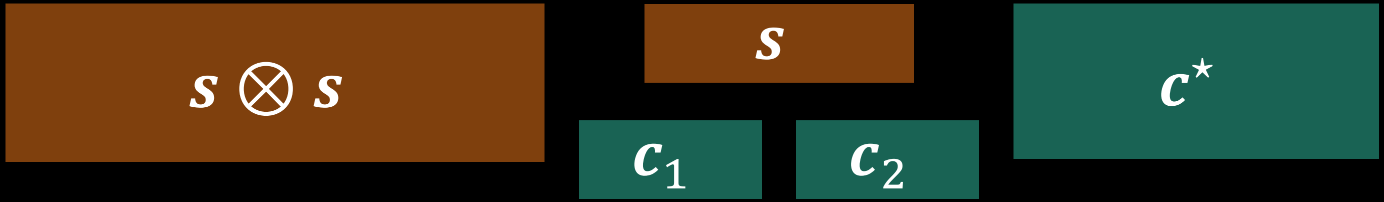
- Ciphertext $\boldsymbol{c} \in R_Q$ for **large** modulus Q is an encryption of m.
- **Scale** $\boldsymbol{c}$ by (q/Q) and **round** appropriately with a <u>smaller</u> modulus $q \ll Q$.

  The resulted ciphertext $\boldsymbol{c}' \in R_q$ is also a valid encryption of m.
- This allows to reduce the ciphertext **noise** by a factor $\approx (q/Q)$ without knowing the secret-key!

$$\boldsymbol{c}' = [(q/Q\,)\cdot \boldsymbol{c}]_q$$



Re-Linearization (Key-Switching)



- Given BGV ciphertexts $\boldsymbol{c}_1, \boldsymbol{c}_2$ decryptable to messages m_1, m_2 under key $\boldsymbol{s}$.
- After multiplication ciphertext $\boldsymbol{c}^\star = \boldsymbol{c}_1 \cdot \boldsymbol{c}_2$ has roughly d^2 elements which is decryptable with a **longer** key $\boldsymbol{s} \otimes \boldsymbol{s}$.

Re-Linearization (Key-Switching)

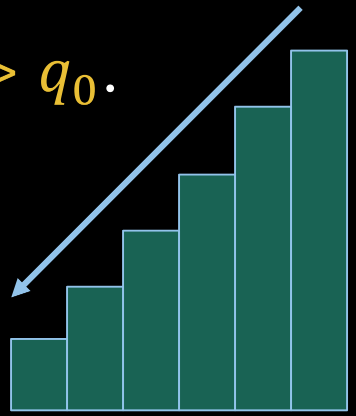
- To reduce the size of $\boldsymbol{c}^{\star}$ back to d we use **Re-Linearization** technique:

1. Encrypt $\boldsymbol{s}[i]$ and $\boldsymbol{s}[i] \cdot \boldsymbol{s}[j]$ for $1 \leq i, j \leq d$ under a new **secret key** $\boldsymbol{t}$.
2. Place encryptions of $\mathrm{Enc}_{\boldsymbol{t}}(\boldsymbol{s}[i])$, $\mathrm{Enc}_{\boldsymbol{t}}(\boldsymbol{s}[i] \cdot \boldsymbol{s}[j])$ for $1 \leq i, j \leq d$ in **public key**.
3. Convert **long** ciphertext $\boldsymbol{c}^{\star}$ to a **short** one $\boldsymbol{c}'$ (decryptable by new key $\boldsymbol{t}$).

- Tradeoff between **long ciphertexts** (many) and **long secret key** (single).

Leveled Homomorphic Encryption

- Apply the **modulus-switching** technique after every multiplication, using a ladder of gradually decreasing moduli $q_L > q_{L-1} > \cdots > q_1 > q_0$.
- Freshly encrypted ciphertexts are over R_{q_L} and on ciphertexts over R_{q_0} we cannot compute anymore.

