

Національний технічний університет України

Київський політехнічний інститут

Фізико-технічний інститут

**Комп’ютерний практикум №4**

**з дисципліни «Методи реалізації криптографічних механізмів захисту інформації»**

**Варіант 2А**

**Виконав**:

Чорний Олег

ФІ-63М  
 **Перевірив**:

Фесенко А.В.

Київ 2017

1. **Схема шифрування Ель-Гамаля**

Схема шифрування Ель-Гамаля була запропонована у 1985 році. Стійкість даної схеми ґрунтується на складності розв’язання задачі дискретного логарифма у групі . Головним недоліком даної схеми є те, що отриманий із її застосуванням шифртекст є вдвічі довшим за початкове повідомлення.

**Побудова схеми шифрування Ель-Гамаля абонентом .**

Для цього абонент виконує наступні кроки.

1. Обирає велике просте число та – примітивний елемент поля ( і є відкритими параметрами).
2. Обирає випадкове число – секретний ключ , який використовується для розшифрування.
3. Обчислює . Параметри складають відкритий ключ, що застосовується для шифрування повідомлень, призначених для абонента .
4. Забезпечує доступність відкритого ключа для всіх абонентів, які бажають відіслати зашифроване повідомлення.

Умовно кажучи, відкритий ключ може бути опублікований.

**Шифрування повідомлення , абонентом для абонента**

Абонент , який знає алгоритм шифрування і відкритий ключ хоче переслати абонентові зашифроване повідомлення, що кодується і є відповідним цілим числом ,. Для цього він виконує наступні дії.

1. Вибирає випадкове число , .
2. Обчислює та .
3. Формує шифрований текст, що є впорядкованою парою чисел: .

**Розшифрування зашифрованого повідомлення абонентом**

Абонент обчислює .

1. **Генерація параметрів для схеми Ель-Гамаля**

Для роботи дана криптосистема потребує генерації двох основних параметрів: великого простого числа та - генератора групи . Для унеможливлення деяких атак, а також для простоти пошуку генератора , рекомендовано обирати модуль вигляду , де – велике просте число. Найбільш зручним є випадок , тоді . Для генерації числа довжиною біт використовується наступний алгоритм:

1. Генеруємо просте число довжиною біт (ймовірнісними або детермінованими методами).
2. Обчислюємо і перевіряємо його на простоту. Якщо тест встановив, що просте, то завершуємо роботу алгоритму. Інакше повертаємось на крок 1.

Для пошуку генератора можна використовувати класичний алгоритм, адаптований для випадку :

1. Обираємо випадкове число .
2. Якщо або , то повертаємось на крок 1. Інакше, завершуємо роботу алгоритму.

Проте для даного алгоритму існує кілька покращень. Перш за все, варто зауважити, що рівність виконується тільки для , тому виключивши дані числа із проміжку, в якому обирається випадкове число , дану умову можна взагалі не перевіряти.

Далі, помітимо, що, оскільки , то умова , згідно із критерієм Ейлера означає, що генератор не може бути квадратичним лишком. У випадку ці умови є еквівалентними, а отже, обчислення можна замінити обчисленням символу Якобі , що набагато простіше, з обчислювальної точки зору.

1. **Схема доповнення OAEP**

У своєму оригінальному вигляді більшість асиметричних криптосистем є вразливими до перебору відкритих текстів та до атак на основі обраних шифрованих текстів, тому вони потребують певної попередньої обробки повідомлень для запобігання таких атак. Одним із варіантів такої обробки є застосування схеми доповнення OAEP.

Стандарт PKCS 1v2 описує застосування схеми OAEP разом із криптосистемою RSA, про в силу своєї універсальності її можна застосовувати і разом з іншими асиметричними криптосистемами.

