Course Notes for

CS 1501
Algorithm Implementation

By

John C. Ramirez

Department of Computer Science

University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- Recall from last lecture…
- In order for RSA to be useable

1) The keys must be able to be generated in a reasonable (polynomial) amount of time
2) The encryption and decryption must take a reasonable (polynomial) amount of time
3) Breaking the code must take an extremely large (exponential) amount of time

   - Let's look at these now

1) What things need to be done to create the keys and how long will they take?

- Long integer multiplication
  - We know this takes Theta(N^2), Theta($N^{1.58}$) or Theta(NlgNlglgN) depending on the algorithm
- Mod
  - Similar to multiplication (Theta(N^2))
- GCD and XGCD
  - Worst case $\approx$ N mods so ~N^3 worst case
- What is left?
- Random Prime Generation

Random Prime Generation

- There is no good (efficient) deterministic algorithm to calculate a large random prime number
- It turns out that the best (current) algorithm is a probabilistic one:

```
Generate random integer X
while (!isPrime(X))
     Generate random integer X
```

- As an alternative, we could instead make sure X is odd and then add 2 at each iteration, since we know all primes other than 2 itself are odd
  - The distribution of primes generated in this way is not as good (from a randomness point of view), but it may obtain a prime slightly faster than just picking a random integer each time

Random Prime Generation

- Runtime can be divided into two parts:
  1) How many iterations of the loop are necessary (or, how many numbers must be tried)?
  2) How long will it take to test the primality of a given number?
     - Overall run-time of this process is the product of 1) and 2)

1) Based on the distribution of primes within all integers, it is likely that a prime will be found within $\ln(2^N)$ random picks
   - This is linear in N, so it is likely that the loop will iterate N or fewer times

Random Prime Generation

2) This is much more difficult: How to determine if a number is prime or composite?

- Brute force algorithm: Try all possible factors
- Well not actually ALL need to be tried:
  - > From 2 up to square root of X
  - > But X is an N bit number, so it can have a value of up to 2^N
  - > This means that we will need to test up to $2^{N/2}$ factors, which will require an excessive amount of time for very large integers
  - > If mod takes Theta(N^2), our runtime is now Theta($N^2 2^{N/2}$) – very poor!
- Is there a better way? Let's use a probabilistic algorithm

Random Prime Generation

- **Miller-Rabin Witness algorithm**:

```
Do K times
   Test a (special) random value (a "witness")
   If it produces a special equality – return true
If no value produces the equality – return false
```

  - If true is ever returned – number is definitely NOT prime, since a factor was found
  - If false is returned (after all K times) probability that the number is NOT prime is 2^{-K}
    - If we choose K to be reasonably large, there is very little chance of an incorrect answer
    - Ex: Let K = 100
      - If false is returned the chance of a mistake (i.e. that the number is actually composite) is 2^{-100}

Random Prime Generation

- Java BigInteger uses this algorithm
  - See: https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/math/BigInteger.html
- Run-time?
  - Each of the K tests requires up to N multiplications for N-bit integers
  - Assuming GS algorithm for multiplication
    - > KN^3 in the worst case
- Recall that this is just the run-time to check if one number is prime (item 2)
- To get the total run-time we need to multiply by the number of numbers expected (item 1) which leads to

  (~N numbers) * (KN^3 to test) = KN^4 total

Random Prime Generation

- This is not outstanding, since N^4 grows quickly, but in reality the algorithm will usually run much more quickly:
  - A better multiplication alg. (Karatsuba or FFT) can be used
  - A prime may be found in fewer than N tries
  - The Miller-Rabin test often finds a factor quickly for the composite numbers, not requiring the full K tests or the full N mults in a given test
    - This will likely be the biggest practical speed-up
- Furthermore, we do this only to generate the keys – not for the encryption and decryption, so some extra overhead can be tolerated

2) How long does it take to use RSA encryption and decryption?

- Power-mod operation – raising a very large integer to a very large integer power mod a very large integer
  - As we discussed, this can be done using the divide and conquer approach
  - Requires Theta(N) multiplications, for a total of Theta(N^3) assuming Gradeschool is used
- This is the dominant time for both encryption and decryption
  - Not great, but feasible

3) How easily can RSA be broken?

