Introduction to Cryptography
Lecture 11

Benny Pinkas

Certification Authorities (CA)

- A method to bootstrap trust
  - Start by trusting a single party and knowing its public key
  - Use this to establish trust with other parties (and associate them with public keys)

- The Certificate Authority (CA) is trusted party.
  - All users have a copy of the public key of the CA
  - The CA signs Alice's digital certificate. A simplified certificate is of the form *(Alice, Alice's public key)*.

Certification Authorities (CA)

- News about CAs used for MiTM attacks.

Revocation

- Revocation is a key component of PKI
  - Each certificate has an expiry date
  - But certificates might get stolen, employees might leave companies, etc.
  - Certificates might therefore need to be revoked before their expiry date
  - New problem: before using a certificate we must verify that it has not been revoked
    - Often the most costly aspect of running a large scale public key infrastructure (PKI)
    - How can this be done efficiently?

Certificate Revocation Lists (CRLs)

- A revocation agency (RA) issues a list of revoked certificates (i.e., “bad” certificates)
  - The list is updated and published regularly (e.g. daily)
  - Before trusting a certificate, users must consult the most recent CRL in addition to checking the expiry date.
- Advantages: simple.
- Drawbacks:
  - Scalability. CRLs can be huge. There is no short proof that a certificate is valid.
  - There is a vulnerability windows between a compromise of certificate and the next publication of a CRL.
  - Need a reliable way of distributing CRLs.
- Improving scalability using “delta CRLs”: a CRL that only lists certificates which were revoked since the issuance of a specific, previously issued CRL.

Explicit revocation: OCSP

- OCSP (Online Certificate Status Protocol)
  - RFC 2560, June 1999.
- OCSP can be used in place, or in addition, to CRLs
- Clients send a request for certificate status information.
  - An OCSP server sends back a response of "current", "expired," or "unknown“.
  - The response is signed (by the CA, or a Trusted Responder, or an Authorized Responder certified by the CA).
- Provides instantaneous status of certificates
  - Overcomes the chief limitation of CRL: the fact that updates must be frequently downloaded and parsed by clients to keep the list current

Certificate Revocation System (CRS)

- Certificate Revocation System (Micali'96)
- *Puts the burden of proof on the certificate holder* (who must prove that the certificate is still valid).

- In theory, we could limit the lifetime of certificates to a single day, and require the certificate holder to ask for a new certificate every day.
  - This would result in a high overhead at the CA

Certificate Revocation System (CRS)

- It is possible to reduce the overhead of the CA by using a hash chain
  - The certificate includes $Y_{365} = f^{365}(Y_0)$. This value is part of the information signed by the CA. f is one-way.
  - On day d,
    - If the certificate is valid, then $Y_{365-d} = f^{365-d}(Y_0)$ is sent by the CA to the certificate holder or to a directory.
    - The certificate receiver uses the daily value ($f^{365-d}(Y_0)$) to verify that the certificate is still valid. (how?)
- Advantage: A short, individual, proof per certificate.
- Disadvantage: Daily overhead, even when a cert is valid.

CA's work

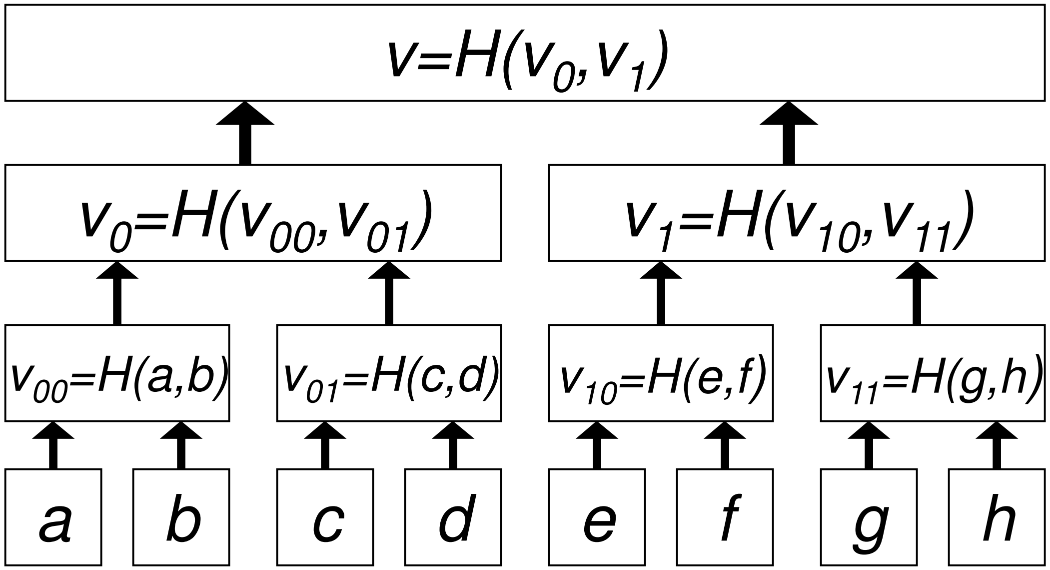
- How can the server can compute $f^{i}(Y_0)$
- There are two straightforward methods
  - Storing all n values
  - Storing Y_0 and computing $f^{i}(Y_0)$ on the fly.