**Параметри та позначення.**

– хеш-функція ( позначає довжину виходу хеш-функції у байтах)

– функція генерації маски

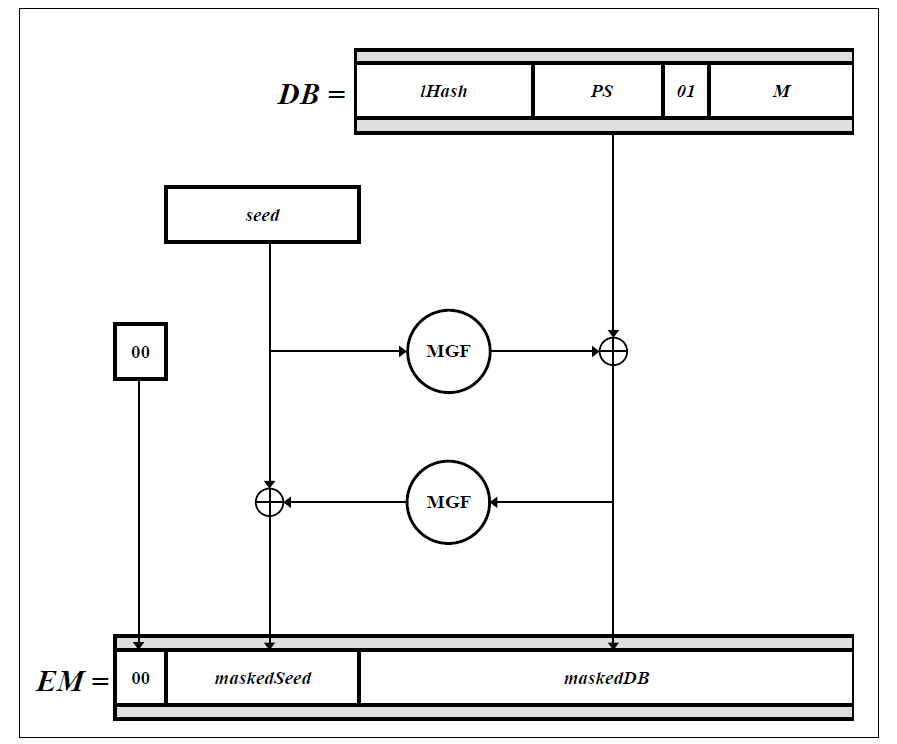
– довжина модуля (числа у схемі RSA або числа у схемі Ель-Гамаля) у байтах.

– довільна мітка, яка може бути додана до повідомлення. Значення за замовчуванням рівне порожньому рядку.

– довжина повідомлення, яке необхідно зашифрувати.

**Кодування повідомлення M в схемі OAEP.**

1. Якщо мітку не вказано, значення є порожнім рядком. Нехай , байтовий рядок довжини .
2. Генеруємо байтовий рядок , який складається із нульових байт. Довжина може бути рівна нулю.
3. Конкатенуємо , , байт із шістнадцятковим значенням і повідомлення , щоб сформувати блок даних довжиною байтів:
4. Генеруємо випадковий байтовий рядок довжиною .
5. Нехай .
6. Нехай .
7. Нехай .
8. Нехай .
9. Конкатенуємо байт із шістнадцятковим значення , та аби сформувати закодоване повідомлення довжиною байт:



Після кодування, до повідомлення EM, яке інтерпретується як число, застосовується примітив обраної асиметричної криптосистеми.

**Декодування повідомлення EM в схемі OAEP.**

1. Якщо мітку не вказано, значення є порожнім рядком. Нехай , байтовий рядок довжини .
2. Розділити закодоване повідомлення на одиничний байт , байтовий рядок довжиною і байтовий рядок довжиною :
3. Нехай .
4. Нехай .
5. Нехай .
6. Нехай .
7. Розділити на байтовий рядок довжини , (можливо порожній) рядок доповнення , що складається із байтів із шістнадцятковим значенням і повідомлення M:

Якщо немає байту із шістнадцятковим значенням , яке відділяє від , якщо не дорівнює або якщо не дорівнює нулю, то повідомляємо про помилку дешифрування і завершуємо роботу.

