Hash Functions

Security notions, HMAC, birthday attacks, RO model/controversies, hash trees, etc.

Hash Function

- Provides way to map long, variable-length input string to short, fixed-length output string

- Input: pre-image
- Output: digest

- Similar to hash tables in data structures:
  - O(1) look-up time
  - Key-value pairs: H(x): key ↔ x: value

Hash Function

- Collision: $H(x) = H(x'); x \neq x'$

- Fundamental differences:
    - In data structures, collisions are undesirable (but tolerable)
    - In crypto, collision-resistance is *requirement*

    - Data structure elements likely chosen independently (of H)
    - Here, A will choose elements to deliberately cause collisions in H

- Data structures collisions: use linear/quadratic probing, use chained linked lists, double hashing, etc.

Keyed Hash Function

- $H^s(x) = H(s,x)$
- s is a "key", x value, s public

- Collision Game $Coll_{A,H}(n)$:
  - Challenger runs $s \leftarrow Gen(1^n)$
  - A given s, H outputs x, x′
  - A wins ("finds a collision") if $x \neq x'$, but $H^s(x) = H^s(x')$, set output = 1

- H is collision-resistant, if for all PPT adversaries A, there is a negl. function, s.t.:

$$\Pr[Coll_{A,H}(n) = 1] \leq negl(n)$$

Collisions

- Collision game defined for *strong* collision resistance, or just “collision resistance”
- Weak-collision resistance
  - Given s, x, H, A can’t find an x′, s.t., $x \neq x'$, but $H^s(x) = H^s(x')$
  - “Second-preimage resistance”
- One-way property
  - Given s, y, H, A can’t find an x, s.t., $H^s(x) = y$
  - “Pre-image resistance”

Collisions, etc.

- Easy to see strong collision resistance implies other two properties (but not other way!)

- Un-keyed H
  - Real-world crypto hash functions usually "un-keyed" — no s
  - $H:\{0,1\}^* \longrightarrow \{0,1\}^n$
  - But still satisfy collision-resistance

Domain Extension

- We require H to be able to process arbitrary-length inputs
- How to construct H?

- Common approach:
  - Construct a collision-resistant compression function, H
  - Use domain extension on H to handle arbitrary-length inputs

Merkle-Damgård Transform

- Domain extension function:
  - Merkle-Damgård transform[1]
  - Used for SHA-family (SHA-1,SHA-2), MD5 too
  - During design, helps restrict attention to fixed-length case

1: Not to be confused with Merkle hash tree!

Merkle-Damgård Transform

- Let h: $\{0,1\}^{2n} \rightarrow \{0,1\}^n$ be a compression function, construct H: $\{0,1\}^* \rightarrow \{0,1\}^n$

- D (digest) <— H(s, $x \in \{0,1\}^*$) /* $|D|=n, |x|<2^n$ */
  - Set B = $\lceil |x|/n \rceil$ /*# of blocks in x*/
  - Parse blocks $x_1,\ldots x_B$, add padding if necessary
  - Set $x_{B+1} = |x|$ /* Needed for knowing size of x*/
  - Set $z_0 = 0^n$ /* This is IV */
  - for i = 1 to (B+1) do
    - Compute $z_i = h^s(z_{i-1}||x_i)$
  - Return D = z_{B+1}

Hash-and-MAC

- Informally: $y \leftarrow H^s(m)$, then tag $\leftarrow$ MAC(y)

- Hash-and-Mac algorithms:
- $(k,s) \leftarrow Gen(1^n)$
  - Randomized
  - Choose $k \in \{0,1\}^n$, generate s, return (k,s)
- $t \leftarrow MAC(k, s, m \in \{0,1\}^*)$
  - Randomized
  - Output $t \leftarrow MAC_k(H^s(m))$
- {"accept","reject"} $\leftarrow$ Verify(k, s, m, t)
  - Deterministic
  - Return accept iff $Verify(k,H^s(m),t) \stackrel{?}{=} 1$

