RSA Algorithm

The RSA algorithm is named after Ron Rivest, Adi Shamir and Len Adleman, who invented it in 1977 [RIVE78]. The basic technique was first discovered in 1973 by Clifford Cocks [COCK73] of CESG (part of the British GCHQ) but this was a secret until 1997. The patent taken out by RSA Labs has expired.

The RSA cryptosystem is the most widely-used public key cryptography algorithm in the world. It can be used to encrypt a message without the need to exchange a secret key separately.

The RSA algorithm can be used for both public key encryption and digital signatures. Its security is based on the difficulty of factoring large integers.

Party A can send an encrypted message to party B without any prior exchange of secret keys. A just uses B's public key to encrypt the message and B decrypts it using the private key, which only he knows. RSA can also be used to sign a message, so A can sign a message using their private key and B can verify it using A's public key.

We look into the mathematics behind the algorithm on our RSA Theory page.

Our pages on public key cryptography using discrete logarithms look at a different kind of public key cryptography which relies on the difficulty of solving the discrete logarithm problem.

This page modified on 9 June 2018 to use the MathJax display engine for mathematics.

Contents

Recommended reading

- Cryptography Engineering by Niels Ferguson, Bruce Schneier and Tadayoshi Kohno. Look Inside
- Handbook of Applied Cryptography by Alfred J. Menezes, Paul C. van Oorschot and Scott A. Vanstone
- Cryptanalysis of RSA and Its Variants by M. Jason Hinek

Join Amazon Student FREE Two-Day Shipping for College Students

Affiliate disclosure: we get a small commission for purchases made through the above links

[Go to top]

Key Generation Algorithm

This is the original algorithm.

1. Generate two large random primes, p and q, of approximately equal size such that their product $n = pq$ is of the required bit length, e.g. 1024 bits. [See note 1].
2. Compute $n = pq$ and $\phi = (p-1)(q-1)$. [See note 6].
3. Choose an integer e, $1 < e < \phi$, such that $\gcd(e, \phi) = 1$. [See note 2].
4. Compute the secret exponent d, $1 < d < \phi$, such that $ed \equiv 1 \bmod \phi$. [See note 3].
5. The public key is (n, e) and the private key (d, p, q). Keep all the values d, p, q and ϕ secret. [Sometimes the private key is written as (n, d) because you need the value of n when using d. Other times we might write the key pair as $((N, e), d)$.]

- n is known as the *modulus*.
- e is known as the *public exponent* or *encryption exponent* or just the *exponent*.
- d is known as the *secret exponent* or *decryption exponent*.

A practical key generation algorithm

Incorporating the advice given in the notes below, a practical algorithm to generate an RSA key pair is given below. Typical bit lengths are $k = 1024, 2048, 3072, 4096, \ldots$, with increasing computational expense for larger values. You will not go far wrong if you choose e as 65537 (=0x10001) in step (1).

Algorithm: Generate an RSA key pair.

INPUT: Required modulus bit length, k.
OUTPUT: An RSA key pair $((N, e), d)$ where N is the modulus, the product of two primes ($N = pq$) not exceeding k bits in length; e is the public exponent, a number less than and coprime to $(p-1)(q-1)$; and d is the private exponent such that $ed \equiv 1 \bmod (p-1)(q-1)$.

```
1. Select a value of e from $3, 5, 17, 257, 65537$
2. repeat
3.    p ← genprime(k/2)
4. until $(p \bmod e) \neq 1$
5. repeat
6.    q ← genprime(k - k/2)
7. until $(q \bmod e) \neq 1$
8. N ← pq
9. L ← (p-1)(q-1)
10. d ← modinv(e, L)
11. return $(N, e, d)$
```

The function `genprime(b)` returns a prime of exactly b bits, with the bth bit set to 1. Note that the operation $k/2$ is *integer* division giving the integer quotient with no fraction.

If you've chosen $e = 65537$ then the chances are that the first prime returned in steps (3) and (6) will pass the tests in steps (4) and (7), so each repeat-until loop will most likely just take one iteration. The final value of N may have a bit length slightly short of the target k. This actually does not matter too much (providing the message m is always < N), but some schemes require a modulus of exact length. If this is the case, then just repeat the entire algorithm until you get one. It should not take too many goes. Alternatively, use the trick setting the two highest bits in the prime candidates described in note 1.

Encryption

Sender A does the following:-

1. Obtains the recipient B's public key (n, e).
2. Represents the plaintext message as a positive integer m with $1 < m < n$ [see note 4].
3. Computes the ciphertext $c = m^e \bmod n$.
4. Sends the ciphertext c to B.

Decryption

Recipient B does the following:-

1. Uses his private key (n, d) to compute $m = c^d \bmod n$.
2. Extracts the plaintext from the message representative m.

Digital signing

Sender A does the following:-

1. Creates a *message digest* of the information to be sent.
2. Represents this digest as an integer m between 1 and $n-1$ [See note 5].
3. Uses her *private* key (n, d) to compute the signature $s = m^d \bmod n$.
4. Sends this signature s to the recipient, B.

Signature verification

Recipient B does the following (*older method*):-

1. Uses sender A's public key (n, e) to compute integer $v = s^e \bmod n$.

2. Extracts the message digest H from this integer.
3. Independently computes the message digest H' of the information that has been signed.
4. If both message digests are identical, i.e. $H = H'$, the signature is valid.

More secure method:-

1. Uses sender A's public key (n, e) to compute integer $v = s^e \bmod n$.
2. Independently computes the message digest H' of the information that has been signed.
3. Computes the expected representative integer v' by encoding the expected message digest H'.
4. If $v = v'$, the signature is valid.

Notes on practical applications

1. To generate the primes p and q, generate a random number of bit length $k/2$ where k is the required bit length of the modulus n; set the low bit (this ensures the number is odd) and set the *two* highest bits (this ensures that the high bit of n is also set); check if prime (use the *Rabin-Miller* test); if not, increment the number by two and check again until you find a prime. This is p. Repeat for q starting with a random integer of length $k - k/2$. If $p < q$, swop p and q (this only matters if you intend using the CRT form of the private key). In the extremely unlikely event that $p = q$, check your random number generator! Alternatively, instead of incrementing by 2, just generate another random number each time.

   There are stricter rules in ANSI X9.31 to produce *strong primes* and other restrictions on p and q to minimise the possibility of certain techniques being used against the algorithm. There is much argument about this topic. It is probably better just to use a longer key length.

2. In practice, common choices for e are 3, 5, 17, 257 and 65537 $(2^{16} + 1)$. These particular values are chosen because they are primes and make the modular exponentiation operation faster, having only two bits of value 1.

   --> *Aside:* These five numbers are the first five *Fermat numbers*, referred to as F_0 to F_4, where $F_x = 2^{2^x} + 1$. Just be careful, these first five Fermat numbers are prime ("Fermat primes"), but the numbers F_5 and above are not prime. For example, $F_5 = 4294967297 = 641 \times 6700417$.

