CRYPTOGRAPHY TOPIC VII
RSA CRYPTOGRAPHY

PAUL L. BAILEY

1. Modular Exponentiation

1.1. **Number Theory.** The motivating mathematical proposition for RSA cryptography is the fact that, in the ring $\mathbb{Z}_n$, multiples are computed modulo n, but exponents may often be computed modulo $\phi(n)$. The precise result from number theory may be stated as follows.

Proposition 1. *Let $a, e, n \in \mathbb{Z}$, with $n \geq 2$, $\gcd(a, n) = 1$, and $e \equiv 1 \pmod{\phi(n)}$. Then*
$$a^e \equiv 1 \pmod{n}.$$

Proof. Since $e \equiv 1 \pmod{\phi(n)}$, we have $e - 1 = \phi(n)k$ for some $k \in \mathbb{Z}$. Then
$$a^e \equiv a^{\phi(n)k+1} \equiv (a^{\phi(n)})^k a \pmod{n}.$$
By Euler's Theorem, $a^{\phi(n)} \equiv 1 \pmod{n}$, so
$$a^e \equiv a \pmod{n}.$$
□

In the case where n is the product of distinct primes, the premise that a and n are relatively prime may be removed, as we now show. Our first lemma may be obvious, but we nevertheless supply a proof.

Lemma 1. *Let $p, q \in \mathbb{Z}$ be distinct positive primes, and let $x \in \mathbb{Z}$. If $p \mid x$ and $q \mid x$, then $pq \mid x$.*

Proof. Suppose $p \mid x$ and $q \mid x$. Then $x = pi = qj$ for some $i, j \in \mathbb{Z}$. Thus $q \mid pi$, and since q is prime, $q \mid p$ or $q \mid i$.

If $q \mid p$, then $p = qk$ for some $k \in \mathbb{Z}$, so either $q = 1$ or $q = p$. But neither of these is the case, since q is a positive prime distinct from p. Thus q does not divide p, whence $q \mid i$, so $i = qm$ for some $m \in \mathbb{Z}$.

From this, $x = pi = pqm$, so $pq \mid x$. □

Date: November 7, 2007.

The second lemma consolidates the argument for the proposition which follows.

Lemma 2. *Let $a, e, n, p \in \mathbb{Z}$, where p is a positive prime, $(p-1) \mid \phi(n)$, and $e \equiv 1 \pmod{\phi(n)}$. Then*

$$a^e \equiv 1 \pmod{p}.$$

Proof. Since $p-1$ divides $\phi(n)$, then there exists $i \in \mathbb{Z}$ such that $\phi(n) = (p-1)i$. Let $m = e - 1$; since $e \equiv 1 \pmod{\phi(n)}$, $e-1$ is divisible by $\phi(n)$, so there exists $j \in \mathbb{Z}$ such that $m = \phi(n)j = (p-1)ij$.

Claim 1: If $\gcd(a,p) = 1$, then $a^m \equiv 1 \pmod{p}$.
We have

$$a^m \equiv a^{k(p-1)(q-1)} \equiv (a^{p-1})^{k(q-1)} \equiv 1^{k(q-1)} \equiv 1 \pmod{p}.$$

Claim 2: If $\gcd(a,p) = 1$, then $a^e \equiv 1 \pmod{p}$.
We have

$$a^e \equiv a^{m+1} \equiv a^m a \equiv 1 \cdot a \equiv a \pmod{p}.$$

Claim 3: If $p \mid a$, then $a^e \equiv a \pmod{p}$.
We have

$$a^e \equiv 0^e \equiv 0 \equiv a \pmod{p}.$$

Finally, we note that, since p is prime, either $p \mid a$ or $\gcd(a,p) = 1$. So, in either case, $a^e \equiv a \pmod{p}$. □

Proposition 2. *Let $a, e, n, p, q \in \mathbb{Z}$, where p and q are distinct positive primes, $n = pq$, and $e \equiv 1 \pmod{n}$. Then $a^e \equiv a \pmod{n}$.*

Proof. We have seen that $\phi(n) = (p-1)(q-1)$. By the previous lemma, $a^e \equiv 1 \pmod{p}$ and $a^e \equiv 1 \pmod{q}$. Thus $a^e - a$ is divisible by p and by q, and since p and q are distinct primes, $a^e - a$ is divisible by $pq = n$. Thus $a^e \equiv a \pmod{n}$. □

1.2. **Software Implementation.** The pertinent number theoretical functions we need are

- **(a)** the euclidean algorithm, which helps find modular inverses;
- **(b)** finding modular inverses (to find d given e and $n = pq$);
- **(c)** efficient modular exponentiation.