- Another option is to store *sqrt(n)* intermediate points and do *sqrt(n)* work per computation of each $f^{i}(Y_0)$

- There are also more advanced methods requiring *log(n)* storage and *O(1)* amortized work per computation

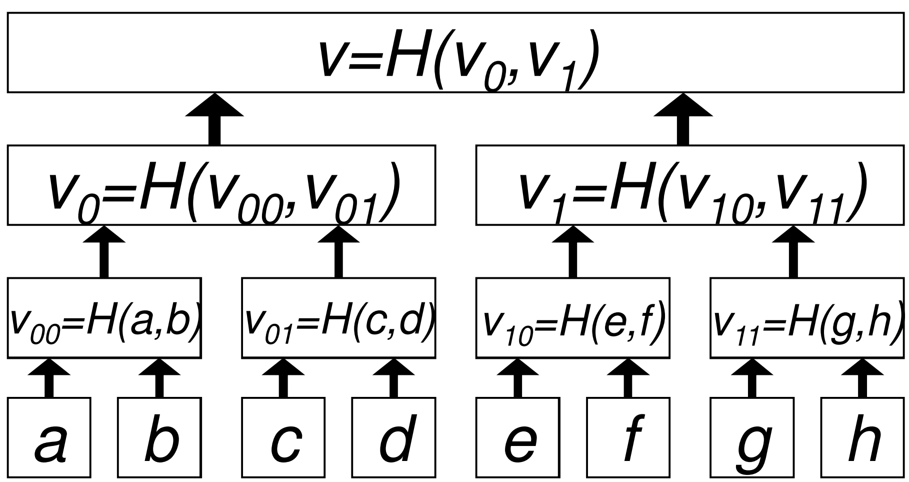
Merkle Hash Tree (will be useful later)

- A method of committing to (by hashing together) n values, $x_1,\ldots,x_n$, such that
  - The result is a single hash value
  - For any x_i, it is possible to prove that it appeared in the original list, using a proof of length O(log n).



