Kryptografia w Javie. Od podstaw

Autor: David Hook
Tłumaczenie: Zbigniew Banach
ISBN: 83-246-0277-1
Tytuł oryginału: Beginning Cryptography with Java
Format: B5, stron: 512
Przykłady na ftp: 600 kB

Stosuj algorytmy kryptograficzne w aplikacjach

- Poznaj architekturę interfejsów kryptograficznych Javy
- Zastosuj klucze symetryczne i asymetryczne
- Naucz się zarządzania certyfikatami w programach

W świecie, w którym najcenniejszym towarem jest informacja, kryptografia coraz bardziej zyskuje na znaczeniu. Cenne dane, przesyłane w sieci lub przechowywane w aplikacjach i bazach danych, muszą być chronione za pomocą skomplikowanych algorytmów szyfrowania i uwierzytelniania. Ponieważ próby włamań do serwerów internetowych zdarzają się regularnie, implementacja mechanizmów kryptograficznych w aplikacjach sieciowych i platformach handlu elektronicznego ma szczególnie wysoki priorytet. Java, wykorzystywana bardzo często do tworzenia takich właśnie rozwiązań, wyposażona została w zestaw interfejsów programistycznych (API), które pozwalają szybko i skutecznie wzbogacać aplikacje o obsługę kryptografii.

Książka „Kryptografia w Javie. Podstawy" to podręcznik przedstawiający na praktycznych przykładach narzędzia kryptograficzne Javy. Opisuje podstawowe zasady ich używania, ułatwia zrozumienie zależności między poszczególnymi interfejsami API i uczy, jak w razie potrzeby korzystać z gotowych rozwiązań, by oszczędzić czas. Daje wiedzę niezbędną do implementowania technik kryptograficznych w aplikacjach bez niepotrzebnego komplikowania kodu źródłowego.

- Architektura interfejsów JCA i JCE
- Szyfrowanie symetryczne
- Generowanie kluczy
- Stosowanie kluczy asymetrycznych
- Podpisy cyfrowe
- Obsługa certyfikatów
- Szyfrowanie poczty elektronicznej

Twórz bezpieczne aplikacje,
wykorzystując nowoczesne mechanizmy kryptograficzne

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

4

Kryptografia asymetryczna

Podstawowy problem związany z używaniem szyfrów, kodów MAC i kodów HMAC z kluczem symetrycznym polega na bezpiecznym dostarczeniu tajnego klucza do odbiorcy, by mógł on odszyfrować otrzymaną wiadomość. Jednego z rozwiązań tego problemu dostarcza kryptografia asymetryczna. Nazwa pochodzi stąd, że do szyfrowania i deszyfrowania używane są różne klucze, najczęściej nazywane kluczem publicznym i prywatnym.

W tym rozdziale poznamy podstawy kryptografii asymetrycznej, w tym możliwości wykorzystania kluczy publicznych i prywatnych do wymiany kluczy tajnych i do generowania podpisów cyfrowych. Istnieje wiele różnych rodzajów algorytmów asymetrycznych, więc przyjrzymy się również parametrom wymaganym do utworzenia klucza oraz odpowiednim mechanizmom dopełnienia podczas szyfrowania wiadomości. W przeciwieństwie do algorytmów symetrycznych algorytmy asymetryczne nie są najlepsze do szyfrowania dużych ilości danych, stąd też zajmiemy się metodami łączenia obu rodzajów szyfrowania, by ten cel osiągnąć.

Pod koniec tego rozdziału powinieneś:

- znać i rozumieć najpopularniejsze obecnie algorytmy,
- rozumieć metody wymiany i uzgadniania klucza, jak również najważniejsze ich wady i zalety,
- rozumieć zasady używania mechanizmów dopełnienia z algorytmami asymetrycznymi,
- umieć tworzyć i weryfikować podpisy cyfrowe.

Poznasz też w praktyce różne API Javy wspomagające generowanie, modyfikację i stosowanie kluczy i szyfrów asymetrycznych.

Klasa narzędziowa

W tym rozdziale zostanie wykorzystana wersja klasy `Utils` rozszerzona o implementację `SecureRandom` zwracającą przewidywalne wyniki. Z kryptograficznego punktu widzenia może się to wydać szaleństwem, ale bardzo ułatwi korzystanie z przykładów w tym rozdziale, gdyż będzie możliwe uzyskanie dokładnie takich wyników, jak w książce.

Oto kod klasy:

```
package rozdzial4;

import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;

/**
 * Klasa narzędziowa dla przykładów z rozdziału 4.
 */
public class Utils extends rozdzial3.Utils
{
   private static class FixedRand extends SecureRandom
   {
      MessageDigest sha;
      byte[] state;

      FixedRand()
      {
         try
         {
            this.sha = MessageDigest.getInstance("SHA-1");
            this.state = sha.digest();
         }
         catch (NoSuchAlgorithmException e)
         {
            throw new RuntimeException("nie znaleziono SHA-1!");
         }
      }

      public void nextBytes(byte[] bytes)
      {
         int off = 0;

         sha.update(state);

         while (off < bytes.length)
         {
            state = sha.digest();

            if (bytes.length - off > state.length)
            {
               System.arraycopy(state, 0, bytes, off, state.length);
            }
            else
            {
               System.arraycopy(state, 0, bytes, off, bytes.length - off);
            }
```

```
                off += state.length;

                sha.update(state);
            }
        }
    }

    /**
    * Zwraca obiekt SecureRandom generujący stałą wartość.
    * <b>Wyłącznie do celów testowych!</b>
    * @return stała wartość losowa
    */
    public static SecureRandom createFixedRandom()
    {
        return new FixedRand();
    }
}
```

Jak widać, wykorzystany tu pomysł opiera się na użyciu skrótu wiadomości do wygenerowania pseudolosowego strumienia bajtów. Przepisz tę wersję klasy `Utils`, skompiluj ją jako pierwszą klasę pakietu `rozdzial4` i możesz zaczynać.

Interfejsy PublicKey i PrivateKey

Omówienie kryptografii asymetrycznej w Javie nie obejdzie się bez dwóch interfejsów: `java.security.PublicKey` i `java.security.PrivateKey`. Każdy klucz wykorzystywany podczas szyfrowania asymetrycznego będzie implementował jeden z tych interfejsów, stąd też często pojawiają się one jako typy obiektów zwracanych przez wszelkiego rodzaju klasy operujące na kluczach i danych klucza.

Same interfejsy bezpośrednio rozszerzają interfejs `java.security.Key`, ale nie wprowadzają żadnych własnych metod — jedynym powodem ich istnienia jest zapewnienie bezpieczeństwa typów. Uzasadnienie takiego stanu rzeczy poznamy podczas szczegółowego omawiania poszczególnych algorytmów. Większość algorytmów ma własne zestawy interfejsów obsługi klucza, gdyż w przeciwieństwie do algorytmów klucza symetrycznego, które można stosować w zasadzie zamiennie, każdy algorytm asymetryczny różni się od pozostałych nie tylko sposobem działania, ale i parametrami potrzebnymi do utworzenia klucza.

Na początek zajmiemy się zwykłym szyfrowaniem danych, ale w dalszej części rozdziału zobaczymy, że kryptografia asymetryczna stanowi też podstawę mechanizmów uzgadniania klucza i podpisów cyfrowych.

Algorytm RSA

Algorytm RSA został opublikowany w 1977 roku przez Ronalda Rivesta, Adi Shamira i Leonarda Adlemana. Odtajnione w ostatnich latach dokumenty ujawniły, że metodę używaną w RSA odkrył jeszcze w 1973 roku Clifford Cocks z brytyjskiego GCHQ.

Podstawą bezpieczeństwa algorytmu jest trudność rozkładu dużych liczb na czynniki pierwsze (faktoryzacji). Mechanizm polega na tym, że klucze publiczny i prywatny są funkcjami pary dużych liczb pierwszych, a odtworzenie tekstu jawnego na podstawie znajomości szyfrogramu i klucza publicznego, którym został zaszyfrowany, jest uważane za zadanie o trudności porównywalnej z zadaniem ustalenia liczb pierwszych, na podstawie których wygenerowano klucze. Istotną kwestią jest wymagana długość klucza. Klucze w przykładach zostały tak dobrane, by było je łatwo wpisać, a wynik ich użycia mieścił się w jednym wierszu, więc nie mają one nic wspólnego z bezpieczeństwem. W praktyce klucz powinien mieć długość co najmniej 1024 bitów, a jeśli tworzona aplikacja ma chronić wiadomości przez więcej niż 10 lat, to długość tę należy podwoić.

Wyprowadzenie pełnego algorytmu znacznie wykracza poza ramy tej książki, ale samą ideę działania można wyrazić bardzo prosto. Niech p i q będą liczbami pierwszymi, a n, e i d będą liczbami takimi, że:

$$n = pq$$

oraz

$$ed \equiv 1 \bmod ((p - 1)(q - 1))$$

Dla danej wiadomości m mamy wtedy:

$$c = m^e \bmod n$$

$$m = c^d \bmod n$$

gdzie c jest szyfrogramem. Liczby n, e i d mają też dłuższe nazwy — są to odpowiednio *moduł*, *wykładnik publiczny* i *wykładnik prywatny*. Ogólną zasadą jest taki dobór wartości wykładnika publicznego, by etap szyfrowania był mało kosztowny obliczeniowo, po czym wykładnik prywatny jest generowany zgodnie z powyższym wzorem. Długość wygenerowanego klucza RSA jest określana długością n, więc każda z liczb p i q musi mieć długość połowy klucza.

Przed skorzystaniem z algorytmu należy się upewnić, że każdą szyfrowaną wiadomość da się zapisać w postaci dużej, dodatniej liczby całkowitej, nieprzekraczającej n. Na szczęście nie jest to trudne. W Javie wystarczy stworzyć dodatni obiekt `BigInteger` na podstawie bajtów składających się na wiadomość.

Cały czas korzystamy z JCE, więc nigdy nie będziemy musieli samodzielnie zamieniać wiadomości na obiekt `BigInteger`. Trzeba jednak mieć świadomość, że taka właśnie operacja odbywa się za kulisami, gdyż ma ona wpływ na bezpieczne stosowanie algorytmu RSA. Wspomniałem na przykład, że liczbowa reprezentacja wiadomości m musi być arytmetycznie mniejsza od liczby n. Powodem takiego wymagania jest to, że obliczenia w ramach algorytmu RSA odbywają się modulo n, czyli wymagają dzielenia przez n i przyjmowania reszty z tego dzielenia jako wyniku. Oznacza to, że każda wartość większa od n zostanie podczas szyfrowania zredukowana do mod n. Nieco później zobaczymy, jak można sobie z tym ograniczeniem poradzić i szyfrować długie wiadomości szyfrem RSA, ale na razie wystarczy pamiętać, że działanie algorytmu opiera się na arytmetyce dużych liczb całkowitych.

Spróbuj sam: Podstawowe RSA

Na początek spróbuj uruchomić poniższy przykład. Wprowadza on podstawowe klasy obsługujące algorytm RSA i pokazuje, że JCE pozwala nadal korzystać z jednolitego API, niezależnie od tego, że faktyczna implementacja RSA diametralnie różni się od wcześniej poznanych algorytmów. Długość klucza wynosi 128 bitów, czyli zaledwie jedną ósmą minimalnej wymaganej długości, ale dla potrzeb przykładu to w zupełności wystarczy.

```
package rozdzial4;

import java.math.BigInteger;
import java.security.KeyFactory;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;

import javax.crypto.Cipher;

/**
 * Podstawowy przykład RSA.
 */
public class BaseRSAExample
{
   public static void main(String[] args) throws Exception
   {
      byte[] input = new byte[] { (byte)0xbe, (byte)0xef };
      Cipher cipher = Cipher.getInstance("RSA/None/NoPadding", "BC");
      KeyFactory keyFactory = KeyFactory.getInstance("RSA", "BC");

      // tworzenie kluczy

      RSAPublicKeySpec pubKeySpec = new RSAPublicKeySpec(
         new BigInteger("d46f473a2d746537de2056ae3092c451", 16),
         new BigInteger("11", 16));
      RSAPrivateKeySpec privKeySpec = new RSAPrivateKeySpec(
         new BigInteger("d46f473a2d746537de2056ae3092c451", 16),
         new BigInteger("57791d5430d593164082036ad8b29fb1", 16));

      RSAPublicKey pubKey = (RSAPublicKey)keyFactory.generatePublic(pubKeySpec);
      RSAPrivateKey privKey =
      (RSAPrivateKey)keyFactory.generatePrivate(privKeySpec);

      System.out.println("dane wejściowe: " + Utils.toHex(input));

      // szyfrowanie

      cipher.init(Cipher.ENCRYPT_MODE, pubKey);

      byte[] cipherText = cipher.doFinal(input);

      System.out.println("dane zaszyfrowane: " + Utils.toHex(cipherText));

      // deszyfrowanie
      cipher.init(Cipher.DECRYPT_MODE, privKey);
      byte[] plainText = cipher.doFinal(cipherText);
```

```
        System.out.println("dane odszyfrowane: " + Utils.toHex(plainText));
    }
}
```

Uruchomienie przykładu powinno dać następujący wynik:

```
dane wejściowe: beef
dane zaszyfrowane: d2db15838f6c1c98702c5d54fe0add42
dane odszyfrowane: beef
```

Jak to działa

Sposób użycia klasy `Cipher` nie różni się tu specjalnie od przykładów z wcześniejszych rozdziałów. Różnica polega jedynie na tym, że klasy reprezentujące wartość klucza oraz interfejsy odpowiadające samym kluczom są bardziej wyspecjalizowane — już z samych ich nazw wynika, że są one przeznaczone dla RSA.

Poszczególne klasy opisujące klucz omówię szczegółowo nieco później, ale z kodu widać, że inicjalizacja szyfru RSA wymaga zdefiniowania obiektów wartości zawierających liczby odpowiadające kluczowi publicznemu i prywatnemu. Obiekty te są następnie przekształcane w klucze i przekazywane obiektowi szyfru za pośrednictwem klasy fabrykującej obsługującej klucze asymetryczne — klasy `KeyFactory`. Potem wszystko postępuje zgodnie ze znanymi już zasadami. Przyjrzyjmy się zatem bliżej klasie `KeyFactory`, pamiętając, że jest ona abstrakcją, więc wszystko, co powiemy na jej temat, odnosi się do wszystkich algorytmów asymetrycznych.

Klasa KeyFactory

Klasa `java.security.KeyFactory` stanowi warstwę abstrakcji pozwalającą zamieniać obiekty parametrów klucza utworzone poza środowiskiem danego dostawcy na obiekty `Key`, jak również konwertować już istniejące klucze na specyfikacje nadające się do eksportu. Podobnie jak klasa `Cipher`, `KeyFactory` używa metody fabrykującej `getInstance()`, która w kwestii lokalizacji i priorytetów dostawców zachowuje się dokładnie tak samo, jak analogiczne metody innych klas JCA.

Klasa `KeyFactory` udostępnia cztery metody konwersji. Metody `generatePublic()` i `generatePrivate()` konwertują specyfikacje klucza na obiekty reprezentujące klucz, `getKeySpec()` pozwala tworzyć i eksportować parametry klucza, a metoda `translateKey()` przekształca klucze jednego dostawcy do postaci nadającej się do użytku z innym dostawcą.

Z punktu widzenia poprzedniego przykładu najciekawsze są metody `generatePublic()` i `generatePrivate()`. Jak widać z kodu, służą one do zamiany obiektów zawierających parametry kluczy RSA na faktyczne obiekty klucza, wypada zatem bliżej przyjrzeć się klasom parametrów kluczy i interfejsom do obsługi kluczy.

Klasa RSAPublicKeySpec i interfejs RSAPublicKey

Jak sugeruje końcówka nazwy, klasa `RSAPublicKeySpec` pozwala tworzyć obiekty wartości zawierające parametry niezbędne do wygenerowania klucza publicznego RSA, czyli moduł i wykładnik publiczny.

Przekazując obiekt `RSAPublicKeySpec` metodzie `KeyFactory.generatePublic()`, można uzyskać obiekt implementujący `PublicKey`. Ponieważ chodzi o szyfr RSA, klasa `KeyFactory` zwróci w tym przypadku obiekt implementujący `RSAPublicKey` o takich samych sygnaturach metod, co klasa `RSAPublicKeySpec`. Istnieje też metoda `getModulus()` zwracająca obiekt `BigInteger` odpowiadający modułowi oraz metoda `getPublicExponent()` zwracająca analogiczny obiekt dla wykładnika publicznego.

Klasa RSAPrivateKeySpec i interfejs RSAPrivateKey

Klasa `RSAPrivateKeySpec` pozwala tworzyć obiekty wartości zawierające parametry niezbędne do wygenerowania klucza prywatnego RSA, czyli moduł i wykładnik prywatny.

Działanie klasy opisującej klucz oraz interfejsu `RSAPrivateKey` stanowi odzwierciedlenie działania ich odpowiedników do tworzenia obiektów klucza publicznego, z tą tylko różnicą, że w miejsce metody `getPublicExponent()` wywoływana jest metoda `getPrivateExponent()`. Symetria klas, metod i interfejsów nie powinna tu dziwić, gdyż wynika bezpośrednio z symetrii samego algorytmu RSA.

Tworzenie losowych kluczy RSA

Zamiast podawać gotowe dane dla klucza, można go też wygenerować losowo. Z opisu działania algorytmów asymetrycznych wiemy, że potrzebne są dwa klucze: publiczny i prywatny. Oznacza to, że zamiast klasy `KeyGenerator` generującej pojedynczy obiekt `Key` trzeba będzie skorzystać z klasy `KeyPairGenerator`, generującej parę kluczy w postaci obiektu `KeyPair`.

Szczegółowym omówieniem tych klas zajmiemy się już niebawem, ale najlepiej zapoznać się z nimi na praktycznym przykładzie.

