CPSC 467b: Cryptography and Computer Security

Michael J. Fischer

Lecture 7
January 30, 2012

Public-key cryptography

Classical cryptography uses a single key for both encryption and decryption. This is also called a *symmetric* or *1-key* cryptography.

There is no logical reason why the encryption and decryption keys should be the same.

Allowing them to differ gives rise to *asymmetric* cryptography, also known as *public-key* or *2-key* cryptography.

Asymmetric cryptosystems

An *asymmetric cryptosystem* has a pair $k = (k_e, k_d)$ of related keys, the *encryption key* k_e and the *decryption key* k_d.

Alice encrypts a message m by computing $c = E_{k_e}(m)$.
Bob decrypts c by computing $m = D_{k_d}(c)$.

- We sometimes write e instead of k_e and d instead of k_d, e.g., $E_e(m)$ and $D_d(c)$.
- We sometimes write k instead of k_e or k_d where the meaning is clear from context, e.g., $E_k(m)$ and $D_k(c)$.

In practice, it isn't generally as confusing as all this, but the potential for misunderstanding is there.

As always, the decryption function inverts the encryption function, so $m = D_d(E_e(m))$.

Security requirement

Should be hard for Eve to find m given $c = E_e(m)$ *and* e.

- The system remains secure even if the encryption key e is made public!
- e is said to be the *public key* and d the *private key*.

Reason to make e public.

- Anybody can send an encrypted message to Bob. Sandra obtains Bob's public key e and sends $c = E_e(m)$ to Bob.
- Bob recovers m by computing $D_d(c)$, using his private key d.

This greatly simplifies key management. No longer need a secure channel between Alice and Bob for the initial key distribution (which I have carefully avoided talking about so far).

Man-in-the-middle attack against 2-key cryptosystem

An active adversary Mallory can carry out a nasty *man-in-the-middle* attack.

- Mallory sends his own encryption key to Sandra when she attempts to obtain Bob's key.
- Not knowing she has been duped, Sandra encrypts her private data using Mallory's public key, so Mallory can read it (but Bob cannot)!
- To keep from being discovered, Mallory intercepts each message from Sandra to Bob, decrypts using his own decryption key, re-encrypts using Bob's public encryption key, and sends it on to Bob. Bob, receiving a validly encrypted message, is none the wiser.

Passive attacks against a 2-key cryptosystem

Making the encryption key public also helps a passive attacker.

1. Chosen-plaintext attacks are available since Eve can generate as many plaintext-ciphertext pairs as she wishes using the public encryption function $E_e()$.
2. The public encryption function also gives Eve a foolproof way to check the validity of a potential decryption. Namely, Eve can verify $D_d(c) = m_0$ for some candidate message m_0 by checking that $c = E_e(m_0)$.
   Redundancy in the set of meaningful messages is no longer necessary for brute force attacks.

Facts about asymmetric cryptosystems

Good asymmetric cryptosystems are much harder to design than good symmetric cryptosystems.

All known asymmetric systems are orders of magnitude slower than corresponding symmetric systems.

Hybrid cryptosystems

Asymmetric and symmetric cryptosystems are often used together. Let (E^2, D^2) be a 2-key cryptosystem and (E^1, D^1) be a 1-key cryptosystem.

Here's how Alice sends a secret message m to Bob.

- Alice generates a random *session key* k.
- Alice computes $c_1 = E^1_k(m)$ and $c_2 = E^2_e(k)$, where e is Bob's public key, and sends (c_1, c_2) to Bob.
- Bob computes $k = D^2_d(c_2)$ using his private decryption key d and then computes $m = D^1_k(c_1)$.

This is much more efficient than simply sending $E^2_e(m)$ in the usual case that m is much longer than k.

Note that the 2-key system is used to encrypt random strings!

RSA

Overview of RSA

Probably the most commonly used asymmetric cryptosystem today is *RSA*, named from the initials of its three inventors, Rivest, Shamir, and Adelman.

Unlike the symmetric systems we have been talking about so far, RSA is based not on substitution and transposition but on arithmetic involving very large integers—numbers that are hundreds or even thousands of bits long.

To understand why RSA works requires knowing a bit of number theory. However, the basic ideas can be presented quite simply, which I will do now.

RSA spaces

The message space, ciphertext space, and key space for RSA is the set of integers $\mathbf{Z}_n = \{0, \ldots, n-1\}$ for some very large integer n.

For now, think of n as a number so large that its binary representation is 1024 bits long.

Such a number is unimaginably big. It is bigger than $2^{1023} \approx 10^{308}$.

For comparison, the number of atoms in the observable universe[1] is estimated to be "only" 10^{80}.

[1] Wikipedia, https://en.wikipedia.org/wiki/Observable_universe

Encoding bit strings by integers

To use RSA as a block cipher on bit strings, Alice must convert each block to an integer $m \in \mathbf{Z}_n$, and Bob must convert m back to a block.

Many such encodings are possible, but perhaps the simplest is to prepend a "1" to the block x and regard the result as a binary integer m.

To decode m to a block, write out m in binary and then delete the initial "1" bit.

To ensure that $m < n$ as required, we limit the length of our blocks to 1022 bits.

RSA 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 encryption and decryption

To encrypt, Alice computes $c = m^e \bmod n$.[2]

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

Here, $a \bmod n$ denotes the remainder when a is divided by n.

This works because e and d are chosen so that, for all m,

$$m = (m^e \bmod n)^d \bmod n. \tag{1}$$

That's all there is to it, once the keys have been found.

Most of the complexity in implementing RSA has to do with key generation, which fortunately is done only infrequently.

[2] For now, assume all messages and ciphertexts are integers in $\mathbf{Z}_n$.

RSA questions

You should already be asking yourself the following questions:

- How does one find n, e, d such that equation 1 is satisfied?
- Why is RSA believed to be secure?
- How can one implement RSA on a computer when most computers only support arithmetic on 32-bit or 64-bit integers, and how long does it take?
- How can one possibly compute m^e mod n for 1024 bit numbers. m^e, before taking the remainder, has size roughly

$$\left(2^{1024}\right)^{2^{1024}} = 2^{1024 \times 2^{1024}} = 2^{2^{10} \times 2^{1024}} = 2^{2^{1034}}.$$

This is a number that is roughly 2^{1034} bits long! No computer has enough memory to store that number, and no computer is fast enough to compute it.

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

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 the product of just two distinct primes

The security of RSA is based on the factoring assumption. No feasible factoring algorithm 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? 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.

Computing with Big Numbers

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 protocols require cryptography, and OpenSSL implements its own big number routines which are contained in its crypto library.

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

Fast Exponentiation Algorithms

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.

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 mod 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 $e = \sum_i 2^{b_i}$, so

$$m^e = m^{\sum_i 2^{b_i}} = \prod_i m^{2^{b_i}}.$$

Hence,

$$m^e \bmod n = \prod_i m^{2^{b_i}} \bmod n = \prod_{i:\ b_I=1} m_i \bmod n.$$

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

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

$$e > 0 \wedge (m_0^{e_0} \bmod n = rm^e \bmod n) \tag{2}$$

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

Proof of correctness:

- It is easily checked that (2) 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

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 $0 \leq r < b$.

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,

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

$$a \div b = \lfloor a/b \rfloor.^3$$

[3] 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.[4]

[4] 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 $10 \bmod 7 = 3$.