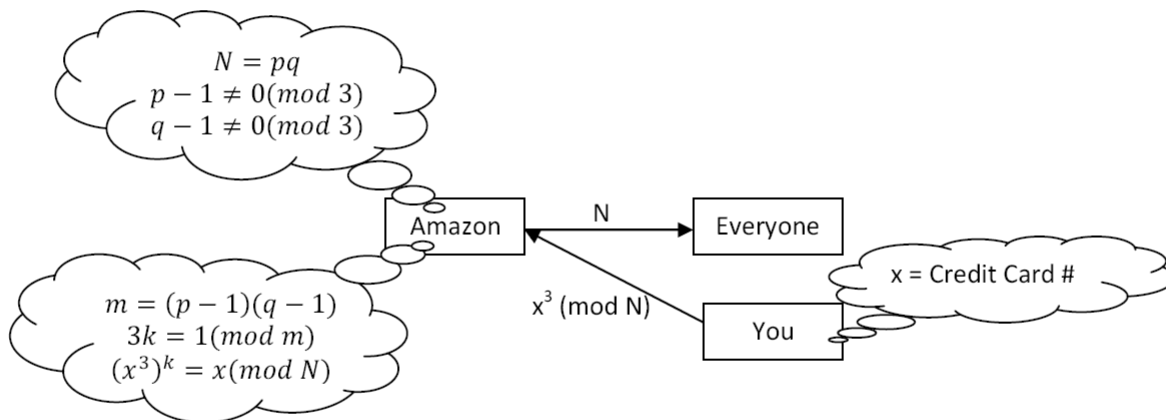
6.080/6.089 GITCS April 17, 2008

Lecture 18

Lecturer: Scott Aaronson *Scribe: Hristo Paskov*

1 Recap

Last time we talked about public key cryptography which falls in the realm of accomplishing bizarre social goals using number theory. Our first example of a public-key cryptosystem, in which two people exchanging messages did not have to meet beforehand, was Diffie-Hellman. We then talked about the RSA cryptosystem, which is probably the most widely used today. Here are the basics of how it works:



The first step is taken by the recipient of the message, by generating two giant prime numbers p and q and setting $N = pq$. Note that p and q must be chosen such that $p - 1$ and $q - 1$ are not divisible by 3. The recipient keeps p and q a closely-guarded secret, but gives out N to anyone who asks. Suppose a sender has a secret message x that she wants to send to the recipient. The sender calculates $x^3 \bmod N$ and sends it to the recipient. Now it's the recipient's turn to recover the message. He can use some number theory together with the fact that he knows p and q, the factors of N. The recipient first finds an integer k such that $3k = 1 \bmod (p - 1)(q - 1)$, which can be done in polynomial time via Euclid's algorithm, and then takes $(x^3)^k \bmod N = x^{3k} \bmod N = x$. The exponentiation can be done in polynomial time by using the trick of repeated squaring. Voila!

When you look at this procedure, you might wonder why are we cubing as opposed to raising to another power; is there anything special about 3? As it turns out, 3 is just the first choice that's convenient. Squaring would lead to a ciphertext that had multiple decryptions (corresponding to the multiple square roots $\bmod N$), while we want the decryption to be unique. Indeed, if we wanted the square root to be unique, then we'd need $p-1$ and $q-1$ to not divisible by 2, which is a problem since p and q (being large prime numbers) are odd!

You could, however, raise to a power higher than 3, and in fact that's what people usually do. If the other components of the cryptosystem—such as the padding out of messages with random garbage—aren't implemented properly, then there's a class of attacks called "small-exponent attacks" which break RSA with small exponents though not with large ones. On the other hand, if everything else is implemented properly, then as far as we know $x^3 \bmod N$ is already secure.

(Just like in biology, everything in cryptography is always more complicated than what you said, *whatever* you said. In particular, as soon as you leave a clean mathematical model and enter the real world, where code is buggy, hardware inadvertently leaks information, etc. etc., there's always further scope for paranoia. And cryptographers are extremely paranoid people.)