Spróbuj sam: Generowanie losowych kluczy RSA

Spróbuj uruchomić poniższy przykład. Generuje on parę kluczy RSA na podstawie losowych danych, szyfruje prostą wiadomość pobranym z tej pary kluczem publicznym, a następnie deszyfruje szyfrogram pobranym z pary kluczem prywatnym.

```
package rozdzial4;

import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
```

```
import javax.crypto.Cipher;

/**
 * Przykład RSA z losowym generowaniem klucza.
 */
public class RandomKeyRSAExample
{
   public static void main(String[] args) throws Exception
   {
      byte[] input = new byte[] { (byte)0xbe, (byte)0xef };
      Cipher cipher = Cipher.getInstance("RSA/NONE/NoPadding", "BC");
      SecureRandom random = Utils.createFixedRandom();

      // generowanie kluczy
      KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA", "BC");

      generator.initialize(256, random);

      KeyPair pair = generator.generateKeyPair();
      Key pubKey = pair.getPublic();
      Key privKey = pair.getPrivate();

      System.out.println("dane wejściowe: " + Utils.toHex(input));

      // szyfrowanie

      cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);

      byte[] cipherText = cipher.doFinal(input);

      System.out.println("dane zaszyfrowane: " + Utils.toHex(cipherText));

      // deszyfrowanie

      cipher.init(Cipher.DECRYPT_MODE, privKey);

      byte[] plainText = cipher.doFinal(cipherText);

      System.out.println("dane odszyfrowane: " + Utils.toHex(plainText));
   }
}
```

Uruchomienie przykładu daje następujący wynik:

```
dane wejściowe: beef
dane zaszyfrowane: 8274caf4a1f54b3b58f6798755d2cfce3e33f710a3f520865c0ccdca0a672601
dane odszyfrowane: beef
```

Jak widać, długość szyfrogramu rośnie wraz z długością klucza. Choć w tym przykładzie wykorzystany został klucz o długości zaledwie 256 bitów (czyli jednej czwartej długości klucza dla aplikacji produkcyjnej), to i tak wyraźnie widać, że zastosowanie RSA spowodowało znaczne wydłużenie pierwotnej 4-bajtowej wiadomości. W przypadku RSA nie ma żadnego odpowiednika trybu CRT, więc trzeba się pogodzić z tym efektem jako ceną tej metody szyfrowania.

Jak to działa

Podobnie jak we wcześniejszym programie `BaseRSAExample`, kod wygląda tu bardzo podobnie do kodów z rozdziału 2. Jedyną znaczącą różnicą jest wprowadzenie obiektów `KeyPair` i `KeyPairGenerator`, gdyż mamy tym razem do czynienia z szyfrem asymetrycznym. Do zrozumienia przykładu wystarczy zrozumienie działania klas biorących udział w generowaniu klucza publicznego i prywatnego.

Klasa KeyPair

Klasa `java.security.KeyPair` stanowi pojemnik dla klucza prywatnego i odpowiadającego mu klucza publicznego. Obiekt `KeyPair` jest prostym obiektem wartości udostępniającym metody `getPrivate()` i `getPublic()`, zwracające odpowiednio klucz prywatny i klucz publiczny.

Klasa KeyPairGenerator

Instancje klasy `java.security.KeyPairGenerator` są pobierane za pomocą typowej metody fabrykującej `getInstance()`, której zachowanie w kwestii wyboru i priorytetu dostawcy usług jest identyczne, jak w przypadku metod `getInstance()` innych poznanych już klas JCA.

Obsługa samej klasy jest bardzo prosta. Dostępne są cztery wersje metody `initialize()`: dwie przyjmujące liczbę całkowitą określającą długość klucza (z czego jedna wymaga dodatkowo przekazania źródła danych losowych) i dwie przyjmujące pełniący tę samą funkcję obiekt `AlgorithmParameterSpec`. Jak się okaże w dalszych przykładach, najwięcej możliwości daje wykorzystanie wersji przyjmujących obiekty `AlgorithmParameterSpec`, gdyż w przypadku algorytmów asymetrycznych długość klucza jest tylko jednym z wielu możliwych parametrów, których wartości przydaje się kontrolować.

Pozostają już tylko metody zwracające generowany przez klasę obiekt pary kluczy: `KeyPairGenerator.generateKeyPair()` i `KeyPairGenerator.genKeyPair()`. Metody te działają identycznie, a różnią się wyłącznie nazwą, więc wybór jednej z nich zależy wyłącznie od uznania programisty.

Istnieje też właściwa algorytmowi RSA klasa implementująca `AlgorithmParameterSpec` — jest to `RSAKeyGenParameterSpec`. Klasa ta jest bardzo prosta, więc przyjrzyjmy jej się od razu.

Klasa RSAKeyGenParameterSpec

Jak już wspomniałem, generowanie par kluczy RSA wiąże się najczęściej z wybraniem wykładnika publicznego i wygenerowaniem dla niego odpowiedniego wykładnika prywatnego. JCA pozwala określić zarówno pożądaną długość klucza, jak i wykładnik publiczny dla pary kluczy generowanej za pomocą obiektu `KeyPairGenerator` — wystarczy przekazać metodzie `KeyPairGenerator.initialize()` odpowiedni parametr w postaci obiektu implementującego `AlgorithmParameterSpec`. Powinien to być obiekt klasy `java.security.spec.RSAKeyGenParameterSpec`, która pojawiła się w JDK 1.3. Stworzenie takiego obiektu jest bardzo proste i wymaga jedynie podania w konstruktorze wykładnika publicznego i długości klucza.

Biorąc konkretny przykład, wykorzystanie jednego ze standardowych wykładników publicznych w postaci zalecanej w standardzie X.509 wartości F4 (w przypadku dostawcy Bouncy Castle jest to wartość domyślna) wymagałoby zamiany wywołania metody `initialize()` obiektu `generator` na:

```
generator.initialize(
   new RSAKeyGenParameterSpec(256, RSAKeyGenParameterSpec.F4), random);
```

Po takiej inicjalizacji wszystkie klucze publiczne RSA generowane przez obiekt `generator` będą miały wykładnik publiczny o wartości odpowiadającej stałej F4, czyli liczbie całkowitej 0x10001.

Przyspieszanie pracy RSA

Wystarczy rzut oka na pierwszy przykład algorytmu RSA, by stwierdzić, że wykładnik prywatny klucza prywatnego jest znacznie większy od wykładnika publicznego. Z tego też względu korzystanie z klucza prywatnego RSA jest znacznie wolniejsze od korzystania z klucza publicznego. Zachowanie to można łatwo uzasadnić — dzięki niemu szyfrowanie danych jest szybkie i można je wykonywać na klientach o ograniczonej mocy obliczeniowej. Jak się wkrótce przekonamy, dodatkową zaletą jest szybkie sprawdzanie podpisów cyfrowych.

Czy da się przyspieszyć proces deszyfrowania kluczem prywatnym? Okazuje się, że tak. Wymaga to wprawdzie wykorzystania znajomości liczb użytych do stworzenia modułu, a te powinny raczej być utrzymywane w tajemnicy, ale skoro wszelkie informacje o wykładniku prywatnym i tak powinny być tajne, to dołożenie jeszcze kilku tajnych danych nie powinno być wielkim problemem. Znajomość wykorzystanych liczb pierwszych jest tu konieczna, gdyż pozwala wykorzystać w praktyce *chińskie twierdzenie o resztach* (ang. *Chinese Remainder Theorem*, CRT).

Chińskie twierdzenie o resztach

Odkryte w pierwszym wieku naszej ery przez chińskiego matematyka Sun Tsu twierdzenie o resztach w uproszczeniu głosi, że:

„Dla liczby n o rozkładzie na liczby pierwsze $p1 * p2 * \ldots * pi$ układ kongruencji postaci:

$$(x \bmod p_j) \equiv a_j, j = 1, 2, \ldots, i$$

ma dokładnie jedno rozwiązanie modulo n".

Ujmując to samo nieco inaczej, dowolną liczbę mniejszą od n można jednoznacznie wyrazić jako ciąg reszt z dzielenia jej przez kolejne czynniki pierwsze n.

Nie będę się zagłębiał w matematyczny dowód prawdziwości tego twierdzenia, gdyż jest on podany w wielu innych publikacjach, ale najważniejszym następstwem praktycznym jest to, że obliczenia dotyczące wykładnika prywatnego można wykonywać inaczej niż obliczenia dla wykładnika publicznego. Obliczenia takie odbywają się na liczbach całkowitych znacznie mniejszych od modułu, dzięki czemu są dużo szybsze. Z tego właśnie względu przedstawione poniżej klasy `RSAPrivateCrtKey` i `RSAPrivateCrtKeySpec` mają metody do pobierania nie dwóch, lecz ośmiu wartości.

Klasy RSAPrivateCrtKeySpec i RSAPrivateCrtKey

Klasa `java.security.spec.RSAPrivateCrtKeySpec` dostarcza obiektów wartości zawierających dane klucza dla klucza prywatnego RSA, które można wykorzystać zgodnie z CRT.

Obiekt `RSAPrivateCrtKeySpec` można przekazać obiektowi `KeyFactory` dla RSA w dokładnie taki sam sposób, jak opisany wcześniej obiekt `RSAPrivateKeySpec`. Klasa fabrykująca powinna zwrócić obiekt klucza implementujący interfejs `java.security.interfaces.RSAPrivateCrtKey`, który z kolei rozszerza `RSAPrivateKey`. Jeśli mimo przekazania `RSAPrivateCrtKeySpec` zwracany jest klucz implementujący jedynie sam `RSAPrivateKey`, to używany dostawca prawdopodobnie nie implementuje CRT. To samo może być powodem powolnego wykonywania operacji RSA z kluczem prywatnym.

`RSAPrivateCrtKeySpec` i `RSAPrivateCrtKey` mają identyczne metody. Dostępne metody pozwalają pobierać poszczególne wartości składające się na klucz CRT:

- `getModulus()` zwraca wartość modułu *n*,
- `getPrivateExponent()` zwraca wartość prywatnego wykładnika *d*,
- `getPrimeP()` zwraca wartość liczby pierwszej *p*,
- `getPrimeQ()` zwraca wartość liczby pierwszej *q*.

Poza tym dostępne są jeszcze metody `getPrimeExponentP()`, `getPrimeExponentQ()` i `getCrtCoefficient()` zwracające wstępnie wyliczone wartości bazujące na *p* i *q*, których wykorzystanie pozwala dodatkowo przyspieszyć wykonywanie obliczeń z kluczem prywatnym RSA.

W przypadku dostawcy Bouncy Castle obiekty `RSAPrivateCrtKey` są zwracane przez implementację klasy `KeyPairGenerator`, ale można się samodzielnie przekonać o znacznym koszcie operacji na kluczu publicznym bez wykorzystania CRT, tworząc własną implementację klasy `RSAPrivateKeySpec`, pobierającą potrzebne wartości z obiektu `RSAPrivateCrtKey`.

Chińskie twierdzenie o resztach z wieloma liczbami pierwszymi

Z podanego wcześniej opisu CRT wynika, że twierdzenie opiera się na swobodnym dostępie do liczb pierwszych stanowiących czynniki modułu. Co ciekawe, ani w CRT, ani w algorytmie RSA nigdzie nie jest powiedziane, że takie liczby muszą być akurat dwie — można równie dobrze wziąć na przykład cztery. Gdyby więc pracować z kluczem 2048-bitowym, to zamiast wykonywać obliczenia dla generowania i używania klucza na dwóch liczbach 1024-bitowych, można by te same operacje wykonywać na czterech liczbach 512-bitowych. Niezależnie od dodatkowych trudności wynikających z obecności więcej niż dwóch czynników w procesie generowania modułu operacje na kluczu prywatnym w wersji z czterema liczbami pierwszymi będą znacznie szybsze od tych samych operacji dla wersji z dwiema liczbami.

Tę metodę obliczeń obsługuje interfejs `java.security.interfaces.RSAMultiPrimePrivateCrtKey` oraz klasy `java.security.spec.RSAMultiPrimePrivateCrtKeySpec` i `java.security.➥spec.RSAOtherPrimeInfo`. W tej książce nie będę tej techniki omawiał, gdyż obecnie nie ma jej implementacji w dostawcy Bouncy Castle, a sam algorytm jest opatentowany. Jeśli

jednak chcesz się dowiedzieć więcej, to polecam zapoznanie się z dokumentacją i dotyczącym między innymi tego zagadnienia standardem PKCS #1 firmy RSA Security. Zawarte tam informacje stanowią naturalne rozwinięcie przedstawionego wyżej opisu CRT.

W ten sposób kończymy omawianie zasad działania samego algorytmu RSA. Pora zająć się mechanizmami służącymi do przekształcenia bajtów szyfrowanej wiadomości w dużą liczbę całkowitą, którą można następnie wprowadzić na wejście tego algorytmu.

Mechanizmy dopełniania RSA

Największą różnicę między RSA a zwykłym szyfrem symetrycznym można zobaczyć, zastępując w ostatnim przykładzie wiersz:

```
byte[] input = new byte[] { (byte)0xbe, (byte)0xef };
```

wierszem:

```
byte[] input = new byte[] { 0x00, (byte)0xbe, (byte)0xef };
```

Po wprowadzeniu tej zmiany wynik działania programu wygląda tak:

```
dane wejściowe: 00beef
dane zaszyfrowane: 8274caf4a1f54b3b58f6798755d2cfce3e33f710a3f520865c0ccdca0a672601
dane odszyfrowane: beef
```

Zwracają tu uwagę dwie rzeczy. Najbardziej oczywiste jest to, że dane odszyfrowane różnią się od danych wejściowych — zniknęło początkowe zero. Nietrudno też zauważyć, że pomimo innych danych wejściowych szyfrogram jest ten sam, co poprzednio. W pewnym sensie może to tłumaczyć brak początkowego zera w tekście odszyfrowanym, ale nadal nie wiadomo, dlaczego zero zostało usunięte przed szyfrowaniem. Czyżby błąd algorytmu RSA, a przynajmniej jego implementacji?

Na szczęście zarówno algorytm, jak i jego implementacja działają bez zarzutu. W części poświęconej opisowi działania algorytmu RSA dowiedzieliśmy się, że przetworzenie tablicy bajtów przekazywanej jako dane wejściowe algorytmu wymaga jej zamiany na dużą liczbę całkowitą. Początkowe zera giną właśnie na etapie konwersji —w świecie liczb początkowe zera nie mają znaczenia.

Oczywiście w naszym przypadku zera początkowe mają znaczenie, więc przydałby się mechanizm dopełnienia, który pozwoliłby je zachować. Dopełnienie ma też znacznie ważniejsze zadanie od zachowywania początkowych zer. Spróbuj zmienić przykład tak, by wykładnikiem publicznym był F0 (wartość 0x3), a nie F4 (wartość 0x100001). W tym celu wystarczy zaimportować klasę `java.security.spec.RSAKeyGenParameterSpec` i zmienić treść wywołania `generator.initialize()` w następujący sposób:

```
generator.initialize(
   new RSAKeyGenParameterSpec(256, RSAKeyGenParameterSpec.F0), random);
```

Wprowadzenie tej zmiany i ponowne uruchomienie programu powinno dać taki wynik:

```
dane wejściowe: 00beef
dane zaszyfrowane: 0000000000000000000000000000000000000000000000000000000006a35ddd3c9cf
dane odszyfrowane: beef
```

Stracić wiodące zero to jedno, ale tutaj stało się coś znacznie gorszego — oryginalną wiadomość można teraz odtworzyć, po prostu obliczając pierwiastek sześcienny szyfrogramu. Dzieje się tak dlatego, że po konwersji na liczbę całkowitą i podniesieniu do potęgi wykładnika publicznego dane wejściowe są mniejsze od modułu. Oznacza to, że etap dzielenia modulo wcale nie utrudnił życia napastnikowi usiłującemu odczytać oryginalną wiadomość, gdyż zwrócił po prostu pierwotną wartość podniesioną do potęgi. Pokazuje to dobitnie, że dopełnienie wiadomości w celu uzyskania przed potęgowaniem liczby bliższej długością modułowi służy nie tylko zachowaniu wiodących zer.

Przy okazji metod wykorzystania chińskiego twierdzenia o resztach wspomniałem dokument PKCS #1. Tak się składa, że ta sama publikacja opisuje też mechanizmy dopełnienia dla algorytmu RSA. Pierwotnie był to tylko jeden mechanizm, stąd też nieco dziwna konwencja nazewnictwa omawianych dalej metod, ale obecnie standard przewiduje też inne mechanizmy dopełnienia. W Javie przyjęło się nazywać oryginalny mechanizm zdefiniowany w PKCS #1 V1.5 jako `PKCS1Padding`, podczas gdy mechanizmy dodane później noszą nazwy nadane im w PKCS #1. Na początek zajmijmy się mechanizmem oryginalnym.

Dopełnienie PKCS #1 V1.5

Oryginalna wersja PKCS #1 opisywała prosty mechanizm z trzema trybami dopełnienia bloku szyfrowanego. Pierwszym jest typ 0, polegający na dopełnieniu samymi zerami i stanowiący odpowiednik trybu `NoPadding` z JCE. Drugi tryb (typ 1) jest używany podczas szyfrowania danych algorytmem RSA z kluczem publicznym, a tryb trzeci (typ 2) jest stosowany przy deszyfrowaniu danych kluczem prywatnym. Ostatnie dwie techniki noszą w JCE właśnie nazwę `PKCS1Padding`.

Dopełnienie PKCS #1 typu 1 jest bardzo proste. Dla danej wiadomości M odpowiadająca jej wiadomość dopełniona M_p to:

$$M_p = 0x00 \,||\, 0x01 \,||\, F \,||\, 0x00 \,||\, M$$

gdzie F jest ciągiem bajtów 0xFF, a || operatorem sklejenia. Dodatkowo istnieje ograniczenie długości M: F musi mieć co najmniej 8 bajtów, więc M nie może być dłuższe od długości klucza w bajtach pomniejszonej o 11 bajtów.

Dopełnianie PKCS #1 typu 2 jest równie proste. Dla danej wiadomości M odpowiadająca jej wiadomość dopełniona M_p to:

$$M_p = 0x00 \,||\, 0x02 \,||\, R \,||\, 0x00 \,||\, M$$

gdzie R jest ciągiem co najmniej 8 bajtów pseudolosowych.

Różnica między tymi mechanizmami jest dość ciekawa. Oba generują duże liczby całkowite o długości zbliżonej do długości klucza (różnice długości są rzędu jednego bajta), ale typ 1 gwarantuje, że ta sama wartość M zaszyfrowana tym samym kluczem zawsze da ten sam szyfrogram, podczas gdy typ 2 gwarantuje zachowanie przeciwne. Innymi słowy, jest bardzo mało prawdopodobne, że ta sama wiadomość z dopełnieniem typu 2 dwukrotnie da ten sam szyfrogram. Z tego też względu (jak się przekonamy w kolejnych przykładach) typ 1 jest stosowany w podpisach, gdyż te są generowane z wykorzystaniem klucza prywatnego

— w końcu ten sam dokument podpisany tym samym kluczem powinien zawsze dawać ten sam podpis. Dopełnianie typu 2 jest z kolei używane przez obiekt `Cipher` stworzony w trybie `PKCS1Padding` na etapie szyfrowania kluczem publicznym.