- Recall that we have 3 values: E, D and N
  - Most obvious way of breaking RSA is to factor N
    - Factoring N gives us X and Y (since N = XY)
    - But PHI = (X-1)(Y-1)
      - Once they know PHI, cryptanalysts can determine D in the same way we generated D and poof!
    - So is it feasible to factor N?
      - There is no known polynomial-time factoring algorithm
      - Thus, it will require exponential time to factor N
      - But, with fast computers, it can be done
        - > Team factored a 768-bit or 232-decimal digit RSA number in 2009 – but computation was considerable
        - > See: http://en.wikipedia.org/wiki/RSA_numbers

Time to Break RSA

- However, if we make N large enough, we can be pretty sure that factoring will not be possible in a reasonable amount of time
  - 1024 bit keys were thought to be good but now are considered inadequate
  - 2048 bits may even be obsolete in the next few decades
  - A move to 3072 bits may be necessary
  - An important thing to remember, is that since the factoring time is still exponential, doubling the size of the key will increase the time to factor it by many thousands/millions/billions of times

Time to Break RSA

- Yet no one has proven that factoring requires exponential time (although it is widely believed). If someone finds a polynomial factoring algorithm, RSA would be useless!
- But we should still be careful in choosing keys, and in how long we keep a key
  - Some specific keys can be broken more easily

- Quantum computers also pose a threat to RSA security
  - See: https://www.technologyreview.com/2019/05/30/65724/how-a-quantum-computer-could-break-2048-bit-rsa-encryption-in-8-hours/
    - Not here yet but maybe sooner than we think…

Time to Break RSA

- Indirectly breaking a code:
  - If we just want to break one message, and not the keys, there are simple tricks that can be tried
    - For example, say you know all of the possible messages that may be sent (and this value is not too large)
      - Ex: All possible credit card numbers
      - Encrypt each one and compare result to the ciphertext
      - The one that matches is the message sent
    - We can combat this by padding messages with extra bits (random is best) before sending them
- Other techniques exist as well (ex: timing)
- Generally, if we are careful and use large keys, RSA is quite secure (for now)

RSA Applications

- Applications
  - For regular data encryption, RSA is relatively slow (~N^3) compared to block ciphers (N)
  - But block ciphers have key-distribution problem
    - How can we get the "best of both"?
  - RSA (Digital) Envelope
    - Idea:
      - Use RSA to transmit symmetric cipher keys between the two communicating parties
      - The use symmetric cipher to do actual communication
      - Since RSA is used only for key transmission, the relative slowness will not slow down the communication

RSA Applications

- In other words:
  - Sender encrypts block cipher key using receiver's public RSA key
  - Sender encrypts message using block cipher
  - Sender sends whole package to receiver

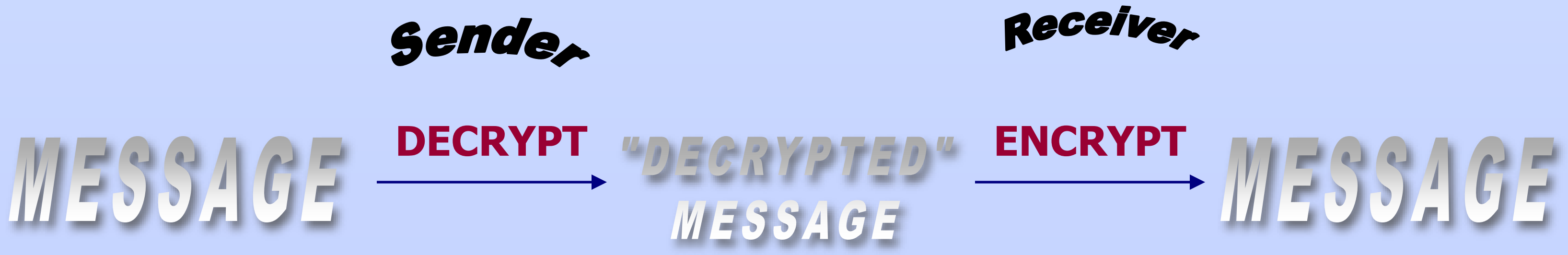
  - Receiver decrypts block cipher key using his / her private RSA key
  - Receiver uses block cipher key to decrypt rest of message
- Note that this can lead to indefinite two-way communication once both parties know the block cipher key
  - Classic solution to key distribution problem

RSA Applications

- RSA Envelopes are often used implicitly through software
  - Ex: SSL / TLS
  - When a client connects to a server, an initial “handshaking” process determines the keys to be used
    - > Client sends list of block encryption schemes it supports
    - > Server selects “best one” that it also supports
    - > Server sends its choice and its public (RSA) key to client
    - > Client generates a random number, encrypts it with the server’s RSA key and sends it to the server (or sends encrypted key itself)
    - > Server and client both use random number to generate a block key in the chosen scheme
  - Data is then transferred back and forth using the block encryption scheme
  - This is just a sketch of TLS – for details see:
    - > https://en.wikipedia.org/wiki/Transport_Layer_Security