Just like signcryption, hash-and-MAC inherits properties of its parents (MAC-unforgeability and H's collision-resistance)

HMAC

- Hashed Message Authentication Code

- HMAC algorithms
- $(k,s) \leftarrow \mathrm{Gen}(1^{n'})$
  - Randomized
  - Choose $k \in \{0,1\}^{n'}$, generate s, return (k,s)
- $t \leftarrow \mathrm{MAC}(k, s, m \in \{0,1\}^*)$
  - Randomized. $|ipad| = |opad| = n'$
  - Output $t \leftarrow H^s((k \oplus opad) \| H^s((k \oplus ipad) \| m))$
- {"accept","reject"} $\leftarrow$ Verify(k, s, m, t)
  - Deterministic
  - Return accept iff $H^s((k \oplus opad) \| H^s((k \oplus ipad) \| m)) \stackrel{?}{=} t$

HMAC

- HMAC an instantiation of Hash-and-MAC

- Regular Hash-and-MAC just does t<— $MAC_k(H^s(m))$
- HMAC does t<— $H^s((k \oplus opad) || H^s((k \oplus ipad) || m))$

- Why the extra parameters?

HMAC

- First, why does k go "inside" twice, e.g., $H^s((k \oplus opad) || H^s((k \oplus ipad) || m))$?
- This guarantees HMAC'll be secure, even if H is only *weakly-collision resistant*
- MD5 was discovered to not be (fully) collision-resistant, HMAC-MD5 was still secure[1]

1: Helped buy time to replace MD5 in HMAC implementations. HMAC-MD5 shouldn't be used now, regardless

HMAC

- Role of ipad and opad:
  - Ensure independent keys in inner/outer computation
  - k used with ipad, opad to derive 2 keys

- $k_{out} = H^s(IV \mid\mid (k \oplus opad))$
- $k_{in} = H^s(IV \mid\mid (k \oplus ipad))$

- $MAC_{kin,kout}(m) = H^s(k_{out} \mid\mid H^s_{kin}(m))$

HMAC

- Industry standard, widely used in practice
- Supported by proof based on standard assumptions on hash functions (one-wayness, weak-collision-resistance)
- Proof in standard model (not random oracle model)
  - Reduction-based – works in usual way

Birthday Attack

- Trivial birthday attack (from pigeon-hole principle):
  - Let H: $\{0,1\}^* \rightarrow \{0,1\}^l$
  - Simply compute $H(1),\ldots,H(2^l+1)$
  - At least 2 outputs *will* collide

- Birthday paradox
  - Min. no. of people in room with > 50% chance of colliding birthdays
  - 23
  - Roughly: collisions after $2^{l/2}$ digests for H

Naïve Birthday Attack

- Alice has 2 messages
  - M = "I'll loan Alice $100"
  - M′ = "I agree to pay Alice $1,000,000"

- Plans to ask Bob to sign H(M), and attach Bob's sig. to H(M′)

- Prepares:
  - $(M_1,\ldots,M_{2^{l/2}})$, $(M'_1,\ldots,M'_{2^{l/2}})$,
  - $(H(M_1),\ldots,H(M_{2^{l/2}}))$, $(H(M'_1),\ldots,H(M'_{2^{l/2}}))$

Naïve Birthday Attack

- Probability of collision between $H(M_i)$, $H(M'_j)$ > 50%

- Some points:
  - All M_i,M'_i must make sense (else Bob won't sign legit. M)
  - Simply write same sentence in $2^{l/2}$ different ways
  - Change syntax, not meaning

- Significant memory overhead – Alice needs to store 2 lists of $2^{l/2}$ values each

Small-space Birthday Attack

- Birthday attack = $O(2^{l/2})$ space, $O(2^{l/2})$ time
  - Since attacker doesn't know which pair of values will yield a collision
- Better method: space-efficient, $O(2^{l/2})$ time, $O(1)$ space
- Idea: pick a random value, repeatedly hash it until collision found

