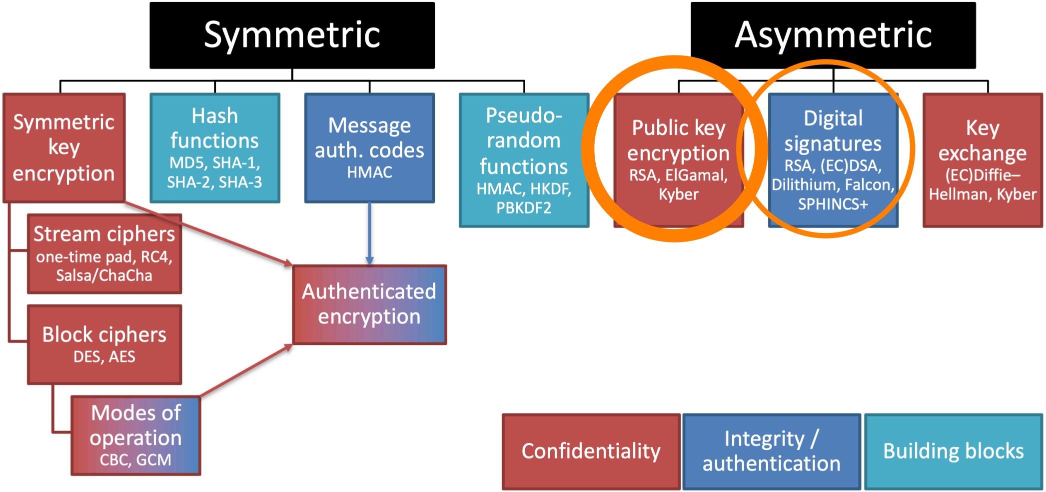
Topic 3.2
Public key cryptography – RSA encryption

Douglas Stebila

CO 487/687: Applied Cryptography

Fall 2024

Map of cryptographic primitives



Outline

RSA encryption

Implementation issues

The RSA encryption scheme

- Ron Rivest, Adi Shamir, and Leonard Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," Communications of the ACM **21** (2): pp. 120–126, 1978.
- Also invented by Clifford Cocks in 1973 (GCHQ); classified until 1997.
- Key generation:
  1. Choose random primes p and q with $\log_2 p \approx \log_2 q \approx \ell/2$.
  2. Compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$.
  3. Choose an integer e with $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
  4. Compute $d = e^{-1} \bmod \varphi(n)$. The public key is (n, e) and the private key is (n, d).
- Message space: $M = C = \mathbb{Z}_n^* = \{m \in \mathbb{Z} : 0 \leq m < n \text{ and } \gcd(m, n) = 1\}$.
- Encryption: $\mathcal{E}((n, e), m) = m^e \bmod n$.
- Decryption: $\mathcal{D}((n, d), c) = c^d \bmod n$.

Integers mod n

Notation:

$$\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\} = \text{the set of integers}$$

$$\mathbb{Z}_n = \begin{cases} \{0, 1, 2, 3, \ldots, n-1\}, & n > 0 \\ \mathbb{Z}, & n = 0 \\ \mathbb{Z}_{-n}, & n < 0 \end{cases}$$

For all $a, b, n \in \mathbb{Z}$:

- We define $a \mid b$ ("a divides b") if there exists $k \in \mathbb{Z}$ such that $ak = b$.
- We define $a \equiv b \pmod{n}$ ("a is congruent to b mod n") if n divides $b - a$.
- We define $a \bmod n$ ("a mod n") to be the unique element $c \in \mathbb{Z}_n$ such that $a \equiv c \pmod{n}$.

Modular arithmetic

Arithmetic in $\mathbb{Z}_n$ is defined as follows:

$$a + b = ((a + b) \bmod n)$$
$$a - b = ((a - b) \bmod n)$$
$$a \cdot b = ((a \cdot b) \bmod n)$$
$$\frac{a}{b} = c \text{ where } c \in \mathbb{Z}_n \text{ is the unique element satisfying}$$
$$b \cdot c = a \text{ in } \mathbb{Z}_n \text{ (if such a unique element exists)}$$

Remark: $\dfrac{a}{b} = a \cdot \dfrac{1}{b}$ whenever either side is defined

Remark: We sometimes write b^{-1} instead of $\frac{1}{b}$.

