Class CS 6903, Lecture n. 3

- Welcome to Lecture 3!
- In Lecture 1-2 we studied:
  - Classical cryptography, encryption with perfect secrecy
  - Background on algorithms, complexity theory. Modern cryptography: principles, primitives, and a public-key cryptosystem

Summary of Lecture 3

- Algorithmic number theory
  - Integer Arithmetic, Modular Arithmetic
  - Groups: exponentiation, inverses, random elements
- Number theory and cryptographic assumptions
  - Basic group theory
  - Primality, Factoring, RSA
- Number theory candidates for cryptographic primitives and schemes
  - Reductions
  - Candidate one-way functions from factoring
  - Candidate trapdoor permutations from RSA
  - Hard-core predicates from any one-way functions
  - Public-key cryptosystems from number theory problems

Candidates for Cryptographic Primitives

- Primitives discussed so far:
  - One-Way Functions: "easy to compute" but "hard to invert" functions
  - Trapdoor Functions: one-way functions with a trapdoor to "easily invert"
  - Hard Core Predicates: "easy to compute" given x, "hard to guess" given f(x)
  - Public-key cryptosystems
- Can we provide concrete examples for them?
  - Recall: showing OW Functions exist implies P≠NP → Approach: showing primitives under the assumption of "candidate" computational problems
  - Main candidates come from Number Theory (a well studied area, where it is easier to be confident about candidate "hard" problems). Most studied candidates are based on two main computational problems (and related ones): (1) Factoring composite integers and (2) computing discrete logarithms modulo primes; in this lecture, we focus on Factoring
  - Other problem areas (which we will only briefly discuss):
    - ⋆ Building on Discrete Logarithm Problem: Elliptic curves, Algebraic Geometry
    - ⋆ Beyond Factoring and Discrete Logarithms: Lattices, Coding Theory, etc.

(Algorithmic) Number Theory

- Number theory: well studied branch of pure mathematics concerned with the study of the properties of numbers, and, in particular, integers
- Algorithmic number theory (or, computational number theory): study of algorithms to perform number theoretic computations
  - An integer p is represented in binary notation, using n ~ log p bits; we use the notation |p|=n
  - In cryptography we look for number theory problems for which:
    - (Easy): There exists an algorithm that is polynomial in n
    - (Candidate hard): There seemingly are no such algorithms

Algorithms for Integer Arithmetic – addition, multiplication

- Basic operations:
  - Addition, subtraction
  - Multiplication, division
- Algorithmic aspects:
  - using grade-school algorithms, we can find polynomial-time algorithms for them
  - Addition of two integers a,b takes time
    - linear in max(|a|,|b|); i.e., O(max(|a|,|b|))
  - Multiplication of two integers a,b takes time
    - O(|a|*|b|); actually, there are faster ($O(n^{\log 3})$) algorithms
- Faster algorithms are very desirable in general, and especially when n is large
  - In cryptography, n ~ thousands is typical

Integer Arithmetic: Primes, Divisibility

- The sets of integers, positive integers are denoted as Z, N
- When a,b are in N, we say that a divides b or is a divisor of b, denoted as a|b, if there exists c in N such that ac=b
  - If a≠1 and a≠b, a is a non-trivial divisor or factor of b
- If a|b and a|c then a|(xb+yc) for any x,y in Z
- An integer p in N is prime if it has no factors and composite otherwise
- Fundamental theorem of arithmetic: every integer >1 can be uniquely (up to ordering) expressed as a product of prime powers; i.e., $n=p_1^{e_1}...p_k^{e_k}$, for some positive integers $e_1,...,e_k$
- Division with remainder: Let a in Z and b in N; then there exist unique q,r in Z s.t. a=qb+r, 0<=r<b
  - We say that a mod b = r

Algorithms for Integer Arithmetic – Euclidean GCD

- Definition: The greatest common divisor (gcd) of a,b in N is the max integer c such that c|a and c|b. Ex.: gdc(209,132)=11.
  - If gcd(a,b)=1 we say that a and b are relatively prime or coprime
- Fact 0: Let a,b in N s.t. a does not divide b; then gcd(a,b) = gcd(b, a mod b)
  - Proof: see proof of proposition B.6 in [KL]

The Euclidean algorithm GCD
Input: positive integers a, b, where a>=b
Output: gcd(a,b)
Instructions: if b|a then return b else return gcd(b, a mod b)