Small-space Birthday Attack

- Goal: given $H:\{0,1\}^* \rightarrow \{0,1\}^l$, find distinct x, x' with $H(x) = H(x')$

- Pick an $x_0 \in \{0,1\}^{l+1}$

- Compute $x_i = H(x_{i-1})$, $x_{2i} = H(H(x_{2(i-1)}))$; $\forall\ i \in [1..2^{l/2}]$
  - $i=1$: $x_1 = H(x_0)$, $x_2 = H^2(x_0)$ (2-fold H)
  - $i=2$: $x_2 = H(x_1)$, $x_4 = H^2(x_2)$
  - $i=3$: $x_3 = H(x_2)$, $x_6 = H^2(x_4)$, …
  - $x_i = H^i(x_0)$ (i-fold H)

Small-space Birthday Attack

- Known result: If $x_1,\ldots,x_q$ is a sequence of values with $x_m = H(x_{m-1})$, and if $x_I=x_J$, $1\leq I<J\leq q$, there exists an $i < J$, s.t., $x_i = x_{2i}$

- Applying result:
  - In each step, compare (x_i, x_{2i}), if $x_i = x_{2i}$ there has to be a collision in $x_0,x_1,\ldots,x_{2i-1}$
  - Find an $x_j = x_{i+j}$
  - Output (x_{j-1}, x_{j+i-1}) as (values that casued) the collision

- Example: If $x_3 = x_6$, has to be a collision in $x_0,x_1,\ldots,x_5$
- So, j=3 (since $x_j = x_{i+j}$), output (x_2, x_5) as the collision

Inverting Hash Functions

- Consider $H:\{0,1\}^l \rightarrow \{0,1\}^l$. Given y = H(x), find x′, s.t., y = H(x′)

- Naïve way: Compute all 2^l digest values of H. Time $O(2^l)$, Space (memory) is O(1)[1]

- Smart way: Incur more space cost upfront, amortize cost over efficient H inversions
  - Spend big on pre-processing time/space, "pay off debt" over time
  - Comes from amortized analysis in algorithms
  - Used for on-line algorithms

1: Test each H(x′) sequentially, if H(x′)≠y, delete H(x′), move onto next

Smart Way 1

- A priori, evaluate and store all $(x, H(x)_1), \ldots, (x, H(x)_{2^l})$ for some x,
- Store in, say, Hash table – look-up time O(1)
- When a "y" comes in, just do a look-up

- Time: O(1), Space: $O(2^l)$

- Naïve, smart way 1: two extremes
  - Could we think of a middle ground? Space-time tradeoff?

Smart Way 2

- Flavor of BSGS

- Assume $H:\{0,1\}^l \longrightarrow \{0,1\}^l$ defines a circle, i.e., x, $H(x)$, $H^2(x)$, $H^3(x)$,..., $H^{2^l}(x)$ covers all of $\{0,1\}^{2^l}$
- Let $N = 2^l$

- Pre-processing (off-line phase):
  - Partition cycle into $\sqrt{N}$ segments (à la giant steps)
  - Store (beginning, end) pairs: $(x_{i\cdot\sqrt{N}}, x_{(i+1)\cdot\sqrt{N}})$; $\forall\, i \in \{0,1,\ldots,\sqrt{N}-1\}$ segments
  - Store $\sqrt{N}$ pairs in table – space $O(\sqrt{N})$

Smart Way 2

- Is the segmentation $(x_{i\bullet\sqrt{N}}, x_{(i+1)\bullet\sqrt{N}})$; $\forall$ i $\in$ {0,1, …,$\sqrt{N}$-1} correct? Whole circle covered?
- Yes!