   The usual choice for e is $F_4 = 65537$ = 0x10001. Also, having chosen e, it is simpler to test whether $\gcd(e, p-1) = 1$ and $\gcd(e, q-1) = 1$ while generating and testing the primes in step 1. Values of p or q that fail this test can be rejected there and then.

   Even better: if e is an odd prime then you can do the less-expensive test $(p \bmod e) \neq 1$ instead of $\gcd(p-1, e) = 1$.

   Why is that? If e is an odd prime then $\gcd(p-1, e) \neq 1$ if and only if $p-1$ is a multiple of e. If $p-1$ is a multiple of e then $p-1 \equiv 0 \pmod e$ or $p \equiv 1 \pmod e$. Conversely, if $p \not\equiv 1 \pmod e$ then $p-1 \not\equiv 0 \pmod e$ and $p-1$ is not a multiple of e. So $\gcd(p-1, e) = 1$.
3. To compute the value for d, use the *Extended Euclidean Algorithm* to calculate $d = e^{-1} \bmod \phi$, also written $d = (1/e) \bmod \phi$. This is known as *modular inversion*. Note that this is not integer division. The modular inverse d is defined as the integer value such that $ed = 1 \bmod \phi$. It only exists if e and ϕ have no common factors.

   For more details of the extended Euclidean algorithm, see our page The Euclidean Algorithm and the Extended Euclidean Algorithm which shows how to use the Euclidean algorithm, answer exam questions on it, and gives source code for an implementation.

4. When representing the plaintext octets as the representative integer m, it is important to add random padding characters to make the size of the integer m large and less susceptible to certain types of attack. If m = 0 or 1 or n-1 there is no security as the ciphertext has the same value. For more details on how to represent the plaintext octets as a suitable representative integer m, see PKCS#1 Schemes below or the reference itself [PKCS1]. It is important to make sure that $m < n$ otherwise the algorithm will fail. This is usually done by making sure the first octet of m is equal to 0x00.
5. Decryption and signing are identical as far as the mathematics is concerned as both use the private key. Similarly, encryption and verification both use the same mathematical operation with the public key. That is, mathematically, for $m < n$,

   $$m = (m^e \bmod n)^d \bmod n = (m^d \bmod n)^e \bmod n$$

   However, note these important differences in implementation:-

   - The signature is derived from a message digest of the original information. The recipient will need to follow exactly the same process to derive the message digest, using an identical set of data.
   - The recommended methods for deriving the representative integers are different for encryption and signing (encryption involves random padding, but signing uses the same padding each time).
6. The original definition of RSA uses the Euler totient function $\phi(n) = (p-1)(q-1)$. More recent standards use the *Charmichael function* $\lambda(n) = \operatorname{lcm}(p-1, q-1)$ instead. $\lambda(n)$ is smaller than $\phi(n)$ and divides it. The value of d' computed by $d' = e^{-1} \bmod \lambda(n)$ is usually different from that derived by $d = e^{-1} \bmod \phi(n)$, but the end result is the same. Both d and d' will decrypt a message $m^e \bmod n$ and both will give the same signature value $s = m^d \bmod n = m^{d'} \bmod n$. To compute $\lambda(n)$, use the relation

   $$\lambda(n) = \frac{(p-1)(q-1)}{\gcd(p-1, q-1)}.$$

7. You might ask if there is a way to find the factors of n given just d and e. This is possible.

   For more details, see our page RSA: how to factorize N given d.

Summary of RSA

- $n = pq$, where p and q are distinct primes.
- $\phi = (p-1)(q-1)$
- $e < n$ such that $\gcd(e, \phi) = 1$
- $d = e^{-1} \bmod \phi$
- $c = m^e \bmod n, 1 < m < n$
- $m = c^d \bmod n$

For more on the theory and mathematics behind the algorithm, see the RSA Theory page.

Key length

When we talk about the *key length* of an RSA key, we are referring to the length of the modulus, n, in bits. The minimum recommended key length for a secure RSA transmission is currently at least 1024 bits. A key length of 512 bits is no longer considered secure, although cracking it is still not a trivial task for the likes of you and me. The longer your information needs to be kept secure, the longer the key you should use. Keep up to date with the latest recommendations in the security journals.

There is one small area of confusion in defining the key length. One convention is that the key length is the position of the most significant bit in n that has value '1', where the least significant bit is at position 1. Equivalently, key length = $\lceil \log_2(n+1)) \rceil$, where $\lceil x \rceil$ is the *ceiling function*, the least integer greater than or equal to x. The other convention, sometimes used, is that the key length is the number of bytes needed to store n multiplied by eight, i.e. $\lceil \log_{256}(n+1) \rceil \times 8$.

The key used in the RSA Example paper [KALI93] is an example. The modulus is represented in hex form as

```
0A 66 79 1D C6 98 81 68 DE 7A B7 74 19 BB 7F B0
C0 01 C6 27 10 27 00 75 14 29 42 E1 9A 8D 8C 51
D0 53 B3 E3 78 2A 1D E5 DC 5A F4 EB E9 94 68 17
01 14 A1 DF E6 7C DC 9A 9A F5 5D 65 56 20 BB AB
```

The most significant byte 0x0A in binary is 00001010'B. The most significant bit is at position 508, so its key length is 508 bits. On the other hand, this value needs 64 bytes to store it, so the key length could also be referred to by some as 64 x 8 = 512 bits. We prefer the former method. You can get into difficulties with the X9.31 method for signatures if you use the latter convention.

Minimum key lengths

The following table is taken from NIST's Recommendation for Key Management [NIST-80057]. It shows the recommended comparable key sizes for symmetrical block ciphers (AES and Triple DES) and the RSA algorithm. That is, the key length you would need to use to have comparable security.

Symmetric key algorithm	Comparable RSA key length	Comparable hash function	Bits of security
2TDEA*	1024	SHA-1	80
3TDEA	2048	SHA-224	112
AES-128	3072	SHA-256	128
AES-192	7680	SHA-384	192
AES-256	15360	SHA-512	256

* 2TDEA is 2-key triple DES - see What's two-key triple DES encryption.

Note just how huge (and impractical) an RSA key needs to be for comparable security with AES-192 or AES-256 (although these two algorithms have had some weaknesses exposed recently; AES-128 is unaffected).

The above table is a few years old now and may be out of date. Existing cryptographic algorithms only get weaker as attacks get better.

Computational Efficiency and the Chinese Remainder Theorem (CRT)

Key generation is only carried out occasionally and so computational efficiency is less of an issue.

The calculation $y = x^e \bmod n$ is known as *modular exponentiation* and one efficient method to carry this out on a computer is the *binary left-to-right method*. To solve, let e be represented in base 2 as

$$e = e_{k-1}e_{k-2} \ldots e_1 e_0$$

where e_{k-1} is the most significant non-zero bit and bit e_0 the least.

```
set y = x
for bit j = k - 2 downto 0
begin
  y = y * y mod n   /* square */
  if e(j) == 1 then
    y = y * x mod n  /* multiply */
end
return y
```

The time to carry out modular exponentation increases with the number of bits set to one in the exponent e. For encryption, an appropriate choice of e can reduce the computational effort required to carry out the computation of $c = m^e \bmod n$. Popular choices like 3, 17 and 65537 are all primes with only two bits set: 3 = 0011'B, 17 = 0x11, 65537 = 0x10001.

