CS 183 Notes

Stephen Kelman
Rafi Ostrovsky, Eli Jaffe

May 2, 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 **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 **NP** for which any solution can be used to solve any other problem in **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 **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 **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. $Pr[M(x) \text{ accepts } |x \in L] > \frac{2}{3}$, and
2. $Pr[M(x) \text{ rejects } |x \notin L] = 1$.

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

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

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

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.)

We cover amplification techniques in depth in lecture 3, in a slightly different context.

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:

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

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

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

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.