Cryptology (4)

Introduction to Public Key Encryption and RSA

John York, Blue Ridge Community College

http://www.brcc.edu

Much of the information in this course came from “Understanding Cryptography” by Christoff Parr and Jan Pelzl, Springer-Verlag 2010
Much of the classical cryptography material and most Python scripts came from “Cracking Codes with Python” by Al Sweigart, NoStarch Press 2018
Also helpful, “Cryptography Engineering” by Ferguson, Schneier, and Kohno, Wiley Publishing, 2010

Problems with symmetric key encryption

- Symmetric algorithms like AES are fast and secure, but…
- Key Exchange
  - How do Alice and Bob agree on a key without Eve discovering it?
- Number of Keys
  - If we have a group of 10 people, each pair needs their own key
  - 10 choose 2 combinations = 45 keys. For 100 people it is 4,950 keys
- What if Alice or Bob cheat?
  - Alice and Bob both have the same key, each can say the other person sent the message with the contract, bill, etc.
  - How do we prove in court which one sent the message?

If it were not for the key exchange problem, we would almost always use symmetric encryption. Symmetric encryption is 1000 times faster than asymmetric encryption and has a much lighter CPU footprint.

If we have a large number of people, and want any given pair of people to be able to communicate without the others overhearing, there has to be one key for every possible pair of people. That number gets large quickly.

The other thing that symmetric encryption is missing is a mechanism for signatures. Since both parties to symmetric encryption have the same key, we have no way of proving which party sent a message. In general, we want a way of proving that a message or document came from a specific person or computer. This done with a signature.

Using public keys

- Key Exchange
  - RSA and Diffie-Hellman Key Exchange (DHKE) allow a secure exchange
  - There are still problems (how do we know Alice's public is really Alice's)
- Number of Keys
  - Each person has one public and one private key for encryption
  - Many systems add a second key pair for signing
- Encryption
  - data encrypted with my public key can only be decrypted with my private key
- Non-repudiation
  - Private keys can create signatures
  - Signature proves I sent the data, I can't change my mind later (no cheating)
- Identification
  - encrypt data with private key to prove who I am

Key Exchange allows Alice and Bob to agree on a secret key, even when they are communication over an unsecure network; Eve may be listening but she cannot determine the key. Bob can encrypt a session key with Eve's public key, and Eve can decrypt with her private key. We still have a problem--Bob has to be sure Alice's public key is really from Alice and not from an attacker.
The number of keys is reduced. With public key encryption, each person has one key pair. Each person can establish a secure connection with another person by using that person's public key. Typically a person/party may have two sets of keys, one for encryption and one for signing. This gets around attacks where someone tricks a user into signing a message/file that they've already encrypted, which breaks the encryption.
Non-repudiation and signature go together. If I sign a document with my private key, I cannot come back later and claim the document didn't come from me.
Identification allows a user to prove who they are. If you ask me to encrypt something with my private key, and you can decrypt it with my public key, you know you are talking to me.

The terms Key Exchange, Non-repudiation, Identification, and Encryption describe important problems that we are trying to solve in cryptology.

Public key algorithms are based on:

- Factoring large integers
  - "Large" means integers are at least 1024 bits long (2048 is common now)
  - Used in the RSA algorithm
- Discrete logarithm problem
  - $a = N^e$, finding e when you know a and N is difficult in discrete math
  - Used in Diffie-Hellman (DH) and Digital Signature Algorithm (DSA)
- Elliptic Curves
  - Discrete logarithm problem, extended to operate over elliptic curves
  - Elliptic curves are more difficult to compute, allowing for smaller keys
  - DH becomes ECDH, DSA becomes ECDSA (EC is elliptic curve)

When we talk about factoring large integers, the word "large" really means gigantic. A 2048 bit number is over 600 digits long when expressed in base 10 (decimal.) There are algorithms for factoring large integers that are faster than brute force (https://math.boisestate.edu/~liljanab/BOISECRYPTFall09/Jacobsen.pdf) but no algorithm has been found that can factor 2048-bit numbers in a reasonable amount of time, provided the factors were properly chosen. A good laptop can factor a 384-bit integer (~315 decimal digits) in about 8 hours (https://techfindings.one/archives/1568) using YAFU (yet another factoring algorithm, https://github.com/DarkenCode/yafu)