Item **(c)** requires a new idea: write the exponent in its binary expansion, repeated squaring the base, and multiply the result into an accumulator if the appropriate bit is set in the exponent. To compute b^w, the algorithm is

1. set $a = 1$ and $c = b$
2. if the i^{th} bit of w is 1, set $a = ac$
3. set $c = c^2$
4. go to 2

For example, we raise 3 to the power 10. The binary expansion of 10 is

$$10 = 0(2^0) + 1(2^1) + 0(2^2) + 1(2^3).$$

So

$$3^10 = 3^{0(2^0)+1(2^1)+0(2^2)+1(2^3)} = 3^{0(2^0)} \cdot 3^{1(2^1)} \cdot 3^{0(2^2)} \cdot 3^{1(2^3)} = 3^2 \cdot 3^8;$$

In this ways, only powers of 3 with exponent a power of two need be computed.

This idea produces a power function. To make it a modular power function, we reduce modulo n after each relevant computation.

Here is code for these three functions. Notice that the power function uses 64-bit temporary variables; if if n is 32-bits, a number less than n squared may require 64-bit prior to reduction modulo n.

```
// Number Theory Functions for RSA

typedef unsigned __int32 UNT;
typedef unsigned __int64 UNX;

UNT euclid(UNT m,UNT n,int& x,int& y)
{ UNT d,q,r;
  int t;
  x=1; y=0;
  q=n/m; r=n%m;
  if (r==0) return m;
  d=euclid(r,m,x,y);
  t=x; x=y-q*x; y=t;
  return d; }

UNT modinv(UNT a,UNT n)
{ UNT b=0;
  int x,y;
  if (euclid(a,n,x,y)>1) return 0;
  x%=int(n);
  if (x<0) x+=n;
  return x; }

UNT modpow(UNT a,UNT w,UNT n)
{ UNX u=a%n,v=1;
  while (w)
  { if (w&1)
    { v*=u; v%=n; }
    u*=u; u%=n; w>>=1; } }
  return UNT(v); }
```

2. RSA (Unblocked)

2.1. **Description.** We now describe the RSA encryption technique.

- Select two distinct positive prime integers p and q.
- Let $n = pq$; this is the *RSA modulus*.
- Select $e \in \mathbb{Z}$ such that $\gcd(e, \phi(n)) = 1$; this is the *RSA exponent*.
- Use the Euclidean algorithm to compute $d \in \mathbb{Z}$ such that $ed \equiv 1 \pmod{\phi(n)}$.
- The public key is (n, e).
- The private key is d.
- The text space consists of integers x such that $0 \le x < n$.
- The encryption mapping is $x \mapsto x^e \pmod{n}$.
- The decryption mapping is $y \mapsto y^d \pmod{n}$.
- Since $ed \equiv 1 \pmod{\phi(n)}$, we have $(a^e)^d \equiv a^{ed} \equiv a \pmod{n}$.

We view $x \in \mathbb{Z}_n$ and $e, d \in \mathbb{Z}_n^*$.

The difficulty in describing RSA as a cryptosystem (A, K, E) arises from the fact that the plaintext space A depends on the chosen RSA modulus. Thus, we must view RSA as a *family* of cryptosystems.

Let n be the product of distinct primes; the RSA cryptosystem (A, K, E) given by n consists of

- $A = \mathbb{Z}_n$;
- $K = \mathbb{Z}_n^*$, where each key $k = e \in K$ is the RSA exponent;
- $E : K \to \mathrm{Sym}(A)$ is given by $E_k : x \mapsto x^k$.

Example 1. Let $p = 11$ and $q = 13$ so that $pq = 143$ and $h = \phi(n) = (p-1)(q-1) = 120$. Let $e = 7$. Then $d = 17$, since $ed \equiv 1 \pmod{h}$.

Let $x = 25$ be a message. Working modulo 143, we have $y = x^e = 64$, and $z = y^d = 25 = x$.

