FAECTOR Workshop Cryptographic Programming

Benne de Weger
b.m.m.d.weger@tue.nl

version 1.11*, March 23, 2021

Contents

*Changes in version 1.1 have to do with support for Java 8. Also advice on paths through the exercises has been added. Changes in version 1.11 are only typos (but essential ones) in Section 2.4.5.

1 Introduction

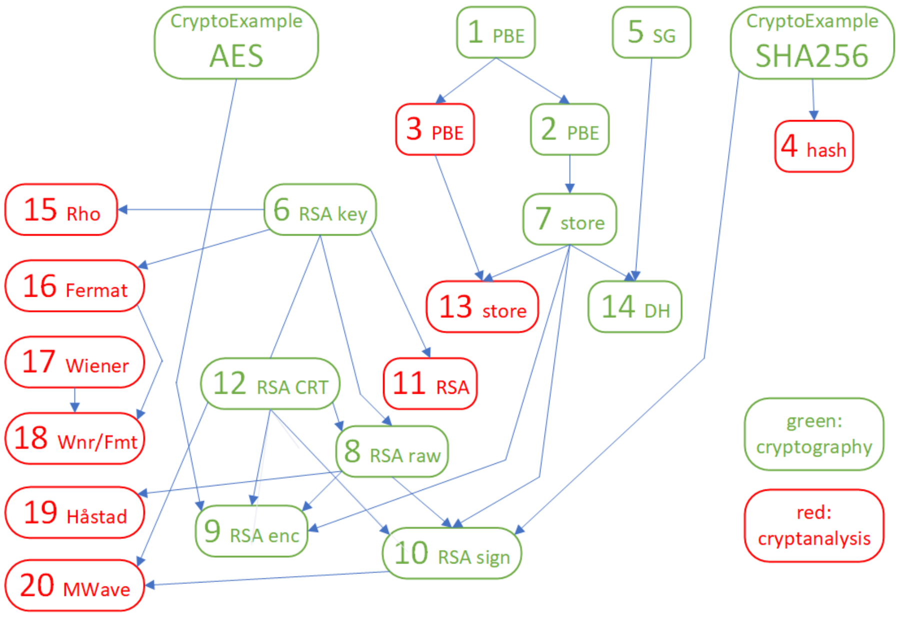
This document provides you with

- a number of programming assignments,
- the necessary mathematical background,
- a description of the provided Java package,
- tips and tricks for programming in Java.

You will be provided with the file `crypto.jar`[2] which contains the Java package `net.deweger.crypto`. This package contains implementations of the cryptographic algorithms AES (symmetric encryption / decryption) and SHA256 (hash function), as well as some convenience utilities.
Goal of the workshop is that you learn some practical, relatively simple, cryptographic and cryptanalytic programming. Subgoals are:

- develop a Java program that does symmetric encryption and decryption of files;
- develop a Java program that computes the SHA256 hash of any file;
- experience how easy it is to program some major asymmetric systems, in particular RSA and Diffie-Hellman;
- experience the importance of using standards: your program should be able to decrypt a file that has been encrypted by your friend's program; your program should be able to use a (public) key that has been generated by your friend's program, etc.;
- program some cryptographic attacks and thus get a feeling for key sizes and weak keys;
- have fun.

There are way too many exercises to do in the only 3 hours of the workshop. There is no need at all to do more than a couple of exercises. Work in groups, divide the work. Pick your favourite topics. Here is an overview of some paths you may want to choose through the assignments.



My advice:

[2]Download it from `https://crypto.deweger.net`.

- Start with 6 and 8; then try 9 and 10 without the secure storage of private keys.
- If you want you can then add 12.
- If you are interested in breaking stuff, then go on with 15, 16, 19, and if you did 12 you can do 20 (maybe the arrow from 10 to 20 should have been from 8 to 20).
- I think 1 and everything that follows from it is more challenging, but worthwile if you have the time.
- The exercises 3, 4, 5, 13, 14 should be doable I think 17 and 18 are really challenging, also from a mathematical viewpoint.

During the workshop I will be available to help you with any problems, be it in understanding the mathematics / cryptography, or in help with programming in Java. When the workshop is over, and you want to continue and need more help, do not hesitate to ask me questions by email.

2 Remarks on Java programming

The workshop assumes that you have a basic understanding of Java. To save time, it is preferable to use simple Java applications (i.e. a Java class that has a `main` method so that it can be seen as a program that can run), and to not spend time on writing nice graphical user interfaces. Of course you can (should) have separate Java classes that do not have a `main` method and that can do specific tasks.

Personally I do not use an IDE (Integrated Development Environment) such as Eclipse; I always write my Java programs on MS Windows, using a very simple character-oriented text editor, and a command window to compile and run the programs. You should of course do what you have experience with.

2.1 Basic number types

A basic concept is that of a `byte`. It is nothing more or less than a sequence of 8 bits, with no further meaning or formatting attached to it. There are $2^8 = 256$ possible values for a byte. The common way to denote them in documentation like this is by “hexadecimal” (hex) notation. The byte is split into two “nibbles” of 4 bits each, and the nibbles then are:

nibble (bits)	hex	numerical value	nibble (bits)	hex	numerical value
`0000`	`0`	0	`1000`	`8`	8
`0001`	`1`	1	`1001`	`9`	9
`0010`	`2`	2	`1010`	`a`	10
`0011`	`3`	3	`1011`	`b`	11
`0100`	`4`	4	`1100`	`c`	12
`0101`	`5`	5	`1101`	`d`	13
`0110`	`6`	6	`1110`	`e`	14
`0111`	`7`	7	`1111`	`f`	15

Cryptographic operations today usually work on (arrays of) bits but for practical purposes one usually works with whole bytes (byte arrays). Byte arrays can be made readable in “hex strings”: the byte array {`01, 9a, ef`} corresponds to the hex string `"019aef"`, and represents the bitstring `0000 0001 1001 1010 1110 1111`. Hex strings always have an even number of characters.

But for the more mathematically oriented asymmetric cryptographic systems one has to work with integers. The Java basic types `int` (often 4 bytes = 32 bits) and `long` (often 8 bytes = 64 bits) may be good enough for most practical purposes; for cryptographic applications however we need to work with much larger integers, sometimes up to 4096 bits (512 bytes). Luckily Java has built-in support for this in the `BigInteger` class. Then we need often conversion from a byte array to a BigInteger and back, and this will be explained below.

Copying bytes from one byte array to another can be done by
`System.arraycopy(source, source_offset, destination, destination_offset, length)`,
this copies from the `byte[] source`, starting at byte `source_offset`, a total of `length` bytes, and puts them into the `byte[] destination`, starting at byte `destination_offset`.

2.2 The provided package `net.deweger.crypto`

The Java package `net.deweger.crypto`, provided in the file `crypto.jar` that can be downloaded from `https://crypto.deweger.net`, has implementations of the symmetric encryption algorithm AES, and the hash function SHA256.

The class `net.deweger.crypto.AES` supports AES encryption and decryption with 128-bit keys only, with different "modes of operation". On my PC this program's AES speed is 10 MB/sec.

The class `net.deweger.crypto.SHA256` supports SHA256 hashing, with feeding data all at once and in parts, and also supports the "Message Authentication Code" HMAC-SHA256. On my PC this program's SHA256 speed is 45 MB/sec.

There is also the class `net.deweger.crypto.Util` which provides easy and secure randomness as well as conversion methods supporting readability of byte arrays.

Below is a brief description of the API. To import the package in your own Java program, use

```
import net.deweger.crypto.*;
```

When running your own Java class `myCryptoClass` that uses the package, the classpath should be set accordingly, e.g. via the commandline call

```
java -cp crypto.jar;. myCryptoClass
```

It is good practice to put all calls inside a `try - catch` construction, as many methods from the `net.deweger.crypto` package may throw exceptions.

The class `CryptoExample` provides simple usage examples. The source code is provided in the Appendix.

2.2.1 The class `net.deweger.crypto.AES`

Public static fields (supported modes of operation):

`MODE_ECB`: ECB mode (Electronic CodeBook), does not require an IV

`MODE_CBC`: CBC mode (Cipher Block Chaining), requires a 16 byte IV

`MODE_CFB`: CFB mode (Cipher FeedBack), requires a 16 byte IV

`MODE_OFB`: OFB mode (Output FeedBack), requires a 16 byte IV

`MODE_CTR`: CTR mode (CounTeR), requires a 16 byte IV (actually uses only the first 8 bytes of the IV, the remaining 8 bytes are overwritten with the counter)

Modes of operation prescribe how multiple blocks are "chained". Counter mode is recommended in most situations.

