CPSC 467: Cryptography and Security

Michael J. Fischer

Lecture 10
October 1, 2020

Public-key Cryptography

RSA

RSA Security

Algorithms
- Computing with Big Numbers
- Fast Exponentiation Algorithms

Public-key Cryptography

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 and d as shorthand for k_e and k_d, respectively.

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 always 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 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 $c_1 = E_k^1(m)$ and $c_2 = E_e^2(k)$, where e is Bob's public key, and sends (c_1, c_2) to Bob.
- Bob computes $k = D_d^2(c_2)$ using his private decryption key d and then computes $m = D_k^1(c_1)$.

This is much more efficient than simply sending $E_e^2(m)$ 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 encode each block to an integer $m \in \mathbf{Z}_n$, and Bob must decode 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 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 such that $ed \equiv 1 \pmod{\phi(n)}$, where $\phi(n)$ is Euler's *totient* function.
- In the RSA case, $\phi(n) = (p-1)(q-1)$.
- 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 and the totient $\phi(n)$ 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.

Decryption works because the conditions on e and d ensure that $m^{ed} \bmod n = m$. Hence,

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

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 compute n, e, and d, and why are the given conditions sufficient for equation (1) to be 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.

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
  $M_{11} = \{0, 11, 22, 33, 44, 55, 66, 77, 88, 99, 110, 121, 132\}$,
  $M_{13} = \{0, 13, 26, 39, 52, 65, 78, 91, 104, 117, 130\}$.
- $$\begin{aligned} |\mathbf{Z}^*_{143}| &= |\mathbf{Z}_{143}| - |M_{11}| - |M_{13}| + |M_{11} \cap M_{13}| \\ &= 143 - 13 - 11 + 1 \\ &= 120 = \phi(143). \end{aligned}$$
- $e = 7$, $d = 103$, $ed = 721 \equiv 1 \pmod{120}$.

RSA Security

Breaking RSA

An attacker of RSA is assumed to have the public key $k_e = (e, n)$, a cyphertext c, and full knowledge of the RSA cryposystem. As usual, the attacker's goal is to find the corresponding plaintext m.

The public key k_e allows an attacker to check the correctness of a candidate decryption m' by testing that $E_e(m') = c$.

The public key also allows the attacker to generate as many chosen plaintext pairs (m_j, c_j) as desired.

As always, it must be hard for the attacker to find the decryption key $k_d = (d, n)$ or the system is completely broken. Because d is easily computed given e and the factorization of n, it must also be hard to factor n to find p and q.

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 depends 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. If such an algorithm is ever found, it can be used to break RSA.

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.

Algorithms

Implementing RSA

To implement RSA, we need clever algorithms:

1. For computing with huge numbers that are hundreds or thousands of bits long;
2. For fast exponentiation in order to compute the encryption and decryption functions;
3. For finding inverses mod n when they exist;
4. For finding large random prime numbers in order to generate an RSA modulus n.

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, $m_j = m^{2^j} \bmod n$ for all j.

Combining the m_j 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[3]

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

[3] The floor $\lfloor e/2 \rfloor$ 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;
}
```

Why doesn't this code work for 32-bit integers?

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;
}
```