CMSC/Math 456: Cryptography (Fall 2022)

Lecture 19

Daniel Gottesman

Administrative

Mid-term exam grades are almost ready …

It looks like the median score will be in the low 60s.

Problem set #6 (a programming assignment) will be out later today, due next Thursday at *midnight*.

Important reminder: You can collaborate with other students, but you must

- Write your own code! You can share an algorithm but you shouldn't be looking at someone else's solution when you write your code.
- Put a comment in the code saying who you collaborated with.

This class is being recorded

Hash Functions

A hash function $H(x)$ maps the input x to a shorter string. The main cryptographic property of hash functions is collision resistance: it is hard to find a pair x, x' such that $H(x) = H(x')$.

> But sometimes we also want to consider weaker or stronger properties of hash functions. One such property is that sometimes we abstract a hash function into a random oracle.

A random oracle is just a random function to which we have only black-box access.

> When we say a hash function can be modeled as a random oracle, we are saying that it has no exploitable structure.

Hash Functions for MACs

Uses the property of collision resistance:

Given a hash function H(x) and a fixed-size MAC Mac(k,m), we can make a new MAC:

$$\mathrm{Mac}'(k,m) = (H(m), \mathrm{Mac}(k, H(m)))$$

This lets us efficiently authenticate long messages with short tags and keys.

> If Eve can't find a collision, to forge a new message she will have to produce a new value of H(m), which in turn requires a new tag.

Hash Functions for Key Derivation

Uses random oracle model:

Suppose Alice and Bob do a public key protocol or have some other method to derive a key k which is not uniformly distributed. Or perhaps Eve has learned partial information about the key k (so from Eve's point of view, k is no longer uniformly random).

> If H is a random oracle, then H(k) is close to uniformly random as long as k has a significant random element to its distribution.
>
> Each possible value of k gives an uncorrelated value of H(k). We can measure randomness via entropy (in this case, min-entropy). The random oracle preserves the entropy but concentrates it into fewer bits.

Hash Functions for Fingerprinting

Uses collision resistance:

Given a set of n files A_i each of length L, how can we quickly determine if $A_i = A_j$ for any pairs (i,j) and find any such pairs?

> Directly comparing every character between each pair of files would take time $O(n^2L)$.

Instead we can calculate a hash $h_i = H(A_i)$ of each file and then compare the hash values h_i.

> This takes time $O(n^2s + L)$ when $|h_i| = s$. (And s needs to be at least $\log n$.)

This algorithm fails only if two different documents A_i and A_j have the same hash value $H(A_i) = H(A_j)$ — which would be a collision in the hash function.

Applications of Fingerprinting

File fingerprinting has many applications:

- Data deduplication: Identify extra copies of a file already stored in the system.
- Virus scanning: Identify if newly received files match any known malware.
- Detect copyright violation: Identify exact copies from a list of copyrighted works.
- Track sensitive information: Keep track of files containing sensitive information such as medical records to be sure they aren’t accidentally sent somewhere insecure.

Hash Functions for Password Files

Uses random oracle model:

When you enter a password into a computer, how does the computer know if the password is correct or not?

The computer stores a list of everyone's passwords.

But if a hacker gets access to this list, everyone's password is compromised.

Password files are a prime hacker target.

Instead, store hashes of the passwords. This:

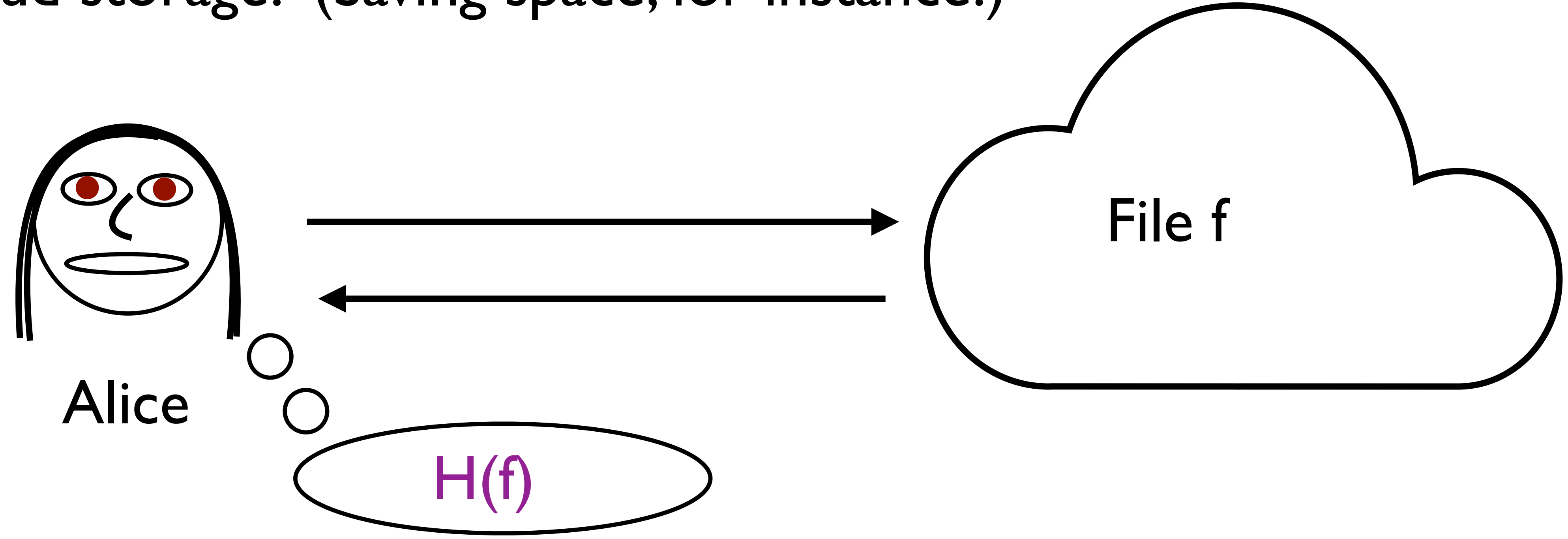
- Is more efficient
- Conceals the file contents unless the attacker can invert the hash function

Passwords are normally hashed with a unique random salt to foil preprocessing of common passwords.

Hash Functions for Verifying Files

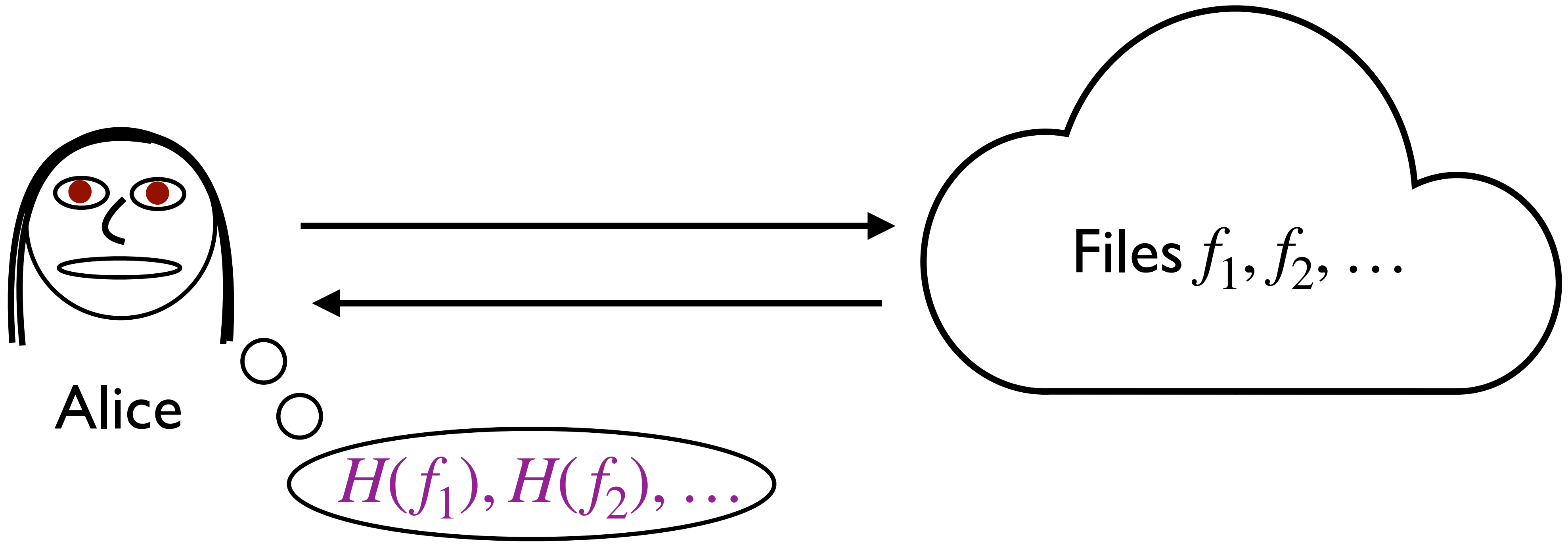
Suppose you want to store some files on a cloud server but you want to be able to verify that the files haven't been corrupted when you retrieve them.