The bits in the decryption exponent d, however, will not be so convenient and so decryption using the standard method of modular exponentiation will take longer than encryption. Don't make the mistake of trying to contrive a small value for d; it will not be secure.

An alternative method of representing the private key uses the The Chinese Remainder Theorem (CRT).

For an explanation of how the CRT is used with RSA, see Using the CRT with RSA.

The private key is represented as a quintuple (p, q, dP, dQ, and qInv), where p and q are prime factors of n, dP and dQ are known as the *CRT exponents*, and qInv is the *CRT coefficient*. The CRT method of decryption is about four times faster overall than calculating $m = c^d \bmod n$. The extra values for the private key are:-

```
dP = (1/e) mod (p-1)
dQ = (1/e) mod (q-1)
qInv = (1/q) mod p  where p > q
```

where the `(1/e)` notation means the *modular inverse* (see note 3 above). These values are pre-computed and saved along with p and q as the private key. To compute the message m given c do the following:-

```
m1 = c^dP mod p
m2 = c^dQ mod q
h = qInv(m1 - m2) mod p
m = m2 + hq
```

Even though there are more steps in this procedure, the modular exponentation to be carried out uses much shorter exponents and so it is less expensive overall.

[2008-09-02] Chris Becke has pointed out that most large integer packages will fail when computing `h` if `m1 < m2`. This can be easily fixed by computing

```
h = qInv(m1 + p - m2) mod p
```

or, alternatively, as we do it in our BigDigits implementation of RSA,

```
 if (bdCompare(m1, m2) < 0)
    bdAdd(m1, m1, p);
 bdSubtract(m1, m1, m2);
 /* Let h = qInv ( m_1 - m_2 ) mod p. */
 bdModMult(h, qInv, m1, p);
```

A very simple example of RSA encryption

This is an extremely simple example using numbers you can work out on a pocket calculator (those of you over the age of 35 45 55 can probably even do it by hand).

1. Select primes p=11, q=3.
2. n = pq = 11.3 = 33
   phi = (p-1)(q-1) = 10.2 = 20
3. Choose e=3
   Check gcd(e, p-1) = gcd(3, 10) = 1 (i.e. 3 and 10 have no common factors except 1),
   and check gcd(e, q-1) = gcd(3, 2) = 1
   therefore gcd(e, phi) = gcd(e, (p-1)(q-1)) = gcd(3, 20) = 1
4. Compute d such that ed ≡ 1 (mod phi)
   i.e. compute d = (1/e) mod phi = (1/3) mod 20
   i.e. find a value for d such that phi divides (ed-1)
   i.e. find d such that 20 divides 3d-1.
   Simple testing (d = 1, 2, ...) gives d = 7
   Check: ed-1 = 3.7 - 1 = 20, which is divisible by phi.
5. Public key = (n, e) = (33, 3)
   Private key = (n, d) = (33, 7).

This is actually the smallest possible value for the modulus n for which the RSA algorithm works.
Now say we want to encrypt the message m = 7,

$$c = m^e \bmod n = 7^3 \bmod 33 = 343 \bmod 33 = 13.$$

Hence the ciphertext c = 13.
To check decryption we compute

$$m' = c^d \bmod n = 13^7 \bmod 33 = 7.$$

Note that we don't have to calculate the full value of 13 to the power 7 here. We can make use of the fact that

$$a = bc \bmod n = (b \bmod n) \cdot (c \bmod n) \bmod n$$

so we can break down a potentially large number into its components and combine the results of easier, smaller calculations to calculate the final value.
One way of calculating m' is as follows:-

Note that any number can be expressed as a sum of powers of 2. In particular 7 = 4 + 2 + 1.
So first compute values of $13^2, 13^4, 13^8, \ldots$ by repeatedly squaring successive values modulo 33.
$13^2 = 169 \equiv 4, 13^4 = 4.4 = 16, 13^8 = 16.16 = 256 \equiv 25.$
Then, since 7 = 4 + 2 + 1, we have $m' = 13^7 = 13^{(4+2+1)} = 13^4.13^2.13^1$
$\equiv 16 \times 4 \times 13 = 832 \equiv 7 \pmod{33}$

Now if we calculate the ciphertext c for all the possible values of m (0 to 32), we get

```
m  0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
c  0  1  8 27 31 26 18 13 17  3 10 11 12 19  5  9  4

m 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
c 29 24 28 14 21 22 23 30 16 20 15  7  2  6 25 32
```

Note that all 33 values of m (0 to 32) map to a unique code c in the same range in a sort of random manner. In this case we have nine values of m that map to the same value of c - these are known as *unconcealed messages*. m = 0, 1 and n-1 will always do this for any n, no matter how large. But in practice, these shouldn't be a problem when we use large values for n in the order of several hundred bits.

If we wanted to use this system to keep secrets, we could let A=2, B=3, ..., Z=27. (We specifically avoid 0 and 1 here for the reason given above). Thus the plaintext message "HELLOWORLD" would be represented by the set of integers $m_1, m_2, \ldots$

$$(9, 6, 13, 13, 16, 24, 16, 19, 13, 5)$$

Using our table above, we obtain ciphertext integers $c_1, c_2, \ldots$

$$(3, 18, 19, 19, 4, 30, 4, 28, 19, 26)$$

Note that this example is no more secure than using a simple Caesar substitution cipher, but it serves to illustrate a simple example of the mechanics of RSA encryption.

Remember that calculating $m^e \bmod n$ is easy, but calculating the inverse $c^{-e} \bmod n$ is very difficult, well, for large n's anyway. However, if we can factor n into its prime factors p and q, the solution becomes easy again, even for large n's. Obviously, if we can get hold of the secret exponent d, the solution is easy, too.

A slightly less simple example of the RSA algorithm

This time, to make life slightly less easy for those who can crack simple Caesar substitution codes, we will group the characters into blocks of three and compute a message representative integer for each block. *Please note that this method is not secure in any way*. It just shows another example of the mechanism of RSA with small numbers.

For this example, to keep things simple, we'll limit our characters to the letters A to Z and the space character.

```
ATTACK AT SEVEN = ATT ACK _AT _SE VEN
```

In the same way that any decimal number can be represented uniquely as the sum of powers of ten, e.g.
$135 = 1 \times 10^2 + 3 \times 10^1 + 5$,
we can represent our blocks of three characters as the sum of powers of 27 using SPACE=0, A=1, B=2, C=3, .. E=5, .. K=11, .. N=14, .. S=19, T=20, .. V=22, ..., Z=26.

```
ATT ⇒  1 x 27^2 + 20 x 27^1 + 20 = 1289
ACK ⇒  1 x 27^2 +  3 x 27^1 + 11 = 821
_AT ⇒  0 x 27^2 +  1 x 27^1 + 20 = 47
_SE ⇒  0 x 27^2 + 19 x 27^1 +  5 = 518
VEN ⇒ 22 x 27^2 +  5 x 27^1 + 14 = 16187
```

Using this system of integer representation, the maximum value of a block (ZZZ) is $27^3 - 1 = 19682$, so we require a modulus n greater than this value.