- Fact 1: $b_{i+2} \leq b_i /2$, where a_i, b_i denote arguments in i-th recursive call
  - Proof (Sketch): note that for all i, it holds that $b_{i+1} < b_i$. Then, when $b_{i+1} > b_i /2$, since $a_{i+1} = b_i$ we have that $b_{i+2} = a_{i+1} \bmod b_{i+1} = b_i \bmod b_{i+1} = b_i - b_{i+1} < b_i /2$.
- Fact 2: The Euclidean algorithm makes at most 2|b|-2 recursive calls
  - Proof (sketch): when $b_i=1$ no more calls are made. Then, from Fact 1, we see that any 2 iterations halve b_i and thus $b_i=1$ after $\leq 2|b|-2$ recursive calls.
- Ex.: (209,132) → (132,77) → (77,55) → (55,22) → (22,11) → gdc is 11

Algorithms for Integer Arithmetic – Extended Euclidean GCD

- Fact: Let a,b in Z. There exist unique c,d in Z such that ac+bd=gcd(a,b), and gcd(a,b) is the smallest integer that can be expressed in this way
  - Proof: see proof of proposition 8.2 in [KL]

The Extended Euclidean algorithm eGCD

Input: positive integers a, b, where a>=b

Output: (d,x,y), where d=gcd(a,b) and xa+yb=d

Instructions:

```
if b|a then return (b,0,1)
    else compute q,r such that a=qb+r, 0<=r<b
        set (d,x,y)=eGCD(b,r)  // note that xb+yr=d
        return (d,y,x-yq)
```

- Ex.: (209,132) → (132,77) → (77,55) → (55,22) → (22,11) → gdc is 11

  (11,-5,8) ← (11,3,-5) ← (11,-2,3) ← (11,1,-2) ← (11,0,1)

- Properties (exercise):
  - Correctness: eGCD returns (d,x,y) s.t. d=gcd(a,b) and xa+yb=d
  - Running time: eGCD runs in polynomial time

Modular Arithmetic: basic facts

- Division with remainder: Let a in Z and b in N; then there exist unique q,r in Z s.t. a=qb+r, 0<=r<b; we say that "a = r mod b" or "a and r are congruent mod b"
- Congruence mod n is an equivalence relation
  - That is: it satisfies the reflexive, symmetric and transitive properties
- If for an integer b, there exists c such that bc=1 mod n, when we say that $c=b^{-1}$ is the inverse of b mod n and b is invertible mod n
  - Ex.: $9 = 7^{-1} \bmod 62$, $13 = 10^{-1} \bmod 43$,
- Fact: let a,n, be integers. Then a is invertible mod n if and only if gcd(a,n)=1
  - Ex.: 9 and 62 are coprime, but all even numbers are not invertible mod 62
- Proof: we need to prove "if" and "only if" directions
  - "only if": a is invertible mod n, let $b=a^{-1}$; then ab-1=cn for some c in Z, or ba-cn=1; use fact in previous slide to derive that gcd(a,n) is the smallest positive integer s.t. ba-cn=gcd(a,n); thus gcd(a,n)=1
  - "if": use same fact in previous slide to derive that there exist x,y such that xa+yn=1; thus, xa=-yn+1, xa =1 mod n, and x mod n is a's inverse

Algorithms for modular arithmetic – addition, multiplication, inverses

- Computing a mod b can be done by a division with reminder over the integers and takes time polynomial in |a|+|b|
- Using this fact, and the fact that addition and multiplication over integers take polynomial time, we obtain that addition and multiplication mod n also take polynomial time
- Computing modular inverses: by Fact in previous slide, we can use Euclidean algorithm to determine whether an input element has an inverse mod n

> An algorithm for computing modular inverses:
> Input: modulus n, element a
> Output: b such that $ab=1 \bmod n$ (if b exists)
> Instructions: let (d,x,y)=eGCD(a,n); if $d \neq 1$ then return "a has no inverse mod n", else return $b=(x \bmod n)$

- Properties (exercise):
  - Correctness: it returns b such that ab=1 mod n (if b exists)
  - Efficient running time: it runs in polynomial time

Algorithms for modular arithmetic - exponentiation

- Computing $a^b \bmod n$ as before (computing a^b over integers and then reducing mod n) would be inefficient
- Reducing mod n at each multiplication, although better, is still inefficient
- Observation: $a^b \bmod n = (a^{b/2})^2 \bmod n$ if b is even or $= a(a^{(b-1)/2})^2 \bmod n$ if b is odd

The "square and multiply" algorithm SaM (modular exponentiation):
Input: modulus n, element a in $Z_n=\{0,..,n-1\}$, integer exponent b
Output: $a^b \bmod n$
Instructions: if $b=1$ return a; else
if b is even then set $t = SaM(n,a,b/2)$ and return $t^2 \bmod n$
if b is odd then set $t = SaM(n,a,(b-1)/2)$ and return $at^2 \bmod n$

- Ex.: $7^{13} \bmod 31 = 7(7^6)^2 \bmod 31 = 7((7^3)^2)^2 \bmod 31 = 7((7(7)^2)^2)^2 \bmod 31$
- Properties (exercise):
  - Correctness: it returns $a^b \bmod n$
  - Efficient running time: it runs in polynomial time