1. Повертаємо повідомлення .

**Функція генерації маски.**

Функція генерації маски отримує байтовий рядок змінної довжини і бажану довжину виходу на вхід і повертає байтовий рядок бажаної довжини. Функція генерації маски є детермінованою: рядок виходу повністю визначається вхідним рядком. Вихід функції генерації маски повинен бути псевдовипадковим: маючи частину виходу, але не маючи входу, має бути неможливим передбачити іншу частину виходу.

**MGF1**

це функція генерації маски на основі хеш-функції. Хеш-функцію можна обрати довільну.

***Позначення:***

– хеш-фунція ( позначає довжину виходу).

– вхідний вектор, на основі якого генерується маска

– бажана довжина маски у байтах

– маска, яка повертається на виході.

***Алгоритм:***

1. Нехай це порожній байтовий рядок.
2. Для від до виконуємо наступне:
   1. Конвертуємо у байтовий рядок довжини 4 наступним чином:

Нехай . Тоді .

* 1. Конкатенуємо значення хеш-функції від та до байтового рядка :

1. Повертаємо перші байтів рядка у якості байтового рядка .
2. **Вихідний код**

**public class** ElGamalKeyGenerator {  
  
 **private static final** BigInteger ***ONE*** = BigInteger.*valueOf*(1);  
 **private static final** BigInteger ***TWO*** = BigInteger.*valueOf*(2);  
  
 **private static final int *CERTAINTY*** = 100;  
  
 **private final** Random **random**;  
  
 **public** ElGamalKeyGenerator() {  
 **this**(**new** Random());  
 }  
  
 **public** ElGamalKeyGenerator(Random random) {  
 **this**.**random** = random;  
 }  
  
 **public** ElGamalKeyPair generateKeyPair(**int** pBitLength) {  
 BigInteger[] safePrimes = *generateSafePrimes*(pBitLength, ***CERTAINTY***, **this**.**random**);  
  
 BigInteger p = safePrimes[0];  
 BigInteger q = safePrimes[1];  
  
 BigInteger g = *selectGenerator*(p, q, **this**.**random**);  
 BigInteger x = RandomUtils.*createRandomInRange*(***TWO***, p.subtract(***TWO***), **this**.**random**);  
 BigInteger y = g.modPow(x, p);  
  
 ElGamalPrivateKey privateKey = **new** ElGamalPrivateKey(p, g, x);  
 ElGamalPublicKey publicKey = **new** ElGamalPublicKey(p, g, y);  
  
 **return new** ElGamalKeyPair(publicKey, privateKey);  
 }  
  
 **private static** BigInteger[] generateSafePrimes(**int** size, **int** certainty, Random random) {  
 BigInteger p, q;  
 **int** qLength = size - 1;  
  
 **do** {  
 q = **new** BigInteger(qLength, certainty, random);  
  
 *// p = 2q + 1* p = q.shiftLeft(1).add(***ONE***);  
  
 } **while** (!p.isProbablePrime(certainty));  
  
 **return new** BigInteger[]{p, q};  
 }  
  
 **private static** BigInteger selectGenerator(BigInteger p, BigInteger q, Random random) {  
 BigInteger pMinusTwo = p.subtract(***TWO***);  
 BigInteger g;  
  
 **do** {  
 g = RandomUtils.*createRandomInRange*(***TWO***, pMinusTwo, random);  
 }  
 **while** (g.modPow(***TWO***, p).equals(***ONE***) || g.modPow(q, p).equals(***ONE***));  
  