Public static fields (supported padding schemes):

`PAD_NONE`: no padding (should only be used when plaintext size is a multiple of 16 bytes)

`PAD_PKCS7`: PKCS#7 padding (appends at least one bye, each byte is equal to the total number of padded bytes)

`PAD_BIT`: Bit padding (appends one `1`-bit and then only `0`-bits)

Padding is used to bring the message length up to a multiple of the AES blocksize (16 bytes). PKCS#7-padding is recommended.

Public methods:

`public AES() throws Exception`
: constructor

`public byte[] encrypt(byte[] plaintext) throws Exception`
: returns the encryption of the provided plaintext

`public byte[] decrypt(byte[] ciphertext) throws Exception`
: returns the decryption of the provided ciphertext

`public byte[] generateKey() throws Exception`
: generates, sets and returns a random key

`public void setKey(byte[] key) throws Exception`
: sets a key (16 bytes)

`public byte[] getKey() throws Exception`
: returns the current key

`public byte[] generateIv() throws Exception`
: generates, sets and returns a random iv

`public void setIv(byte[] iv) throws Exception`
: sets an iv (16 bytes), not needed in ECB mode

`public byte[] getIv() throws Exception`
: returns the current iv

`public void setMode(int mode) throws Exception`
: sets a mode, should be one of the `MODE_xxx` fields, default mode is `MODE_ECB`

`public int getMode() throws Exception`
: returns the current mode

`public void setPadding(int padding) throws Exception`
: sets a padding, should be one of the `PAD_xxx` fields, default padding is `PAD_NONE`

`public int getPadding() throws Exception`
: returns the current padding

See the Appendix for source code of `CryptoExample.java` for simple usage examples.

2.2.2 The class `net.deweger.crypto.SHA256`

Public methods:

`public SHA256()`
: constructor

`public byte[] hash(byte[] data) throws Exception`
: returns the hash of the provided data in “one pass”

`public byte[] hMac(byte[] key, byte[] message) throws Exception`
: returns the HMAC-SHA256 of a key and a message

`public void hashInit() throws Exception`
: initializes a hash computation

`public void hashUpdate(byte[] data) throws Exception`
: feeds data to the hash computation, can be called multiple times

`public byte[] hashFinal() throws Exception`
: finalizes a hash computation, returns the hash

`public byte[] hashFinal(byte[] data) throws Exception`
: feeds data to the hash computation, then finalizes it, returns the hash

See the Appendix for source code of `CryptoExample.java` for simple usage examples.

2.2.3 The class `net.deweger.crypto.Util`

Public fields:

`public java.security.SecureRandom secureRandom`
: an instance of the class `java.security.SecureRandom`

Public methods:

`public Util()`
: constructor; instantiating this class is only required if the method `randomBytes` is to be used

`public byte[] randomBytes(int n)`
: returns `n` random bytes

`public static String byteToHex(byte b)`
: returns a hex string from one byte

`public static String bytesToHex(byte[] b)`
: returns a hex string from a byte array

`public static byte[] hexToBytes(String h) throws Exception`
: returns a byte array from a hex string

`public static String bytesToHex64(byte[] b)`
: returns a hex string from a byte array, with a newline after every 64 characters

`public static byte[] hexToBytes64(String h) throws Exception`
: returns a byte array from a hex string, ignoring newline characters

`public static byte[] bigIntegerToByteArray(BigInteger bi) throws Exception`
: returns a byte array from a BigInteger, with leading zero byte removed

`public static String bigIntegerToHex64(BigInteger bi) throws Exception`
: returns a hex string from a BigInteger, with newline after every 32 bytes (64 characters), and leading zero byte removed

`public static BigInteger squareRoot(BigInteger d) throws Exception`
: (*only available in the Java 8 version of* `crypto.jar`*; with a Java* ≥ 9 *version the* `BigInteger` *class has the method* `sqrt` *that can be used instead*)
returns $\left\lfloor \sqrt{d} \right\rfloor$, i.e. the integer part of $\sqrt{d}$ (uses Newton's method for $x^2 - d = 0$)

`public static BigInteger thirdRoot(BigInteger d) throws Exception`
: returns $\sqrt[3]{d}$, but only for an exact third power (uses Newton's method for $x^3 - d = 0$)

See the Appendix for source code of `CryptoExample.java` for simple usage examples.

2.3 Dealing with files

2.3.1 Raw byte files

A file containing "raw bytes" can be any file that has to be encrypted or decrypted, of which the content not necessarily can be understood, by inspection or by using some software.

```
import java.nio.file.*;
```

All file operations may throw `IOExceptions`. Below `fileName` is a String, `bytes` a `byte[]`.

Reading a "raw byte" file into a Java byte array:

```
    bytes = Files.readAllBytes(Paths.get(fileName));
```

Writing a Java byte array to a "raw byte" file:

```
    Files.write(Paths.get(fileName), bytes);
```

Study the source code of `CryptoExample.java` in the Appendix.

2.3.2 Tag-value formatted files

[[This section assumes you are familiar with Java `ArrayLists`. If that is not the case, it should be not too difficult to adapt it to e.g. old-fashioned arrays.]]

Some files can have a structured human-readable content according to a “tag-value” format, they will be used for storing cryptographic keys and certificates. These files can be read by any basic text editor such as Notepad. For example, an RSA public key may be formatted as follows:

```
[id] (identity)
www.eur.nl
[n] (modulus)
    ca04809b5975ad6a9e5851f10eb075c1a70cce85cee1399c48c835de68a575ac
    b049327180edf4c52d5eda1e0642e0db697cb18fce5e4f817c1d17b82ce1094d
    38bdf36d5d16dd2330e0abd1054e45c5333521a8f08003c8b5d90099e7240588
    d865b6801126848621430fdab643ffeeddee9d57c8ed33ab8b24fbd3414e6225
    d3418efcdf5624be8217e0326baa24244fdc345df932b2808aa8cf765cb7ffd8
    f2a3502d5535eddae3a4030f88fa03169cbd1f5c61881dcf3661427d781123c5
    5921ce0e3ffc9e2d11e5f994154738746b6fb5284a3607ec0d2f56dfd82d185e
    4da74043e83b6c313f1d2a992f2ec5ca94abd695f4a782888584d204ee49831f
    b09b73e57a64fb4cd95198689fecd13b0eedd87ef78eb362097ca17af46af9b2
    7097ee2a4c0f660c9244ab00d2fdb80f6b3ccd8fff56f25599442ff0727715a1
    99c68712cb2eccfbe8b360e7556ff03fa73a4b68ce6fc152521dd8817030f69c
    7736f8ecc22c48c6ead741289d8e504d19e921e98c95b1cf786dc116723f7bc9
    71cd5a00dd91845dd73e87774fa88f8b9dd40be2b5ebbfdc63e969ee690d4e6a
    4da0a2e4996a87bef4d122205e4a3d0a00a9ee9276406b63dc8a7b4215b900ba
    a6941c4696d860841ca87441d4497329ca17631216b1567a8d4a2b957cf8cccf
    2fa415639bb7cccfbfd698466a25385cd27e13c27edd476214bca4705ccf8e11
[e] (public exponent)
    010001
```

(this, by the way, is indeed the public key of the website `www.eur.nl`). Here, the tags `id`, `n`, `e` are between square brackets, the data on the tag line after the tag are comments only that should be ignored by any program, the lines until the following tag or the end of the file are the value, where spaces and newlines are to be ignored. So the 16 lines after the tag `[n]`, with each 64 hexadecimal characters, are to be seen as one string of hexadecimals, length $16 \times 64 = 1024$, representing the $1024 \times 4 = 4096$ bits of the modulus (an integer between 2^{1023} and 2^{1024}).

You will be provided now with Java code to read and write those (tag,value) pairs. Usually the following imports are needed:

```
import java.io.*;
import java.nio.file.*;
import java.nio.charset.StandardCharsets;
import java.util.List;
import java.util.ArrayList;
```

All file operations may throw `IOExceptions`. Below `fileName` is a String, `tagValue` an `ArrayList<String[]>` in which each element is a string of length 2.