Groups

- Let G be a set. A binary operation ○ over G is a function ○ that takes an input 2 elements a,b from G and returns a third element, denoted as a ○ b
- (G,○) is a group if
  - Closure: for all a,b in G, a ○ b is in G
  - Identity: there is e in G s.t. a ○ e = a for all a in G
  - Inverses: for all a in G, there is b in G s.t. a ○ b = e
  - Associativity: for all a,b,c in G, (a ○ b) ○ c = a ○ (b ○ c)
- If G has a finite number of elements, we say that G is finite and let |G| denote the group's order, defined as the number of elements in G
- (G,○), or G, is abelian if it satisfies:
  - Commutativity: for all a,b in G, a ○ b = b ○ a
- Examples:
  - Z is an abelian group when ○ = addition but is not a group when ○ = multiplication
  - The set of real numbers R is not a group under multiplication but R\{0} is
  - Z_n is an abelian group when ○ = addition mod n, but not when ○ = mult mod n

Group exponentiation

- Group exponentiation is a generalization of modular exponentiation (when ○ = multiplication) or modular addition (when ○ = addition), where instead of considering Z_n we consider a generic group G
- For simplicity, we consider ○ = multiplication
- Fact: Let (G,○) be a group with finite order m=|G|>1. Then for any g in G, it holds that $g^m = 1$
- Proof: see proof of theorem 7.14 in [KL]
- Corollary 1: Let (G,○) be a group with finite order m=|G|>1. Then for any g in G and any integer a, it holds that $g^a = g^{a \bmod m}$
- Proof: if we let a=qm+r, for q,r integers and r = a mod m, we have that
  - $g^a = g^{qm+r} = g^{qm}\, g^r = (g^m)^q g^r = (1)^q g^r = g^r$
- Corollary 2: Let (G,○) be a group with finite order m=|G|>1. Then for any e in G, let function $f_e(g) = g^e$. If gcd(e,m)=1 then f_e is a permutation. Moreover, if d is such that de=1 mod m then f_d is the inverse of f_e
- Proof: gcd(e,m)=1 implies that e has an inverse d mod m.
  - $f_d(f_e(g)) = f_d(g^e) = (g^e)^d = g^{ed} = (g)^1 = g$ for any g implies that f_d is a permutation

The group Z_n^*

- Recall that Z_n, the set of positive integers mod n, is an abelian group when ○ = addition mod n but not a group when ○ = multiplication mod n
- Instead, Z_n^* , the set of positive integers mod n that are relatively prime with n, is a group when ○ = multiplication mod n
  - It is also abelian
- Define $\phi(n)$ as the order of Z_n^*, $\phi(n) = |Z_n^*|$
- When n is prime then all elements in {1,..,n-1} are coprime with n, so $\phi(n) = n-1$
- When n=pq, for p,q primes, all elements in {1,..,n-1} except for p, 2p, 3p,…(q-1)p and q, 2q, …, (p-1)q are coprime with n; then $\phi(n) = n-1-(q-1)-(p-1)=(p-1)(q-1)$

Choosing Random Group Elements

- Let $(G,\circ)$ be a group. We represent elements of G as bit-strings (or, strings), where each element has a unique representation and thus a unique string.

> An algorithm for randomly choosing a group element:
> Input: a description of a group G, a length parameter m
> Output: a random element of G
> Instructions: Repeat k times: (randomly choose x from $\{0,1\}^m$ and return x if x is in G). Return "fail".

- If x is returned, then x is random in G
- The probability that "fail" is returned is $(1-|G|/2^m)^k$
  - Note: as long as $|G|>2^m/p(m)$ for some polynomial p, we can choose $k = m\,p(m)$ and obtain $(1-|G|/2^m)^k < (1-1/p(m))^{m\,p(m)} < (1/e)^m$ which is negligible
  - Assuming $|G|>2^m/p(m)$ is wlog or we could never find an element in G
  - $G= Z_n$ implies that $(1-|G|/2^n)^k < (1/2)^k$ as $|G|>2^{n-1}$
  - $G= Z_n^*$ for n=pq, p and q being primes, implies that $(1-|G|/2^n)^k < (1-\phi(n)/2^n)^k < (1-(1-\text{negligible}(n)))^k = \text{negligible}(n)$