2.2. **Software.** The following function test the unblocked RSA encryption scheme.

```
// Test RSA number theory functions

void test(void)
{ UNT n,p,q,h,e,d,x,y,z;

  p=229;
  q=139;
  e=257;
  n=p*q;
  h=(p-1)*(q-1);
  d=modinv(e,h);

  x=449;
  y=modpow(x,e,n);
  z=modpow(y,d,n); // should equal x
  printf("p: %u  q: %u  n: %u  h: %u  e: %u  d: %u  x: %u  y: %u  z: %u\n",
    p,q,n,h,e,d,x,y,z); }
```

3. Packing Integers

3.1. **Description.** We describe an algorithm for packing integers.

Let $A = \{0, \ldots, N-1\}$ be a set of cardinality N. We would like to pack as many of these as possible into a number which is less than a given number n. That is, we would like to assign a number less than n to specific ordered tuples from A.

If we choose k members from A at random, the number of possible choices is N^k. So, the maximum number of arbitrary members of A we can pack into a number less than n is the largest integer k such that $N^k < n$. This number is $k = \lfloor \log_N n \rfloor$.

Recall the identity

$$\frac{1 - x^k}{1 - x} = 1 + x + x^2 + \cdots + x^{k-1},$$

which is used in the derivation of the geometric series.

Construct a function

$$\nu : A^k \to \{0, \dots, n\}$$

by, for $\vec{a} = (a_0, \dots, a_{k-1})$,

$$\nu(\vec{a}) = \sum_{i=0}^{k-1} a_i N^i.$$

This function is injective, and the largest possible value obtained is given by the identity

$$(N-1)(1 + N + N^2 + \cdots + N^{k-1}) = N^k - 1.$$

We see that $\vec{a}$ is essentially the base N expansion of $\nu(\vec{a})$, and $\nu(\vec{a}) \leq N^k - 1 < n$.

Example 2. Let $A = \{0, 1, 2\}$ and $n = 91$. Then $N = |A| = 3$, and the smallest power of N less than n is $3^4 = 81 < 91$. So, let $k = 4 = \lfloor \log_3 91 \rfloor$.

Let $\vec{a} = (1, 0, 2, 2)$. Then

$$\nu(\vec{a}) = 1 + 0(3) + 2(3)^2 + 2(3^3) = 1 + 18 + 54 = 73.$$

3.2. **Software Implementation.** Our packing algorithm requires three functions:

- an implementation of the greatest integer logarithm
- a function to pack numbers
- a function to unpack numbers

Since we do not know a priori how many numbers we will have to pack, we use an array of indeterminate size to store them. We assume in the code that n can be stored in 32 bits. It should be noted that modular exponentiation is relatively faster if the exponent has fewer bits on in its binary expansion.

```
// Packing functions

UNT paksze(UNT num,UNT bas)             // integer log
{ UNT sze=0,pow=1;
  while ((pow*=bas)<num) sze++;
  return sze; }

UNT pakarr(BYT* arr,UNT sze,UNT bas)    // pack
{ UNT pow=1,ctr=0;
  UNT pak=0;
  while (ctr<sze)
  { pak+=pow*arr[ctr++];
    pow*=bas; }
  return pak; }

void pakbak(BYT* arr,UNT sze,UNT bas,UNT pak) // unpack
{ UNT ctr=0;
  memset(arr,0,sze);
  while (ctr<sze)
  { arr[ctr++]=pak%bas;
    pak/=bas; } }
```

4. RSA (Blocked)

4.1. **Description.** RSA can be used to encrypt positive integers less than the RSA modulus n. If we are working with a stream over an alphabet A, we implement RSA as a block cipher on this stream as follows.

If $|A| = N$, we translate A so that $A = \{0, \ldots, N-1\}$. Compute $k = \lfloor \log_N n \rfloor$, and collect k entries off the stream to obtain $\vec{a} = (a_0, \ldots, a_{k-1})$. Now encrypt $\nu(\vec{a})$, and continue down the stream.

Example 3. Consider the alphabet of capital letters, converted to numbers so that

$$A = \{0, 1, \ldots, 25\}.$$

Then $N = |A| = 26$. We wish to encrypt the message `cipher`. Converted into numbers, this message is $(2, 8, 15, 7, 17)$.

Let $p = 229$ and $q = 139$ so that $n = pq = 31831$ is the RSA modulus. Let $h = \phi(n) = (p-1)(q-1) = 31464$. Let $e = 257$ be the RSA encryption exponent, which is prime to 31464 and is a power of two plus one, and so is relatively faster for modular exponentiation. The euclidean algorithm gives the RSA decryption exponent to be $d = 857$, so that $ed \equiv 1 \pmod{31464}$.