Reading and parsing a “tag-value” formatted file (name `fn`) via a Java list of (tag,value) pairs into BigIntegers:

```
ArrayList<String[]> key = new ArrayList<String[]>();
List<String> lines = Files.readAllLines(Paths.get(fn));
String[] tagValue = new String[2];
boolean first = true;
for(String line: lines)
{
    if (line.startsWith("["))
    {
        if (first)
            first = false;
        else
            key.add(tagValue);
        tagValue = new String[2];
        tagValue[0] = line;
        tagValue[1] = "";
    }
    else
        tagValue[1] += line.trim();
}
key.add(tagValue);

for (String[] s: key)
{
    if (s[0].startsWith("[a]"))
        a = new BigInteger(s[1], 16);
    if (s[0].startsWith("[b]"))
        b = new BigInteger(s[1], 16);
}
```

Writing BigIntegers via a Java list of (tag,value) pairs to a “tag-value” formatted file (name `fn`):

```
ArrayList<String[]> key = new ArrayList<String[]>();
String[] tagValue = new String[2];
tagValue[0] = "[a] (the number a)";
tagValue[1] = Util.bigIntegerToHex64(a);
key.add(tagValue);
tagValue[0] = "[b] (the number b)";
tagValue[1] = Util.bigIntegerToHex64(b);
key.add(tagValue);
List<String> lines = new ArrayList<String>();
for (String[] item: key)
    lines.add(item[0] + "\n" + item[1]);
Files.write(Paths.get(fn), lines, StandardCharsets.UTF_8, StandardOpenOption.CREATE,
    StandardOpenOption.TRUNCATE_EXISTING);
```

2.4 The class `java.math.BigInteger`

This section provides an overview (not exhaustive) of the useful methods of the class `BigInteger`. Do not forget to import the class:

```
     import java.math.BigInteger;
```

2.4.1 Static fields:

`BigInteger.ZERO`: 0

`BigInteger.ONE`: 1

`BigInteger.TWO`: 2 (*not available in Java version 8*)

`BigInteger.TEN`: 10

2.4.2 Constructors from a byte array

`new BigInteger(byte[] b)`
: the byte array `b` is interpreted as two's-complement binary representation (this means that if the byte array of n bytes is interpreted as the binary representation of an integer $a \in [0, 2^{8n} - 1]$, then the byte array should be interpreted as the representation $\pmod{2^{8n}}$ in the interval $[-2^{8n-1}, 2^{8n-1} - 1]$; examples: `00` means 0, `01` means 1, `7e` means 126, `7f` means 127, `80` means −128, `81` means −127, `fe` means −2, `ff` means −1)
[[All this is tricky and a source for many bugs; this method is to be avoided, and the next method is preferred.]]

`new BigInteger(int sign, byte[] b)`
: the byte array `b` is interpreted as the binary representation of the absolute value, `sign` is ± 1

2.4.3 Constructors from a string

`new BigInteger(String s)`
: `s` is a decimal string representation

`new BigInteger(String s, int r)`
: `s` is a radix `r` string representation (here $2 \leq \texttt{r} \leq 36$), very useful with $r = 16$ for printing out hexadecimal strings

2.4.4 Constructors for random big integers

`new BigInteger(int bitlength, int certainty, Random rnd)`
: creates a random probable prime of given `bitlength`,
the probability that the result is not prime is less than $2^{-\texttt{certainty}}$
note: this can be quite slow when `bitlength` or `certainty` are very large;
`certainty` = 100 is certainly secure and not too big

`new BigInteger(int bitlength, Random rnd)`
: creates a random integer of given `bitlength`, uniformly sampled from $[0, 2^{\texttt{bitlength}} - 1]$

2.4.5 Methods for integer arithmetic (a, b are `BigIntegers`)

`b.add(a)`
: returns $\texttt{a} + \texttt{b}$

`b.subtract(a)`
: returns $\texttt{b} - \texttt{a}$

`b.negate()`
: returns $-\texttt{b}$

`b.multiply(a)`
: returns $\texttt{a}\cdot\texttt{b}$

`b.divide(a)`
: returns the integer part of `b/a` (watch out with negative numbers)

`b.divideAndRemainder(a)`
: returns the integer part and remainder of `b/a` (watch out with negative numbers)

`b.gcd(a)`
: returns $\gcd(\texttt{a}, \texttt{b})$

`b.remainder(a)`
: returns the remainder of `b/a` (watch out with negative numbers)
for positive `a`, `b` this is the same as `b.mod(a)`

`b.sqrt()`
: (*not available in Java version 8, for this reason* `net.deweger.crypto.Util` *has a method* `squareRoot` *which is only available in the* `crypto.jar` *for Java version 8*)
returns the integer part of the square root of `b`

`b.bitLength()`
: returns the bitlength of `b` (rounded logarithm to the base 2)

`b.shiftLeft(n)`
: returns $\mathtt{b} \times 2^{\mathtt{n}}$ (adds `n` `0`-bits to the end)

`b.shiftRight(n)`
: returns the integer part of $\mathtt{b}/2^{\mathtt{n}}$ (removes the last `n` bits)

2.4.6 Methods for modular arithmetic (b, k, m are `BigIntegers`)

`b.mod(m)`
: returns representation of `b` (mod `m`) in $[0, \mathtt{m}-1]$

`b.modInverse(m)`
: returns $\mathtt{b}^{-1}$ (mod `m`) (throws an exception if $\gcd(\mathtt{b},\mathtt{m}) \neq 1$)
the same as `b.modPow(BigInteger.ONE.negate(), m)` but may be slightly faster

`b.modPow(k, m)`
: returns $\mathtt{b}^{\mathtt{k}}$ (mod `m`) (note: `k` is also a `BigInteger`)
note: this is a very efficient method, contrary to first computing $\mathtt{b}^{\mathtt{k}}$ as an integer and then reducing (mod `m`), which is horribly inefficient already for small values of `b`, `k` and `m`, and totally impossible for somewhat bigger values.

Other forms of modular arithmetic, such as addition ($\mathtt{a} + \mathtt{b}$ (mod `m`)), can be done as `b.add(a).mod(m)`, etc.

2.4.7 Methods for comparing

`b.compareTo(a)`
: returns -1 if $\mathtt{b} < \mathtt{a}$, 0 if $\mathtt{b} = \mathtt{a}$, 1 if $\mathtt{b} > \mathtt{a}$

`b.equals(a)`
: returns boolean, true iff $\mathtt{b} = \mathtt{a}$

2.4.8 Methods for prime numbers

`b.isProbablePrime(certainty)`
: returns boolean, if false then `b` is definitely composite, if true then `b` is prime with probability $> 1 - 2^{-\mathtt{certainty}}$, $\mathtt{certainty} = 100$ is certainly secure and not too big

`BigInteger.probablePrime(bitLength, rnd)`
: static method, same as the constructor `new BigInteger(bitLength, 100, rnd)`

2.4.9 Conversion methods (see also the constructors)

`BigInteger.valueOf(x)`
: returns a `BigInteger` with the value of the `int` (or `long`) `x`

`b.toByteArray()`
: returns a byte array containing the two's-complement representation (so be careful; if you have a number between 2^{8n-1} and 2^{8n} you will get an additonal zero byte at the front before the n expected bytes

`b.toString()`
: returns a string with the decimal representation

`b.toString(r)`
: returns a string with the radix `r` representation ($2 \leq$ `r` ≤ 36) note: with `r` $= 16$ this may give a string with odd length, if you want a hex string where each byte is represented with two characters, use `Util.bytesToHex(b.toByteArray())`

3 Password-based Encryption

3.1 Key Derivation

It is possible to derive cryptographic keys from passwords. A popular method to do so is `PBKDF2` (Password Based Key Derivation Function). It can be based on HMAC-SHA256[12], and for AES keys it then works as follows (here some parameters are preset, a general description is a bit more involved).

The function `PBKDF2` takes 3 arguments: a string `password`, a byte array `salt`, used to make (rainbow) table lookups more difficult (the salt is not secret), and an integer `iterationCount`, that is small enough to make a one-time key derivation still quick, but becomes a problem for a brute force attacker, who has to try many combinations of `password`s and `salt`s.

The function computes $u_1, u_2, \ldots, u_{\texttt{iterationCount}}$ recursively:
u_0 is `salt` with four bytes `00000001` appended to it,
$u_i = \texttt{HMAC-SHA256}(\texttt{password}, u_{i-1})$ for $i = 1, 2, \ldots, \texttt{iterationCount}$.

The output then is $u_1 \oplus u_2 \oplus \ldots \oplus u_{\texttt{iterationCount}}$ ($\oplus$ stand for bitwise XOR).

This gives a 256-bit (32-byte) pre-key, an AES key is then derived as the first 128 bits (16 bytes).

3.2 Encryption and Decryption

Assignment 1: Write a Java class PBE that provides a method `PBKDF2`, and methods that provide password-based encryption and decryption with AES, in counter mode with PKCS#7 padding, and with an all-zero IV (you can also take a random IV but then it has to be communicated with the ciphertext). Choose a not too large salt (you can e.g. use a "username" and convert it into a byte array), and choose as iteration count e.g. 1000. Clearly the recipient of the ciphertext should use the same settings to decrypt as the sender has used to encrypt the plaintext. The encrypt and decrypt should operate on byte arrays (i.e. plaintext and ciphertext are byte arrays). The password must be a string.