Question set 7

- Prove that for any a,b,c in N,

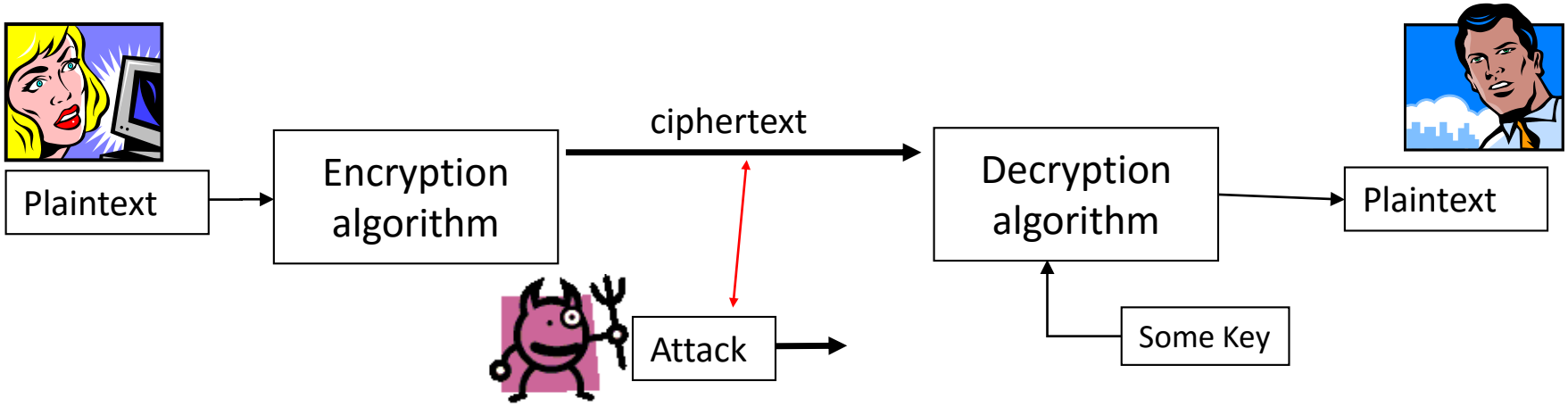
  if a|b and a|c then a|(xb+yz) for all x,y in Z

- Prove the correctness and running time properties of the following algorithms:
  - extended Euclidean algorithm
  - modular inverses
  - modular exponentiation
  - random choice of group elements
- Prove the previously made claims about some group examples
  - The set Z of integers is an abelian group when ○ = addition but is not a group when ○ = multiplication
  - The set R of real numbers is not a group under multiplication but R\{0} is
  - The set Z_n is an abelian group when ○ = addition mod n but not when ○ = multiplication mod n

Summary of Lecture 3

- Algorithmic number theory
  - Integer Arithmetic, Modular Arithmetic
  - Groups: exponentiation, inverses, random elements
- Number theory and cryptographic assumptions
  - Basic group theory
  - Primality, Factoring, RSA
- Number theory candidates for cryptographic primitives and schemes
  - Reductions
  - Candidate one-way functions from factoring
  - Candidate trapdoor permutations from RSA
  - Hard-core predicates from any one-way functions
  - Public-key cryptosystems from number theory problems

Cryptographic Assumptions



- Recall:
  - The adversary's task to, say, obtain the plaintext from the ciphertext, should be a hard problem (say, from number theory) and thus require an inefficient algorithm (in the sense of complexity theory)
  - Alice and Bob need to run efficient encryption and decryption algorithms (also from number theory)
- Two candidate hard problems in this lecture:
  - factoring composite integers, inverting RSA function

Choosing Random Primes

- Algorithm consists of choosing k random n-bit positive integers and stopping as soon as a prime is found
  - Makes sure integer has exactly, rather than at most, n bits (see algorithm 8.31 in [KL])
- Is this algorithm efficient (i.e., polynomial)? This depends on:
  - What is the probability that a random n-bit integer is a prime
  - How to test whether an integer is a prime
- Prime number theorem: there is a constant c s.t. for any n>1, the number of n-bit primes is $\geq c\, 2^{n-1}/n$
  - $\rightarrow$ prob that random n-bit integer is prime is $\geq (c2^{n-1})/(n2^{n-1}) = c/n$
  - If $k= n^2/c$, then prob(algorithm fails) $\leq (1-c/n)^k = ((1-c/n)^{n/c})^n \leq (e^{-1})^n = e^{-n}$

Primality Testing

- Old problem, first efficient probabilistic algorithm in ~1970, first efficient deterministic algorithm in 2002
  - Probabilistic algorithm is faster
- Theorem: The probabilistic Miller-Rabin primality testing algorithm satisfies the following:
  - If input is prime, it outputs "prime" with prob = 1
  - If input is composite, it outputs "prime" with prob < 2^{-k}

Towards the Miller-Rabin primality testing algorithm:
Input: an integer n and a parameter k
Output: 1/0 denoting prime/composite
Instructions: Repeat k times: (randomly choose x from $\{1,..,n-1\}$ and return 0 if $\gcd(x,n)\neq 1$ or $x^{n-1}\neq 1$). Return 1.

