CS 6160 Cryptology Lecture 14: Cryptographic Hardness Assumptions (RSA & Diffie-Hellman Assumptions)

Maria Francis

October 27, 2020

Integer Factorization/Factoring

- Basic idea: Given a composite integer N the factoring problem is to find integers $p, q > 1$ such that $pq = N$.
- It is a classic hard problem – simple to describe and has been recognized as a hard computational problem for a long time.
- Can be solved in exponential time $\mathcal{O}(\sqrt{N} \cdot \mathrm{polylog}(N))$ by checking whether $p = 2, \ldots, \lfloor \sqrt{N} \rfloor$ divides N. Requires $\sqrt{N}$ divisions each taking $||N||^c = \log(N)^c$ time.
- Why does this work? Largest prime factor of N may be as large as $N/2$ but smallest prime factor of N can be at most $\lfloor \sqrt{N} \rfloor$.
- There are better algorithms than this but no polynomial time algorithm.
- The problem is suspected to be outside P, NP-complete, and co-NP-complete, a candidate for the NP-intermediate complexity class.

Integer Factorization

- In the previous definition, if the adversary $\mathcal{A}$ given $N = x_1 x_2$ has to just find x_1', x_2' s.t. $N = x_1' \cdot x_2'$ then it may not be always very hard.
- If N is even, then 2 is always a factor. This happens with probability 3/4!
- It is easy every time x_1, x_2 has small prime factors.
- We need to find the hard instances.
- We need to consider x_1, x_2 two random n-bit primes (large primes) and integers.
- Now the question: How does one generate random n-bit primes efficiently?

Generating a random prime

Generating a random prime – high-level outline

Input: Length n; parameter t
Output: A uniform n-bit prime

for $i = 1$ to t:
  $p' \leftarrow \{0,1\}^{n-1}$
  $p := 1 \| p'$
  if p is prime **return** p
return fail

The output is of length exactly n and not at most n by setting the high-order bit of p to 1.
In crypto, an integer of length n means MSB is 1 and is exactly n bits long.

Generating a random prime

- The above algorithm needs a way to determine whether or not a given integer p is prime.
- Also, the probability that the algorithm outputs fail depends on t, so we need to set t so as to obtain a failure probability that is $\mathrm{negl}(n)$.
- T.S.T. the algorithm is a poly-time algo for generating primes we need to understand :
  - the probability that a uniform n bit integer is prime,
  - and how to efficiently test whether a given p is prime.

Distribution of primes

- The first point relates to the distribution of primes, an important research area in mathematics.

Theorem (Bertrand's postulate)

For any $n > 1$ the fraction of n-bit integers that are prime is at least $1/3n$.

- The stronger result is called Prime Number Theorem: The no of primes upto x is $\approx \frac{x}{\ln x}$.
- A deep result using Riemann zeta function! It says there are many more primes in the interval than guaranteed by BP.
- Riemann Hypothesis : the most important open problem in pure math! If established it will give a better estimate of the prime distribution.

Outputting a prime

- So back to the algorithm for generating prime: if we set $t = 3n^2$ then the probability that a prime is not chosen in all t iterations of the algorithm is at most

$$(1 - \frac{1}{3n})^t = ((1 - \frac{1}{3n})^{3n})^n \leq (e^{-1})^n = e^{-n} = \mathrm{negl}(n).$$

- And generally there are more primes than that stated by BP so by doing $\mathrm{poly}(n)$ iterations probability of outputting fail is $\mathrm{negl}(n)$.

Testing primality

- In the 1970s the first efficient probabilistic algorithms were developed.
- They had the following guarantee: if the input p were prime, the algorithm would always output prime. But If composite then the algo *almost always outputs composite but may output prime with a negligible (in length of p) probability.*
- This is another source of error along with the algo outputting fail! Not a big issue practically.
- A deterministic poly-time algo for testing primality was given in 2002! But it is slower than probabilistic ones.
- A probabilistic poly-time primality testing algo: Miller-Rabin algorithm. If p is a composite of length t the algo outputs composite except with prob. 2^{-t}.

The final algorithm

Generating a random prime

Input: Length n
Output: A uniform n-bit prime

for $i = 1$ to $3n^2$:
  $p' \leftarrow \{0,1\}^{n-1}$
  $p := 1\|p'$
  run the Miller–Rabin test on input p and parameter 1^n
  if the output is “prime,” **return** p
return fail

For details on Miller-Rabin algorithm check Section 8.2.2 of the Yehuda-Lindell Textbook.

