0368.4162: Introduction to Cryptography	Ran Canetti
Lecture 2	
10 November 2008	Fall 2008

Scribes: Yacov Lifshits, Ilia Lotosh

Today's lecture concerns:

- Hard problems
- One-way functions

Hard problems

Most of the useful cryptographic schemes need some hard computational problem as the basis, for their security.
A lot of hard problems exist – but not all of them are useful for building some cryptographic scheme. For example, NPc problems are hard – but only guarantee worst-case hardness, and are not useful for a cryptographic scheme.
For our purposes, we require a problem that allows efficient generation of instances that are almost always hard.

Examples of cryptographically useful problems:

1. Factoring

   The problem:

   Given an integer N, find its prime components
   (*Prime* $p_1, p_2, \ldots p_k$, *s.t.* $p_1 * \ldots * p_k = N$, where N representation is at most n-bit long, and thus n is roughly $\log N$).

   Multiplication of two integers of length n bit each takes $O(n^2)$ – which is feasible.
   In contrast, factoring a given integer into its prime components seems to be hard, in most cases. If two or more prime factors of the integers are large, then the best known algorithms are exponential.
   For the case of $N = p * q$, where p,q are primes and roughly equal in size, the best algorithms (Dixon's algorithm, Number Field Sieve) run in time $2^{O(\sqrt{\log n \log \log n})}$.

   Remarks:

   - Factoring may be easy in some cases. For example, if all the prime factors of N

are small enough, then factoring N is polynomial in the number of bits in N.
- Factoring is not known to be NPc. In fact, it is probably not NPc (or else NP=coNP).

2. Discrete Log

Before stating the discrete log problem, let us recall the definitions of groups:

A group is a set S and an operation # with the following properties:
- Closeness: $\forall a,b \in S, a\#b \in S$
- Associativity: $\forall a,b,c \in S, (a\#b)\#c = a\#(b\#c)$
- Identity: $\exists I \in S, s.t. \forall a \in S, a\#I = I\#a = a$
- Inverse: $\forall a \in S, \exists b \in S, s.t. a\#b = b\#a = I$

A group is Abelian if $\forall a,b \in S, a\#b = b\#a$.

An *order* of an element a in group <S, #> is defined as the minimum *i*, s.t. $a^i = I$.
It can be proven that $i \leq |S|$.
a is called a *generator* if its order is |S|.

For a prime p, Z_p^* is the multiplicative group modulo p, that is,
$S = \{1,..., p-1\}$ where the group operation is multiplication $mod\ p$.

Now, we state the discrete log problem:

Given $a,b \in Z_p^*$ find *i* s.t. $a^i = b \bmod p$.

Given $b \in Z_p^*$ and an integer *i*, it is relatively easy to compute $a^i \bmod p$. It can be done in $O(n^3)$, where n is the number of bits in representation of p.
In contrast, no efficient algorithm is known for solving the discrete log problem – even if it is known that a solution exists.
The most efficient algorithms known (Index Calculus, Number Field Sieve) run in time $2^{O(\sqrt[3]{\log n \log\log n})}$.

Remarks:
- Just like factoring, discrete log may be easy in some cases, for instance if the order of a is small.

3. Subset Sum

The problem:

Given n n-bit integers $x_1, x_2,..., x_n$ and a target integer **T**, find a subset ***I*** of

$\{1,...,n\}$, so that $\sum_{i \in I} x_i = T$.

The corresponding decision problem (given $x_1, x_2, ..., x_n$ and **T**, decide whether such ***I*** exists) is in NPc.
Recall, that in the beginning of today's lecture we stated that not all NPc problems are cryptographically useful. Subset Sum is an example of an NPc problem that appears to be useful.

Like the previous examples, this problem, too, has cases where finding a solution is easy. When the integers are random, however, finding a solution seems hard. The best known algorithms run in time $2^{n^{\Omega(1)}}$.

4. Expander-based

Before stating the expander problem, we define an expander graph:
A graph of n nodes is called an expander if:

- the degree of each node is small (e.g., constant)
- the graph is highly connected:
  Let S be a set of nodes, and let N(S) be S plus the set of neighbors of nodes in S. If S is sufficiently large (e.g. $|S| < \frac{n}{2}$), then|N(S)| > c|S|, for some constant c > 1.

We will use expander graphs that are bipartite.