As mentioned in the last lecture, we know that a fast factoring algorithm would lead to a break of RSA. However, we don't know the opposite direction: could you break RSA without factoring? In 1979, Rabin showed that if you squared the plaintext x instead of cubing it, then recovering x *would* be as hard as factoring. But as discussed earlier, in that case you'd lose the property that every decryption is unique. This problem is what's prevented widespread adoption of Rabin's variant of RSA.

2 Trapdoor One-Way Functions

The operation $x^3 \bmod N$ in RSA is an example of what's called *trapdoor one way function*, or TDOWF. A trapdoor one-way function is a one-way function with the additional property that if you know some secret "trapdoor" information then you can efficiently invert it. So for example, the function $f(x) = x^3 \bmod N$ is believed to be a one-way function, yet is easy to invert by someone who knows the prime factors of N.

2.1 Different Classes of TDOWF's

Question from the floor: Are there any candidate TDOWF's *not* based on modular arithmetic (like RSA is)?

Answer: One class that's been the subject of much recent research is based on lattices. (Strictly speaking, the objects in this class are not TDOWF's, but something called *lossy TDOWF's*, but they still suffice for public-key encryption.) Part of the motivation for studying this class is that the cryptosystems based on modular arithmetic could all be broken by a quantum computer, if we had one. By contrast, even with a quantum computer we don't yet know how to break lattice cryptosystems. Right now, however, lattice cryptosystems are not used much. Part of the problem is that, while the message and key lengths are polynomial in n, there are large polynomial blowups. Thus, these cryptosystems aren't considered to be as practical as RSA. On the other hand, in recent years people have come up with better constructions, so it's becoming more practical.

There's also a third class of public-key cryptosystems based on elliptic curves, and elliptic-curve cryptography *is* currently practical. Like RSA, elliptic-curve cryptography is based on abelian groups, and like RSA it can be broken by a quantum computer. However, elliptic-curve cryptography has certain nice properties that are not known to be shared by RSA.

In summary, we only know of a few classes of candidate TDOWF's, and all of them are based on *some* sort of interesting math. When you ask for a trapdoor that makes your one-way function easy to invert again, you're really asking for something mathematically special. It almost seems like an accident that plausible candidates exist at all! By contrast, if you just want an ordinary, *non*-trapdoor OWF, then as far as we know, all sorts of "generic" computational processes that scramble up the input will work.

3 NP-completeness and Cryptography

An open problem for decades has been to base cryptography on an NP-complete problem. There are strong heuristic arguments, however, that suggest that if this is possible, it'll require very

different ideas from what we know today. One reason (discussed last time) is that cryptography requires average-case hardness rather than worst-case. A second reason is that many problems in cryptography actually belong to the class $NP \cap coNP$. For example, given an encrypted message, we could ask if the first bit of the plaintext is 1. If it is, then a short proof is to decrypt the message. If it's not, then a short proof is *also* to decrypt the message. However, problems in $NP \cap coNP$ can't be NP-complete under the most common reductions unless $NP = coNP$.

3.1 Impagliazzo's Five Worlds

A famous paper by Impagliazzo discusses five possible worlds of computational complexity and cryptography, corresponding to five different assumptions you can make. You don't need to remember the names of the worlds, but I thought you might enjoy seeing them.

1. Algorithmica - $P = NP$ or at the least fast probabalistic algorithms exist to solve all NP problems.

2. Heuristica - $P \neq NP$, but while NP problems are hard in the worst case, they are easy on average.

3. Pessiland - NP-complete problems are hard on average *but* one-way functions don't exist, hence no cryptography

4. Minicrypt - One-way functions exist (hence private-key cryptography, pseudorandom number generators, etc.), but there's no public-key cryptography

5. Cryptomania - Public-key cryptography exists; there are TDOWF's

The reigning belief is that we live in Cryptomania, or at the very least in Minicrypt.

4 Fun with Encryption

4.1 Message Authentication

Besides encrypting a message, can you prove that a message actually came from you? Think back to the one-time pad, the first decent cryptosystem we saw. On its face, the one-time pad seems to provide authentication as a side benefit. Recall that this system involves you and a friend sharing a secret key k, you transmitting a message x securely by sending $y = x \oplus k$, and your friend decoding the message by computing $x = y \oplus k$. Your friend might reason as follows: if it was anyone other than you who sent the message, then why would $y \oplus k$ yield an intelligible message as opposed to gobbledygook?

