Cryptology (5)

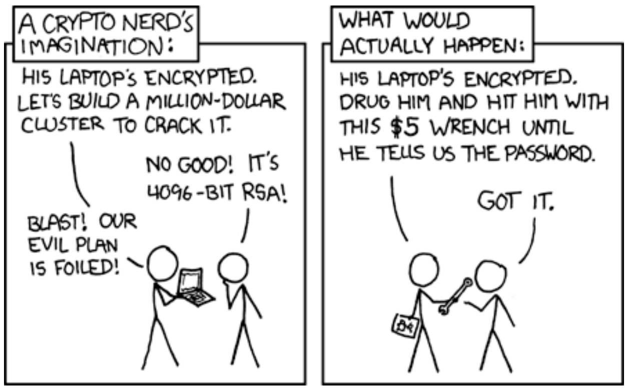
Public Key Encryption—RSA Math

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

Let's not get carried away, though…



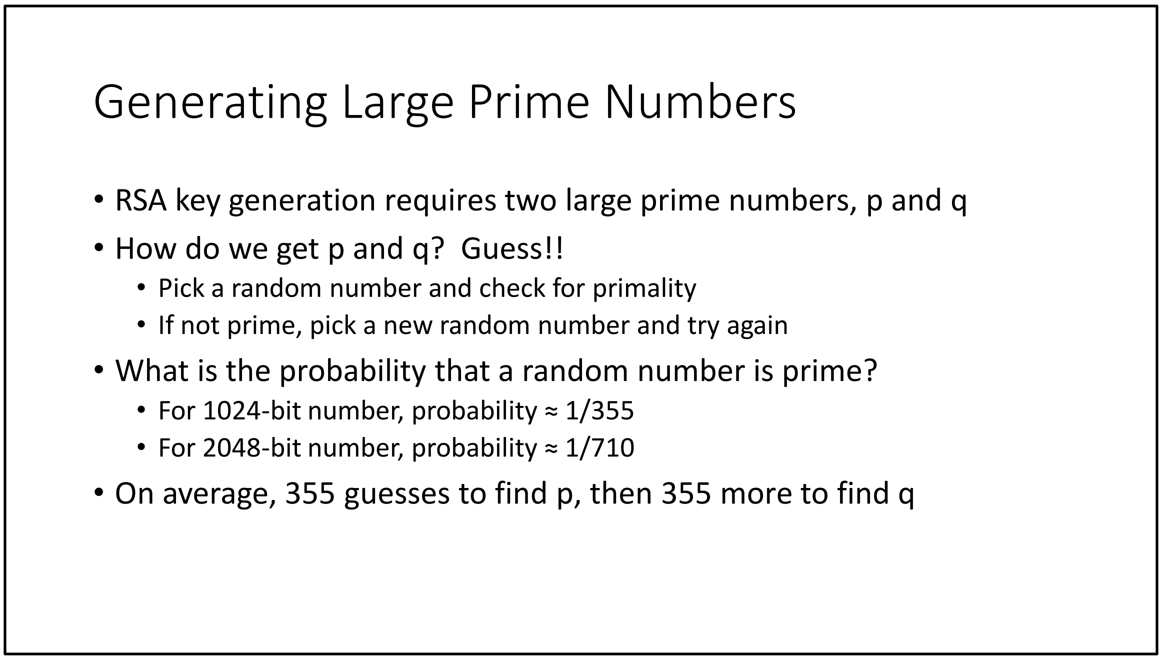
A CRYPTO NERD'S IMAGINATION:
HIS LAPTOP'S ENCRYPTED. LET'S BUILD A MILLION-DOLLAR CLUSTER TO CRACK IT.
NO GOOD! IT'S 4096-BIT RSA!
BLAST! OUR EVIL PLAN IS FOILED!
WHAT WOULD ACTUALLY HAPPEN:
HIS LAPTOP'S ENCRYPTED. DRUG HIM AND HIT HIM WITH THIS $5 WRENCH UNTIL HE TELLS US THE PASSWORD.
GOT IT.

https://imgs.xkcd.com/comics/security.png
"Actual reality: nobody cares about his secrets. (Also, I would be hard-pressed to find that wrench for $5.)"

This is probably the key to cryptography in the real world. Developers use very secure algorithms in their software (RSA 2048, AES 256, etc.) but make some minor implementation error. The secure algorithms are still secure, but the error causes the entire system to unravel.

If you replace "hit him with this $5 wrench" with "hammer the site until we find a bug," then you are in the real world.

Even if the bug is obscure and difficult, once it is published your security is gone.