The Factoring Assumption

- Integer factorization or factoring is the (rather old) problem of, given n in N, finding p,q in N such that n=pq
- Can be inefficiently ($O(\sqrt{n}$ polylog n)) solved by trial division; no efficient solutions are known after many years of attempts
- Experiment(A,s):
  - randomly choose s-bit primes p,q
  - compute n=pq
  - run (attacker, or adversary) algorithm A on input n, and let p',q' be its output;
  - if n=p'q' then return 1 (for attack succeeded)

    else return 0 (for attack failed)
- Factoring assumption: for all probabilistic poly-time A, prob(Experiment(A,s)=1) is negligible (in s)

Question set 8

- What type of problems do we need in cryptosystems?
  - E.g.: Easy/hard to compute/generate for some/all of the parties…
- Are the following problems easy or (conjectured to be) hard?
  - determining whether an integer is prime
  - computing the factorization of an (arbitrary) integer
  - computing modular exponentiation in a cyclic group
  - computing the discrete logarithm in a cyclic group
- Which of these three sets of assumptions are useful in cryptography? Why?
  - $P \neq NP$, $BPP \neq NP$, etc.
  - The existence of one-way function, trapdoor function, etc.
  - the factoring assumption, and the discrete logarithm assumption, etc.

Summary of Lecture 3

- Algorithmic number theory
  - Integer Arithmetic, Modular Arithmetic
  - Groups: exponentiation, inverses, random elements
- Number theory and cryptographic assumptions
  - Basic group theory
  - Primality, Factoring, RSA
- Number theory candidates for cryptographic primitives and schemes
  - Reductions
  - Candidate one-way functions from factoring
  - Candidate trapdoor permutations from RSA
  - Hard-core predicates from any one-way functions
  - Public-key cryptosystems from number theory problems

Reductions

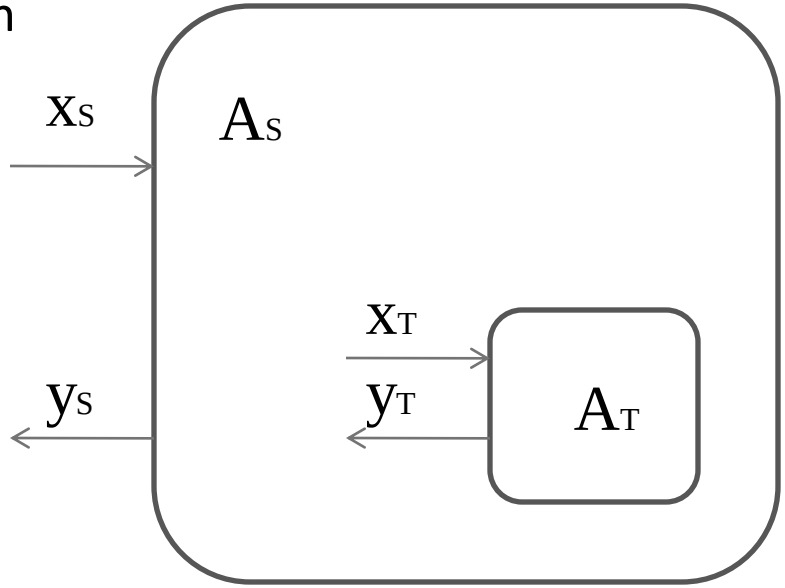
- In modern cryptography we typically prove statements of the type "if S then T", where S and T can be statements about computational problems, cryptographic primitives, schemes or protocols
- A typical approach to proving this statement consists of rephrasing this statement into its (equivalent) contrapositive "if not(T) then not(S)" and then proving the latter via a "proof by reduction"
- Sketch of typical proof by reduction:

1) not(T) → there exists algorithm A_T that on input x_T returns output y_T

2) the reduction step consists of defining an algorithm A_S that on input x_S returns y_S

3) A_S prepares one or more x_T, uses A_T to compute one or more y_T and finally computes y_S

4) The proof is concluded by showing that this A_S implies not(S)



Reductions: examples

- To show that a language L is NP-complete, we show that (a) L is in NP and that (b) any language L' in NP can be reduced to L; this latter statement involves a polynomial-time computable reduction; that is, a function f that maps an instance x for L' to an instance y for L such that " x in L' " if and only if " y in L"
- Previously we proved the statement: if S then T, where
  - S = "there exists a trapdoor permutation with a hard-core predicate"
  - T= "it is hard for an eavesdropper to guess a bit encrypted via the encryption scheme in slide 4 with prob > ½ + negligible"
- Another example: if S then T, where
  - S = "f is a one-way function"
  - T= "g is a one-way function", where g is constructed using f
- Another example: if S then T, where
  - S = "finding the complete factorization of a positive integer is hard"
  - T= "finding a non-trivial factor of a positive integer is hard"