Assignment 2: Write a Java program that can read in any (raw byte) file, and encrypt / decrypt it with your class PBE. You should be able to use it to exchange encrypted files by email, or WhatsApp, or whatever, so that your communication partner can use her own implementation. Hint: it may make sense to let the ciphertext file contain not raw bytes but hex strings.

Assignment 3: Assume a plaintext consisting only of readable ASCII symbols (bytes `0a`, `0d` and `20` – `7e`) has been encrypted by PBE with a short password consisting of only letters (upper case and lower case) and digits (bytes `30` – `39`, `40` – `59`, `60` – `79`), and with salt "`FAECTOR`". Write a program that tries to find the AES key from just the ciphertext. Experiment with it.

4 Hashing

`SHA256` is an example of a cryptographic hash function. It takes as input any bit string of any length, and returns a 256-bit hash with the following requirements:

[12]HMAC is a Message Authentication Code; basically this is a hash function for messages that is used in combination with a key; sort of a symmetric digital signature.

Preimage resistance: given any 256-bit `h`, it is computationally infeasible to find an input `m` such that $\mathtt{SHA256(m)} = \mathtt{h}$.

Second-preimage resistance: given any 256-bit $\mathtt{m}_1$, it is computationally infeasible to find an input $\mathtt{m}_2$ such that $\mathtt{m}_2 \neq \mathtt{m}_1$ and $\mathtt{SHA256}(\mathtt{m}_2) = \mathtt{SHA256}(\mathtt{m}_1)$.

Collision resistance: It is computationally infeasible to find two inputs $\mathtt{m}_1, \mathtt{m}_2$ such that $\mathtt{m}_1 \neq \mathtt{m}_2$ $\mathtt{SHA256}(\mathtt{m}_1) = \mathtt{SHA256}(\mathtt{m}_2)$.

The differences between those requirements are subtle. We will explore them with a truncated version of `SHA256`, for convenience.

Assignment 4: Consider for $n < 256$ the hash function $\mathtt{SHA}_n$ that simply is `SHA256` truncated at n bits ($n/8$ bytes, it is convenient to take n a multiple of 8).
Write a program that searches for a preimage / second preimage for $\mathtt{SHA}_{24}$ by simply trying random inputs, and record how many $\mathtt{SHA}_{24}$ hashes had to be computed until one preimage / second preimage was found.
Also write a program that searches for a collision by storing all pairs $(\mathtt{m}, \mathtt{SHA}_{24}(m))$ (it makes sense to sort the list for the hash value), and again record how many $\mathtt{SHA}_{24}$ hashes had to be computed until one collision was found. If you want to do this collision search in a memory-efficient way, search on the Internet for Floyd's Cycle Finding Algorithm[a].
Do these experiments a number of times so that you get a fair idea on probable average counts. Is the result surprising? If you want to understand, search the Internet for the "Birthday Paradox".
If you have a lot of patience you could also try experiments for $\mathtt{SHA}_{32}$.

[a]In Section 9.1.2 on the Pollard Rho method you will meet a somewhat simpler example of Floyd's method.

5 Prime number generation

There are efficient prime number generation methods, that can easily produce random primes of several thousands of bits. In practice one uses a pretty simple method: just generate a random integer of the desired bitlength, and apply a primality test to it; if it fails, start all over.

However, the only really practical primality tests are probabilistic, in the sense that they either detect compositeness with mathematical certainty (and without a factorization!), or probable primality. Parameters can be chosen such that the probability of a composite number detected as probable prime is so low that it won't happen before the sun burns down.

Java offers `BigInteger.probablePrime()` (prime generation) and `BigInteger.isProbablePrime()` (a probabilistic primality test). This is very useful for asymmetric cryptographic algorithms such as RSA and Diffie-Hellman.

A prime number p is called a *Sophie Germain* prime when it is > 5 and also $\frac{1}{2}(p-1)$ is prime. This type of primes is especially useful for Diffie-Hellman.

Assignment 5: Write a Sophie Germain prime generator that produces a random Sophie Germain prime of n bits.
Hint: start with generating an $(n-1)$-bit prime q, and then test whether $p = 2q + 1$ is prime.
Test your program for $n = 256, 512, 1024$. It may take some time (this will vary wildly); if you are patient enough you may try $n = 2048$.

6 RSA

6.1 Key Pair Generation

To generate an RSA key pair with an n-bit modulus, do the following:

1. generate, independently, two primes p, q of $\lceil n/2 \rceil$ bits; and compute $n = pq$, you may want to reject it if its bitsize is one off;

2. generate an e with $2 < e < \phi(n) = (p-1)(q-1)$, such that $\gcd(e, \phi(n)) = 1$; most used in practice is $e = 65537$, it is not bad to always choose this;
3. compute $d \equiv e^{-1} \pmod{\phi(n)}$;
4. store n, e as public key, and n, d as private key;
5. forget $p, q, \phi(n)$.

It is allowed to first choose d and then compute e. What also is often done is to first choose e (especially if one wants $e = 65537$), and then generate the primes and the modulus, and then d.

To store your keys, it is recommended to use the following format (`123...` stands for a hex string that may use multiple lines):

private key:

```
[id] (identity)
your-name
[n] (modulus)
123...
[d] (private exponent)
123...
```

public key:

```
[id] (identity)
your-name
[n] (modulus)
123...
[e] (public exponent)
123...
```

You may want to store the value of e also with your private key. The identity is useful but optional.

Assignment 6: Write a Java class that does RSA key pair generation and stores public and private keys in the given format in separate files on the hard disk.

6.2 Secure Private Key Storage

Assignment 7: Use PBE to encrypt / decrypt private key files under a password (see Assingnmenmt 2). Incorporate this in the RSA Key Pair Generation program of Assignment 6.

6.3 Encryption and Decryption

So called "raw RSA" (or "textbook RSA") for encryption works as follows:

Encoding: encode the plaintext as an integer m with $0 \leq m < n$;

Encryption: given m, the ciphertext is computed as the number $c \equiv m^e \pmod{n}$;

Decryption: given c, the plaintext is computed as the number $m \equiv c^d \pmod{n}$;

Decoding: decode the integer m into the plaintext.

Encoding and decoding usually is as simple as reading a byte array as an integer in radix $2^8 = 256$.

It is always wise to randomize plaintext before encrypting. As usually only small symmetric keys are encrypted, there is enough space "below" the RSA modulus to add randomness. This process is called "padding".

With RSA the use of PKCS#1v1.5-padding can be recommended. For a given message M (actually, a symmetric key) of b bytes (for AES, $b = 16$), with an RSA modulus of n bits, so that the number of bytes is $\lceil n/8 \rceil$, this can be done when $b \leq \lceil n/8 \rceil - 11$, as follows[15]:

[15]The notation $\|$ means "concatenation".

$m = \texttt{0002}\|PS\|\texttt{00}\|M$
where PS consists of random nonzero (this is important!) bytes, at least 8, but as many as one wants, preferably $\lceil n/8\rceil - b - 3$. Then m is to be interpreted as an integer with $0 < m < n$, serving as the raw-RSA plaintext.

At decryption this padding has to be removed, which can be done unambiguously. It is good practice to verify if the correct padding format is found, and to reject the decryption if an error is found.

The randomness should (of course!) be generated freshly at encryption time.

Assignment 8: Write a Java class that implements RSA encryption and decryption using PKCS#1v1.5-padding, only encrypt byte arrays that could contain AES keys.

Hybrid encryption uses a symmetric encryption method such as AES to encrypt the plaintext message, with a random key, that is used only for that particular message. The key then is encrypted by RSA, and the AES-encrypted message and RSA-encrypted AES-key are then sent to the recipient.

Assignment 9: Write a Java program that implements hybrid encryption and decryption of any file with RSA and AES, with RSA private keys accessible through a password only. Use PKCS#1v1.5-padding. You will need the recipient's public key to encrypt.
Write your code in such a way that your communication partner and you can understand the files that you exchange, each working with your own developed programs.

6.4 Signature Generation and Verification

For signatures "raw RSA" (or "textbook RSA") works as follows:

Hash: compute the hash h of the to-be-signed message, and interpret it as an integer with $0 \leq h < n$;

Signature Generation: given h, the signature is computed as the number $s \equiv h^d \pmod{n}$;

Signature Verification: given h and s, the signature is found to be valid if the number $s^e \pmod{n}$ is equal to h.

Also for digital signatures a padding scheme is to be recommended. The PKCS#1v1.5-padding here works as follows (note that no randomness is involved here) (here we describe it only for the hash function SHA256).

