Implementing RSA – Draft

F. De Santis
Sichere Implementierung kryptographischer Verfahren WS 2015-2016
Technische Universität München **October 28, 2015**

Contents

1 Background

In this section, some definitions and results from elementary number theory and algebra are recalled. The interested reader is referred to [Chi09, Kob94] for a more complete and rigorous treatment.

Definition 1.1. Let $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, -2, \ldots\}$ denote the set of integers and $\mathbb{Z}^+ = \{1, 2, 3 \ldots\}$ be the set of *positive* integers.

Definition 1.2. For two integers k and n, k divides n if n is a multiple of k.

Definition 1.3. An integer $p \geq 2$ is *prime* if its only divisors are 1 and p.

An integer that is not prime is *composite* and can always be written as a product of primes, possibly with repetitions.

Definition 1.4. For two integers a and b, the greatest common divisor of a and b, denoted by $\gcd(a, b)$, is the largest positive integer that divides both a and b.

Definition 1.5. Two integers a and b are *relatively prime* or *coprime* if $\gcd(a, b) = 1$.

Definition 1.6 (Group)**.** A *group* $\mathcal{G} = (S, \cdot, e)$ is an algebraic structure whose support set S is endowed with a binary operation $\cdot$ and an element e such that the following properties hold:

1. $\forall a, b \in S : a \cdot b \in S$ [Closure]
2. $\forall a, b, c \in S : (a \cdot b) \cdot c = a \cdot (b \cdot c)$ [Associativity]
3. $\exists e \in S \, \forall a \in S : a \cdot e = e \cdot a = a$ [Identity]
4. $\forall a \in S \, \exists a' \in S : \; a \cdot a' = a' \cdot a = e.$ [Inverse]

If the binary operation $\cdot$ is also commutative, then $\langle S, \cdot, e \rangle$ is called *abelian group* or *commutative group*:

5. $\forall a, b \in S, \; a \cdot b = b \cdot a$ [Commutativity]

If the support set S is finite, then G is called *finite group*. If there exist an element $g \in S$ such that the iterative application of the group operation $\cdot$ to the element g generates all the elements of S, then the group $\mathcal{G}$ is called *cyclic group* and the element g is called the *generator* of the group $\mathcal{G}$.

The binary operation $\cdot$ is typically called *product*, the element e is called *identity element* or *neutral element* and the element a' is called *inverse element* of a. The inverse element is also referred to as $-a$ or a^{-1}, depending on whether $\cdot$ is viewed as a sum operator or as a product operator. Similarly, the iteration of group operation $\cdot\ k$ times is also referred to as $x = kg$ or $x = g^k$.

Definition 1.7. For an integer n, let $\mathbb{Z}_n = \{0, \ldots, n-1\}$ denote the set of integers modulo n.

Please note that in literature, the notation $\mathbb{Z}/n\mathbb{Z}$ is sometimes preferred in place of $\mathbb{Z}_n$.

Definition 1.8. Let $\mod : \mathbb{Z} \times \mathbb{Z}^+ \to \mathbb{Z}_n$ denote the remainder (or, residue) r of the integer division a/n:

$$\mod(a, n) = a \pmod{n} = \begin{cases} r & \text{if } a \geq 0 \\ n - r & \text{otherwise} \end{cases}$$

The modulo operation has the following important properties:

1. The remainder is independent of the sign of the divisor n:

$$a \pmod{n} = a \pmod{-n}.$$

2. Adding multiples of the divisor to the dividend a does not change the remainder:

$$a + jn \pmod{n} = a \pmod{n}.$$

3. The remainder of a *sum* equals the *sum* of the remainders:

$$a_1 + a_2 \pmod{n} = [a_1 \bmod n] + [a_2 \bmod n] \pmod{n}.$$

4. The remainder of a *product* equals the *product* of the remainders:

$$a_1 a_2 \pmod{n} = [a_1 \bmod n][a_2 \bmod n] \pmod{n}.$$

Definition 1.9. For an integer n, let $\mathbb{Z}_n^* = \{a : 1 \leq a < n, \gcd(a, n) = 1\}$ denote the set of integers modulo n which are coprime with n.