 **return** g;  
 }  
}

**public interface** AsymmetricBlockCipher {  
  
 **void** init(**boolean** forEncryption, CipherParameters param);  
  
 **byte**[] processBlock(**byte**[] block, **int** offset, **int** length);  
  
 **int** getInputBlockSize();  
  
 **int** getOutputBlockSize();  
}

**public class** Cipher {  
  
 **private static final byte**[] ***EMPTY\_BUFFER*** = **new byte**[0];  
  
 **private final** AsymmetricBlockCipher **engine**;  
  
 **private byte**[] **buff**;  
 **private int buffOffset**;  
  
 **public** Cipher(AsymmetricBlockCipher engine) {  
 **this**.**engine** = engine;  
 }  
  
 **public void** init(**boolean** forEncryption, CipherParameters parameters) {  
 **this**.**engine**.init(forEncryption, parameters);  
 **this**.**buff** = **new byte**[**this**.**engine**.getInputBlockSize()];  
 }  
  
 **public int** getInputBlockSize() {  
 **return engine**.getInputBlockSize();  
 }  
  
 **public int** getOutputBlockSize() {  
 **return engine**.getOutputBlockSize();  
 }  
  
 **public byte**[] update(**byte**[] input) {  
 **return** update(input, 0, input.**length**);  
 }  
  
 **public byte**[] update(**byte**[] input, **int** offset, **int** length) {  
 **final int** inputBlockSize = **engine**.getInputBlockSize();  
 **final int** outputBlockSize = **engine**.getOutputBlockSize();  
  
 **final int** totalInputLen = length + **buffOffset**;  
  
 **if** (totalInputLen < inputBlockSize) {  
 copyToBuffer(input, offset, length);  
  
 **return *EMPTY\_BUFFER***;  
 }  
  
 **final int** outputBlocks = totalInputLen / inputBlockSize;  
 **final int** outputLength = outputBlocks \* outputBlockSize;  
  
 **byte**[] outputBuffer = **new byte**[outputLength];  
  
 **int** bytesWritten = doUpdate(input, offset, length, outputBuffer, 0);  
 **return** EncryptionUtils.*slice*(outputBuffer, 0, bytesWritten);  
 }  
  
 **public int** update(**byte**[] input, **int** offset, **int** length,  
 **byte**[] output, **int** outputOffset) {  
  
 **final int** inputBlockSize = **engine**.getInputBlockSize();  
 **final int** outputBlockSize = **engine**.getOutputBlockSize();  
  
 **final int** totalInputLen = length + **buffOffset**;  
  
 **if** (totalInputLen < inputBlockSize) {  
 copyToBuffer(input, offset, length);  
  
 **return** 0;  
 }  
  
 **final int** outputBlocks = totalInputLen / inputBlockSize;  
 **final int** outputLength = outputBlocks \* outputBlockSize;  
  
 **if** (outputLength > output.**length** - outputOffset) {  
 **throw new** IllegalStateException(**"Output buffer doesn't have enough space"**);  
 }  
  
 **return** doUpdate(input, offset, length, output, outputOffset);  
 }  
  
 **private int** doUpdate(**byte**[] input, **int** offset, **int** length,  
 **byte**[] output, **int** outputOffset) {  
  
 **final int** inputBlockSize = **engine**.getInputBlockSize();  
 **final int** totalInputLength = **buffOffset** + length;  
 **final int** outputBlocks = totalInputLength / inputBlockSize;  
  
 **int** bytesWritten = 0;  
  
 **for** (**int** i = 0; i < outputBlocks; i++) {  
  
 **byte**[] processedBytes;  
  
 **if** (**buffOffset** != 0) {  
 offset += fillBuffer(input, offset);  
 processedBytes = **engine**.processBlock(**buff**, 0, **buff**.**length**);  
  
 cleanBuffer();  
 } **else** {  
 processedBytes = **engine**.processBlock(input, offset, inputBlockSize);  
 offset += inputBlockSize;  
 }  
  
 System.*arraycopy*(processedBytes, 0, output, outputOffset, processedBytes.**length**);  
 outputOffset += processedBytes.**length**;  
 bytesWritten += processedBytes.**length**;  
 }  
  
 **int** remainedBytes = totalInputLength - outputBlocks \* inputBlockSize;  
 copyToBuffer(input, offset, remainedBytes);  
  