Spróbuj sam: Dopełnienie PKCS #1 V1.5

Spróbuj uruchomić poniższy przykład. Szyfrowane dane zaczynają się teraz od zera, a do szyfrowania podany został tryb dopełnienia `PKCS1Padding`. W tym przypadku szyfrowanie odbywa się z wykorzystaniem klucza publicznego, więc stosowany jest typ 2 dopełnienia, co na ogół oznacza użycie danych losowych. Aby zapewnić przewidywalność wyników przykładu, podałem jako źródło danych losowych stały generator liczb losowych z klasy narzędziowej.

```
package rozdzial4;

import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;

import javax.crypto.Cipher;

/**
 * Przykład RSA z dopełnieniem PKCS #1.
 */
public class PKCS1PaddedRSAExample
{
   public static void main(String[] args) throws Exception
   {
      byte[] input = new byte[] { 0x00, (byte)0xbe, (byte)0xef };
      Cipher cipher = Cipher.getInstance("RSA/NONE/PKCS1Padding", "BC");
      SecureRandom random = Utils.createFixedRandom();

      // tworzenie kluczy
      KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA", "BC");

      generator.initialize(256, random);

      KeyPair pair = generator.generateKeyPair();
      Key pubKey = pair.getPublic();
      Key privKey = pair.getPrivate();

      System.out.println("dane wejściowe: " + Utils.toHex(input));

      // szyfrowanie

      cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);

      byte[] cipherText = cipher.doFinal(input);

      System.out.println("dane zaszyfrowane: " + Utils.toHex(cipherText));

      // deszyfrowanie
```

```
        cipher.init(Cipher.DECRYPT_MODE, privKey);

        byte[] plainText = cipher.doFinal(cipherText);

        System.out.println("dane odszyfrowane: " + Utils.toHex(plainText));
    }
}
```

Powinno się okazać, że wprowadzenie dopełnienia pozwoliło zachować wiodące zera w szyfrowanym tekście:

```
dane wejściowe: 00beef
dane zaszyfrowane: 01fce4a90b326bb1c3ebc2f969a84024d157499038f73ee03635c4e6ffb3377e
dane odszyfrowane: 00beef
```

Spróbuj też wprowadzić opisaną wcześniej zmianę wykładnika publicznego na wartość F0. Jak widać, dopełnienie nie tylko zachowuje treść oryginalnej wiadomości, lecz również pomaga ją chronić.

Jak to działa

Różnica polega oczywiście na tym, że wiodące zero jest zachowywane, gdyż oryginalna wiadomość jest najpierw dopełniona, a dopiero wynikowy ciąg oktetów jest zamieniany na liczbę całkowitą. W poprzedniej wersji (bez dopełnienia) konwersji podlegała sama wiadomość.

Innym skutkiem dopełnienia jest to, że obliczenia odbywają się na liczbie całkowitej o długości znacznie bliższej długości modułu, dzięki czemu arytmetyka modulo może sensownie działać i stosowanie RSA ma sens, nawet w przypadku tak małych wartości wykładnika publicznego, jak F0 (czyli wartość 3).

Dopełnienie OAEP

Optymalne dopełnianie dla szyfrowania asymetrycznego, czyli OAEP (ang. *Optimal Asymmetric Encryption Padding*), jest najmłodszą z metod dopełnienia określonych w PKCS #1. Metodę tę pierwotnie opracowali Mihir Bellare i Phillip Rogaway w 1994 roku, a obecnie jest ona oficjalnie jedną z metod PKCS #1 o pełnej nazwie RSAES-OAEP. Zaletą OAEP jest możliwość udowodnienia jej bezpieczeństwa w ramach modelu losowej wyroczni (ang. *random oracle model*), co oznacza, że jeśli funkcje skrótu wykorzystane w OAEP są idealne, to jedynym sposobem złamania wiadomości zaszyfrowanej RSA z kodowaniem OAEP jest złamanie samego RSA. Zanim przyjrzymy się faktycznej postaci wiadomości zakodowanych metodą OAEP, warto dokładniej przeanalizować znaczenie tego ostatniego stwierdzenia.

W tym kontekście wyrocznia jest swego rodzaju czarną skrzynką, z której można uzyskać pewne informacje — w tym przypadku deszyfruje ona wiadomości zaszyfrowane z dopełnianiem PKCS #1. Załóżmy, że chcemy poznać odszyfrowaną treść pewnej wiadomości i możemy nakłonić wyrocznię, by zwracała wyniki deszyfrowania przesyłanych jej wiadomości. W takiej sytuacji możliwy jest atak z milionem wiadomości (szczegółowo przedstawiony w dokumencie RFC 3218, a oryginalnie opisany w artykule Daniela Bleichenbachera

„Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1"), polegający na wygenerowaniu dużej liczby przekształceń odczytywanego szyfrogramu, a następnie wykorzystaniu informacji zwracanych przez wyrocznię otrzymującą takie wiadomości do stopniowego odtworzenia oryginalnego tekstu kryjącego się za szyfrogramem.

Jest to możliwe, gdyż niektóre cechy struktury wiadomości PKCS #1 z dopełnianiem typu 2 można przewidzieć. Przewaga dopełniania OAEP polega na tym, że wynikowa wiadomość jest nierozróżnialna od losowego ciągu bajtów. Co więcej, poprawnie odszyfrowana wiadomość będzie wewnętrznie spójna, ale dzięki zastosowaniu funkcji maskującej bazującej na kryptograficznej funkcji skrótu spójności tej nie da się sprawdzić, dopóki całość nie zostanie XOR-owana z odpowiednim strumieniem bajtów. Strumień jest generowany przez funkcję, której działania nie można przewidzieć bez znajomości poprawnych danych. Efekt połączenia funkcji skrótu i generowania maski jest tu podobny, jak w przypadku zastosowania funkcji skrótu przy uzgadnianiu klucza: pozbawienie atakującego możliwości wykorzystania matematycznych własności algorytmu bazowego. Nawet w przypadku dostępu do wyroczni odszyfrowane dane będą zawsze równie „losowe".

> Więcej informacji na temat modelu losowej wyroczni można znaleźć w artykule Bellare'a i Rogawaya „Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols". Jakość metody losowej wyroczni jako sposobu dowodzenia bezpieczeństwa pozostaje wprawdzie kwestią kontrowersyjną, ale wypada przyznać, że jest to krok we właściwym kierunku.

Pomimo zasadności powyższego wywodu i zaleceń odnośnie do stosowania w aplikacjach dopełnienia OAEP, gdy tylko jest to możliwe, cała ta historia niekoniecznie musi świadczyć o nieadekwatności dopełnienia PKCS #1. Prawdziwy morał jest taki, że odrzucając wiadomość, należy unikać ujawniania czytelnych informacji. Obsłużenie odrzucenia wiadomości powinno trwać dokładnie tyle samo, co jej przyjęcie, a ilość zwracanych informacji należy ograniczyć do minimum — najlepiej do zera (jeśli tylko jest to możliwe). Po prostu unikaj wycieków!

Pora przyjrzeć się samej metodzie OAEP. Jak już wspomniałem, korzysta ona z jednokierunkowej funkcji skrótu, dalej oznaczanej `H()`, i opartej na niej funkcji generowania maski `Mask()`. Nieco upraszczając, cały proces składa się z następujących trzech etapów przetwarzania wiadomości M, ciągu parametryzującego P i losowego ziarna S:

1. $M_1 = \text{Mask}((\text{H}(P) \parallel PS \parallel 0\text{x}01 \parallel M), S)$
2. $M_2 = \text{Mask}(S, M_1)$
3. $M_p = 0\text{x}00 \parallel M_2 \parallel M_1$

gdzie PS jest dopełniającym ciągiem zer, $\parallel$ jest operatorem sklejania, a M_p jest ostateczną wiadomością po dopełnieniu i zamaskowaniu. Działanie funkcji maski polega na wykorzystaniu drugiego argumentu jako ziarna dla generatora pseudolosowych oktetów, którego wynik służy następnie do zamaskowania oktetów podanych w ramach pierwszego argumentu. Funkcji generującej maskę nie będę tu szczegółowo omawiał, gdyż jest ona dokładnie opisana w dokumencie PKCS #1, więc podam jedynie jej nazwę — MGF1. Funkcja ta pojawi się jeszcze w kilku innych miejscach. Ciąg parametryzujący P jest domyślnie pusty, więc na ogół nie trzeba go podawać (co zobaczymy na przykładach).

Rzut oka na równania pokazuje, że ostateczna zamaskowana i dopełniona wiadomość M_p jest tworzona przez sklejenie dwóch zamaskowanych wiadomości. Pierwsza z nich ukrywa ziarno wykorzystane do zamaskowania chronionej wiadomości, natomiast druga maskuje samą wiadomość. Przy okazji widać wyraźnie, że korzystanie z tego mechanizmu wiąże się ze sporym narzutem, gdyż wymaga przechowania dodatkowo ziarna oraz skrótu ciągu parametryzującego *P*. Ziarno jest długości samego skrótu, więc przyjmując *hLen* za długość skrótu i *kLen* za długość klucza (obie długości w oktetach), można wyrazić maksymalną długość szyfrowanej wiadomości jako:

$$MaxLen = kLen - 2hLen - 2$$

W przeciwieństwie do oryginalnego dopełniania PKCS #1 ciąg dopełniający *PS* może mieć długość zerową.

Spróbuj sam: Dopełnienie OAEP

Porównaj poniższy kod z wcześniejszą klasą `PKCS1PaddedRSAExample`. Choć długość szyfrowanej wiadomości nie uległa zmianie, to jednak długość klucza musiała wzrosnąć z 256 do 384 bitów, by pomieścić dopełnienie.

```
package rozdzial4;

import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;

import javax.crypto.Cipher;

/**
 * Przykład RSA z dopełnieniem OAEP i generowaniem losowego klucza.
 */
public class OAEPPaddedRSAExample
{
   public static void main(String[] args) throws Exception
   {
      byte[] input = new byte[] { 0x00, (byte)0xbe, (byte)0xef };
      Cipher cipher = Cipher.getInstance("RSA/NONE/OAEPWithSHA1AndMGF1Padding", "BC");
      SecureRandom random = Utils.createFixedRandom();

      // tworzenie kluczy
      KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA", "BC");

      generator.initialize(386, random);

      KeyPair pair = generator.generateKeyPair();
      Key pubKey = pair.getPublic();
      Key privKey = pair.getPrivate();

      System.out.println("dane wejściowe: " + Utils.toHex(input));

      // szyfrowanie

      cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);
```

```
        byte[] cipherText = cipher.doFinal(input);

        System.out.println("dane zaszyfrowane: " + Utils.toHex(cipherText));

        // deszyfrowanie

        cipher.init(Cipher.DECRYPT_MODE, privKey);

        byte[] plainText = cipher.doFinal(cipherText);

        System.out.println("dane odszyfrowane: " + Utils.toHex(plainText));
    }
}
```

Uruchomienie przykładu powinno spowodować wyświetlenie wyniku z dłuższym niż dotąd szyfrogramem:

```
dane wejściowe: 00beef
dane zaszyfrowane:
020692d99b7b73e8284134590f1f04dbdbdfeee627d3da72a18acf244e41da4a012a834c1c890213a850
8f5406816ef74b
dane odszyfrowane: 00beef
```

384 bity to oczywiście długość znacznie mniejsza od dopuszczalnej długości klucza, więc wymagane przez OAEP dodatkowe miejsce nie sprawia w praktyce problemów. Trzeba jedynie pamiętać, że wykorzystanie OAEP pozostawia w bloku RSA mniej miejsca niż starszy mechanizm dopełnienia określony w PKCS #1 V1.5.

Jak to działa

Od strony praktycznej cała filozofia sprowadza się do podania w nazwie szyfru przekazywanej do `Cipher.getInstance()` innego ciągu dla metody dopełnienia — resztą zajmuje się JCE.

Warto w tym miejscu zwrócić uwagę na format nazwy trybu dopełnienia. Nazwa trybu jest tworzona według wzorca `OAEPWith`*`FunkcjaSkrótu`*`And`*`FunkcjaGenerowaniaMaski`*`Padding`, zgodnie z opisem podanym w dokumentacji w sekcji „Java Cryptography Architecture API Specification and Reference". Konwencja ta pozwala tworzyć nazwy trybów korzystających z wielu różnych funkcji skrótu i funkcji generowania maski, więc na przykład kodowaniu OAEP opartym na SHA-256 i domyślnej funkcji generowania maski odpowiadałby ciąg `OAEPWithSHA256AndMGF1Padding`. W podobny sposób można korzystać z dowolnych obsługiwanych przez danego dostawcę funkcji skrótu i generowania maski.

Poprzedni przykład powinien działać z dowolną wersją JCE z dostawcą BC. W wersji JDK 1.5 do JCE dodano jednak kilka nowych klas mających na celu lepszą obsługę OAEP, a ponieważ ta właśnie metoda dopełnienia jest obecnie zalecana, więc wypada przyjrzeć im się bliżej.

Klasa PSource

Klasa `javax.crypto.spec.PSource` pozwala określić źródło wartości parametru *P*, który w przedstawionych wyżej równaniach uczestniczy w tworzeniu dopełnionego ciągu. Jak dotąd obsługiwany jest tylko jeden typ parametru w postaci publicznego rozszerzenia `PSource` —

klasy PSource.PSpecified. Konstruktor obiektu parametru przyjmuje tablicę bajtów. Domyślną długością tej tablicy jest zero, więc utworzenie domyślnego obiektu PSource używanego do tworzenia dopełnienia OAEP i równoważnego PSource.PSpecified.DEFAULT wyglądałoby tak:

```
PSource defaultPSource = new PSource.PSpecified(new byte[]);
```

Najczęściej nie ma powodu, by to ustawienie zmieniać. Gdyby jednak zaszła potrzeba jawnego etykietowania każdego komunikatu kodowanego metodą OAEP z tym samym kluczem tego samego serwera za pomocą numeru wiadomości, to zamiast powyższego kodu można by użyć:

```
PSource labeledPSource = new PSource.PSpecified(msgNumber);
```

gdzie msgNumber jest tablicą bajtów odpowiadającą numerowi wiadomości dla wysyłanego bloku szyfrogramu. Wszystkie wiadomości o nieprawidłowej wartości parametru *P* zostaną po odszyfrowaniu odrzucone.

Klasa PSource ma tylko jedną metodę pobierającą — jest to PSource.getAlgorithm(), zwracająca nazwę algorytmu. Jeszcze jedna metoda pochodzi z klasy rozszerzającej PSource.
➥PSpecified i jest to metoda getValue(), zwracająca po prostu tablicę bajtów, która posłużyła do utworzenia obiektu. Z metod tych na ogół bezpośrednio korzysta jedynie dostawca JCE.

Klasa MGF1ParameterSpec

Klasa java.security.spec.MGF1ParameterSpec reprezentuje standardową funkcję generowania maski MGF1. Konstruktor tej klasy również przyjmuje tylko jeden argument w postaci nazwy funkcji skrótu, która ma być wykorzystana w ramach funkcji maski.

Domyślnym algorytmem skrótu dla MGF1 jest SHA-1, więc domyślna definicja klasy MGF1ParameterSpec dla zwykłego OAEP wygląda tak:

```
MGF1ParameterSpec defaultMGF1 = new MGF1ParameterSpec("SHA-1");
```

Klasa zawiera też kilka obiektów wartości: MGF1ParameterSpec.SHA1, MGF1ParameterSpec.SHA256, MGF1ParameterSpec.SHA384 i MGF1ParameterSpec.512. Pozwalają one definiować wersje funkcji MGF1 odpowiednio dla algorytmów SHA-1, SHA-256, SHA-384 i SHA-512.

Klasa MGF1ParameterSpec ma tylko jedną własną metodę i jest nią getDigestAlgorithm(), zwracająca nazwę algorytmu skrótu używanego przez funkcję MGF1.

Klasa OAEPParameterSpec

Wykorzystanie obiektów dwóch poprzednich klas polega na przekazaniu ich obiektowi klasy javax.crypto.spec.OAEPParameterSpec. Podobnie jak klasa PSource.PSpecified, ma ona wartość domyślną, dostępną jako OAEPParameterSpec.DEFAULT. Pełna ręczna definicja domyślnej wartości OAEPParameterSpec wyglądałaby tak:

```
OAEPParameterSpec defaultOAEPSpec = new OAEPParameterSpec(
   "SHA-1", "MGF1", MGF1ParametersSpec.SHA1, PSource.PSpecified.DEFAULT);
```

Jak widać, stworzenie obiektu `OAEPParameterSpec` wymaga przekazania wartości odpowiadających parametrom z równania opisującego działanie mechanizmu OAEP. Pierwszy argument to funkcja skrótu `H`, zaś drugi to nazwa funkcji generowania maski `Mask`. Trzecim argumentem jest obiekt zawierający parametry dla funkcji generowania maski — domyślnie będzie to obiekt odpowiadający funkcji MGF1 opartej na SHA-1. Jako ostatni argument przekazywany jest ciąg parametryzujący `P` — w tym przypadku pusta tablica bajtów.

Celem użycia obiektu `OAEPParameterSpec` jest najczęściej określenie własnego ciągu `P`. Samodzielnie tworząc taki obiekt, należy jedynie pamiętać, że dokument PKCS #1 zaleca, by funkcja `H` i funkcja generująca maskę korzystały z tego samego algorytmu funkcji skrótu. Oznacza to, że w przypadku wykorzystania SHA-256 jako algorytmu dla `H` należałoby podać mniej więcej taki kod:

```
OAEPParameterSpec sha256OAEPSpec = new OAEPParameterSpec(
   "SHA-256", "MGF1", MGF1ParameterSpec.SHA256, PSource.PSpecified.DEFAULT);
```

Oczywiście stosownie do potrzeb można tu podać inną wartość ciągu parametryzującego `P`. MGF1 jest na razie jedyną obsługiwaną funkcją generowania maski, a kierunki aktualnych zmian w tej kwestii wskazują dokumenty PKCS #1 i IEEE P1361.

Klasa `OAEPParameterSpec` ma kilka metod `get` pobierających wartości wykorzystane do jej stworzenia, ale podobnie jak w przypadku klas `PSource` i `MGF1ParameterSpec`, są one przede wszystkim używane wewnętrznie przez dostawcę.