1. We choose e = 3
2. We select primes p=173 and q=149 and check
   - gcd(e, p-1) = gcd(3, 172) = 1 ⇒ OK
   - gcd(e, q-1) = gcd(3, 148) = 1 ⇒ OK.
3. Thus we have $n = pq = 173 \times 149 = 25777$, and
   $\phi = (p-1)(q-1) = 172 \times 148 = 25456$.
4. We compute $d = e^{-1} \bmod \phi = 3^{-1} \bmod 25456 = 16971$.
   - Note that $ed = 3 \times 16971 = 50913 = 2 \times 25456 + 1$
   - That is, $ed \equiv 1 \bmod 25456 \equiv 1 \bmod \phi$
5. Hence our public key is (n, e) = (25777, 3) and our private key is (n, d) = (25777, 16971). We keep the values of p, q, d and φ secret.

To encrypt the first integer that represents "ATT", we have
$c = m^e \bmod n = 1289^3 \bmod 25777 = 18524$.

Overall, our plaintext `ATTACK AT SEVEN` is represented by the sequence of five integers m_1, m_2, m_3, m_4, m_5:

```
m_i = (1289, 821, 47, 518, 16187)
```

We compute corresponding ciphertext integers $c_i = m_i^e \bmod n$, (which is still possible by using a calculator, honest):

$$c_1 = 1289^3 \bmod 25777 = 18524$$
$$c_2 = 821^3 \bmod 25777 = 7025$$
$$c_3 = 47^3 \bmod 25777 = 715$$
$$c_4 = 518^3 \bmod 25777 = 2248$$
$$c_5 = 16187^3 \bmod 25777 = 24465$$

We can send this sequence of integers, c_i, to the person who has the private key.

```
c_i = (18524, 7025, 715, 2248, 24465)
```

We can compute the inverse of these ciphertext integers using $m = c^d \bmod n$ to verify that the RSA algorithm still holds. However, this is now outside the realm of hand calculations.

To help you carry out these modular arithmetic calculations, download our free modular arithmetic command line programs.

For example, to compute $18524^{16971} \bmod 25777$, use the `bd_modexp` command:

```
bd_modexp 18524 16971 25777
18524^16971 mod 25777 = 1289
```

You should get the results:

$$
\begin{aligned}
m_1 &= 18524^{16971} \bmod 25777 = 1289 \\
m_2 &= 7025^{16971} \bmod 25777 = 821 \\
m_3 &= 715^{16971} \bmod 25777 = 47 \\
m_4 &= 2248^{16971} \bmod 25777 = 518 \\
m_5 &= 24465^{16971} \bmod 25777 = 16187
\end{aligned}
$$

To convert these integers back to the block of three letters, do the following. For example, given m = 16187,

```
16187 ÷ 27^2 = 16187 ÷ 729 = 22 rem 149, 22 → 'V'
149   ÷ 27^1 = 149   ÷ 27  = 5 rem 14,    5 → 'E'
14    ÷ 27^0 = 14    ÷ 1   = 14 rem 0,   14 → 'N'
```

Hence the integer m = 16187 represents the string "VEN".
Similarly, m = 47 is encoded as follows:

```
47 ÷ 27^2 = 0 rem 47, 0  → SPACE;
47 ÷ 27^1 = 1 rem 20, 1  → 'A';
20 ÷ 27^0 = 20 rem 0, 20 → 'T'
```

giving the string "_AT".

Question: *Why can't we use this method of encoding integers into blocks of three letters to encode the ciphertext?*

A caution about this example

Note that this example is a very insecure method of encryption and should not be used in practice. We can easily factorize the modulus and hence break the cipher.

Factorising a small RSA modulus

Starting with the knowledge that the modulus 25777 is the product of exactly two distinct prime numbers, and that one of these must be less than its integer square root (why?), a little testing of suitable candidates from a table of prime numbers will get you the answer pretty quickly.

Given n = 25777, compute √25777 = 160.55, and then work downwards through the prime numbers < 160, i.e. (157, 151, 149, 139, ...), and try to divide into n in turn:
  157: 25777 / 157 = 164 remainder 29, so not a factor;
  151: 25777 / 151 = 170 remainder 107, so not a factor;
  149: 25777 / 149 = 173 exactly, so we have it.

You could also write a simple computer program to factor n that just divides it by every odd number starting from 3 until it either finds an exact factor or stops when it reaches a number greater than the square root of n.

A real example

In practice, we use a modulus of size in the order of 1024 bits. That is a number over 300 decimal digits long. One example is

```
n =
119294134840169509055527211331255649644606569661527638012067481954943056851150333
806315957037715620297305000118628770846689969112892212245457118060574995989517080
042105263427376322274266393116193517839570773505632231596681121927337473973220
312512599061231322250945506260066557538238517575390621262940383913963
```

This is composed of the two primes

```
p =
109337661836325758176115170347306682871557999846322234541387456711212734562876700
082908433028755212749702453145932229461290645383585810186155398284791464 69
```

```
q =
109106169673491102317237340786149226453370608821417489682098342251389760111799933
94299810159736904468554021708289824396553412180514827996444845438176099727
```

With a number this large, we can encode all the information we need in one big integer. We put our message into an octet string and then convert to a large integer.

Also, rather than trying to represent the plaintext as an integer directly, we generate a random *session key* and use that to encrypt the plaintext with a conventional, much faster symmetrical algorithm like Triple DES or AES-128. We then use the much slower public key encryption algorithm to encrypt just the session key.

The sender A then transmits a message to the recipient B in a format something like this:-

```
Session key encrypted with RSA = xxxx
Plaintext encrypted with session key = xxxxxxxxxxxxxxxxx
```

The recipient B would extract the encrypted session key and use his private key (n,d) to decrypt it. He would then use this session key with a conventional symmetrical decryption algorithm to decrypt the actual message. Typically the transmission would include in plaintext details of the encryption algorithms used, padding and encoding methods, initialisation vectors and other details required by the recipient. The only secret required to be kept, as always, should be the private key.

If Mallory intercepts the transmission, he can either try and crack the conventionally-encrypted plaintext directly, or he can try and decrypt the encryped session key and then use that in turn. Obviously, this system is as strong as its weakest link.

When signing, it is usual to use RSA to sign the message digest of the message rather than the message itself. A one-way hash function like SHA-1 or SHA-256 is used. The sender A then sends the signed message to B in a format like this

```
Hash algorithm = hh
Message content = xxxxxxxxx...xxx
Signature = digest signed with RSA = xxxx
```

The recipient will decrypt the signature to extract the signed message digest, m; independently compute the message digest, m', of the actual message content; and check that m and m' are equal. Putting the message digest algorithm at the beginning of the message enables the recipient to compute the message digest on the fly while reading the message.