For a given hash H of 32 bytes, with an RSA modulus of n bits, so that the number of bytes is $\lceil n/8\rceil$, this can be done when $\lceil n/8\rceil \geq 62$, as follows:
$h = \texttt{0001}\|PS\|\texttt{00}\|AI\|H$
where PS consists of bytes with value `ff`, at least 8, but as many as one wants, preferably $\lceil n/8\rceil - 54$, and AI is the algorithm identifier, for SHA-256 this is the 19-byte value
`3031300d060960864801650304020105000420`.
Then h is to be interpreted as an integer with $0 < h < n$, serving as the input for the private key operation.

At verification this padding has to be removed, which can be done unambiguously. It is good practice to verify if the correct padding format is found, and to reject the signature if an error is found.

Assignment 10: Write a Java program that implements digital signatures of any file with RSA and SHA256, with RSA private keys accessible through a password only. Use PKCS#1v1.5-padding. The recipient will need your public key to verify your signatures.
Write your code in such a way that your communication partner and you can understand the files that you exchange, each working with your own developed programs.

6.5 Factoring the modulus from the public and private keys

From knowledge of only n, e, d it is possible to recover the prime factors of n. This works as follows.

1. Compute $ed - 1$, and write it as $ed - 1 = 2^s t$ for an $s > 0$ such that t is odd (i.e. pull out all factors 2).
2. Choose some integer $a \geq 2$ (may be random, but may just as well be some small number such as $a = 2, 3$, etc.).
3. Compute $x_0 \equiv a^t \pmod{n}$, and then start squaring it (modulo n), so compute $x_1 \equiv x_0^2 \pmod{n}$, $x_2 \equiv x_1^2 \pmod{n}$, $x_3 \equiv x_2^2 \pmod{n}$, etc. In fact you compute $x_i \equiv a^{2^i t} \pmod{n}$, so $x_s \equiv a^{ed-1} \equiv 1 \pmod{n}$ (because of Euler's Theorem).
4. If in the sequence there is an index i for which $x_i \equiv 1 \pmod{n}$ and $x_{i-1} \not\equiv \pm 1 \pmod{n}$, then the two prime factors of n are found as $\gcd(x_{i-1} - 1, n)$ and $\gcd(x_{i-1} + 1, n)$. This will happen with probability 50%.
5. If you had bad luck, try another value for a (but not a power of a value tried before; it is advisable do try e.g. the smallest few prime numbers $a = 2, 3, 5, 7, \ldots.$).

In practice this works pretty good.

Note that this implies that RSA moduli should never be shared: two persons with each a key pair with identical RSA moduli but different values for e can recover each other's private key d.

Assignment 11: Write a Java progran that implements recovering the private key of a second person who has a keypair with the same modulus as you have, but with a different value for e.

6.6 RSA-CRT

RSA-CRT is a method to speed up decryption (encryption can already be sped up by choosing a small e). Instead of working with the full modulus n it works using its prime factors p, q (which are half the length so save a lot of time in computations), and afterwards recombining results modulo p, q into the result $\pmod{n}$ (this recombination step is based on the "Chinese Remainder Theorem", hence the acronym CRT).

It works as follows. At key pair generation, instead of storing the private exponent d, one actually stores the values of $p, q, d_p \equiv d \pmod{p-1}, d_q \equiv d \pmod{q-1}, u = q \pmod{p}$, all of half the bitlength of n.

CRT private key storage:

```
[id] (identity)
your-name
[p] (first prime)
123...
[q] (second prime)
123...
[dp] (d (mod p-1))
123...
[dq] (d (mod q-1))
123...
[u] (q^{-1} (mod p))
123...
```

The public key and the encryption process are the same as for "ordinary" RSA. But decryption is done differently:

Decryption modulo primes: Compute $m_p \equiv c^{d_p} \pmod{p}$ and $m_q \equiv c^{d_q} \pmod{q}$;

Recombination: Compute $m \equiv m_q + qu(m_p - m_q) \pmod{n}$.

Because of Fermat's Theorem it suffices to have d modulo $p-1$ and $q-1$ only. Note that the recombination formula is carefully constructed to have $m \equiv m_p \pmod{p}$ and $m \equiv m_q \pmod{q}$, using that $qu \equiv 1 \pmod{p}$. Instead of storing u one could also store qu.

Assignment 12: Incorporate RSA-CRT in the earlier implemented "raw RSA" classes.

For your information, some average speeds on my PC:

bitsize	1024	2048	4096
key pair generation	0.039 sec.	0.295 sec.	2.933 sec.
encryption with $e = 65537$	0.00004 sec.	0.00009 sec.	0.00034 sec.
encryption with random big e	0.0017 sec.	0.0105 sec.	0.0799 sec.
decryption with CRT	0.0006 sec.	0.0033 sec.	0.0228 sec.
encryption without CRT	0.0015 sec.	0.0107 sec.	0.0804 sec.

Absolute figures do not say much (except about the suboptimality of my implementation); but comparing different figures gives a nice view of the advantages of $e = 65537$ and CRT, and of the behaviour w.r.t. modulus size. Key pair generation times vary wildly so should be taken with a grain of salt (the figures given were averaged over 100 key pair generations and 1000 encryptions / decryptions). And a comparison with AES speed is interesting.

7 A Brute Force attack on Password-based Encryption

Assignment 13: See Assignment 3. A similar attack can be done on an RSA private key file if it has been protected with PBE. Use your program from Assignment 3 to recover an RSA private key that had been protected with a way too short password.

8 Discrete Logarithm Based methods

8.1 Diffie-Hellman Key Exchange

The Diffie-Hellman key exchange protocol is a cute alternative for RSA for encrypting a symmetric key: Alice and Bob can compute a shared secret upon communicating only public keys (strictly speaking Diffie-Hellman does not do encryption, while RSA does). It is based on a different hard problem than factoring, namely the "Discrete Logarithm Problem" (DLP). For a big prime number p, we work in the multiplicative group $\mathbb{Z}_p^* = \{1, 2, \dots, p-1\}$ with multiplication $\pmod{p}$. Let $g \in \mathbb{Z}_p^\star$ a fixed element, the "generator" ($g \not\equiv \pm 1 \pmod{p}$), and what it generates is its powers $\{1 = g^0, g = g^1, g^2, g^3, \dots, g^{e-1}\}$, forming a "cyclic subgroup" of $\mathbb{Z}_p^*$ of order e, where e is the smallest positive integer for which $g^e \equiv 1 \pmod{p}$. Then it is always true that $e | p-1$. One wants this e to be pretty big, and that is why Sophie Germain primes are useful: then $p-1$ has only one nontrivial divisor, namely $\frac{1}{2}(p-1)$, so then either $e = \frac{1}{2}(p-1)$ or $e = p-1$, which is good.

Alice and Bob somehow agree on p and g. These numbers are "system parameters" and they are not secret.

The DLP is the problem of, given a value $y \equiv g^x \pmod{p}$, to determine what x is from knowing p, g and $y \equiv g^x \pmod{p}$. This problem is approximately as hard as factoring, for similar bitsizes.

The Diffie-Hellman protocol is as follows:

- Alice takes a random private key $x_A \in \{2, 3, \dots p-2\}$ and computes the corresponding public key $y_A \equiv g^{x_A} \pmod{p}$.
- Bob takes a random private key $x_B \in \{2, 3, \dots p-2\}$ and computes the corresponding public key $y_B \equiv g^{x_B} \pmod{p}$.
- Alice and Bob exchange public keys.

- Alice computes $s_{\mathrm{A}} \equiv y_{\mathrm{B}}^{x_{\mathrm{A}}} \pmod{p}$.
- Bob computes $s_{\mathrm{B}} \equiv y_{\mathrm{A}}^{x_{\mathrm{B}}} \pmod{p}$.

Note that only public information is exchanged, and when the protocol has finished, Alice and Bob share the same secret, since

$$s_{\mathrm{A}} \equiv y_{\mathrm{B}}^{x_{\mathrm{A}}} \equiv (g^{x_{\mathrm{B}}})^{x_{\mathrm{A}}} = g^{x_{\mathrm{B}}x_{\mathrm{A}}} = g^{x_{\mathrm{A}}x_{\mathrm{B}}} = (g^{x_{\mathrm{A}}})^{x_{\mathrm{B}}} \equiv y_{\mathrm{A}}^{x_{\mathrm{B}}} \equiv s_{\mathrm{B}} \pmod{p}.$$

A good way to derive a symmetric AES key from the shared secret is to apply a hash function, e.g. taking the first 128 bits of `SHA256`(s_{A}) = `SHA256`(s_{B}).