Pozostaje jeszcze kwestia wykorzystania utworzonego obiektu `OAEPParameterSpec`. Jak można się spodziewać, jest on po prostu przekazywany metodzie `Cipher.init()`, co w przypadku stworzonego powyżej obiektu `sha256OAEPSPec` mogłoby wyglądać na przykład tak:

```
Cipher c = Cipher.getInstance("RSA/None/OAEPWithSHA256AndMGF1Padding");

c.init(Cipher.ENCRYPT_MODE, publicKey, sha256OAEPSpec, new SecureRandom());
```

Opakowywanie kluczy RSA

W rozdziale 2. poznaliśmy zasady używania kluczy symetrycznych do opakowywania innych kluczy symetrycznych. Jak się okazuje, dokładnie to samo API można wykorzystać do opakowywania kluczy asymetrycznych.

Spróbuj sam: Opakowywanie klucza prywatnego RSA

Przyjrzyj się poniższemu przykładowi. W odróżnieniu od kodu z rozdziału 2. w tym przypadku używany jest tryb `Cipher.PRIVATE_KEY` informujący szyfr opakowujący, że powinien się spodziewać klucza prywatnego. Gdyby opakowywać klucz publiczny (choć byłoby to raczej działanie pozbawione sensu), należałoby podać `Cipher.PUBLIC_KEY`.

```
package rozdzial4;

import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
```

```
import java.security.SecureRandom;

import javax.crypto.Cipher;

/**
 * Opakowywanie klucza RSA za pomocą AES.
 */
public class AESWrapRSAExample
{
   public static void main(String[] args) throws Exception
   {
      Cipher cipher = Cipher.getInstance("AES/ECB/PKCS7Padding", "BC");
      SecureRandom random = new SecureRandom();

      KeyPairGenerator fact = KeyPairGenerator.getInstance("RSA", "BC");
      fact.initialize(1024, new SecureRandom());

      KeyPair keyPair = fact.generateKeyPair();
      Key wrapKey = Utils.createKeyForAES(256, random);

      // opakowanie klucza prywatnego RSA
      cipher.init(Cipher.WRAP_MODE, wrapKey);

      byte[] wrappedKey = cipher.wrap(keyPair.getPrivate());

      // odpakowanie klucza prywatnego RSA
      cipher.init(Cipher.UNWRAP_MODE, wrapKey);

      Key key = cipher.unwrap(wrappedKey, "RSA", Cipher.PRIVATE_KEY);

      if (keyPair.getPrivate().equals(key))
      {
         System.out.println("Klucz odzyskany pomyślnie.");
      }
      else
      {
         System.out.println("Odzyskanie klucza zakończone niepowodzeniem.");
      }
   }
}
```

Wykonanie programu powinno się szczęśliwie zakończyć komunikatem `Klucz odzyskany pomyślnie`.

Jak to działa

Podobnie jak w przykładzie z rozdziału 2., zajmujący się opakowywaniem klucza obiekt `Cipher` wewnętrznie wywołuje metodę `Key.getEncoded()` obiektu `PrivateKey` przekazanego metodzie `Cipher.wrap()`, uzyskując w ten sposób zaszyfrowany strumień bajtów zawierający opakowany klucz. Przez analogię, odpakowanie klucza jest kwestią przekazania szyfrowanych bajtów metodzie `Cipher.unwrap()`. Zaletą opakowywania kluczy w przypadku implementowanych sprzętowo dostawców nieujawniających klucza na zewnątrz jest to, że wewnętrzne wywołanie odpowiednika metody `getEncoded()` i towarzyszące mu operacje szyfrowania mogą się odbywać wyłącznie w obrębie sprzętowego urządzenia szyfrującego.

Co ciekawe, nie trzeba tu korzystać z żadnego konkretnego mechanizmu opakowującego. W przypadku kluczy symetrycznych metoda `getEncoded()` zwracała same bajty składające się na klucz, natomiast dla kluczy asymetrycznych zwraca ona nie tylko wartość klucza, ale również sporo informacji dotyczących ich struktury. Oznacza to, że podanie niewłaściwego klucza tajnego przy wykonaniu metody `unwrap()` dla klucza asymetrycznego zakończy się niepowodzeniem odpakowania, gdyż algorytm odpakowujący nie będzie w stanie zrekonstruować klucza.

Wymiana kluczy tajnych

Jak już wiemy, korzystanie z RSA wiąże się z ograniczeniem rozmiaru szyfrowanej wiadomości do długości jednego bloku. Dostępne miejsce często kurczy się jeszcze bardziej ze względu na obecność dopełnienia. Pozostaje więc pytanie: jak wykorzystać RSA do bezpiecznego szyfrowania większych ilości danych? Teoretycznie można by po prostu podzielić dane na bloki, ale przy większych ilościach bloków byłoby to niezwykle powolne. Co więcej, jeśli napastnik posiada pewne informacje na temat szyfrowanych danych, to dzielenie tych danych na bloki może dodatkowo zagrozić bezpieczeństwu klucza prywatnego odbiorcy.

Na szczęście rozwiązanie tego problemu jest bardzo proste.

Spróbuj sam: Wymiana kluczy tajnych

Spróbuj uruchomić następujący program:

```
package rozdzial4;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;

import javax.crypto.Cipher;
import javax.crypto.spec.IvParameterSpec;
import javax.crypto.spec.SecretKeySpec;

/**
 * Przykład szyfrowania RSA z dopełnieniem OAEP i generowaniem losowych kluczy.
 */
public class RSAKeyExchangeExample
{
   private static byte[] packKeyAndIv(
      Key key,
      IvParameterSpec ivSpec)
      throws IOException
   {
      ByteArrayOutputStream bOut = new ByteArrayOutputStream();

      bOut.write(ivSpec.getIV());
```

```
        bOut.write(key.getEncoded());

        return bOut.toByteArray();
    }

    private static Object[] unpackKeyAndIV(byte[] data)
    {
        byte[] keyD = new byte[16];
        byte[] iv = new byte[data.length - 16];

        return new Object[] {
            new SecretKeySpec(data, 16, data.length - 16, "AES"),
            new IvParameterSpec(data, 0, 16)
        };
    }

    public static void main(String[] args) throws Exception
    {
        byte[] input = new byte[] { 0x00, (byte)0xbe, (byte)0xef };
        SecureRandom     random = Utils.createFixedRandom();

        // utworzenie klucza RSA
        KeyPairGenerator generator = KeyPairGenerator.getInstance("RSA", "BC");

        generator.initialize(1024, random);

        KeyPair pair = generator.generateKeyPair();
        Key pubKey = pair.getPublic();
        Key privKey = pair.getPrivate();

        System.out.println("dane wejściowe: " + Utils.toHex(input));

        // utworzenie klucza symetrycznego i IV
        Key sKey = Utils.createKeyForAES(256, random);
        IvParameterSpec sIvSpec = Utils.createCtrIvForAES(0, random);

        // opakowanie klucza symetrycznego i IV
        Cipher xCipher = Cipher.getInstance("RSA/NONE/OAEPWithSHA1AndMGF1Padding", "BC");

        xCipher.init(Cipher.ENCRYPT_MODE, pubKey, random);

        byte[] keyBlock = xCipher.doFinal(packKeyAndIv(sKey, sIvSpec));

        // szyfrowanie
        Cipher sCipher = Cipher.getInstance("AES/CRT/NoPadding", "BC");

        sCipher.init(Cipher.ENCRYPT_MODE, sKey, sIvSpec);

        byte[] cipherText = sCipher.doFinal(input);

        System.out.println("długość bloku klucza: " + keyBlock.length);
        System.out.println("długość szyfrogramu: " + cipherText.length);

        // opakowanie klucza symetrycznego i IV
        xCipher.init(Cipher.DECRYPT_MODE, privKey);

        Object[] keyIv = unpackKeyAndIV(xCipher.doFinal(keyBlock));
```

```
        // deszyfrowanie
        sCipher.init(Cipher.DECRYPT_MODE, (Key)keyIv[0], (IvParameterSpec)keyIv[1]);

        byte[] plainText = sCipher.doFinal(cipherText);

        System.out.println("dane odszyfrowane: " + Utils.toHex(plainText));
    }
}
```

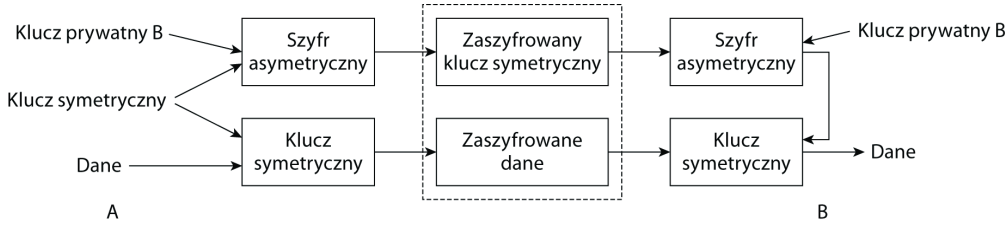
Uruchomienie przykładu powinno dać następujący wynik:

```
dane wejściowe: 00beef
długość bloku klucza: 128
długość szyfrogramu: 3
dane odszyfrowane: 00beef
```

Jak widać, z wykorzystaniem tej techniki wiąże się pewien narzut, ale z drugiej strony pozwala ona przezwyciężyć ograniczenia rozmiaru wiadomości i w pełni wykorzystać większą szybkość szyfrowania przez algorytmy symetryczne.

Jak to działa

Działanie przykładu opiera się na opakowaniu klucza symetrycznego za pomocą obiektu reprezentującego szyfr asymetryczny `xCipher`, a następnie zaszyfrowaniu danych tym kluczem symetrycznym (obiekt `sCipher`). Na rysunku 4.1 widać, że do odbiorcy przesyłana jest zarówno zaszyfrowana wiadomość, jak i zaszyfrowany klucz. Odbiorca odtwarza tajny klucz symetryczny, deszyfrując go swoim kluczem prywatnym, po czym z jego pomocą deszyfruje wiadomość.



A przesyła B zaszyfrowane dane, korzystając z wymiany klucza

Rysunek 4.1.

Do tego typu szyfrowania można używać algorytmu RSA (o czym przekonaliśmy się w powyższym przykładzie) lub algorytmu El Gamal. Cały proces sprawia bardzo dobre wrażenie. Algorytm asymetryczny jest używany do tego, do czego się najlepiej nadaje, a algorytm symetryczny do tego, do czego on z kolei najlepiej się nadaje — wszystko na swoim miejscu.

Do pełni bezpieczeństwa w powyższym przykładzie brakuje jeszcze jednego elementu: nie dostarczyliśmy żadnego sposobu sprawdzenia integralności szyfrowanych danych. W rzeczywistych zastosowaniach koniecznie trzeba pamiętać o dołączaniu do wiadomości kodu MAC, skrótu lub innych danych pozwalających zweryfikować integralność deszyfrowanych informacji.

Uzgadnianie klucza

Algorytmy uzgadniania klucza nie pozwalają szyfrować danych w taki sposób, jak na przykład RSA. Dostarczają one za to mechanizmów uzgodnienia przez komunikujące się strony (najczęściej dwie, ale może być ich więcej) wspólnego klucza tajnego, który posłuży następnie do szyfrowania przesyłanych informacji.

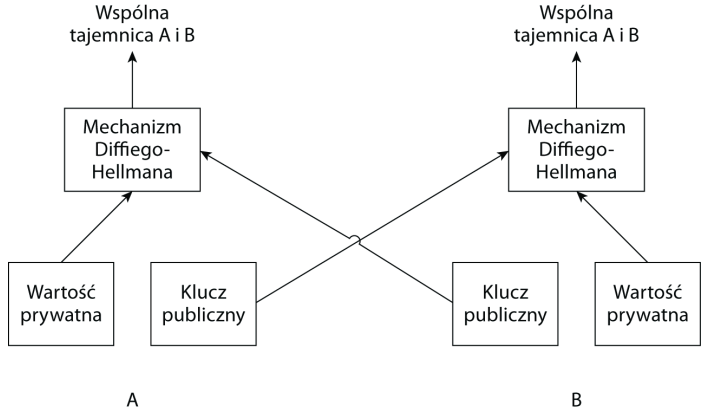
Algorytm Diffiego-Hellmana

Nazwa algorytmu Diffiego-Hellmana pochodzi od nazwisk jego twórców, Whitfielda Diffiego i Martina Hellmana, którzy wynaleźli go w roku 1976. Odtajnione niedawno dokumenty brytyjskiego GCHQ wskazują na to, że ten sam algorytm opracował dwa lata wcześniej Malcolm Williamson. Podobnie jak RSA, algorytm ten korzysta z arytmetyki modulo, jednak jego bezpieczeństwo opiera się na trudności obliczania logarytmów dyskretnych oraz rozwiązania problemu Diffiego-Hellmana. Problemy te są ze sobą powiązane i oba sprowadzają się do trudności wyznaczenia dla danego y liczby x takiej, że $G^x = y$, gdzie G jest generatorem liczb z ciała skończonego nad P (dużą liczbą pierwszą). Jak się okazuje, zadanie to jest na tyle trudne (przynajmniej przy obecnym stanie wiedzy), że można je uznać za przeszkodę nie do przejścia.

Zastosowany proces nie nadaje się do szyfrowania, ale określa zestaw obliczeń pozwalających dwóm stronom wyliczyć ten sam klucz tajny. Porównanie rysunków 4.1 i 4.2 wyraźnie wykazuje różnice między uzgadnianiem klucza a szyfrowaniem. Sam algorytm jest dość prosty. Dla danej dużej liczby pierwszej P i generatora G dla grupy nad P strona A wybiera liczbę prywatną U, a strona B wybiera liczbę prywatną V. Wtedy:

1. A przesyła B wartość G^U mod P (klucz publiczny A).
2. B przesyła A wartość G^V mod P (klucz publiczny B).
3. Odbywa się wymiana danych szyfrowanych kluczem sesji opartym na G^{UV} mod P.

Rysunek 4.2.



Uzgadnianie klucza między dwoma stronami metodą Diffiego-Hellmana

Jeszcze jedna informacja w kwestii konwencji zapisu: przyjęło się oznaczać wartość prywatną każdej ze stron literą X, a wartość publiczną (G^X mod P) literą Y.

Algorytm wygląda na prosty i na szczęście wykorzystujący go kod również nie jest skomplikowany.

Spróbuj sam: Uzgadnianie klucza metodą Diffiego-Hellmana

Spróbuj uruchomić następujący przykład:

```
package rozdzial4;

import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.MessageDigest;

import javax.crypto.KeyAgreement;
import javax.crypto.spec.DHParameterSpec;

/**
 * Dwustronne uzgadnianie klucza metodą Diffiego-Hellmana
 */
public class BasicDHExample
{
   private static BigInteger g512 = new BigInteger(
         "153d5d6172adb43045b68ae8e1de1070b6137005686d29d3d73a7"
      + "749199681ee5b212c9b96bfdcfa5b20cd5e3fd2044895d609cf9b"
      + "410b7a0f12ca1cb9a428cc", 16);
   private static BigInteger p512 = new BigInteger(
         "9494fec095f3b85ee286542b3836fc81a5dd0a0349b4c239dd387"
      + "44d488cf8e31db8bcb7d33b41abb9e5a33cca9144b1cef332c94b"
      + "f0573bf047a3aca98cdf3b", 16);

   public static void main(String[] args) throws Exception
   {
      DHParameterSpec dhParams = new DHParameterSpec(p512, g512);

      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DH", "BC");

      keyGen.initialize(dhParams, Utils.createFixedRandom());

      // przygotowania
      KeyAgreement aKeyAgree = KeyAgreement.getInstance("DH", "BC");
      KeyPair aPair = keyGen.generateKeyPair();
      KeyAgreement bKeyAgree = KeyAgreement.getInstance("DH", "BC");
      KeyPair bPair = keyGen.generateKeyPair();

      // uzgadnianie dwustronne
      aKeyAgree.init(aPair.getPrivate());
      bKeyAgree.init(bPair.getPrivate());

      aKeyAgree.doPhase(bPair.getPublic(), true);
      bKeyAgree.doPhase(aPair.getPublic(), true);

      // generowanie bajtów klucza
      MessageDigest hash = MessageDigest.getInstance("SHA1", "BC");
```

```
        byte[] aShared = hash.digest(aKeyAgree.generateSecret());
        byte[] bShared = hash.digest(bKeyAgree.generateSecret());

        System.out.println(Utils.toHex(aShared));
        System.out.println(Utils.toHex(bShared));
    }
}
```

Wykonanie programu powinno wykazać, że każda ze stron wygenerowała taki sam klucz tajny:

```
98f2669e0458195dece063e99f0b355598eb096b
98f2669e0458195dece063e99f0b355598eb096b
```

W kwestii generowania par kluczy program ten nie różni się specjalnie od poprzednich, jednak dokładniejsze oględziny pozwalają docenić konsekwencje korzystania z metody Diffiego-Hellmana i algorytmów uzgadniania klucza w ogóle.

Pierwszą konsekwencję wyraźnie ilustruje fakt, że podanie stałej zamiast wartości losowej pozwala uzyskać przewidywalny wynik ostateczny. Oznacza to, że ponowne wykorzystanie kluczy użytych do uzgodnienia klucza da w wyniku tę samą tajną wartość, a więc ten sam tajny klucz symetryczny. Stąd też wynika ogólna zasada dotycząca długowieczności kluczy używanych z klasą `KeyAgreement`:

> Czas życia kluczy używanych w ramach procedury uzgadniania klucza nie powinien przekraczać czasu życia klucza symetrycznego, który jest z ich pomocą generowany. Klucze używane z klasą `KeyAgreement` powinny z definicji być tymczasowe.

Druga, mniej oczywista konsekwencja jest taka, że uzgadnianie klucza rzadko odbywa się w bezpiecznych czterech ścianach pliku, a częściej ma miejsce w okolicznościach, gdzie napastnik mógłby podstawić własne etapy procesu uzgadniania. Jest to tak zwany atak pośrednika[1] (ang. *man-in-the-middle*), przedstawiony na rysunku 4.3. Kod poprzedniego programu nie uwzględnia żadnego sposobu weryfikacji pochodzenia klucza nadesłanego przez drugą stronę.

