Bloki szyfrowe - sprawozdanie

Katarzyna Jaromirska 155910

# Pomiar czasów szyfrowania i deszyfrowania dla wybranych trybów

## Implementacja

Do szyfrowania użyto biblioteki javax.crypto, który zawiera implementację bloków szyfrowych jak i potrzebne funkcje do parametrów tych bloków. W celu dokładnego pomiaru czasów użyto biblioteki benchmarkowej org.openjdk.jmh.

import org.openjdk.jmh.annotations.\*;  
  
import javax.crypto.Cipher;  
import javax.crypto.KeyGenerator;  
import javax.crypto.SecretKey;  
import javax.crypto.spec.IvParameterSpec;  
import java.security.SecureRandom;  
import java.util.concurrent.TimeUnit;  
  
@BenchmarkMode(Mode.AverageTime)  
@OutputTimeUnit(TimeUnit.NANOSECONDS)  
@State(Scope.Thread)  
public class BlockCipherBenchmark {  
 @Param({  
 "AES/ECB/PKCS5Padding",  
 "AES/CBC/PKCS5Padding",  
 "AES/CTR/NoPadding"

})  
 public String transformation;  
  
 @Param({  
 "262144",

"524288",

"1048576"

})  
 public int size;  
  
 private SecretKey key;  
 Cipher cipher;  
  
 private byte[] iv;  
  
 byte[] messageBytes, encryptedBytes;  
  
 @Setup(Level.Trial)  
 public void initialize() throws Exception {  
 messageBytes = new byte[size];  
 SecureRandom.getInstanceStrong().nextBytes(messageBytes);  
  
 KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");  
 keyGenerator.init(128);  
 key = keyGenerator.generateKey();  
  
 if (transformation.contains("CBC") || transformation.contains("CTR")) {  
 iv = new byte[16];  
 SecureRandom.getInstanceStrong().nextBytes(iv);  
 }  
  
 Cipher encryptionCipher = Cipher.getInstance(transformation);  
 if (iv != null) {  
 encryptionCipher.init(Cipher.ENCRYPT\_MODE, key, new IvParameterSpec(iv));  
 } else {  
 encryptionCipher.init(Cipher.ENCRYPT\_MODE, key);  
 }  
 encryptedBytes = encryptionCipher.doFinal(messageBytes);  
  
 encrypt();  
 decrypt();  
 }  
  
 @Benchmark

public byte[] encrypt() throws Exception {  
 Cipher cipher = Cipher.getInstance(transformation);  
 if (iv != null) {  
 cipher.init(Cipher.ENCRYPT\_MODE, key, new IvParameterSpec(iv));  
 } else {  
 cipher.init(Cipher.ENCRYPT\_MODE, key);  
 }  
  
 return cipher.doFinal(messageBytes);  
 }  
  
 @Benchmark

public byte[] decrypt() throws Exception {  
 Cipher cipher = Cipher.getInstance(transformation);  
 if (iv != null) {  
 cipher.init(Cipher.DECRYPT\_MODE, key, new IvParameterSpec(iv));  
 } else {  
 cipher.init(Cipher.DECRYPT\_MODE, key);  
 }  
  