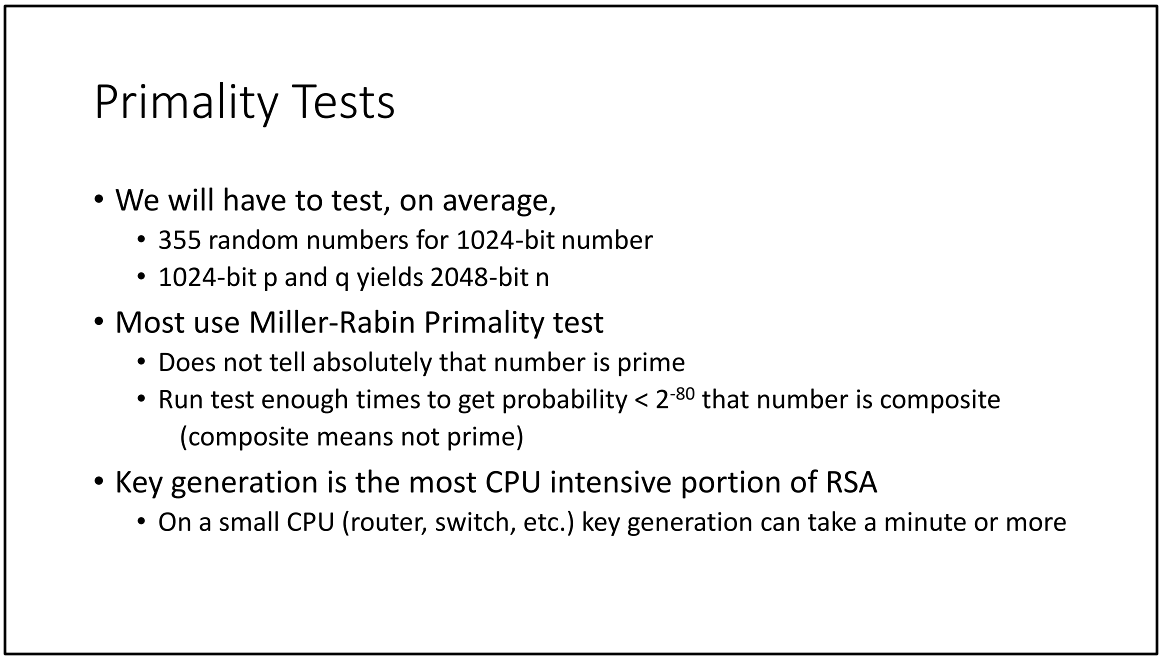
Generating Large Prime Numbers

- RSA key generation requires two large prime numbers, p and q
- How do we get p and q? Guess!!
  - Pick a random number and check for primality
  - If not prime, pick a new random number and try again
- What is the probability that a random number is prime?
  - For 1024-bit number, probability ≈ 1/355
  - For 2048-bit number, probability ≈ 1/710
- On average, 355 guesses to find p, then 355 more to find q

In selecting p and q, we have requirements:
-p and q must be prime numbers
-the product pq should have the length we need, usually 2048 bits
-p and q both be large (in the vicinity of 1024 bits for 2048 bit encryption) and close to the same size. “Cryptography Engineering”, pg 203.

Rather than use some formula to find p or q, we simply pick a random number of the proper number of bits. Then we test the number to see if it is prime. Obviously we will pick a lot of numbers that are not prime, and will have to use a lot of random numbers until we find one that is prime. This is why key generation is slow for small computers and routers.

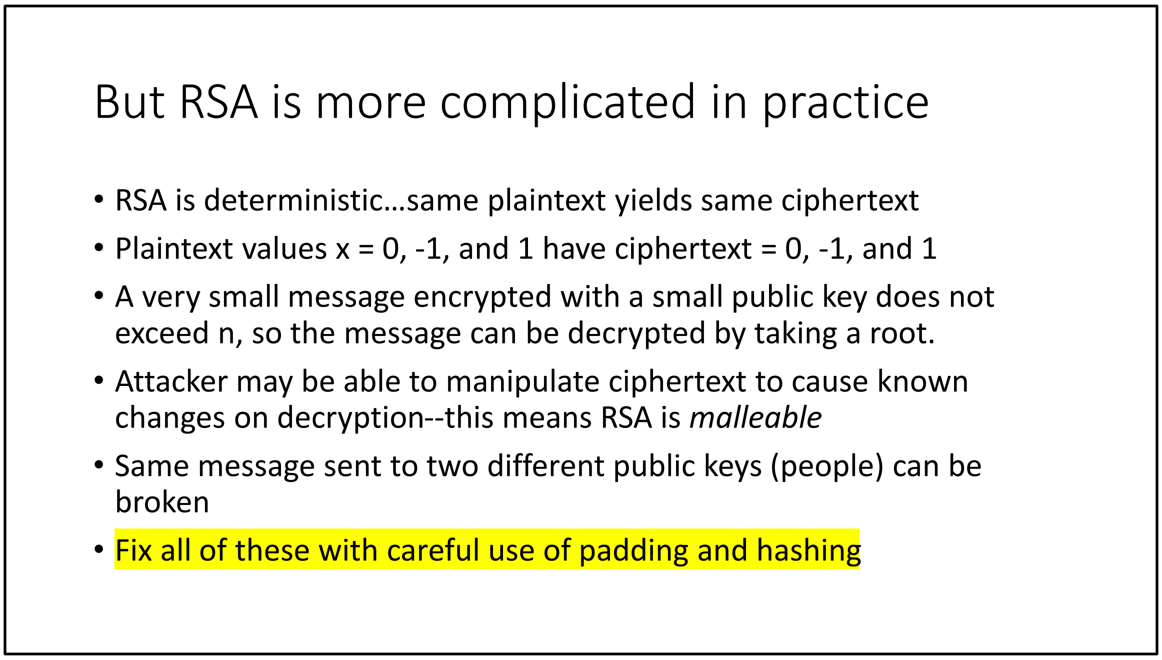
Probabilities from “understanding Cryptography,” Parr and Pelzel, Springer 2010, page 188.