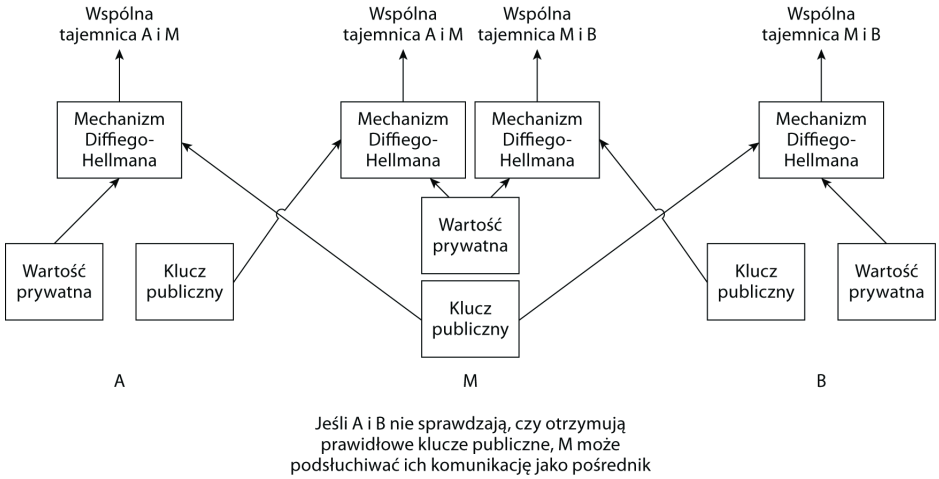
> Klasy `KeyAgreement` należy używać wyłącznie w połączeniu z mechanizmem uwierzytelniania pozwalającym zapobiec atakowi pośrednika.

Jak już wspomniałem, faza przygotowawcza programu jest bardzo podobna do przygotowań RSA. Obsługa metody Diffiego-Hellmana objawia się między innymi podaniem ciągu `DH` przy wyborze generatora pary kluczy, ale jest też kilka osobnych klas JCE obsługujących tę procedurę uzgadniania klucza. Klasy te zostaną omówione poniżej.

Klasa DHParameterSpec

Klasa `javax.crypto.spec.DHParameterSpec` pozwala tworzyć obiekty wartości zawierające dwa lub trzy parametry. Jak widać z kodu, standardowy konstruktor tej klasy przyjmuje dwa argumenty w postaci wartości *P* i *G*, czyli odpowiednio liczby pierwszej i generatora dla grupy odpowiadającej tej liczbie.

[1] Lub: atak metodą przechwytywania przez podmiot pośredniczący — *przyp. red.*



Rysunek 4.3.

Istnieje też drugi konstruktor, pozwalający przyspieszyć proces wymiany klucza. Konstruktor ten przyjmuje długość wartości prywatnej w bitach i ogranicza generowaną wartość do podanego rozmiaru. Wartość prywatna tworzona domyślnie w ramach procesu generowania klucza jest krótsza od *P*, ale o nie więcej niż osiem bitów. Jeśli *P* ma 1024 bity, to wyliczanie klucza publicznego dla potrzeb procesu uzgadniania klucza będzie (często niepotrzebnie) dość powolne. Może się okazać, że w konkretnym zastosowaniu wystarczy wartość prywatna o długości nieprzekraczającej 512 bitów. Przyjmując, że zmienne `p1024` i `g1024` odpowiadają 1024-bitowej liczbie pierwszej i jej generatorowi, można stworzyć obiekt `DHParameterSpec` postaci następującej:

```
DHParameterSpec dhSpec = new DHParameterSpec(p1024, g1024, 512);
```

Do generowania obiektów `DHParameterSpec` można też wykorzystać dostawcę — metodzie tej przyjrzymy się nieco później, przy okazji omawiania klasy `AlgorithmParameterGenerator`.

Obiekty parametrów dla kluczy Diffiego-Hellmana

JCE udostępnia kilka obiektów pozwalających przenosić dane parametrów kluczy dla metody Diffiego-Hellmana w sposób niezależny od konkretnego dostawcy. Używane są do tego celu klasy `javax.crypto.spec.DHPrivateKeySpec` i `javax.crypto.spec.DHPublicKeySpec`, których obiekty mogą przenosić informacje dla dowolnego algorytmu bazującego na tej metodzie.

Konstruktor `DHPrivateKeySpec` przyjmuje wartość prywatną *X* oraz niezbędne do jej używania wartości *P* i *G*. Przykładowe wywołanie konstruktora wygląda więc tak:

```
DHPrivateKeySpec dhPrivateSpec = new DHPrivateKeySpec(x, p, g);
```

Z kolei konstruktor `DHPublicKeySpec` przyjmuje wartość publiczną *Y* i użyte do jej stworzenia wartości *P* i *G*. W kodzie wygląda to po prostu tak:

```
DHPublicKeySpec dhPublicSpec = new DHPublicKeySpec(y, p, g);
```

Podobnie jak w przypadku innych klas specyfikacji, obiekty `DHPrivateKeySpec` i `DHPublicKeySpec` są prostymi obiektami wartości, więc jedynymi ich metodami są metody `get()` pobierające poszczególne wartości.

Interfejsy dla kluczy Diffiego-Hellmana

W poprzednim przykładzie wykorzystana została klasa `Key`. Gdyby potrzebne było bezpieczniejsze typowanie, można skorzystać z interfejsów `javax.crypto.interfaces.DHPrivateKey`, `javax.crypto.interfaces.DHPublicKey` i ich interfejsu nadrzędnego `javax.crypto.interfaces.DHKey`.

Interfejs `DHKey` deklaruje tylko jedną metodę `DHKey.getParams()`, zwracającą obiekt `DHParameterSpec` zawierający klucz publiczny i prywatny.

`DHPrivateKey` również deklaruje tylko jedną metodę — jest to `DHPrivateKey.getX()`, zwracająca wartość prywatną nieprzesyłaną w procedurze uzgadniania klucza.

Wreszcie `DHPublicKey` też ma jedną metodę `DHPublicKey.getY()`. Zwraca ona wartość G^X mod P, gdzie X jest wartością zwracaną przez metodę `getX()` odpowiedniego klucza prywatnego, a G i P są wartościami pobranymi z obiektu `DHParameterSpec`.

Diffie-Hellman bazujący na krzywych eliptycznych

Tradycyjny algorytm Diffiego-Hellmana nie jest jedyną metodą uzgadniania klucza — to samo zadanie można wykonać z pomocą kryptografii bazującej na krzywych eliptycznych.

Wykorzystanie krzywych eliptycznych w kryptografii opiera się na własnościach ciał skończonych. Jako metoda kryptograficzna została niezależnie opracowana przez Neala Koblitza z University of Washington i Victora Millera z IBM w 1985 roku.

Ciało to struktura matematyczna pozwalająca wykonywać na swych elementach operacje dodawania, odejmowania, mnożenia i dzielenia zwracające wyniki należące do tego ciała. Mowa tu o ciałach *skończonych*, czyli posiadających skończoną liczbę elementów, dzięki czemu mogą one być wykorzystane do operacji kryptograficznych korzystających z krzywych nad takimi ciałami. Szczególnie istotne są tu dwie konstrukcje ciała: F_p zawiera krzywe nad skończonym ciałem liczb pierwszych p, a F_2^m jest ciałem złożonym z krzywych dających się wyprowadzić z tzw. optymalnej reprezentacji bazy normalnej, czyli wielomianowej reprezentacji wyprowadzonej z ciągów m-bitowych. F_2^m jest przestrzenią o tyle ciekawą, że za sprawą swej binarnej natury może ona być bardzo wydajnie przetwarzana przez komputery. Niestety algorytmy jej przetwarzania są objęte kilkoma patentami, więc dla potrzeb tej książki będziemy się zajmować przede wszystkim krzywymi nad F_p. Na przykładach przekonamy się jednak, że z punktu widzenia JCE/JCA nie ma specjalnej różnicy między korzystaniem z krzywych nad F_2^m i nad F_p.

Bezpieczeństwo metody opiera się na trudności rozwiązania problemu logarytmu dyskretnego na krzywej eliptycznej, który można pokrótce przedstawić następująco:

> Dla danych dwóch punktów na krzywej P i Q znaleźć taką liczbę k, że $kP = Q$.

Okazuje się, że dla dostatecznie dużego *k* znalezienie rozwiązania jest zadaniem bardzo trudnym obliczeniowo. Zrozumienie źródeł bezpieczeństwa kryptosystemu na krzywej eliptycznej pozwoli też lepiej zrozumieć sam algorytm.

Załóżmy, że A i B uzgadniają pewną krzywą i wybierają punkt *G*, przy czym informacje te nie muszą być utrzymywane w tajemnicy. Podobnie jak w przypadku algorytmu Diffiego-Hellmana, A i B wybierają następnie dwie tajne wartości *x* i *y*, dzięki którym możliwa będzie bezpieczna komunikacja:

1. A wysyła B *xG* (klucz publiczny A).
2. B wysyła A *yG* (klucz publiczny B).
3. Odbywa się wymiana danych szyfrowanych kluczem sesji opartym na *xyG*, który każda ze stron wylicza niezależnie.

Istotną kwestią praktyczną jest znalezienie odpowiedniej krzywej. Na szczęście opisanych zostało wiele takich krzywych, na przykład w dokumentach X9.62 i FIPS PUB 186-2. Poniższy przykład wykorzystuje właśnie jedną z tych standardowych krzywych.

Spróbuj sam: Diffie-Hellman na krzywej eliptycznej

Przyjrzyj się następującemu przykładowi:

```
package rozdzial4;

import java.math.BigInteger;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.MessageDigest;
import java.security.spec.ECFieldFp;
import java.security.spec.ECParameterSpec;
import java.security.spec.ECPoint;
import java.security.spec.EllipticCurve;

import javax.crypto.KeyAgreement;

/**
 * Diffie-Hellman w kryptosystemie na krzywej eliptycznej.
 */
public class BasicECDHExample
{
   public static void main(String[] args) throws Exception
   {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("ECDH", "BC");
      EllipticCurve curve = new EllipticCurve(
         new ECFieldFp(new
         BigInteger("ffffffffffffffffffffffffffffffffefffffffffffffff", 16)),
         new BigInteger("ffffffffffffffffffffffffffffffffefffffffffffffffc", 16),
         new BigInteger("64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1", 16));

      ECParameterSpec  ecSpec = new ECParameterSpec(
         curve,
         new ECPoint(
            new BigInteger("188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012", 16),
            new BigInteger("f8e6d46a003725879cefee1294db32298c06885ee186b7ee", 16)),
```

```
            new BigInteger("ffffffffffffffffffffffff99def836146bc9b1b4d22831", 16),
            1);

        keyGen.initialize(ecSpec, Utils.createFixedRandom());

        // przygotowania
        KeyAgreement aKeyAgree = KeyAgreement.getInstance("ECDH", "BC");
        KeyPair aPair = keyGen.generateKeyPair();
        KeyAgreement bKeyAgree = KeyAgreement.getInstance("ECDH", "BC");
        KeyPair bPair = keyGen.generateKeyPair();

        // uzgadnianie dwustronne
        aKeyAgree.init(aPair.getPrivate());
        bKeyAgree.init(bPair.getPrivate());

        aKeyAgree.doPhase(bPair.getPublic(), true);
        bKeyAgree.doPhase(aPair.getPublic(), true);

        // generowanie bajtów klucza
        MessageDigest hash = MessageDigest.getInstance("SHA1", "BC");
        byte[] aShared = hash.digest(aKeyAgree.generateSecret());
        byte[] bShared = hash.digest(bKeyAgree.generateSecret());

        System.out.println(Utils.toHex(aShared));
        System.out.println(Utils.toHex(bShared));
    }
}
```

Uruchomienie programu powinno dać następujący wynik:

```
5ea61569aed14f67b67377dc6ca223e7ab013844
5ea61569aed14f67b67377dc6ca223e7ab013844
```

Zgodnie z planem po obu stronach wygenerowany został ten sam klucz.

Jak to działa

Rzut oka na kod pokazuje wyraźnie, że poza zmianą parametrów początkowych dla generatora klucza i podaniem w ramach argumentu metody `KeyAgreement.getInstance()` ciągu `ECDH` zamiast `DH` całość nie różni się niczym od kodu dla tradycyjnej metody Diffiego-Hellmana. Stojące za całym procesem operacje matematyczne są wprawdzie zupełnie inne, ale na poziomie warstwy abstrakcji JCE korzystanie z obu metod wygląda w zasadzie tak samo.

A gdzie różnice? Przyjrzyj się bliżej procesowi tworzenia parametrów — pozwoli to dokładniej zrozumieć działanie całej metody i pokaże związek między obiektami w kodzie a parametrami omówionymi w części teoretycznej.

ECField, ECFieldFp i ECFieldF2m

Interfejs `java.security.spec.ECField` jest bazowym interfejsem do obsługi ciał skończonych dla krzywych eliptycznych. Implementują go klasy `java.security.spec.ECFieldFp` i `java.security.spec.ECFieldF2m`. `ECField` deklaruje tylko jedną metodę `ECField.get-FieldSize()`,

zwracającą długość ciała skończonego — w przypadku `ECFieldFp` będzie to długość liczby pierwszej *p* w bitach, natomiast dla `ECFieldF2m` będzie to długość *m* (również w bitach).

`ECFieldFp` jest prostym pojemnikiem dla liczby pierwszej określającej ciało, nad którym operuje krzywa. Klasa ma jeden konstruktor przyjmujący liczbę pierwszą i jedną metodę `ECFieldFp.getP()`, która zwraca tę liczbę.

`ECFieldF2m` jest — jak sama nazwa wskazuje — pojemnikiem dla informacji określających ciało, nad którym operuje dowolna z używanych krzywych. Klasa ta ma więcej możliwości konstrukcji od `ECFieldFp`, gdyż pozwala stworzyć F_2^m krzywych zarówno nad bazą normalną, jak i bazą wielomianową. Z tego też względu niektóre metody `get()` tej klasy mogą zwracać `NULL`, w zależności od użytej bazy. Dogłębne zrozumienie tej klasy nie obejdzie się bez samodzielnego zgłębienia teorii krzywych eliptycznych.

Na szczęście od strony czysto praktycznej wystarczy pamiętać, że zarówno klasa `ECFieldF2m`, jak i `ECFieldFp` służą po prostu do identyfikacji ciała.

Po ustaleniu ciała, nad którym krzywa będzie operować, trzeba zdefiniować samą krzywą.

Klasa EllipticCurve

Klasa `java.security.spec.EllipticCurve` pełni rolę pojemnika dla wartości opisujących krzywą eliptyczną. Jak można się spodziewać, posiada ona komplet metod `get()` pozwalających pobierać obiekty odpowiadające poszczególnym składowym, jednak najciekawszy jest w jej przypadku przebieg konstrukcji obiektu.

Zrozumienie zasad tworzenia obiektu `EllipticCurve` znacznie ułatwi znajomość typowego równania dla używanej w tym przypadku krzywej eliptycznej (z drobnymi wariacjami):

$$y^2 = x^3 + ax^2 + b$$

gdzie wartości *x* i *y* należą do danego ciała skończonego, a wartości *a* i *b* są stałe.

Spójrzmy raz jeszcze na kod tworzący krzywą wykorzystaną w poprzednim przykładzie. Jest to wywołanie podstawowego konstruktora klasy `EllipticCurve`:

```
EllipticCurve curve = new EllipticCurve(
   new ECFieldFp(new BigInteger(
      "fffffffffffffffffffffffffffffffeffffffffffffffff", 16)),
   new BigInteger("fffffffffffffffffffffffffffffffefffffffffffffffc", 16),
   new BigInteger("64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1", 16));
```

Porównując kod ze wzorem krzywej, widzimy, że pierwszym argumentem jest samo pole, w tym przypadku oparte na liczbie pierwszej reprezentowanej przez obiekt `BigInteger` przekazany do konstruktora obiektu `ECFieldFp`. Pozostałe dwa argumenty to (zgodnie z równaniem) wartości stałych *a* i *b*.

Drugi konstruktor krzywej eliptycznej dodatkowo przyjmuje wartość inicjalizującą.

Krzywa została już określona, więc do kompletu informacji pozostaje jeszcze ustalić punkt na krzywej, na podstawie którego wyliczane będą klucze publiczne.

Klasa ECPoint

Klasa `java.security.spec.ECPoint` stanowi pojemnik dla współrzędnych punktu na krzywej eliptycznej (czyli również współrzędnych w przestrzeni operowania krzywej).

Klasa ta nie jest specjalnie ciekawa. Jest ona tworzona na podstawie współrzędnych (X, Y) punktu (dwóch obiektów `BigInteger`), natomiast metody `ECPoint.getAffineX()` i `ECPoint.getAffineY()` zwracają wartości poszczególnych współrzędnych. Dobrze tylko pamiętać, że klasa ta zawiera statyczny obiekt reprezentujący punkt w nieskończoności `ECPoint.POINT_INFINITY`, zwracający dla obu metod wartość `NULL`.

Klasa ECParameterSpec

Klasa `java.security.spec.ECParameterSpec` pozwala połączyć wszystkie wymienione wyżej elementy w całość i wykorzystać je w kryptosystemie opartym na krzywej eliptycznej.

Klasa ma jeden konstruktor, którego argumentami są: krzywa, punkt bazowy, rząd punktu bazowego oraz dopełnienie algebraiczne. Wiemy już, jak wygląda obiekt krzywej i czym jest punkt bazowy (czyli generator). Pozostaje jeszcze przybliżyć ostatnie dwa parametry. Tym razem przyda się mieć przed oczami definicję klasy — oto odpowiedni fragment kodu z ostatniego przykładu:

```
ECParameterSpec ecSpec = new ECParameterSpec(
   curve,
   new ECPoint(
      new BigInteger("188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012", 16),
      new BigInteger("f8e6d46a003725879cefee1294db32298c06885ee186b7ee", 16)),
   new BigInteger("ffffffffffffffffffffffff99def836146bc9b1b4d22831", 16),
   1);
```

Rząd to ostatni argument typu `BigInteger` przekazywany do konstruktora. Wraz z dopełnieniem algebraicznym (w tym przypadku równym 1) wyraża on pewne właściwości punktu bazowego względem krzywej. Rząd jest dużą liczbą pierwszą, która po pomnożeniu przez dopełnienie daje liczbę punktów dostępnych na krzywej. Ze względów wydajności wartość dopełnienia algebraicznego powinna być możliwie najmniejsza, stąd też najczęściej spotyka się wartości od 1 do 4.