 **return** bytesWritten;  
 }  
  
 **private int** fillBuffer(**byte**[] input, **int** offset) {  
 **int** bytesToCopy = **engine**.getInputBlockSize() - **buffOffset**;  
 copyToBuffer(input, offset, bytesToCopy);  
  
 **return** bytesToCopy;  
 }  
  
 **private void** copyToBuffer(**byte**[] input, **int** offset, **int** length) {  
 System.*arraycopy*(input, offset, **buff**, **buffOffset**, length);  
 **buffOffset** += length;  
 }  
  
 **private void** cleanBuffer() {  
 Arrays.*fill*(**buff**, (**byte**) 0);  
 **buffOffset** = 0;  
 }  
  
 **public byte**[] doFinal() {  
  
 **if** (**buffOffset** == 0) {  
 **return *EMPTY\_BUFFER***;  
 }  
  
 **byte**[] finalBlock = **engine**.processBlock(**buff**, 0, **buffOffset**);  
 **buffOffset** = 0;  
  
 **return** finalBlock;  
 }  
}

**public interface** MGF {  
  
 **byte**[] mask(**byte**[] seed, **int** seedOffset, **int** seedLength,  
 **int** targetLength);  
  
 **default byte**[] mask(**byte**[] seed, **int** length) {  
 **return** mask(seed, 0, seed.**length**, length);  
 }  
}

**public class** MGF1 **implements** MGF {  
  
 **private final** MessageDigest **hash**;  
 **private final int hLen**;  
  
 **public** MGF1(MessageDigest hash) {  
 **this**.**hash** = hash;  
 **this**.**hLen** = hash.getDigestLength();  
 }  
  
 @Override  
 **public byte**[] mask(**byte**[] seed, **int** seedOffset, **int** seedLength, **int** length) {  
 **byte**[] mask = **new byte**[length];  
 **byte**[] C = **new byte**[4];  
  
 *// C = I2OSP(counter, 4)  
 // T = T || HASH(seed || C)* **final int** iterations = length / **hLen**;  
 **int** maskPos = 0;  
 **int** counter = 0;  
  
 **while** (counter < iterations) {  
 i2osp(counter, C);  
 System.*arraycopy*(hash(seed, seedOffset, seedLength, C), 0, mask, maskPos, **hLen**);  
  
 maskPos += **hLen**;  
 counter++;  
 }  
  
 **if** (maskPos < length) {  
 i2osp(counter, C);  
 System.*arraycopy*(hash(seed, seedOffset, seedLength, C), 0, mask, maskPos, length - maskPos);  
 }  
  
 **return** mask;  
 }  
  
 **private byte**[] hash(**byte**[] seed, **int** seedOffset, **int** seedLength, **byte**[] C) {  
 **hash**.update(seed, seedOffset, seedLength);  
 **hash**.update(C);  
 **return hash**.digest();  
 }  
  
 **private void** i2osp(**int** value, **byte**[] holder) {  
 holder[0] = (**byte**) (value & 0xFF);  
 holder[1] = (**byte**) ((value >> 8) & 0xFF);  
 holder[2] = (**byte**) ((value >> 16) & 0xFF);  
 holder[3] = (**byte**) ((value >> 24) & 0xFF);  
 }  
}

**public class** OAEPPadding **implements** AsymmetricBlockCipher {  
  
 **private final** Random **random** = **new** Random();  
  
 **private final** AsymmetricBlockCipher **engine**;  
 **private final** MGF **mgf**;  
  
 **private final byte**[] **lHash**;  
 **private final int hLen**;  
  
 **private boolean forEncryption**;  
  
 **public** OAEPPadding(AsymmetricBlockCipher cipher) {  
 **this**(cipher, DigestFactory.*createSHA1*());  
 }  
  
 **public** OAEPPadding(  
 AsymmetricBlockCipher cipher,  
 MessageDigest hash) {  
 **this**(cipher, hash, **new** MGF1(hash));  
 }  
  