A proof by reduction

- As an example for a proof by reduction, we want to prove statement: if S then T, where
  - S = “f is a one-way function”
  - T= “g is a one-way function”, where g is defined as g(a1,a2)=(f(a1),f(a2)) for any a1,a2
- We prove the (logically equivalent) contrapositive statement “if not(T) then not(S)” statement by a “proof by reduction” technique:
  - (Assume not(T)): Assume g is not one-way, then there exists a polynomial-time algorithm A_T that inverts g with not negligible probability; that is, on input (b1,b2)=g(a1,a2), returns (a1’,a2’) such that (b1,b2)=g(a1’,a2’) with not negligible probability
  - (Reduction step): Now we try to construct an algorithm A_S that tries to invert f; on input an arbitrary b=f(a), A_S randomly chooses a’, computes b’=f(a’) and runs A_T on input (b,b’), thus obtaining (a,a’) as output; finally, A_S returns: a.
  - (Prove reduction implies not(S)): Since A_T inverts g with not negligible probability, A_T computes preimages under f of both b1 and b2 with not negligible probability; thus, A_S computes a preimage under f of b with not negligible probability (because he had set b1=b before running A_T); thus, A_S inverts f with not negligible probability

Reductions in modern crypto

- Reductions allow us to prove the statements "if S then T" where S and T are statements about cryptographic objects in the same class or in different classes

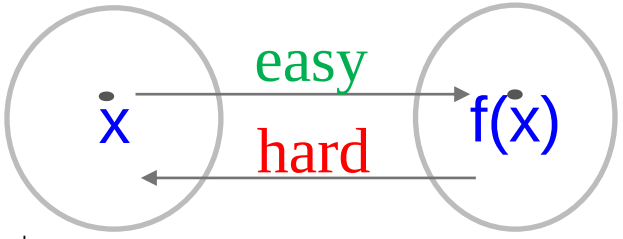
Classes of cryptographic objects	Examples of objects in the class	Statements about objects
1) Computational problems	Factoring integers, computing discrete logarithms, etc.	"Problem P is hard",
2) Assumptions (based on problems)	Hardness of Factoring, Discrete log, etc.	"Assumption A holds"
3) Primitives	One-way functions, Trapdoor permutations, Hard-core predicates, etc.	"Primitive F exists"
4) Construction, Schemes, Protocols	Encryption Schemes, etc.	"Scheme S is secure"

- Ex1: S = "factoring is hard", T = "Encryption scheme ES1 is secure"
- Ex2: S = "f is a trapdoor permutation", T = "Encryption scheme ES2 is secure"
- Ex3: S = "factoring is hard", T = "f is a trapdoor permutation"
- Ex4: S = "f is a one-way function", T="P is a hard-core predicate for f"

Candidate one-way functions

- $f:\{0,1\}^n \rightarrow \{0,1\}^n$, defined as $f(x)=y$, where
  - $x=(p|q)$, p,q are positive integers of length n/2
  - $y=pq$ (integer product)
- Running time: $O(n^2)$
- Is f one-way?
  - Not if y has many small non-trivial factors
  - Probably yes if y is the product of a few (e.g., 2) large primes
- Prob (y is product of two large primes) ~ $1/n^2$ → f is hard on a "noticeable" subset of inputs → f is a candidate for a "weak" one-way function

- $g:S \rightarrow \{0,1\}^n$, where S is a specific subset of $\{0,1\}^n$, defined as $g(x)=y$, where
  - $x=(p|q)$, p,q are primes of length n/2
  - $y=pq$ (prime number product)
- Running time: $O(n^2)$
- Is g one-way?
- Trial division is not efficient, as it runs in time $O(\sqrt{y})$, which is super-polynomial in n
- Many other attempts for efficient algorithms failed → g is a good candidate for a one-way function
- Theorem: if factoring (prime number products) is hard then g is a one-way function

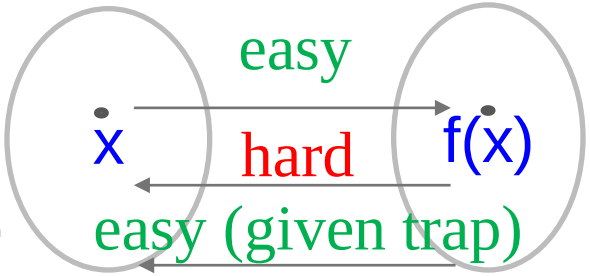


Factoring → One-way function

- g:S→$\{0,1\}^n$, where S is a specific subset of $\{0,1\}^n$, defined as g(x)=y, where
  - x=(p|q), p,q are primes of length n/2
  - y=pq (prime number product)
- Theorem: if factoring prime number products is hard then g is a one-way function
- Proof: As usual, to prove "if S then T", we prove "if not(T) then not(S)";
- not(T) →g is not one-way → there exists A_T that inverts g (i.e. computes p',q' from y such that y=p'q') with not negligible probability
- We now construct A_S that solves factoring; on input a product m of two primes, A_S runs A_T on input y=m, thus obtaining (p',q'). A_S returns precisely (p',q')
- Note that with not negligible probability the following holds: A_T returns p',q' such that y=p'q', and since A_S set y=m, A_U obtains a pair p',q, such that m=p',q', thus factoring m → not(S)

