Recitation 6: Hash functions

6.5610, Spring 2023

March 17, 2023

1 Review of material

- In cryptography, a hash function H is a deterministic, efficient, and public function that takes inputs in $\{0,1\}^*$ to outputs in $\{0,1\}^k$, for some k.
- In practice, we use SHA256 or SHA3 as hash functions.
- We care about multiple useful properties of hash functions:
  1. **One-wayness**: Informally, given a random output, it is hard to find an input to H that produced this output.
  2. **Collision resistance**: Informally, it is computationally hard to find two inputs $x_1 \neq x_2$ such that $H(x_1) = H(x_2)$.
  3. **Target collision resistance**: Informally, it is computationally hard to find an input x' such that $H(x') = H(x)$, for an adversarially chosen x (chosen *before* the hash function has been fixed).
  4. **Random oracle model**: Informally, the outputs of the hash function H look like the outputs of a truly random function (i.e., for every input x, $H(x)$ is randomly chosen in $\{0,1\}^k$).
- Applications of hash functions:
  1. **Integrity checks**: checking that a file stored in the cloud has not been tampered with; checking that a software update downloaded the right code binary; ...
     $\implies$ generally requires collision resistance/target collision resistance
  2. **Storing sensitive data**: password storage
     $\implies$ generally requires one-wayness
  3. **Hash-then-sign**: it is faster and cheaper to sign long messages if we hash them down first.
     $\implies$ generally requires collision resistance
  4. **Fiat-Shamir**: to eliminate interactions in public-coin interactive proof systems, we can replace the verifier's random messages by hashes of the protocol state.
     $\implies$ generally requires random oracle model
  5. **Commitments**: like a digital lock-box. Alice can "commit" to a value, without revealing what it is (if the commitment scheme is statistically/computationally hiding). Later, Alice can "open" this commitment to reveal her original value; if the commitment scheme is statistically/computationally binding, Alice can only open the commitment to her original value.
     $\implies$ generally requires collision resistance for computational binding and ROM for hiding.

2 Practice problem: Commitment schemes

2.1 Problem (from 2022 6.857 Pset 3)

Consider the following two commitment schemes:

1. Let $H : \{0,1\}^* \to \{0,1\}^n$ be a hash function, which we model as a random oracle, on security parameter n. We will use message space $\mathcal{M} = \{0,1\}^n$ and randomness space $\mathcal{R} = \{0,1\}^*$. Finally, we define:
$$\mathsf{Commit}(m, r) = H(m||r||0^n).$$

2. Let $H : \{0,1\}^{3n} \to \{0,1\}^{5n}$ be a hash function, which we model as a random oracle, on security parameter n. We will use message space $\mathcal{M} = \{0,1\}^n$ and randomness space $\mathcal{R} = \{0,1\}^n$. Finally, we define:
$$\mathsf{Commit}(m, r) = H(m||r||0^n).$$

(a) One of the schemes above is computationally hiding and statistically binding. Decide which one and justify your answer.

(b) One of the schemes above is statistically hiding and computationally binding. Decide which one and justify your answer.

(c) Argue that no commitment scheme can be both statistically binding and hiding.

(d) Does a statistically hiding commitment scheme need to be randomized?

(e) Can we implement a commitment scheme with encryption?

2.2 Solution

(a) The second commitment scheme is computationally hiding and statistically binding.

- It is statistically binding because the output space of H is much larger than the input space, so the probability that a collision occurs (i.e., that there exists some $m_1, m_2 \in \mathcal{M}$ and $r_1, r_2 \in \mathcal{R}$ such that $\mathsf{Commit}(m_1, r_1) = \mathsf{Commit}(m_2, r_2)$) is negligible in n. We prove this as follows:

  Let E_{m_1,m_2,r_1,r_2} be the event that $\mathsf{Commit}(m_1, r_1) = \mathsf{Commit}(m_2, r_2)$. Then, by a union bound:
$$\begin{aligned}\Pr\left[\bigcup_{m_1,m_2,r_1,r_2} E_{m_1,m_2,r_1,r_2}\right] &\leq \sum_{m_1,m_2,r_1,r_2} \Pr\left[E_{m_1,m_2,r_1,r_2}\right] \\ &\leq |\mathcal{M}|^2 \cdot |\mathcal{R}^2| \cdot \frac{1}{2^{5n}} \\ &\leq \frac{(2^n)^2 \cdot (2^n)^2}{2^{5n}} = \frac{2^{4n}}{2^{5n}} = \frac{1}{2^n}.\end{aligned}$$

  The second step here comes from the random oracle model: since H behaves like a truly random function, the probability that its output is the same on two given, distinct inputs is $1/2^{5n}$.

