CS 183 Notes

Stephen Kelman
Rafi Ostrovsky, Eli Jaffe

April 9, 2022

1 Lecture 1: Intro and 1-Way Functions

1.1 A Brief Introduction to Cryptography

Originally, cryptography was something done in order to provide some level of privacy or secrecy. It was often used in war, with examples including Ancient Roman scalp "encryption", Caesar Ciphers, and more recently, the Enigma Machines of WWII. Many of these older methods are examples of "security by obscurity", which generally only works when the adversary doesn't know what method you are using. Once they know, reversing it to reveal the secret is often easy, or trivial.

More recently, there have been some important watershed moments in cryptography. For instance, in the 1970s, Diffie and Hellman introduced the idea of public-key encryption, which allows anyone to use a public key to encrypt a message, but only people with a corresponding private key are able to decrypt it.

The more important moment for this class came in the 1980s, when Micali and Blum introduced the idea that cryptographic systems could be **"provably secure"**. The idea of something being provably secure is that if the cryptosystem were to be broken, this would be equivalent to solving some other problem which is assumed to be difficult (usually an old, famous, unsolved problem from mathematics). So, one could begin with one of these difficult problems and design a cryptosystem based on creating a nontrivial instance of the problem. Then, the system would be provably secure in the sense that we can prove that breaking it (in the general case) is as difficult as solving the problem.

A proof of this type is called a **reduction**, since we "reduce" the difficult problem to our cryptosystem by showing that anything (i.e., some algorithm) that breaks our cryptosystem could then be used (i.e. as a subroutine) to solve the difficult problem. In particular, Micali and Blum showed via reduction that breaking their pseudo-random number generator would be equivalent to solving the discrete logarithm problem.

It's important to note that a reduction proves that a cryptosystem is secure *as long as the problem it is based on is actually as difficult as we think.* Of course, there's a long list of these difficult problems, and we can choose any of them, but that's where our job ends. If someone does break our cryptosystem, we can blame the centuries of other people who couldn't solve the hard problem and told us it was difficult.

1.2 Review of P/NP/NP-complete

There are a few important classes of problems (languages) to know:

P is the set of languages which are *computable* in polynomial time. That is, if $L \in \mathbf{P}$, then there exists a nonnegative integer c such that for any string x, we can solve whether or not $x \in L$ in time $O(|x|^c)$.

NP is the set of languages which are *verifiable* in polynomial time. That is, if $L \in \mathbf{NP}$, then there exists a nonnegative integer c such that for any $x \in L$, there exists a polynomial-size witness w, with which we can verify that $x \in L$ in time $O(|x|)^c$. If $x \notin L$, then no such witness exists.

In particular, there exists some "verifier" V for every language L in $\mathbf{NP}$. V takes as input some string x and some (polynomial size) "witness" w, and accepts if and only if w proves in some way that $x \in L$. If $x \in L$, then there exists some polynomial-size witness w for which $V(x, w)$ accepts. If $x \notin L$, then $V(x, w)$ rejects, for any witness w.

NP-complete is the set of languages within $\mathbf{NP}$ for which any solution can be used to solve any other problem in $\mathbf{NP}$. In particular, if $L \in \mathbf{NP}$-complete, then any A solving L in polynomial time could be used as a subroutine to solve L' in polynomial time, for any $L' \in \mathbf{NP}$.

Note: **NP** stands for "Nondeterministic Polynomial Time", which means that a nondeterministic Turing Machine would be able to decide the language in polynomial time, whereas P stands for polynomial time, meaning that a (deterministic) Turing Machine would be able to decide the language in polynomial time.

It is important to always remember that $\mathbf{P} \subset \mathbf{NP}$. For all problems in $\mathbf{NP}$, we either (1) know of a way to solve them in polynomial time, or (2) do not yet know of such a way. There are no problems in $\mathbf{NP}$ that we have shown *cannot be solved in polynomial time.* If one were to show this for some problem, it would prove that $\mathbf{P} \neq \mathbf{NP}$, which is currently a (very difficult) unsolved problem and is, in a sense, at the heart of this course.