CAUTION: We cannot emphasise enough that you *never* use the unadorned versions of RSA described in the simple examples above. You *must* use a proper scheme like PCKS#1 below. In particular when encrypting a message, you must use random padding.

PKCS#1 Schemes

The most common scheme using RSA is PKCS#1 version 1.5 [PKCS1]. This standard describes schemes for both encryption and signing. The encryption scheme PKCS#1v1.5 has some known weaknesses, but these can easily be avoided. See Weaknesses in RSA below.

There is an excellent paper by Burt Kalinski of RSA Laboratories written in the early 1990s [KALI93] that describes in great detail everything you need to know about encoding and signing using RSA. There are full examples right down to listing out the bytes. OK, it uses MD2 and a small 508-bit modulus and obviously doesn't deal with refinements built up over the last decade to deal with more subtle security threats, but it's an excellent introduction.

The conventions we use here are explained below in Notation and Conventions.

Encryption using PKCS#1v1.5

Algorithm: Encryption using PKCS#1v1.5

INPUT: Recipient's RSA public key, (n, e), of length $k = |n|$ bytes; data D (typically a session key) of length $|D|$ bytes with $|D| <= k-11$.
OUTPUT: Encrypted data block of length k bytes

1. Form the k-byte encoded message block, EB,

   `EB = 00 || 02 || PS || 00 || D`

   where $||$ denotes concatenation and PS is a string of $k-|D|-3$ non-zero randomly-generated bytes (i.e. at least eight random bytes).
2. Convert the byte string, EB, to an integer, m, most significant byte first,

   `m = StringToInteger(EB)`

3. Encrypt with the RSA algorithm

   `c = m^e mod n`

4. Convert the resulting ciphertext, c, to a k-byte output block, OB‡

   `OB = IntegerToString(c, k)`

5. Output OB.

The conversions in steps (2) and (4) from byte string to large integer representative and back again may not be immediately obvious. Large integers and byte (bit) strings are conceptually different even though they may both be stored as arrays of bytes in your computer. See What is the difference between a bit string and an integer?

‡2012-05-23: Thanks to "dani torwS" for pointing out a typo in the formula.

Worked Example

Bob's 1024-bit RSA encryption key in hex format:

```
n=
A9E167983F39D55FF2A093415EA6798985C8355D9A915BFB1D01DA197026170F
BDA522D035856D7A986614415CCFB7B7083B09C991B81969376DF9651E7BD9A9
3324A37F3BBBAF460186363432CB07035952FC858B3104B8CC18081448E64F1C
FB5D60C4E05C1F53D37F53D86901F105F87A70D1BE83C65F38CF1C2CAA6AA7EB
e=010001
d=
67CD484C9A0D8F98C21B65FF22839C6DF0A6061DBCEDA7038894F21C6B0F8B35
DE0E827830CBE7BA6A56AD77C6EB517970790AA0F4FE45E0A9B2F419DA8798D6
308474E4FC596CC1C677DCA991D07C30A0A2C5085E217143FC0D073DF0FA6D14
9E4E63F01758791C4B981C3D3DB01BDFFA253BA3C02C9805F61009D887DB0319
```

A randomly-generated one-off session key for AES-128 might be

```
D=4E636AF98E40F3ADCFCCB698F4E80B9F
```

The encoded message block, EB, after encoding but before encryption, with random padding bytes shown in green,

```
0002257F48FD1F1793B7E5E02306F2D3228F5C95ADF5F31566729F132AA12009
E3FC9B2B475CD6944EF191E3F59545E671E474B555799FE3756099F044964038
B16B2148E9A2F9C6F44BB5C52E3C6C8061CF694145FAFDB24402AD1819EACEDF
4A36C6E4D2CD8FC1D62E5A1268F496004E636AF98E40F3ADCFCCB698F4E80B9F
```

After RSA encryption, the output is

```
3D2AB25B1EB667A40F504CC4D778EC399A899C8790EDECEF062CD739492C9CE5
8B92B9ECF32AF4AAC7A61EAEC346449891F49A722378E008EFF0B0A8DBC6E621
EDC90CEC64CF34C640F5B36C48EE9322808AF8F4A0212B28715C76F3CB99AC7E
609787ADCE055839829E0142C44B676D218111FFE69F9D41424E177CBA3A435B
```

The above hex data in C format.

Note that the output for encryption will be different each time (or should be!) because of the random padding used.

Encrypting a message

For a plaintext message, say,

```
PT="Hello world!"
```

that is, the 12 bytes in hex format,

```
PT=48656C6C6F20776F726C6421
```

Then, using the 128-bit session key from above,

```
KY=4E636AF98E40F3ADCFCCB698F4E80B9F
```

and the uniquely-generated 128-bit initialization vector (IV)

```
IV=5732164B3ABB6C4969ABA381C1CA75BA
```

the ciphertext using AES-128 in CBC mode with PKCS#5 padding is,

```
CT=67290EF00818827C777929A56BC3305B
```

The sender would then send a transmission to the recipient (in this case, Bob) including the following information in some agreed format

```
Recipient: Bob
Key Encryption Algorithm: rsaEncryption
Encrypted Key:
3D2AB25B1EB667A40F504CC4D778EC399A899C8790EDECEF062CD739492C9CE5
8B92B9ECF32AF4AAC7A61EAEC346449891F49A722378E008EFF0B0A8DBC6E621
EDC90CEC64CF34C640F5B36C48EE9322808AF8F4A0212B28715C76F3CB99AC7E
609787ADCE055839829E0142C44B676D218111FFE69F9D41424E177CBA3A435B
Content Encryption Algorithm: aes128-cbc
IV: 5732164B3ABB6C4969ABA381C1CA75BA
Encrypted Content:
67290EF00818827C777929A56BC3305B
```

The usual formats used for such a message are either a *CMS enveloped-data object* or XML, but the above summary includes all the necessary info (well, perhaps "Bob" might be defined a bit more accurately).

Cryptographic Message Syntax (CMS) [CMS] is a less-ambiguous version of the earlier PKCS#7 standard (also of the same name) and is designed to be used in S/MIME messages. CMS enveloped-data objects (yes, that's *enveloped* not *encrypted*) use ASN.1 and are encoded using either DER- or BER-encoding. (DER-encoding is a stricter subset of BER).

The terminology for CMS and ASN.1 may sound messy, but the end results are well-defined and universally-accepted. On the other hand, the XML cryptographic standards are, to be honest, a complete mess: see XML is xhite. Pretty Good Privacy (PGP) also has a format for RSA messages, although PGP stopped using RSA because of patent issues back in the 1990s.

Nothing, of course, stops you and your recipient from agreeing on your own format and using that. But be careful, even the experts get these things wrong and accidentally give away more than they realise.