There are some holes in this argument (see if you can spot any), but the basic idea is sound. However, to accomplish this sort of authentication, you do need the other person to share a secret with you, in this case the key. It's like a secret handshake of fraternity brothers.

Going with the analogy of private vs. public key cryptography, we can ask whether there's such a thing public-key authentication. That is, if a person trusts that some public key N came from you, he or she should be able to trust any further message that you send as *also* coming from you. As a side benefit, RSA gives you this ability to authenticate yourself, but we won't go into the details.

4.2 Computer Scientists and Dating

Once you have cryptographic primitives like the RSA function, there are all sorts of games you can play. Take, for instance, the problem of Alice and Bob wanting to find out if they're both interested in dating each other. Being shy computer scientists, however, they should only find out they like each other if they're both interested; if one of them is *not* interested, then that one shouldn't be able to find out the other is interested.

An obvious solution (sometimes used in practice) would be to bring in a trusted mutual friend, Carl, but then Alice and Bob wold have to trust Carl not to spill the beans. Apparently there are websites out there that give this sort of functionality. However, ideally we would like to not have to rely on a third party.

Suggestion from the floor: Alice and Bob could face each other with their eyes closed, and each open their eyes only if they're interested.

Response: If neither one is interested, then there seems to be a termination problem! Also, we'd like a protocol that doesn't require physical proximity – remember that they're shy computer scientists!

4.2.1 The Dating Protocol

So let's suppose Alice and Bob are at their computers, just sending messages back and forth. If we make no assumptions about computational complexity, then the dating task is clearly impossible. Why? Intuitively it's "obvious": because eventually one of them will have to say something, without yet knowing whether his or her interest will be reciprocated or not! And indeed one can make this intuitive argument more formal.

So we're going to need a cryptographic assumption. In particular, let's assume RSA is secure. Let's also assume, for the time being, that Alice and Bob are what the cryptographers call *honest but curious.* In other words, we'll assume that they can both be trusted to follow the protocol correctly, but that they'll also try to gain as much information as possible from whatever messages they see. Later we'll see how to remove the honest-but-curious assumption, to get a protocol that's valid even if one player is trying to cheat.

Before we give the protocol, three further remarks might be in order. First, the very fact that Alice and Bob are carrying out a dating protocol in the first place, might be seen as *prima facie* evidence that they're interested! So you should imagine, if it helps, that Alice and Bob are at a singles party where *every* pair of people has to carry out the protocol. Second, it's an unavoidable feature of any protocol that if one player is interested and the other one isn't, then the one who's interested will learn that the other one isn't. (Why?) Third, it's also unavoidable that one player could *pretend* to be interested, and then after learning of the other player's interest, say "ha ha! I wasn't serious. Just wanted to know if you were interested."

In other words, we can't ask cryptography to solve the problem of heartbreak, or of people being jerks. All we can ask it to do is ensure that each player can't learn whether the other player has stated an interest in them, without stating interest themselves.

Without further ado, then, here's how Alice and Bob can solve the dating problem:

1. Alice goes through the standard procedure of picking two huge primes, p and q, such that $p-1$ and $q-1$ are not divisible by 3, and then taking $N = pq$. She keeps p and q secret, but sends Bob N together with $x^3 \bmod N$ and $y^3 \bmod N$ for some x and y. If she's not interested, then x and y are both 0 with random garbage padded onto them. If she *is* interested, then x is again 0 with random garbage, but y is 1 with random garbage.

2. Assuming RSA is secure, Bob (not knowing the prime factors of N) doesn't know how to take cube roots mod N efficiently, so $x^3 \bmod N$ and $y^3 \bmod N$ both look completely random to him. Bob does the following: he first picks a random integer r from 0 to $N-1$. Then, if he's not interested in Alice, he sends her $x^3 r^3 \bmod N$. If he *is* interested, he sends her $y^3 r^3 \bmod N$.