 return cipher.doFinal(encryptedBytes);  
 }  
}

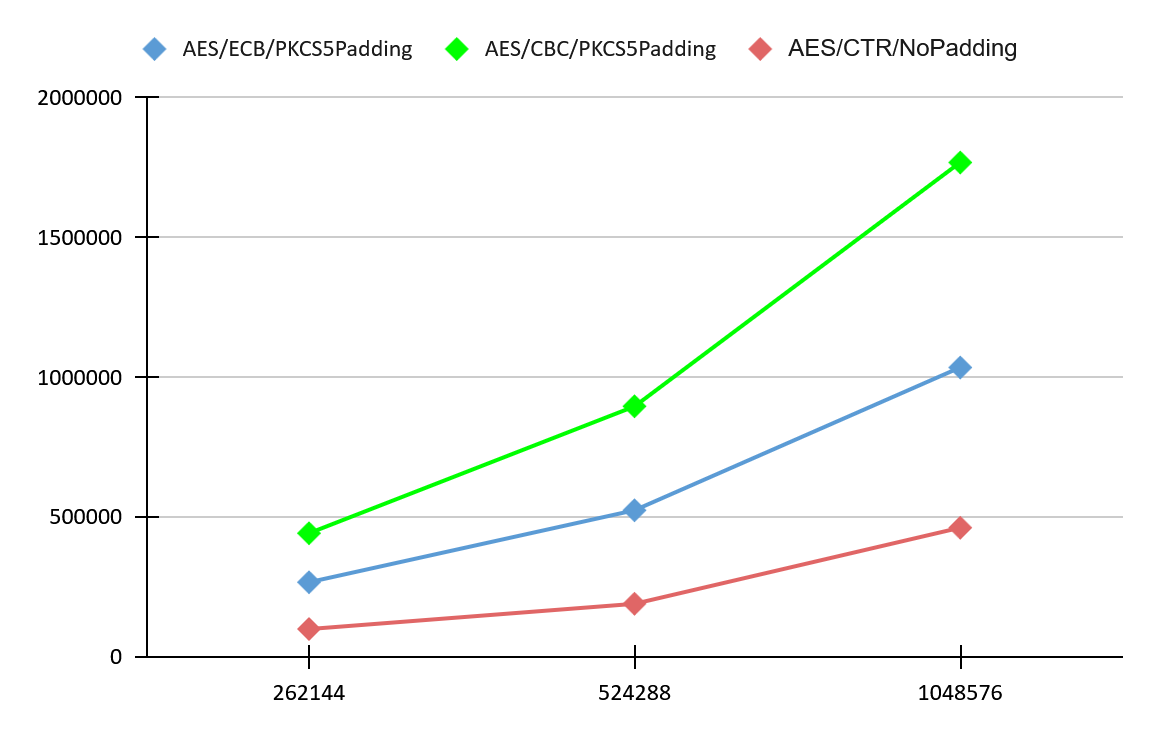
W funkcji wstępnej (Level.Trial) generowany jest ciąg losowych bajtów wiadomości, klucz do AES i w przypadku trybu CBC i CTR również parametr IV. Na końcu tworzona jest zaszyfrowana wiadomość w celach deszyfrowania. W celu pomiaru czasu ta zaszyfrowana wiadomość musi być w pierwszej funkcji. Następnie wywoływane są dwie funkcje do pomiaru.

Wszystkie tryby bloków szyfrowych są z rodziny AES. W przypadku CTR nie jest używany padding, ponieważ działa on na ciągu szyfrowania, a nie na bloku jak ECB i CBC.

Parametrami benchmarku są: rozmiar wiadomości (w bajtach) oraz tryb szyfrowania blokowego. Zbadano czasy dla rozmiarów wiadomości 256 kB, 512 kB oraz 1 MB oraz trzech trybów: ECB, CBC oraz CTR.