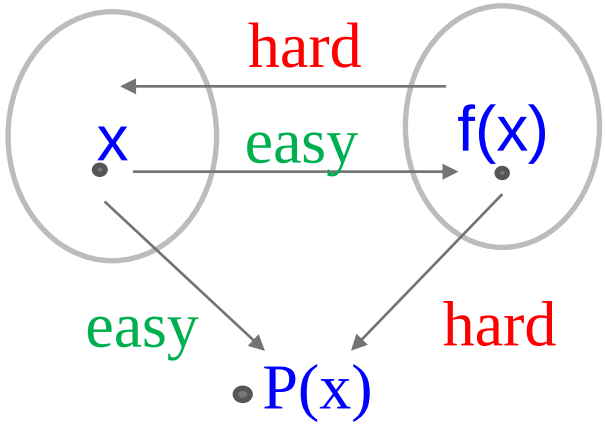
The RSA trapdoor permutation candidate

- Let n=pq, where p,q are primes of same length, let $Z_n^* = \{x : (x,n) = 1\}$ and define phi(n)=(p-1)(q-1) (Euler's totient function)
- Choose e s.t. (e, phi(n))=1 and d s.t. ed =1 mod phi(n)
  - Unique d exists as (e, phi(n))=1
- Permutation $f_{n,e} : Z_n^* \rightarrow Z_n^*$ is defined as $f_{n,e}(x)=x^e \bmod n$, for any x in Z_n^*
- Trapdoor = d
- RSA problem: computing x from $y=f_{n,e}(x)$
- Theorem: if the RSA problem is hard, then $f_{n,e}$ is a trapdoor permutation
- Proof idea: observe that $f_{n,e}$ is a permutation and that d is a trapdoor using Z_n^*'s properties; then prove that it is one-way using a proof by reduction, very similarly as in the factoring-based candidate one-way function.
- Remarks:
  - the RSA problem is different from (and may be strictly less hard than) factoring
  - Theorem: RSA problem is hard → factoring problem is hard
  - But this is unknown: Factoring problem is hard → RSA problem is hard



Towards candidate hard-core predicates

- Hard-core predicates are predicates P associated with a given one-way function $f : \{0,1\}^n \rightarrow \{0,1\}^n$



- We could construct a hard-core predicate hc for a specific (candidate) one-way function
- We will actually construct one for any one-way function
  - A first (incorrect!) suggestion is hc(x) = x[1] xor … xor x[n]
    - Note: it is not hard-core for the function g(x)=(f(x), hc(x))
    - In fact, any predicate returning a single or a few bits as output has this problem

Hard-core predicates for any one-way function

- Theorem: Given a one-way function $f : \{0,1\}^n \rightarrow \{0,1\}^n$, we can define a function $g: \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ as $g(x,r)=(f(x),r)$, such that (1) g is one-way and (2) predicate $hc(x,r) = (r[1] \text{ AND } x[1]) \text{ XOR } \ldots \text{ XOR } (r[n] \text{ AND } x[n])$ is hard-core for g
  - Note: AND = product mod 2, XOR = sum mod 2, → value output by hc can be seen as (1) inner product of vectors r and x, or (2) xor of a random subset of g's input bits

- Consequence: there is a (probabilistic) hard-core predicate for the one-way functions based on factoring, discrete log

- Proof (only of a simpler case here, but see 7.3 in [KL] textbook): A proof by reduction
  - We assume hc is not hard-core and try to prove that f is not one-way
  - hc not hard core → there exists a polynomial-time adversary A that, on input (f(x),r), computes hc(x,r) with not negligible probability
  - Here, we only consider the simpler case that this probability is actually 1