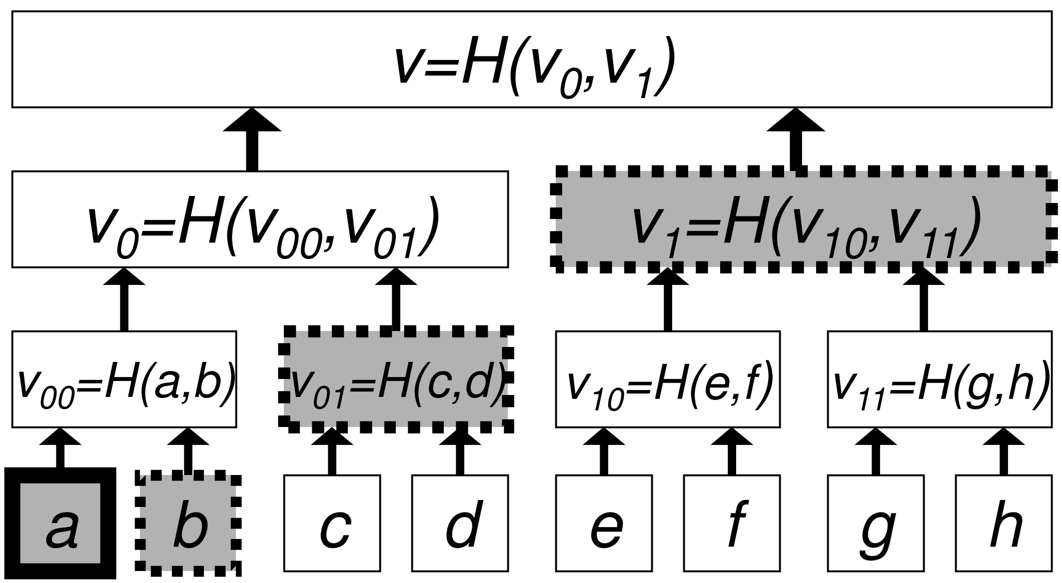
Merkle Hash Tree

- H is a collision intractable hash function
- Any change to a leaf results in a change to the root
- To sign the set of values it is sufficient to sign the root (a single signature instead of *n*).
- How do we verify that an element appeared in the signed set?



Verifying that *a* appears in the signed set

- Provide *a*'s leaf, and the siblings of the nodes in the path from *a* to the root. (O(log n) values)
- The verifier can use *H* to compute the values of the nodes in the path from the leaf to the root.
- It then compares the computed root to the signed value.



Using hash trees to improve the overhead of CRS

- Originally (for a year long certificate)
  - the certificate includes $f^{365}(Y_0)$
  - On day *d*, certificate holder obtains $f^{365-d}(Y_0)$
  - The certificate receiver computes $f^{365}(Y_0)$ from $f^{365-d}(Y_0)$ by invoking *f()* *d* times.
- Slight improvement:
  - The CA assigns a different leaf for every day, constructs a hash tree, and signs the root.
  - On day *d*, it releases node *d* and the siblings of the path from it to the root.
  - This is the proof that the certificate is valid on day *d*
  - The overhead of verification is O(log 365).

Certificate Revocation Tree (CRT) [Kocher]

- (A different usage of a hash tree)
- A CRT is a hash tree with leaves corresponding to statements about ranges of certificates
  - Statements describe regions of certificate ids, in which only the smallest id is revoked.
    - For example, a leaf might read: “if 100 ≤ id <234, then cert is revoked iff id=100”.
  - Each certificate matches exactly one statement.
  - The statements are the leaves of a signed hash tree, ordered according to the ranges of certificate values.
  - To examine the state of a certificate we retrieve the statement for the corresponding region.
  - A single hash tree is used for all certs.

Certificate Revocation Tree (CRT)

- Preferred operation mode:
  - Every day the CA constructs an updated tree.
  - The CA signs a statement including the root of the tree and the date.
  - It is Alice's responsibility to retrieve the leaf which shows that her certificate is valid, the route from this leaf to the root, and the CA's signature of the root.
  - To prove the validity of her cert, Alice sends this information.
  - The receiver verifies the value in the leaf, the route to the tree, and the signature.
- Advantage:
  - a short proof for the status of a certificate.
  - The CA does not have to handle individual requests.
- Drawback: the entire hash tree must be updated daily.

SSL / TLS

SSL/TLS

- General structure of secure HTTP connections
  - To connect to a secure web site using SSL or TLS, we send an https:// command
  - The web site sends back a public key[1], and a certificate.
  - Our browser
    - Checks that the certificate belongs to the url we're visiting
    - Checks the expiration date
    - Checks that the certificate is signed by a CA whose public key is known to the browser
    - Checks the signature
    - If everything is fine, it chooses a session key and sends it to the server encrypted with RSA using the server's public key

(1) This is a very simplified version of the actual protocol.