Primality Tests

- We will have to test, on average,
  - 355 random numbers for 1024-bit number
  - 1024-bit p and q yields 2048-bit n
- Most use Miller-Rabin Primality test
  - Does not tell absolutely that number is prime
  - Run test enough times to get probability < 2^{-80} that number is composite (composite means not prime)
- Key generation is the most CPU intensive portion of RSA
  - On a small CPU (router, switch, etc.) key generation can take a minute or more

Testing a number for primality is fairly efficient (factoring a large number is hard.) Still, your computer may have to test many random numbers to find a prime. (The average number of attempts for a 1024 number is 355, but generating a number could easily take 500. Since you need two numbers, it may take 1000 attempts in total.
The primality test does not factor our candidate number. It simply tells us if the number is *likely* to be prime, using an algorithm that includes the candidate number and a random number. It is entirely possible that in a pass through the algorithm with a single random number, the algorithm may tell us the candidate is likely prime, even when the candidate is known to be composite (not prime.) However, if we run the test enough times with enough random numbers, we can reduce the probability of making an error to 2^{-80} or below. If our candidate number is 250 bits long, we need 11 tests. If the number is 600 bits long, we need only 3 tests. (Paar, page 191).
If Miller-Rabin says a number is composite, that is absolutely true. If it says the number is prime, there is an error probability of 2^{-80}.
A mathematical explanation of the Miller-Rabin primality test is available here.
https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test or in page 190 of Paar. A Python function for the test is available in Sweigart, page 331.



But RSA is more complicated in practice

- RSA is deterministic…same plaintext yields same ciphertext
- Plaintext values x = 0, -1, and 1 have ciphertext = 0, -1, and 1
- A very small message encrypted with a small public key does not exceed n, so the message can be decrypted by taking a root.
- Attacker may be able to manipulate ciphertext to cause known changes on decryption--this means RSA is *malleable*
- Same message sent to two different public keys (people) can be broken
- Fix all of these with careful use of padding and hashing

If you run RSA in the manner discussed so far, you will be vulnerable to many attacks. You can find them discussed

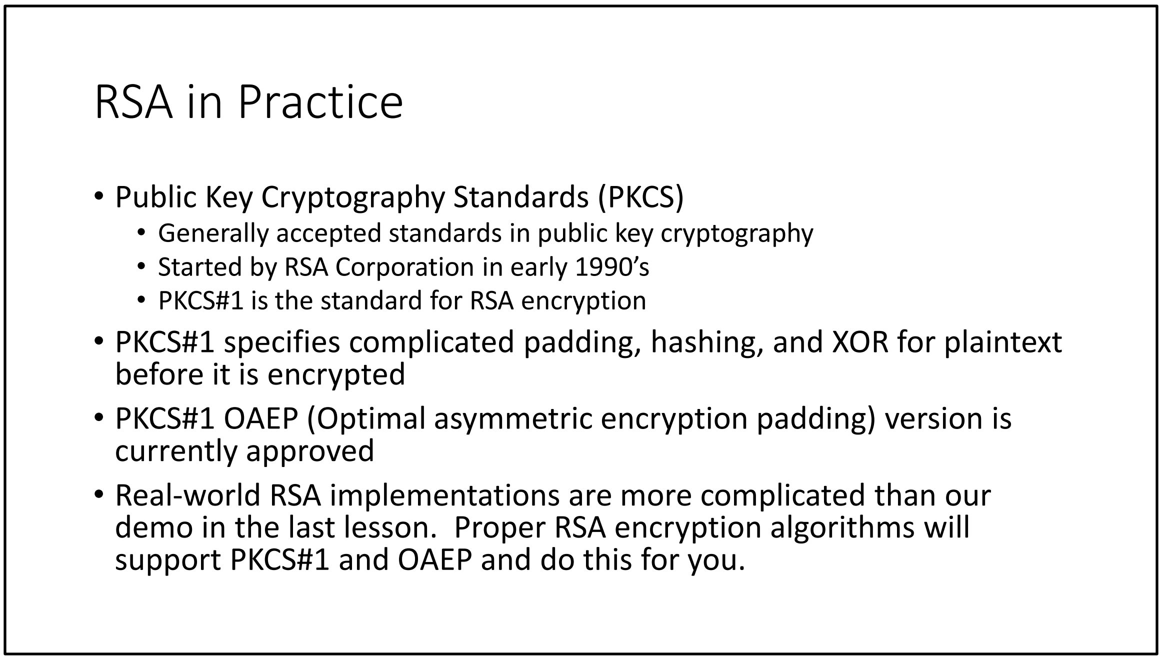
- Understanding Cryptography, Paar, section 7.7 RSA in Practice pg 192
- Cryptography Engineering, Ferguson, section 12.5 Pitfalls Using RSA pg 205
- https://en.wikipedia.org/wiki/RSA_(cryptosystem), Attacks against Plain RSA section