Storing the original file would remove the point of using the cloud storage. (Saving space, for instance.)



But Alice can locally store a fingerprint H(f), which is much shorter, and verify the file easily once it is retrieved.

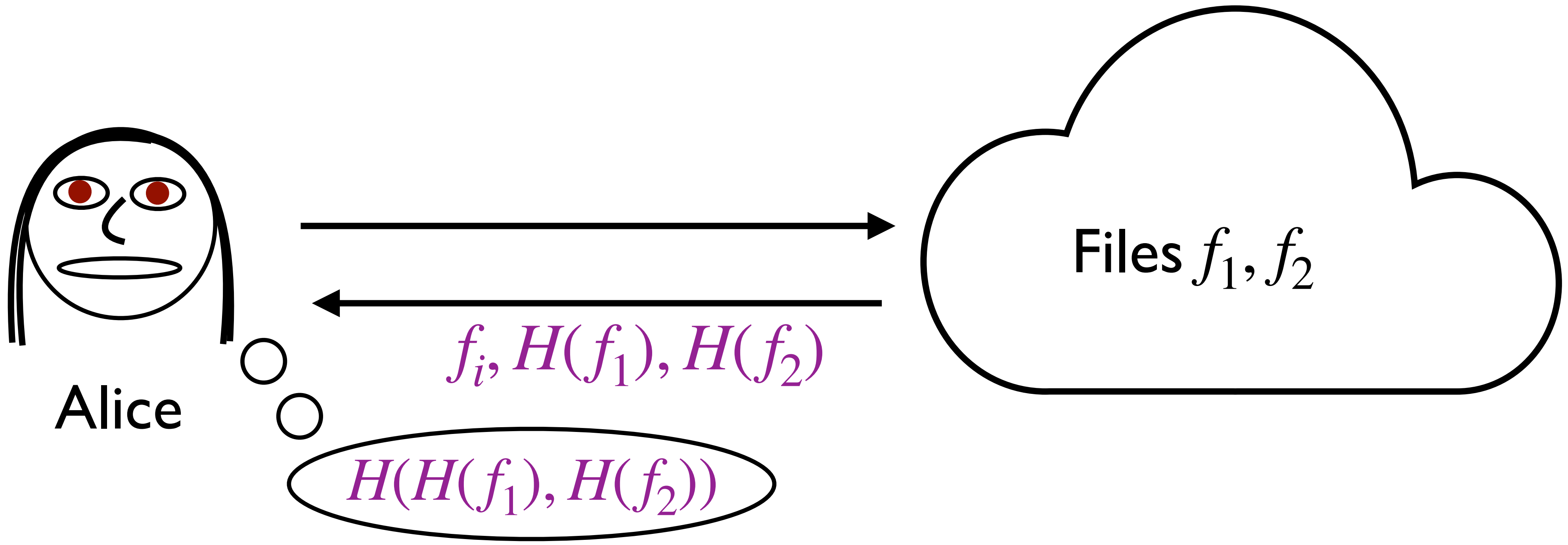
Storing Many Files



But what if Alice wants to store many files on the cloud and have the ability to check any of them?

She could store the fingerprint of each file, but that starts to get large. If there are n files, she would be storing O(n) bits.

Storing Two Files



If Alice wants to store just 2 files on the cloud, she can save space by keeping just $H(H(f_1), H(f_2))$. When she retrieves a file f_i, she can also ask the cloud for hashes $H(f_1), H(f_2)$ and whichever file she wanted.

To verify, Alice computes $H(f_i)$ and verifies the hash value. Then she also computes $H(H(f_1), H(f_2))$.

Storing Many Files Efficiently

Now suppose Alice is storing n files on the cloud.

$$\begin{tikzcd}[column sep=tiny]
 & & & & & & & h_{1\ldots 8} = H(h_{1234}, h_{5678}) \arrow[dllll, dash] \arrow[drrrr, dash] & & & & & & & \\
 & & & h_{1234} = H(h_{12}, h_{34}) \arrow[dll, dash] \arrow[drr, dash] & & & & & & & & h_{5678} = H(h_{56}, h_{78}) \arrow[dll, dash] \arrow[drr, dash] & & & \\
 & h_{12} = H(h_1, h_2) \arrow[dl, dash] \arrow[dr, dash] & & & & h_{34} = H(h_3, h_4) \arrow[dl, dash] \arrow[dr, dash] & & & & h_{56} = H(h_5, h_6) \arrow[dl, dash] \arrow[dr, dash] & & & & h_{78} = H(h_7, h_8) \arrow[dl, dash] \arrow[dr, dash] & \\
h_1 & & h_2 & & h_3 & & h_4 & & h_5 & & h_6 & & h_7 & & h_8
\end{tikzcd}$$

$$h_i = H(f_i)$$

She can initially compute hashes in a tree structure and store only a single hash, the root of the tree ($h_{1\ldots 8}$ in the example).