Because of problems with selecting large integers and factors for RSA (https://eprint.iacr.org/2012/064.pdf), the discrete logarithm problem (and its variant, elliptic curve cryptography) is becoming the dominant public key encryption. The logarithm problem in continuous math is easy. You all have log and ln buttons on your calculators that work very well. You can solve $142 = 10^x$ in continuous math quickly. The equivalent problem in discrete math is not so easy.

Elliptic curve encryption is essentially the discrete logarithm problem, except that multiplication has been replaced by a new operation that makes the encryption work

better.

Key Length vs. Security

- Strength of encryption depends on key length and algorithm
- Chart compares Public Key strength to AES

- 3TDEA is 3DES or triple DES
- AES-128 is AES with a key length of 128 bits
- FFC is Finite Field Cryptography, i.e Discrete Logarithm, Diffie-Hellman
- IFC is Integer-Factorization Cryptography, i.e. RSA
- ECC is Elliptic Curve Cryptography
- L, k, and f ~ key length in bits
- RSA and DH need long keys, 2048 bits is current minimum
- ECC keys only need to be about 2X AES key length
- Yellow shading is for compatibility, not security

Security Strength	Symmetric key algorithms	FFC (e.g., DSA, D-H)	IFC (e.g., RSA)	ECC (e.g., ECDSA)
≤ 80	2TDEA[21]	$L = 1024$ $N = 160$	$k = 1024$	$f = 160\text{-}223$
112	3TDEA	$L = 2048$ $N = 224$	$k = 2048$	$f = 224\text{-}255$
128	AES-128	$L = 3072$ $N = 256$	$k = 3072$	$f = 256\text{-}383$
192	AES-192	$L = 7680$ $N = 384$	$k = 7680$	$f = 384\text{-}511$
256	AES-256	$L = 15360$ $N = 512$	$k = 15360$	$f = 512+$

https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf

This diagram is from the US National Institute of Standards (NIST). It attempts to compare the security of different encryption methods by comparing the difficulty of breaking that method with the difficulty of breaking symmetric encryption with a certain key length.

The left column is the equivalent security provided by symmetric encryption with that length of key. The minimum acceptable key length for symmetric encryption is currently 112 bits, although AES with at least 128 bit keys is usual. For Diffie-Helman (FFC column) and RSA (IFC column) the current key length is 2048 bits.

Note that the Elliptic Curve Cryptography (ECC) key length to provide the same security is much shorter than the key length for RSA or Diffie-Helman. The ECC key length is generally about twice as long as the symmetric key length. ECC has the same security for shorter keys because it replaces multiplication with a more complicated operation.

Note: In the FFC column, the key length is L, and N is the size of the subgroup the key belongs to. This is explained more in the next lesson.

RSA Public Key Encryption

- Ron Rivest, Adi Shamir, and Leonard Adleman, then professors at MIT, are credited with discovery in 1978
  - Rivest, Shamir, and Adlemen went on to found RSA Corporation
- Clifford Cocks, James Ellis, and Malcom Williamson invented a similar system at Britain's GCHQ (Government Communications HQ, similar to NSA for United States) in 1973
  - Their work was classified, and not released to the public until 1997

This article from the British GCHQ details the work their people did on public key encryption before it was discovered at MIT:
https://www.gchq.gov.uk/news-article/malcolm-john-williamson-1950-2015

An interesting version can be found here:
https://www.wired.com/1999/04/crypto/

So, the British GCHQ people discovered public key encryption first. Because of government secrecy they were unknown. The MIT people discovered it slightly later and became rich and famous.