 **public** OAEPPadding(  
 AsymmetricBlockCipher cipher,  
 MessageDigest hash,  
 MGF mgf) {  
 **this**(cipher, hash, mgf, **null**);  
 }  
  
 **public** OAEPPadding(  
 AsymmetricBlockCipher cipher,  
 MessageDigest hash,  
 MGF mgf,  
 **byte**[] label) {  
 **this**.**engine** = cipher;  
 **this**.**mgf** = mgf;  
  
 hash.reset();  
  
 **if** (label != **null**) {  
 hash.update(label);  
 }  
  
 **this**.**lHash** = hash.digest();  
 **this**.**hLen** = **lHash**.**length**;  
 }  
  
 @Override  
 **public void** init(**boolean** forEncryption, CipherParameters param) {  
 **this**.**forEncryption** = forEncryption;  
 **this**.**engine**.init(forEncryption, param);  
  
 checkEngineInputLength();  
 }  
  
 **private void** checkEngineInputLength() {  
 **if** (**this**.**forEncryption** && **this**.getInputBlockSize() < 0) {  
 **throw new** IllegalStateException(**"Size of input block of supplied engine"** +  
 **" is less than necessary for using it with the OAEP scheme"**);  
 }  
  
 **if** (!**this**.**forEncryption** && **this**.getOutputBlockSize() < 0) {  
 **throw new** IllegalStateException(**"Size of output block of supplied engine"** +  
 **" is less than necessary for using it with the OAEP scheme"**);  
  
 }  
 }  
  
 @Override  
 **public byte**[] processBlock(**byte**[] block, **int** offset, **int** length) {  
 **if** (**forEncryption**) {  
 **return** encode(block, offset, length);  
 } **else** {  
 **return** decode(block, offset, length);  
 }  
 }  
  
 **private byte**[] encode(**byte**[] block, **int** offset, **int** length) {  
 EncryptionUtils.*checkInputLengthLTE*(getInputBlockSize(), length);  
  
 **final int** k = **engine**.getInputBlockSize();  
 **final int** dbLen = k - **hLen** - 1;  
  
 **byte**[] em = **new byte**[k];  
 **byte**[] seed = randomBytes(**hLen**);  
  
 *// 00, seed, DB = lHash | 00 ... 00 | 01 | M* System.*arraycopy*(**lHash**, 0, em, **hLen** + 1, **hLen**);  
 em[k - 1 - length] = 1;  
 System.*arraycopy*(block, offset, em, k - length, length);  
  
 *//maskedDB = DB ^ MGF(seed)  
 xor*(em, **hLen** + 1, **mgf**.mask(seed, dbLen), 0, dbLen);  
  
 *//maskedSeed = seed ^ maskedDB  
 xor*(seed, 0, **mgf**.mask(em, **hLen** + 1, dbLen, **hLen**), 0, **hLen**);  
 System.*arraycopy*(seed, 0, em, 1, **hLen**);  
  
 *//EM = 00 | maskedSeed | maskedDB* **return engine**.processBlock(em, 0, em.**length**);  
 }  
  
 **private byte**[] decode(**byte**[] input, **int** offset, **int** length) {  
 **byte**[] decrypted = **engine**.processBlock(input, offset, length);  
  
 **final int** k = **engine**.getOutputBlockSize();  
 EncryptionUtils.*checkInputLengthLTE*(k, decrypted.**length**);  
  
 *//we need to extend block with leading zeros to necessary length* **byte**[] block = **new byte**[**engine**.getOutputBlockSize()];  
 System.*arraycopy*(decrypted, 0, block, block.**length** - decrypted.**length**, decrypted.**length**);  
  
 **final int** dbLen = k - **hLen** - 1;  
  
 *xor*(block, 1, **mgf**.mask(block, **hLen** + 1, dbLen, **hLen**), 0, **hLen**);  
 *xor*(block, **hLen** + 1, **mgf**.mask(block, 1, **hLen**, dbLen), 0, dbLen);  
  