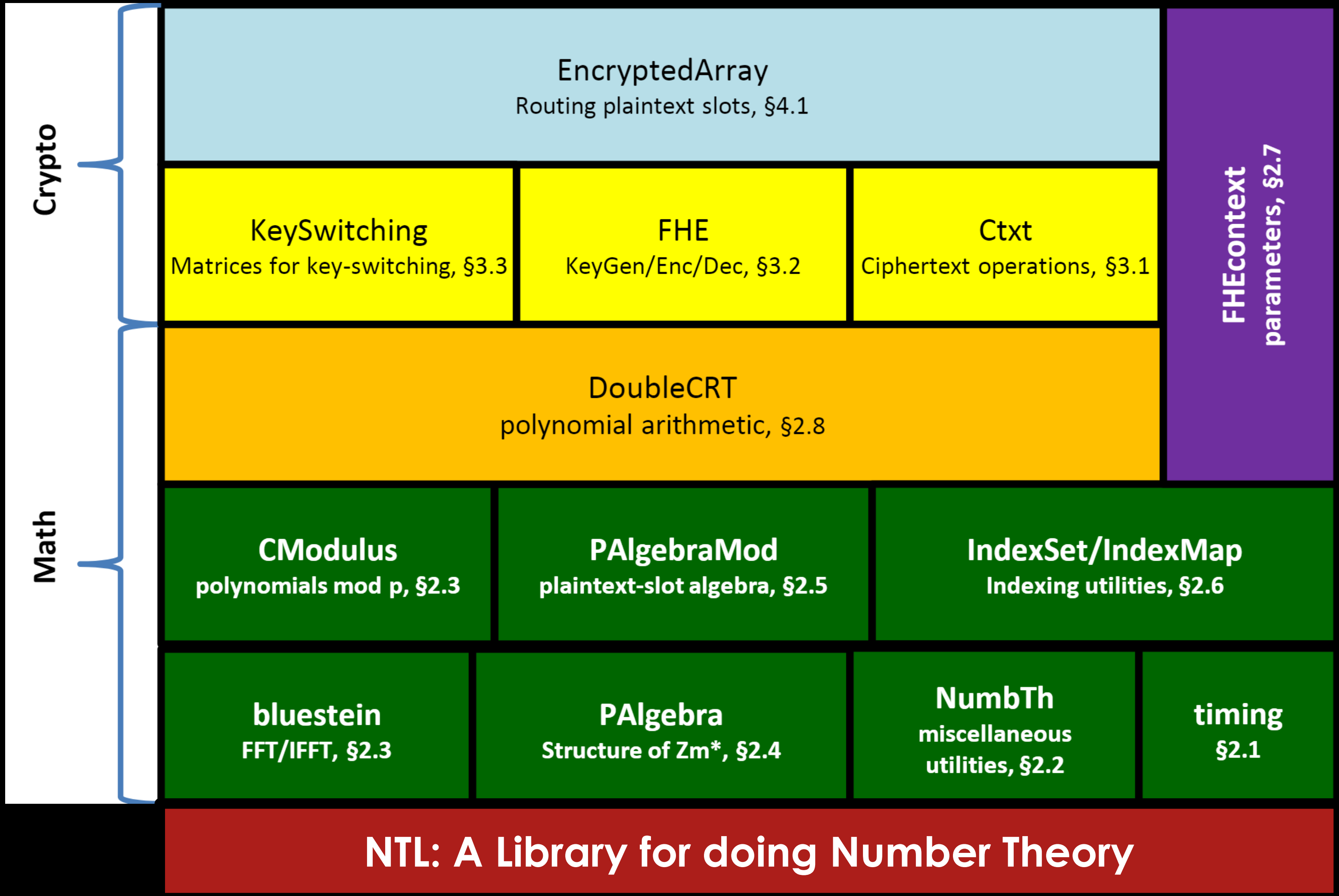


- After each multiplication preform a **Re-Linearization** on the resulted ciphertext.
- Performing these two operations together is sometimes called: "**Ciphertext Refresh**".

HElib

https://github.com/shaih/HElib

- C++ library that implements BGV HE scheme, along with many optimizations.



Crypto

Math

EncryptedArray
Routing plaintext slots, §4.1

KeySwitching
Matrices for key-switching, §3.3

FHE
KeyGen/Enc/Dec, §3.2

Ctxt
Ciphertext operations, §3.1

FHEcontext
parameters, §2.7

DoubleCRT
polynomial arithmetic, §2.8

CModulus
polynomials mod p, §2.3

PAlgebraMod
plaintext-slot algebra, §2.5

IndexSet/IndexMap
Indexing utilities, §2.6

bluestein
FFT/IFFT, §2.3

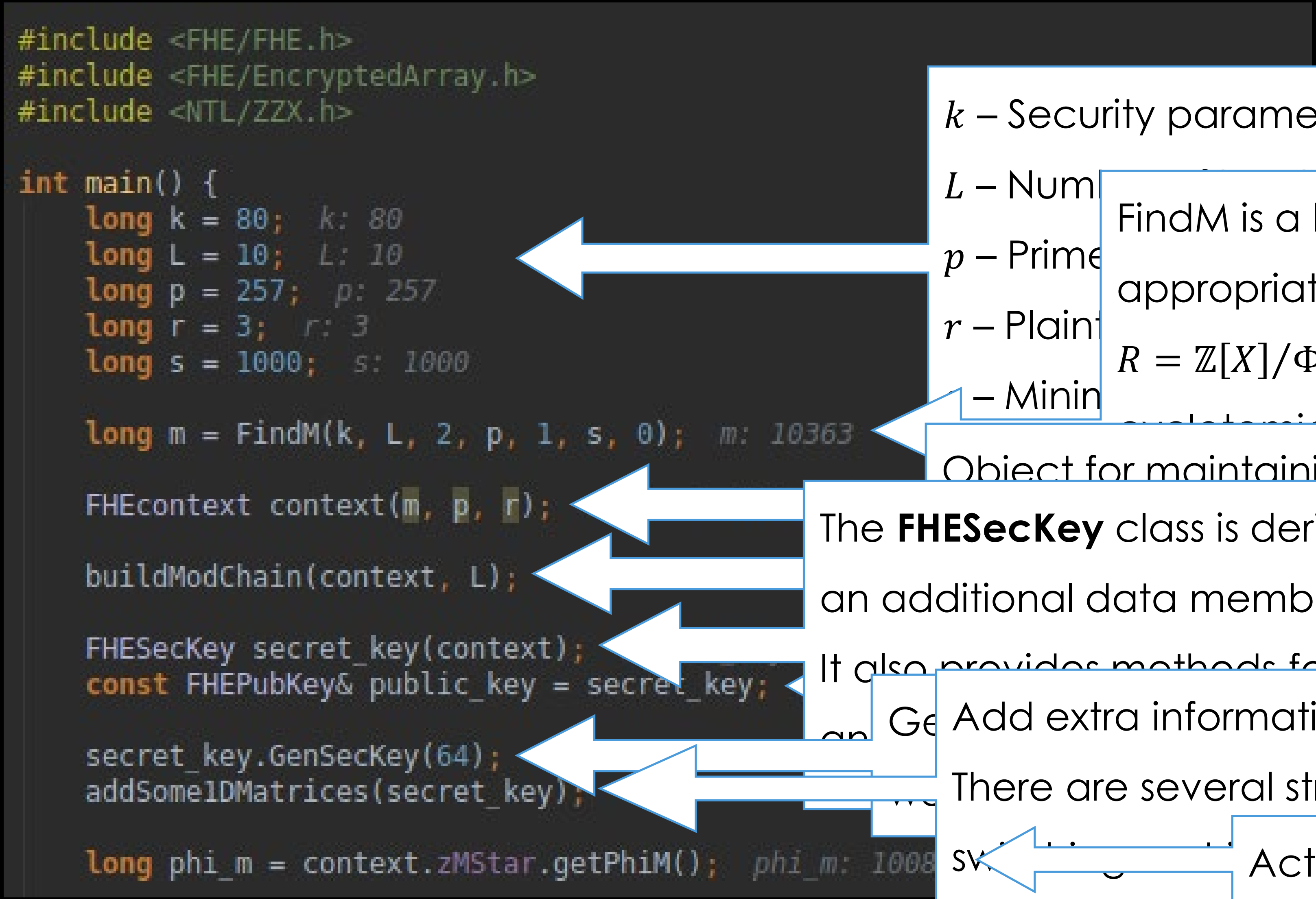
PAlgebra
Structure of Zm*, §2.4

NumbTh
miscellaneous utilities, §2.2

timing
§2.1

NTL: A Library for doing Number Theory

Initialization



```
#include <FHE/FHE.h>
#include <FHE/EncryptedArray.h>
#include <NTL/ZZX.h>

int main() {
    long k = 80;  k: 80
    long L = 10;  L: 10
    long p = 257;  p: 257
    long r = 3;  r: 3
    long s = 1000;  s: 1000

    long m = FindM(k, L, 2, p, 1, s, 0);  m: 10363

    FHEcontext context(m, p, r);

    buildModChain(context, L);

    FHESecKey secret_key(context);
    const FHEPubKey& public_key = secret_key;

    secret_key.GenSecKey(64);
    addSome1DMatrices(secret_key);

    long phi_m = context.zMStar.getPhiM();  phi_m: 1008
```

k – Security parameter (2^k)

L – Num

p – Prime

r – Plaint

– Minin

FindM is a helper function to find the appropriate size of the polynomial ring $R = \mathbb{Z}[X]/\Phi_m(X)$ where $\Phi_m(X)$ is the m-th

Object for maintaining parameters

The **FHESecKey** class is derived from **FHEPubKey**. It contains an additional data member with the <u>secret key</u>.