For every positive integer n, the set $\mathbb{Z}_n^*$ endowed with the modular multiplication $\cdot$ is an abelian group $\mathcal{G} = (\mathbb{Z}_n^*, \cdot)$ and 1 is the identity element. Therefore, for every element of the set, there exists a multiplicative inverse. Please note that in literature, the notation $(\mathbb{Z}/n\mathbb{Z})^*$ is sometimes preferred in place of $\mathbb{Z}_n^*$.

Definition 1.10. Let $\Phi(n)$ denote the number of elements of $\mathbb{Z}_n^*$ or, equivalently, the number of integers in $\mathbb{Z}_n$ relatively prime to n.

Theorem 1.1. If $n = p_1^{e_1} p_2^{e_2} \ldots p_m^{e_m}$, where p_i are prime numbers and e_i integers, then $\Phi(n) = \prod_{i=1}^{m} (p_i^{e_i} - p_i^{e_i - 1})$.

Please note that given a factorization of n is computationally easy to compute $\Phi(n)$. In contrast, if the factorization of n is unknown, computing $\Phi(n)$ is *nowadays* believed to be as hard as factorizing n.

Definition 1.11. For a prime p, let $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$ denote the set of integers modulo p.

Please note that $\mathbb{Z}_p^*$ is an abelian group, being a special case of $\mathbb{Z}_n^*$. $\mathbb{Z}_p^*$ is also cyclic and has $\Phi(p-1)$ generators. Most importantly, when n is the product of two primes p and q, $\mathbb{Z}_n^*$ is isomorphic to the product group $\mathbb{Z}_p^* \times \mathbb{Z}_q^*$.

Theorem 1.2 (Euler's Theorem)**.** If $\gcd(a, n) = 1$, then $a^{\Phi(n)} \equiv 1 \pmod{n}$.

Theorem 1.3 (Fermat's Little Theorem)**.** If p is a prime integer, then $a^p \equiv a \pmod{p}$ for any integer a.

The Fermat's little theorem is a special case of Euler's theorem. Interestingly, it can be restated as $a^{p-1} \equiv 1 \pmod{p}$.

Theorem 1.4 (Chinese Reminder Theorem)**.** Let $n_1, n_2, \ldots, n_k$ be positive integers such that are pairwise co-prime, $\gcd(n_i, n_j) = 1$ for all $i \neq j$, and let $a_1, a_2, \ldots, a_k$ be integers. Then, the system of linear congruences

$$\begin{cases} x \equiv a_1 \pmod{n_1} \\ x \equiv a_2 \pmod{n_2} \\ \ldots \\ x \equiv a_k \pmod{n_k} \end{cases}$$

has a unique simultaneous solution modulo $N = n_1 n_2 \cdots n_k$.

2 Public-Key Cryptography

In order to secure communications, symmetric cryptography requires a *secure* channel where two parties can securely share a secret key. The deployment of symmetric cryptography becomes impractical as soon as n parties want to communicate securely. In fact, the number of channels which would be required to share the secret keys among all the n parties is $O(n^2)$ and therefore not practical for large n. This issue is known as the key exchange problem.

Public-key cryptography was initially developed to solve the key exchange problem, but nowadays covers several other applications such as: public-key encryption, digital signature schemes, key exchange schemes, zero knowledge schemes, authentication schemes, ...

Public-key cryptography is based on the existence of *one-way functions*, namely *trapdoor functions*.

Definition 2.1. A function f is a one-way function if $y = f(x)$ can be computed efficiently, but the inverse $x = f^{-1}(y)$ is hard to compute.

Definition 2.2. A trapdoor function is a one-way function whose inverse can be computed efficiently given a (secret) trapdoor information.

Trapdoor functions are typically derived from difficult problems in number theory or other mathematical branches. Practical instances of difficult problems are:

- Integer factorization problem (*e.g.*, RSA, Fiat-Shamir, ...)
- Discrete logarithm problem (*e.g.*, DH, DSA, El Gamal, ...)
- Discrete logarithm problem on elliptic curves (*e.g.*, ECDSA, ECDH, ...)
- Lattices problems as the shortest or the closest vector problems (*e.g.*, NTRU, ...)

In practice, the performance of public-key cryptography implementations are typically worse than corresponding private-key cryptography implementations (*e.g.*, block cipher based). For this reason, public-key algorithms are mainly used for key exchange and digital signatures schemes, and rarely used for bulk encryption.