- $i = 0$: $(x_{0\bullet\sqrt{N}}, x_{1\bullet\sqrt{N}}) = (x_0, x_{\sqrt{N}})$
- $i = 1$: $(x_{\sqrt{N}}, x_{2\bullet\sqrt{N}})$
- $i = 2$: $(x_{2\sqrt{N}}, x_{3\bullet\sqrt{N}})$
- $i = 3$: $(x_{3\sqrt{N}}, x_{4\bullet\sqrt{N}})$
- …
- $i = \sqrt{N}-1$: $(x_{(\sqrt{N}-1)\bullet\sqrt{N}}, x_{(\sqrt{N}-1+1)\bullet\sqrt{N}}) = (x_{N-\sqrt{N}}, x_N)$

Smart Way 2

- On-line phase: a “y” comes in, need to find $H^{-1}(y)$
- Check if y, $H(y)$, $H^2(y)$, $H^3(y)$,…, $H^{\sqrt{N}}(y)$ correspond to an endpoint
  - Basically check entire circumference

- Guaranteed to hit an endpoint in time $O(\sqrt{N})$ (since y lies in the circle)

Smart Way 2

- Once endpoint $x = x_{(i+1)\bullet\sqrt{N}}$ found, take starting point, $x' = x_{i\bullet\sqrt{N}}$

- Baby steps:
  - Compute $H(x'), H^2(x'), H^3(x'),\ldots,H^{N-\sqrt{N}}(x')$, until y is found
  - Pr[finding y] = 1 — since y lies in the circle

- Entire time: $O(\sqrt{N})$, entire space $O(\sqrt{N})$
- I.e., $O(2^{l/2})$, $O(2^{l/2})$ resp.

Smart Way 2

- Problem: assumes H's range forms a circle

- Hellman's optimization:
  - Generalized Smart Way 2, with H's range not a circle
  - Also, $H:\{0,1\}^* \rightarrow \{0,1\}^l$ – domain, range different
  - Time: $O(2^{2l/3})$, space: $O(2^{2l/3})$

- Rainbow table – modification of Hellman algorithm

Random Oracle (RO) Model

- Rigorous, formal proofs backbone of modern crypto
- Hash functions used in many crypto protocols
- What if the proof of security requires more than just collision-resistance of H, to go through…?
- Choice: no proof, or use unproven protocols
  - …which have no security justification

RO Model

- Hugely popular approach: Prove protocols secure in an *idealized* model

- What does that entail?
  - Assume a magic hash “oracle”, O exists
  - Adversary, honest parties can query O, e.g., send x
  - O replies with H(x)
  - H(x) completely random
  - O inscrutable[1] — no-one knows internal workings of O, H

1: Also called as “black-box”

RO Model

- Approach:
  - First, prove protocols secure in RO model
  - Then, replace H with real-world hash function H′

- Hope transition is seamless, security preserved

- Core problem: No justification for this hope!
  - Many proofs in RO model *break down* when H replaced by H′
  - Famous example: F-S paradigm

RO Model

- Proofs “break down”…? Huh?

- H has *infinite* range; real-world H′: $\{0,1\}^*$ —> $\{0,1\}^l$ – always finite range

- Proofs by reduction (sometimes) require *programmability*
    - O *chooses* value of H(x) returned
    - H′ output *cannot* be chosen arbitrarily

- Proofs by reduction (sometimes) require *extractability*
    - O sees all queries, x – A’s x queries are known
    - A’s H′ queries oracle-independent

RO Model

- Source of controversy and heartburn for 10+ years
- Around 2006-2007 heated arguments break out
- Supporters: RO model flawed, but better than nothing – most of theoretical crypto community
- Naysayers: RO model worthless, proofs in RO useless – a *few*

Cons of RO Model

- Biggest objection

No justification at all to believe proofs with O translate into proofs with real-world H′!

- No real-world H′ can emulate O's H
- Are such RO "proofs" even useful? Waste-of-time?
- See Lindell—Menzes argument, Damgård's response, etc.