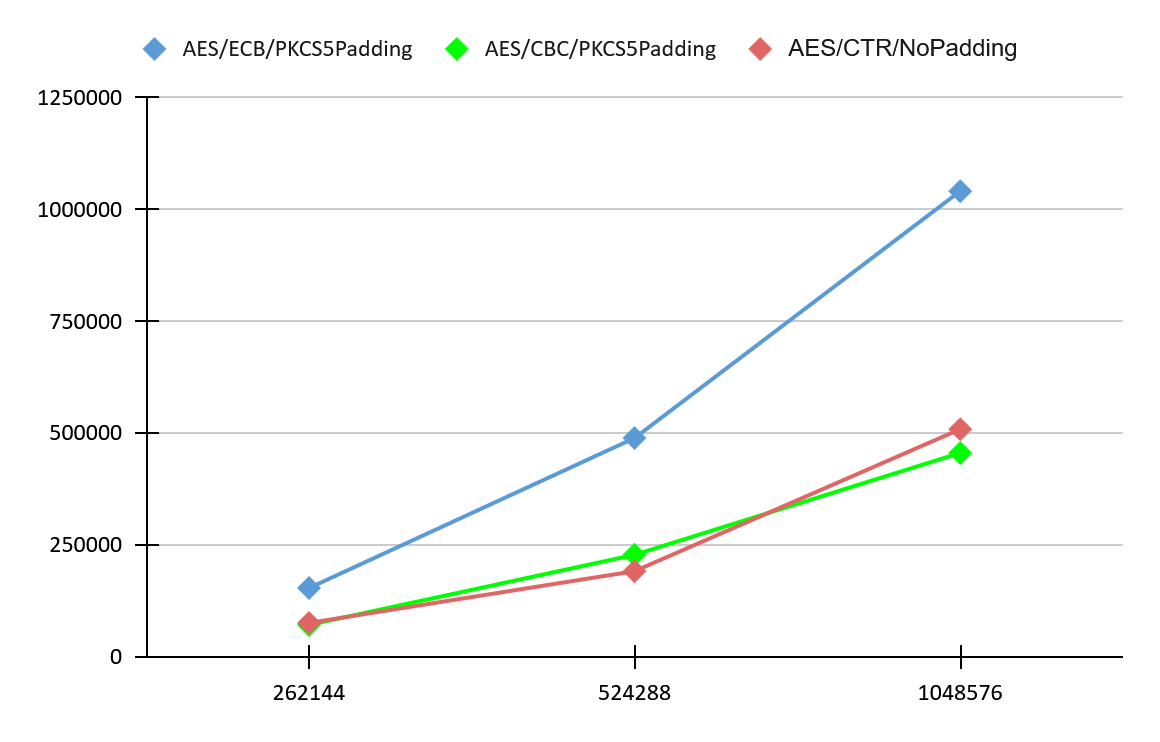
Poniżej przedstawione zostały wyniki pomiaru czasu wybranych trybów szyfrowania. Czas podany został w nanosekundach [1n = 10^-9]

## Szyfrowanie



Na powyższym wykresie można zauważyć, iż najszybszy okazuje się być tryb CTR. Najgorszy czasowo natomiast jest CBC. Jest to bardziej bezpieczna kryptograficznie wersja trybu ECB, która bazuje na feedbacku pomiędzy kolejnymi blokami. Na czas szyfrowania może wpływać możliwość zrównoleglenia działania algorytmów. W przypadku ECB jak i CTR każdy blok szyfrowany jest oddzielnie, natomiast CBC potrzebuje “feedbacku” poprzedniego bloku do zaszyfrowania.

## Deszyfrowanie



W przypadku deszyfrowania najwolniejszy okazuje się być tryb ECB. Ten czas jednakże jest bardzo podobny do czasu szyfrowania. Dzieje się tak ponieważ ECB deszyfruje w taki sam sposób jak szyfruje: przepuszczając blok przez funkcję szyfrującą. Taka sama sytuacja występuje również w przypadku trybu CTR. Natomiast CBC w przypadku deszyfrowania umożliwia zrównoleglenie obliczeń, ponieważ “feedbackiem” jest w tym wypadku poprzedni zaszyfrowany blok, nie potrzeba czekać na poprzednie bloki.

# Analiza propagacji błędu

## Implementacja

public class ErrorPropagation {

static final int SIZE = 1024;

static final String[] transformations = new String[]{

"AES/ECB/PKCS5Padding",

"AES/CBC/PKCS5Padding",

"AES/CTR/NoPadding"

};

public static void main(String[] args) throws Exception {

byte[] messageBytes = new byte[SIZE];

SecureRandom.getInstanceStrong().nextBytes(messageBytes);

System.out.println("Original message: " + Arrays.toString(messageBytes));

for(String transformation : transformations) {

System.out.println("==================================");

System.out.println("Transformation >>> " + transformation);

try {

testPropagation(transformation, messageBytes);

} catch (Exception e) {

System.out.println("Error during tests" + e.getMessage());

}

}

}

static private void testPropagation(String transformation, byte[] message) throws Exception {

KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");

keyGenerator.init(128);

SecretKey key = keyGenerator.generateKey();

byte[] iv = null;

if (transformation.contains("CBC") || transformation.contains("CTR")) {

iv = new byte[16];

SecureRandom.getInstanceStrong().nextBytes(iv);