3 RSA

RSA is one of the most popular public-key cryptosystem invented by R. Rivest, A. Shamir and L. Adleman in 1977 at Massachusetts Institute of Technology (MIT) [RSA78]. RSA is based on the intractability of the integer factorization problem. State of the art key size for RSA is 2048-bits for equivalent symmetric key 112-bit security.

3.1 Public-Key Encryption (PKE)

The RSA cryptosystem consists of three algorithms: key generation, encryption and decryption.

Key Generation The key generation algorithm generates a public key k_{pub} and private key k_{pr} starting from two large primes p and q as follows:

1. Two large primes p and q are chosen and the product $n = pq$ is computed together with $\Phi(n) = (p-1)(q-1)$.
2. An integer e is chosen in $]1, \Phi(n)[$, such that $\gcd(e, \Phi(n)) = 1$ and the multiplicative inverse $d = e^{-1} \bmod \Phi(n)$ is computed. Please note that this condition guarantees the existence of the inverse for every e.
3. Therefore the *public key* is $k_{pub} = (n, e)$ and the *private key* is $k_{pr} = (n, d)$.

Encryption The encryption algorithm produces a integer $y \in \mathbb{Z}_n$ from a integer $x \in \mathbb{Z}_n$ using the public key $k_{pub} = (n, e)$ as follows:

$$y = e_{k_{pub}}(x) = x^e \pmod{n}$$

Decryption The decryption algorithm reproduces the integer $x \in \mathbb{Z}_n$ from a integer $y \in \mathbb{Z}_n$ using the private key $k_{pr} = (n, d)$ as follows:

$$x = d_{k_{pr}}(y) = y^d \pmod{n}$$

Please note that the modular exponentiation $(x, e) \mapsto y = x^e \pmod{n}$ is the *trapdoor function* of RSA for p and q large primes and $n = pq$.

3.2 Digital Signature Schemes (DSS)

Digital signature schemes provide authentication, integrity and non-repudiation and are widely used for software distribution and financial transactions. A digital signature scheme consists of a signature generation algorithm sig and a signature verification algorithm ver. The signature generation algorithm generates a signature $y = \mathsf{sig}^{\mathcal{A}}(x)$ which associates a message x with the entity $\mathcal{A}$ signing the message. Conversely, the signature verification algorithm $\mathsf{ver}^{\mathcal{B}}(x, y)$ is typically run by a different entity $\mathcal{B}$ which verifies that the signature y is authentic and it was indeed created by the entity $\mathcal{A}$.

RSA can be used for digital signature schemes by reversing the role of the encryption and decryption as shown in Figure 3.2.

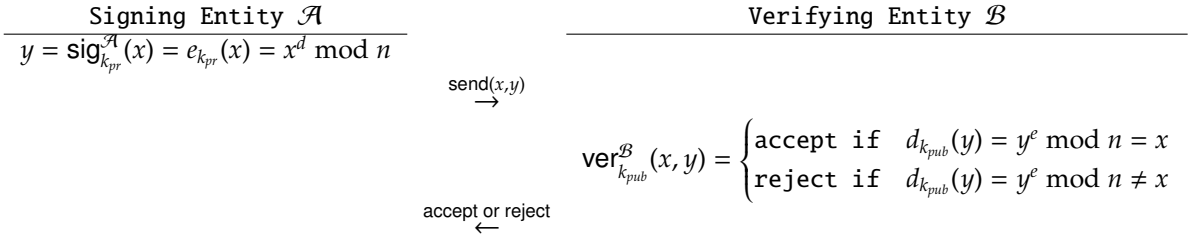

Figure 1: The RSA Signature Scheme

4 Implementation

In this section the algorithms necessary to implement RSA are introduced. Efficient algorithms to generate prime numbers, compute the greatest common divisor and the modular multiplicative inverse are required to generate the keys. Similarly, efficient algorithms to compute the modular exponentiation and the modular multiplication are required for RSA encryption and decryption.

4.1 Prime Generation

In RSA, the key generation algorithm must select two large primes p and q. In practice, there exist randomized algorithms, such as the Miller-Rabin algorithm, and deterministic algorithms, such as the Agrawal–Kayal–Saxena, to test the primality of a given integer. These algorithms are polynomial time algorithms and do *not* require integer factorization. Please refer to [Die04, Sti05] for more details.