The first few powers of 26 are $N^0 = 1$, $N^1 = 26$, $N^2 = 676$, $N^3 = 17576$, $N^4 = 456976$, so $k = \lfloor \log_{26} 31831 \rfloor = 3$. Thus we can pack three letters into a block.

We collect blocks of three letters off the stream, apply ν, and encrypt. The resulting ciphertext is a sequence of numbers less than $n = 31464$.

The first block is $(2, 8, 15)$; we have

$$m = \nu(2, 8, 15) = 2(26^0) + 8(26^1) + 15(26^2) = 2 + 208 + 10140 = 10350.$$

The ciphertext is

$$m^e = 10350^{257} \pmod{31831} = 9507.$$

The second block is padded to $(7, 17, 0)$; we have

$$m = \nu(7, 17, 0) = 7(26^0) + 17(26^1) + 0(26^2) = 7 + 442 = 449.$$

The ciphertext is

$$m^e = 449^{257} \pmod{31831} = 26561.$$

5. Software Implementation

We implement RSA on plain text files. The alphabet is $A = \{0, 1, \ldots, 25\}$. Set $N = |A| = 26$.

Select p, q, and e. Set $n = pq$, $h = (p-1)(q-1)$, and find d so that $ed \equiv 1 \pmod{h}$. Let $k = \lfloor \log_{26} n \rfloor$.

The encryption converts blocks of letters of size k to blocks of size $k+1$; this is because the modular power function may output a number between N^k and n. Thus, the decryption must convert blocks of size $k+1$ to blocks of size k.

We have these subroutines:

- `chrupr`, `chrval`, `chrlet` handle conversion from characters to numbers
- `getarr`, `putarr` handle stream input/output to/from arrays of numbers

```
// Stream/Array functions

#define BAS 26

char chrupr(char chr)
{ if (chr >= 'a' && chr <= 'z') return chr-'a'+'A';
  if (chr >= 'A' && chr <= 'Z') return chr;
  return 0; }

char chrval(char chr)
{ return chr-'A'; }

char chrlet(char chr)
{ return chr+'A'; }

int getarr(FILE* ori,BYT* arr,UNT sze)
{ UNT ctr=0;
  int chr;
  memset(arr,0,sze);
  while (ctr<sze)
  { chr = fgetc(ori);
    if (chr==EOF) return ctr;
    if (!(chr=chrupr(chr))) continue;
    arr[ctr++]=chrval(chr); }
  return true; }

void putarr(FILE* trg,BYT* arr,UNT sze)
{ UNT ctr=0;
  while (ctr<sze)
  { fputc(chrlet(arr[ctr++]),trg); } }
```

Now we code the encryption/decryption functions thusly.

- `encarr`, `decarr` handle (un)packing and (de)encrypting of arrays
- `rsa` encrypts, `asr` decrypts

```
// Encryption/Decryption Functions

void encarr(BYT* arr,UNT sze,UNT num,UNT exp) // encrypt array
{ UNT pak=0;
  pak = pakarr(arr,sze,BAS);
  pak = modpow(pak,exp,num);
  pakbak(arr,sze+1,BAS,pak); }

void decarr(BYT* arr,UNT sze,UNT num,UNT exp) // decrypt array
{ UNT pak=0;
  pak = pakarr(arr,sze+1,BAS);
  pak = modpow(pak,exp,num);
  pakbak(arr,sze,BAS,pak); }

void rsa(FILE *ori,FILE *trg,UNT num,UNT exp) // encrypt file
{ UNT sze=paksze(num,BAS);
  BYT* arr = new BYT[sze+1];
  while (getarr(ori,arr,sze))
  { encarr(arr,sze,num,exp);
    putarr(trg,arr,sze+1); }
  delete arr; }

void asr(FILE *ori,FILE *trg,UNT num,UNT exp) // decrypt file
{ UNT sze=paksze(num,BAS);
  BYT* arr = new BYT[sze+1];
  while (getarr(ori,arr,sze+1))
  { decarr(arr,sze,num,exp);
    putarr(trg,arr,sze); }
  delete arr; }
```

Department of Mathematics and CSci, Southern Arkansas University
E-mail address: `plbailey@saumag.edu`