Arithmetic in $\mathbb{Z}_n$

Computing $a+b$, $a-b$, and $a \cdot b$ in $\mathbb{Z}_n$ is trivial.

As for a/b:

Theorem

Let $a, b \in \mathbb{Z}_n$. Then a/b is defined in $\mathbb{Z}_n$ if and only if $\gcd(b, n) = 1$. Furthermore, when a/b is defined, there is an efficient algorithm to compute its value.

Proof.

Extended Euclidean Algorithm. □

RSA key generation

To generate an RSA public/private key pair:

1. Choose random primes p and q with $\log_2 p \approx \log_2 q \approx \ell/2$.
2. Compute $n = pq$ and $\varphi(n) = (p-1)(q-1)$.
3. Choose an integer e with $1 < e < \varphi(n)$ and $\gcd(e, \varphi(n)) = 1$.
4. Compute $d = e^{-1}$ in $\mathbb{Z}_{\varphi(n)}$.
   4.1 Use the Extended Euclidean Algorithm to compute d.
   If the Extended Euclidean Algorithm succeeds, then you are guaranteed that $\gcd(e, \varphi(n)) = 1$.

   Note: $de \equiv 1 \pmod{\varphi(n)}$ by definition of e^{-1}.
5. The public key is (n, e).
6. The private key is (n, d).

RSA encryption and decryption

Recall that (n, e) is the public key and (n, d) is the private key.

The encryption and decryption functions are:

- Encryption: $\mathcal{E}((n, e), m) = m^e \bmod n$.
- Decryption: $\mathcal{D}((n, d), c) = c^d \bmod n$.

Two questions:

- Efficiency?
- Correctness?

Correctness of RSA

Theorem

Let (n, e) *be an RSA public key with private key* (n, d)*. Then*

$$\mathcal{D}((n, d), \mathcal{E}((n, e), m)) = m$$

for all $m \in \mathbb{Z}_n$ *such that* $\gcd(m, n) = 1$.

Theorem (Fermat's Little Theorem)

Let p be a prime. For all integers a, it holds that

$$a^p \equiv a \bmod p$$

Moreover, if a is coprime to p, then

$$a^{p-1} \equiv 1 \bmod p$$

Some elementary number theory

Definition (Euler's phi function a.k.a. Euler's totient function)

Let $\varphi(n)$ denote the number of integers $k \in [1, n]$ that are coprime with n.

Theorem (Formula for Euler's phi function)

- $\varphi(p) = p - 1$, *if* p *is prime*
- $\varphi(pq) = (p-1)(q-1)$, *if* p *and* q *are both prime*
- $\varphi(n) = p_1^{e_1-1}(p_1 - 1) \dots p_r^{e_r-1}(p_r - 1)$, *if* n *has prime factorization* $p_1^{e_1} \dots p_r^{e_r}$

Theorem (Euler's Theorem)

Let n *be a non-negative integer. For all integers* a *coprime to* n*, it holds that*

$$a^{\varphi(n)} \equiv 1 \bmod n$$

Correctness of RSA

Correctness of RSA.

Assume $\gcd(m, n) = 1$. In particular, $p \nmid m$ and $q \nmid m$.

1. By definition of $\mathcal{E}$ and $\mathcal{D}$:

$$\mathcal{D}((n, d), \mathcal{E}((n, e), m)) = \mathcal{D}((n, d), m^e \bmod n) = (m^e \bmod n)^d \bmod n = m^{ed} \bmod n.$$

2. By definition of e and d, we have $ed \equiv 1 \pmod{\varphi(n)}$. In other words, $ed = 1 + k \cdot \varphi(n)$ for some $k \in \mathbb{Z}$.
3. Since $m \not\equiv 0 \pmod p$, Fermat's Little Theorem implies: $m^{\varphi(n)} = m^{(p-1)(q-1)} = (m^{p-1})^{q-1} \equiv 1^{q-1} = 1 \pmod p$.
4. Similarly, $m^{\varphi(n)} \equiv 1 \pmod q$, and hence $m^{\varphi(n)} \equiv 1 \pmod n$.
5. Therefore

$$m^{ed} = m^{1+k\cdot\varphi(n)} = m^1 \cdot (m^{\varphi(n)})^k \equiv m^1 \cdot 1^k = m \pmod n. \qquad \square$$

Outline

RSA encryption

Implementation issues

Non-trivial algorithms involved in implementing RSA:

- How to obtain random numbers?
- How to generate random large primes?
- How to compute $\gcd(e, \varphi(n))$?
- How to compute $e^{-1} \bmod \varphi(n)$?
- How to compute $m^e \bmod n$ for (potentially) large e?
- How to compute $c^d \bmod n$ for (potentially) large d?