It also provides methods for key generation, decryption

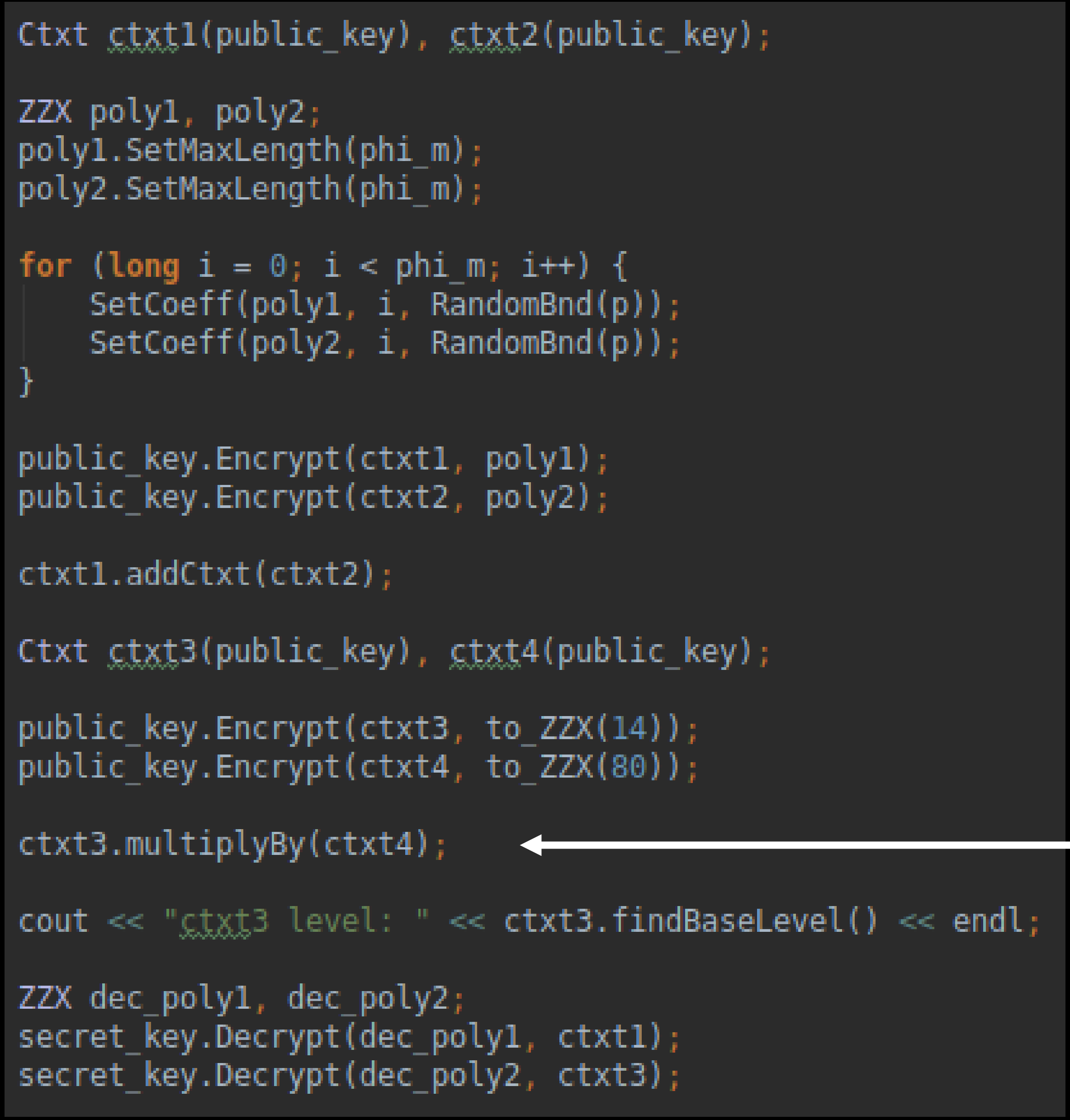
Ge

Add extra information for <u>re-linearization</u>.

There are several strategies for deciding what key-

Actual size of the ciphertext vector

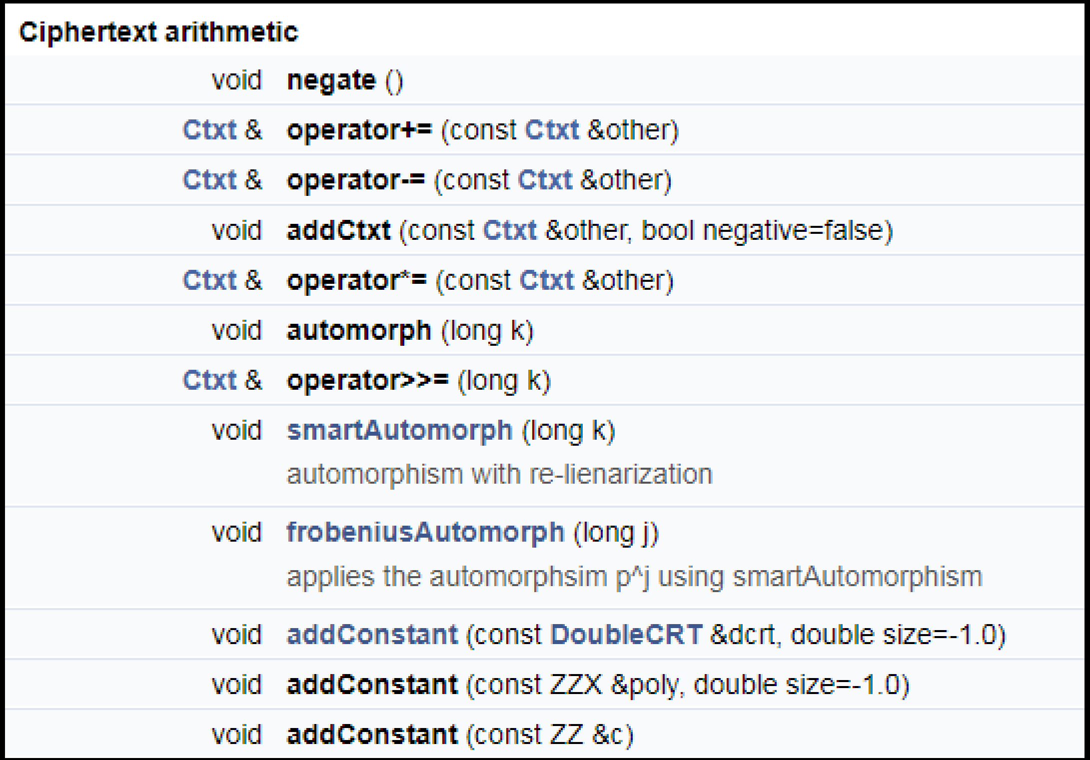
Basic Arithmetic



```
Ctxt ctxt1(public_key), ctxt2(public_key);

ZZX poly1, poly2;
poly1.SetMaxLength(phi_m);
poly2.SetMaxLength(phi_m);

for (long i = 0; i < phi_m; i++) {
    SetCoeff(poly1, i, RandomBnd(p));
    SetCoeff(poly2, i, RandomBnd(p));
}

public_key.Encrypt(ctxt1, poly1);
public_key.Encrypt(ctxt2, poly2);

ctxt1.addCtxt(ctxt2);

Ctxt ctxt3(public_key), ctxt4(public_key);

public_key.Encrypt(ctxt3, to_ZZX(14));
public_key.Encrypt(ctxt4, to_ZZX(80));

ctxt3.multiplyBy(ctxt4);

cout << "ctxt3 level: " << ctxt3.findBaseLevel() << endl;

ZZX dec_poly1, dec_poly2;
secret_key.Decrypt(dec_poly1, ctxt1);
secret_key.Decrypt(dec_poly2, ctxt3);
```