4.2 Greatest Common Divisor

In RSA, the public exponent e must be chosen such that $\gcd(e, \Phi(n)) = 1$. The greatest common divisor can be computed using the Euclid's algorithm. The Euclid's algorithm is based on the observation that $\gcd(a_1, a_2) = \gcd(a_2, a_1 \bmod a_2)$ for any two positive integers $a_1, a_2 \in \mathbb{Z}^+$ with $a_2 < a_1$. Thus, the greatest common divisor can be computed recursively by iterating successive reductions of the inputs:

$$\begin{aligned} &\gcd(a_1, a_2) \\ &\quad = \gcd(a_2, a_1 \bmod a_2) = \gcd(a_2, r_1) && r_1 < a_2 < a_1 \\ &\quad = \gcd(r_1, a_2 \bmod r_1) = \gcd(r_1, r_2) && r_2 < r_1 < a_2 < a_1 \\ &\quad = \gcd(r_2, r_1 \bmod r_2) = \gcd(r_2, r_3) && r_3 < r_2 < r_1 < a_2 < a_1 \\ &\qquad\quad \vdots \qquad\qquad\quad \vdots \end{aligned}$$

In practice, the Euclid's algorithm proceeds by iteratively computing the residues $r_i = a_i \pmod{a_{i+1}}$ till the halting condition $r_i = 0$ is reached. Then the greatest common divisor between a_1 and a_2 is given by the last computed residue $\gcd(a_1, a_2) = r_{i-1}$:

$$\begin{aligned} r_1 &= a_1 \bmod a_2 \\ r_2 &= a_2 \bmod r_1 \\ r_3 &= r_1 \bmod r_2 \\ &\vdots \qquad \vdots \end{aligned}$$

The pseudocode of the Euclid's algorithm is listed by the Algorithm 1.

Algorithm 1 Euclid's Algorithm

Input: $a, b \in \mathbb{Z}^+$
Output: $g = \gcd(a, b)$

```
a = max(a, b)                          (find out the larger input)
b = min(a, b)                          (find out the smaller input)
while b ≠ 0 do                         (repeat till the halting condition is met)
    r ← a (mod b)                      (do the reduction)
    a ← b                              (candidate gcd)
    b ← r                              (new remainder)
end while
g ← a                                  (return the gcd)
return g
```

4.3 Modular Multiplicative Inverse

In RSA, the private exponent d is chosen such that $d = e^{-1} \bmod \Phi(n)$. The existence of the inverse element d is guaranteed since e and $\Phi(n)$ are coprime. In general, given two positive integers a and b, the Euclid's algorithm can be extended to compute d and t such that $ad + bt = \gcd(a, b)$. If a and b are coprime, it follows that $ad + bt = \gcd(a, b) = 1$, d is the modular multiplicative inverse of a modulo b and t is the modular multiplicative inverse of b modulo a:

$$\begin{aligned} ad + bt \equiv 1 \pmod{b} \quad &\Leftrightarrow \quad [ad \bmod b] + [bt \bmod b] \equiv 1 \pmod{b} \quad \Leftrightarrow \quad ad \equiv 1 \pmod{b} \quad \Leftrightarrow d \equiv a^{-1} \pmod{b} \\ ad + bt \equiv 1 \pmod{a} \quad &\Leftrightarrow \quad [ad \bmod a] + [bt \bmod a] \equiv 1 \pmod{b} \quad \Leftrightarrow \quad bt \equiv 1 \pmod{a} \quad \Leftrightarrow t \equiv b^{-1} \pmod{a} \end{aligned}$$

In RSA, the public exponent e is coprime to $\Phi(n)$, therefore it follows that $ed + \Phi(n)t = 1$, i.e. $ed \equiv 1 \bmod \Phi(n)$ and $d = e^{-1} \bmod \Phi(n)$.

A naïve implementation of the extended Euclidean algorithm is nothing but a careful backtracking of the Euclid's algorithm. The pseudocode of the extended Euclidean algorithm is provided by the Algorithm 2.