The difficult part is performing these operations *efficiently*.

Basic concepts from complexity theory

- An algorithm is a "well-defined computational procedure" (e.g., a Turing machine) that takes a variable input and eventually halts with some output.
  - For an integer factorization algorithm, the input is a positive integer n, and the output is the prime factorization of n.
- The efficiency of an algorithm is measured by the scarce resources it consumes (e.g. time, space, number of processors).
- The input size is the number of bits required to write down the input using a reasonable encoding.
  - The size of a positive integer n is $\lfloor \log_2 n \rfloor + 1$ bits.
  - Exception: The input size of $1^\ell = \underbrace{111\ldots1}_{\ell}$ is ℓ.

Basic concepts from complexity theory

- The running time of an algorithm is an upper bound as a function of the input size, of the worst case number of basic steps the algorithm takes over all inputs of a fixed size.
- An algorithm is a polynomial-time (efficient) algorithm if its (expected) running time is $O(\ell^c)$, where c is a fixed positive integer, and ℓ is the input size.
- Recall that if $f(n)$ and $g(n)$ are functions from the positive integers to the positive real numbers, then $f(n) = O(g(n))$ means that there exists a positive constant c and a positive integer n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$.
    - For example, $7.5n^3 + 1000n^2 - 99 = O(n^3)$.

Basic integer operations

Input: Two ℓ-bit positive integers a and b.

Input size: $O(\ell)$ bits.

Operation	Running time of naive algorithm (in bit operations)
Addition: $a + b$	$O(\ell)$
Subtraction: $a - b$	$O(\ell)$
Multiplication: $a \cdot b$	$O(\ell^2)$
Division: $a = qb + r$	$O(\ell^2)$
GCD: $\gcd(a, b)$	$O(\ell^2)$

Greatest common divisor

Algorithm 1 Algorithm for computing $\gcd(a, b)$

Given: $a, b \in \mathbb{N}$

if $b = 0$ **then**
    output a
else
    output $\gcd(b, a \bmod b)$

Basic Modular Operations

Input: An ℓ-bit integer n, and integers $a, b \in [0, n-1]$.

Input size: $O(\ell)$ bits.

Operation	Running time of naive algorithm (in bit operations)
Addition: $a + b \bmod n$	$O(\ell)$
Subtraction: $a - b \bmod n$	$O(\ell)$
Multiplication: $a \cdot b \bmod n$	$O(\ell^2)$
Inversion: $a^{-1} \bmod n$	$O(\ell^2)$
Exponentiation: $a^b \bmod n$	$O(\ell^3)$

Extended Euclidean algorithm

Given $a, b \in \mathbb{N}$, with $d = \gcd(a, b)$, an *extended gcd* of a and b is a pair of integers (x, y) such that $ax + by = d$.

Algorithm 2 Algorithm for computing $\text{extended\_gcd}(a, b)$

Given: $a, b \in \mathbb{N}$

1: **if** $b = 0$ **then**
2: output $(1, 0)$
3: **else**
4: $q \leftarrow \lfloor \frac{a}{b} \rfloor$
5: $r \leftarrow a \bmod b$
6: $(s, t) \leftarrow \text{extended\_gcd}(b, r)$
7: output $(t, s - qt)$

Complexity analysis: At most $O(\log_2 q)$ operations per division step, and $O(\ell^2)$ operations overall. (If q is large then there will be fewer subsequent steps.)

Computing modular inverses

Algorithm 3 Algorithm for computing $a^{-1} \bmod n$

Given: $a, n \in \mathbb{N}$, $a < n$, $\gcd(a, n) = 1$

1: $(x, y) \leftarrow \text{extended\_gcd}(a, n)$
2: output $x \bmod n$

This works because $ax + ny = 1$, so $ax \equiv 1 \bmod n$.

Modular exponentiation: naive versions

Input: An ℓ-bit integer n, and integers $a, b \in [1, n-1]$.