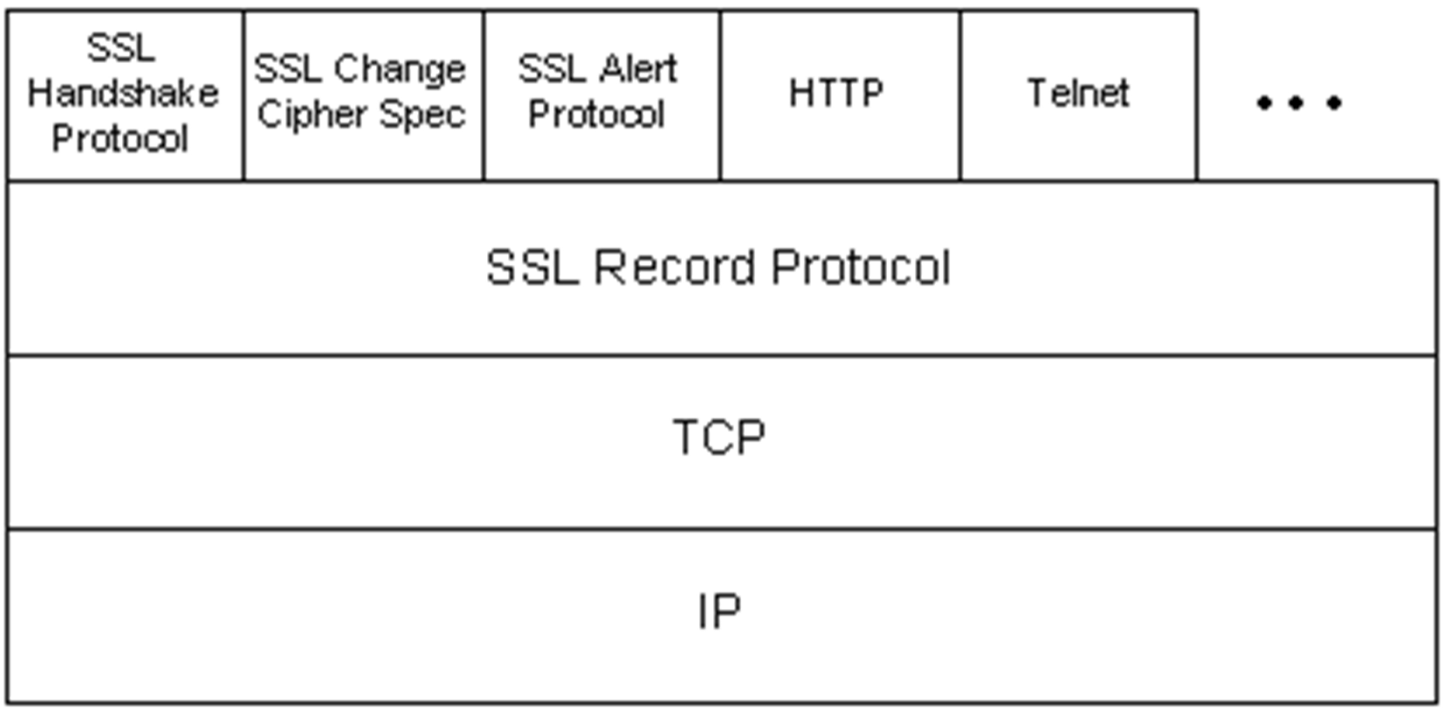
SSL/TLS

- SSL (Secure Sockets Layer)
  - SSL v2
    - Released in 1995 with Netscape 1.1
    - A flaw found in the key generation algorithm
  - SSL v3
    - Improved, released in 1996
    - Public design process
- TLS (Transport Layer Security)
  - IETF standard, RFC 2246

- Common browsers support all these protocols

SSL Protocol Stack

- SSL/TLS operates over TCP, which ensures reliable transport.
- Supports any application protocol (usually used with http).



SSL/TLS Overview

- Handshake Protocol - establishes a session
  - Agreement on algorithms and security parameters
  - Identity authentication
  - Agreement on a key
  - Report error conditions to each other
- Record Protocol - Secures the transferred data
  - Message encryption and authentication
- Alert Protocol – Error notification (including “fatal” errors).
- Change Cipher Protocol – Activates the pending crypto suite

Simplified SSL Handshake

Client

Server

I want to talk, ciphers I support, R_C →

← Certificate (PK_{Server}), cipher I choose, R_S

$\{S\}_{PKserver}$, {keyed hash of handshake message} →

← {keyed hash of handshake message}

compute $K = f(S, R_C, R_S)$ (client)

compute $K = f(S, R_C, R_S)$ (server)