Hard-core predicates for any one-way function: proof

- Theorem: Given one-way functions $f : \{0,1\}^n \rightarrow \{0,1\}^n$, and $g: \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ defined as $g(x,r)=(f(x),r)$, the predicate $hc(x,r)=r_1x_1$ xor … xor r_nx_n is hard-core for g
- Proof (continued):
- We show a polynomial-time algorithm A' that uses A to invert function f
- On input y such that |y|=n, A' does the following:
  - Set e(i) = n-bit string with 1 in the i-th position and 0 in remaining ones, for i=1,…,n
  - compute x'(i)=A(y,e(i)), for i=1,…,n
  - output: x' = x'(1)|…|x'(n)
- Note:
  - Prob [A' inverts f] = Prob [x'(i) = i-th bit x(i) of y's preimage under f, for all i]
- As we assumed that A always computes hc exactly, it holds that
  - x'(i) = x(1)e(i,1) xor … xor x(n)e(i,n), where x(j) is j-th bit of x, e(i,j) is j-th bit of e(i)
- By definition of e(i), only e(i,i)=1 and e(i,j)=0 for all j<>i → x'(i)=x(i) for all i=1,..,n
  → Prob [A' inverts f] = Prob [A inverts y n times] = 1 (by our assumption)
  → f is not one-way

Public-key cryptosystems from number-theory problems

- In Lecture 2, we discussed the following general (but abstract) result:
  - If there exists a trapdoor permutation TP and a hard-core predicate HCP for TP, then there exists a public-key cryptosystem that is secure (against a ciphertext eavesdropper)

- In this lecture, we proved:
  - 1) If the RSA problem is hard then there exists a trapdoor permutation;
  - 2) if there exists a one-way function f, there exists a hard-core predicate for (a simple variation of) f.

- Combining (1), (2) and the above general result, we obtain a more specific (but concrete) result:
  - if the RSA problem is hard, there exists a public-key cryptosystem that is secure (against a ciphertext-eavesdropping adversary)

Factoring problem – key lengths, etc.

- Key length has to be large: It is generally presumed that factoring is a hard problem if the length of the input integer *n* is sufficiently large
- RSA vs factoring: Inverting RSA trapdoor permutations is not harder than factoring and theoretically speaking may be easier (although no faster algorithm has been found); however, many researchers expect it to be similarly hard; thus, considerations on the hardness of factoring often apply on the hardness of RSA
- Key length and the hardness of factoring:
  - As of 2010, the largest (known) number factored by a general-purpose factoring algorithm was 768 bits long (see RSA-768), using a state-of-the-art distributed implementation
  - If *n* is 300 bits long or shorter, it can be factored in a few hours on a personal computer, using freely available software. Keys of 512 bits have been shown to be practically breakable in 1999 using several hundred computers and are now factored in a few weeks using common hardware. A theoretical hardware device named TWIRL and described in 2003 called into question the security of 1024 bit keys; thus, it is often recommended today that *n* be at least 2048 bits long.
  - In practice: RSA keys are typically 1024 or 2048 bits long; some experts believe that 1024-bit keys may become breakable in the near future; this does not seem true today for 4096-bit keys
  - But note: In 1994, a breakthrough result showed that a quantum computer (if one could ever be practically created for the purpose) would be able to factor integers in polynomial time, thus also inverting the RSA trapdoor permutation.

Question set 9

- Are these problems easy or (conjectured to be) hard?
  - deciding if an integer is a prime number
  - deciding if an integer is a composite number
  - factoring an integer
  - the RSA problem
- Did we prove/disprove any of these theorems?
  - if the factoring problem is hard, then the public-key cryptosystem in Lecture 2 is secure (against a ciphertext-eavesdropping adversary)
  - if the problem of deciding primality is hard, then the public-key cryptosystem in Lecture 2 is secure (against a ciphertext-eavesdropping adversary)
- What are the hypothesis and conclusion in the following statements?
  - If S then T
  - If (S1 and S2) then (T1 or T2)
  - If (S1 or S2) then (T1 and T2)
- What is the contrapositive of the above statements? What is the converse of the above statements?
- To prove that any one of the above statements is true, which version of the statement do you typically try to prove in cryptography?
- To prove that any one of the above statements is false, which version of the statement do you typically try to prove in cryptography?

Class CS 6903, End of Lecture n. 3

Reference → Topic ↓	[KL]	[MOV]	[FSK]
Algorithmic number theory	B.1, B.2	2.4	10, 11
Number theory and cryptographic assumptions	8.1.1-8.1.3, 8.2.1-8.2.3, 8.3.1-8.3.2 (edition 1: chapter 7, same subsection numbers)	3	
Reductions, Proof by reductions	3.1.3		
Number theory candidates for cryptographic primitives and schemes	7.1.2-7.1.3, 7.3.1-7.3.2 (edition 1: chapter 6, same subsection numbers) 8.2.4, 8.3.2-8.3.4, 8.4.1 (edition 1: chapter 7, same subsection numbers), 13.1.1 (edition 1: 10.7.1)	2.4, 3	10
Factoring – key lengths, practical considerations, etc.	Wikipedia (RSA)		