CS174 Spr 99 Lecture 27 Summary John Canny

Bit Commitment

Bit commitment is a way to store away a secret in a box so that you can't change the secret, but others can't find out the secret until you unlock it with a key. We already saw an example of this last time in the zero-knowledge proof of graph coloring. Each time you recolored the graph, you covered the vertices with "gray paint". That means you committed those bits. Another example would be if you wanted to make some predictions (e.g. on the stock market) and then prove later that you had been right. You dont want people to see your predictions until you're ready, but you have to be able to convince them that you didnt change your predictions when you saw what actually happened.

Bit commitment can be done in a few different ways. The simplest way is to use a one-way function like a secure hash function (MD5 or SHA). Take the message M, add some random bits R (to prevent someone from guessing and verifying your secret) and then compute $S = H(M, R)$. You publish S somewhere and tell people about it. People can save a copy of S and note when they received it, and you can also get S time stamped as described below. Because the hash function is secure, they can't invert it to get (M, R). And because you added the random bits, there are too many possible (M, R) for people to enumerate and plug into H.

Then when you want to reveal your bits, you simply publish (M, R). Anyone can check that $H(M, R) = S$. Because secure hash functions prevent you from finding another (M', R') such that $H(M', R') = S$, people know that you didnt cheat by picking another M' after you published S.

Kerberos

Kerberos is a system for allowing users on an insecure network to identify themselves to an application program (that might cause damage if the wrong person used it). Kerberos is something like a zero-knowledge proof system. The idea is to convince the application program that you know a secret piece of data (your Kerberos password) without divulging the password. The added complication here is that the prover and verifier are connected by an insecure network.

In fact there are three actors in a Kerberos transaction:

Authentication Server (AS) This server is known and trusted by both the user and the verifier.

The User (U) Is the actor trying to use a service (application program) on the verifier.

The Verifier (V) Is the actor which checks the identity of the user. Normally is a service that the user wants to run.

Note that while both user and verifier trust the AS, they can't trust the packets that come over the network from the AS because the network is not secure.

Basic Kerberos

Initially, both the user and verifier have secret passwords P_U and P_V, which are known also to the AS. That is how the AS knows and checks the identities of those actors. The AS cannot authenticate any actors that it is not "acquainted" with in this way. These passwords must never pass unencrypted over the network.

For the user to use a "kerberized" service on the verifier, the following sequence of events happens. Note that the description uses the metaphor of "locks", "boxes" and "keys". In reality, a key is an encryption key or password, a box is just a string, locking the box with a key is encrypting with that key, and unlocking is decrypting. Kerberos uses the secret-key protocol DES, meaning that keys must not be sent plain across the network.

1. First the user sends a message to the AS: "I, J Random User, would like to talk to Foo service."

2. When the AS receives this message, it chooses a new random key called the session key k_{UV}. In Box 1 it puts k_{UV} and the string "Foo Server". It locks this box with the user's key k_U which is the hash of the user's password. In a Box 2, it puts k_{UV} and the string "J Random User". It locks this box with the services's key k_V which is the hash of V's password. It returns both boxes to the user.

3. The user unlocks Box 1 with his key k_U, extracting k_{UV} and the string "Foo service". The user can't open Box 2 (since it's locked with the service's password). Instead, he puts the current time in Box 3 and locks it with the session key k_{UV}. He then hands Boxes 2 and 3 to the verifier (service).

4. The verifier opens Box 2 with its own key, extracting the session key k_{UV} and the string "J Random User". It then opens Box 3 with k_{UV} and extracts the current time. These items demonstrate the identity of the user.

Notice that after the protocol has run, both the user and the verifier have the secret session key k_{UV}. Since this is known only to them, they can exchange secret messages if needed.

In Kerberos jargon, Box 2 is called a *ticket* and Box 3 is called an *authenticator*. The protocol is fairly secure. The encryption of all messages ensures that any other actor that changes a message on the network can't do any damage. Changing an encrypted message will only make it unintelligible. That will disrupt the transaction, but not allow illegal access.

The other way a hacker could cause damage is by intercepting a message and saving it for their own use (e.g. by impersonating the IP address of the user's machine). But Box 3 guards against this. In order to interact with the user, the verifier expects to get a Box 3 with a time stamp that is

not to old. If a hacker intercepted Box 3 and tried to impersonate the user, they would have to do so almost immediately, because if they stopped Box 3 and forwarded it later, the verifier would not accept it because it had been excessively delayed (which indicates network failure or a hacker). If the hacker tries to impersonate the user immediately, the user will be interacting with the service as well, and is likely to discover that the hacker is impersonating them.

Secret Sharing

Another basic mechanism in cryptography is secret-sharing. In secret-sharing, there is a secret message M that is split among k distinct people. Each person gets a string M_i which has the same length as the message M but which is indistinguishable from a random string. To get the message back, all k people must combine their messages M_i. With only $k-1$ of the messages M_i, it is impossible to get any information about the message M.

A simple form of secret sharing is the following: Let M be a message, and let p be a prime greater than M. Choose $k-1$ random numbers r_i for $i = 1, \ldots, k-1$ which are uniformly chosen from $\{0, \ldots, p-1\}$. Choose r_k such that $r_1 + \cdots + r_k = 0 (\mathrm{mod}\ p)$. The distribution of r_k is uniform as well. Any $k-1$ of the r_i's are independent.

Now define $M_i = r_i$ for $i = 1, \ldots, k-1$, and $M_k = M + r_k (\mathrm{mod}\ p)$. Now all of the M_k have uniform distributions, and any $k-1$ of them are independent. But the sum of all of the M_i's for $i = 1, \ldots, k$ is M. Thus the secret M is perfectly shared among the k partipants. All k of them must cooperate for the secret to be revealed. There is no information in any $k-1$ of the messages M_i.

One important property of secret-sharing is that it is based on information theory rather than cryptography. That means it can never be broken if either (i) the cryptographic scheme is broken (which is possible for RSA or DES) (ii) computers become fast enough to crack the normal-size keys. It doesnt matter how good computers or algorithms get, there is simply not enough information in shared secrets to construct the message unless you have all the parts.

Digital Timestamping

Copyright and Patents are important classical ways of protecting information. But they are (relatively) slow and cumbersome to create, and even more difficult to defend. Digital timestamps are a relatively simple means of instantly marking the authoring date of a document, which can show its priority (that it existed before other related documents).

1. You the user compute the hash $H(M)$ of your document M using e.g. MD5, and send it to the timestamping service over the network.

2. The timestamping service receives $H(M)$ (and probably a digital cash fee) from you. It signs

the pair $(T, H(M))$ where T is the current time, and returns $(T, H(M), S)$ to you, where S is its signature.

3. To prove the time of the document at some later date, you produce the sequence (M, T, S). Any other actor can verify (using H) that the timestamp service signed that document at time T, which shows in particular that the document existed then.

Notice that your document is hidden by the hash function when it is sent to the timestamping service. So neither the service, nor any other actor can copy it. Once you get the timestamp from the service, you are well-protected. Even if someone else copies your document later, or even if you reveal it yourself, no-one else will be able to prove an earlier authoring date.