3. Alice takes the cube root of whatever number Bob sent. If Bob wasn't interested, this cube root will be $xr \bmod N$, while if he was interested it will be $yr \bmod N$. Either way, the outcome will look completely random to Alice, since she doesn't know r (which was chosen randomly). She then sends the cube root back to Bob.

4. Since Bob knows r, he can divide out r. We see that if Bob was not interested, he simply gets x which reveals nothing about Alice's interest. Otherwise he gets y which is 1 if and only if Alice is interested.

So there we have it. It seems that, at least in principle, computer scientists have solved the problem of flirting for shy people (assuming RSA is secure). This is truly nontrivial for computer scientists. However, this is just one example of what's called *secure multiparty computation*; a general theory to solve essentially all such problems was developed in the 1980's. So for example: suppose two people want to find out who makes more money, but without either of them learning anything else about the other's wealth. Or a group of friends want to know how much money they have *in total*, without any individual revealing her own amount. All of these problems, and many more, are known to be solvable cryptographically.

5 Zero-Knowledge Proofs

5.1 Motivation

In our dating protocol, we made the critical assumption that Alice and Bob were "honest but curious," i.e. they both followed the protocol correctly. We'd now like to move away from this assumption, and have the protocol work even if one of the players is cheating. (Naturally, if they're *both* cheating then there's nothing we can do.)

As discussed earlier, we're not concerned with the case where Bob pretends that he likes Alice just to find out whether she likes him. There's no cryptographic protocol that helps with Bob being a jerk, and we can only hope he'll get caught being one. Rather, the situation we're concerned with is when one of the players *looks* like they're following the protocol, but are actually just trying to find out the other player's interest.

What we need is for Alice and Bob to *prove* to each other at each step of the protocol that they're correctly following the protocol—i.e., sending whatever message they're supposed to send, given whether they're interested or not. The trouble is, they have to do this without *revealing* whether they're interested or not! Abstractly, then, the question is how it's possible to prove something to someone without revealing a crucial piece of information on which the proof is based.

5.2 History

Zero-knowledge proofs have been a major idea in cryptography since the 1980's. They were introduced by Goldwasser, Micali, and Rackoff in 1985. Interestingly, their paper was rejected multiple times before publication but is now one of the foundational papers of theoretical computer science.

5.3 Interactive Proofs

For thousands of years, the definition of a proof accepted by mathematicians was a sequence of logical deductions that could be shared with anyone to convince them of a mathematical truth. But developments in theoretical computer science over the last few decades have required generalizing the concept of proof, to *any sort of computational process or interaction* that can terminate a certain way only if the statement to be proven is true. Zero-knowledge proofs fall into the latter category, as we'll see next.

5.4 Simple Example: Graph Nonisomorphism

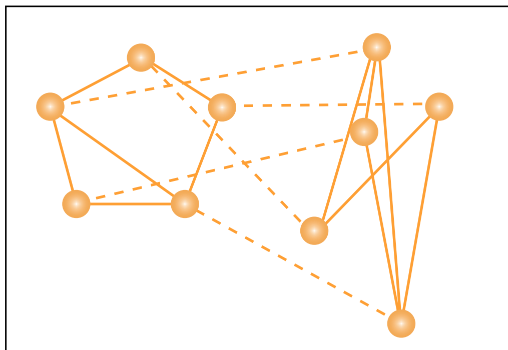


Figure by MIT OpenCourseWare.

To explain the concept of zero-knowledge proofs, it's easiest to start with a concrete example. The simplest example concerns the Graph Isomorphism problem. Here we're given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, which are defined by lists of their edges and vertices. The graphs are called *isomorphic* if there's a way to permute their vertices so that they are the same graph.

5.4.1 Complexity

It's clear that the Graph Isomorphism problem is in NP, since a short proof that G_1 and G_2 are isomorphic is just to specify the isomorphism (i.e., a mapping between the vertices of G_1 and G_2).