Algorithm 2 Extended Euclidean Algorithm

Input: $a, b \in \mathbb{Z}^+$
Output: $d, t \in \mathbb{Z}$ such that $ad + bt = \gcd(a, b)$

```
ExtEuclidAlg(a, b) :
if b = 0 then                          (check if the halting condition is met)
    return 1, 0                        (return the solution d = 1, t = 0)
else                                   (otherwise)
    q ← ⌊a/b⌋                          (compute the quotient)
    r ← a (mod b)                      (compute the remainder)
    x, y ← ExtEuclidAlg(b, r)          (recursively find x, y such that bx + ry divides both b and r)
end if
d ← y
t ← x − qy
return (d, t)
```

4.4 Modular Exponentiation

The modular exponentiation is used for both encryption and decryption. The naïve approach to compute the modular exponentiation $y^d \bmod n$ by the means of successive multiplications require $d - 1$ multiplications followed by the reduction modulo n:

$$y^d \pmod{d} = \underbrace{y \cdot y \cdot y \cdot \cdots \cdot y}_{d-1} \pmod{n}$$

This approach is clearly impractical for d large, as it would require huge memory storage to store y^d before the reduction. Therefore, an approach which perform intermediate reductions and possibly reduce the amount of multiplications should be preferred in practice.

Let k be the size of the private exponent d in bits. The natural binary representation of the exponent $d = \sum_{i=0}^{k-1} d_i 2^i$ allow to rewrite the exponentiation as follows using the properties of powers and modular arithmetic:

$$y^d \pmod n = y^{\sum_{i=0}^{k-1} d_i 2^i} \pmod n = \prod_{i=0}^{k-1} \left(y^{2^i}\right)^{d_i} \pmod n = \prod_{i:d_i=1} \left[y^{2^i} \pmod n\right] \pmod n$$

The key observation is that $y^{2^i} = y^{2 \cdot 2^{i-1}} = \left(y^{2^{i-1}}\right)^2$, therefore the exponentiation can be computed by successive squaring operations and conditional multiplications depending on whether the key bit d_i is set. This recursive formulation of the exponentiation directly translates into the so called left-to-right square-and-multiply algorithm provided by Algorithm 3.

Algorithm 3 Left-to-right Binary Exponentiation Algorithm

Input: $y, n, d = (d_{k-1}, d_{k-2}, \ldots, d_1, d_0)_2$
Output: $x = y^d \pmod n$

```
1: x ← 1
2: for i = k − 1 downto 0 do                (scan through the key bits d_i from MSB to LSB)
3:     x ← x^2 (mod n)                      (do the squaring always)
4:     if d_i = 1 then                      (if the key bit d_i is set)
5:         x ← xy (mod n)                   (do the conditional multiplication)
6:     end if
7: end for
8: return x
```

This approach reduces the number of multiplications from $d - 1$ to $1.5k$ on average, as it requires k multiplications for the squaring operations (being the squaring operation a multiplication itself) and a number of multiplications which equals Hamming weight of the exponent ($k/2$ on average).

For example, let consider $k = 4$ and $d = 9 = (1, 0, 0, 1)_2$. Thus, the exponent can be represented as:

$$d = d_3 2^3 + d_2 2^2 + d_1 2 + d_0 = (d_3 2 + d_2)2^2 + d_1 2 + d_0 = ((\underbrace{(d_3)}_{D_1} 2 + d_2)2 + d_1)2 + d_0$$

D_1, D_2, D_3, D_4

The execution of the left-to-right binary exponentiation algorithm is simulated in the following table:

i	d_i	$D_{k-i} = 2D_{k-i-1} + d_i$	x
3	1	$D_1 = 2D_0 + d_3$	$1^2 \cdot y = y$
2	0	$D_2 = 2D_1 + d_2$	$(y)^2$
1	0	$D_3 = 2D_2 + d_1$	$(y^2)^2 = y^4$
0	1	$D_4 = 2D_3 + d_0$	$(y^4)^2 \cdot y = y^9$

Interestingly, by replacing $k - i$ by j and changing the execution order in the recursive formula, the dual algorithm called right-to-left square-and-multiply algorithm can be obtained as provided by Algorithm 4.