If you encrypt the same plaintext, the ciphertext will also be the same. We've already talked about this one.

For the small message and key problem, assume the public key e = 7, message m = 8. The encryption will perform $m^e = 8^7 = 2,097,152$. If n is larger than this, say 10,000,000, modular arithmetic will not come into play since 2,097,152 mod 10,000,000 = 2,097,152. (The number does not "wrap.") Then the attacker can recover m by taking the 7th root of 2,097,152.

Malleable (from Paar, pg 192). If an attacker doesn't know what's been encrypted, but just wants to double it, $(2^e * y)^d = 2^{ed} * y^{ed} = 2 * x \bmod n$. The plaintext has been doubled and the receiver may not realize it.

To use RSA in practice you have to use carefully chosen padding and hashing.



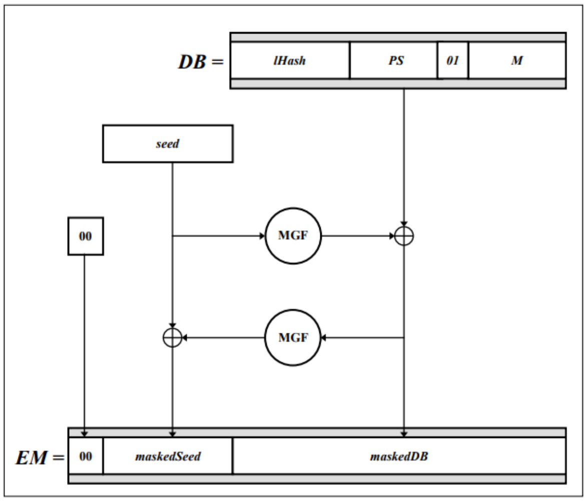
RSA in Practice

- Public Key Cryptography Standards (PKCS)
  - Generally accepted standards in public key cryptography
  - Started by RSA Corporation in early 1990's
  - PKCS#1 is the standard for RSA encryption
- PKCS#1 specifies complicated padding, hashing, and XOR for plaintext before it is encrypted
- PKCS#1 OAEP (Optimal asymmetric encryption padding) version is currently approved
- Real-world RSA implementations are more complicated than our demo in the last lesson. Proper RSA encryption algorithms will support PKCS#1 and OAEP and do this for you.

PKCS1 is specified in RFC 2013, 3447, and 8017. https://tools.ietf.org/html/rfc8017

This link is a copy of PKCS1, but with better drawings.
https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf
See Figure 1: EME-OAEP Encoding Operation on page 20. This is shown in the next slide.

PKCS#1 and OAEP are fancy names for a padding and hashing scheme that has been proven to fix the problems with schoolbook RSA.