Is Graph Isomorphism in P? Is it NP-complete? We don't yet know the answer to either question, though we do have strong evidence that it isn't NP-complete. Specifically, we know that if Graph Isomorphism is NP-complete then $NP^{NP} = coNP^{NP}$, or "the polynomial hierarchy collapses" (proving this statement is beyond the scope of the course). Some computer scientists conjecture that Graph Isomorphism is intermediate between P and NP-complete, just as we believe Factoring to be. Others conjecture that Graph Isomorphism is in P, and we simply don't know enough about graphs yet to give an algorithm. (Note that we have efficient algorithms for Graph Isomorphism that work extremely well *in practice* – just not any that we can prove will work in all cases.)

As an amusing side note, it's said that the reason Levin wasn't the first to publish on NP-completeness is that he got stuck trying to show the Graph Isomorphism problem was NP-complete.

5.4.2 Proving No Isomorphism Exists

We said before that Graph Isomorphism is in NP. But is it in coNP? That is, can you give a short proof that two graphs are *not* isomorphic? Enumerating all the possibilities obviously won't work, since it's exponentially inefficient (there are $n!$ possible mappings). To this day, we don't know whether Graph Isomorphism is in coNP (though there are some deep recent results suggesting that it is).

Still, let's see an incredibly simple way that an all-knowing prover could convince a polynomial-time verifier that two graphs are not isomorphic. To illustrate, consider Coke and Pepsi. Suppose you claim that the two drinks are different but I maintain they're the same. How can you convince me you're right, short of giving me the chemical formula for both? By doing a blind taste test! If I blindfold you and you can reliably tell which is which, then you'll have convinced me that they must be different, even if I don't understand how.

The same idea applies to proving that G_1 and G_2 are not isomorphic. Suppose you're some wizard who has unlimited computational power, but the person you are trying to convince does not. The person can pick one of the two graphs at random and permute the vertices in a random way to form a new graph G', then send you G' and ask which graph she started with. If the graphs are indeed not isomorphic, then you'll be able to answer correctly every time, whereas if G_1 and G_2 *are* isomorphic, then your chance of guessing correctly will be at most $1/2$ (since a random permutation of G_1 is the same as a random permutation of G_2). If the verifier repeats this test 100 times and you answer correctly every time, then she can be sure to an extremely high confidence (namely $1 - 2^{-100}$) that the graphs are not isomorphic.

But notice something interesting: even though the verifier became convinced, she did so without gaining any new knowledge about G_1 and G_2 (by which, for example, she could convince someone else that they're not isomorphic)! In other words, if she'd merely trusted you, then she could have simulated her entire interaction with you on her own, without ever involving you at all. Any interactive proof system that has this property – that the prover only tells the verifier things that the latter "already knew" – is called a *zero-knowledge proof system*.

(Admittedly, it's only obvious that the verifier doesn't learn anything if she's "honest" – that is, if she follows the protocol correctly. Conceivably a *dishonest* verifier who violated the protocol could learn something she didn't know at the start. This is a distinction we'll see again later.)

5.5 The General Case

How can we extend this notion of a zero-knowledge proof to arbitrary problems, besides Graph Isomorphism? For example, suppose that you've proven the Riemann Hypothesis, but are paranoid and do not want anyone else to know your proof just yet. That might sound silly, but it's essentially how mathematicians worked in the Middle Ages: each knew how to solve some equation but didn't want to divulge the general method for solving it to competitors.

So suppose you have a proof of some arbitrary statement, and you want to convince people you have a proof without divulging any of the details. It turns out that there's a way to convert *any* mathematical proof into zero-knowledge form; what's more, the conversion can even be done in polynomial time. However, we'll need to make cryptographic assumptions.

5.6 Goldreich-Micali-Wigderson

In what follows, we'll assume that your proof is written out in machine-checkable form, in some formal system like Zermelo-Fraenkel set theory. We know that THEOREM, the problem of proving a theorem in at most n symbols, is an NP-complete problem, and is therefore efficiently reducible to any other NP-complete problem. Thus, we just need to find *some* NP-complete problem for which we can prove that we have a solution, without divulging the solution. Out of the thousands of known NP-complete problems, it turns out that the most convenient for our purpose will be the problem of 3-coloring a graph.

5.6.1 3-Coloring Proof

Suppose we have a 3-coloring of a graph and we want to prove that we have this 3-coloring without divulging it. Also, suppose that for each vertex of the graph, there's a magical box in which we can store the color of that vertex. What makes these boxes magical is that we can open them but the verifier can't. The key point is that, by storing colors in the boxes, we can "commit" to them: that is, we can assure the verifier that we've picked the color of each vertex beforehand, and are not just making them up based on which questions she asks.