Assignment 14: Write a program that generates, reads and writes Diffie-Hellman system parameters, private keys and public keys (define proper formats for the different files and use PBE on private key files).
Write a program that computes shared secrets and turns them into AES keys, so that you can encrypt and decrypt.
Use this to encrypt / decrypt files, and experiment with exchanging encrypted files with friends.

9 RSA Cryptanalysis

We discuss some easy cryptanalytic methods for RSA. Note that a lot more efficient methods are known, which are also a lot more complicated to describe, and certainly to implement.

9.1 Factoring

The first thing one thinks about in trying to break RSA, is factoring the modulus.

9.1.1 Fermat Factoring

Fermat's factoring method is very simple. It is based on the trivial observation that $a^2 - b^2 = (a-b)(a+b)$, so a differences of two squares is very easy to factor. So try to write n as a difference of two squares. This leads to the following algorithm:

- Let $X = \lceil \sqrt{n} \rceil$.
- While $X^2 - n$ is not a square, increase X by 1.
- As soon as $X^2 - n = Y^2$ for an integer Y, set $p = X - Y$ and $q = X + Y$, they are factors of n.

With an RSA modulus $n = pq$, where for convenience we assume $p < q$, this will happen at $p = X - Y, q = X + Y$, so $X = \frac{1}{2}(p+q)$, and the number of computations is $\approx X - \sqrt{n} = \frac{1}{2}(p+q) - \sqrt{n} = \dfrac{\frac{1}{4}(p+q)^2 - n}{\frac{1}{2}(p+q) + \sqrt{n}} = \dfrac{(q-p)^2}{8\sqrt{n}}$. This is very interesting, as it depends mainly on the difference $\Delta = q - p$ of the primes. In a generic case, when p, q are generated independently at random of the same bitsize, then their difference will, with overwhelming probability, be of order of magnitude $\sqrt{n}$, as are p and q themselves. This shows that Fermat's factoring method is not much better than a brute force search for p and q. However, with a bit more cleverness it can be sped up. Instead we will show another method first, and then revisit Fermat's method.

9.1.2 Pollard Rho

Pollard Rho is a simple and nice method for attempting to factor a composite integer n such as an RSA modulus that is essentially faster than brute force or Fermat factoring. It usually detects a factor p of n in time $\approx \sqrt{p}$, so it detects some factor usually in time at most $n^{1/4}$, but one may have good or bad luck and have to compute shorter or longer.

The idea is very simple: take some function $f(x)$ (e.g. $f(x) = x^2 + 1$ is often a good choice) and a starting value x_0, and then iterate f modulo n: compute $x_i \equiv f(x_{i-1}) \pmod{n}$ for $i = 1, 2, 3, \ldots$. What to do with those numbers?

Assume that $n = pq$ for prime numbers p, q that are unknown. The sequence $(x_i) \pmod{p}$ will of course be ultimately periodic: since the range of $f \pmod{p}$ is finite, there must be a "collision" of values $x_i \equiv x_j \pmod{p}$ for some $i \neq j$, and then $x_{i+k} \equiv x_{j+k} \pmod{p}$ for all $k = 1, 2, \ldots$, i.e. after an initial "tail" the numbers end up in a "cycle"[21].

Now the hope is that collisions $\pmod{p}$ and collisions $\pmod{q}$ do not happen at the same time. Then, when $x_i \equiv x_j \pmod{p}$ happens, most probably $x_i \not\equiv x_j \pmod{q}$, and this means that $\gcd(x_i - x_j, n) = p$, and this is quite easy to compute without knowing p or q. An example: for $n = 187$ and $x_0 = 1$ we get the sequence $1, 2, 5, 26, 116, 180, 50, \ldots$, and $\gcd(180 - 26, 187)$ can be easily computed to be 11. Thus we found a factor.

At first sight it seems that to find a collision one needs to store the entire sequence. This is not needed. We have Floyd's Cycle Finding Algorithm, which uses a tortoise-and-hare approach, as follows: start with $t = h = x_0$; then at each iteration, compute the tortoise $t \leftarrow f(t) \pmod{n}$ and the hare $h \leftarrow f(f(h)) \pmod{n}$, and check $\gcd(t - h, n)$. Almost no memory needed, and it will find a collision (probably not the first one but that does not matter). For the example above the tortoise-hare pairs are $(1, 1), (2, 5), (5, 116), (26, 50)(116, 39)$, and now $\gcd(116 - 39, 187) = 11$.

One can prove that the typical period length of $x_i \pmod{p}$ is of size $\sqrt{p}$ (birthday paradox). This shows that the algorithm indeed can be expected to halt in time $\approx \sqrt{p}$.

There is also a Pollard Rho method for Discrete Logarithms, but that is more complicated.

Assignment 15: Write a Java program that performs the Pollard Rho attack for factoring, using Floyd's Cycle Finding Algorithm.
Make sure you build in a stopping criterium that stops the cycle finding when it takes too long (and try it out on not too large numbers; you should be able to easily factor 80 bit numbers which should take up to about one million iterations). Note: Java BigInteger has the gcd built in.

9.2 Weak Keys

Weak keys for RSA are keys that have some additional structure that make them a lot easier to break than by known methods such as the Brute Force method of trying out many private keys. Such additional properties can often be found for asymmetric cryptographic systems, because they usually rely on some number theoretic structure that may be used to the advantage of the "bad guys". Here we do not mean simply too short private keys, as that is not really a structural matter. We will study a few types of weak RSA keys.

It is of interest to note that actually generating weak keys requires some additional effort. If RSA key generation is done as described in Section 6.1, the result will with overwhelming probability be a strong key pair. Nevertheless we have seen serious implementations having totally screwed things up, such as different RSA moduli sharing one of their prime factors.

9.2.1 Fermat factoring revisited

We saw that the complexity of Fermat factoring is directly depending on the prime difference $\Delta = q - p$. This immediately implies one class of weak RSA keys, namely those where Δ is too small. Indeed, from the analysis above it follows that for $\Delta < 2.8n^{1/4}$ only 1 computation in Fermat's factoring method suffices. Notice that $p, q \approx n^{1/2}$, so $q - p \approx n^{1/4}$ means that about the upper half of the bits of p and q are identical, but the lower half can be completely arbitrary, and are easy to compute. And if you are willing to do, say, 2^{40} work, then you can already reach

[21]This explains the name "Rho method"; John Pollard was the inventor.

$\Delta \approx 3\,000\,000 n^{1/4}$ as the difference of the two primes.

Assignment 16: Write a Java program that implements Fermat Factoring. To be able to test it, you need to adapt the RSA key pair generation method (see Assignment 6) to generating weak keys, and in this case it means that p and q are not anymore independently generated, but they have to share (somewhat less than) their upper half bits.
You may be able to speed up your program by not only trying the method on n, but also on kn for a bunch of small values for k.
Tip: To prevent the program from running too long, build in a stopping criterium.

9.2.2 Wiener's attack with twodimensional lattices

This attack is more involved. Clearly, when the private exponent d is small, a brute force attack on it is possible. It it therefore recommended to avoid weak keys for which, e.g., $d < 2^{128}$. Wiener's attack shows that actually this bound should be increased to well above $d < n^{1/4}$ in the case that e has "full size", i.e. (almost) the same number of bits as n has.

Assume that $d < n^{1/4}$ (for n a 2048-bit RSA-modulus, this comes down to $d < 2^{512}$), and that the prime factors p, q have the same number of bits (this is seen as good practice). We write down the equation that connects the public and private keys: $ed \equiv 1 \pmod{\phi(n)}$. This means that there is an integer k such that $ed = 1 + k\phi(n) = 1 + k(n-p-q+1)$. Looking at where the big numbers are, we see that $ed \approx kn$, and since e is about as big as n, we find that k is of the same size as d, i.e. is also quite small. This means that $\frac{k}{d}$ is a fraction that is quite close to $\frac{e}{n}$ (indeed, $\left|\frac{e}{n} - \frac{k}{d}\right| \approx \frac{1}{\sqrt{n}}$), but with much smaller numerator and denominator. There is a part of Number Theory, called "Diophantine Approximation Theory", that studies exactly this type of questions, and shows that a small variant of the Euclidean Algorithm can compute those approximations quite fast. We will describe this theory (known as "continued fraction" theory) in terms of lattices.

Given two independent integer vectors $\underline{\mathbf{a}}, \underline{\mathbf{b}} \in \mathbb{Z}^2$, we can define the lattice spanned by them as the set of all their linear combinations with integer coefficients. Given a lattice basis $\underline{\mathbf{a}}, \underline{\mathbf{b}}$, we can do a variant of Euclid's Algorithm on them, as follows:

- if $|\underline{\mathbf{a}}| > |\underline{\mathbf{b}}|$, swap them;
- $\lambda \leftarrow 1$
- while $\lambda \neq 0$ do
  - compute an integer λ for which $|\underline{\mathbf{b}} - \lambda\underline{\mathbf{a}}|$ is minimal;
  - $\underline{\mathbf{b}} \leftarrow \underline{\mathbf{b}} - \lambda\underline{\mathbf{a}}$;
  - if $|\underline{\mathbf{a}}| > |\underline{\mathbf{b}}|$, swap them.