1.3 Probabilistic Complexity Classes

Also important to us is the idea of randomness. We will come back to why it is so important briefly, but for now, we introduce a few sets of languages which are categorized probabilistically:

RP (Randomized Polytime) is the set of all languages that can be recognized by some machine M in polynomial time such that:

1. $Pr[M(x)$ accepts $|x \in L] > \frac{2}{3}$, and
2. $Pr[M(x)$ rejects $|x \notin L] = 1$.

Co-RP (Co- Randomized Polytime) is the set of all languages that can be recognized in polynomial time such that:

1. $Pr[M(x)$ accepts $|x \in L] = 1$, and
2. $Pr[M(x)$ rejects $|x \notin L] > \frac{2}{3}$.

PPT (Probabilistic Polynomial Time), also sometimes denoted as **BPP** (Bounded Probabilistic Poly-time), is the set of all languages that can be recognized in polynomial time such that:

1. $Pr[M(x)$ accepts $|x \in L] > \frac{2}{3}$, and
2. $Pr[M(x)$ rejects $|x \notin L] > \frac{2}{3}$.

Note that **RP** allows errors only on strings in the language, **Co-RP** allows errors only on strings not in the language (hence the "Co-"), and **PPT** allows errors on all strings. For this reason, we say that **RP** and **Co-RP** are "one-sided", or that they have "one-sided" error. Of course, this means that **PPT/BPP** has "two-sided" error.

Additionally, the choice of $\frac{2}{3}$ is completely arbitrary. For all of these definitions, it may be replaced by any real number strictly between $\frac{1}{2}$ and 1. As long as the probability is bounded away from $\frac{1}{2}$, we can "amplify" it through repeated trials so that the probability of correctness is as close to 1 as we would like. (Though, of course, we will never actually be able to achieve a probability of 1.)

1.4 "Hard" Problems and Randomness

As mentioned in the introduction, "hard" problems are at the center of cryptography. Essentially, we want problems of the following form:

1. They should be "easy" to create. So, when using them for cryptography, it should take a reasonable amount of time and power to encrypt our information.
2. They should have solutions, and with the right amount of information, they should not be difficult to solve or invert. So, if we want someone to be able to get our information, we may simply give them the necessary information and they should be able to decrypt it with a reasonable amount of time and power.
3. They should be **"hard"** to solve without the necessary information. So, if someone who we do not want to see our information somehow gets ahold of an encrypted method, it should take them an unreasonable, or infeasible amount of time and/or computational power to decrypt it.

One problem often regarded as "hard", and often given as an introductory example in cryptography is integer factorization. That is, given an integer N that is the product of two primes p, q, find p and q. While this is certainly difficult, and we have only shown to be solvable in $O(2^{\sqrt[3]{N \log\log(N)^2}})$, it isn't a great assumption for hardness. Steady improvement has been made on it over a long period of time, it is not NP-complete, and realization of large-scale quantum computing could make it much easier to solve.

So, this brings us to one of our main questions: **What is the minimal assumption we must make in order to build interesting cryptosystems?**

One answer to this question is: **the existence of One-way Functions (1WF, OWF)**.

The idea of one-way functions is simply to satisfy the criteria we listed above. That is, they are ***"easy"*** to compute and possible to invert but if someone doesn't have all the necessary information, they are ***"hard"*** to invert.

But what do we really mean by "easy" and "hard"? For "easy", it turns out to be enough to just say that we can consider something easy if we can do it in (probabilistic) polynomial time. Deciding when to call something "hard" is a bit more complicated.

At first glance, we might try declaring that something should be considered "hard" if we don't know some way for it to be solved in polynomial time (by a deterministic Turing Machine). But this isn't good enough, because it doesn't assume that our adversary is as strong as possible. The opponent should be allowed to use any available resources, especially *randomness*.

We single out randomness like this because it is often a very powerful computational tool. By relaxing how strict we are about what it means to solve a problem from being right 100% of the time to something close, like 99%, the problem can become much easier to solve (i.e., determining primality can be done very quickly with a probabilistic algorithm). In general, we can trade a little bit of accuracy for a great deal of efficiency.