- It is computationally hiding because the hash function H, which we model as a random oracle, must be one-way. In particular, to recover m from just $c = \mathsf{Commit}(m, r)$, an adversary must brute-force search over all m and all r until it finds a pair (m, r) that produces c, since under the ROM all inputs are equally likely to map to c. The space of possible inputs grows exponentially with n (namely, it has size 2^{2n}), so any computationally-bounded adversary has only a negligible success probability.

(b) The first commitment scheme is statistically hiding and computationally binding.

- It is statistically hiding in the ROM because, for any two $m_1, m_2 \in \mathcal{M}$, the commitments to m_1 and m_2 will be uniformly random n-bit strings. With overwhelming probability, for any $c \in \{0,1\}^n$ and $m \in \{0,1\}^n$, there exists some $r \in \{0,1\}^*$ such that $\mathsf{Commit}(m, r) = c$.

  So, the distributions $\{\mathsf{Commit}(m_1, r) : r \leftarrow \mathcal{R}\}$ and $\{\mathsf{Commit}(m_2, r) : r \leftarrow \mathcal{R}\}$ are *statistically close*.

- It is computationally binding, because finding two messages $m_1, m_2 \in \mathcal{M}$ and randomness $r_1, r_2 \in \mathcal{R}$ such that $\mathsf{Commit}(m_1, r_1) = \mathsf{Commit}(m_2, r_2)$ is hard if H is collision-resistant. Specifically, if an adversary finds any such $m_1, m_2 \in \mathcal{M}$ and $r_1, r_2 \in \mathcal{R}$, then the adversary has found a collision on H – which we assume to be computationally hard. The best an adversary can do here is brute-force search over all m and all r until it finds a pair that produces c. The space of possible outputs is exponential in n (namely, 2^n), so any computationally bounded adversary has only negligible success probability.

(c) Assume that a commitment scheme is statistically binding. Then, by definition, an all-powerful adversary cannot find two distinct messages $m_1, m_2 \in \mathcal{M}$ that produce the same commitment, except with negligible probability. So, given any commitment c, the all-powerful adversary can iterate over all possible messages, and over all possible randomness, until it finds a message that produces the desired commitment c. Except with negligible probability, this will be the only message that produces this commitment c. So, the commitment scheme cannot be statistically hiding.

(d) Yes, a commitment scheme must be randomized – otherwise an all-powerful adversary could just iterate over every possible message and find the one committed to.

(e) No, because an encryption scheme might not be *binding*. For example, the one-time-pad is not binding.

3 Practice problem: Password storage

3.1 Problem (from 2022 exam)

To prevent adversaries from stealing lists of (username, password) pairs, servers generally store password lists in a form that is both *hashed* and *salted*. A *salt* is a random unique string that is concatenated with the password before hashing. So, what the server stores is not (*username*, *password*) but (*username*, *salt*, *H(password || salt)*), where H is a hash function such as SHA256.

(a) How would this scheme break if, instead of SHA256, the adversary used some hash function that was **not** one-way?

(b) The salt serves to increase the computational effort required for an adversary to recover passwords after a database breach. Explain why.

(c) In practice, when Alice logs in, she sends her password to the server (encrypted to prevent eavesdropping). Then, the server

  (a) looks up Alice's salt,

  (b) computes *credential = H(password || salt)*, and

  (c) verifies that the computed credential matches the one stored in the database.

  If they match, then Alice has successfully logged in.

  Why doesn't Alice just store the salt and compute the hash locally? Could the server then just check if the *credential* that she sends over matches the one it has stored, and never even see Alice's password?

3.2 Solution

(a) The adversary would be able to recover the password from *H(password || salt)* if H is not a one-way function.

(b) Without the salt, the adversary could recover a password by building a table of all (*username*, *H(password)*) pairs. With one such table (which would take an immense time to compute), the attacker could recover all passwords.

  With the salt, to mount such a precomputation attack, the adversary would need to build one huge table *for each salt*. So, each table only lets the attack recover a single password, which is much less efficient.

  In a similar vein, multiple users with the same password no longer have the same *H(password || salt)* value stored by the server.

(c) This would be equivalent to just using the credential as Alice's new password!

Sources. These recitation notes are based off of the following handouts:
`https://65610.csail.mit.edu/2023/lec/l11-hash.pdf`
`https://courses.csail.mit.edu/6.857/2022/files/H04-pset3.pdf`
`https://courses.csail.mit.edu/6.857/2022/files/R04-hash-functions.pdf`