Multiplication followed by ciphertext refresh

Many Operations on Ciphertexts



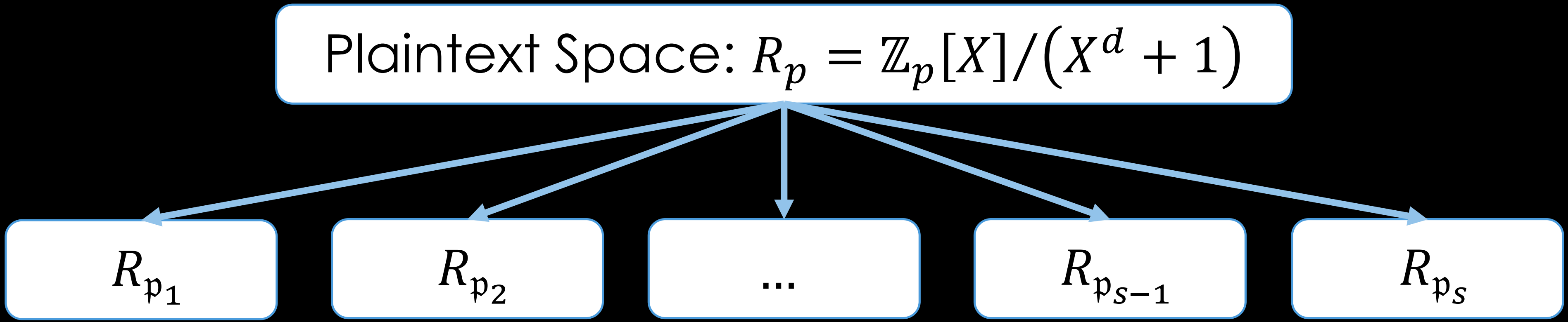
Ciphertext arithmetic

void	**negate** ()
Ctxt &	**operator+=** (const Ctxt &other)
Ctxt &	**operator-=** (const Ctxt &other)
void	**addCtxt** (const Ctxt &other, bool negative=false)
Ctxt &	**operator*=** (const Ctxt &other)
void	**automorph** (long k)
Ctxt &	**operator>>=** (long k)
void	**smartAutomorph** (long k) automorphism with re-lienarization
void	**frobeniusAutomorph** (long j) applies the automorphsim p^j using smartAutomorphism
void	**addConstant** (const DoubleCRT &dcrt, double size=-1.0)
void	**addConstant** (const ZZX &poly, double size=-1.0)
void	**addConstant** (const ZZ &c)

void	**multByConstant** (const DoubleCRT &dcrt, double size=-1.0)
void	**multByConstant** (const ZZX &poly, double size=-1.0)
void	**multByConstant** (const zzX &poly, double size=-1.0)
void	**multByConstant** (const ZZ &c)
void	**xorConstant** (const DoubleCRT &poly, double size=-1.0)
void	**xorConstant** (const ZZX &poly, double size=-1.0)
void	**nxorConstant** (const DoubleCRT &poly, double size=-1.0)
void	**nxorConstant** (const ZZX &poly, double size=-1.0)
void	**divideByP** ()
void	**multByP** (long e=1)
void	**divideBy2** ()
void	**extractBits** (vector< Ctxt > &bits, long nBits2extract=0)
void	**multiplyBy** (const Ctxt &other)
void	**multiplyBy2** (const Ctxt &other1, const Ctxt &other2)
void	**square** ()
void	**cube** ()
void	**power** (long e) raise ciphertext to some power

SIMD (Ciphertext Packing)

- Encrypt and pack **multiple** plaintext values into a **single** ciphertext.
- Main idea: **Chinese Remainder Theorem** over Polynomial Rings.



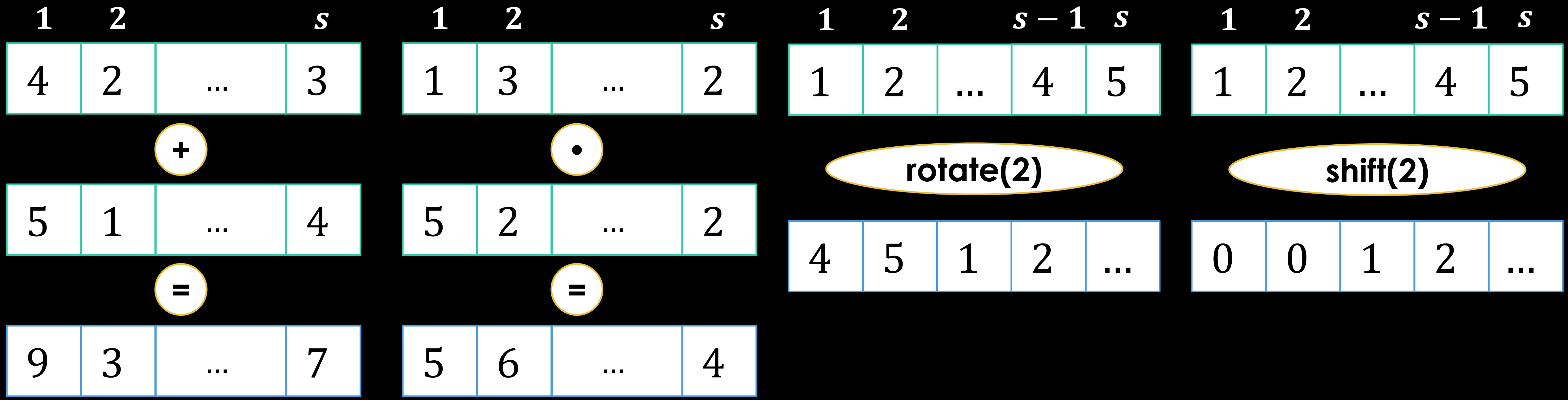
Plaintext Space: $R_p = \mathbb{Z}_p[X]/\left(X^d + 1\right)$