The Factoring Experiment

$Factor_{\mathcal{A},GenModulus}(1^n)$:

1. Run GenModulus(1^n) to obtain (N, p, q).
2. $\mathcal{A}$ is given N and outputs $p', q' > 1$.
3. Output is 1 i $p'q' = N$ and 0 otherwise.

- GenModulus(1^n) is a poly-time algorithm that outputs (N, p, q) where $N = pq$ and p,q are n-bit primes except with negligible probability in 1^n.
- Just generate two primes (from prime generating algo) and then multiply them to get N.
- When factoring experiment returns 1, $\{p', q'\} = \{p, q\}$, unless p or q is composite and that has prob. $\mathrm{negl}(n)$.

The Factoring Assumption

- Factoring is hard relative to GenModulus if for all PPT $\mathcal{A}$,

$$Pr[Factor_{\mathcal{A},GenModulus}(1^n) = 1] \leq \text{negl}(n).$$

- The factoring assumption is the assumption that there exists a GenModulus relative to which factoring is hard.

The RSA Assumption

- Even though the factoring assumption gives a OWF (we saw this in Lecture 3), it does not directly yield a practical cryptosystem.
- We see how to construct cryptosystems whose hardness is equivalent to that of factoring.
- We give a problem whose hardness is *related to the hardness of factoring* : RSA problem
- Introduced in 1978 by Rivest, Shamir, and Adleman.
- Recalling the material we discussed in Lecture 3:
- The modular exponentiation function $x^e \bmod N$, $e > 2$ and $gcd(e, \varphi(N))$ is a a one-way function that is a *permutation*. (How? Check Number Theory recap)
- Actually, RSA function is a TDP, OWP with a trapdoor.

Overview of RSA as TDP

- RSA function $f_{N,e}(x) = x^e \bmod N$, N is the product of two primes p, q, $x \in \mathbb{Z}_N{}^*$, $e \in \mathbb{Z}_{\varphi(N)}{}^*$.
- For RSA function f, e is always chosen to be in $\mathbb{Z}_{\varphi(N)}{}^*$ $\Rightarrow \gcd(e, \varphi(N)) = 1$ and e has an inverse mod $\varphi(N)$!
- Consider $c = f_{N,e}(x) = x^e \bmod N$, how to get back x?

$$
\begin{aligned}
c^d = (x^e)^d \bmod N &= x^{ed} \bmod N \\
&= x^{1+l(\varphi(N))} \bmod N, l \in \mathbb{N} \\
&= x^1 \cdot (x^{\varphi(N)} \bmod n)^l \bmod N \\
&= x \bmod N \quad \text{(by Euler's theorem)}.
\end{aligned}
$$

RSA function as TDP

- $x \in \mathbb{Z}_N{}^*$ and $\mathbb{Z}_n{}^*$ is a group, so any power of x is in the group, $\Rightarrow f(x) = x^e \mod N$ is also in $\mathbb{Z}_N{}^*$, f is a permutation in $\mathbb{Z}_N{}^*$.
- Public values: N, e and the method to obtain $f_{N,e}(y)$ for some y, c is known.
- Private: x, p, q, d
- How to invert?
  - If you know how to factor N, then $\varphi(N) = (p-1)(q-1)$ is known and then finding d is easy!
  - Other methods that directly try to compute $\varphi(N)$ or d are just as hard.
- Knowing the factoring is the secret information making f a TDP.

RSA experiment

We have a mathematical proof, now for formalizing the security definitions.

$RSA - inv_{\mathcal{A},GenRSA}(1^n)$

1. Run GenRSA(1^n) to obtain (N, e, d).
2. Choose a uniform $y \in \mathbb{Z}_N^*$.
3. $\mathcal{A}$ is given N, e, y and outputs $x \in \mathbb{Z}*_N$.
4. Output if 1 if $x^e = y \mod N$ and 0 otherwise.

The RSA problem is hard relative to GenRSA if for all PPT $\mathcal{A}$,

$$Pr[RSA - inv_{\mathcal{A},GenRSA}(1^n) = 1] \leq \mathrm{negl}(n).$$

- The RSA assumption is that there exists a GenRSA algorithm relative to which the RSA problem is hard.

GenRSA algorithm

A suitable GenRSA algo can be constructed from any algorithm GenModulus that generates a composite modulus along with its factorization.