 **if** (block[0] != 0) {  
 **throw new** IllegalStateException(**"First byte should be 0x00"**);  
 }  
  
 **for** (**int** i = 0; i < **hLen**; i++) {  
 **if** (**lHash**[i] != block[**hLen** + 1 + i]) {  
 **throw new** IllegalStateException(**"Label hash mismatch the original value"**);  
 }  
 }  
  
 *// find the position where real message starts. In other words - skip padding bytes.* **int** mPos;  
 **for** (mPos = 2 \* **hLen** + 1; mPos < k; mPos++) {  
 **if** (block[mPos] == 1) {  
 mPos += 1;  
 **break**;  
 }  
  
 **if** (block[mPos] != 0) {  
 **throw new** IllegalArgumentException(**"Invalid padding"**);  
 }  
 }  
  
 **return** Arrays.*copyOfRange*(block, mPos, k);  
 }  
  
 @Override  
 **public int** getInputBlockSize() {  
 **if** (**this**.**forEncryption**) {  
 **return engine**.getInputBlockSize() - 2 \* **hLen** - 2;  
 } **else** {  
 **return engine**.getInputBlockSize();  
 }  
 }  
  
 @Override  
 **public int** getOutputBlockSize() {  
 **if** (**this**.**forEncryption**) {  
 **return engine**.getOutputBlockSize();  
 } **else** {  
 **return engine**.getOutputBlockSize() - 2 \* **hLen** - 2;  
 }  
 }  
  
 **private byte**[] randomBytes(**int** length) {  
 **byte**[] randomBuffer = **new byte**[length];  
 **random**.nextBytes(randomBuffer);  
  
 **return** randomBuffer;  
 }  
  
 **private static void** xor(**byte**[] accumulator,  
 **int** accOffset,  
 **byte**[] bytes,  
 **int** offset,  
 **int** length) {  
 **for** (**int** i = 0; i < length; i++) {  
 accumulator[i + accOffset] ^= bytes[i + offset];  
 }  
 }  
}

**public class** ElGamalCipher **implements** AsymmetricBlockCipher {  
  
 **private static final** BigInteger ***TWO*** = BigInteger.*valueOf*(2);  
  
 **private final** Random **random**;  
  
 **private** ElGamalKey **key**;  
 **private int bitSize**;  
 **private boolean forEncryption**;  
  
 **public** ElGamalCipher() {  
 **this**(**new** Random());  
 }  
  
 **public** ElGamalCipher(Random random) {  
 **this**.**random** = random;  
 }  
  
 @Override  
 **public void** init(**boolean** forEncryption, CipherParameters param) {  
 ElGamalCipherParameters egParams = (ElGamalCipherParameters) param;  
  
 **this**.**key** = egParams.getKey();  
 **this**.**bitSize** = **this**.**key**.getP().bitLength();  
 **this**.**forEncryption** = forEncryption;  
  
 **if** (**this**.**forEncryption** && !(**this**.**key instanceof** ElGamalPublicKey)) {  
 **throw new** IllegalArgumentException(**"Key type doesn't match working mode"**);  
 }  
 }  
  
 @Override  
 **public byte**[] processBlock(**byte**[] block, **int** offset, **int** length) {  
 EncryptionUtils.*checkInputLengthLTE*(getInputBlockSize(), length);  
  
 **if** (**forEncryption**) {  
 **return** encrypt(block, offset, length);  
 } **else** {  
 **return** decrypt(block, offset, length);  
 }  
 }  
  
 **private byte**[] encrypt(**byte**[] block, **int** offset, **int** length) {  
 **if** (length > getInputBlockSize()) {  
 **throw new** IllegalArgumentException(**"Input block too large for current ElGamal parameters"**);  
 }  
  
 **byte**[] input = *copyIfNecessary*(block, offset, length);  
  
 BigInteger m = ConversionUtil.*fromUnsignedByteArray*(input);  
 BigInteger[] cipherText = encrypt((ElGamalPublicKey) **key**, m);  
  