}

// --------------- Corrupted message encryption

byte[] corruptedMessage = Arrays.copyOf(message, message.length);

corruptedMessage[0] ^= 0b00000001;

Cipher encryptCipherCorruption = Cipher.getInstance(transformation);

if (iv != null) {

encryptCipherCorruption.init(Cipher.ENCRYPT\_MODE, key, new IvParameterSpec(iv));

} else {

encryptCipherCorruption.init(Cipher.ENCRYPT\_MODE, key);

}

byte[] encryptedCorruptedMessage = encryptCipherCorruption.doFinal(corruptedMessage);

Cipher decryptCipher1 = Cipher.getInstance(transformation);

if (iv != null) {

decryptCipher1.init(Cipher.DECRYPT\_MODE, key, new IvParameterSpec(iv));

} else {

decryptCipher1.init(Cipher.DECRYPT\_MODE, key);

}

byte[] decryptedFromCorruptMessage = decryptCipher1.doFinal(encryptedCorruptedMessage);

System.out.println("Error on original message");

System.out.println("Found bit errors: " + countBitErrors(message, decryptedFromCorruptMessage));

printBitDifferences(message, decryptedFromCorruptMessage);

// ----------------- Correct message encryption

Cipher encryptCipher = Cipher.getInstance(transformation);

if (iv != null) {

encryptCipher.init(Cipher.ENCRYPT\_MODE, key, new IvParameterSpec(iv));

} else {

encryptCipher.init(Cipher.ENCRYPT\_MODE, key);

}

byte[] encryptedMessage = encryptCipher.doFinal(message);

// ------------------- Corruption on encryption

byte[] corruptedEncryption = Arrays.copyOf(encryptedMessage, encryptedMessage.length);

corruptedEncryption[0] ^= 0b00000001; // one bit change

Cipher decryptCipher2 = Cipher.getInstance(transformation);

if(transformation.contains("CBC") || transformation.contains("CTR")) {

decryptCipher2.init(Cipher.DECRYPT\_MODE, key, new IvParameterSpec(iv));

} else {

decryptCipher2.init(Cipher.DECRYPT\_MODE, key);

}

byte[] decryptedMessage = decryptCipher2.doFinal(corruptedEncryption);

System.out.println("Error on encrypted message");

System.out.println("Found bit errors: " + countBitErrors(message, decryptedMessage));

printBitDifferences(message, decryptedMessage);

}

static int countBitErrors(byte[] message1, byte[] message2) {

int errors = 0;

for(int i = 0; i < Math.min(message1.length, message2.length); i++) {

errors += Integer.bitCount(message1[i] ^ message2[i]);

}

return errors;

}

static void printBitDifferences(byte[] message1, byte[] message2) {

System.out.println("Byte comparison");

for (int i = 0; i < Math.min(message1.length, message2.length); i++) {

if (message1[i] != message2[i]) {

System.out.printf(" - Byte %d: %02X - %02X%n",

i, message1[i], message2[i]);

}

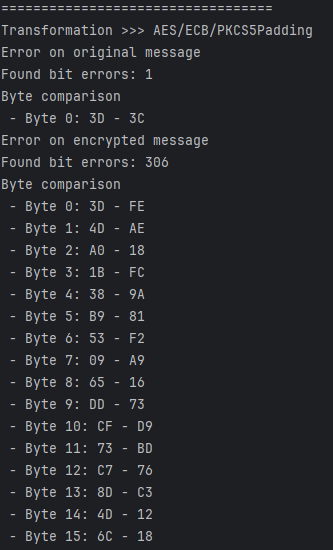
}

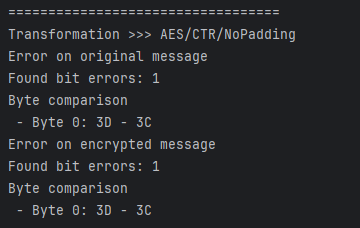
}

}

Dla każdego trybu przeprowadzany jest proces szyfrowania i deszyfrowania losowej wiadomości z błędem na jednym bicie w pierwszym bloku. Najpierw sprawdzany jest błąd w oryginalnej wiadomości, a potem błąd podczas enkrypcji.