Factoring Large Integers—RSA

- Factoring large (2048 bits long) integers is difficult
  - 768 bit RSA-style integer factored in 2009 with multi-university computing
    - Would take 1500 years on a single AMD 2.2 GHz Opteron CPU
  - 1024 bit integers may soon be in range of large-scale computing
  - 2048 bit integer is current recommendation
- No **known** algorithm for quick (enough) factoring
  - However, no one has proven that such an algorithm does not exist
- Quantum computing will break RSA
  - Will require many more quantum bits (qubits) than is currently feasible
  - The number 15 was factored in 2001 with 7 qubits
  - 56153 factored in 2012
  - 1005973 factored in 2018 with 89 qubits

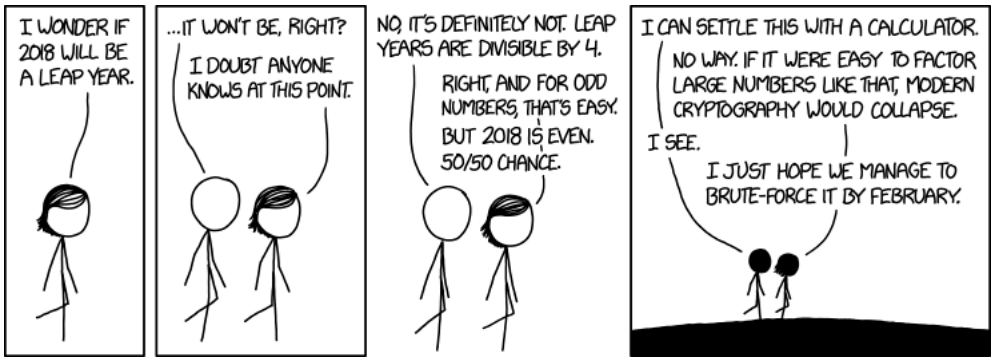
A 307 decimal digit number was factored in 2007. That's equivalent to a number 1000 bits long. Nation states can certainly factor 1024 bit numbers in 2018.
https://phys.org/news/2007-05-mighty-falls.html

Today, a good laptop can probably factor an integer in the range of 300-400 bits. In 2014 a person factored a 210 digit (base 10) number, 697 bits, using $7600 of time on AWS and 4 months of time on a personal computer.
https://en.wikipedia.org/wiki/Integer_factorization_records
2018--https://quantumcomputingreport.com/news/chinese-scientists-set-new-quantum-factoring-record/ 89 qubits

This link gives a general list of factoring algorithms.
https://en.wikipedia.org/wiki/Integer_factorization

If I had to break an RSA implementation for a CTF contest, I would first Google n to see if it were already factored. If that failed and n is less than 300 bits, I would try to factor it with YAFU (https://github.com/DarkenCode/yafu or https://sourceforge.net/projects/yafu/files/
"Post quantum cryptography" is a major field of research now. People are working

hard to have new algorithms in place before quantum computing is practical. If you Google “post quantum cryptography” you will get ~ 11 million results.



https://xkcd.com/1935/

This is a 1024 bit number in hex—much bigger than the integer 2018…

0xd0c5bafa70ed9a2547ac68d9053deb1ad062bc143bcb6ce4044697f6958a13178f462980cc2a0f8b27703e07c
9a4b6d9280324a188ab92bfb70c652dfada8be5aaf510484be333252073fd29eb383f9141ea2508211d79e1156
65470cfc169419a8cbe23b94550b343d30e70cca144a31eae15046b216e12a05787075536a170db4f7b2e67993
ae51d14ba016c2f3f90c5746d68e16725cacea56e8ae56fc619c597f2546d25888dd8c11db12052a91f7f3f58128
6cae294a1f7e0e380cb9fb1f24095d7de4b35d9de44c1b6340ab2c8429e4b0247fd385a881571b0cfce3b9e5b05
cdc799ff68ea3a260934d32a5b8fca3698b8b7d65fcef7fc3ace2503804b

And a 2048 bit number would be twice as long…

RSA Encryption

- Alice gives her public key to Bob
  - Key_{pub} = [n, e]

→

- Bob encrypts plaintext, x
  - Blocks of plaintext are small
  - Plaintext block smaller than key size
- Ciphertext $y = x^e \bmod n$
- Send y to Alice

←

- Alice decrypts with private key
  - Key_{priv} = [n, d]
  - $y^d = (x^e)^d = x^{ed} = x \bmod n$ is plaintext

The important point here is that e and d are multiplicative inverses in mod Λ(n) or Φ(n).