$R_{\mathfrak{p}_1}$ $R_{\mathfrak{p}_2}$ … $R_{\mathfrak{p}_{s-1}}$ $R_{\mathfrak{p}_s}$

- Choose p such that R_p splits into s smaller rings $R_{\mathfrak{p}_1}, \dots, R_{\mathfrak{p}_s}$

SIMD (Ciphertext Packing)

- This way we can process arrays of values at almost no extra cost.



1	2		s
4	2	…	3

$+$

5	1	…	4

$=$

9	3	…	7

1	2		s
1	3	…	2

$\cdot$

5	2	…	2

$=$

5	6	…	4

1	2		$s-1$	s
1	2	…	4	5

rotate(2)

4	5	1	2	…

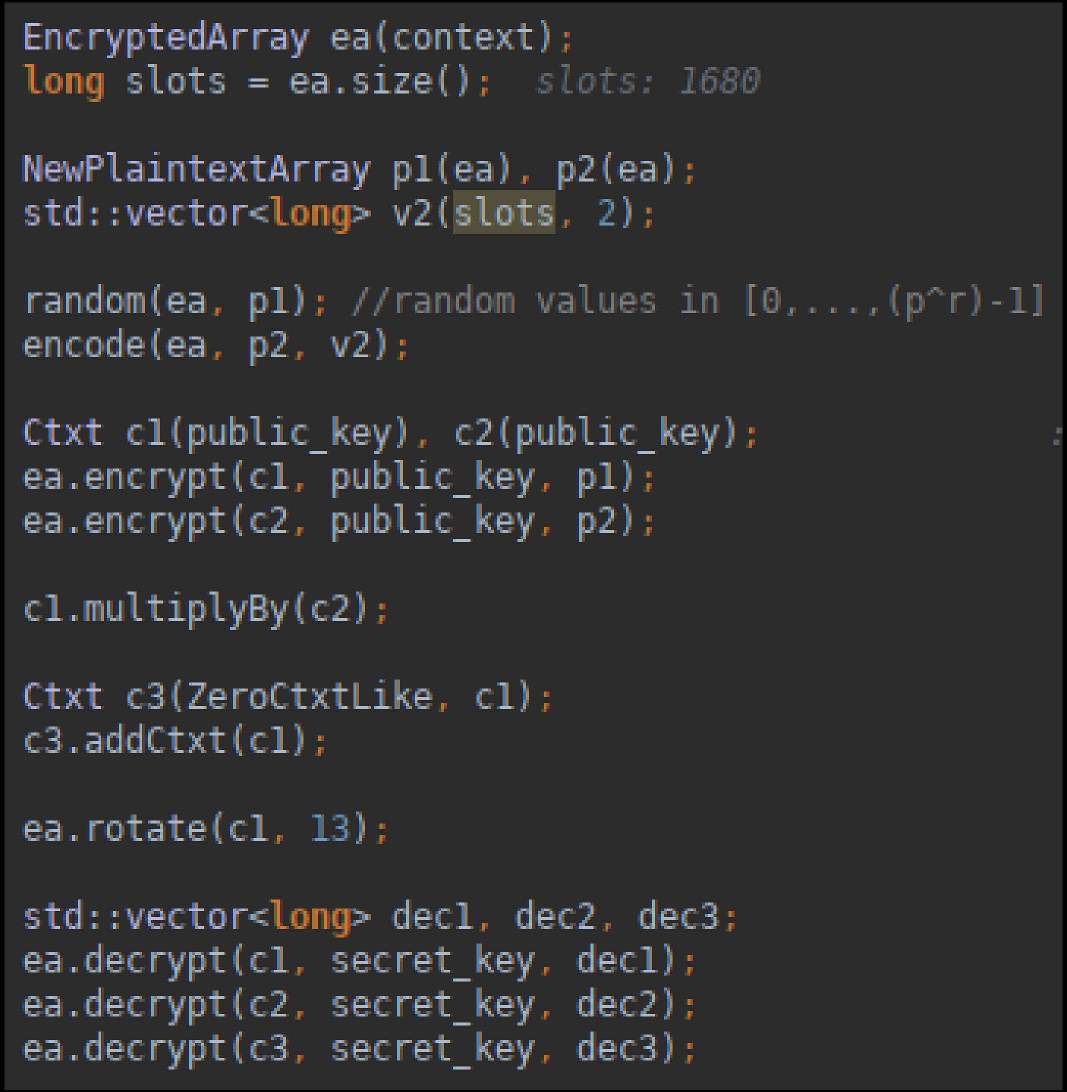
1	2		$s-1$	s
1	2	…	4	5

shift(2)