Signing using PKCS#1v1.5

Algorithm: Signing using PKCS#1v1.5

INPUT: Sender's RSA private key, (n, d) of length `k = |n|` bytes; message, M, to be signed; message digest algorithm, Hash.
OUTPUT: Signed data block of length k bytes

1. Compute the message digest H of the message,

   ```
   H = Hash(M)
   ```

2. Form the byte string, T, from the message digest, H, according to the message digest algorithm, Hash, as follows

Hash	**T**
MD5	`30 20 30 0c 06 08 2a 86 48 86 f7 0d 02 05 05 00 04 10 || H`
SHA-1	`30 21 30 09 06 05 2b 0e 03 02 1a 05 00 04 14 || H`
SHA-224	`30 2d 30 0d 06 09 60 86 48 01 65 03 04 02 04 05 00 04 1c || H`
SHA-256	`30 31 30 0d 06 09 60 86 48 01 65 03 04 02 01 05 00 04 20 || H`
SHA-384	`30 41 30 0d 06 09 60 86 48 01 65 03 04 02 02 05 00 04 30 || H`
SHA-512	`30 51 30 0d 06 09 60 86 48 01 65 03 04 02 03 05 00 04 40 || H`

   where T is an ASN.1 value of type *DigestInfo* encoded using the Distinguished Encoding Rules (DER).
3. Form the k-byte encoded message block, EB,

   ```
   EB = 00 || 01 || PS || 00 || T
   ```

   where `||` denotes concatenation and PS is a string of bytes all of value 0xFF of such length so that `|EB|=k`.
4. Convert the byte string, EB, to an integer m, most significant byte first,

   ```
   m = StringToInteger(EB)
   ```

5. Sign with the RSA algorithm

   ```
   s = m^d mod n
   ```

6. Convert the resulting signature value, s, to a k-byte output block, OB

   ```
   OB = IntegerToString(s, k)
   ```

7. Output OB.

Worked Example

Alice's 1024-bit RSA signing key in hex format:

```
n=
E08973398DD8F5F5E88776397F4EB005BB5383DE0FB7ABDC7DC775290D052E6D
12DFA68626D4D26FAA5829FC97ECFA82510F3080BEB1509E4644F12CBBD832CF
C6686F07D9B060ACBEEE34096A13F5F7050593DF5EBA3556D961FF197FC981E6
F86CEA874070EFAC6D2C749F2DFA553AB9997702A648528C4EF357385774575F
e=010001
d=
00A403C327477634346CA686B57949014B2E8AD2C862B2C7D748096A8B91F736
F275D6E8CD15906027314735644D95CD6763CEB49F56AC2F376E1CEE0EBF282D
F439906F34D86E085BD5656AD841F313D72D395EFE33CBFF29E4030B3D05A28F
B7F18EA27637B07957D32F2BDE8706227D04665EC91BAF8B1AC3EC9144AB7F21
```

The message to be signed is, of course,

```
M="abc"
```

that is, the 3 bytes in hex format,

```
M=616263
```

The message digest algorithm is SHA-1, so

```
H = Hash("abc") = A9993E364706816ABA3E25717850C26C9CD0D89D
```

The DigestInfo value for SHA-1 is

```
T=
3021300906052B0E03021A05000414A9993E364706816ABA3E25717850C26C9C
D0D89D
```

The encoded message block, EB, after encoding but before signing is

```
0001FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00302130
0906052B0E03021A05000414A9993E364706816ABA3E25717850C26C9CD0D89D
```

After RSA signing, the output is

```
60AD5A78FB4A4030EC542C8974CD15F55384E836554CEDD9A322D5F4135C6267
A9D20970C54E6651070B0144D43844C899320DD8FA7819F7EBC6A7715287332E
C8675C136183B3F8A1F81EF969418267130A756FDBB2C71D9A667446E34E0EAD
9CF31BFB66F816F319D0B7E430A5F2891553986E003720261C7E9022C0D9F11F
```

The above hex data in C format.

Weaknesses in RSA

Small encryption exponent
: If you use a small exponent like e=3 ***and*** send the same message to different recipients ***and*** just use the RSA algorithm without adding random padding to the message, then an eavesdropper could recover the plaintext.

For an example of this, see Cracking RSA on our page on the The Chinese Remainder Theorem.

Small encryption exponent and small message
: If you use $e = 3$ and just encrypt a small message m without padding where $m^3 < n$ then your ciphertext c can easily be broken by simply computing its real cube root. For example, if we have the public key $(n, e) = (25777, 3)$ and just encrypt the small message $m = 10$ then the ciphertext is $c = 1000$. The secure properties of RSA encryption only work if $m^e > n$.

Using the same key for encryption and signing
: Given that the underlying mathematics is the same for encryption and signing, only in reverse, if an attacker can convince a key holder to sign an unformatted encrypted message using the same key then she gets the original.

Using a common modulus for different users
: Do not use the same modulus n with different (e_i, d_i) pairs for different users in a group. Given his own pair (e_1, d_1), user 1 can factorize the common n into p and q and hence compute the private exponents d_i of all the other users.

For more details, see our page RSA: how to factorize N given d.

Acting as an oracle
: There are techniques to recover the plaintext if a user just blindly returns the RSA transformation of the input. So don't do that.

Solutions

1. Don't use the same RSA key for encryption and signing.
2. Don't encrypt or sign a blind message.
3. If using PKCS#v1.5 encoding, use e=0x10001 for your public exponent.
4. Always format your input before encrypting or signing.
5. Always add *fresh* random padding - at least 8 bytes - to your message before encrypting.
6. When decrypting, check the format of the decrypted block. If it is not as expected, return an error message, not the decrypted string.
7. Similarly, when verifying a signature, if there is any error whatsoever, just respond with "Invalid Signature".

More Advanced Schemes

The underlying RSA operations

$$c = m^e \bmod n,\ m' = c^d \bmod n; \quad s = m^d \bmod n,\ s' = s^e \bmod n$$

are always the same, but there are many variants of how these can be used inside an encryption or digital signature *scheme*. Here are some of them.

RSAES-OAEP

The OAEP encoding technique for encryption is described in PKCS#1 version 2 and in IEEE P136. It is more secure than the PKCS#1v1.5 encoding method described above, perhaps provably secure. The encoding technique involves a mask generation function (MGF) based on a hash function and there is no obvious structure in the encoded block, unlike the PKCS#1v1.5 encoding method. Despite being the recommended method for the last decade, OAEP is not used in practice as much as you'd expect. In fact, hardly at all. That said, if you have a choice in the matter, we recommend that you *should* use OAEP if you can.

RSASSA-PSS

The PSS encoding method is used to encode before creating a signature. The technique is described in PKCS#1v2.1 and is similar in design to the OAEP encoding used for encryption involving an MGF based on a hash function. However, there were active patents associated with this method until recently and so it is still not supported well. There are currently no known weaknesses with the PKCS#1v1.5 signature scheme.