It is not too hard to prove that this algorithm turns any basis $\underline{\mathbf{a}}, \underline{\mathbf{b}}$ of a lattice into a "reduced" basis $\underline{\mathbf{a}}', \underline{\mathbf{b}}'$ of the same lattice, where "reduced" means:

- the only lattice point $\underline{\mathbf{x}}$ such that $|\underline{\mathbf{x}}| < |\underline{\mathbf{a}}'|$ is $\underline{\mathbf{x}} = \underline{\mathbf{0}}$;
- the only lattice points $\underline{\mathbf{x}}$ such that $0 < |\underline{\mathbf{x}}| < |\underline{\mathbf{b}}'|$ are multiples of $\underline{\mathbf{a}}'$.

In popular language: a reduced basis is as short and as orthogonal as it can get in the lattice. A reduced basis makes it very easy to do a *brute force* search for *all* lattice points at a given maximal distance from the origin. It should not come as a big surprise that in general the minimal distances in the lattice are of size $\sqrt{\det\{\underline{\mathbf{a}}, \underline{\mathbf{b}}\}}$.

Actually the way to compute λ simply is: take an integer nearest to $\frac{\underline{\mathbf{a}} \cdot \underline{\mathbf{b}}}{\underline{\mathbf{a}} \cdot \underline{\mathbf{a}}}$. It is no coincidence that this looks like an orthogonal projection.

Now, for given e, n (those are public) we take the lattice defined by

$$\underline{\mathbf{a}} = \begin{pmatrix} K \\ e \end{pmatrix}, \quad \underline{\mathbf{b}} = \begin{pmatrix} 0 \\ n \end{pmatrix},$$

for a suitable constant K (the precise value of K does not matter much as long as its size is well chosen; it simply can be a power of 2). Clearly this lattice has $\det\{\underline{\mathbf{a}}, \underline{\mathbf{b}}\} = Kn$, and we expect the shortest distances in the lattice to be of the size of $K^{1/2}n^{1/2}$. We know that there is an unknown pair (d, k) for which $ed - kn = -k(p + q - 1) + 1$ which is of the size of $dn^{1/2}$ (because k is of the size of d and $p + q$ is of the size of $n^{1/2}$), and we know that $d\underline{\mathbf{a}} - k\underline{\mathbf{b}}$ must be in the lattice, and will actually be pretty short, because

$$d\underline{\mathbf{a}} - k\underline{\mathbf{b}} = \begin{pmatrix} Kd \\ ed - nk \end{pmatrix}$$

which is of the size $\max\{Kd, ed - nk\} = \max\{Kd, dn^{1/2}\}$. Clearly the optimal choice for K is going to be near $n^{1/2}$, and we simply take $K = 2^{b/2}$ where b is the bitsize of n. It then follows that the values for d, k can be easily found from a reduced basis.

An example: let the public key be $n = 38529353$, $e = 19621709$. Then we get the following vectors in the Euclidean algorithm (each time we only mention the new vector):

$$\underline{\mathbf{a}} = \begin{pmatrix} 0 \\ 38529353 \end{pmatrix}, \underline{\mathbf{b}} = \begin{pmatrix} 8192 \\ 19621709 \end{pmatrix}, \lambda = 2, \begin{pmatrix} -16384 \\ -714065 \end{pmatrix}, \lambda = -27, \begin{pmatrix} -434176 \\ 341954 \end{pmatrix}, \lambda = -1,$$
$$\begin{pmatrix} -450560 \\ -372111 \end{pmatrix}, \lambda = 0.$$

So a reduced basis is $\underline{\mathbf{a}}' = \begin{pmatrix} -434176 \\ 341954 \end{pmatrix}, \underline{\mathbf{b}}' = \begin{pmatrix} -450560 \\ -372111 \end{pmatrix}$. We have $\underline{\mathbf{a}}' = -53\underline{\mathbf{a}} + 27\underline{\mathbf{b}}$ and $\underline{\mathbf{b}}' = -55\underline{\mathbf{a}} + 28\underline{\mathbf{b}}$. Our first guess is that $d\underline{\mathbf{a}} - k\underline{\mathbf{b}} = -\underline{\mathbf{a}}'$, and our second guess would be $d\underline{\mathbf{a}} - k\underline{\mathbf{b}} = -\underline{\mathbf{b}}'$ (minus-signs because the first coordinate must be positive). But we may have to look further for linear combinations of $\underline{\mathbf{a}}'$ and $\underline{\mathbf{b}}'$ with small integer coefficients. How to try?

Let's guess $d = 53, k = 27$. Then $ed - nk = -341954$ (from the algorithm), and this must be exactly equal to $1 - k(p + q - 1) = 1 - 27(p + q - 1) \equiv 1 \pmod{27}$. Indeed, $341954 \equiv -1 \pmod{27}$. For erroneous candidates already this check often fails. So now we get $p + q = (341954 + 1)/27 + 1 = 12666$. Together with $pq = n = 38529353$ it's now easy to find $p, q = \frac{1}{2}\left(12666 \pm \sqrt{12666^2 - 4 \cdot 38529353}\right) = \frac{1}{2}(12666 \pm 2512) = 7589, 5077$, and we have found the private key because $5077 \cdot 7589 = 38529353$.

If we would have guessed $d = 55, k = 28$, then $ed - nk = 372111 \equiv 19 \pmod{28}$, which shows it's not the solution.

If we would not have found the solution, we may look further for $d\underline{\mathbf{a}} - k\underline{\mathbf{b}} = \alpha\underline{\mathbf{a}}' + \beta\underline{\mathbf{b}}$ for integer values of α, β near to 0. Note that matrix algebra is very useful here, also to find the coefficients $-53, 27, -55, 28$ above.

Assignment 17: Write a Java program to implement the Wiener attack. To test it, you need to adapt the RSA key generation program to produce key pairs for which d is not much larger than $n^{1/4}$.

9.2.3 Combining Fermat and Wiener

The efficiency of Wiener's attack is based on the fact that n and $\phi(n)$ are close to each other: with $n = pq$ we have $\phi(n) = n - (p + q) + 1$, so $n - \phi(n) \approx p + q \approx 2\sqrt{n}$, which is way smaller than n. Because $2\sqrt{n}$ is also known to the attacker, we can make the approximation even better, and replace the unknown $\phi(n)$ not by n but by $n - \lfloor 2\sqrt{n} \rceil$. In the above example this would have lead to using as lattice basis

$$\underline{\mathbf{a}} = \begin{pmatrix} 0 \\ 38516939 \end{pmatrix}, \underline{\mathbf{b}} = \begin{pmatrix} 8192 \\ 19621709 \end{pmatrix}, \lambda = 2, \begin{pmatrix} -16384 \\ -726479 \end{pmatrix}, \lambda = -27, \begin{pmatrix} -434176 \\ 6776 \end{pmatrix}, \lambda = 0.$$

The smaller 6776 instead of 341954 indeed indicates a better approximation.

Analyzing this situation, we see that actually $n - 2\sqrt{n} - \phi(n) = (p - \sqrt{n}) + (q - \sqrt{n}) - 1$, and when p, q get closer to each other, they both get closer to $\sqrt{n}$, because $pq = n$. This means that Wiener's method should perform considerably better when p and q get closer together, which is what also was useful in Fermat's method. Thus we can try to interpolate between Wiener and Fermat.

So let $\Delta = q - p \ll \sqrt{n}$, then $p \approx \sqrt{n} - \frac{1}{2}\Delta$ and $q = \dfrac{n}{p} \approx \dfrac{n}{\sqrt{n} - \frac{1}{2}\Delta} = \dfrac{n(\sqrt{n} + \frac{1}{2}\Delta)}{n - \frac{1}{4}\Delta^2} \approx \sqrt{n} + \frac{1}{2}\Delta + \frac{1}{4}\dfrac{\Delta^2}{\sqrt{n}}$, so $p + q - 2\sqrt{n} \approx \dfrac{\Delta^2}{\sqrt{n}}$. In Wiener's estimate leading to $n^{1/4}$ as a natural upper bound, we now replace $ed - kn \approx 2k\sqrt{n}$ by the much more accurate $ed - k(n - 2\sqrt{n}) \approx k\dfrac{\Delta^2}{\sqrt{n}}$. With $K \approx \left\lfloor \dfrac{\Delta^2}{\sqrt{n}} \right\rceil$ as optimal choice (again the precise value does not matter, simply take a power of 2) we see that the method works for d up to $\dfrac{n^{3/4}}{\Delta}$.