0	0	1	2	…

- In practice: hundreds – thousands of slots in each ciphertext

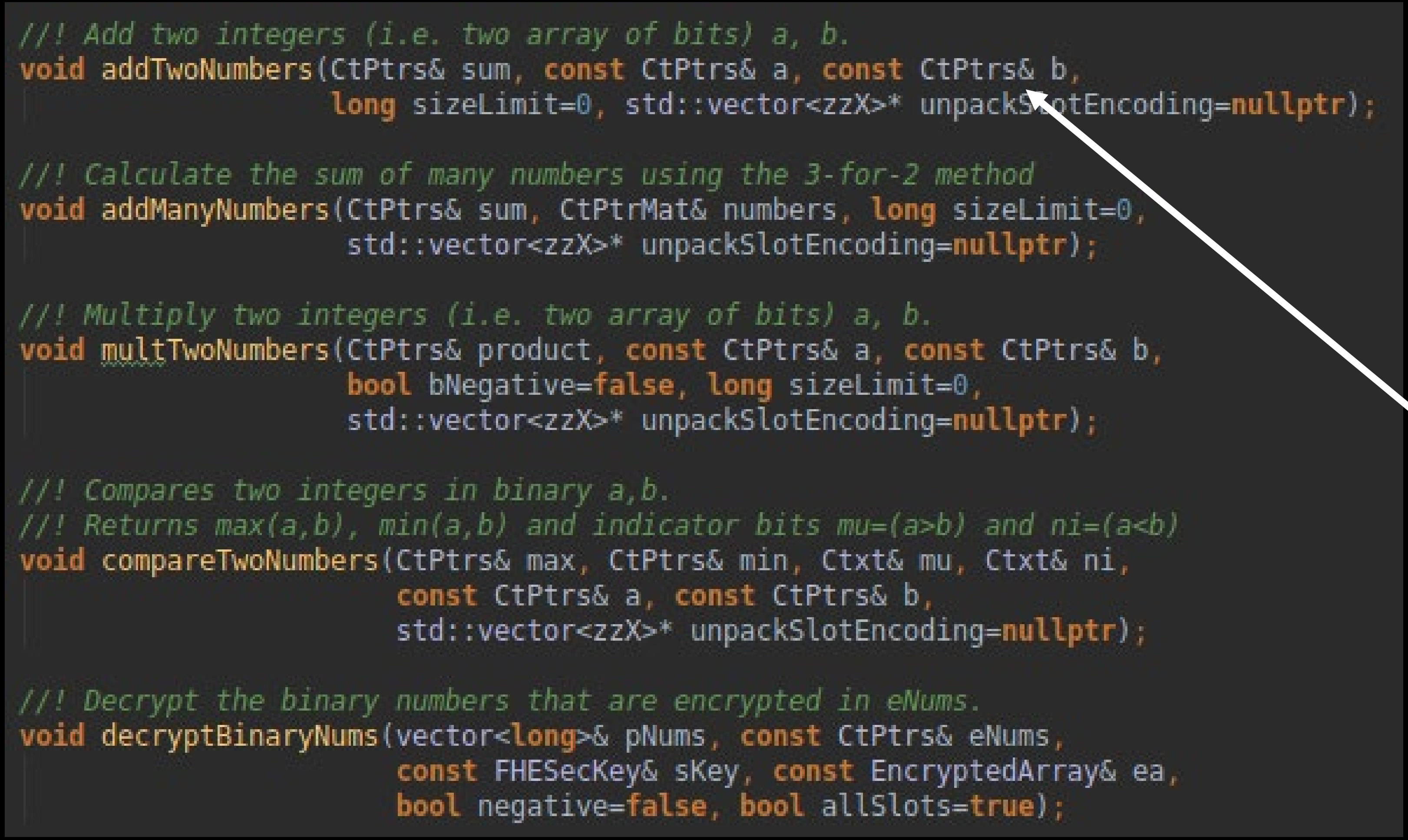
EncryptedArray:
Operations on Arrays of Slots



```
EncryptedArray ea(context);
long slots = ea.size();  slots: 1680

NewPlaintextArray p1(ea), p2(ea);
std::vector<long> v2(slots, 2);

random(ea, p1); //random values in [0,...,(p^r)-1]
encode(ea, p2, v2);

Ctxt c1(public_key), c2(public_key);
ea.encrypt(c1, public_key, p1);
ea.encrypt(c2, public_key, p2);

c1.multiplyBy(c2);

Ctxt c3(ZeroCtxtLike, c1);
c3.addCtxt(c1);

ea.rotate(c1, 13);

std::vector<long> dec1, dec2, dec3;
ea.decrypt(c1, secret_key, dec1);
ea.decrypt(c2, secret_key, dec2);
ea.decrypt(c3, secret_key, dec3);
```

Computing On Integers

- Includes routines for addition/multiplication and comparisons of integers in binary representation (binaryArith.h, binaryCompare.h):



```
//! Add two integers (i.e. two array of bits) a, b.
void addTwoNumbers(CtPtrs& sum, const CtPtrs& a, const CtPtrs& b,
                   long sizeLimit=0, std::vector<zzX>* unpackSlotEncoding=nullptr);

//! Calculate the sum of many numbers using the 3-for-2 method
void addManyNumbers(CtPtrs& sum, CtPtrMat& numbers, long sizeLimit=0,
                    std::vector<zzX>* unpackSlotEncoding=nullptr);

//! Multiply two integers (i.e. two array of bits) a, b.
void multTwoNumbers(CtPtrs& product, const CtPtrs& a, const CtPtrs& b,
                    bool bNegative=false, long sizeLimit=0,
                    std::vector<zzX>* unpackSlotEncoding=nullptr);

//! Compares two integers in binary a,b.
//! Returns max(a,b), min(a,b) and indicator bits mu=(a>b) and ni=(a<b)
void compareTwoNumbers(CtPtrs& max, CtPtrs& min, Ctxt& mu, Ctxt& ni,
                       const CtPtrs& a, const CtPtrs& b,
                       std::vector<zzX>* unpackSlotEncoding=nullptr);

//! Decrypt the binary numbers that are encrypted in eNums.
void decryptBinaryNums(vector<long>& pNums, const CtPtrs& eNums,
                       const FHESecKey& sKey, const EncryptedArray& ea,
                       bool negative=false, bool allSlots=true);
```

<ins>CtPtrs</ins>:
Unified interface for vector of pointers to ciphertexts

Part 4:
Fully Homomorphic Encryption

What have we achieved until now?

- We saw **Somewhat** and **Leveled** Homomorphic Encryption schemes.
- Still unable to compute arbitrary circuits / functions on encrypted data!
- Since suggested in 1978 by Rivest, Adleman and Dertouzos not feasible.
- This was the general situation until October 2008…

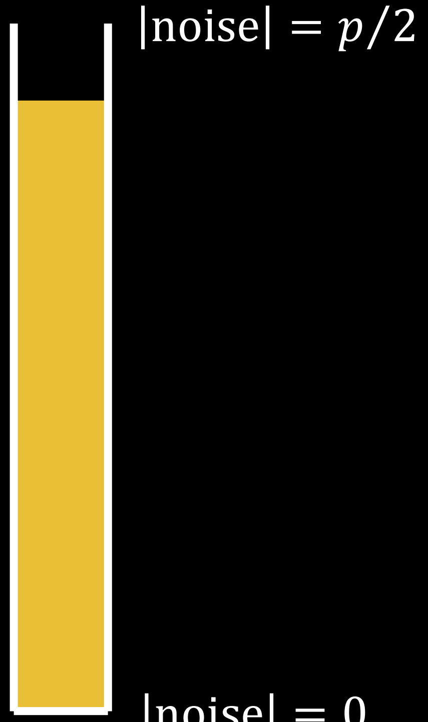
when Craig Gentry came up with the

first suggested scheme for a

Fully Homomorphic Encryption!


