CPSC 467b: Cryptography and Computer Security

Lecture 8

Michael J. Fischer

Department of Computer Science
Yale University

February 3, 2010

RSA review

Key generation

Here's how Bob generates an RSA key pair.

- Bob chooses two sufficiently large distinct prime numbers p and q and computes $n = pq$.
  For security, p and q should be about the same length (when written in binary).
- He computes two numbers e and d with a certain number-theoretic relationship.
- The public key is the pair $k_e = (e, n)$. The private key is the pair $k_d = (d, n)$. The primes p and q are no longer needed and should be discarded.

RSA review

Encryption and decryption

To encrypt, Alice computes $c = m^e \bmod n$.

To decrypt, Bob computes $m = c^d \bmod n$.

This works because $m = (m^e \bmod n)^d \bmod n$ for all m.

Questions

- How does one find n, e, d?
- Why is RSA believed to be secure?
- How can one implement RSA when most computers only support arithmetic on 32-bit or 64-bit integers?
- How can one possibly compute $m^e \bmod n$ for 1024 bit numbers?

Tools needed to answer RSA questions

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.

Factoring assumption

The security of RSA is based on the premise that the factoring problem on large integers is infeasible, even when the integers are known to be the product of just two distinct primes.

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

No feasible algorithm for solving the factoring problem is known, even in the special case of an RSA modulus n.

How big is big enough?

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

What is sufficiently large? That's hard to say, but n is typically chosen to be at least 1024 bits long, or for better security, 2048 bits long.

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.

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 we are talking about.

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. They are also hard to get right because of many special cases that must be handled correctly!

Most people 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 protocol requires cryptography, and OpenSSL implements its own big number routines which are contained in its crypo library.

Type `man crypto` for general information about the library, and `man bn` for the specifics of the big number routines.

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 $t \bmod n$.

Difficulty of modular exponentiation

1. 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 $10^{80} \approx 2^{266}$).
2. 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 $e - 1$.

Controlling the size of intermediate results

The trick to get around the first 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 here is to use a more efficient exponentiation algorithm based on repeated squaring. To compute $m^e \bmod n$ where $e = 2^k$, one computes

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

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

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

Express e in binary as $e = (b_s b_{s-1} \ldots b_2 b_1 b_0)_2$ and include m_i in the final product if and only if $b_i = 1$.

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.

A recursive exponentiation algorithm

Here is a recursive version written in C notation, but it should be understood that the C programs only work 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 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.

Quotient and remainder

Theorem (division theorem)

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 $0 \leq r < b$.

We denote the quotient by $a \div b$ and the remainder by $a \bmod b$. It follows that

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

or equivalently,

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

The latter actually defines mod in terms of $\div$.
$\div$ in turn can be defined as $a \div b = \lfloor a/b \rfloor$.[1]

[1] 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.

mod 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,

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

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.[2]

[2] For those of you who are interested, the C standard defines $a \,\%\, b$ to be the number satisfying the equation $(a/b) * b + (a \,\%\, b) = a$. C also defines a/b to be the result of rounding the real number a/b towards zero, so $-5/3 = -1$. Hence, $-5 \,\%\, 3 = -5 - (-5/3) * 3 = -5 + 3 = -2$ in C.

Divides

We say that *b divides a* (exactly) and write $b \mid a$ in case $a \bmod b = 0$.

Fact

If $d \mid (a + b)$, then either d divides both a and b, or d divides neither of them.

To see this, suppose $d \mid (a + b)$ and $d \mid a$. Then by the division theorem, $a + b = dq_1$ and $a = dq_2$ for some integers q_1 and q_2. Substituting for a and solving for b, we get

$$b = dq_1 - dq_2 = d(q_1 - q_2).$$

But this implies $d \mid b$, again by the division theorem.

The mod relation

We just saw that mod is a binary operation on integers.

Mod is also used to denote a relationship on integers:

$$a \equiv b \pmod{n} \quad \text{iff} \quad n \mid (a - b).$$

That is, a and b have the same remainder when divided by n. An immediate consequence of this definition is that

$$a \equiv b \pmod{n} \quad \text{iff} \quad (a \bmod n) = (b \bmod n).$$

Thus, the two notions of mod aren't so different after all!

We sometimes write $a \equiv_n b$ to mean $a \equiv b \pmod{n}$.

Mod is an equivalence relation

The two-place relationship $\equiv_n$ is an *equivalence relation*.

Its equivalence classes are called *residue* classes modulo n and are denoted by $[b]_{\equiv_n} = \{a \mid a \equiv b \pmod{n}\}$ or simply by $[b]$.

For example, if $n = 7$, then $[10] = \{\ldots -11, -4, 3, 10, 17, \ldots\}$.

Fact

$[a] = [b]$ *iff* $a \equiv b \pmod{n}$.

Canonical names

If $x \in [b]$, then x is said to be a *representative* or *name* of the equivalence class $[b]$. Obviously, b is a representative of $[b]$. Thus, $[-11]$, $[-4]$, $[3]$, $[10]$, $[17]$ are all names for the same equivalence class.

The *canonical* or preferred name for the class $[b]$ is the unique integer in $[b] \cap \{0, 1, \ldots, n-1\}$.

Thus, the canonical name for $[10]$ is $10 \bmod 7 = 3$.