Data protected by keys derived from K ← — — — →

A typical run of a TLS protocol

- $C \Rightarrow S$
  - ClientHello.protocol.version = “TLS version 1.0”
  - ClientHello.random = T_C, N_C
  - ClientHello.session_id = “NULL”
  - ClientHello.crypto_suite = “RSA: encryption.SHA-1:HMAC”
  - ClientHello.compression_method = “NULL”
- $S \Rightarrow C$
  - ServerHello.protocol.version = “TLS version 1.0”
  - ServerHello.random = T_S, N_S
  - ServerHello.session_id = “1234”
  - ServerHello.crypto_suite = “RSA: encryption.SHA-1:HMAC”
  - ServerHello.compression_method = “NULL”
  - ServerCertificate = pointer to server’s certificate
  - ServerHelloDone

Some additional issues

- More on S $\Rightarrow$ C
  - The ServerHello message can also contain Certificate Request Message
  - I.e., server may request client to send its certificate
  - Two fields: certificate type and acceptable CAs

- Negotiating crypto suites
  - The crypto suite defines the encryption and authentication algorithms and the key lengths to be used.
  - ~30 predefined standard crypto suites
  - Selection (SSL v3): Client proposes a set of suites. Server selects one.

Key generation

- Key computation:
  - The key is generated in two steps:
  - *pre-master secret S* is exchanged during handshake
  - *master secret K* is a 48 byte value calculated using pre-master secret and the random nonces

- Session vs. Connection: a *session* is relatively long lived. Multiple TCP *connections* can be supported under the same SSL/TSL connection.
- For each connection: 6 keys are generated from the master secret *K* and from the nonces. (For each direction: encryption key, authentication key, IV.)