Support for RO Model

- Biggest support

Some proof better than no proof at all

- Gives confidence to schemes being considered for standardization (NIST, ENISA, etc. won't touch candidates otherwise)

- Deployed schemes with proofs in RO model:
  - RSA, RSA-PKCS, ElGamal, RSA-FDH, Schnorr schemes, DSA-ECDSA
  - Note: DSA, ECDSA proven secure if H,F can be modeled as RO

- So far, no real-world attacks on schemes with proofs in RO model

Closing RO Remarks

- Standard model — gold standard
  - I.e., assume only real-world hash functions H
- … but if no proof exists in standard model, RO acceptable alternative
- No self-respecting standards org. will accept a "proof-less" candidate (nor will anyone in the crypto community take seriously)
- Think about Damgård's essay

Hash Function Applications

- H(x) serves as unique id for any x

- Applications:
- Fingerprinting (virus)
  - Store hashes of known viruses (and mutations)
  - Compare with hash of apps/downloads

- Data de-duplication (remove redundancies)
  - Many users store H(F) in cloud, F=popular movie
  - New upload: add pointer/symbolic link to H(F)

File-checking

- Hash functions useful in file-integrity, file-freshness checking
- E.g., files stored on untrusted server, user periodically does read/update

- Threat Model
  - Untrusted server
  - May modify files
  - Or return valid, but old copies

File-checking

- Naïve solution 1:
  - Server stores $x_1,\ldots,x_n$, user stores $y_1 \leftarrow H(x_1), \ldots, y_n \leftarrow H(x_n)$
  - User check $H(x_i) \stackrel{?}{=} y_i$ for downloaded x_i
  - Space-cost at user: $O(n)$, time = $O(1)$

- Naïve solution 2:
  - User stores $y \leftarrow H(x_1,\ldots,x_n)$, server stores $x_1,\ldots,x_n$
  - User wants to check x_i, downloads all $x_1,\ldots,x_n$
  - User time = space = $O(1)$, communication cost = $O(n)$