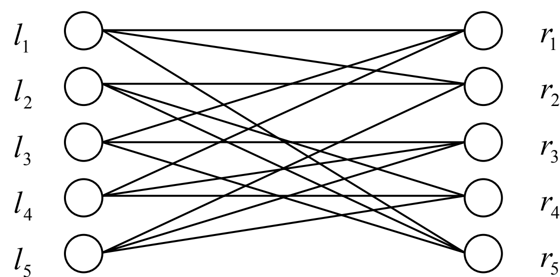

Fig 1. A bipartite graph of degree d=3

We will now state the expander graph problem, suggested by Goldreich in http://www.wisdom.weizmann.ac.il/~oded/PS/ow-candid.ps

Let G be a bipartite graph of degree d with 2n nodes, with $l_1,...,l_n$ denoting "left" nodes and $r_1, r_2, ..., r_n$ – "right" nodes.
Let $P: \{0,1\}^d \rightarrow \{0,1\}$ be a predicate on d bits (note that to describe P we need 2^{d+1} bits – so d should be relatively small).

The problem:

Given n (bits) $r_1, r_2, ..., r_n$, find bits $l_1, ..., l_n$, s.t. $\forall i, r_i = P(l_{j_1}, ... l_{j_d})$, where $l_{j_1}, ..., l_{j_d}$ are neighbors of node j on the right.

Note: since the degree of G is d, $\{l_j \mid (j,i) \in G\}$ is d bits, and thus $l_{j_1}, ..., l_{j_d}$ is a valid input for P.

We do not have any proof or rigorous analysis for this problem. However, it seems hard to come up with solutions when P is a random predicate, and G is an expander. The "rationale" is that each bit on the left "influences" many bits on the right. In fact, any subset of bits on the left influences a large set of bits on the right. Thus, it is hoped to be hard to find by simple trial and error the right setting of bits on the left that matches all the bits on the right. (But there may well be methods other than trial and error, that exploit structure in say P or G).

The principle of constructing a function so that any change in some bit of the output leads to a change in many bits of the input is a central principle in constructing cryptographic functions.

The notion of One Way Functions

So far we have seen several examples of hard problems. Now, how do we put all these problems under one roof? It would be very handful to have a single construct/abstraction that captures the "essence" in all these function.

Intuitively the common theme in all of the problems is: there is an "easy part" (it's easy to generate an instance of the problem) and a "hard part" (it's hard to solve a given instance of a problem). We'll capture this duality via the notion of a function that is easy to compute and hard to invert:

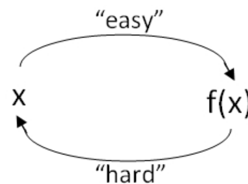

Fig 2. Intuition behind One Way Function

But how do we formalize this intuitive definition?

Attempt 1

Let's define a function $f:\{0,1\}^n \rightarrow \{0,1\}^*$ to be (t_1,t_2)-**one way** if:

a) It can be computed in time t_1
b) For all algorithms A that run in time t_2 and for all $x \in \{0,1\}^n$, $A(f(x)) \neq x$

This definition is problematic, since requirement (b) is too strong: for every function f and every $x \in \{0,1\}^n$, we can construct an algorithm that will always output x and thus $A(f(x)) = x$. Hence there is no function satisfying this definition, but the intuition tells us that there are OWF in the world!

Attempt 2

Let's try to make a probalistic statement – we will allow A to invert f on some very small fraction of the points:

A function $f:\{0,1\}^n \rightarrow \{0,1\}^*$ is (t_1,t_2,ε)-**one way** if:

a) It can be computed in time t_1
b) For all algorithms A that run in time t_2, $\Pr_{x\in\{0,1\}^n,c(A)}[A(f(x)) = x] < \varepsilon$. (Here we also allow A to be a randomized algorithm with a set of coins c(A))

This definition is also problematic, since it's too easy to satisfy: For example let's take $f \equiv 0$, it's definitely easy to compute, also given a random x there is no way to guess a pre-image of $f(x)$ since there every point is a pre-image of $0 = f(x)$. So this function satisfies the definition, but it doesn't seem to capture any notion of computational hardness.

Attempt 3

To overcome this problem we will only require now that A returns some point in $f(x)$'s pre-image (with good probability). The formal definition will be:

A function $f:\{0,1\}^n \to \{0,1\}^*$ is (t_1,t_2,ε) -**one way** if:

a) It can be computed in time t_1
b) For all algorithms A that run in time t_2, $\Pr_{x\in\{0,1\}^n, c(A)}[f(A(f(x)))=f(x)]<\varepsilon$.

This definition guarantees us the desired properties (easy to compute, hard to reverse), but only for a specific set of parameters. What if the adversary runs for slightly more than t_2 time?

Attempt 4
In order to guarantee OWF properties for any runtime of the adversary we can write:
A function $f:\{0,1\}^n \to \{0,1\}^*$ is $(t_1,\varepsilon(t))$ -**one way** if:

a) It can be computed in time t_1
b) For all algorithms A that run in time t, $\Pr_{x\in\{0,1\}^n, c(A)}[f(A(f(x)))=f(x)]<\varepsilon(t)$.