Bob uses the public key, e, to computer x^e mod n. When Alice computes $(x^e)^d$ mod n, e * d = 1, so she gets the original x.

The fact that you can't compute d, the inverse of e, without knowing how to factor n is what makes breaking the encryption difficult.

It is very important to remember that d and e are inverses mod Λ(n), or mod Φ(n). They are *not* inverses mod n. We are doing the encryption and decryption operations in mod n, but d and e were computed in mod Λ(n), or mod Φ(n). If you get these mixed up, you will cause yourself pain when you are solving RSA problems.

RSA key Generation

- Alice creates her keys
  - Choose Prime Numbers p and q
  - n = p * q
  - Compute Λ = lcm((p – 1)(q – 1))
  - Choose exponent, e, for public key
  - Make sure that GCD(e, Λ) = 1
  - Compute exponent, d, for private key
    - e and d are inverses, e * d = 1 mod Λ
  - Public key is [n, e]
  - Private key is [n, d]
  - p, q, Λ, and d are kept secret

- Basis for security is:
- Λ depends on p and q
- Need Λ to compute inverse of e
- Impractical to factor n to get p and q
- lcm is Least Common Multiple
  lcm(a, b) = a * b // gcd(a, b)
- Many books use Φ = (p-1)(q-1) instead of Λ. More on this later.

The key to the entire process is choosing two large prime numbers, randomly. If an attacker can guess either p or q they can break your encryption. Also, you want numbers p and q that haven't been used by anyone else. If someone else has used them, they will recognize your n and know how to factor it. See the "Importance of strong random number generation" heading in https://en.wikipedia.org/wiki/RSA_(cryptosystem) See also "Ron was wrong. Whit was right", https://eprint.iacr.org/2012/064.pdf

People have not always done well in selecting the prime numbers to create their n.

The key to RSA cryptography is that we encrypt by taking large integers to large powers mod n. To decrypt, we have to know the inverse of that large power, and computing the inverse mod n is difficult. We can use a shortcut to compute the inverse mod n because we know the factors of n. Since the attackers cannot factor n (hopefully) they cannot compute the inverse they need for decryption.

Note that the difference between the public and private key is arbitrary. Of the two keys, we choose one to be public and keep the other secret. In practice, it is better to use the smaller number for the public key, especially if it is very small.

RSA Notes

- Each encryption/decryption requires taking a large number to a large power, mod n
  - Takes a lot of computing, even with special algorithms
- Plaintext blocks are small
  - Plaintext number of bits < keysize
  - ~250 ASCII characters with 2048 bit key

For each block of data, either encryption or decryption requires you to take a ~2000 bit (or more) number to a ~2000 bit power. This takes time. Research has shown that using a small number for the public key still provides good security. However the private key still has to be large to provide good security. Generally this means that encryption is faster than decryption in RSA.

Standard power functions in programming languages and calculators overflow when using numbers this large. You must use special big number (or arbitrary length) libraries that are based on square and multiply algorithms. This is a good explanation. https://www.coursera.org/lecture/mathematical-foundations-cryptography/square-and-multiply-ty62K

Luckily for us, Python uses arbitrary length arithmetic all the time. The Python power function works fine, as long as we remember to use the modulo version. If we use pow(x, y) we get x^y using normal methods, which won't work for us. If we use pow(x, y, n) we get x^y mod n and get valid results for our large numbers.

Another problem is that the plaintext (converted to an integer) has to be smaller than the key size. We have to do a lot of work to encrypt/decrypt a block of 2047 bits.