Digital Signatures

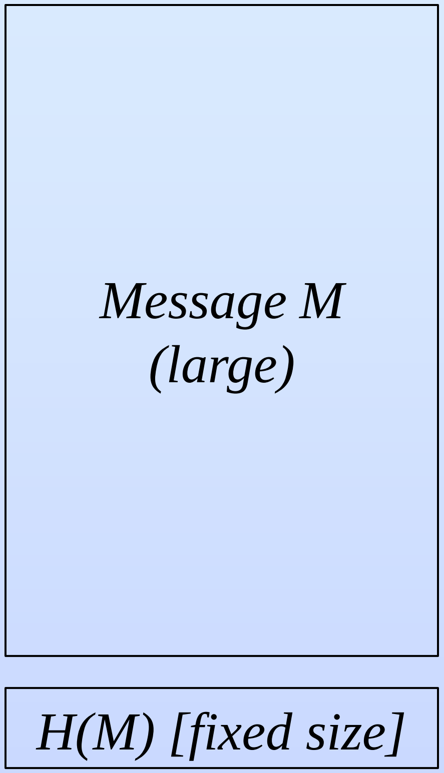
- Notice that E and D are inverses
- It doesn't matter mathematically which we do first
- If I DECRYPT my message (with signature appended) and send it, anyone who then ENCRYPTS it can see the original message and verify authenticity
  - Since my public key reproduced plaintext, my private key must have been used to "decrypt"
  - However, there are a few issues to resolve



RSA Applications

- Note in this case that we are NOT trying to keep anyone from reading the message
  - Everyone can get the public key and can thus successfully "encrypt" the message
- What about the issues to resolve?
  1) As with regular encryption, RSA used in this way is a slow (still Theta(N^3))
  - Can we avoid decrypting and encrypting the entire message?
    > Yes, instead of decrypting the whole message, the sender processes the message using a cryptographic hashing technique (ex: SHA-256). This produces a "signature" for the message and the signature is what we actually "decrypt" using the private RSA key. Then the message and signature are sent to the receiver.

Cryptographic Hash



- What is a cryptographic hash?
  - Recall hashing from earlier in the term
    - H(X) should incorporate the entire key
    - Maps key values into a fixed address space (the hash table)
    - Goal is to allow fast lookup of keys with relatively few collisions
  - With cryptographic hashing the process is similar
    - H(M) will be a function of the entire message
    - Will map the message into a fixed size result (ex: 128 or 256 bits)
    - However we are not using an actual hash table in this case
    - The goals here are different

Cryptographic Hash

- With a cryptographic hash our goal is to detect any changes to the message via a change in the hash value
- Ex: Suppose H(M) = K
  - If M is altered in some way to M' we would like

  H(M') != H(M)
  - Note that in hashing we can always have collisions
  - This is possible in cryptographic hashes as well since our "table size" relatively small (a 128 or 256 bit value) and our "key space" is (all possible values of our original message)
  - **HOWEVER**, an arbitrary change to M will clearly not produce a collision
  - Assume a cryptanalyst wants to change M and have it not detected by the signature

Cryptographic Hash

- Would have to come up with a very specific alteration to the message, M', such that
  - > H(M') = H(M)
  - > M' is a reasonable message (i.e. not gibberish) that receiver would accept
- Doing this would require a way to somehow effectively generate messages that will "collide" with M
- A good cryptographic hash function should make this very difficult (i.e. computationally infeasible)
- Historically, some cryptographic hashes were thought to be good but are now considered vulnerable (ex: MD5, SHA1)
- However, there are good ones available (ex: SHA256)
- See: http://en.wikipedia.org/wiki/Cryptographic_hash_function

RSA Applications

- [Now back to our process (from slide 20)]
  - > The receiver receives (message + signature) from sender
  - > The receiver "encrypts" the decrypted signature (using sender's public RSA key) to restore the original signature
  - > The receiver then runs the same cryptographic hash on the received message, and compares result with the signature. If they match, the message is valid – otherwise it has been tampered with

- This may be clearer if we do a simple example…

RSA Applications

- Assume Sender wants to send a message, M to Receiver (not encrypting, just verifying authenticity)
- Sender:
  - Calculate H(M) (signature, using cryptographic hash)
  - Calculate D(H(M)) (with sender's private RSA key, D)
  - Send D(H(M)) + M to receiver
- Receiver:
  - Calculate E(D(H(M))) (with sender's public RSA key, E)
  - Calculate H(M) (using received M)
  - If E(D(H(M))) == H(M)
    - Everything is ok – message is authentic
  - Otherwise, something is wrong, so reject the message

RSA Applications

2) How do we know that the key used actually belonged to the sender?
   - Ex: Joe Schmoe gets fired by his boss and sends a digitally signed message stating that his boss is a total weasel. Before getting the message, his boss decides to rehire Joe, but then sees the message. Joe says "well, I didn't send that message – I haven't used that RSA key for years – it must be from someone pretending to be me"
   - How does the boss know if Joe is lying?
   - We need authentication of keys (ex: Verisign)
     - Some authority "vouches" for the keys, in effect establishing a reliable connection between keys and owners

- See: Wikipedia article