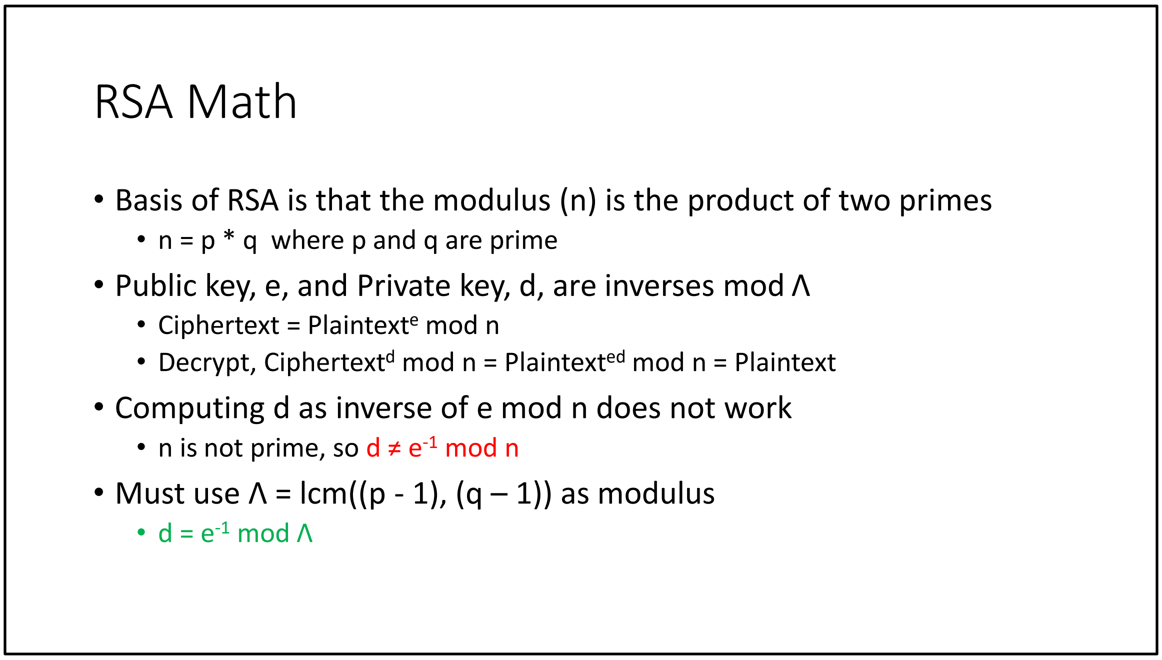
PKCS#1 Hash/Padding Scheme—not simple

M is the message
lHash—usually SHA hash of an empty string
PS—padding, 0's as needed for size
MGF—Mask Generation Function, usually SHA-1 or SHA-256
Seed--random

https://www.emc.com/collateral/white-papers/h11300-pkcs-1v2-2-rsa-cryptography-standard-wp.pdf

Students do not have to understand this figure, other than to know:

- Because OAEP uses a seed, encrypting the same plaintext will result in a different ciphertext every time.
- Even if the message is short, it expands it so that the Encoded Message EM fills the message block
- Attacks against short or repeated messages won't work
- It uses up some of your precious bits--if lHash, and seed are each 256 bits, you've lost 512 bits.
- However, if you like puzzles and really want to decode EM in the message above and recover the message M:
- XOR is its own inverse. If A xor B = C, then B xor C = A, or A xor C = B
- Consider the fields lHash and PS to be known to everyone, since they are part of the implementation. You can consider MGF to be a SHA hash, again known.
- You can recover the seed, and then the message M in this manner
    - Hash the maskedDB with the MGF (known), then XOR the result with the maskedSeed. This gives you the seed.
    - Hash the seed with the MGF, then XOR the result with the maskedDB. This gives you the DB
    - Since lHash, PS, and 01 are known, you can extract the message



RSA Math

- Basis of RSA is that the modulus (n) is the product of two primes
  - $n = p * q$ where p and q are prime
- Public key, e, and Private key, d, are inverses mod Λ
  - Ciphertext = Plaintext^e mod n
  - Decrypt, Ciphertext^d mod n = Plaintext^{ed} mod n = Plaintext
- Computing d as inverse of e mod n does not work
  - n is not prime, so $d \neq e^{-1} \bmod n$
- Must use $\Lambda = \text{lcm}((p - 1), (q - 1))$ as modulus
  - $d = e^{-1} \bmod \Lambda$

Everyone knows n, but only we know p and q because we chose them

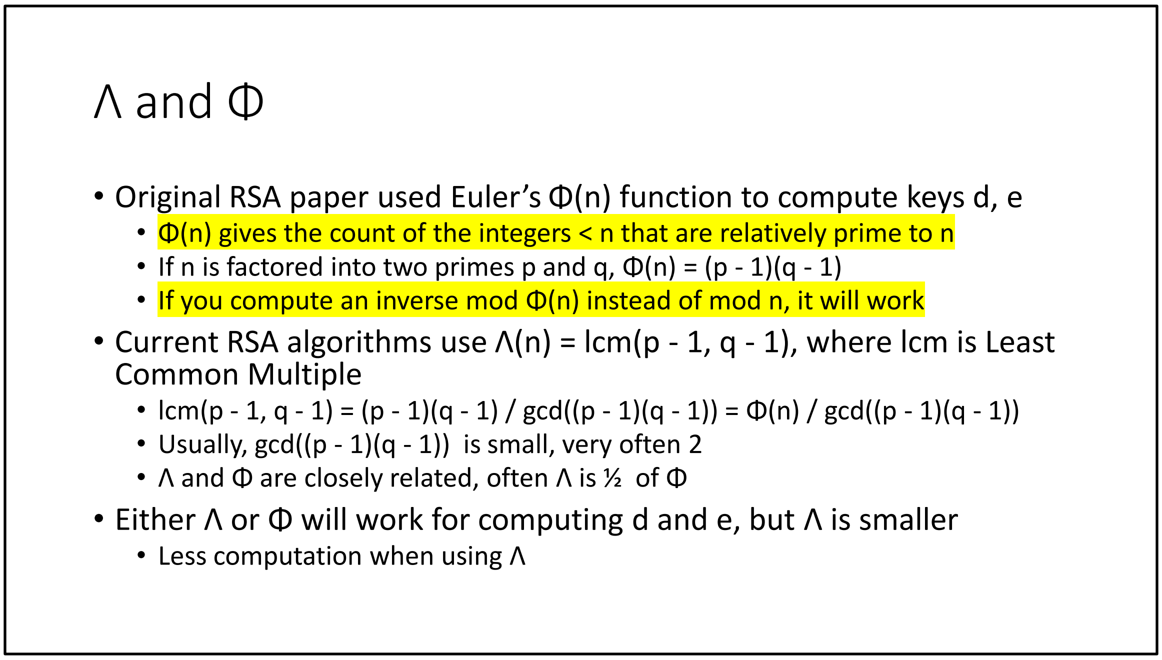
We pick the public key and compute the private key. Only we can compute the private key, because the computation requires that p and q are known. The private key is computed *modulo Λ(n) or Φ(n),* which depend on p and q.