TLS Record Protocol

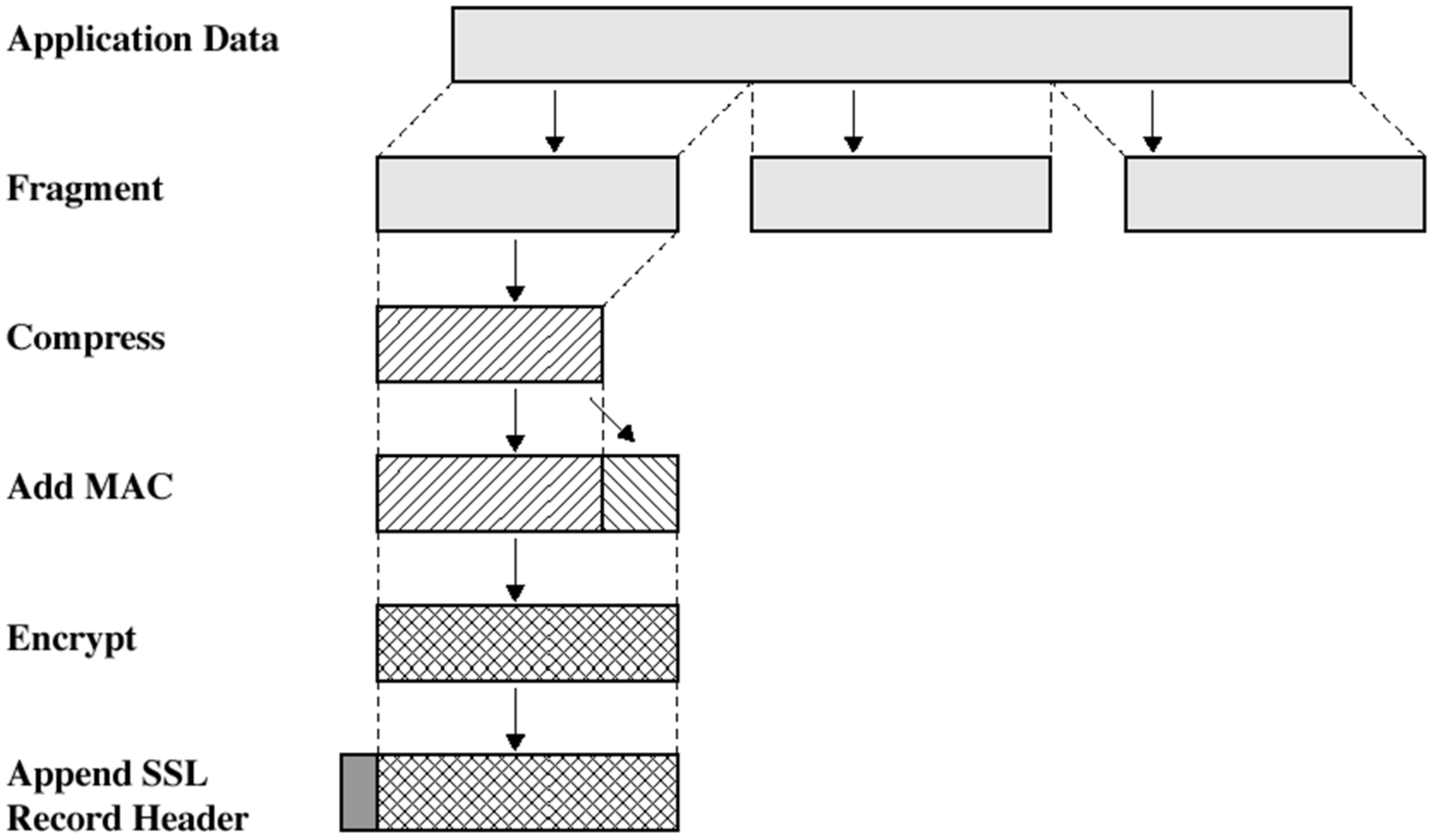


Figure 17.3 SSL Record Protocol Operation

Some practical issues in number theory

Primality testing

- Why do we need primality testing?
  - Essentially all public key cryptographic algorithms use large prime numbers
  - We therefore need an algorithm for prime number generation
  - Suppose we have an algorithm "PrimalityTest" with a binary output.
  - We can generate random primes as follows

```
GeneratePrime(a,b)

  1. Choose random number x ∈ [a,b]

  2. If PrimalityTest(x) then output "x is
  prime"; otherwise goto line 1.
```

Density of prime numbers

- How long will GeneratePrime run?
- Let $\pi(n)$ specify number of primes $\leq n$.

- Prime number theorem:
  - $\pi(n)$ goes to $n / \ln n$ as n goes to infinity.
- Pretty accurate even for small n (e.g. for $n=2^{30}$ it is off by 6%).

- Corollary: a random number in [1,n] is prime with probability 1/ln n. (e.g. for $n=2^{512}$, probability is *1/355*).
  - The GeneratePrime algorithm is expected to take ln n rounds.
  - If we skip even numbers, we cut running time by ½.

Primality testing

- Primality testing is a decision problem: "is x prime or composite?"
- Different than the search problem "find all prime factors of x" ("factor x").
- In this case, the decision problem has an efficient solution while the search problem does not.

- First algorithm for primality testing: Trial division
  - Try to divide x by every prime integer smaller than $\sqrt{x}$ (sqrt(x)).
  - Infeasible for large x.

Fermat's test

- Fermat's theorem: if p is prime then for all $1 \le a < p$ it holds that $a^{p-1} = 1 \bmod p$.

- If we can find an a s.t $a^{x-1} \neq 1 \bmod x$, then x is surely composite.
  - Surprisingly, the converse is almost always true, and for a large percentage of the choices of a.
  - Suppose we check only for $a=2$.
    - `If` 2^{x-1} `!= 1 mod x`
      - `Then return COMPOSITE     /for sure`
      - `Otherwise, return PRIME   /we hope`

  - How accurate is this program?

Fermat's test

- Surprisingly, this test is almost always right
  - Wrong for only 22 values of x smaller than 100,000
  - Probability of error goes down to 0 as x grows
    - For |x|=512 bits, probability of error is $< 10^{-20} \approx 2^{-66}$
    - For |x|=1024 bits, probability of error is $< 10^{-41} \approx 2^{-136}$
- The test is therefore sufficient for randomly chosen candidate primes

- But we need a better test if x is not chosen at random
- Cannot eliminate errors by checking for bases ≠ 2
  - x is a Charmichael number if it is composite, but $a^{x-1} = 1 \bmod x$ for all $1 \leq a < x$.
  - There are infinitely many Charmichael numbers
  - But they are very rare

Miller-Rabin test

Works for all numbers (even Charmichael numbers).

- Checks several randomly chosen bases *a*
- If it finds out that $a^{x-1} = 1$ mod *x*, it checks whether the process found a nontrivial root of 1 $(\neq 1,-1)$. If so, it outputs COMPOSITE.