Połączenie wszystkich tych wartości pozwala wyliczyć wartość prywatną i wybrać punkt na krzywej, którego parametry można następnie udostępnić stronom, z którymi chcemy się komunikować — czy to w ramach uzgadniania klucza, czy też w celu weryfikacji podpisów cyfrowych.

Klasa ECGenParameterSpec

Wspomniałem wcześniej, że istnieje wiele predefiniowanych (czyli *nazwanych*) krzywych eliptycznych wraz ze skojarzonymi z nimi punktami. Źródłem informacji o takich krzywych są przede wszystkim dokumenty X9.62 i FIPS PUB-186-2. W zależności od krzywych obsługiwanych przez konkretnego dostawcę można korzystać z krzywych nazwanych i parametrów określonych na nich punktów za pomocą klasy `java.security.spec.ECGenParameterSpec`.

Tak się składa, że krzywa wykorzystana w programie `BasicECDHExample` nosi nazwę prime192v1, a jej specyfikacja podana jest w dokumencie X9.62. Oznacza to, że zamiast ręcznie wypisywać definicję krzywej i jej parametrów jako:

```
EllipticCurve curve = new EllipticCurve(
   new ECFieldFp(new BigInteger(
      "fffffffffffffffffffffffffffffffeffffffffffffffff", 16)),
   new BigInteger("fffffffffffffffffffffffffffffffefffffffffffffffc", 16),
   new BigInteger("64210519e59c80e70fa7e9ab72243049feb8deecc146b9b1", 16));

ECParameterSpec ecSpec = new ECParameterSpec(
   curve,
   new ECPoint(
      new BigInteger("188da80eb03090f67cbf20eb43a18800f4ff0afd82ff1012", 16),
      new BigInteger("f8e6d46a003725879cefee1294db32298c06885ee186b7ee", 16)),
   new BigInteger("ffffffffffffffffffffffff99def836146bc9b1b4d22831", 16),
   1);
```

można było importować klasę `java.security.spec.ECGenParameterSpec` i po prostu napisać:

```
ECGenParameterSpec ecSpec = new ECGenParameterSpec("prime192v1");
```

Spróbuj wprowadzić taką zmianę w programie — wynik powinien być dokładnie taki sam.

Wykaz nazwanych krzywych eliptycznych obsługiwanych przez dostawcę Bouncy Castle przedstawia dodatek B. Listy krzywych obsługiwanych przez innych dostawców są na ogół podawane w ich dokumentacji.

Kryptografia krzywej eliptycznej przed JDK 1.5

W wersjach Javy poprzedzających JDK 1.5 nie było bezpośredniej obsługi kryptografii krzywej eliptycznej, więc API używane do tego celu zależały od konkretnego dostawcy. Dodatek B przedstawia opis obsługi kryptografii krzywej eliptycznej w wersjach JDK starszych od 1.5 za pomocą API Bouncy Castle. Stosowane tam klasy są podobne do swych odpowiedników ze współczesnego JCA, ale jednak nie są identyczne (i dotyczy to większości dostawców). Jeśli zamierzasz intensywnie korzystać z kryptografii opartej na krzywych eliptycznych w Javie, to obecnie warto się przesiąść na JDK 1.5 — pozwala to zaoszczędzić sporo pisania.

Diffie-Hellman z wieloma stronami

Ciekawą cechą uzgadniania klucza tradycyjną metodą Diffiego-Hellmana jest to, że może ona posłużyć do uzgodnienia klucza między więcej niż dwoma stronami. Właśnie z tego względu metoda `KeyAgreement.doPhase()` jest w stanie zwrócić klucz.

Spróbuj sam: Trójstronny Diffie-Hellman

Przerobienie wcześniejszego przykładu `BaseDHExample` tak, by uzgadnianie klucza dotyczyło trzech stron, wymaga jedynie kilku prostych zmian. Na początek trzeba będzie dodać trzeci obiekt `KeyAgreement` i utworzyć dla niego klucz:

```
KeyAgreement cKeyAgree = KeyAgreement.getInstance("DH", "BC");
KeyPair cPair = keyGen.generateKeyPair();
```

Teraz trzeba inicjalizować utworzony obiekt `KeyAgreement` tak samo, jak pozostałe dwa obiekty:

```
cKeyAgree.init(cPair.getPrivate());
```

W tym miejscu następuje zmiana w dotychczasowym algorytmie postępowania. Przed wywołaniem metod `doPhase()` obiektów `KeyAgreement` z wartością `true` parametru `lastPhase` konieczne jest wprowadzenie dodatkowego etapu, w ramach którego metoda ta będzie wywoływana z wartością `false` dla `lastPhase`. Wynikiem wykonania tego etapu są trzy klucze pośrednie, które będą używane w następnej fazie. Poniższy kod wymaga dodatkowo importowania interfejsu `java.security.Key`:

```
Key ac = aKeyAgree.doPhase(cPair.getPublic(), false);
Key ba = bKeyAgree.doPhase(aPair.getPublic(), false);
Key cb = cKeyAgree.doPhase(bPair.getPublic(), false);
```

Wygenerowane w ten sposób klucze pośrednie są następnie używane w wywołaniach odpowiadających ostatniej fazie negocjacji:

```
aKeyAgree.doPhase(cb, true);
bKeyAgree.doPhase(ac, true);
cKeyAgree.doPhase(ba, true);
```

Na tym etapie wszystkie trzy obiekty `KeyAgreement` powinny już być w takim stanie, by wywołania metody `KeyAgreement.generateSecret()` zwracały tę samą wartość dla każdego obiektu. Można to sprawdzić, dodając na końcu kodu fragment obsługujący generowanie i wyświetlanie wyniku dla trzeciego obiektu:

```
byte[] cShared = hash.digest(cKeyAgree.generateSecret());
System.out.println(Utils.toHex(cShared));
```

Jeśli program nadal korzysta ze stałej wartości losowej z klasy `Utils`, to jego uruchomienie powinno dać następujący wynik:

```
dd602f60ae382db7b435dd71b0674f5ab64bbb7b
dd602f60ae382db7b435dd71b0674f5ab64bbb7b
dd602f60ae382db7b435dd71b0674f5ab64bbb7b
```

Jak to działa

Jak widać, pomimo dość podeszłego wieku algorytm Diffiego-Hellmana nadal jest przydatny. Spróbuj sobie wyobrazić trójstronne uzgadnianie klucza za pomocą algorytmu RSA. Przy podejściu naiwnym wymagałoby to wymiany kluczy z każdą z pozostałych stron, przez co uzgodnienie klucza dla trójstronnej rozmowy mogłoby wymagać nawet sześciokrotnego przesłania klucza RSA. Każda ze stron miałaby wtedy dodatkowy problem w postaci konieczności śledzenia, którego klucza używać dla którego rozmówcy — zaradzenie tej trudności wymagałoby wprowadzenia jeszcze jednej rundy negocjacji, by wszystkie strony mogły uzgodnić wspólny klucz. Uzgadnianie klucza metodą Diffiego-Hellmana jest więc nadal najprostszym sposobem uzgodnienia wspólnego klucza dla kilku stron, choć należy cały czas pamiętać o problemie uwierzytelnienia klucza.

Algorytm El Gamala[2]

Algorytm El Gamala jest odmianą metody Diffiego-Hellmana, więc jego bezpieczeństwo opiera się na tych samych zagadnieniach matematycznych. Choć można twierdzić, że jest mniej elegancki od RSA, algorytm ten jest powszechnie używany, na przykład jako domyślny algorytm dla większości kluczy szyfrujących w Open PGP (RFC 2440).

Aby metodą El Gamala wysłać wiadomość do odbiorcy o kluczu publicznym G^y mod P, należy utworzyć tymczasowy klucz publiczny G^x mod P, zaszyfrować wiadomość, mnożąc ją modulo P przez G^{xy} mod P, i wysłać szyfrogram w jednym bloku z tymczasowym kluczem publicznym. Algorytm działa bez zarzutu, ale (jak się przekonamy w praktyce) ma tę wadę, że szyfrogram jest dwukrotnej długości klucza.

Spróbuj sam: Szyfrowanie metodą El Gamala

Poniższy kod ilustruje generowanie losowego klucza i szyfrowanie algorytmem El Gamala. Całość wykonuje się znacznie wolniej od równoważnego kodu dla RSA. Przyczyny takiego stanu rzeczy poznamy nieco później — spróbuj uruchomić ten przykład, a przekonasz się, co w tym przypadku znaczy „dużo wolniej”.

```
package rozdzial4;

import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;

import javax.crypto.Cipher;

/**
 * Przykład użycia algorytmu El Gamala z losowym generowaniem klucza.
 */
public class RandomKeyElGamalExample
{
   public static void main(String[] args) throws Exception
   {
      byte[] input = new byte[] { (byte)0xbe, (byte)0xef };
      Cipher cipher = Cipher.getInstance("ElGamal/None/NoPadding", "BC");
      KeyPairGenerator generator = KeyPairGenerator.getInstance("ElGamal", "BC");
      SecureRandom random = Utils.createFixedRandom();

      // tworzenie kluczy

      generator.initialize(256, random);

      KeyPair pair = generator.generateKeyPair();
      Key pubKey = pair.getPublic();
      Key privKey = pair.getPrivate();
```

[2] Nazwa algorytmu pochodzi od nazwiska jego twórcy, dr. Tahera Elgamala. W literaturze występuje kilka pisowni nazwy algorytmu, w tym *Elgamal*, *ElGamal* i (stosowana w tej książce) *El Gamal* — *przyp. tłum.*

```
        System.out.println("dane wejściowe: " + Utils.toHex(input));

        // szyfrowanie

        cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);

        byte[] cipherText = cipher.doFinal(input);

        System.out.println("dane zaszyfrowane: " + Utils.toHex(cipherText));

        // deszyfrowanie

        cipher.init(Cipher.DECRYPT_MODE, privKey);

        byte[] plainText = cipher.doFinal(cipherText);

        System.out.println("dane odszyfrowane: " + Utils.toHex(plainText));
    }
}
```

Uruchomienie programu daje następujący wynik:

```
dane wejściowe: beef
dane zaszyfrowane: 8c2e699772c14496bc82400d11decae4f662fe90864e8c55
➥3b78136679fcdfaa60c378b569083525c021fcf77e40f661525da56ed4133df92848aaba2459dff5
dane odszyfrowane : beef
```

Jak to działa

Nie ma powodów, by specjalnie się rozwodzić nad samym sposobem korzystania z algorytmu. W kwestii szyfrowania danych kluczem publicznym El Gamal nie różni się zbytnio od RSA — szyfrowanie bazuje na operacjach arytmetycznych, więc dla poprawnego obsłużenia danych zaczynających się zerami trzeba korzystać z dopełnienia. Najważniejszą różnicą w porównaniu z RSA jest to, że szyfrogram jest dwa razy dłuższy od klucza (w RSA miał on długość klucza). Praktyczne znaczenie tego faktu zależy od ograniczeń konkretnej aplikacji, ale dłuższy szyfrogram jest jednym z powodów, dla których El Gamal nie jest ogólnie algorytmem preferowanym.

Największym problemem (przynajmniej w przypadku naszego przykładowego programu) jest mizerna szybkość generowania klucza. Jak się wcześniej przekonaliśmy, wygenerowanie pary kluczy dla algorytmu El Gamala wymaga takich samych parametrów, jak algorytm Diffiego-Hellmana, a wyliczenie tych wartości od zera jest bardzo kosztowne. Obiekt `KeyPairGenerator` zainicjalizowany wyłącznie długością klucza musi najpierw wygenerować wartości *P* i *G*, a dopiero potem może wygenerować parę kluczy. W praktyce jest to koszt ponoszony jednokrotnie (przynajmniej w przypadku dostawcy Bouncy Castle) — generowanie kolejnych par kluczy jest już znacznie szybsze, gdyż można używać gotowych wartości *P* i *G* wyliczonych dla pierwszej pary. Nie powinno dziwić, że generator par kluczy dla metody Diffiego-Hellmana zachowuje się podobnie.

Najwygodniej byłoby więc wstępnie wygenerować obiekt `DHParameterSpec`, za pomocą którego można by podać gotowe parametry (identycznie jak w przypadku algorytmu Diffiego-Hellmana). Na szczęście jest to możliwe — wystarczy wygenerować niezbędne parametry, korzystając z obiektu klasy `AlgorithmParameterGenerator`.

Klasa AlgorithmParameterGenerator

Podobnie do wielu innych klas w JCA, instancje klasy `AlgorithmParameterGenerator` są tworzone za pośrednictwem metody fabrykującej `getInstance()` i podobnie jak one przestrzegają ogólnych reguł priorytetu dostawców w przypadku znalezienia więcej niż jednego dostawcy implementującego żądany algorytm. Spośród metod klasy `AlgorithmParameterGenerator` najbardziej będzie nas interesować metoda `init()` (dostępna w czterech odmianach) oraz metoda `generateParameters()` służąca do pobrania wygenerowanego obiektu `AlgorithmParameters`.

AlgorithmParameterGenerator.init()

Metoda `init()` jest dostępna w czterech wersjach. Pierwsza z nich przyjmuje tylko wielkość generatora, druga — wielkość generatora i źródło danych losowych, a pozostałe dwie przyjmują obiekty klas implementujących `AlgorithmParameterSpec`, co przydaje się w sytuacjach wymagających przekazania innych parametrów niż tylko rozmiar generatora. Wybór metody zależy tu od sytuacji. Dla algorytmów Diffiego-Hellmana czy El Gamala wystarczy sam rozmiar, by wygenerować podstawowe parametry w postaci liczby pierwszej P i generatora G. Jeśli jednak zajdzie potrzeba utworzenia obiektu parametrów pozwalającego ograniczyć długość wartości prywatnej w metodzie Diffiego-Hellmana, to sama wielkość generatora nie wystarczy — dodatkowe informacje trzeba będzie przekazać w ramach obiektu klas implementujących `AlgorithmParameterSpec`.

Kolejne punkty przedstawiają przykłady obu zastosowań.

AlgorithmParameterGenerator.generateParameters()

Metoda ta zwraca obiekt klasy `AlgorithmParameters` zawierający generowane parametry. Szczegółowy opis tej klasy przedstawiłem już w rozdziale 2., więc nie będę się tu nad nim rozwodzić. Wspomnę tylko, że (jak zapewne już się domyślasz) obiekty `AlgorithmParameters` przydają się dosłownie wszędzie.

Starczy tej teorii — pora zapoznać się z kodem.

Spróbuj sam: Algorytm El Gamala z obiektem AlgorithmParameterGenerator

Przyjrzyj się poniższemu przykładowi i porównaj go z kodem programu `RandomKeyElGamalExample`. Uruchom program i czytaj dalej.

```
package rozdzial4;

import java.security.AlgorithmParameterGenerator;
import java.security.AlgorithmParameters;
import java.security.Key;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.spec.AlgorithmParameterSpec;
```

```
import javax.crypto.Cipher;
import javax.crypto.spec.DHParameterSpec;

/**
 * Użycie algorytmu El Gamal z generowaniem losowego klucza i obiektem klasy AlgorithmParameters.
 */
public class AlgorithmParameterExample
{
   public static void main(String[] args) throws Exception
   {
      byte[] input = new byte[] { (byte)0xbe, (byte)0xef };
      Cipher cipher = Cipher.getInstance("ElGamal/None/NoPadding", "BC");
      SecureRandom random = Utils.createFixedRandom();

      // tworzenie parametrów
      AlgorithmParameterGenerator a =
      AlgorithmParameterGenerator.getInstance("ElGamal", "BC");

      a.init(256, random);

      AlgorithmParameters params = a.generateParameters();
      AlgorithmParameterSpec dhSpec = params.getParameterSpec(DHParameterSpec.class);

      // tworzenie kluczy
      KeyPairGenerator generator = KeyPairGenerator.getInstance("ElGamal", "BC");

      generator.initialize(dhSpec, random);

      KeyPair pair = generator.generateKeyPair();
      Key pubKey = pair.getPublic();
      Key privKey = pair.getPrivate();

      System.out.println("dane wejściowe: " + Utils.toHex(input));

      // szyfrowanie

      cipher.init(Cipher.ENCRYPT_MODE, pubKey, random);

      byte[] cipherText = cipher.doFinal(input);

      System.out.println("dane zaszyfrowane: " + Utils.toHex(cipherText));

      // deszyfrowanie

      cipher.init(Cipher.DECRYPT_MODE, privKey);

      byte[] plainText = cipher.doFinal(cipherText);

      System.out.println("dane odszyfrowane: " + Utils.toHex(plainText));
   }
}
```

Wykorzystany tu generator liczb „losowych” otrzymuje to samo ziarno, co w poprzednich przykładach, dzięki czemu program powinien wypisać taki sam wynik, jak wcześniejszy `RandomKeyElGamalExample`. Przyczyna jest prosta: przetwarzanie danych wykonywane wewnętrznie przez dostawcę jest (przynajmniej w przypadku dostawcy Bouncy Castle) dokładnie takie samo, jak ręcznie zakodowane operacje z powyższego programu.

Jak to działa

W tym przykładzie parametry odpowiadające wartościom *P* i *G* są jawnie generowane i przekazywane do obiektu generatora pary kluczy, podczas gdy w poprzednim programie były one tworzone automatycznie na podstawie zadanej długości klucza. Zwalnia to generator par kluczy z obowiązku samodzielnego generowania parametrów.

Różnicę w szybkości działania trudno ocenić, po prostu uruchamiając oba przykłady, ale wystarczy wokół operacji generowania klucza w obu programach dodać kod mierzący czas wykonania, by przekonać się, że czas wykonywania metody `KeyPairGenerator.generateKeyPair()` jest znacznie krótszy dla drugiego programu.

Wcześniej wspominałem, że klasa `AlgorithmParameterGenerator` może też przyjmować obiekt implementujący `AlgorithmParameterSpec` jako argument swej metody `init()`. Tak się składa, że istnieje implementująca ten interfejs klasa przeznaczona właśnie do generowania parametrów dla algorytmów w rodzaju Diffiego-Hellmana — jest to klasa `DHGenParameterSpec`.