Retrieving One of Many Files

$$\begin{tikzcd}
 & & h_{1\ldots 8} = H(h_{1234}, h_{5678}) \arrow[dl, dash] \arrow[dr, dash] & \\
 & h_{1234} = H(h_{12}, h_{34}) \arrow[dl, dash] \arrow[dr, dash] & & h_{5678} = H(h_{56}, h_{78}) \\
h_{12} = H(h_1, h_2) & & h_{34} = H(h_3, h_4) \arrow[dl, dash] \arrow[dr, dash] & \\
 & h_3 & & h_4
\end{tikzcd}$$

When Alice wishes to retrieve a file, she asks for the file f_i and the cloud server returns it, along with $h_i = h(f_i)$, and all of hashes above it in the tree. For each of those hashes, the cloud server also returns the other child so that Alice can verify all the needed hashes.

File Storage Summary

This method is known as a Merkle tree.

Alice only needs to store one hash value.

When she retrieves a single file, she receives an additional $O(\log n)$ hash values.

This compares to $O(n)$ stored hash values if she tries to store them all.

Note that this is a different problem from a MAC in two ways:

- Alice doesn't have to worry above the authenticity of the hash that she keeps, so she doesn't need to authenticate it.
- Alice is only concerned about verifying *part* of the full set of stored files rather than the whole set.

Multiparty Computation

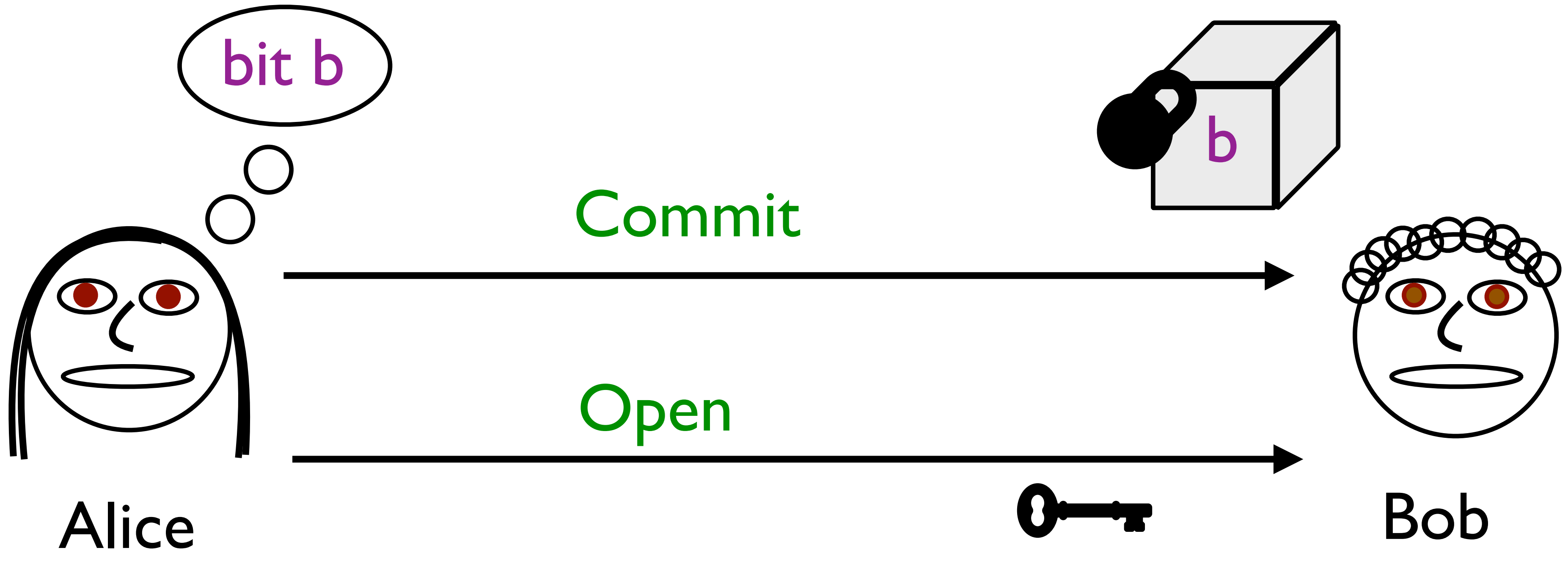
A different type of cryptographic protocol is a secure multiparty computation, in which two or more people are trying to perform some computational task but don't trust each other.