The Miller-Rabin test:

1. Write $x-1=2^c r$ for an odd r. set comp=0.
2. For i=1 to T
   - Pick random a ∈ [1,x-1]. If gcd(a,x)> 1 set comp=1.
   - Compute $y_0=a^r$ mod x, $y_i=(y_{i-1})^2$ mod x for $i=1..c$. If $y_c\neq 1$, or $\exists i$, $y_i=1$, $y_{i-1}\neq\pm 1$, set comp=1.
3. If comp=1 return COMPOSITE, else PRIME.

Miller-Rabin test

- Possible values for the sequence $y_0=a^r$, $y_1=a^{2r}$… $y_c=a^{x-1}$.
  - <…,d>, where $d\neq 1$, decide COMPOSITE.
  - <1,1,…,1>, decide PRIME.
  - <..,-1,1,..,1>, decide PRIME.
  - <…,d,1,…,1>, where $d\neq\pm 1$, decide COMPOSITE.
  - For a composite number *x*, we denote a base *a* as a non-witness if it results in the output being "PRIME".
- Lemma: if x is an odd composite number then the number of non-witnesses is at most $x/4$.
- Therefore, for any odd integer *x*, *T* trials give the wrong answer with probability < $(1/4)^T$.

Breaking News (some years ago)

- Primes $\in$ P
  - Agrawal, Kayal, Saxena (2004)

Integer factorization

- The RSA and Rabin cryptosystems use a modulus N and are insecure if it is possible to factor N.
- Factorization: given N find all prime factors of N.
- Factoring is the search problem corresponding to the primality testing decision problem.
  - Primality testing is easy
  - What about factoring?

Pollard's Rho method

- Factoring N
- Trivial algorithm: trial division by all integers < $N^{1/2}$.
- Pollard's rho method:
  - $O(N^{1/4})$ computation.
  - O(1) memory.
  - A heuristic algorithm.

Pollard's rho method

```
1. i=1; x1∈[1,n-1]; y=x1;
2. i = i+1.
3. xi = ((xi-1)^2 - 1) mod n.
4. d = gcd(y-xi, n)
5. If d>1 then output d, and stop.
6. If i is a power of 2, then y=xi
7. Goto line 2.
```

Always a factor of n

- x_i is a series of numbers in *0..n-1*.
- y takes the values of $x_1, x_2, x_4, x_8, \ldots, x_{2^i}, \ldots$
- If $(y-x_i) = 0$ mod p, then most likely gcd$(y-x_i,n)=p$.

Pollard's rho method

- The running time is not guaranteed, but is expected to be sqrt(p) ≤ $n^{1/4}$.
- The sequence x_i is in 1..n.
  - x_i depends only on x_{i-1} $(x_i = ((x_{i-1})^2 - 1) \bmod n)$
  - The sequence is shaped like the letter Rho.
  - Assume that $f_n(x)=x^2-1 \bmod n$ behaves like a random function. Then the tail and the circle are about sqrt(n) long.
- Let $x'_i = x_i \bmod p$, where p factors n.
- $x'_{i+1} = x_{i+1} \bmod p = (x_i^2-1 \bmod n) \bmod p = x_i^2-1 \bmod p = (x_i')^2-1 \bmod p$
- The sequence x'_i therefore follows x_i, but is in $0..p-1$. Therefore, its tail and circle are about sqrt(p) long.

Pollard's rho method

- The sequence x'_i:
  - Let t be the first repeated value in x'_i
  - Let u be the length of the cycle
  - $\forall i \quad x'_{t+i} = x'_{t+i+u} \mod p$
  - Therefore $x_{t+i} = x_{t+i+u} \mod p$
  - $gcd(x_{t+i} - x_{t+i+u}, n) = cp$.
- Once the algorithm saves $y=x_j$ for $j>t$, it is on the circle. If the circle length u is smaller than j, the algorithm computes $gcd(x_{j+u}-x_j, n)$ and factors n.
- The algorithm fails if
  - The cycle and tail are long $\Rightarrow$ running time is slow.
  - The cycle and tail are of the same length for both p and q.

Modern factoring algorithms

- The number-theoretic running time function $L_n(a,c)$

$$L_n(a,c) = e^{c(\ln n)^a (\ln\ln n)^{1-a}}$$

  - For a=0, the running time is polynomial in ln(n).
  - For a=1, the running time is exponential in ln(n).
  - For 0<a<1, the running time is subexponential.