Klasa DHGenParameterSpec

Wiemy już, że algorytm Diffiego-Hellmana można przyspieszyć, ograniczając długość wartości prywatnej powiązanej z kluczem publicznym. Wiemy też, że można stworzyć obiekt `DHParameterSpec`, który nałoży takie ograniczenie na długość wartości prywatnych generowanych podczas używania go z odpowiednim obiektem `KeyPairGenerator`. Dobrze byłoby mieć możliwość uwzględnienia tego ograniczenia również w generowanych parametrach, stąd też JCE udostępnia klasę pozwalającą konfigurować obiekt `AlgorithmParameterSpec` tworzony dla potrzeb algorytmu Diffiego-Hellmana — jest to klasa `javax.crypto.spec.DHGenParameterSpec`. Zamiast określać wyłącznie długość liczby pierwszej *P*, jak ma to miejsce w wierszu:

```
apg.init(256, random);
```

można ograniczyć długość wartości prywatnej na przykład do 200 bitów, wywołując:

```
apg.init(new DHGenParameterSpec(256, 200), random);
```

gdzie argumentami dla konstruktora `DHGenParameterSpec` są długość liczby pierwszej *P* w bitach oraz maksymalna długość wartości prywatnej *Y* (również w bitach). Takie wywołanie powodowałoby tworzenie obiektów, których metoda `DHParameterSpec.getL()` zwracałaby wartość 200, co tym samym ograniczałoby długość wartości prywatnych do 200 bitów.

Podpisy cyfrowe

Użyteczność podpisów cyfrowych polega na tym, że podpis jest tworzony na podstawie tajnych danych znanych wyłącznie jego autorowi, ale może zostać sprawdzony za pomocą informacji publicznie udostępnianych przez podpisującego. Podpis cyfrowy nie tylko poświadcza

autentyczność wiadomości, ale również czyni ją niezaprzeczalną. Jeśli konkretny użytkownik podpisze wiadomość, a następnie zaprzeczy, że ją napisał, to istnieją tylko dwie możliwości: albo tajne dane zostały wykradzione, albo podpisujący mija się z prawdą.

Wszystkie algorytmy podpisów cyfrowych bazują na jednokierunkowych funkcjach skrótu, co wynika z ograniczonej długości wiadomości, jakie mogą przetwarzać algorytmy asymetryczne. Ma to jedną bardzo istotną konsekwencję: maksymalna ilość danych, jaką można bezpiecznie podpisać danym algorytmem, jest ograniczona możliwościami używanej funkcji skrótu. Długość klucza wyznacza jedynie bezpieczeństwo samego skrótu. Jeśli zostanie podpisana wiadomość o zbyt dużej długości dla danego algorytmu skrótu, to zwiększenie długości klucza nic nie pomoże.

> Ilość danych, jakie można bezpiecznie podpisać daną metodą, jest ograniczona maksymalną długością danych wejściowych dla używanego algorytmu skrótu.

Z podpisami cyfrowymi wiążą się dwa procesy: tworzenie podpisu i sprawdzanie podpisu. Wbrew pozorom operacje te różnią się nieco od swych odpowiedników w świecie szyfrów, czyli szyfrowania i deszyfrowania — odbiorcy wystarczy sprawdzić podpis, bez konieczności jego czytania. Operacje tworzenia i sprawdzania podpisów cyfrowych hermetyzuje w Javie klasa `Signature`.

Klasa Signature

Obiekty udostępniające operacje klasy `java.security.Signature` są tworzone za pomocą metody fabrykującej `getInstance()`, podobnie jak wiele innych klas w JCA. Metoda ta przestrzega też zwykłych priorytetów określających zasady wyboru dostawcy w przypadku braku żądania konkretnego dostawcy. W przeciwieństwie do klasy `Cipher` tryby działania klasy `Signature` nie są określane przez składowe statyczne, lecz przez wywołanie jednej z dwóch metod inicjalizacyjnych. Wywołanie `Signature.initSign()` przełączy obiekt `Signature` w tryb tworzenia podpisu, natomiast `Signature.initVerify()` spowoduje przejście w tryb sprawdzania.

Używanie klasy Signature w trybie tworzenia podpisu

Istnieją dwie wersje metody `Signature.initSign()` służącej do przełączenia obiektu `Signature` w tryb tworzenia podpisu. Obie przyjmują najważniejszą informację, czyli klucz prywatny, a jedna przyjmuje dodatkowo źródło danych losowych.

Gdy obiekt `Signature` zostanie już inicjalizowany w trybie składania podpisu cyfrowego, korzystanie z niego bardzo przypomina pracę z poznaną wcześniej klasą `MessageDigest`. Dane są przekazywane do zainicjalizowanego obiektu `Signature` za pośrednictwem jednej z kilku metod `update()`, a po wprowadzeniu wszystkich danych wywoływana jest metoda `Signature.sign()`, która w zależności od wersji może zwrócić podpis cyfrowy w postaci tablicy bajtów lub zapisać go w tablicy bajtów przekazanej podczas wywołania.

Używanie klasy Signature w trybie sprawdzania podpisu

Również metoda `Signature.initVerify()` ma dwie odmiany, lecz w odróżnieniu od metody `initSign()` jedna z wersji służy wyłącznie wygodzie programisty — przyjmuje ona za argument obiekt klasy `Certificate`, podczas gdy wersja podstawowa przyjmuje klucz publiczny.

Wprowadzanie danych do obiektu `Signature` odbywa się tu identycznie, jak w trybie tworzenia klucza, czyli poprzez wywołania metody `update()`, jednak etap weryfikacji wygląda już nieco inaczej. Gdy obiektowi `Signature` zostanie załadowany komplet danych do weryfikacji podpisu, wywoływana jest zwracająca wartość logiczną metoda `Signature.verify()`, której przekazywana jest tablica bajtów zawierająca sprawdzany podpis. Jeśli metoda `verify()` zwróci wartość `true`, to podpis jest w porządku; jeśli `false`, to dane nie pasują do podpisu.

Metody Signature.setParameter() i Signature.getParameters()

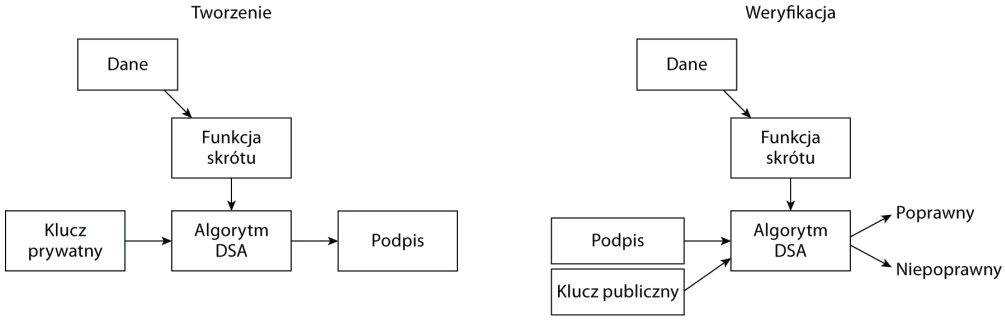
Podobnie jak obiekty `Cipher` czy `KeyPairGenerator`, obiekty klasy `Signature` mogą korzystać z obiektów `AlgorithmParameterSpec` w celu dostarczenia dodatkowych informacji lub zwrócenia obiektu `AlgorithmParameters` zawierającego ustawienia pozwalające w razie potrzeby powtórzyć wykonywaną operację. Jak można się spodziewać, metoda `Signature.getParameters()` zwraca obiekt `AlgorithmParameterSpec` zawierający parametry bieżącej operacji. Nieco inaczej wygląda korzystanie z takiego obiektu z klasą `Signature`, gdyż nie jest on przekazywany w ramach inicjalizacji obiektu, lecz za pomocą metody `Signature.setParameter()`. Jeśli przekazane metodzie parametry nie nadają się dla danej implementacji algorytmu podpisu, metoda ta zgłosi wyjątek `InvalidAlgorithmParameterException`.

Przykład użycia parametrów zobaczymy nieco później, przy okazji omawiania podpisów cyfrowych typu RSA PSS. Najpierw jednak zajmiemy się algorytmem specjalnie stworzonym wyłącznie do tworzenia i sprawdzania podpisów cyfrowych.

Algorytm podpisu cyfrowego DSA

Pierwszą wersję algorytmu podpisu cyfrowego DSA (ang. *Digital Signature Algorithm*) przedstawił w sierpniu 1991 roku NIST. DSA stał się następnie pierwszym na świecie algorytmem podpisu cyfrowego oficjalnie zatwierdzonym przez rząd — został on opisany w dokumencie FIPS PUB 186 jako standard podpisu cyfrowego DSS (ang. *Digital Signature Standard*). Nazwy DSA i DSS są niekiedy stosowane zamiennie, choć między algorytmami jest pewna różnica: DSS wymaga korzystania konkretnie z SHA-1 jako funkcji skrótu.

Ciekawą cechą DSA jest to, że nie da się go użyć do szyfrowania. Na rysunku 4.4 widać, że celem działania algorytmu jest wyłącznie weryfikacja podpisu za pomocą klucza publicznego, czyli bez ujawniania danych użytych do utworzenia podpisu.



Podpis cyfrowy z użyciem algorytmu DSA

Rysunek 4.4.

Zwykły DSA

Podobnie jak w przypadku algorytmu Diffiego-Hellmana, bezpieczeństwo tradycyjnego algorytmu DSA opiera się na trudności obliczania logarytmów dyskretnych. Do skorzystania z mechanizmu DSA potrzebne są następujące dane:

- liczba pierwsza Q taka, że $2^{159} < Q < 2^{160}$;
- liczba pierwsza P taka, że Q jest dzielnikiem $(P - 1)$;
- generator G dla niepowtarzalnej grupy cyklicznej rzędu Q w ciele P.

Nie będę się tu zagłębiać w operacje matematyczne związane z obliczaniem generatora. Wystarczy wiedzieć, że znając P, Q i G, można utworzyć klucz publiczny, wybierając wartość prywatną X taką, że $1 \leq X \leq Q$, a następnie obliczając wartość publiczną $Y = G^X \bmod P$. Klucz publiczny stanowi zbiór wartości Y, P, Q i G, natomiast kluczem prywatnym są X, P, Q i G.

Znając klucz publiczny i prywatny, można już tworzyć i sprawdzać podpisy cyfrowe. W odróżnieniu od algorytmu RSA, gdzie operacje z kluczem publicznym i prywatnym są wykonywane w ten sam sposób, algorytm DSA korzysta z różnych obliczeń do generowania klucza i sprawdzania go.

Dla danej funkcji skrótu `H()` (w przypadku DSS będzie to zawsze SHA-1) generowanie podpisu DSA dla wiadomości M obejmuje następujące etapy:

1. Wybór tajnej, losowej liczby całkowitej K takiej, że $0 < K < Q$.
2. Obliczenie $R = (G^K \bmod P) \bmod Q$.
3. Obliczenie $S = ((K^{-1} \bmod Q)(H(M) + XR)) \bmod Q$.

Podpis stanowią liczby R i S.

Sprawdzanie podpisu wymaga znajomości klucza publicznego strony podpisującej oraz wartości R i S. Etapy weryfikacji podpisu to:

1. Sprawdzenie, czy $0 < R < Q$ i $0 < S < Q$ — jeśli nie, podpis jest odrzucany.
2. Obliczenie $A = S^{-1} \bmod Q$, $B = (A\mathrm{H}(M)) \bmod Q$ oraz $C = (RA) \bmod Q$.
3. Wyliczenie $V = (G^B Y^C \bmod P) \bmod Q$.

Jeśli V jest równe R, to podpis jest akceptowany. W przeciwnym razie jest on odrzucany.

Nie będę specjalnie wnikać w matematyczną stronę całej procedury, gdyż szczegółowy jej opis można znaleźć w książkach przedstawionych w dodatku D, na przykład w *Handbook of Applied Cryptography* Menezesa, van Oorschota i Vanstone'a. W tym miejscu bardziej interesuje nas związek między tymi obliczeniami a praktyczną stroną korzystania z klasy `Signature` w Javie.

Spróbuj sam: DSA

Oto przykład zastosowania algorytmu DSA, a przy okazji pierwszy program korzystający z klasy `Signature`. Jak widać, w zasadzie wystarczy znać nazwę algorytmu — klasa `Signature` skutecznie ukrywa faktyczne obliczenia, dzięki czemu praca z DSA sprowadza się do korzystania z wyników procesu tworzenia i sprawdzania podpisu.

```
package rozdzial4;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Signature;

public class BasicDSAExample
{
   public static void main(String[] args) throws Exception
   {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("DSA", "BC");

      keyGen.initialize(512, new SecureRandom());

      KeyPair keyPair = keyGen.generateKeyPair();
      Signature signature = Signature.getInstance("DSA", "BC");

      // tworzenie podpisu
      signature.initSign(keyPair.getPrivate(), Utils.createFixedRandom());

      byte[] message = new byte[] { (byte)'a', (byte)'b', (byte)'c' };

      signature.update(message);

      byte[]  sigBytes = signature.sign();

      // sprawdzanie podpisu
      signature.initVerify(keyPair.getPublic());

      signature.update(message);

      if (signature.verify(sigBytes))
      {
```

```
            System.out.println("Weryfikacja podpisu zakończona powodzeniem.");
        }
        else
        {
            System.out.println("Błąd weryfikacji podpisu.");
        }
    }
}
```

Uruchomienie programu powinno się zakończyć komunikatem:

```
Weryfikacja podpisu zakończona powodzeniem.
```

Jak to działa

Podobnie jak w każdym algorytmie asymetrycznym, pierwszym krokiem jest stworzenie pary kluczy dla używanego algorytmu. Następnie trzeba inicjalizować obiekt `signature` tak, by korzystał z klucza prywatnego:

```
signature.initSign(keyPair.getPrivate(), Utils.createFixedRandom());
```

Jak wspominałem w teoretycznym opisie algorytmu DSA, do wygenerowania podpisu potrzebna jest liczba losowa. W tym przykładzie wykorzystana została wersja metody `initSign()` przyjmująca poza kluczem prywatnym również źródło danych losowych. Gdyby źródło to nie zostało jawnie podane, dostawca utworzyłby źródło domyślne.

Kolejnym etapem jest wprowadzenie podpisywanych danych do obiektu `signature` poprzez wywołanie metody `update()`. Wywołanie metody `sign()` powoduje obliczenie podpisu i zwrócenie zawierającej go tablicy bajtów.

W drugiej części przykładu wykonywane jest sprawdzenie podpisu utworzonego w części pierwszej. W tym celu dla obiektu podpisu wywoływana jest metoda `initVerify()` z kluczem publicznym podpisującego, po czym treść podpisywanej wiadomości jest przekazywana do obiektu jako argument metody `update()`. Samo sprawdzenie polega na wywołaniu metody `verify()` z argumentem w postaci tablicy bajtowej zawierającej otrzymany podpis. Jeśli podpis pasuje do danych (a w tym przykładzie tak jest), to metoda `verify()` zwróci wartość `true` — w przeciwnym przypadku zwrócona zostanie wartość `false`.

Rzut oka na użycie obiektu klasy `KeyPairGenerator` pokazuje brak omawianych wcześniej parametrów DSA. Są one przekazywane niejawnie w analogiczny sposób, jak w pierwszym przykładzie z algorytmem El Gamala — z braku jawnie przekazanych wartości obiekt `KeyPairGenerator` generuje parametry samodzielnie. Podobnie jak w przypadku algorytmów Diffiego-Hellmana i El Gamala, możliwe jest wstępne wyliczenie niezbędnych parametrów. Do ich przenoszenia służy klasa `DSAParameterSpec`.

Klasa DSAParameterSpec

Obiekty klasy `java.security.spec.DSAParameterSpec` pełnią rolę pojemników dla omówionych wcześniej parametrów algorytmu DSA. Klasa ta ma tylko jeden konstruktor, przyjmujący jako parametry używane do generowania klucza wartości *P*, *Q* i *G*, oraz trzy metody `get()` pobierające wartości poszczególnych parametrów.

Podobnie jak obiekty klasy DHParameterSpec, obiekty klasy DSAParameterSpec można generować za pomocą obiektów klasy AlgorithmParameterGenerator. Tak więc do wygenerowania obiektu DSAParameterSpec dla kluczy 512-bitowych (z których korzystaliśmy już we wcześniejszych przykładach) można by utworzyć obiekt parametrów postaci następującej:

```
AlgorithmParameterGenerator apg = AlgorithmParameterGenerator.getInstance("DSA", "BC");

apg.init(512, new SecureRandom());

AlgorithmParameters params = apg.generateParameters();
AlgorithmParameterSpec dsaSpec = params.getParameterSpec(DSAParameterSpec.class);
```

Utworzonego w ten sposób obiektu parametrów można używać do generowania własnych par kluczy poprzez przekazanie go jako argumentu wywołania metody KeyPairGenerator.
➥initialize().

Poza generowaniem kluczy losowo można je również tworzyć za pomocą klasy KeyFactory. JCA obsługuje taką możliwość, dostarczając klasy pozwalające przenosić parametry dla kluczy DSA.

Obiekty parametrów dla kluczy DSA

Dostarczane przez JCA obiekty parametrów pozwalają przenosić dane i parametry dla kluczy DSA. Są to proste obiekty wartości, mogące służyć do przekazywania informacji określających klucze DSA — java.security.spec.DSAPrivateKeySpec dla klucza prywatnego i java.security.spec.DSAPublicKeySpec dla klucza publicznego.

Klasa DSAPrivateKeySpec ma pojedynczy konstruktor przyjmujący po prostu wartość prywatną *X* i parametry pozwalające wyliczyć z niej wartość publiczną, czyli pobrane z obiektu DSAParameterSpec wartości *P*, *Q* i *G*. Przykładowe wywołanie może wyglądać tak:

```
DSAPrivateKeySpec dsaPrivateSpec = new DSAPrivateKeySpec(x, p, q, g);
```

Klasa DSAPublicKeySpec również ma tylko jeden konstruktor, przyjmujący wartość publiczną *Y* oraz parametry *P*, *Q* i *G*, z których została ona wygenerowana. Wywołanie konstruktora wygląda zatem tak:

```
DSAPublicKeySpec dsaPublicSpec = new DSAPublicKeySpec(y, p, q, g);
```

Podobnie jak w przypadku innych obiektów wartości przenoszących dane klucza, jedynymi metodami klas DSAPrivateKeySpec i DSAPublicKeySpec są metody get() pobierające poszczególne parametry składowe.

Interfejsy dla kluczy DSA

W programie BasicDSAExample klasa KeyPair została wykorzystana bezpośrednio po wygenerowaniu kluczy. W sytuacjach, w których potrzebne jest bezpieczniejsze typowanie, można korzystać z interfejsów java.security.interfaces.DSAPrivateKey i java.security.
➥interfaces.DSAPublicKey, jak również ich interfejsu nadrzędnego java.security.interfaces.DSAKey (rozszerzającego Key). Interfejsy te pozwalają odróżnić klucze DSA od innych kluczy asymetrycznych.