So, we make another attempt at formulating the idea of "hardness" in terms of a game. We have two players, a challenger CH and an adversary ADV. The game is for ADV to try to invert the function f, and is played as follows:

0. There is a shared function f, and $k \in \mathbb{N}$ is defined to be some measure of the computational power of ADV.
1. CH randomly generates a string x with $|x|$ being polynomial in k.
2. CH computes $f(x)$ and sends this value to ADV.
3. ADV does some (probabilistic) polynomial time computation and computes some x' from $f(x)$.
4. ADV sends x' to CH.
5. If $f(x') = f(x)$, then ADV wins. Otherwise, CH wins.

Now, we can define the "hardness" of inverting f based on this game and who wins. Since we allow probabilistic play, we can't just say that ADV should *never* get a correct value, since they could just guess until they get it right by chance.

We need to consider what might happen if ADV just guesses at x (or some winning value of x'), or flips coins and uses a probabilistic algorithm of some sort. Certainly, if ADV really just got lucky, we shouldn't really count it. But, if their guessing seems to be very successful, we should say that they're onto something, and f is probably not that hard to invert.

So, we can say that **f is "hard" to invert if ADV has a "negligible" probability of winning this game**. We specify a formal definition of "negligible", and our definition of "hardness" in our final formal definition of a 1WF:

A function f is a **One-Way Function** if:

1. f is polynomial-time computable
2. f is "hard" to invert in the following sense:

$$\forall c \in \mathbb{N}, A \in \textbf{PPT}, \exists N_c \in \mathbb{N} \text{ such that}$$

$$\forall x \text{ randomly generated with } |x| = n > N_c,$$

$$Pr[A(f(x)) = x' \mid f(x') = f(x)] < \frac{1}{n^c}$$

Note that N_c doesn't seem necessary from what we have done so far, but we include it so that we can avoid the possibility of an adversary pre-computing a large lookup table and then just using that to invert f.

As we have been discussing, the final statement involves probability because ADV may employ guessing and randomness in their algorithm.

1.5 Aside: Correlated Randomness (CR)

The idea of correlated randomness is to have a lot of separate random strings/variables/values, generally distributed among different parties, such that the random strings are related in some way (i.e., they could combine to form a key or secret of some sort). Similar to one-way functions, Correlated Randomness can be used to build a lot of modern cryptography. Ostrovsky mentions CR as an alternative to 1WF's, but in a practical sense, the most difficult part of CR is generating it, for which we usually use 1WF's anyways.

It's more often seen as an alternative in a proof sense, as 1WF's can often be tricky to work with directly when proving something about a cryptosystem, but CR may be easier to use to achieve the same goal.

2 Lecture 2: Introduction to Hard-Core Bits

2.1 Coin-flipping Protocol

Let's say two separate (distant) parties A and B want to settle a dispute using a coin flip. How can either ensure that the other is giving truthfully random results (i.e. not manipulating the data to win), or that any falsification cannot change the output to favor one side over the other in a predictable way? The solution to this issue is to use something called a **"commitment protocol"**.

The idea of a commitment protocol is to force the parties to be honest. Or, more realistically, it is to force the parties to make and finalize their decisions before it is possible for them to know anything about what the other party is doing or has done. A commitment protocol involves one party "committing" some information, like a bit, which can be revealed, or "opened", at some later time. Essentially, a commitment protocol is like locking something in an ordinary safe. It must have two key properties:

1. A **hiding property**: Suppose A commits some information using the protocol, and then passes the committed form of this information to B. There is no (reasonable/feasible/"easy") way for B to figure out what it is that A committed using only the committed form of the information.

   That is, the information is "hidden", as if A had locked it in a safe and gave the safe to B, without the combination.

2. A **binding property**: Once A commits some information and passes the committed form of this information to B, there is no way for A to manipulate the information in any way.

   That is, the information is "bound" to a value once it has been committed and sent, as if A had locked an ordinary safe, and B did not allow A to reopen the safe and change its contents.

Now, we can use a committment protocol to establish a fair way for A and B to flip a coin remotely:

1. A flips a coin and commits the value, b_1.
2. A sends this committed value, $com(b_1)$ to B. This is binding.
3. B flips a coin, b_2, and sends the value directly to A. Note that this step does not involve any sort of committment protocol in any way.
4. A "opens", or tells B how to "open" the committed value to reveal b_1
5. The two parties compute a shared coin, $b_1 \oplus b_2$.

So how do we know this is fair/safe?

Well, assuming that our committment protocol has the desired properties, suppose that A cheats. This means that A somehow based the final value of b_1 on the value of b_2. But due to the binding property, the value of b_1 could not have been changed after step 2, which is before b_2 has even been decided.

Similarly, if B cheats, then B somehow knew the value of b_1 before sending b_2. But due to the hiding property, B could not have possibly known b_1 before step 4, and b_2 was finalized in step 3.

Thus, neither party could have possibly cheated, so the coin flip works as desired due to our commitment protocol!

2.2 Building a Commitment Protocol

Now, of course, we have to find some way to actually build a commitment protocol of some sort. We will do this by assuming the existence of, and then using, one-way functions. In fact, we assume something a little bit stronger: we assume that there exist **one-way permutations**, which are one-way functions that are bijective (for our purposes, we assume $|x| = |g(x)|$). This allows us to avoid having to worry about lookup tables based on the range of our function, since it will certainly be the same size as the domain.

Now, before we go any further, remember the definition of a one-way function. We really can only guarantee that it is "hard to invert" for very large x. In fact, since we are working with permutations, we certainly have that they must not be hard to invert for $|x| = 1$, since there are only two possible permutations on $\{0, 1\}$, so even guessing can yield a probability of 1/2.

But this presents a new problem, especially if we want to make a commitment protocol that we can use for a coin flip. We may have to have both parties randomly generate and commit strings that are thousands of bits long, but we only want each party to be able to use one bit (i.e., so they don't have some way to "change their minds" later on).

A quick, natural idea to solve this might be to agree to use the first bit of the random values. However, we quickly run into an issue here. Namely, it's not hard to create a one-way function that quietly "leaks" the first bit by placing it at a certain index in the output where B can read it and use it to cheat. The same goes for the second bit, third bit, etc., as well as any agreed-upon logical combination of the bits in x. So how do we deal with this? It seems like the main issue here is that we are allowing the one-way function to be created after we decide which bit(s) to use.

This brings up the following question: **Is it possible, given a one-way function (permutation) g, to design a predicate B so that for any string x, the value of B is difficult to predict if we are given only $g(x)$?.**

The answer to this question is $\underline{\text{yes}}$.

2.3 Hard-Core Bits (HCB)

A **hard-core bit** b of a 1WF g is some predicate for which it is as difficult to predict $b(x)$ from $g(x)$ as it is to invert $g(x)$.

In other words, any (probabilistic) polynomial-time adversary can only gain at most a negligible advantage (over the base probability of 1/2) in predicting $b(x)$ from $g(x)$.

More formally, a predicate b is a **hard-core bit** of a one-way function g if:

$$\forall c \in \mathbb{N}, A \in \mathbf{PPT}, \exists N_c \in \mathbb{N} \text{ such that}$$
$$\forall x \text{ randomly generated with } |x| = n > N_c,$$
$$Pr[A(f(x)) = b(x)] < \frac{1}{2} + 1n^c.$$

Suppose B is a hard-core bit of g, and some ADV predicts B from $g(x)$ with probability $\frac{1}{2} + \varepsilon$ where ε is non-negligible. Then, ADV can be used (as a subroutine of some other algorithm) to invert g in polynomial time with non-negligible probability (probability $> 1/(|x|^c)$ for some $c \in \mathbb{N}$).

This is our first look at a reduction. What we are basically saying here is that given a one-way function, inverting it could be reduced to predicting some hard-core bit.

Now, we prove an important theorem about hard-core bits and one-way functions:

2.3.1 (Goldreich, Levin) For any 1WF f, there exists a hardcore bit b of the form

$$b(x) = \langle p, x\rangle := \bigoplus_{i=1}^{|x|} p_i \cdot x_i,$$

where p is some "random" string with $|p| = |x|$.

This theorem is the explanation of the "yes" that we gave above. As long as we pick the 1WF f first, we can always be sure that there is some "random" p such that $b(x) = \langle p, x\rangle$ is a HCB. So, as long as we pick the 1WF before doing anything else, A doesn't have to worry about it secretly "leaking" the secret bit to B before B chooses/generates their own string/bit.