Note that if you try to compute the private key as the inverse of the private key *modulo n*, it won't work. You can test this yourself.

In theory, the public and private keys are interchangeable until you give your public key to the world. In practice, there is good reason to pick your public key first and make it a small number. Most keys are used much more often for verifying signatures (use public key) than in generating signatures (private key.) Having a small public key saves time.

Small public keys have been shown to be secure, as small as 3, but NIST requires public keys to be 65537 or larger.

Small private keys are insecure, and should never be used. The key, d, should be as many bits long as n (the modulus.)



Λ and Φ

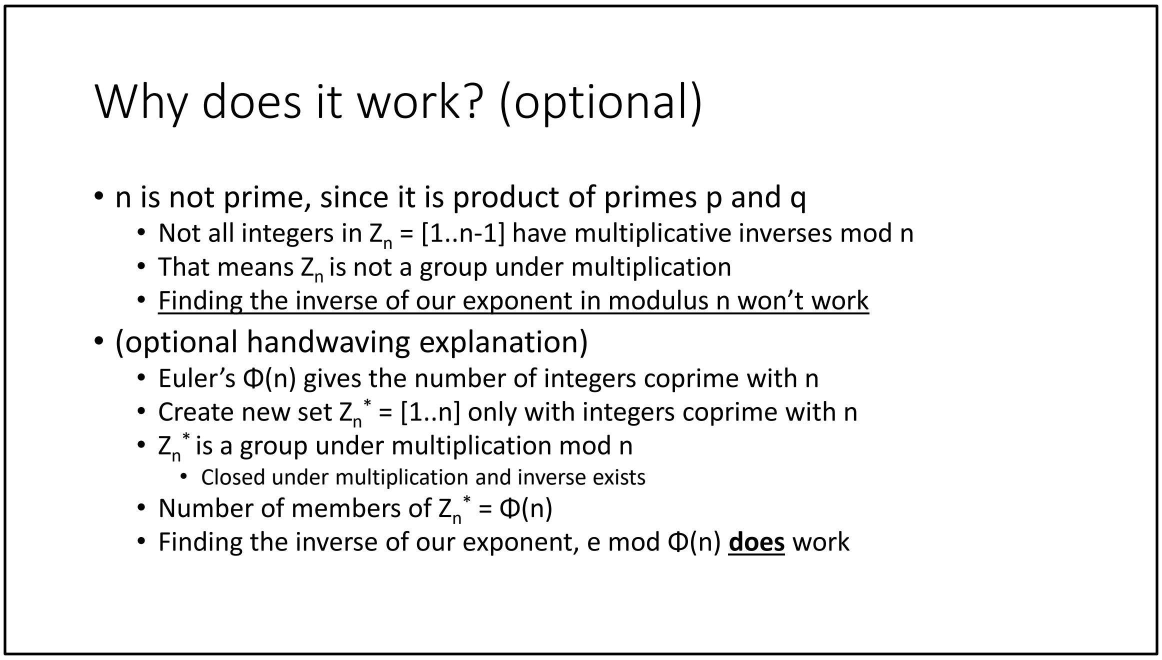
- Original RSA paper used Euler's Φ(n) function to compute keys d, e
  - Φ(n) gives the count of the integers < n that are relatively prime to n
  - If n is factored into two primes p and q, Φ(n) = (p - 1)(q - 1)
  - If you compute an inverse mod Φ(n) instead of mod n, it will work
- Current RSA algorithms use Λ(n) = lcm(p - 1, q - 1), where lcm is Least Common Multiple
  - lcm(p - 1, q - 1) = (p - 1)(q - 1) / gcd((p - 1)(q - 1)) = Φ(n) / gcd((p - 1)(q - 1))
  - Usually, gcd((p - 1)(q - 1)) is small, very often 2
  - Λ and Φ are closely related, often Λ is ½ of Φ
- Either Λ or Φ will work for computing d and e, but Λ is smaller
  - Less computation when using Λ

Many textbooks explain RSA encryption using Φ(n) to compute the private key as the inverse of the public key, modulo Φ(n), because that's the what the original paper used. There are several proofs that the system works when the inverse of the public key is computed in modulo Φ(n). The key point is that Φ(n) gives you the number of elements in a set where multiplicative inverse works. Multiplicative inverse mod n does not work, because n is not a prime number.