This is a reasonable definition, but it's not convenient to work with since it's very precise in its guarantees on the adversary – it provides one specific tradeoff between runtime and inversion probability.

We would like to have a notion that:

- Is more general/abstract (i.e. has less parameters)
- Gives more "slack" in the analysis
- Allows "tuning" the level of security to be larger or smaller.

We will achieve this by moving to an asymptotic formulation:

- We'll use a "security parameter", n, that tends to infinity (in our case, n is the length of f 's input).
- We'll equate "feasible adversarial computation" with "polynomial in n".
- We'll equate "small success probability" with "negligible probability":
  Definition: A function $f:N\to R$ is negligible if it tends to 0 faster than any reverse polynomial: $\forall c>0 \exists n_c \forall n>n_c \Rightarrow f(n)<\frac{1}{n^c}$
  (or, in shorthand, $f(n)=n^{-\omega(1)}$)

Attempt 5
A function $f:\{0,1\}^n \to \{0,1\}^*$ is a **OWF** if:

a) It can be computed using "efficient" algorithm (i.e. in time polynomial in the input length)
b) For all polynomial-time algorithms A, $p(n)=\Pr_{x\in\{0,1\}^n, c(A)}[f(A(f(x)))=f(x)]$ is a negligible function.

Transfer to asymptotic formulation has introduced a new problem: if $f(x)$ is much shorter than x then A won't have enough time to work on $f(x)$ (its input and it must run in time polynomial to the length of its input), so once again it's too easy to satisfy the definition.

Attempt 6

To solve the problem – we have to ensure that A has time polynomial in size of x to run. We achieve this by providing A with additional n bits. And formally:

A function $f:\{0,1\}^n \to \{0,1\}^*$ is a **OWF** if:

a) It can be computed using "efficient" algorithm (i.e. in time polynomial in the input length)
b) For all polynomial-time algorithms A, $p(n)=\Pr_{x\in\{0,1\}^n, c(A)}[f(A(1^n, f(x)))=f(x)]$ is a negligible function.

This definition works, but may be it's too permissive? What if there is no single polytime algorithm that inverts f, but for each length n there is an algorithm for inverting f on that particular n in time $poly(n)$? This problem arises from the fact that we treat A as a standard polytime algorithm. To deal with this concern we will be modeling A as a **non-uniform** polytime machine:

> *A Turing machine is* ***non-uniform polynomial time*** *if it has an additional "advice tape" that depends only on the input length. (The length of this tape is polynomial in length of the input; otherwise the machine will not be able to read all the advice)*
>
> **Note:** *In complexity theory this additional tape can be shown to give substantially more power to algorithms. For instance,* `NU-P` *is much stronger than* `P`*; in particular it contains some undecidable languages.*

Another point in favor of the non-uniform treatment is that typically security analysis becomes simpler in this model. As an example, we have the following. It turns out that for non-uniform algorithms, the ability to toss coins doesn't help in the task of inverting functions:

Claim: For any f and any randomized non-uniform polytime A, there exists a deterministic non-uniform polytime A' such that:

$$\Pr_{x\in\{0,1\}^n, c(A)}[f(A(1^n, f(x)))=f(x)] \le \Pr_{x\in\{0,1\}^n}[f(A'(1^n, f(x)))=f(x)].$$

Proof: Let r be the set of random choices of A that maximizes the likelihood of success. Then A' will have r in its advice string and run A(r,…). (r has polynomial length, since A runs for polynomial time).

■

So, the final definition is:

Definition: A function $f:\{0,1\}^n \to \{0,1\}^*$ is **OWF** if:

a) It can be computed using "efficient" algorithm (i.e. in time polynomial in the input length)

b) For all non-uniform polynomial-time algorithms A,
$p(n) = \Pr_{x \in \{0,1\}^n}[f(A(1^n, f(x))) = f(x)]$ is a negligible function.

Note:
The choice of polynomial time and negligible probability is a "natural pairing". In particular, it is closed under simple operations such as repetition in order to boost success probability, and thus they lead to a "robust" notion of OWFs.
Still, there is nothing "holy" about these choices. For instance, it is often useful to strengthen the definition to require that $p(n) = n^{-\omega(\log n)}$ or even $p(n) = 2^{-n^{\Omega(1)}}$. (For all the candidates that we mentioned, the best known inversion algorithms are consistent with these stronger assumptions.)

Examples of One Way Functions

Let's return now to the list of hard problems that we started with, and try to formalize them as One Way Functions:

1. Factoring

Let $f_{mul}(x, y) = xy$. We restrict the domain to consist of all pairs of integers which have the same length and both are prime numbers. We can implement such restriction since there is a polynomial time algorithm that checks whether given number is prime or not.