2.3.2 Proof

As with most reductions, we prove this by contradiction/contrapositive. Namely, we do the following:

Suppose, towards a contradiction, that there is some 1WF g for which there does not exist a hard-core bit b of the form $\langle p, x\rangle$. Then, there is some adversary $ADV \in \mathbf{PPT}$ which predicts $b = \langle p, x\rangle$ from $g(x)$, for all p, with probability $Pr = 1/2 + \varepsilon$, where ε is not negligible. Then, we may show that ADV can be used to invert g with non-negligible probability, which would contradict the definition of a 1WF, meaning that our assumption must have been false.

We do not prove the theorem in its entirety, but instead do two simplified versions: one here, where we assume that ADV can predict b with probability 1, and another in the next section, where we assume that ADV can predict b with probability $3/4 + \varepsilon$, where ε is not negligible.

(1) $Pr = 1$.

This one is quite easy, as we have asserted that $ADV(p, g(x)) = \langle p, x\rangle$ 100% of the time. So, if $p_k = 0^{k-1}10^{|x|-k}$, then we know with complete certainty that $ADV(p_k, g(x)) = \langle p_k, x\rangle = x_k$, so we can use each of $p[1], p[2], \ldots, p[|x|]$ to completely determine x from $g(x)$.

(2) $Pr = 3/4 + \varepsilon$.

This one is slightly more difficult since we have to deal with a little bit of uncertainty, but it is pretty much just as straightforward.

For any random string p, we define $p^i = p_1 \cdots p_{i-1}\overline{p_i}p_{i+1} \cdots p_{|x|}$. Then, we may use p^i_k and p_k in pairs for all i, k. In particular, we get $b_1 = ADV(g(x), p_k)$, $b_2 = ADV(g(x), p^i_k)$, from which we have $b_1 \oplus b_2 = x_i$ if both b_1 and b_2 are correct. Since ADV is correct with probability $3/4 + \varepsilon$, we have that b_1 and b_2 are both correct with probability:

$$1 - Pr[b_1 \text{ wrong}] - Pr[b_2 \text{ wrong}] = 1 - 2(1/4 - \varepsilon) = 1/2 + 2\varepsilon.$$

Then, we may "**amplify**" our probability of correctly finding x_i to be as close to 1 as we want (without ever quite getting there) by repeating some polynomial number of times and taking the value for x_i that appears the majority of the time, since we are correct with probability $1/2 + \varepsilon > 1/2$. In fact, as we repeat more times, the probability that the majority is the correct answer approaches 1 exponentially fast (due to the Chernoff Bound, from probability theory).

2.4 Building Bit Commitment from HCB

Now, we are ready to use HCB's to build a commitment protocol. Our two parties A and B follow these steps:

0. We begin with the following shared information: a security parameter k, a one-way function g, and a random string p with $|p| = n$ (which determines a hard-core bit $b(x) = \langle x, p \rangle$ for g.
1. A generates a random string x of length k
2. A generates a random bit b_1
3. A computes $com(b_1) := b_1 \oplus b(x)$
4. A computes $y = g(x)$
5. A commits b_1 by sending y and $com(b_1)$ to B
6. A may open the committed value by sending x to B
7. At this point, B may verify that A did not cheat by checking that $g(x) = y$

Now, we verify the two properties of a commitment protocol:

(1) Hiding: Since $b(x)$ is a hardcore bit, B will not be able to predict it from $g(x)$ with any non-negligible advantage, so B cannot figure out b_1.

(2) Binding: Since $g(x)$ is a one-way function, A will not be able to find a $x' \neq x$ such that $g(x') = g(x)$, as this would require inverting the one-way function. Clearly, A cannot change the values of $com(b_1)$ and y after sending them, and if they cannot find a different x' to make the results work in their favor, there's nothing else they can do once the values have been committed. That is, once A picks x and sends $y, com(b_1)$, the values are bound.

So, we have ourselves a working commitment protocol!

3 Lecture 3: Proof of Hard-Core Bits

4 Lecture 4: Pseudorandom Number Generators