## Wynik





W przypadku błędu na oryginalnej wiadomości uzyskana została z powrotem ta sama wiadomość z jednym błędem. Dla ECB cały blok zawierający błędny bit został zmieniony. W CBC jest zależność od poprzedniego bloku. Dlatego też błąd jest w pierwszym bloku (jak ECB) oraz na pierwszym bajcie następnego bloku. Ponieważ CTR jest strumieniowym szyfrem blokowym, zmianie ulegnie tylko bajt wiadomości, dla której bajt zaszyfrowany miał ten błąd.

# Implementacja CBC z wykorzystaniem ECB

public class CustomCBC {

static final int BLOCK\_SIZE = 16;

public static void main(String[] args) throws Exception{

byte[] message = "Test message that maybe needs padding".getBytes();

int padding = BLOCK\_SIZE - (message.length % BLOCK\_SIZE);

byte[] paddedMessage = Arrays.copyOf(message, message.length + padding);

for (int i = message.length; i < paddedMessage.length; i++) {

paddedMessage[i] = (byte) padding;

}

KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");

keyGenerator.init(128);

SecretKey key = keyGenerator.generateKey();

byte[] iv = new byte[BLOCK\_SIZE];

SecureRandom.getInstanceStrong().nextBytes(iv);

byte[] encryptedMessage = encrypt(paddedMessage, key, iv);

byte[] decryptedMessage = decrypt(encryptedMessage, key, iv);

byte[] result = Arrays.copyOf(decryptedMessage, message.length);

System.out.println("Original message: " + new String(message));

System.out.println("Decrypted message: " + new String(result));

}

static byte[] encrypt(byte[] message, SecretKey key, byte[] iv) throws Exception {

Cipher ecb = Cipher.getInstance("AES/ECB/NoPadding");

ecb.init(Cipher.ENCRYPT\_MODE, key);

byte[] encryptedMessage = new byte[message.length];

byte[] feedback = iv;

for (int i = 0; i < message.length; i += BLOCK\_SIZE) {

byte[] block = Arrays.copyOfRange(message, i, i + BLOCK\_SIZE);

byte[] xored = xor(block, feedback);

byte[] encrypted = ecb.doFinal(xored);

System.arraycopy(encrypted, 0, encryptedMessage, i, BLOCK\_SIZE);

feedback = encrypted;

}

return encryptedMessage;

}

static byte[] decrypt(byte[] encryptedMessage, SecretKey key, byte[] iv) throws Exception {

Cipher ecb = Cipher.getInstance("AES/ECB/NoPadding");

ecb.init(Cipher.DECRYPT\_MODE, key);

byte[] originalMessage = new byte[encryptedMessage.length];

byte[] feedback = iv;

for (int i = 0; i < encryptedMessage.length; i += BLOCK\_SIZE) {

byte[] block = Arrays.copyOfRange(encryptedMessage, i, i + BLOCK\_SIZE);

byte[] decrypted = ecb.doFinal(block);

byte[] xored = xor(decrypted, feedback);

System.arraycopy(xored, 0, originalMessage, i, BLOCK\_SIZE);

feedback = block;

}

return originalMessage;

}

static byte[] xor(byte[] block1, byte[] block2) {

byte[] result = new byte[block1.length];

for (int i = 0; i < block1.length; i++) {

result[i] = (byte)(block1[i] ^ block2[i]);

}

return result;

}

}

W tej implementacji jako wiadomość użyto ciągu znaków zamiast losowej tablicy bajtów. Na początku do tablica bajtów zostaje powiększona, aby zawierała równą liczbę bloków. Następnie wykonywane jest szyfrowanie oraz deszyfrowanie. Aby dokonać szyfrowania w trybie CBC z użyciem trybu ECB, należy przejść przez każdy blok, wykonać operację XOR pomiędzy nim a „feedbackiem”, a następnie zaszyfrować wynik tej operacji. Feedbackiem jest wektor inicjujący (IV) (jeśli jest to pierwszy blok) lub poprzedni zaszyfrowany blok (dla każdego następnego bloku).

Wynik:

