Java implemented Wiener attack simulation

Cryptography Term Project

Gioacchino Castorio

Università dell'Aquila

2017-06-20

Table of contents

RSA: an overview

Story of RSA

- Invented in 1977 in MIT
- Its name is made up from the initial letters the the surnames of the inventors (R. **R**ivest, A. **S**hamir and L. **A**dleman)
- Its algorithm was under patent in US till 2000-09-06, altough it was publicly known

Public-key (asymmetric) cryptography

- The one who wants to receive encrypted messages **generates one pair of keys**:
  - a **public key** that is publicly retrievable
  - a **private key** that is kept secret by the owner
- This pair accomplish multiple function:
  - **encryption**: a sender can use the public key to encrypt the message, while the owner can use the private key decrypt the incoming messages;
  - **authentication**: the public key could be used to verify that a holder of the corresponding private key sent the message (i.e. the owner “signs” the message).

RSA key generation algorithm

1. Choose $p, q \in \mathbb{Z}$, $p \neq q$ big primes;
2. Compute $n = p \cdot q$ and $\phi(n) = (p-1) \cdot (q-1)$;
3. Find $e \in \mathbb{Z}$, called "**encryption exponent**", so that $1 \leq e < \phi(n)$ and $gcd(e, \phi(n)) = 1$;
4. Find $d \in \mathbb{Z}$, called "**decryption exponent**", so that $e \cdot d \equiv_{\phi(n)} 1$
5. Now we can build the keys:
   - **Public key**: [**e**, n];
   - **Private key**: [**d**, q, p];

RSA encrypt and decrypt method

Once the sender has retrieved the public key, he can easily encrypt a **plain message m**, such that $1 \leq m < n$:

$$c \equiv_n m^e$$

Symmetrically, the owner of the private key that receives the **cipher text c** can obtain m:

$$m \equiv_n c^d = m^{e \cdot d} \equiv m^1$$

Attack RSA

Why is RSA considered so secure?

- Its security lies especially in the fact that it is very difficult (or roughly impossible) to **factorize very big integers** (represented as very long strings of bits);
- Big integer factorization is proved to be a **Not deterministic Polynomial** problem, although it might not be NP-complete
- It is fundamental to use p and q such that n would be made up of $\geq$ **1024 bits** (it takes years to be factorized)

Wiener theorem

- Given two primes p and q such that $q < p < 2q$;
- Compute $n = p \cdot q$, $\phi(n) = (p-1) \cdot (q-1)$;
- Given $1 < e, d < \phi(n)$ such that $e \cdot d \equiv_{\phi(n)} 1$;
- If $d < \frac{1}{3} n^{\frac{1}{4}}$ then **d is simply computable**.

Wiener attack algorithm

1. Find the i^{th} **convergent** (that we will call $\frac{A_i}{B_i}$) of $\frac{e}{n} \in \mathbb{Q}$;
2. If $C = \frac{e \cdot B_i - 1}{A_i} \in \mathbb{Z}$ then is C is a candidate for $\phi(n)$, otherwise return to *step 1*;
3. If the equation $x^2 - (n - C + 1)x + n = 0$ has integer solution x_1, x_2 then $x_1 = p$, $x_2 = q$, otherwise return to *step 1*.

Java implementation of Wiener-vulnerable RSA

What is this project about?

It simulates a communication between a message sender and a receiver, using **1024 bit Wiener-vulnerable** $n = p \cdot q$ product on a single message block. After that, it simulates an attack against the public key with the Wiener algorithm (successful).

Project structure

The project consists essentially of a packaged RSA library built with:

- Custom implemented classes:
  - **IRSAChiper.java**: RSA Cipher interface and implementation;
  - **BigRational.java**: data structure for arbitrary dimension rational numbers representation (with operations);
  - **PublicKey.java** and **PrivateKey.java**: data structure that contains each part of the keys.
- Default Java 8 SE classes:
  - **BigInteger.java**: data structure for arbitrary dimension integer numbers representation (with operations like *gcd*, *power*, *modulo*).