 **return** ConversionUtil.*packToByteArray*(cipherText[0], cipherText[1], getOutputBlockSize());  
 }  
  
 **private** BigInteger[] encrypt(ElGamalPublicKey key, BigInteger m) {  
  
 BigInteger g = key.getG();  
 BigInteger p = key.getP();  
 BigInteger y = key.getY();  
  
 BigInteger k = RandomUtils.*createRandomInRange*(***ONE***, p.subtract(***TWO***), **random**);  
  
 *//c1 = g^k (mod p)  
 //c2 = y^k \* M (mod p) = g^(kx) \* M (mod p)* BigInteger c1 = g.modPow(k, p);  
 BigInteger c2 = y.modPow(k, p).multiply(m).mod(p);  
  
 **return new** BigInteger[]{c1, c2};  
 }  
  
 **private byte**[] decrypt(**byte**[] block, **int** offset, **int** length) {  
 **if** (length > getInputBlockSize()) {  
 **throw new** IllegalArgumentException(**"Input block too large for current ElGamal parameters"**);  
 }  
  
 BigInteger[] c = ConversionUtil.*unpackFromByteArray*(block, offset, length);  
 BigInteger plainText = decrypt((ElGamalPrivateKey) **key**, c);  
  
 **return** ConversionUtil.*asUnsignedByteArray*(plainText);  
 }  
  
 **private** BigInteger decrypt(ElGamalPrivateKey key, BigInteger[] c) {  
 BigInteger p = key.getP();  
 BigInteger x = key.getX();  
  
 BigInteger c1 = c[0];  
 BigInteger c2 = c[1];  
  
 *//M = c2 \* (c1^x)^(-1) (mod p) = g^(kx) \* M \* (g^(kx))^(-1) (mod p) = M* **return** c2.multiply(c1.modPow(x, p).modInverse(p)).mod(p);  
 }  
  
 @Override  
 **public int** getInputBlockSize() {  
 **if** (**forEncryption**) {  
 **return** (**bitSize** - 1) / 8;  
 }  
  
 **return** 2 \* ((**bitSize** + 7) / 8);  
 }  
  
 @Override  
 **public int** getOutputBlockSize() {  
 **if** (**forEncryption**) {  
 **return** 2 \* ((**bitSize** + 7) / 8);  
 }  
  
 **return** (**bitSize** - 1) / 8;  
 }  
  
 **private static byte**[] copyIfNecessary(**byte**[] block, **int** offset, **int** length) {  
 **if** (offset == 0 && length == block.**length**) {  
 **return** block;  
 }  
  
 **return** Arrays.*copyOfRange*(block, offset, offset + length);  
 }  
}

**public class** EncryptionUtils {  
  
 **public static byte**[] slice(**byte**[] source, **int** srcOffset, **int** targetLength) {  
 **byte**[] sliced = **new byte**[targetLength];  
  
 System.*arraycopy*(source, srcOffset, sliced, 0, targetLength);  
  
 **return** sliced;  
 }  
  
 **public static byte**[] concat(**byte**[] left, **byte**[] right) {  
  
 **byte**[] result = **new byte**[left.**length** + right.**length**];  
  
 System.*arraycopy*(left, 0, result, 0, left.**length**);  
 System.*arraycopy*(right, 0, result, left.**length**, right.**length**);  
  
 **return** result;  
 }  
  
 **public static void** checkInputLengthEQ(**int** expected, **int** actual) {  
 **if** (expected != actual) {  
 **throw new** IllegalArgumentException(**"Input message has incorrect length: "** +  
 **"expected = "** + expected +  
 **", actual = "** + actual);  
 }  
 }  
  
 **public static void** checkInputLengthLTE(**int** expected, **int** actual) {  
 **if** (expected < actual) {  
 **throw new** IllegalArgumentException(**"Input message too long"**);  
 }  
 }  
}