X9.31 Signature Scheme

ANSI standard X9.31 [AX931] requires using *strong primes* derived in a way to avoid particular attacks that are probably no longer relevant. X9.31 uses a method of encoding the message digest specific to the hash algorithm. It expects a key with length an exact multiple of 256 bits. The same algorithm is also specified in P1363 [P1363] where it is called IFSP-RSA2. The scheme allows for the public exponent to be an even value, but we do not consider that case here; all our values of e are assumed to be odd. The message digest hash, H, is encapsulated to form a byte string as follows

```
EB = 06 || PS || 0xBA || H || 0x33 || 0xCC
```

where PS is a string of bytes all of value 0xBB of length such that $|\text{EB}|=|\text{n}|$, and 0x33 is the ISO/IEC 10118 part number† for SHA-1. The byte string, EB, is converted to an integer value, the message representative, f.

† ISO/IEC 10118 part numbers for other hash functions are: SHA-1=0x33, SHA-256=0x34, SHA-384=0x36, SHA-512=0x35, RIPEMD=0x31.

Algorithm: Forming an X9.31/RSA2 signature value from the message representative (for odd e).

INPUT: Signer's RSA private key, (n, d); integer, f, where $0 \le f < n$ and $f \equiv 12 \pmod{16}$.
OUTPUT: Signature, an integer s where $0 \le s < n/2$, i.e. a value at least one bit shorter than n.

1. $t = f^d \bmod n$
2. $s = \min(t, n-t)$
3. Output s.

The integer, s, is converted to a byte string of length $|\text{n}|$ bytes.

Algorithm: Extracting the message representative from an X9.31/RSA2 signature value (for odd e).

INPUT: Signer's RSA public key, (n, e); signature, s.
OUTPUT: Message representative, f, such that $f \equiv 12 \pmod{16}$, or "invalid signature".

1. If s is not in $[0, (n-1)/2]$, output "invalid signature" and stop.
2. Compute $t = s^e \bmod n$
3. If $t \equiv 12 \pmod{16}$ then let $f = t$.
4. Else let $f = n - t$. If $f \not\equiv 12 \pmod{16}$, output "invalid signature" and stop.
5. Output f.

The integer f is converted to a byte string of length $|\text{n}|$ bytes and then parsed to confirm that *all* bytes match the required format

```
EB = 06 || PS || 0xBA || H || 0x33 || 0xCC
```

If not, output "invalid signature" and stop; otherwise output the extracted message digest hash, H.

ISO/IEC 9796

IOS/IEC 9796 is an old standard devised before there was a recognised message digest function like MD5 or SHA-1. It allows the entire message to be recovered. Unfortunately, it is considered broken for signing plain text messages, but is still OK for signing message digest values. It is used in the AUTACK scheme described in [EDIFACT].

The encapsulation mechanism weaves the input bytes into a format exactly one bit shorter than the RSA key. The signing mechanism is similar to that in ANSI X9.31 described above, but the message representative, f, is required to be $f \equiv 6 \pmod{16}$, instead of modulo 12. In other words, make sure the last 4 bits are equal to 0x6 instead of 0xC.

RSA-KEM

The RSA-KEM Key Transport Algorithm encrypts a *random* integer with the recipient's public key, and then uses a symmetric key-wrapping scheme to encrypt the keying data. KEM stands for *Key Encapsulation Mechanism*. The general algorithm is as follows

1. Generate a random integer z between 0 and $n-1$.
2. Encrypt the integer z with the recipient's RSA public key: $c = z^e \bmod n$.
3. Derive a key-encrypting key KEK from the integer z.
4. Wrap the keying data using KEK to obtain wrapped keying data WK.
5. Output c and WK as the encrypted keying data.

This method has a higher security assurance than PKCS#1v1.5 because the input to the underlying RSA operation is random and independent of the message, and the key-encrypting key KEK is derived from it in a strong way. The downside is that you need to implement a key derivation method (of which there are many varieties) and a key wrapping algorithm. The encoding of the final data into the recommended ASN.1 format is messy, too. For more details, see the latest version of [CMSRSAKEM].

Ferguson-Schneier Encryption

In their book [FERG03], Niels Ferguson and Bruce Schneier suggest a much simpler method of encryption. They suggest using the same modulus n for both encryption and signatures but to use $e = 3$ for signatures and $e = 5$ for encryption. You need to make sure that the modulus $n = pq$ satisfies both $\gcd(3, (p-1)(q-1)) = 1$ and $\gcd(5, (p-1)(q-1)) = 1$.

Their method uses RSA to encrypt a random integer and use a hash function to derive the actual content encryption key, thus removing any structural similarities between the actual CEK and the data encrypted by the RSA. They recommend using the function, `Hash(x):=SHA256(SHA256(x))`, for hashing data.

Algorithm: Ferguson-Schneier Encrypt Random Key with RSA.

INPUT: Recipient's RSA public key, (n, e).
OUTPUT: Content encryption key, CEK; RSA-encrypted CEK, c.

1. Compute the exact bit length of the RSA key, $k = \lceil log_2(n+1) \rceil$.
2. Choose a random r in the interval $[0, 2^k - 1]$.
3. Compute the content encryption key by hashing r, CEK=Hash(r).
4. $c = r^e \bmod n$.
5. Output CEK and c.

For a plaintext message, m, the transmission sent to the recipient is `IntegerToString(c)||E_CEK(m)`, where $E_{CEK}(m)$ is the result of encrypting m with a symmetrical encryption algorithm using key, CEK. Given that the recipient knows the size of the RSA key and hence the exact number of bytes needed to encode c, it is a simple matter to parse the input received from the sender.

For example code of this algorithm in Visual Basic (both VB6 and VB.NET) using our CryptoSys PKI Toolkit, see Ferguson-Schneier RSA Encryption.

Notation and Conventions

We use the following notation and conventions in this page.

- A || B denotes concatenation of byte (or bit) strings A and B.
- |B| denotes the length of the byte (or bit) string B in bytes.
- |n| denotes the length** of the non-negative integer n in bytes, $|n| = \lceil log_2(n+1) \rceil$.
- `IntegerToString(i, n)` is an n-byte encoding of the integer i with the most significant byte first (i.e. in "big-endian" order). So, for example,

```
IntegerToString(1,4)="00000001"
IntegerToString(7658,3)="001DEA"
```

- `StringToInteger(S)` is the integer represented by the byte string S with the most significant byte first.
- $\lceil x \rceil$ is the smallest integer, n, such that $n \geq x$.

** Strictly speaking, this is the length of the shortest byte string that can encode the integer.

What is the difference between a bit string and an integer?

A **string** is a contiguous sequence of symbols, so the string "cat" is a sequence of the letters 'c', 'a' and 't'. A **bit string** is sequence of binary digits (bits) '0' and '1'. A **byte string** is similar except it consists of bytes, which are in turn sequences of 8 bits. So a **bit string** and a **byte string** are the same thing, except the latter is restricted to multiples of 8 bits. For example, using hexadecimal representation, the byte string "8002EA" is a sequence of 3 bytes, 0x80, 0x02 and 0xEA; and is equal to the bit string "100000000000001011101010".