`DSAKey` deklaruje tylko jedną metodę o nazwie `getParams()`, zwracającą obiekt klasy `DSAParameterSpec` odpowiadający parametrom kluczy: publicznego i prywatnego.

`DSAPrivateKey` też deklaruje tylko jedną metodę — jest nią `getX()`, zwracająca tajną wartość prywatną.

Wreszcie `DSAPublicKey` również posiada pojedynczą metodę o nazwie `getY()`. Zwraca ona wartość G^X mod P, gdzie X jest wartością zwracaną przez metodę `getX()` danego klucza prywatnego, a wartości G i P są zawarte w opisującym klucz obiekcie `DSAParameterSpec`.

DSA oparte na krzywej eliptycznej

Istnieje też algorytm generowania podpisów DSA za pomocą krzywej eliptycznej zamiast metod przewidzianych w standardzie DSS — nosi on nazwę ECDSA. Tak jak uzgadnianie klucza metodą Diffiego-Hellmana bazującą na krzywej eliptycznej było podobne do wersji tego algorytmu korzystającej z liczb pierwszych, tak również DSA oparte na krzywej eliptycznej polega na stworzeniu podpisu składającego się z wartości R i S. Poprawność klucza jest sprawdzana tak samo, jak w wersji oryginalnej — poprzez sprawdzenie, czy wartość V wyliczona na podstawie podpisu i klucza publicznego podpisującego jest równa wartości R.

Algorytm ECDSA jest opisany szczegółowo w standardzie X9.62, a jego dokładne zrozumienie wymaga pewnej znajomości matematycznych podstaw kryptosystemów opartych na krzywych eliptycznych, więc nie będę się tu zagłębiał w szczegóły implementacji. Jak się przekonamy, pomimo zupełnie innego aparatu matematycznego korzystanie z ECDSA w Javie nie różni się specjalnie od zwykłego DSA.

Spróbuj sam: DSA oparte na krzywej eliptycznej

Przyjrzyj się poniższemu przykładowi i porównaj go z kodem stworzonej wcześniej klasy `BasicDSAExample`. Jak można się było spodziewać, tworzony jest inny obiekt `KeyPairGenerator`, ale poza tym jedyną różnicą jest parametr `ECDSA` przekazywany podczas wywołania metody `Signature.getInstance()`. Przebieg procesów podpisywania i sprawdzania podpisu jest identyczny, jak w poprzednim przykładzie.

```
package rozdzial4;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Signature;
import java.security.spec.ECGenParameterSpec;

/**
 * Prosty przykład tworzenia i weryfikacji podpisu algorytmem ECDSA.
 */
public class BasicECDSAExample
{
   public static void main(String[] args) throws Exception
   {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("ECDSA", "BC");
      ECGenParameterSpec ecSpec = new ECGenParameterSpec("prime192v1");

      keyGen.initialize(ecSpec, new SecureRandom());
```

```
        KeyPair keyPair = keyGen.generateKeyPair();
        Signature signature = Signature.getInstance("ECDSA", "BC");

        // tworzenie podpisu

        signature.initSign(keyPair.getPrivate(), Utils.createFixedRandom());

        byte[] message = new byte[] { (byte)'a', (byte)'b', (byte)'c' };

        signature.update(message);

        byte[]  sigBytes = signature.sign();

        // sprawdzanie podpisu

        signature.initVerify(keyPair.getPublic());

        signature.update(message);

        if (signature.verify(sigBytes))
        {
           System.out.println("Weryfikacja podpisu zakończona powodzeniem.");
        }
        else
        {
           System.out.println("Błąd weryfikacji podpisu.");
        }
     }
  }
```

Uruchomienie programu powinno się zakończyć komunikatem `Weryfikacja podpisu zakończona powodzeniem`.

Jak to działa

Jak już wspominałem, całą brudną robotę ukrywa tu warstwa abstrakcji JCA. Dzięki temu wystarczą kosmetyczne zmiany w kodzie, by korzystał on z zupełnie innej implementacji.

Warto zwrócić uwagę, że powyższy przykład korzysta z omówionej wcześniej klasy `ECGenParameterSpec`, więc bardzo łatwo można przejść na zupełnie inną krzywą i klucz, po prostu zmieniając ciąg przekazywany do konstruktora obiektu `ECGenParameterSpec`. Na przykład przejście z klucza 192-bitowego na 239-bitowy wymagałoby zamiany tego wiersza:

```
ECGenParameterSpec ecSpec = new ECGenParameterSpec("prime192v1");
```

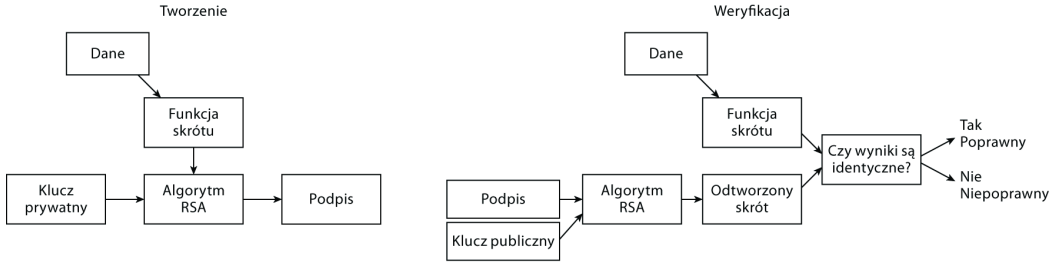
na ten:

```
ECGenParameterSpec ecSpec = new ECGenParameterSpec("prime239v1");
```

Algorytmy podpisu oparte na RSA

Stworzenie podpisu algorytmem RSA sprowadza się do zaszyfrowania odpowiednich danych kluczem prywatnym i udostępnienia wyniku jako podpisu. Mechanizm działania algorytmu RSA oznacza, że taki podpis można odszyfrować kluczem publicznym — proces ten

ilustruje rysunek 4.5. Skuteczność tej metody wynika stąd, że podpis dający się odszyfrować danym kluczem publicznym mógł być utworzony wyłącznie za pomocą odpowiadającego mu klucza prywatnego.



Podpis cyfrowy z użyciem algorytmu RSA

Rysunek 4.5.

Podobnie jak w przypadku trybów dopełnienia, najpopularniejsze metody używania RSA do tworzenia podpisów są opisane w PKCS #1. Analogicznie do dopełnień, istnieją dwa typy podpisu: jeden oparty na wersji 1.5 PKCS #1, a drugi (nowszy) bazujący na wersji 2 PKCS#1.

Podpisy PKCS #1 V1.5

Podpisy w wersji 1.5 PKCS #1 wykorzystują dopełnienie PKCS1 typu 1. Z opisu dopełnienia RSA pamiętamy, że dla dopełnienia PKCS #1 typu 1 dopełniona wersja M_p wiadomości M tworzona jest następująco:

$$M_p = 0x00 \,||\, 0x01 \,||\, F \,||\, 0x00 \,||\, M$$

gdzie F jest ciągiem oktetów 0xFF, a || jest operatorem sklejenia. F musi zawierać co najmniej osiem oktetów, a długość M nie może przekraczać długości klucza w oktetach pomniejszonej o 11.

Spróbuj sam: Generowanie podpisu algorytmem RSA

Spróbuj uruchomić poniższy przykład. Jak się przekonasz, również w tym przypadku zmiany są niewielkie.

```
package rozdzial4;

import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.SecureRandom;
import java.security.Signature;

public class PKCS1SignatureExample
{
   public static void main(String[] args) throws Exception
   {
      KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA", "BC");
```

```
        keyGen.initialize(512, new SecureRandom());

        KeyPair keyPair = keyGen.generateKeyPair();
        Signature signature = Signature.getInstance("SHA1withRSA", "BC");

        // tworzenie podpisu
        signature.initSign(keyPair.getPrivate(), Utils.createFixedRandom());

        byte[] message = new byte[] { (byte)'a', (byte)'b', (byte)'c' };

        signature.update(message);

        byte[]  sigBytes = signature.sign();

        // sprawdzenie podpisu
        signature.initVerify(keyPair.getPublic());

        signature.update(message);

        if (signature.verify(sigBytes))
        {
           System.out.println("Weryfikacja podpisu zakończona powodzeniem.");
        }
        else
        {
           System.out.println("Błąd weryfikacji podpisu.");
        }
     }
  }
```

Po raz kolejny powinien się pojawić komunikat `Weryfikacja podpisu zakończona powodzeniem`.

Jak to działa

Podobnie jak w poprzednim przykładzie, wystarczy tylko zmienić postać wywołania metody `Signature.getInstance()` i dostarczyć odpowiednią parę kluczy, a cała reszta dzieje się w zasadzie sama.

W kodzie można zauważyć coś, czego nie było w poprzednich przykładach z klasą `Signature`, a mianowicie wyraźną strukturę nazwy algorytmu podpisu, opisaną zresztą w dokumentacji JCA wraz z innymi konwencjami nazewniczymi. Nazwa ma format *Skrót*`with`*Szyfr*, tak więc ciąg określający podpis RSA ze skrótem SHA-224 to `SHA224withRSA`, ze skrótem SHA-256 — `SHA256withRSA` i tak dalej.

Podpisy PSS

Metoda RSASSA-PSS, lub krócej PSS, pełni podobną funkcję do dopełnienia OAEP i jest opisana w wersji drugiej standardu PKCS #1. Podobnie jak w przypadku zwykłych podpisów, wymaga ona użycia funkcji skrótu `H()` do obliczenia skrótu podpisywanej wiadomości. Blok danych zawierający skrót jest następnie dopełniany i maskowany funkcją generującą maskę `Mask()`, opartą na funkcji skrótu `H()` z dodatkiem losowej soli *S*. Tak utworzony blok danych jest ostatecznie szyfrowany kluczem prywatnym podpisującego.

Utworzenie podpisu M_s dla wiadomości M z kluczem K obejmuje następujące etapy:

1. M_1 = 0x00 || 0x00 || 0x00 || 0x00 || 0x00 || 0x00 || 0x00 || 0x00 || H(M) || S
2. $M_2 = P$ || 0x01 || S
3. M_3 = Mask(M_2, H(M_1)) || H(M_1) || 0xBC
4. M_s = RSAEncrypt(K, M_3)

P jest w tym przypadku dopełnieniem składającym się z oktetów o wartości 0x00. Długość P wyraża się wzorem:

$$kLen - sLen - hLen - 2$$

gdzie *kLen* jest najczęściej długością bloku, *sLen* jest długością S, a *hLen* jest długością skrótu (wszystkie długości w oktetach).

Sprawdzenie podpisu PSS wymaga odszyfrowania go kluczem publicznym, oddzielenia soli i odtworzenia H(M_1) na podstawie podpisywanej wiadomości. Jeśli odtworzona wartość jest taka sama jak wartość w odszyfrowanym bloku, to podpis jest ważny dla tych danych. W przeciwnym wypadku podpis trzeba odrzucić.

W przeciwieństwie do starszej metody podpisywania obecność losowej soli w procesie generowania podpisu oznacza, że kolejne podpisy generowane tym samym kluczem prywatnym dla tych samych danych będą różne. Jeśli taka przewidywalność jest potrzebna, to można skorzystać z soli o zerowej długości — wtedy wynik obliczeń dla tych samych danych i tego samego klucza prywatnego będzie zawsze taki sam.

Jak widać, PSS znacznie się różni od starszego kodowania metodą PKCS #1, ale ponieważ jest on oparty na algorytmie RSA, więc przejście na PSS wymaga zmiany tylko jednego wiersza w kodzie programu `PKCS1SignatureExample`.

Spróbuj zastąpić wiersz:

```
Signature signature = Signature.getInstance("SHA1withRSA", "BC");
```

wierszem następującym:

```
Signature signature = Signature.getInstance("SHA1withRSAandMGF1", "BC");
```

i już mamy algorytm PSS.

Podobnie jak w przypadku mechanizmu PKCS #1, widać, że nazwa algorytmu dla podpisu PSS również ma określoną strukturę. Format ciągu przypomina nam, że PSS ma wiele wspólnego z OAEP — jest to *Skrót*`withRSAand`*FunkcjaMaski*. O ile jedyną obsługiwaną jak dotąd funkcją generowania maski jest MGF1, o tyle zamiast SHA-1 można używać innych funkcji skrótu, na przykład SHA-256, SHA-384 czy SHA-512. Podając nazwy tych funkcji, trzeba pamiętać o pomijaniu łącznika (-), tak więc SHA-256 zostanie zapisane jako `SHA256`.

Podpisy PSS mogą też przyjmować obiekty parametrów, którym przyjrzymy się w następnej kolejności.

Klasa PSSParameterSpec

Klasa `java.security.spec.PSSParameterSpec` po raz pierwszy pojawiła się w JDK 1.4, gdzie pozwalała zmieniać rozmiar soli używanej przy tworzeniu podpisu. W nowszym JDK 1.5 klasa ta pozwala modyfikować wszystkie dostępne parametry mechanizmu generowania podpisów PSS. Oznacza to, że klasa ma dwa konstruktory, a wybór jednego z nich zależy od parametrów, które chcemy zmienić. Działanie starszego konstruktora mieści się w zakresie działania konstruktora nowszego, więc omówimy tylko ten drugi.

Klasa `PSSParameterSpec` zawiera wartość domyślną, dostępną jako `PSSParameterSpec.DEFAULT`. Stanowi ona odpowiednik ręcznie utworzonego obiektu postaci:

```
PSSParameterSpec defaultSpec = new PSSParameterSpec(
   "SHA-1", "MGF1", MGF1ParameterSpec.SHA1, 20, 1);
```

Przekazanie tak utworzonego obiektu parametrów obiektowi `signature` używającemu PSS wyglądałoby tak:

```
signature.setParameter(defaultSpec);
```

Porównanie konstruktora klasy `PSSParameterSpec` z konstruktorem `OAEPParameterSpec` wykazuje oczywiste podobieństwa. Pierwsze trzy parametry są identyczne: nazwa funkcji skrótu, nazwa funkcji generowania maski i algorytm stosowany przez funkcję generowania maski. Różnią się natomiast ostatnie dwa parametry.

Czwarty parametr to rozmiar soli używanej w procesie generowania podpisu. Starsza wersja konstruktora przyjmuje tylko ten jeden parametr.

Piąty parametr to tak zwane pole końcowe (ang. *trailer field*), informujące generator podpisów, jaki bajt ma się znaleźć na końcu podpisu. Jak dotąd obsługiwana jest tylko jedna wartość pola końcowego — jest to wartość 1, faktycznie odwzorowywana na bajt końcowy podpisu 0xBC. Procesem odwzorowywania zajmiemy się w następnym rozdziale, więc na razie musisz mi uwierzyć na słowo.

Jak sama nazwa wskazuje, obiekty klasy `PSSParameterSpec` są prostymi obiektami wartości, więc jedynymi ich metodami są metody `get()` pobierające wartości poszczególnych parametrów dla danej instancji `PSSParameterSpec`.

Podsumowanie

Tak oto dotarliśmy do końca rozdziału poświęconego podstawowym zagadnieniom kryptografii asymetrycznej. W trakcie rozdziału poznaliśmy podstawy szyfrowania asymetrycznego, wymiany kluczy symetrycznych i tworzenia podpisów cyfrowych, wraz z parametrami algorytmów, których te procesy mogą wymagać. Omówione też zostały algorytmy szyfrujące RSA i El Gamala, algorytmy uzgadniania klucza Diffiego-Hellmana i Diffiego-Hellmana bazującego na krzywej eliptycznej oraz algorytmy podpisów cyfrowych wykorzystujące metody RSA, DSA i DSA oparte na krzywej eliptycznej.

W trakcie tego rozdziału nauczyłeś się:

- tworzyć klucze asymetryczne z obiektów parametrów klucza za pomocą klasy `KeyFactory`,
- tworzyć losowe klucze asymetryczne za pomocą klasy `KeyPairGenerator`,
- przeprowadzać szyfrowanie asymetryczne z wykorzystaniem klasy `Cipher`,
- przeprowadzać procedurę uzgadniania klucza za pomocą klasy `KeyAgreement`,
- tworzyć podpisy cyfrowe z użyciem klasy `Signature`,
- używać obiektów `AlgorithmParameters` z klasami `Cipher` i `Signature`,
- używać klasy `AlgorithmParameterGenerator` do tworzenia obiektów `AlgorithmParameters`.

Co równie ważne, poznaliśmy metody szyfrowania (czyli opakowywania) kluczy symetrycznych kluczami asymetrycznymi, jak również opakowywania kluczy asymetrycznych kluczami tajnymi.

Wspomniałem wcześniej, że zakodowane klucze asymetryczne zawierają nie tylko wartość klucza, ale również sporo informacji opisujących ich strukturę. To samo dotyczy parametrów algorytmów oraz treści niektórych typów podpisów. Informacje te są zapisywane w języku stanowiącym podstawę certyfikatów X.509 oraz licznych protokołów związanych z kryptografią i zarządzaniem certyfikatami. Zanim zagłębimy się w bardziej zaawansowane kwestie kryptograficzne, przydałoby się zatem poznać podstawy tego języka opisu struktury obiektów i tym właśnie zajmiemy się w następnym rozdziale.

Ćwiczenia

1. Kolega próbuje wykorzystać algorytm RSA do wymiany kluczy, ale jego implementacja zawodzi, gdy tylko wiodącym bajtem klucza jest zero. Co powoduje problem? Jak można mu zaradzić?
2. Maksymalna długość danych, jakie można zaszyfrować algorytmami w rodzaju RSA czy El Gamala, jest z reguły ograniczona długością klucza, dodatkowo pomniejszoną o ewentualne bajty dopełnienia. W jaki sposób można wykorzystać jeden z tych algorytmów w procesie szyfrowania danych o dowolnej długości?
3. Uzgadnianie klucza tym się różni od wymiany klucza, że pozwala komunikującym się stronom niezależnie wyliczyć ten sam klucz. Co trzeba dodać do każdej procedury uzgadniania klucza, by była ona bezpieczna?
4. We wcześniejszym rozdziale zobaczyliśmy, że możliwe jest uwierzytelnianie danych za pomocą kodu MAC, ale jego wadą jest konieczność znajomości klucza tajnego przez wszystkie strony mające ten kod sprawdzać. Jaka technika asymetryczna pozwala uniknąć tego problemu? Jaki jej aspekt ułatwia to zadanie?