Modern implementations use the Least Common Multiple of p-1 and q-1 instead of just the product. The lcm is usually smaller than Φ(n), so the private key is smaller as well. Smaller keys mean less computation.

Euler's Φ(n) function
n factors into $p_1^{e1} * p_2^{e2} * \ldots$
$\Phi(n) = (p_1^{e1} - p_1^{e1-1})(p_2^{e2} - p_2^{e2-1})\ldots$

If n = 84, $n = 2^2 * 3^1 * 7^1$
$\Phi(n) = (2^2 - 2^1)(3^1 - 3^0)(7^1 - 7^0) = (4 - 2)(3 - 1)(7 - 1)$
$= 2*2*6 = 24$
There are 24 numbers between 1 and 83 that are relatively prime to 84



Why does it work? (optional)

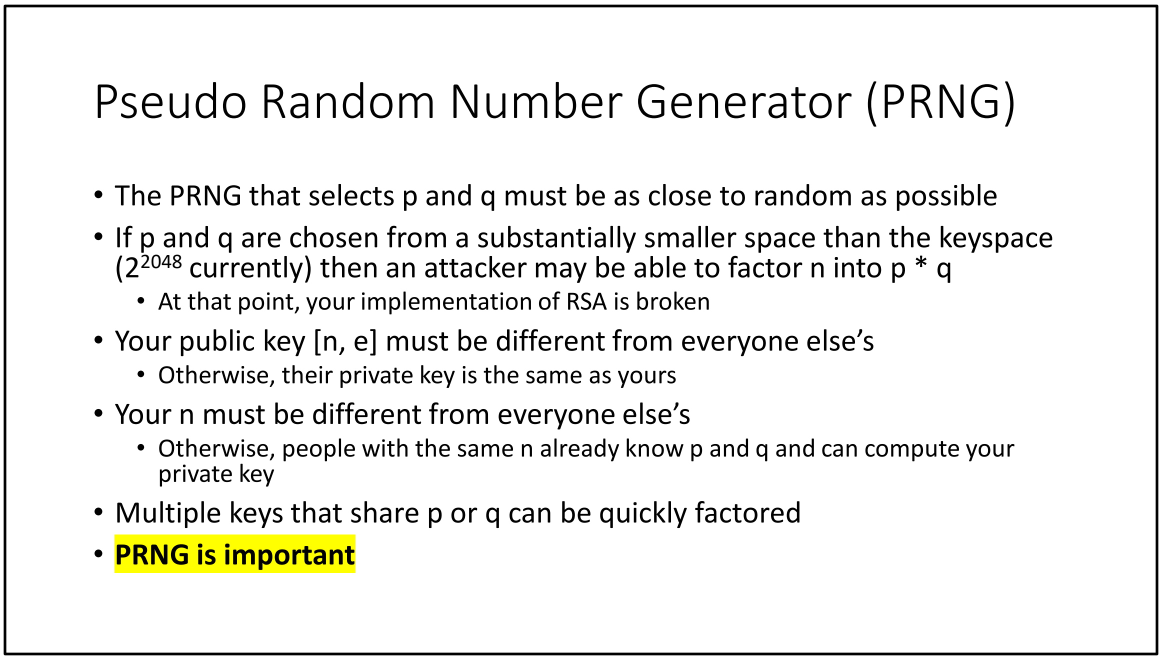
- n is not prime, since it is product of primes p and q
  - Not all integers in Z_n = [1..n-1] have multiplicative inverses mod n
  - That means Z_n is not a group under multiplication
  - <u>Finding the inverse of our exponent in modulus n won't work</u>
- (optional handwaving explanation)
  - Euler's Φ(n) gives the number of integers coprime with n
  - Create new set Z_n^* = [1..n] only with integers coprime with n
  - Z_n^* is a group under multiplication mod n
    - Closed under multiplication and inverse exists
  - Number of members of Z_n^* = Φ(n)
  - Finding the inverse of our exponent, e mod Φ(n) **<u>does</u>** work

There are several mathematical proofs that RSA encryption works, usually called proofs of correctness. They are available in the "Understanding Cryptography" and "Cryptography Engineering" texts, as well as on Wikipedia. However, none have helped me gain an intuitive understanding of why RSA works. What follows is a hand-waving non-proof that helps me understand it.

The concept of a group comes up in Diffie-Hellman encryption, but we have had a glimpse of it already with ciphers and multiplicative inverses. In the Affine cipher, we could only use keys (multiply by numbers) that were relatively prime to the modulus n (26 for the alphabet.)