GenRSA – **high-level outline**

Input: Security parameter 1^n
Output: N, e, d as described in the text

$(N, p, q) \leftarrow \mathsf{GenModulus}(1^n)$
$\phi(N) := (p-1)(q-1)$
choose $e > 1$ such that $\gcd(e, \phi(N)) = 1$
compute $d := [e^{-1} \bmod \phi(N)]$
return N, e, d

What values can e take?

- In RSA e is publicly known, d is secret (as well as x, p, q).
- There does not appear to be any difference in the hardness of the RSA problem for different choices.
- $e = 3$ is popular since it requires only two multiplications.
- $e = 2^{16} + 1 = 65537$ is also a popular choice because it is a prime number with low Hamming weight.
- Unlike e choosing small values of d is a bad idea since brute force search for d can be done!
- Knowing d breaks the whole thing!

Relation between RSA and Factoring Assumptions

- If N can be factored (computing primes p, q) we can easily compute $\varphi(N) = (p-1)(q-1)$and then use that to compute $d = e^{-1} \bmod \varphi(N)$ for any e.
- I.e. The RSA problem cannot be harder than factoring.
- What about other direction? Open question.
- The best we can show is computing d from N and e, computing the Euler totient function are all polynomiall equivalent to factoring.

Discrete-Logarithm/Diffie-Hellman Assumptions

- The aim is to introduce computational problems that can be defined for any class of cyclic groups.
- $\mathcal{G}$ – denotes a generic, poly-time group generation algorithm.
- What does $\mathcal{G}(1^n)$ do?
- It outputs a description of a cyclic group G with order q (whose length is n) and a generator g for this group.
- Description? Specifies how elements of the group are represented as bit strings.
- We need the group operation and membership testing to be done by $\mathrm{poly}(n)$ algorithms.
- Group operations include exponentiation in G and sampling a uniform element h in G.

Discrete-Logarithm

- If G is a cyclic group of order q and generator g, for every $h \in G$ there is a unique $x \in \mathbb{Z}_q$ s.t. $g^x = h$.
- We call x discrete logarithm of h w.r.t. g, i.e. $x = \log_g h$.
- It is called discrete since it takes values from a fixed range and not from an infinite set as in $\mathbb{R}$.
- Discrete logarithms obey many rules as standard logarithms.
  - $\log_g 1 = 0$ (1 is the identity element of G)
  - $\log_g(h^r) = r \cdot \log_g h \bmod q$
  - $\log_g(h_1 h_2) = \log_g h_1 + \log_g h_2 \bmod q$

Discrete Logarithm Problem (DLP) in G

- Very little resemblance to the continuous logarithm.
- If you look at the values it has a random looking behavior apparent from a graph contrary to its continuous analogue.

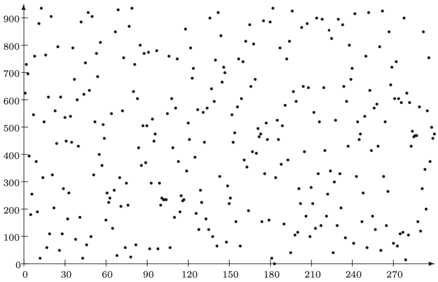

Figure 2.2: Powers $627^i \bmod 941$ for $i = 1, 2, 3, \ldots$

Discrete-Logarithm Problem

- In a cyclic group G with generator g compute $\log_g h$ for a uniform $h \in G$.
- Discrete-Logarithm experiment $DLog_{\mathcal{A},\mathcal{G}}(1^n)$:
  1. Run $\mathcal{G}(1^n)$ to obtain (G, q, g).
  2. Choose a uniform $h \in G$.
  3. $\mathcal{A}$ is given G, q, g, h and outputs $x \in \mathbb{Z}_q$.
  4. Output is 1 if $g^x = h$ and 0 otherwise.

The discrete-logarithm problem is hard relative to $\mathcal{G}$ if for all PPT $\mathcal{A}$, $Pr[DLog_{\mathcal{A},\mathcal{G}}(1^n) = 1] \leq \mathrm{negl}(n)$.
The discrete-logarithm assumption is simply the assumption that there exists a $\mathcal{G}$ for which the discrete-logarithm problem is hard.

Recap of exponentiation using repeated multiplication

- The idea is to use the binary expansion of x to convert the calculation of g^x into a succession of squarings and multiplications.
- Suppose we want to compute 2^{10}. That is 9 multiplications, naive way.
- Now consider $10 = 2^3 + 2^1$.
- Then $2^{10} = 2^{2^3+2} = 2^{2^3} \cdot 2^2$.
- For $2^{2^3} = ((2^4) \cdot (2^4))$ and $2^4 = 2^2 \cdot 2^2$.
- Therefore we have 4 multiplications, less than 9.