IRSACipher.java interface [1/2]

```
public interface IRSACipher {

    KeyBundle getWienerAttackableKeys(int factorlength);

    BigInteger encryptBlock(BigInteger plainmessage, PublicKey key);

    BigInteger decryptBlock(BigInteger chipertex, PrivateKey key);

    KeyBundle attackWiener(PublicKey publicKey);

    boolean isWienerAttackable(PrivateKey privateKey);

}
```

IRSACipher.java interface [2/2]

Essential method description:

- **getWienerAttackableKeys**
  - parameter *factorlenght*: it is the bit length of the factors p and q, pseudo-randomly generated and tested by Rabin-Miller algorithm ($q < p < 2q$ checked);
  - it generates the decryption key d forcing it to be such that $bit(d) = bit[\frac{1}{3}n^{\frac{1}{4}}] - 1$ (Wiener is compulsorily respected);
  - it returns a *KeyBundle* with the public and private keys of the owner.
- **attackWiener**
  - parameter *publicKey*: couple [e, n] to be attacked
  - it returns a *KeyBundle* with the hacked public and private keys .

BigRational.java [1/2]

```java
public class BigRational {

    private BigInteger numerator;
    private BigInteger denominator;

    public static BigRational recomposeConvergent(List<BigInteger> expansion) {...}

    public BigRational(BigInteger numerator, BigInteger denominator) {...}

    public List<BigInteger> getListIntegersContinuedFraction() {...}

   // ... omissis ...

}
```

BigRational.java [2/2]

Essential method description:

- **getListIntegersContinuedFraction**
  - It decomposes the rational in $r = a_0 + r_1 = a_0 + \frac{1}{\frac{1}{r_1}}$;
  - It returns the list of ordered integers that make up the continued fraction expansion of the rational;
- **recomposeConvergent** (*static method*)
  - Retuns the n^{th} convergent from the passed integer list

Workflow [1/3]

```
CHIAVI GENERATE
--------------------------------
PUBLIC KEY
n = 93834361460226264424637498837795330870625721950864997686726483767668228578335558959650823197767256803317717036202475988556961798426090244051432385344350742815959225363007186582450863355027
e = 26856290516396894318272679573819368221452853121971332610204468681307058575595438030024929102553334896157809511253592090216435264421444285194066672184159010474148952940012894986710773745002


PRIVATE KEY
d = 1074624559864568348589195621410295822971882924026146709420187854893301447487,
p = 11151940562655701561214734901091686978772350725020930971999581702323920911574050606073048479033689939924766443526298429628770158633914490104317409868725169,
q = 8414173383818746881206348889916350559409301640560562696578455613905981780147862837840778445385388281764164348126251619989897960659599874569839057027723043,
n = 93834361460226264424637498837795330870625721950864997686726483767668228578335558959650823197767256803317717036202475988556961798426090244051432385344350742815959225363007186582450863355027

--------------------------------
Attaccabile con Wiener: true
--------------------------------


Testo da criptare: esame di crittografia
```

What does this image show:

1. The systems generates the key cuple and promts it;
2. It prompts the results of the Wiener vulnerability check;
3. The system requests the sender to write a message.

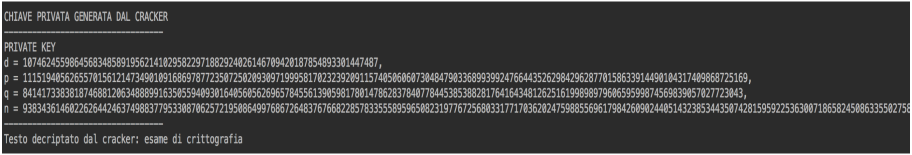
Workflow [2/3]

```
CONFRONTO DEI MESSAGGI
---------------------------------
Testo chiaro originale: esame di crittografia
Testo criptato: K���&'�Vx���<�i�it�Ő{ s�}!�ÈK:���1dd�?��¿���Mb&� 3�� n� Y �$n q]��dq G� 2����� �3v��X��#(�iч LP�n ��Zcb� .i��,�n��N�
Testo decriptato: esame di crittografia
---------------------------------
```

Let's see these lines:

1. The plain message (not encrypted);
2. The chiper text (encrypted by the public key);
3. The result of the decryption (same as the plain message).

Workflow [2/3]



What has Eveline managed to do?

1. She has obtained the (p, q) cuple;
2. She has computed the **decription exponent** easely ($e \cdot d \equiv_{\phi(n)} 1$)!
3. She has decrypted the whole message!

Thank you for your attention

Let's try it!