© Evan Kafka

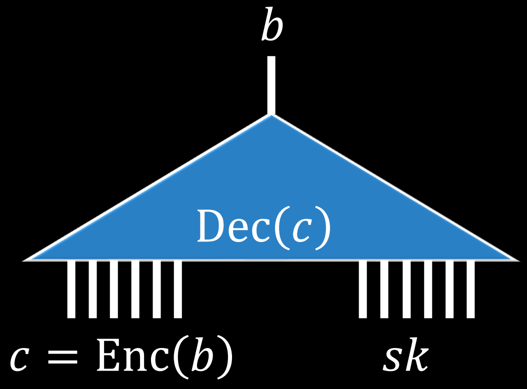
The “Bootstrapping method”



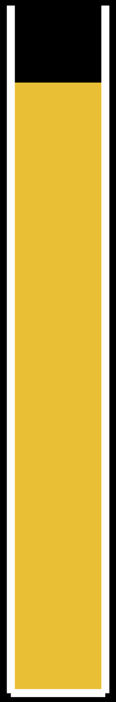
Decryption!

What is the best noise-reduction procedure?

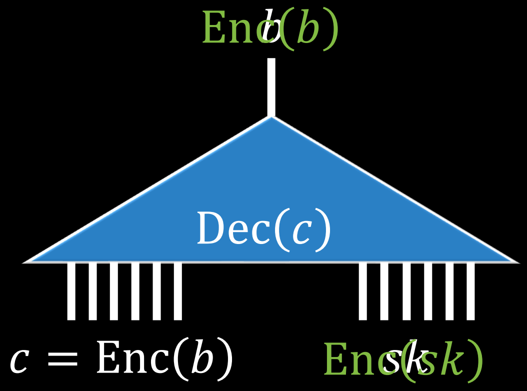
It acts on ciphertext and eliminates the noise



The “Bootstrapping method”



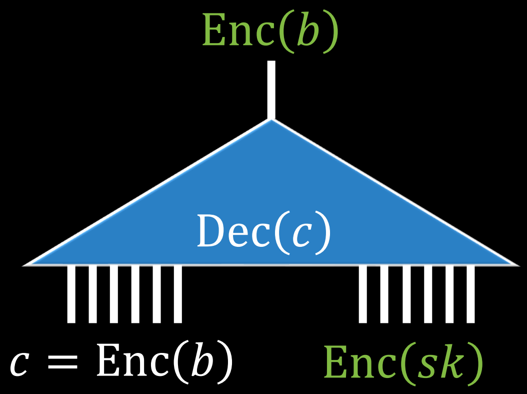
Now to reduce noise: New problem: Solution: homomorphically given Enc(sk) “Circular Security” evaluate Dec(c)!



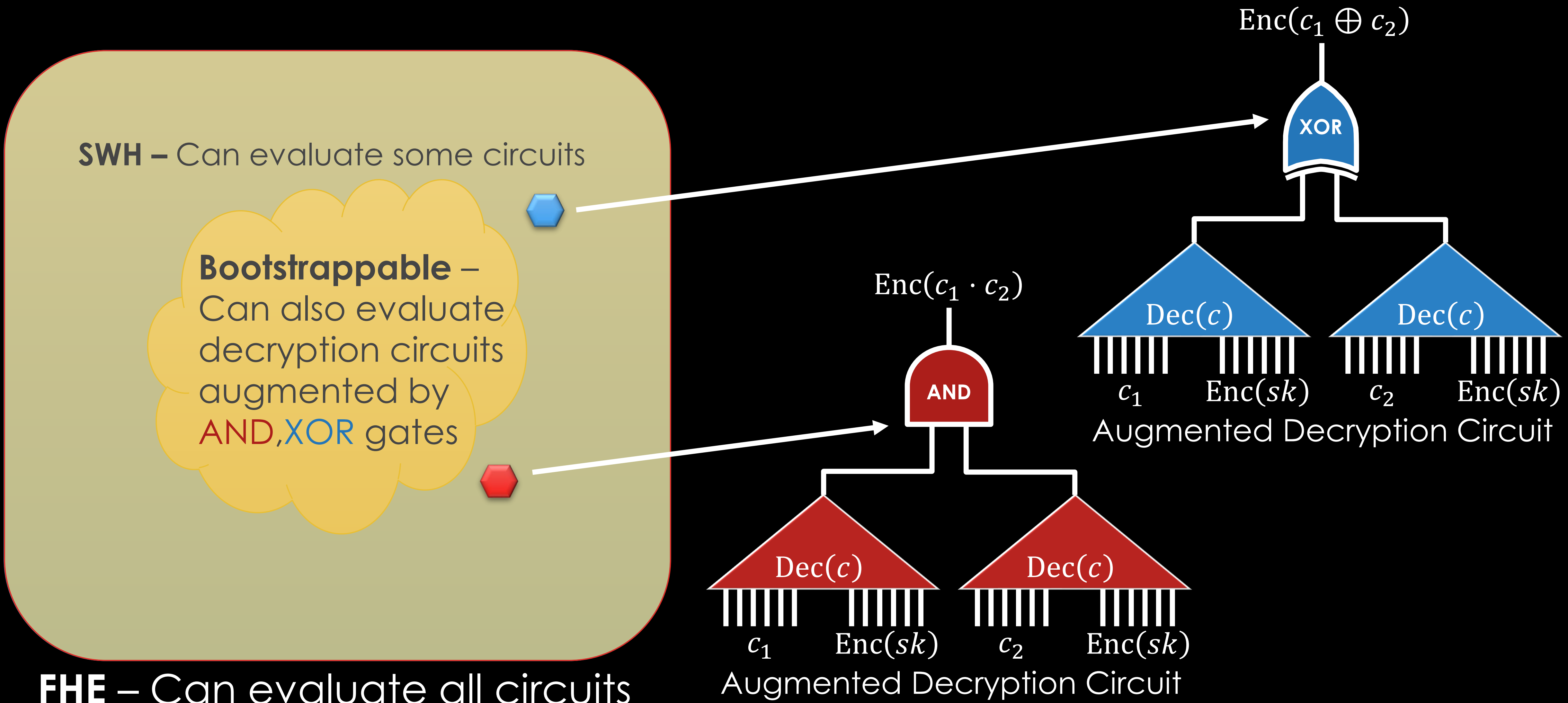
The “Bootstrapping method”

Regardless of the noise in the input $\text{Enc}(b)$, the noise level in the output $\text{Enc}(b)$ is fixed.

Bottomline: whenever noise increases beyond a limit, use bootstrapping to reset it to a fixed level, and repeat until done!



