CPSC 367: Cryptography and Security

Michael J. Fischer

Lecture 9
February 14, 2019

Tools Needed for RSA

Recall: RSA cryptosystem

- Choose two large primes p, q.
- Compute $n = pq$ and $\phi(n) = (p-1)(q-1)$.
- Choose exponents e, d such that $ed \equiv 1 \pmod{\phi(n)}$.
- It follows that $m^{ed} \equiv 1 \pmod{n}$.
- $E_{k_e}(m) = c = m^e \bmod n$.
- $D_{k_d}(c) = m = c^d \bmod n$.

Tools

Two kinds of tools are needed to understand and implement RSA.

Algorithms: Need clever algorithms for primality testing, fast exponentiation, and modular inverse computation.

Number theory: Need some theory of Z_n, the integers modulo n, and some special properties of numbers n that are the product of two primes.

Algorithms

Algorithms for arithmetic on big numbers

The arithmetic built into typical computers can handle only 32-bit or 64-bit integers. Hence, all arithmetic on large integers must be performed by software routines.

The straightforward algorithms for addition and multiplication have time complexities $O(N)$ and $O(N^2)$, respectively, where N is the length (in bits) of the integers involved.

Asymptotically faster multiplication algorithms are known, but they involve large constant factor overheads. It's not clear whether they are practical for numbers of the sizes used for cryptography.

Big number libraries

A lot of cleverness *is* possible in the careful implementation of even the $O(N^2)$ multiplication algorithms, and a good implementation can be many times faster in practice than a poor one. Big number multiplication is also hard to get right because of many special cases that must be handled correctly!

Most people sensibly choose to use big number libraries written by others rather than write their own code.

Two such libraries that you can use in this course:

1. GMP (GNU Multiple Precision Arithmetic Library);
2. The big number routines in the openssl crypto library.

GMP

GMP provides a large number of highly-optimized function calls for use with C and C++.

It is preinstalled on all of the Zoo nodes and supported by the open source community. Type `info gmp` at a shell for documentation.

Openssl crypto package

OpenSSL is a cryptography toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) network protocols and related cryptography standards required by them.

It is widely used and pretty well debugged. The implementation requires computation on big numbers. OpenSSL implements its own big number routines which are contained in its crypto library.

Type `man crypto` for general information about the crypto library. Details on the hundreds of individual functions are summarized here. Big number man pages are located on the Zoo in `/usr/share/man/man3/` and begin with the "`BN_`" prefix.

Modular exponentiation

The basic operation of RSA is modular exponentiation of big numbers, i.e., computing $m^e \bmod n$ for big numbers m, e, and n.

The obvious way to compute this would be to compute first $t = m^e$ and then compute $t \bmod n$.

This has two serious drawbacks.

Computing m^e the conventional way is too slow

The simple iterative loop to compute m^e requires e multiplications, or about 2^{1024} operations in all. This computation would run longer than the current age of the universe (which is estimated to be 15 billion years).

Assuming one loop iteration could be done in one microsecond (very optimistic seeing as each iteration requires computing a product and remainder of big numbers), only about 30×10^{12} iterations could be performed per year, and only about 450×10^{21} iterations in the lifetime of the universe. But $450 \times 10^{21} \approx 2^{79}$, far less than $e - 1$.

The result of computing m^e is too big to write down.

The number m^e is too big to store! This number, when written in binary, is about $1024 * 2^{1024}$ bits long, a number far larger than the number of atoms in the universe (which is estimated to be only around $10^{80} \approx 2^{266}$).

Controlling the size of intermediate results

The trick to get around the second problem is to do all arithmetic modulo n, that is, reduce the result modulo n after each arithmetic operation.

The product of two length ℓ numbers is only length 2ℓ before reduction mod n, so in this way, one never has to deal with numbers longer than about 2048 bits.

Question to think about: Why is it correct to do this?

Efficient exponentiation

The trick to avoiding the first problem is to use a more efficient exponentiation algorithm based on repeated squaring.

For the special case of $e = 2^k$, one computes $m^e \bmod n$ as follows:

$$\begin{aligned} m_0 &= m \\ m_1 &= (m_0 * m_0) \bmod n \\ m_2 &= (m_1 * m_1) \bmod n \\ &\vdots \\ m_k &= (m_{k-1} * m_{k-1}) \bmod n. \end{aligned}$$

Clearly, $m_i = m^{2^i} \bmod n$ for all i.