DLP in G of order p

1. Naive algorithm : $O(p \log p) = O(n \cdot 2^n)$ multiplications in G.
2. How? We have to find out for each g^x which is equal to the value given a. In the fast exponentiation method we have seen g^x takes $O(log_2(x))$ multiplications to compute g^x. Suppose p is a n-bit number then there are 2^n such numbers to check. So $O(n \cdot 2^n)$.
3. Best known for $G = \mathbb{Z}_p{}^*$: $O(n^{1/3 \log^{2/3} n})$, sub exponential.
4. Best known for $G =$ " elliptic curve group " : $O(2^{n/2})$, exponential.

Diffie-Hellman Problems

- They are related to computing discrete logarithms but not known to be equivalent.
- There are two variants: the computational Diffie-Hellman (CDH) problem and the decisional Diffie-Hellman (DDH) problem.
- Fix a cyclic group G and a generator $g \in G$.
- Given $h_1, h_2 \in G$, define $DH_g(h_1, h_2) := g^{\log_g h_1 \cdot \log_g h_2}$
- CDH problem: Compute $DH_g(h_1, h_2)$ for uniform h_1, h_2
- I.e. if $h_1 = g^{x_1}$ and $h_2 = g^{x_2}$, then $DH_g(h_1, h_2) = g^{x_1 \cdot x_2} = {h_1}^{x_2} = {h_2}^{x_1}$.
- Similar experiment can be designed.
- If the discrete-log problem is easy relative to $\mathcal{G}$ then CDH is easy too.

Decisional Diffie-Hellman (DDH) problem

- Distinguish $DH_g(h_1, h_2)$ from a uniform group element when h_1, h_2 are uniform.
- Formally, DDH problem is hard relative to $\mathcal{G}$ if for all PPT $\mathcal{A}$

$$Pr[\mathcal{A}(G, q, g, g^x, g^y, g^z) = 1] - Pr[\mathcal{A}(G, q, g, g^x, g^y, g^{xy}) = 1] \leq \mathrm{negl}(n).$$

- $x, y, z \in \mathbb{Z}_q$ are uniformly chosen and g^z is uniformly distributed in G.
- If CDH is easy then DDH is. Converse? Does not appear to be true.

Using Prime Order Groups

- There are various (classes of) cyclic groups in which the discrete-log and Diffie-Hellman problems are believed to be hard.
- There is a preference for cyclic groups of prime order, in a certain sense the discrete-log problem is hardest in such groups.
- It is trivial to find a generator in such groups, every element is a generator!
- Any nonzero exponent is invertible in prime order groups.
- In DDH there is another reason::
  - Distinguishing between $(h_1, h_2, DH_g(h_1, h_2))$ for uniform h_1, h_2 and (h_1, h_2, y) for uniform h_1, h_2, y.
  - $DH_g(h_1, h_2)$ by itself is close to uniform group element when the group order q is prime, not necessarily true otherwise.

Subgroups of $\mathbb{Z}_p^*$

- $\mathbb{Z}_p^*$ has a trivial representation with elements between 1 and $p-1$.
- $\mathbb{Z}_p^*$ does not have prime order, it has order $p-1$.
- DDH is in general not hard in these groups.
- What is usually done is find a prime order subgroup of $\mathbb{Z}_p^*$.
- Let $p = rq + 1$ with p, q prime. Then,

$$G := \{h^r \bmod p : h \in \mathbb{Z}_p^*\}$$

is a subgroup of $\mathbb{Z}_p^*$ of order q.

Algorithm for prime-order subgroup of $\mathbb{Z}_p^*$

A group-generation algorithm $\mathcal{G}$

Input: Security parameter 1^n, parameter $\ell = \ell(n)$
Output: Cyclic group $\mathbb{G}$, its (prime) order q, and a generator g

generate a uniform n-bit prime q
generate an ℓ-bit prime p such that $q \mid (p-1)$
  // we omit the details of how this is done
choose a uniform $h \in \mathbb{Z}_p^*$ with $h \neq 1$
set $g := [h^{(p-1)/q} \bmod p]$
return p, q, g // $\mathbb{G}$ is the order-q subgroup of $\mathbb{Z}_p^*$