Output: $a^b \bmod n$.

Algorithm 4 Naive algorithm for computing $a^b \bmod n$

Given: $a, b, n \in \mathbb{N}$

1: $d \leftarrow a^b$ (in $\mathbb{Z}$)
2: output $d \bmod n$

Algorithm 5 Another naive algorithm for computing $a^b \bmod n$

Given: $a, b, n \in \mathbb{N}$

1: $A \leftarrow a$
2: **for** $i = 2$ to b **do**
3: $\quad A \leftarrow A \cdot a \bmod n$
4: output A

Modular exponentiation: naive version

The naive versions of modular exponentiation are inefficient and have exponential runtime.

The "for loop" in the naive algorithm is **for** $i = 2$ **to** b where $b \in [1, n-1]$; in other words, it takes roughly n iterations.

The input to the algorithm are three numbers a, b, n each of which is at most $\lceil \log_2 n \rceil$ bits long. So the length of the input is 3ℓ where $\ell = \lceil \log_2 n \rceil$.

The runtime of the algorithm is at least $O(n) = O(2^\ell)$ which is exponential in the input size ℓ.

Modular exponentiation: square and multiple algorithm

Example ($a^b \bmod n$)

- Let $n = 851$, $a = 3$, $b = 631$. Write $b = 631$ in binary:

$$b = 631 = 2^9 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0.$$

- Square: Compute successive powers of $a = 3$ modulo n:

$$\begin{array}{ccc} 3 \equiv 3 \pmod{851} & 3^2 \equiv 9 \pmod{851} & 3^{2^2} \equiv 81 \pmod{851} \\ 3^{2^3} \equiv 604 \pmod{851} & 3^{2^4} \equiv 588 \pmod{851} & 3^{2^5} \equiv 238 \pmod{851} \\ 3^{2^6} \equiv 478 \pmod{851} & 3^{2^7} \equiv 416 \pmod{851} & 3^{2^8} \equiv 303 \pmod{851} \\ 3^{2^9} \equiv 752 \pmod{851}. & & \end{array}$$

- Multiply:

$$\begin{aligned} 3^{631} &= 3^{2^9+2^6+2^5+2^4+2^2+2^1+2^0} \\ &= 3^{2^9} \cdot 3^{2^6} \cdot 3^{2^5} \cdot 3^{2^4} \cdot 3^{2^2} \cdot 3^{2^1} \cdot 3^{2^0} \\ &\equiv 752 \cdot 478 \cdot 238 \cdot 588 \cdot 81 \cdot 9 \cdot 3 \pmod{851} \qquad \equiv 817 \pmod{851}. \end{aligned}$$

Modular exponentiation: square-and-multiply algorithm, sequential version

Algorithm 6 Sequential algorithm for computing $a^b \bmod n$.

Given: $a, b, n \in \mathbb{N}$

1: Write $b = b_{t-1} b_{t-2} \ldots b_1 b_0$ in binary.

2: $y_0 \leftarrow a$

3: **for** $i = 1, \ldots, t-1$: $a_i \leftarrow y_{i-1}^2 \bmod n$

4: $z \leftarrow 1$

5: **for** $i = 0, \ldots, t-1$: **if** $b_i = 1$ **then** $z \leftarrow z \cdot y_i \bmod n$

6: **return** z

Each **for** loop has $\ell = \log_2 n$ iterations, and each iteration does at most one modular multiplication ($O(\ell^2)$ runtime) so the total runtime is polynomial in the input size: $O(\ell^3)$.

Modular exponentiation: square-and-multiply algorithm, loop version

Here's an equivalent formulation of the double and add algorithm as a loop.

Algorithm 7 Iterative algorithm for computing $a^b \bmod n$.

Given: $a, b, n \in \mathbb{N}$

```
Write b = b_{t-1} b_{t-2} ... b_1 b_0 in binary.
z ← 1
for i from t − 1 down to 0 do
    z ← 2z mod n
    if b_i = 1 then
        z ← z · a mod n
return z
```

Modular exponentiation: square-and-multiply algorithm, recursive version

Here's an equivalent recursive version:

Algorithm 8 Recursive algorithm for computing $a^b \bmod n$.