Since n = p * q, and is not prime, any number in our set that is a multiple of p or q will not be relatively prime with n and will not have a multiplicative inverse. That means the set is not a field (or group with the operator as multiplication); the inverse and exponentiation (which is just repeated multiplication) *modulo n* will not work.

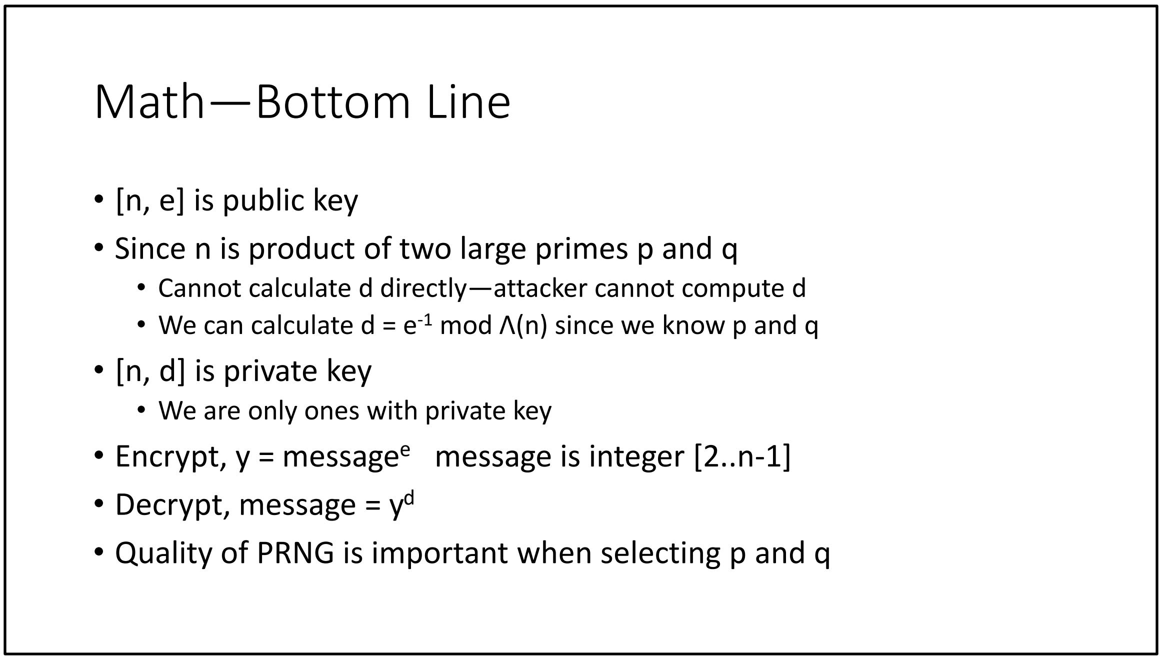
If we remove all the numbers without inverses from our set, what's left will be a group because every number will have a multiplicative inverse (You also have to prove the group is closed, but I'm handwaving past that.) There are (p-1)(q-1) numbers left when we remove multiples of p and q, and we called the group without the multiples Z_n^* . When we picked the public key e, we made sure it was relatively prime with $\Phi(n)$, which made sure e was in Z_n^*. That ensures that d will also be in Z_n^*.



Pseudo Random Number Generator (PRNG)

- The PRNG that selects p and q must be as close to random as possible
- If p and q are chosen from a substantially smaller space than the keyspace (2^{2048} currently) then an attacker may be able to factor n into p * q
  - At that point, your implementation of RSA is broken
- Your public key [n, e] must be different from everyone else's
  - Otherwise, their private key is the same as yours
- Your n must be different from everyone else's
  - Otherwise, people with the same n already know p and q and can compute your private key
- Multiple keys that share p or q can be quickly factored
- **PRNG is important**

If p and q are not truly random, and not chosen properly, security using those keys is weak. These two papers describe problems with insecure RSA keys that are presently in use.
https://eprint.iacr.org/2012/064.pdf "Ron was wrong, Whit is right"
https://factorable.net/weakkeys12.extended.pdf "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices"



Math—Bottom Line

- [n, e] is public key
- Since n is product of two large primes p and q
  - Cannot calculate d directly—attacker cannot compute d
  - We can calculate $d = e^{-1} \bmod \Lambda(n)$ since we know p and q
- [n, d] is private key
  - We are only ones with private key
- Encrypt, $y = \text{message}^e$ message is integer [2..n-1]
- Decrypt, $\text{message} = y^d$
- Quality of PRNG is important when selecting p and q

So, to recap…

The numbers p, q, and e are chosen. The modulus n is just p*q. The private key d is the inverse of e, and can only be computed if you know p or q.

If we choose p, q, and e wisely, and keep p and q secret, attackers cannot compute d and cannot decrypt our messages.

The encryption/decryption works because d and e are multiplicative inverses.
$y^d = x^{de} = x^1 = x$ (x is the message)

If we don't choose p, q, and e well, or don't use padding that includes a random seed, our encryption can easily be broken.