> **A note on notation:** To differentiate between byte strings and integers here, we show byte strings in hexadecimal representation inside quotes, so "8002EA" denotes the string of three consecutive bytes with hexadecimal values 80, 02 and EA, respectively. We use the "0x" prefix to denote integers, so 0x80 denotes the integer with hexadecimal value 80 (decimal 128), and 0x8002EA denotes the integer with hexadecimal value 8002EA (decimal 8389354). Everything here is in "big-endian" order, with the left-most bits the most significant.

A string can be split into sub-strings (e.g. "8002" and "EA") and two strings can be concatenated (joined up) to make another string. The order of symbols is important. The usual convention is to write byte strings with the most significant byte first ("big-endian" or network byte order).

An **integer** is a whole number that obeys the usual rules of *integer* arithmetic ($1+1=2,\ 5-2=3,\ 3\times 2=6,\ 6/3=2$) and modular arithmetic ($10+6 \equiv 4 \pmod{12}$)). There is no limit in theory as to how large an integer can be: you can always add one to any integer. The integer 8,389,354 in decimal is the same as the number 0x8002EA in hexadecimal notation, but is not the same as the byte string "8002EA", even though it looks the same and may well be stored in your computer in the same form.

You can increment the integer 8389354 to get 8389354+1=8389355 (0x8002EB); but you cannot "increment" the byte string "8002EA". On the other hand, you can concatenate the byte strings "8002" and "EA" to make "8002EA"; but the integers 8389 and 354 do not add to make 8389354. The byte string "00008002EA" is strictly not the same as "8002EA" (the former has two extra bytes of value 0x00 at the start); but the integers 0x008002EA and 0x8002EA are equal (leading zeros do not count).

With RSA encryption, we typically want to encrypt a session key which is a *bit string*, but the RSA operation $c = m^e \bmod n$ is done with *integers*, so we need to *represent* the bit string as an integer first (in practice, we usually add some random bytes and other padding, but we'll ignore that for the time being). Once the RSA operation has been completed, we have another integer, c, but we need to store the result as a *bit string*, so we *encode* the integer as a bit (byte) string and pass that string onto our recipient.

Example: Suppose we wish to encrypt the 3-byte/24-bit key bit string "8002EA" using the RSA public key (n=25009997=0x017D9F4D, e=5)†. For simplicity in this insecure example, we will use the basic RSA algorithm with no padding.

1. The message block is the byte string "8002EA".
2. Compute the message representative

```
m = StringToInteger("8002EA") = 8389354
```

3. Encrypt with the RSA algorithm

```
c = 8389354^5 mod 25009997 = 2242555
```

4. Encode the result as a byte string

```
OB = IntegerToString(2242555, 4) = 002237FB
```

Note that the maximum length of the output block is 4 bytes, because the largest possible integer result is 0x017D9F4C ($= n - 1$), which requires 4 bytes to store in encoded form.

† Thanks to "doctorjay" for pointing out that e=3 did not work for the earlier version of this example.

Implementation in C and VB

We show an example implementation of the RSA algorithm in C in our BigDigits library. It's not necessarily the most efficient way, and could be improved in its security, but it shows the maths involved. Look in the BigDigits Test Functions.

There is an example in VB6/VBA code at RSA and Diffie-Hellman in Visual Basic.

For a professional implementation, see our commercial CryptoSys PKI Toolkit which can be used with Visual Basic, VB6, VBA, VB2005+, C/C++ and C# applications. There are examples using the "raw" RSA functions to carry out RSA Encryption and RSA Signing.

References

- **[AX931]** ANSI X9.31-1998 *Digital Signatures using Reversible Public Key Cryptography for the Financial Services Industry (rDSA)*, Appendix A, American National Standards Institute, 1998.
- **[CMS]** RFC 5652. *Cryptographic Message Syntax (CMS)*, R. Housley, September 2009 (obsoletes RFC3852, RFC3369, RFC2630).
- **[CMSRSAKEM]** RFC 5990 *Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS)*, J. Randall, B.Kaliski, J. Brainard, S. Turner. September 2010.
- **[COCK73]** Clifford Cocks. *A Note on 'Non-Secret Encryption'*, CESG Research Report, 20 November 1973.
- **[EDIFACT]** UN/EDIFACT Finance Group D6 SWG-F. *Recommended Practice For Message Flow And Security For EDIFACT Payments*, Version 2v03, 1 October 2000.
- **[FERG03]** Niels Ferguson and Bruce Schneier, *Practical Cryptography*, Wiley, 2003. Note that this book has since been re-issued in 2010 almost unchanged as *Cryptography Engineering* by Niels Ferguson, Bruce Schneier and Tadayoshi Kohno.
- **[KALI93]** Burton Kalinski. *Some Examples of the PKCS Standards*, RSA Laboratories, 1999, <link>.
- **[NIST-80057]** NIST Special Publication 800-57, *Recommendation for Key Management - Part 1: General (Rev. 4)*, Elaine Barker et al, National Institute of Standards and Technology, January 2016, <link>.
- **[P1363]** IEEE P1363 *Standard Specifications for Public Key Cryptography*, IEEE, November 1993.
- **[PKCS1]** RSA Laboratories. *PKCS #1 Cryptography Standard Version 2.2*, October 2012. Republished as RFC 8017.
- **[RIVE78]** R. Rivest, A. Shamir and L. Adleman. *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems.* Communications of the ACM, 21 (2), pp. 120-126, February 1978, <link>.

Author

Feedback

Contact us to give feedback or ask a question. To comment on this page, send us a message with `rsa_alg` in the message and we'll add it to the comments. Please note we are not an oracle for college assignment questions. Your tutors read this page, too!

This page last updated 21 December 2019

Comments

[Go to last comment]

123 comments so far

I learned this site from http://www.issociate.de/board/post/267506/Encryption_size.html. This is an excellent and indepth artical.

mimime | *usa - Fri, Feb 19 2010 00:30 GMT*

it is one of the most great tutorial about RSA , it was very helpful

Thank you very much

mallek balli | *libya - Wed, Mar 10 2010 16:39 GMT*

I PREPARED MY WHOLE PROJECT ABOUT RSA COMPLETLY,,,,,,,,, THANQ VERY MUCH

MANDA GOPAL | *INDIA - Sun, Mar 14 2010 04:52 GMT*

this explanation is better than the best

rishi | *- Sat, Mar 20 2010 18:23 GMT*

this is awsome article thank u very much

yogesh kumbhar | *- Tue, Mar 23 2010 06:22 GMT*

tHANKS A LOt....

vijac | *iNDIA - Fri, Mar 26 2010 00:48 GMT*

thankyou so much for such a great content....... now i can understand rsa in a better way and can prepare my project.......

shivani anand | *- Sat, Mar 27 2010 14:14 GMT*

really useful tutorial on rsa encryption - cheers! im a bit confused about calculating d from ed=1%phi though. in your example phi=20 and e=3 so i would have thought 1%20 = 1, therefore 3d=1, therefore d = 1/3. i know im wrong since