> This differs from the situation in the communication protocols we have seen so far in that the adversary controls one (or more) of the expected users of the protocol.

There are a wide variety of multiparty computation protocols. An example includes zero-knowledge proofs, a method by which it is possible to convince someone of a fact without revealing any information about the proof itself.

Bit Commitment

A useful cryptographic primitive for multiparty computations is bit commitment.



In the Commit phase, Alice sends Bob a bit b encoded in some way, in a virtual lockbox which Bob cannot open to learn b.

In the Open phase, Alice sends Bob the virtual key to open the lockbox, revealing b. Alice should not be able to change b from the value she originally put in the lockbox.

Hash Functions for Commitment

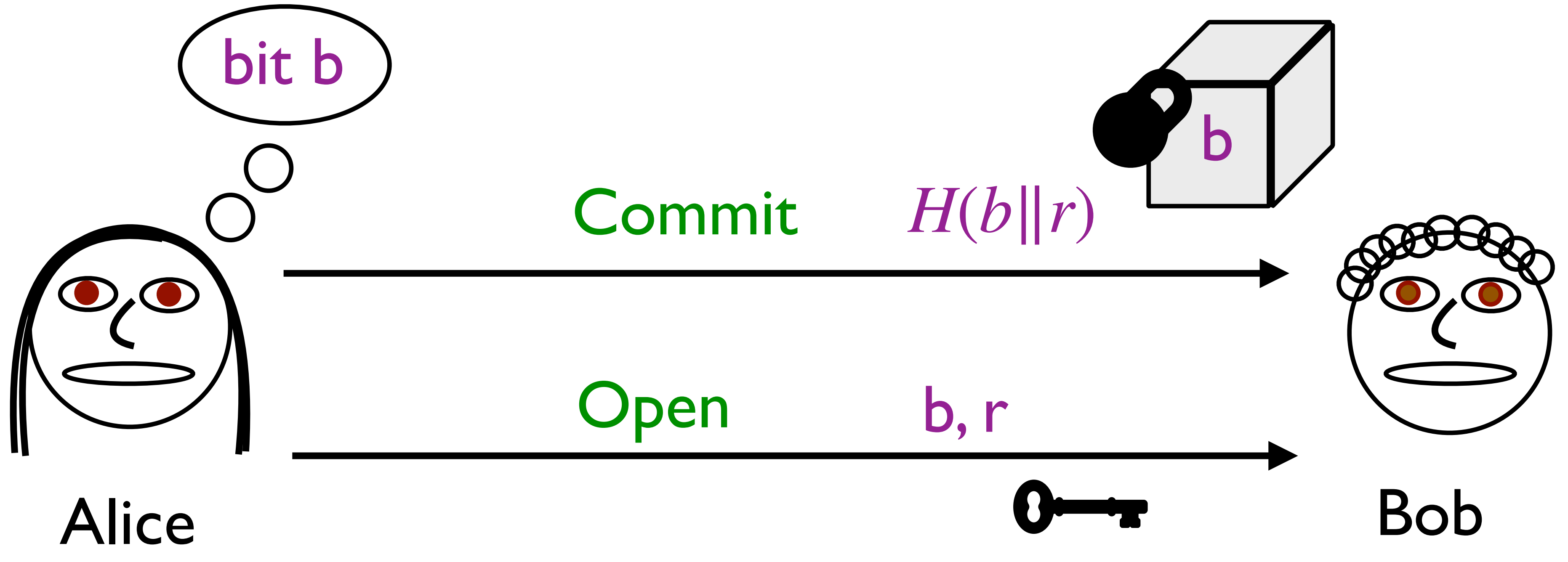
Commit Phase: Alice chooses random r and then sends Bob $H(b\|r)$ to commit to the bit b.

Open Phase: Alice sends to Bob b and r. Then Bob computes $H(b\|r)$ to verify the commitment.

The security requires that Bob not be able to determine b before the open phase. (The protocol is hiding or concealing.) Note that it is possible that the commitment is *never* opened, and it should continue to be binding indefinitely.

Security also requires that Alice not be able to open the commitment to more than one possible value. (The protocol is binding.)

Security of Commitment with Hash



The hiding property follows if Bob is unable to invert the hash function to learn b. This is certainly true if the hash function is modeled as a random oracle. (But you only need it to be a one-way function.)

The binding property follows from the collision resistance of the hash function: To open the commitment with a different b, Alice would need to find (b', r') with $H(b, r) = H(b', r')$

This class is being recorded