- Factoring algorithms
  - Quadratic field sieve: $L_n(1/2, 1)$
  - General number field sieve: $L_n(1/3, 1.9323)$
  - Elliptic curve method $L_p(1/2, 1.41)$ (preferable only if p<<sqrt(n))

Modulus size recommendations

- Factoring algorithms are run on massively distributed networks of computers (running in their idle time).
- RSA published a list of factoring challenges.
- A 512 bit challenge was factored in 1999.
- The largest factored number $n=pq$.
  - 768 bits (RSA-768)
  - Factored on January 7, 2010 using the NFS

- Typical current choices:
  - At least 1024-bit RSA moduli should be used
  - For better security, longer RSA moduli are used
  - For more sensitive applications, key lengths of 2048 bits (or higher) are used

RSA with a modulus with more factors

- The best factoring algorithms:
  - General number field sieve (NFS): $L_n(1/3, 1.9323)$
  - Elliptic curve method $L_p(1/2, 1.41)$
- If n=pq, where |p|=|q|, then the NFS is faster.
  - Common parameters: |p|=|q|=512 bits
  - Factoring using the NFS is infeasible, but more likely than factoring using the elliptic curve method.
- How about using N=pqr, where |p|=|q|=|r|=512?
  - The factors are of the same length, so factoring using the elliptic curve method is still infeasible. ☺
  - The NFS method has to work on a larger modulus ☺
  - Decryption time is slower (but not by much). ☹

RSA for paranoids

- Suppose $N=pq$, $|p|=500$ bits, $|q|=4500$ bits.
- Factoring is extremely hard.
- Decryption is also very slow. (Encryption is done using a short exponent, so it is pretty efficient.)

- However, in most applications RSA is used to transfer session keys, which are rather short.
- Assume message length is < 500 bits.
  - In the decryption process, it is only required to decrypt the message modulo p. (As, or more, efficient, as a 1024 bit n.)
  - Encryption must use a slightly longer *e*. Say, e=20.

Discrete log algorithms

- Input: *(g,y)* in a finite group G. Output: x s.t. $g^x = y$ in G.
- Generic vs. special purpose algorithms: generic algorithms do not exploit the representation of group elements.

- Algorithms
  - Baby-step giant-step: Generic. |G| can be unknown. Sqrt(|G|) running time and memory.
  - Pollard's rho method: Generic. |G| must be known. Sqrt(|G|) running time and O(1) memory.
  - No generic algorithm can do better than O(sqrt(q)), where q is the largest prime factor of |G|
  - Pohlig-Hellman: Generic. |G| and its factorization must be known. O(sqrt(q) ln q), where q is largest prime factor of |G|.
  - Therefore for Z^*_p, p-1 must have a large prime factor.
  - Index calculus algorithm for Z^*_p: L(1/2, c)
  - Number field size for Z^*_p: L(1/3, 1.923)

Elliptic Curves

- The best discrete log algorithm which works even if |G| can be unknown is the baby-step giant-step algorithm.
  - Sqrt(|G|) running time and memory.
- Other (more efficient) algorithms must know |G|.
  - In Z_p^* we know that | Z_p^* |=p-1.

- Elliptic curves are groups G where
  - The Diffie-Hellman assumption is assumed to hold, and therefore we can run DH an ElGamal encryption/sigs.
  - |G| is unknown and therefore the best discrete log algorithm us pretty slow
  - It is therefore believed that a small Elliptic Curve group is as secure as larger Z_p^* group.
  - Smaller group -> smaller keys and more efficient operations.

Baby-step giant-step DL algorithm

- Let t=sqrt(|G|).
- x can be represented as $x=ut-v$, where u,v < sqrt(|G|).

- The algorithm:
  – Giant step: compute the pairs $(j, g^{j \cdot t})$, for $0 \leq j \leq t$. Store in a table keyed by $g^{j \cdot t}$.
  – Baby step: compute $y \cdot g^i$ for $i=0,1,2\ldots$, until you hit an item $(j, g^{j \cdot t})$ in the table. x $= jt - i$.

- Memory and running time are O(sqrt|G|).

Baby-step giant-step DL algorithm