RSA Generate Key (small numbers)

```
>>> from Crypto.Util.number import GCD, inverse
>>> p = 31
>>> q = 47
>>> n = p * q
>>> n
1457
>>> L = (p-1) * (q-1) // GCD((p-1), (q-1))
>>> L
690
>>> #pick e, make sure it is relatively prime with L
>>> e = 341
>>> GCD(e, L)
1
>>> d = inverse(e, L)
>>> d
431
```

- Alice creates her keys
  - (small numbers, not secure)
  - n = p * q = 1457
  - e = 341, d = 431
- Public key
  - 1457, 341 [n, e]
- Private key
  - 1457, 431 [n, d]

This example uses small (insecure) numbers in Python. An easy way to pick small prime numbers for practice problems is to use a page that lists the first 100 prime numbers like https://www.rsok.com/~jrm/first100primes.html.

Note, if you use the phi function = (p - 1)*(q - 1) instead of the lcm function you get L = 1380 instead of 690 as shown above. Keeping e = 341, the findModInverse function will return 1121 instead of 431 for d. Both will work. Common implementations use lcm instead of phi because the numbers are smaller, and faster to compute.

RSA Message (small numbers)

- Bob has Alice's public key
  - [n, e] = [1457, 341]
- Bob encrypts 1200
  - $1200^{341} \bmod 1457 = 1040$
- Bob sends 1040 to Alice

```
>>> n = 1457
>>> pubkey = 341
>>> plain = 1200
>>> pow(plain, pubkey, n)
1040
```

- Alice has her private key
  - [n, d] = [1457, 431]
- Alice receives 1040 from Bob
  - $1040^{431} \bmod 1457 = 1200$
- Alice has Bob's message

```
>>> n = 1457
>>> privkey = 431
>>> cipher = 1040
>>> pow(cipher, privkey, n)
1200
```

Note that the same example works with a private key computed from phi instead of lcm. In that case the private key is 1121 instead of 431.

```
>>> n = 1457
>>> pubkey = 341
>>> plain = 1200
>>> pow(plain, pubkey, n)
1040
>>>
>>> privkey = 1121
>>> cipher = 1040
>>> pow(cipher, privkey, n)
1200
>>>
```

Problems with the example

- Key is small—1457 could be factored by hand
  - Try each prime number
  - 31 is the 11^{th} prime, so it would not take long
- Small key size means message blocks are small
  - Possible numbers for plaintext are 2 – 1456
  - Using 7-bit ASCII ($2^7 = 128$) symbol set, that's 10 characters
  - Using only upper-case letters (26 symbols), that's 55 characters

Most of you could break this example with paper and pencil, or at least a calculator. If you download the first 100 prime numbers, and divide them into 1457 until you have the factors p and q. You would probably want to use Python or another language with big integers to compute 1200^{341} mod 1457 and 1040^{431} mod 1457.

If the message you are trying to encrypt, when converted to an integer, is bigger than n the algorithm won't work. We are using arithmetic modulo n, so any message larger than n will get changed to something between 0 and n - 1.

Bottom Line

- Everyone knows public keys
  - Everyone can encrypt data with public key
  - Since we are the only ones with the private key, only we can decrypt the data
- For two-way traffic between Alice and Bob
  - Alice and Bob each have their own public-private key pair
  - Alice and Bob know each other's public key
  - Alice encrypts messages to Bob with Bob's public key
  - Bob encrypts messages to Alice with Alice's public key
- Or, one encrypts a symmetric key with the other's public key
  - Alice and Bob switch to AES with the symmetric key

Encryption
Encrypt with public key => Decrypt with private key
Our private key is secret, so only we can decrypt

Signing
Encrypt with private key => Decrypt with public key
Only we have private key, so anyone can decrypt with our well known public key. What good is that? We can prove we sent the message, since only we have the private key.

Note: Do not encrypt and sign with the same key pair. You will undo your encryption. Oops!

The most common encryption method is to use RSA only for key exchange, to send the session key in a secure manner. Once both sides know the session key, they switch to a symmetric encryption algorithm so they can enjoy the speed advantage.