Combining the m_i for general e

For values of e that are not powers of 2, $m^e \bmod n$ can be obtained as the product modulo n of certain m_i's.

Express e in binary as $e = (b_s b_{s-1} \ldots b_2 b_1 b_0)_2$. Then $e = \sum_i b_i 2^i$, so

$$m^e = m^{\sum_i b_i 2^i} = \prod_i m^{b_i 2^i} = \prod_i (m^{2^i})^{b_i} = \prod_{i\,:\,b_I=1} m_i.$$

Since each $b_i \in \{0, 1\}$, we include exactly those m_i in the final product for which $b_i = 1$. Hence,

$$m^e \bmod n = \prod_{i\,:\,b_I=1} m_i \bmod n.$$

Towards greater efficiency

It is not necessary to perform this computation in two phases.

Rather, the two phases can be combined together, resulting in slicker and simpler algorithms that do not require the explicit storage of the m_i's.

We give both a recursive and an iterative version. They're both based on the identities[1]

$$m^e = \begin{cases} (m^2)^{\lfloor e/2 \rfloor} & \text{if } e \text{ is even;} \\ (m^2)^{\lfloor e/2 \rfloor} \times m & \text{if } e \text{ is odd;} \end{cases}$$

[1] $\lfloor e/2 \rfloor$ is the greatest integer less than or equal to $e/2$.

A recursive exponentiation algorithm

Here is a recursive version written in C notation, but it should be understood that this C program only works for numbers smaller than 2^{16}. To handle larger numbers requires the use of big number functions.

```
/* computes m^e mod n recursively */
int modexp( int m, int e, int n) {
  int r;
  if ( e == 0 ) return 1;          /* m^0 = 1 */
  r = modexp(m*m % n, e/2, n);     /* r = (m^2)^(e/2) mod n */
  if ( (e&1) == 1 ) r = r*m % n;   /* handle case of odd e */
  return r;
}
```

An iterative exponentiation algorithm

This same idea can be expressed iteratively to achieve even greater efficiency.

```
/* computes m^e mod n iteratively */
int modexp( int m, int e, int n) {
  int r = 1;
  while ( e > 0 ) {
    if ( (e&1) == 1 ) r = r*m % n;
    e /= 2;
    m = m*m % n;
  }
  return r;
}
```

Correctness

The loop invariant is

$$e > 0 \wedge (m_0^{e_0} \bmod n = rm^e \bmod n) \tag{1}$$

where m_0 and e_0 are the initial values of m and e, respectively.

Proof of correctness:

- It is easily checked that (1) holds at the start of each iteration.
- If the loop exits, then $e = 0$, so $r \bmod n$ is the desired result.
- Termination is ensured since e gets reduced during each iteration.

A minor optimization

Note that the last iteration of the loop computes a new value of m that is never used. A slight efficiency improvement results from restructuring the code to eliminate this unnecessary computation. Following is one way of doing so.

```
/* computes m^e mod n iteratively */
int modexp( int m, int e, int n) {
  int r = ( (e&1) == 1 ) ? m % n : 1;
  e /= 2;
  while ( e > 0 ) {
    m = m*m % n;
    if ( (e&1) == 1 ) r = r*m % n;
    e /= 2;
  }
  return r;
}
```

Number Theory

Factoring assumption

The *factoring problem* is to find a prime divisor of a composite number n.

The *factoring assumption* is that there is no probabilistic polynomial-time algorithm for solving the factoring problem, even for the special case of an integer n that is known to be the product of just two distinct primes.

The security of RSA is based on the factoring assumption. No feasible factoring algorithm for such numbers is known, but there is no proof that such an algorithm does not exist.

How big is big enough?

The security of RSA depends on n, p, q being sufficiently large.

What is sufficiently large? Nowadays, n is typically chosen to be 2048 or 3072 bits long. (See NIST Special Publication (SP) 800-57 Part 3, Rev. 1.)

The primes p and q whose product is n are generally chosen be roughly the same length, so each will be about half as long as n.

Number theory overview

In this and following sections, we review some number theory that is needed for understanding RSA.

I will provide only a high-level overview. Further details are contained in course handouts and the textbooks.

Bare-bones definitions

The following definitions apply to the RSA parameters.