From “Somewhat” to “Fully”



SWH – Can evaluate some circuits

Bootstrappable – Can also evaluate decryption circuits augmented by AND,XOR gates

FHE – Can evaluate all circuits

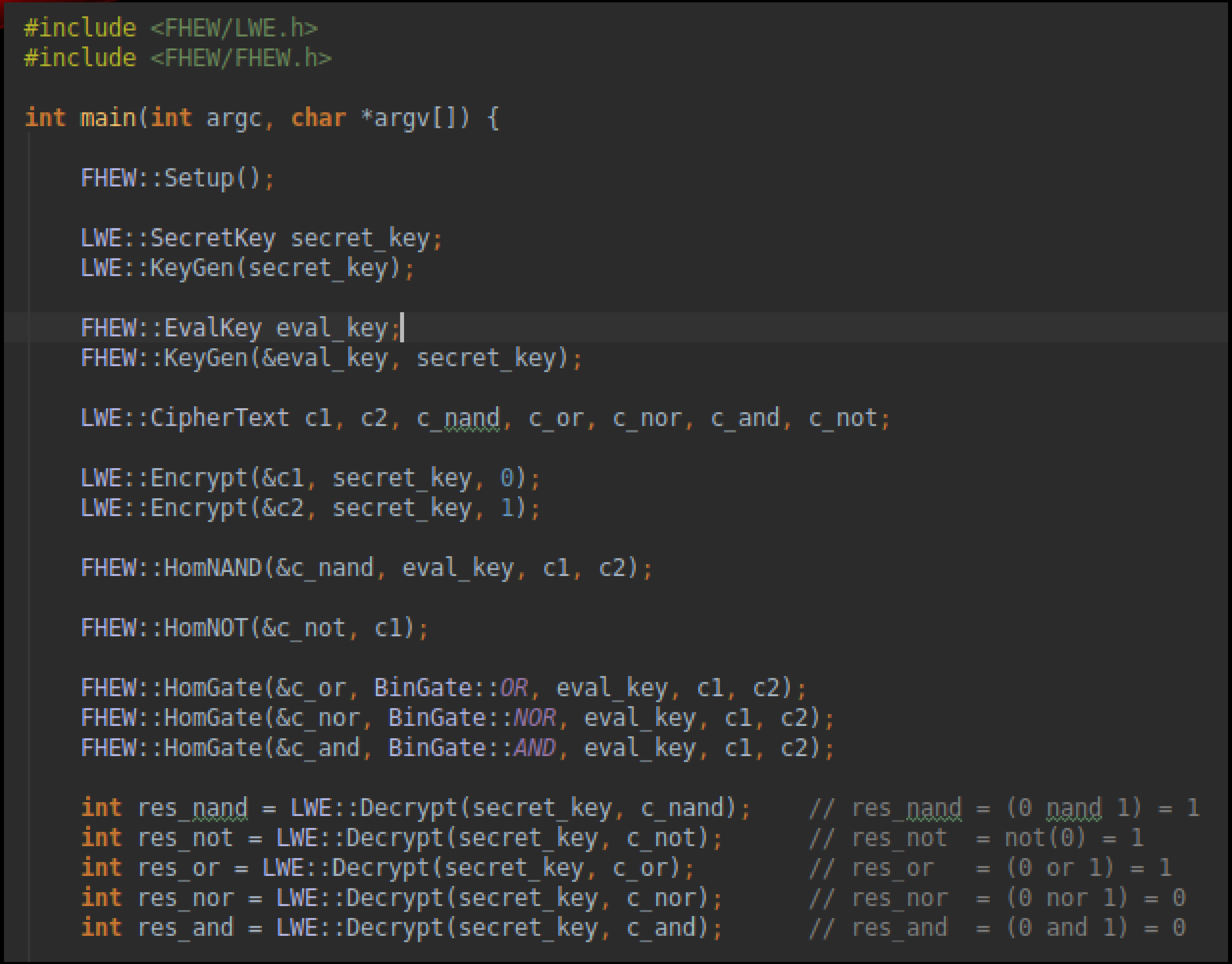
FHEW

https://github.com/lducas/FHEW

- Problem: Bootstrapping is an expensive process (5-30 min. per ctxt in HElib)

- Solution: Bootstrapping Homomorphic Encryption in less than a second!
  - Bootstrapping ciphertexts after every single operation.
  - Use of "cheap" and **functionally complete** NAND gate.
  - Works only over binary plaintext.

FHEW



```
#include <FHEW/LWE.h>
#include <FHEW/FHEW.h>

int main(int argc, char *argv[]) {

    FHEW::Setup();

    LWE::SecretKey secret_key;
    LWE::KeyGen(secret_key);

    FHEW::EvalKey eval_key;
    FHEW::KeyGen(&eval_key, secret_key);

    LWE::CipherText c1, c2, c_nand, c_or, c_nor, c_and, c_not;

    LWE::Encrypt(&c1, secret_key, 0);
    LWE::Encrypt(&c2, secret_key, 1);

    FHEW::HomNAND(&c_nand, eval_key, c1, c2);

    FHEW::HomNOT(&c_not, c1);

    FHEW::HomGate(&c_or, BinGate::OR, eval_key, c1, c2);
    FHEW::HomGate(&c_nor, BinGate::NOR, eval_key, c1, c2);
    FHEW::HomGate(&c_and, BinGate::AND, eval_key, c1, c2);

    int res_nand = LWE::Decrypt(secret_key, c_nand);    // res_nand = (0 nand 1) = 1
    int res_not = LWE::Decrypt(secret_key, c_not);      // res_not  = not(0) = 1
    int res_or = LWE::Decrypt(secret_key, c_or);        // res_or   = (0 or 1) = 1
    int res_nor = LWE::Decrypt(secret_key, c_nor);      // res_nor  = (0 nor 1) = 0
    int res_and = LWE::Decrypt(secret_key, c_and);      // res_and  = (0 and 1) = 0
```

TFHE

https://github.com/tfhe/tfhe

- Very fast gate-by-gate bootstrapping (≈ 13 milliseconds).
- Supports the homomorphic evaluation of the 10 binary gates (NAND, OR, AND, XOR, XNOR, NOR, etc.), as well as the negation (NOT) and the MUX(a,b,c) = a ? b : c gate.
- Both FHEW an TFHE are based on the GSW cryptosystem.

Additional Implementations & Links

- HEAAN – Supports fixed point arithmetics (also with Bootstrapping)
- SEAL – Well-documented C++ library by Microsoft
- PALISADE – General purpose C++ library for lattice cryptography
- cuFHE – CUDA (NVIDIA GPU) accelerated FHE library
- Daniele Micciancio FHE Page
- Vinod Vaikuntanathan FHE Page
- FHE Standardization Webpage

Summary

- We have seen:
  - computing over encrypted data is possible via FHE.
  - it is still quite challenging, not trivial, and relatively slow.
  - several C++ implementations of FHE exist.
- There are other methods to compute over encrypted data, e.g.:
  - Secure Multi-Party Computation (MPC)

Thank You!