Given: $a, b, n \in \mathbb{N}$

if $b = 0$ **then**
    output 1
else if b is even **then**
    output $(a^{\frac{b}{2}} \bmod n)^2 \bmod n$
else if b is odd **then**
    output $(a^{b-1} \bmod n) \cdot (a \bmod n) \bmod n$

Alice does the following:

1. Selects primes $p = 23$ and $q = 37$.
2. Computes $n = pq = 851$ and $\varphi(n) = (p-1)(q-1) = 792$.
3. Selects $e = 631$ satisfying $\gcd(631, 792) = 1$.
4. Solves $631d \equiv 1 \pmod{792}$ to get $d \equiv -305 \equiv 487 \pmod{792}$, and selects $d = 487$.
5. Alice's public key is $(n = 851,\ e = 631)$;
   her private key is $d = 487$.

Toy Example: RSA Encryption

To encrypt a message $m = 2$ for Alice, Bob does:

1. Obtains Alice's public key ($n = 851$, $e = 631$).
2. Computes $c = 2^{631} \bmod 851$ using the repeated-square-and-multiply algorithm:
   (a) Write $e = 631$ in binary:
   $$e = 2^9 + 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0.$$
   (b) Compute successive squarings of $m = 2$ modulo $n = 851$:
   $$\begin{aligned} 2 &\equiv 2 \pmod{851} & 2^2 &\equiv 4 \pmod{851} & 2^{2^2} &\equiv 16 \pmod{851} \\ 2^{2^3} &\equiv 256 \pmod{851} & 2^{2^4} &\equiv 9 \pmod{851} & 2^{2^5} &\equiv 81 \pmod{851} \\ 2^{2^6} &\equiv 604 \pmod{851} & 2^{2^7} &\equiv 588 \pmod{851} & 2^{2^8} &\equiv 238 \pmod{851} \\ 2^{2^9} &\equiv 478 \pmod{851}. \end{aligned}$$

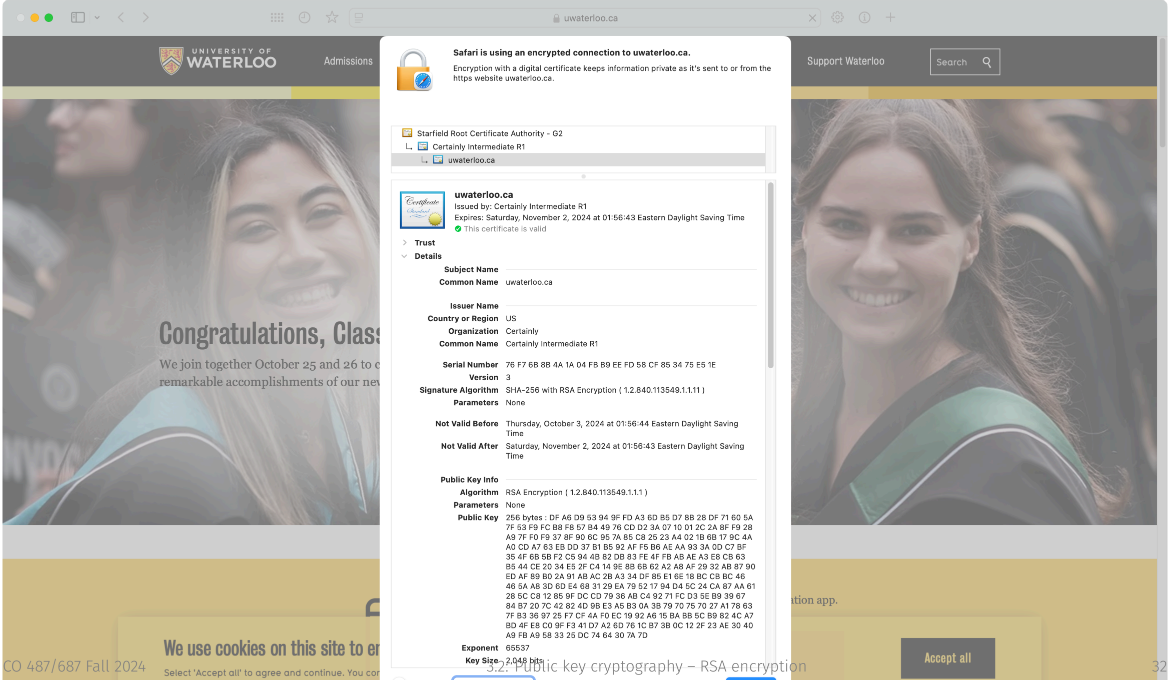
(c) Multiply together the squares 2^{2^i} for which the ith bit (where $0 \leq i \leq 9$) of the binary representation of 631 is 1:

$$\begin{aligned} 2^{631} &= 2^{2^9+2^6+2^5+2^4+2^2+2^1+2^0} \\ &= 2^{2^9} \cdot 2^{2^6} \cdot 2^{2^5} \cdot 2^{2^4} \cdot 2^{2^2} \cdot 2^{2^1} \cdot 2^{2^0} \\ &\equiv 478 \cdot 604 \cdot 81 \cdot 9 \cdot 16 \cdot 4 \cdot 2 \pmod{851} \\ &\equiv 775 \pmod{851}. \end{aligned}$$

3. Bob sends $c = 775$ to Alice.

To decrypt $c = 775$, Alice uses her private key $d = 487$ as follows:

1. Compute $m = 775^{487} \bmod 851$ to get $m = 2$.


uwaterloo.ca
UNIVERSITY OF WATERLOO
Admissions
Support Waterloo
Search
Congratulations, Class
We join together October 25 and 26 to c
remarkable accomplishments of our nev
ation app.
We use cookies on this site to er
Select 'Accept all' to agree and continue. You cor
Accept all
Safari is using an encrypted connection to uwaterloo.ca.
Encryption with a digital certificate keeps information private as it's sent to or from the https website uwaterloo.ca.
Starfield Root Certificate Authority - G2
Certainly Intermediate R1
uwaterloo.ca
uwaterloo.ca
Issued by: Certainly Intermediate R1
Expires: Saturday, November 2, 2024 at 01:56:43 Eastern Daylight Saving Time
This certificate is valid
Trust
Details
Subject Name
Common Name uwaterloo.ca
Issuer Name
Country or Region US
Organization Certainly
Common Name Certainly Intermediate R1
Serial Number 76 F7 6B 8B 4A 1A 04 FB B9 EE FD 58 CF 85 34 75 E5 1E
Version 3
Signature Algorithm SHA-256 with RSA Encryption (1.2.840.113549.1.1.11)
Parameters None
Not Valid Before Thursday, October 3, 2024 at 01:56:44 Eastern Daylight Saving Time
Not Valid After Saturday, November 2, 2024 at 01:56:43 Eastern Daylight Saving Time
Public Key Info
Algorithm RSA Encryption (1.2.840.113549.1.1.1)
Parameters None
Public Key 256 bytes : DF A6 D9 53 94 9F FD A3 6D B5 D7 8B 28 DF 71 60 5A 7F 53 F9 FC B8 F8 57 B4 49 76 CD D2 3A 07 10 01 2C 2A 8F F9 28 A9 7F F0 F9 37 8F 90 6C 95 7A 85 C8 25 23 A4 02 1B 6B 17 9C 4A A0 CD A7 63 EB DD 37 B1 B5 92 AF F5 B6 AE AA 93 3A 0D C7 BF 35 4F 6B 5B F2 C5 94 4B 82 DB 83 FE 4F FB AB AE A3 E8 CB 63 B5 44 CE 20 34 E5 2F C4 14 9E 8B 6B 62 A2 A8 AF 29 32 AB 87 90 ED AF 89 B0 2A 91 AB AC 2B A3 34 DF 85 E1 6E 18 BC CB BC 46 46 5A A8 3D 6D E4 68 31 29 EA 79 52 17 94 D4 5C 24 CA 87 AA 61 28 5C C8 12 85 9F DC CD 79 36 AB C4 92 71 FC D3 5E B9 39 67 84 B7 20 7C 42 82 4D 9B E3 A5 B3 0A 3B 79 70 75 70 27 A1 78 63 7F B3 36 97 25 F7 CF 4A F0 EC 19 92 A6 15 BA BB 5C B9 82 4C A7 BD 4F E8 C0 9F F3 41 D7 A2 6D 76 1C B7 3B 0C 12 2F 23 AE 30 40 A9 FB A9 58 33 25 DC 74 64 30 7A 7D
Exponent 65537
Key Size 2,048 bits