Algorithm 4 Right-to-left Binary Exponentiation Algorithm

Input: $y, n, d = (d_{k-1}, d_{k-2}, \ldots, k_1, k_0)_2$
Output: $x = y^d \pmod n$

```
1: t = 1
2: x = y
3: for i = 0 to k − 1 do                    (scan through the key bits d_i from LSB to MSB)
4:     if d_i = 1 then                      (if the key bit d_i is set)
5:         t ← xt (mod n)                   (do the conditional multiplication)
6:     end if
7:     x ← x^2 (mod n)                      (do the squaring always)
8: end for
9: return x
```

4.5 Modular Multiplication

The naïve way of implementing the modular multiplication for two integers a and b is to compute the product ab and then perform the reduction modulo n by computing: $ab \pmod{n} = ab - n\lfloor \frac{ab}{n} \rfloor$. This approach is typically not efficient as it requires an integer division operation, which implementation is typically not efficient and also not included in the instruction set of many small microcontrollers.

The Montgomery algorithm allows to get rid of the integer division by replacing the reduction modulo n by a reduction modulo 2^k, where $k = \lceil \log_2(n) \rceil$ is the size of n in bits. Please note that the implementation of a reduction modulo 2^k is a simple truncation of the input to k-bits.

The Montgomery multiplication MM is defined as follows:

$$MM(a, b, n, z) = abz^{-1} \bmod n$$

The algorithm to compute the Montogomery multiplication is provided by the Algorithm 5.

Algorithm 5 Montgomery Multiplication Algorithm MM

Input: a, b, n, z
Output: $abz^{-1} \bmod n$

1: $c \leftarrow ab$ (integer multiplication)
2: $n' \leftarrow (-n)^{-1} \bmod z$ (precomputation)
3: $d \leftarrow cn' \bmod z$ (reduction is a truncation to k bits being $z = 2^k$)
4: $e \leftarrow c + nd$ (integer multiplication and addition)
5: $f \leftarrow e/z$ (right shift by k bits being $z = 2^k$)
6: **if** $f \geq n$ **then**
7: $\quad f \leftarrow f - n$ (integer substraction)
8: **end if**
9: **return** f

Therefore, the modular multiplication can be computed using four Montgomery multiplications as shown in Algorithm 6, but requires no integer division.

Algorithm 6 Modular Multiplication using the Montgomery Multiplication

Input: a, b, n
Output: $c = ab \bmod n$

1: $z \leftarrow 2^k$
2: $z^2 \leftarrow zz \bmod n$
(Conversion to the Montgomery domain)
3: $a' \leftarrow MM(a, z^2)$ ($a' = az \bmod n$)
4: $b' \leftarrow MM(b, z^2)$ ($b' = bz \bmod n$)
(Multiplication in the Montgomery domain)
5: $c' \leftarrow MM(a', b')$ ($c' = abz \bmod n$)
(Conversion back to the integer domain)
6: $c \leftarrow MM(c', 1)$ ($c = ab \bmod n$)
7: **return** c

The modular multiplication using the Montgomery multiplication is particularly useful when the conversions back and forth the Montgomery domain are only computed once, while many multiplications in the Montgomery domain are performed (*e.g.*, modular exponentiation). Hence, the left-to-right exponentiation algorithm for modular exponentiation using the Montgomery multiplication is provided by the Algorithm 7

Algorithm 7 Left-to-right Binary Exponentiation using the Montgomery Multiplication

Input: $y, n, d = (d_{k-1}, d_{k-2}, \ldots, k_1, k_0)_2$
Output: $x = y^d \bmod n$

1: $z \leftarrow 2^k$ (precomputation)
2: $z^2 \leftarrow zz \bmod n$ (precomputation)
3: $n' \leftarrow (-n)^{-1} \bmod z$ (precomputation)
4: $y' \leftarrow MM(y, z^2)$ (convesion to the Montgomery domain)
5: $x' \leftarrow z$ (convesion to the Montgomery domain)
6: **for** $i = k - 1$ downto 0 **do** (scan through the key bits d_i from MSB to LSB)
7: $\quad x' \leftarrow MM(x', x')$ (do the squaring always)
8: $\quad$ **if** $d_i = 1$ **then** (if the key bit d_i is set)
9: $\quad\quad x' \leftarrow MM(x', y')$ (do the conditional multiplication)
10: $\quad$ **end if**
11: **end for**
12: $x \leftarrow MM(x', 1)$ (convesion back to the integer domain)
13: **return** x

5 Assignment

The goal of the assignment is to implement the *decryption* algorithm of RSA in Python 2.x using the left-to-right square and multiply algorithm and the Montgomery multiplication. For the purpose of the assignment, 64-bits integers must be considered.