Using these boxes, we can run the following protocol:

1. Start with a 3-coloring of the graph; then randomly permute the colors of the vertices. There are $3! = 6$ ways to permute the colors. For example, red⇒green, green⇒red, blue stays the same.

2. Write the color of each vertex on a slip of paper and place it in the magic box that's labeled with that vertex's number. Give all of the magic boxes to the verifier.

3. Let the challenger pick any two neighboring vertices, and open the boxes corresponding to those vertices.

4. Throw away the boxes and repeat the whole protocol as many times as desired.

If we really have a 3-coloring of the graph, then the verifier will see two different colors every time she chooses two neighboring vertices. On the other hand, suppose we were lying and didn't have a 3-coloring. Then eventually the verifier will find a conflict. Note that there are $O(n^2)$ edges, where n is the number of vertices of the graph. Therefore, since we commit to the colors in advance, there's a $\Omega(1/n^2)$ chance of catching us if we were lying. By repeating the whole protocol, say, n^3 times, the verifier can boost the probability of catching a lie exponentially close to 1, and can therefore (assuming everything checks out) become exponentially confident that we were telling the truth.

On the other hand, since we permute the colors randomly and reshuffle every time, the verifier learns nothing about the actual 3-coloring; she just sees two different random colors every time and thereby gains no knowledge!

Of course, the whole protocol relied on the existence of "magic boxes." So what if we don't have the magic boxes available? Is there any way we could *simulate* their functionality, if we were just sending messages back and forth over the Internet?

Yes: using cryptography! Instead of locking each vertex's color in a box, we can *encrypt* each color and send the verifier the encrypted messages. Then, when the verifier picks two adjacent vertices and asks us for their colors, we can decrypt the corresponding messages (though not the encrypted messages for any other vertices). For this to work, we just need to ensure two things:

1. A polynomial-time verifier shouldn't be able to learn *anything* from the encrypted messages. In particular, this means that even if two vertices are colored the same, the corresponding encrypted messages should look completely different. Fortunately, this is easy to arrange, for example by padding out the color data with random garbage prior to encrypting it.

2. When, in the last step, we decrypt two chosen messages, we should be able to *prove* to the verifier that the messages were decrypted correctly. In other words, every encrypted message should have one and only one decryption. As discussed earlier, the most popular public-key cryptosystems, like RSA, satisfy this property by construction. But even with more

"generic" cryptosystems (based on arbitrary one-way functions), it's known how to simulate the unique-decryption property by adding in more rounds of communication.

5.6.2 Back to Dating

Recall our original goal in discussing zero-knowledge: we wanted to make the dating protocol work correctly, even if Alice or Bob might be cheating. How can we do that? Well, first have Alice and Bob send each other encrypted messages that encode whether or not they're interested in each other, as well as their secret numbers p, q, and r. Then have them follow the dating protocol exactly as before, but with one addition: *at each step, a player not only sends the message that's called for in the protocol, but also provides a zero-knowledge proof that that's exactly the message they were supposed to send—given the sequence of previous messages, whether or not they're interested, and* p, q, r. Note that this is possible, since decrypting all the encrypted messages and verifying that the protocol is being followed correctly is an NP problem, which is therefore reducible to SAT and thence to 3-Coloring. And by definition, a zero-knowledge proof leaks no information about Alice and Bob's private information, so the protocol remains secure.

To clarify one point, it's not known how to implement this dating protocol using an arbitrary OWF—only how to implement the GMW part of it (the part that makes the protocol secure against a cheating Alice or Bob). To implement the protocol itself, we seem to need a stronger assumption, like the security of RSA or something similar. (Indeed, it's not even known how to implement the dating protocol using an arbitrary *trapdoor* OWF, although if we know further that the trapdoor OWF is a permutation, then it's possible.)

MIT OpenCourseWare
http://ocw.mit.edu

6.045J / 18.400J Automata, Computability, and Complexity
Spring 2011

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.