HW 1 Answers – Question 2

CSCI-UA.0480-63

September 29, 2025

2. Hash functions and privacy

Prompt (abridged): BobCrypt uploads hashes of phone numbers to discover friends while claiming one-wayness preserves privacy.

(a) Why plain hashing phone numbers does not protect privacy

Phone numbers live in a tiny, structured space (e.g., 10–15 digits with known national formats). An adversary can simply enumerate all plausible phone numbers, hash each, and match the server's values. This defeats confidentiality despite hash one-wayness (one-wayness resists inverting a random preimage, not exhaustive enumeration of low-entropy domains). It also leaks users' entire address books to the server via set-membership queries.

Citations from Notes - Notes 2025.01.03: hash properties (collision, second-preimage), random-oracle intuition; one-wayness does not stop brute-force over small domains. - Notes 2025.02.02: hash usage in protocols and pitfalls when metadata or side channels leak information.

(b) Two mitigations and trade-offs

Mitigation 1: Per-user salting (H(phone — user_salt)) with keyed hashing for queries

- Server stores for each account a keyed digest, e.g., HMAC_k(phone) with server-held key k. Client fetches a short-lived token (MAC key or PRF token) and computes HMAC_k(address-book entries) locally to query. Without k, bulk offline rainbow/dictionary attacks on leaked hashes are infeasible; with k rotated, tokens limit abuse.
- *Trade-offs*: Server learns query results and could still enumerate if it misuses k. Requires online token issuance and rate-limiting.

Mitigation 2: Privacy-preserving contact discovery (k-anonymity bucketization or PSI)

- *K-anonymous buckets*: Client sends first r bits of H(phone) to fetch a bucket of size $\geq k$; performs local filter with full H(phone). Server sees only bucket queries, not exact hashes.
- *Private Set Intersection (PSI)*: Use a PSI protocol so client learns intersection of its address book with server's registered numbers, and server learns nothing about the rest. Efficient variants use OPRF/PRF-based PSI with rate limits.
- *Trade-offs*: K-anon leaks coarse prefix information and allows frequency attacks; PSI adds engineering and compute cost but offers strongest privacy.

(c) A better baseline design

Combine defense-in-depth:

- Use an OPRF/PSI-based contact discovery service as the default.
- If PSI unavailable, fall back to k-anonymous buckets with per-session OPRF tokens and strict rate limits.
- Never store raw hashes of phone numbers; store HMAC_k(phone) server-side with periodic key rotation and audit.

This aligns with the principle to avoid security by obscurity and rely on secret keys and provable primitives.

Selected excerpts (for traceability)

- Security Text 2025.01.03.txt, around line 1:
  Chapter 3: Authenticated storage, Hash functions and MACs Suggested reading: • Security Engineering Chapters 5.3.1, 5.6
- Security Text 2025.01.03.txt, around line 3:
  • Security Engineering Chapters 5.3.1, 5.6 Advanced reading: • GCAC 8 (Hash functions) Bonus reading: • The first SHA-1 collision (2017)
- Security Text 2025.01.03.txt, around line 5:
  • GCAC 8 (Hash functions) Bonus reading: • The first SHA-1 collision (2017) • More practical SHA-1 collisions (2019) • SHA-256 visualization
- Security Text 2025.01.03.txt, around line 6:
  Bonus reading: • The first SHA-1 collision (2017) • More practical SHA-1 collisions (2019) • SHA-256 visualization • SHA-3 visualization
- Security Text 2025.01.03.txt, around line 10:
  • SHA-3 visualization

  One more essential primitive to introduce is the cryptographic hash function. It's perhaps the most widely used primitive in all of cryptography. Hash functions are the workhorse underlying almost every protocol from encryption to cryptocurrency.

  We'll start with a simple application, a client uploading a large file to a cloud server:
- Security Text 2025.01.03.txt, around line 22:
  The client doesn't care about keeping the file secret, but wants to be sure that later when it downloads the file, it receives the same file f'=f. Obviously the client could just store the entire original file, but then why use a cloud server?

  Instead, we'll have the client store the hash of the file H(f), and then later check if H(f)=H(f'). Hash functions are similar to PRFs in that the output looks somewhat random. Hash functions only take one input, not two and that input can be infinitely long. That's right: hash functions are defined as a function H: $\{0,1\}^* \rightarrow \{0,1\}t$, they take any number of bits as input and then produce a t-bit output, sometimes called a digest.

  Our file-storage protocol will be secure as long as the server can't find any other file with the same hash. This property is called second-preimage-resistance and is defined with the following game:
- Security Text 2025.01.03.txt, around line 24:
  Instead, we'll have the client store the hash of the file H(f), and then later check if H(f)=H(f'). Hash functions are similar to PRFs in that the output looks somewhat random. Hash functions only take one input, not two and that input can be infinitely long. That's right: hash functions are defined as a function

H: {0,1}* → {0,1}t, they take any number of bits as input and then produce a t-bit output, sometimes called a digest.

Our file-storage protocol will be secure as long as the server can't find any other file with the same hash. This property is called second-preimage-resistance and is defined with the following game:

• The challenger randomly chooses x and sends it to the adversary.

- Security Text 2025.01.03.txt, around line 32:
  You should be able to convince yourself that if this property is true, the above file storage protocol is secure. This approach is widely used in practice. For example, official releases of software packages like Ubuntu Linux often publish the hash of the authentic version of the software. If you obtain the hash of the desired file securely, you can then download the (often quite large) file from any untrusted service like BitTorrent and be sure you've obtained the correct version by checking its hash.

  Cryptographic hash functions are also designed to be one-way, but the definition is quite similar to that for one-way functions as above so we'll skip it here. Hash functions and collision-resistance