The framework is available for download at the address https://tueisec-sica.sec.ei.tum.de/file.php?f=SIKA_Aufgabe_2.zip. The package contains the files `project.nja`, `main.py`, `student.py`.

The file `project.nja` is the project file for the Ninja integrated development environment, cf. http://ninja-ide.org/. The files `main.py` and `student.py` must be used as a skeleton for the implementation. The file `main.py` verifies the correctness of 10 decryptions by displaying the input and output results in hexadecimal and binary format as shown in the Listing 1.

Listing 1: main.py

```
import student, random, sys, time

n = long(0xB935E2B84B83E9EB)
d = long(0xDEADBEEF00211989)

print "n: ", "".join(student.hex64(n)), " (", student.bin64(n), ")"
print "d: ", "".join(student.hex64(d)), " (", student.bin64(d), ")"

start_time = time.clock()

for i in xrange(10):
    y = long(random.getrandbits(64))
    w = pow(y, d, n)
    x = student.RSA_Decrypt(y,d,n)

    print "\n"
    print "y: ", "".join(student.hex64(y)), " (", student.bin64(y), ")"
    print "x: ", "".join(student.hex64(x)), " (", student.bin64(x), ")"
    print "w: ", "".join(student.hex64(w)), " (", student.bin64(w), ")"

    if x != w:
        print "FAILED"
        sys.exit(100)

print "\nALL PASSED"
print "\nExecution time: %f s" % (time.clock()-start_time)
sys.exit(0)
```

The file `student.py` is shown in Listing 2. The `student.py` contains the implementation of the routine `bin64` and `hex64` to convert integers value to their binary and hexadecimal representation, respectively. Furthermore, the implementation of extended Euclidean algorithm is provided by the function `modInvEuclid`. To complete the assignment, the following functions must be implemented as previously described:

- `MontgomeryMul` to compute the Montgomery multiplication
- `MontgomeryExp` to compute the left-to-right binary exponentiation using the Montgomery multiplication

Listing 2: student.py

```
import os, sys, math, numpy

def bin64(x):
    """
    Return binary string representation of x (fixed size 64)
    """
    count=64
    return "".join(map(lambda y:str((x>>y)&1), range(count-1, -1, -1)))

def hex64(x):
    """
    Return hex string representation of x (fixed size 64)
    """
    count=64
    return ["%02X" % (x >> count-(8*(i+1)) & 0x00000000000000FF ) \
        for i in range(count/8) ]

def extEuclideanAlg(a, b) :
    """
    Extended Euclidean algorithm
    """
    if b == 0 :
        return 1,0,a
    else :
        x, y, gcd = extEuclideanAlg(b, a % b)
        return y, x - y * (a // b),gcd

def modInvEuclid(a,n) :
    """
    Computes the modular multiplicative inverse using the
    extended Euclidean algorithm
    Returns: multiplicative inverse of a modulo n
    """
    x,y,gcd = extEuclideanAlg(a,n)
    if gcd == 1 :
        return x % n
    else :
        return None

def MontgomeryMul(a,b,n,n1,z):
    """
    Montgomery Multiplication
    """
    # WRITE YOUR CODE HERE

def MontgomeryExp(y,d,z2,n,n1,z):
    """
    Left-to-right Square & Multiply using Montgomery Multiplication
    """

def RSA_Decrypt(y,d,n):
    # WRITE YOUR CODE HERE
    return 0
```

References

[Chi09] Lindsay N. Childs, *A concrete introduction to higher algebra. 3rd ed.*, 3rd ed. ed., New York, NY: Springer, 2009 (English).

[Die04] M. Dietzfelbinger, *Primality testing in polynomial time: From randomized algorithms to "primes is in p"*, Lecture Notes in Computer Science, Springer, 2004.

[Kob94] Neal Koblitz, *A course in number theory and cryptography. 2nd ed.*, 2nd ed. ed., New York, NY: Springer-Verlag, 1994 (English).

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman, *A method for obtaining digital signatures and public-key cryptosystems*, Commun. ACM **21** (1978), no. 2, 120–126.

[Sti05] Douglas R. Stinson, *Cryptography: Theory and Practice, Third Edition (Discrete Mathematics and Its Applications)*, Chapman & Hall/CRC, 2005.