- p, q are distinct large primes of roughly the same length.
- $n = pq$.
- $\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$.
- $\phi(n) = (p-1)(q-1)$. [ϕ is Euler's totient function.]
- $\mathbf{Z}_n^*$ are the numbers in $\mathbf{Z}_n$ that are relatively prime to n (that is, not divisible by either p or q, which is most of them).

Fact: $|\mathbf{Z}_n^*| = \phi(n)$.

Example: Small RSA

- $p = 11$, $q = 13$.
- $n = p \times q = 143$.
- $\mathbf{Z}_{143} = \{0, 1, 2, \ldots, 141, 142\}$.
- $\phi(143) = (p-1)(q-1) = 10 \times 12 = 120$.
- $\mathbf{Z}^*_{143} = \mathbf{Z}_{143} - M_{11} - M_{13}$, where
  $M_{11} = \{0, 11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132\}$,
  $M_{13} = \{0, 13, 26, 39, 52, 65, 78, 91, 104, 117, 130\}$.
- $$\begin{aligned} |\mathbf{Z}^*_{143}| &= |\mathbf{Z}_{143}| - |M_{11}| - |M_{13}| + |M_{11} \cap M_{13}| \\ &= 143 - 13 - 11 + 1 \\ &= 120 = \phi(143). \end{aligned}$$

Summary of what is needed

Here's a summary of the number theory needed to understand RSA and its associated algorithms.

- Greatest common divisor, $\mathbf{Z}_n$, mod n, $\phi(n)$, $\mathbf{Z}_n^*$, and how to add, subtract, multiply, and find inverses mod n.
- Euler's theorem: $a^{\phi(n)} \equiv 1 \pmod{n}$ for $a \in \mathbf{Z}_n^*$.
- How to generate large prime numbers: density of primes and testing primality.

Generating the RSA modulus

To generate an RSA modulus n, we find primes p and q of the desired lengths and compute $n = pq$.

To find a large prime, repeatedly choose numbers of the desired length and *test each for primality*. We must show that the *density of primes* is large enough for this procedure to be feasible.

Finding the RSA key pair

- The RSA key pair (e, d) is chosen to satisfy the *modular equation* $ed \equiv 1 \pmod{\phi(n)}$.
- To find (e, d):
  1. Choose random $e \in \mathbf{Z}^*_{\phi(n)}$. Do this by repeatedly choosing e at random from $\mathbf{Z}_{\phi(n)}$ and using gcd to test membership in $\mathbf{Z}^*_{\phi(n)}$.
  2. *Solve* the modular equation $ed \equiv 1 \pmod{\phi(n)}$ for d.

Using *Euler's theorem*, we can show

$$m^{ed} \equiv m \pmod{n}$$

for all $m \in \mathbf{Z}^*_n$. This implies $D_d(E_e(m)) = m$.

To show that decryption works even in the rare case that $m \in \mathbf{Z}_n - \mathbf{Z}^*_n$ requires some more number theory that we will omit.

Quotient and remainder

Theorem (Euclidean division)

Let a, b be integers and assume $b > 0$. There are unique integers q (the quotient) and r (the remainder) such that $a = bq + r$ and $0 \leq r < b$.

Write the quotient as $a \div b$ and the remainder as $a \bmod b$. Then

$$a = b \times (a \div b) + (a \bmod b).$$

Equivalently,

$$a \bmod b = a - b \times (a \div b).$$

$$a \div b = \lfloor a/b \rfloor.^2$$

[2] Here, / is ordinary real division and $\lfloor x \rfloor$, the *floor* of x, is the greatest integer $\leq x$. In C, / is used for both $\div$ and / depending on its operand types.

The mod operator for negative numbers

When either a or b is negative, there is no consensus on the definition of $a \bmod b$.

By our definition, $a \bmod b$ is always in the range $[0 \ldots b-1]$, even when a is negative.

Example,

$$(-5) \bmod 3 = (-5) - 3 \times ((-5) \div 3) = -5 - 3 \times (-2) = 1.$$

The mod operator % in C

In the C programming language, the mod operator % is defined differently, so $(a \,\%\, b) \neq (a \bmod b)$ when a is negative and b is positive.

The C standard defines $a \,\%\, b$ to be the number r satisfying the equation $(a/b) * b + r = a$, so $r = a - (a/b) * b$.

C also defines a/b to be the result of rounding the real number a/b towards zero, so $-5/3 = -1$. Hence,

$$-5 \,\%\, 3 = -5 - (-5/3) * 3 = -5 + 3 = -2.$$