For generic p, q we have that Δ will be of the size of $\sqrt{n}$, and the new bound does not substantially improve on Wiener's. For Δ of smaller size than $n^{1/4}$ Fermat's method gives an immediate result so we don't bother. But when Δ is in between $n^{1/4}$ and $n^{1/2}$ we see that we can get a better range for Wiener's method than $n^{1/4}$, and also a better result than only Fermat would have given.

Assignment 18: Try out your Wiener attack program on the case described in this section. Test it on a case which is out of reach for both Fermat's and Wieners attacks separately, but is doable for the combined attack.

9.2.4 Håstad's attack on small public exponent

Håstad's attack against "raw" RSA is quite simple but works only in special circumstances: three users have different (pairwise coprime!) RSA moduli but all use $e = 3$, and one sender sends them all the exact same encrypted message m. Then an attacker who intercepts the encrypted messages is able to decrypt them. The attack is very simple indeed: if the attacker intercepts the three ciphertexts c_1, c_2, c_3 that have been encrypted via $c_i \equiv m^3 \pmod{n_i}$ for $i = 1, 2, 3$, then the attacker can compute by the Chinese Remainder Theorem the unique positive integer $c < n_1 n_2 n_3$ such that

$$c \equiv \begin{cases} c_1 \pmod{n_1}, \\ c_2 \pmod{n_2}, \\ c_3 \pmod{n_3}. \end{cases}$$

This means that $c \equiv m^3 \pmod{n_1}, \pmod{n_2}, \pmod{n_3}$, and because also $m^3 < n_1 n_2 n_3$ it must be true that $c = m^3$ exactly, not just modulo something. Then m is easily found by extracting the 3rd root of c.

It remains to show how the Chinese Remainder Theorem works in this case. Because $c \equiv c_1 \pmod{n_1}$ we get that there exists some integer x such that $c = c_1 + xn_1$. Then the second equation $c \equiv c_2 \pmod{n_2}$ becomes $c_1 + xn_1 \equiv c_2 \pmod{n_2}$, and from this x (between 0 and n_2) can be solved $\pmod{n_2}$. Let us write $c_{1,2} = c_1 + xn_1$ for this value of x . It follows that we now know $c \equiv c_{1,2} \pmod{n_1 n_2}$. There is an integer y such that $c = c_{1,2} + yn_1 n_2$, and the third equation $c \equiv c_3 \pmod{n_3}$ then turns into $c_{1,2} + yn_1 n_2 \equiv c_3 \pmod{n_3}$, and now y (between 0 and n_3) can be solved $\pmod{n_3}$. Then $c = c_{1,2} + yn_1 n_2$ is the value we need.

The same attack works for e users who all use the same private exponent e and get the exact same message encrypted for them.

An obvious countermeasure is to always use randomized encryption, as RSA with PKCS#1v1.5-padding provides.

Assignment 19: Write a Java program that demonstrates Håstad's attack. Note that `net.deweger.crypto.Util` has a method for extracting 3rd roots.

9.3 The Microwave Attack

This is a really funny (yet practical!) attack on RSA-CRT in special circumstances. The setting is that an RSA-CRT private key resides on a smartcard, and cannot be exported from it. For convenience we assume that the private key is used for generating signatures, and that "Raw RSA" is used, i.e. one may ask the card on input h to compute $s \equiv h^d \pmod{n}$. The card has the values for p, q, d_p, d_q and u as described in Section 6.6.

Putting the card inside a microwave oven and then "cooking" it for a short time may result in the radiation causing a random bitflip in the card's memory. Let us assume that in the value of d_q one bit is flipped, and all other memory bits are undamaged. When the card then is asked to generate a signature on the value h of the choice of the "attacker", the resulting signature s' will be erroneous because of the flipped bit, but modulo p the value will be correct; only modulo q it will be wrong. The "attacker" will thus be able to compute $s'^e - h \pmod{n}$, which will have p as a factor but not q. Then p will leak as $\gcd(n, s'^e - h \pmod{n})$, and the private key thus can be recovered.

Assignment 20: Write a Java program that implements a simulation of the microwave attack (you better not try out a physical attack on your bank card). Experiment a bit, i.e. with flipping bits in other places.

This type of attack is called "fault injection". A countermeasure is to let the smartcard check the private key operation by immediately doing the corresponding public key operation after it for verification.

Literature

Keith M. Martin, *Everyday Cryptography*, Oxford University Press, 2nd ed., 2017.
Benne de Weger, *Elementaire getaltheorie en asymmetrische cryptografie*, Epsilon Uitgaven, 3e druk, 2016.

Appendix - `CryptoExample.java` source code

```java
// Simple examples for net.deweger.crypto
// version 1.0, March 18, 2021
// Copyright: Benne de Weger, TU/e

import net.deweger.crypto.*;
import java.nio.file.*;

public class CryptoExample
{
    // variables
    static byte[] key;
    static byte[] iv;
    static byte[] plaintext;
    static byte[] ciphertext;
    static byte[] decryptedtext;

    // read data from a file to a byte array
    private static byte[] readFile(String fileName) throws Exception
    {
        byte[] data = Files.readAllBytes(Paths.get(fileName));
        System.out.println("file " + fileName + " read (" + data.length + " bytes)");
        return data;
    }
```

```
// write data to a file from a byte array
private static void writeFile(String fileName, byte[] data) throws Exception
{
    Files.write(Paths.get(fileName), data);
    System.out.println("file " + fileName + " written (" + data.length + " bytes)");
}

// basic encryption of a file
private static void AESexampleEncrypt(String fileName) throws Exception
{
    AES aes = new AES();
    // generate a random AES key and iv
    key = aes.generateKey();
    iv = aes.generateIv();
    // set mode and padding
    aes.setMode(AES.MODE_CTR);
    aes.setPadding(AES.PAD_PKCS7);
    // print key and iv in hex
    System.out.println("AES key: " + Util.bytesToHex(key));
    System.out.println("AES IV:  " + Util.bytesToHex(iv));
    // read file
    plaintext = readFile(fileName);
    // encrypt
    ciphertext = aes.encrypt(plaintext);
    // write ciphertext to filename.enc
    writeFile(fileName + ".enc", ciphertext);
}

// basic decryption of a file
private static void AESexampleDecrypt(String fileName) throws Exception
{
    AES aes = new AES();
    // set AES key and iv
    aes.setKey(key);
    aes.setIv(iv);
    // set mode and padding
    aes.setMode(AES.MODE_CTR);
    aes.setPadding(AES.PAD_PKCS7);
    // read file
    ciphertext = readFile(fileName);
    // decrypt
    decryptedtext = aes.decrypt(ciphertext);
    // write decryptedtext to filename.dec
    writeFile(fileName + ".dec", decryptedtext);
}

// basic hashing of a file
private static void SHA256example(String fileName) throws Exception
{
    SHA256 sha256 = new SHA256();
    // read file
    byte[] data = readFile(fileName);
    // compute hash
    byte[] hash = sha256.hash(data);
    // print result
    System.out.println("SHA256(" + fileName + "):\n    " + Util.bytesToHex(hash));
    // and now with splitting up data, say 10 bytes at a time
    byte[] datapart = new byte[10];
```

```
        int len = data.length;
        sha256.hashInit();
        while (len >= 10)
        {
            System.arraycopy(data, data.length - len, datapart, 0, 10);
            sha256.hashUpdate(datapart);
            System.out.println("processed bytes " + (data.length - len) + " to " +
                (data.length - len + 9));
            len -= 10;
        }
        if (len > 0)
        {
            datapart = new byte[len];
            System.arraycopy(data, data.length - len, datapart, 0, len);
            sha256.hashUpdate(datapart);
            System.out.println("processed bytes " + (data.length - len) + " to " +
                (data.length - 1));
        }
        hash = sha256.hashFinal();
        System.out.println("SHA256(" + fileName + "):\n    " + Util.bytesToHex(hash));
    }

    public static void main(String[] args)
    {
        try
        {
            // encrypt example.txt to example.txt.enc
            AESexampleEncrypt("example.txt");
            // decrypt example.txt.enc to example.txt.enc.dec
            AESexampleDecrypt("example.txt.enc");
            // hashes of the three files
            SHA256example("example.txt");
            SHA256example("example.txt.enc");
            SHA256example("example.txt.enc.dec");
            // first and third hash should be identical
        }
        catch (Exception exc)
        {
            System.out.println("ERROR: " + exc.getMessage());
        }
    }
}
```