Note: f_{mul} is length preserving, but it is neither one-to-one (injective) nor onto (surjective).

2. Discrete log

Let $f_{DL}(p, g, i) = (p, g, g^i \bmod p)$. We restrict the domain to consist of all triplets of integer where p is prime number (in a same way as above), g is a generator for Z_p^* group. This way we ensure that we have a correct instance of a discrete log problem.

3. Subset sum

Let $f_{SS}(x_1, x_2, ..., x_n, I) = (x_1, x_2, ..., x_n, \sum_{i \in I} x_i)$.

4. Expander based

Let $f_{EXP}(G, P, l_1, l_2, ... l_n) = (G, P, P(N_G(l_1)), ..., N_G(l_n))$. Here we also have to add input validation to the definition of f_{EXP}, that will make sure that G is indeed expander and bi-partite. We restrict our domain to expanders that are easy to validate.

Weak One Way Functions

We have shown that OWFs exist if the problems that we presented are indeed hard. Can we show that OWFs exist based on other ("weaker") assumptions?

Note: Existence of OWFs implies $P \neq NP$. (Seems intuitively obvious, we'll see a real proof in next lecture).

Does $P \neq NP$ imply existence of OWFs? We don't know... (Even for the case of OWFs against uniform adversaries). There is some evidence that a proof would be hard; still, it's an actively studied research problem.

Below we will show a more modest implication. We will define a very weak variant of OWFs, and show that the weak variant implies the standard (strong) definition.

Definition: A function $f : \{0,1\}^n \to \{0,1\}^*$ is weakly one way (**Weak OWF**) if:

a) It can be computed using "efficient" algorithm (i.e. in time polynomial in the input length)
b) There exists a constant $c > 0$ such that for all non-uniform polynomial-time algorithms A, $p(n) = \Pr_{x\in\{0,1\}^n}[f(A(1^n, f(x))) = f(x)] < 1 - \frac{1}{n^c}$.

That is, there exists a polynomial n^c such that any inversion algorithm fails to invert f on at least $\frac{1}{n^c}$ of the points.

The difference between weak and standard OWFs can be viewed graphically in the following way:

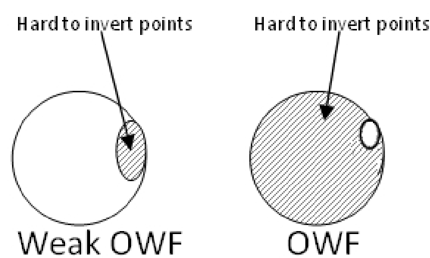

Fig 3. Difference between Weak OWF and OWF

This does not seem like a very useful definition cryptographically (since f can be inverted almost always). But:

- This definition still captures interesting examples (e.g. f_{mul} may not be strong OWF; however, if factoring products of two equal-length primes is hard then f_{mul} is weakly one-way).
- We will see that, given any weak OWF, it is possible to construct a strong OWF.

Theorem: Let $f:\{0,1\}^n \rightarrow \{0,1\}^*$ be a weakly OWF, and construct a function g in the following way: $g(x_1, x_2, ..x_t) = (f(x_1),..., f(x_t))$, where each x_i is n-bit long and $t(n) = nn^c$ (c is the hardness constant from the definition of weak OWF). Then g is strongly OWF.
Proof idea: In order to invert g, the adversary needs to invert f on many points (which are randomly chosen). Thus, if f has even a small (but still noticeable) fraction of "hard points" then inverting g would be hard almost always.

Making this simple idea work is non-trivial…

Attempt 1

Let A be an adversary that inverts f on at most $1 - \frac{1}{n^c}$ of the points. Then A inverts g on at most negligible fraction of the tuples $(f(x_1),..., f(x_t))$.

Proof: The x_i's are independent, thus Pr[A inverts all t instances]≤(Pr[A inverts one instance])^t $< (1-\frac{1}{n^c})^{nn^c} \approx \frac{1}{e^n}$.

■

But is it a correct argument?
The problem is that we assume here that A works by inverting the instances of f one by one, independently of each other. However, recall that A sees all the t instances at the same time, and then does some computation that potentially depends on all of the instances. So the independence assumption is not justified, even though the x_i's are independent.

The actual argument is more involved, and works by reduction (this is going to be a typical structure of a proof in cryptography):

Let $n \in N$, and let A' be the algorithm that inverts g on $\frac{1}{n^{c'}}$ fraction of the tuples $(x_1, x_2, ..x_t)$ for some c'>0. We construct an adversary A that inverts f on all points in $\{0,1\}^n$ except for $\frac{1}{n^c}$-fraction of them.

The rest of the proof will be shown in next lecture…