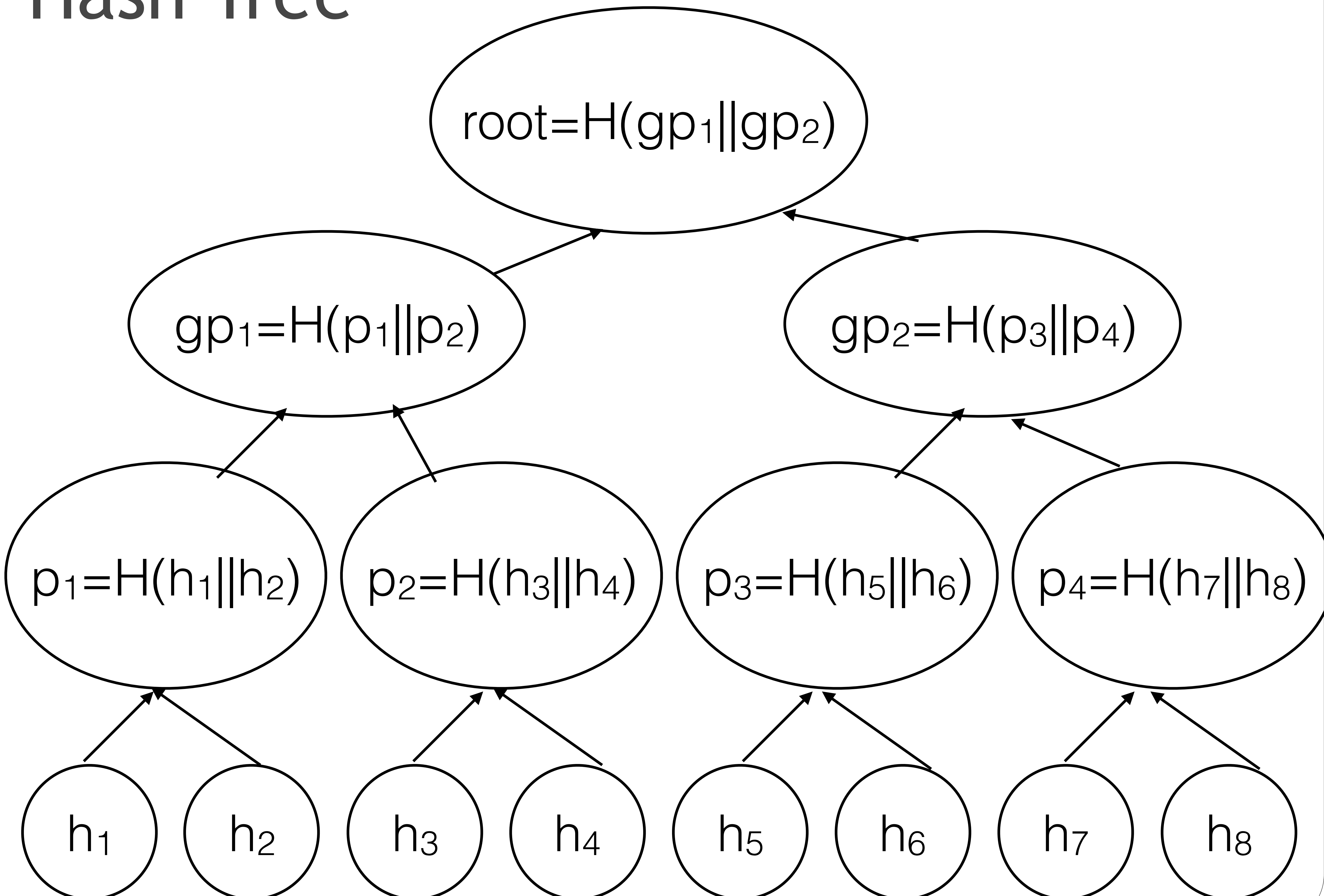
Merkle Hash Tree

- Ralph Merkle, 1979

- Balanced binary tree, with special property
- Insert, update, delete: O(log n), for tree with n leaves
- Hashes of files stored at leaves

- Have files: $f_1, f_2, f_3, f_4, f_5, f_6, f_7, f_8$
- Compute $h_1 = H(f_1)$, $h_2 = H(f_2)$, $h_3 = H(f_3)$, $h_4 = H(f_4)$, $h_5 = H(f_5)$, $h_6 = H(f_6)$, $h_7 = H(f_7)$, $h_8 = H(f_8)$

Hash Tree



root=H(gp1||gp2)
gp1=H(p1||p2)
gp2=H(p3||p4)
p1=H(h1||h2)
p2=H(h3||h4)
p3=H(h5||h6)
p4=H(h7||h8)
h1
h2
h3
h4
h5
h6
h7
h8

Hash Tree - Read

User

Server

Has files: $f_1,f_2,f_3,f_4,f_5,f_6,f_7,f_8$
Computes and stores rootHash, deletes files

- Computes root over $f_1,f_2,f_3,f_4,f_5,f_6,f_7,f_8$
- Stores entire tree

Sends $f_1,f_2,f_3,f_4,f_5,f_6,f_7,f_8$ →

Sends read request for f_4 →

← Sends f_4, h_4, sibling-hashes:(h_3, p_1, gp_2)

Verifies:

- h_4
- $p_2=H(h_3||h_4)$
- $gp_1=H(p_1||p_2)$
- $root=H(gp_1||gp_2)$
- Accepts if root = rootHash

Hash Tree - Update

Entire tree stored at Bob's side

User

Server

Needs to update f_6

Sends read request for f_6 →

Sends f_6, h_6, sibling-hashes:(h_5, p_4, gp_1) ←

- First verifies current f_6 integrity. If ok:
- Computes h'_6, sibling-hashes:(p'_3, gp'_2), root′
- Stores root′

Sends f'_6, h'_6, sibling-hashes:(p'_3, gp'_2), root′ →

- Updates tree with recd. values
- Stores root′

Hash Tree

- Some points:
  - Minimize user-server comm: send only the bare minimum (for verification)
  - If (log n) not power of 2, add dummy (null) leaves
    - Unfortunate case: no. of files = 2^h+1, h = height of tree,
    - E.g., 17 files, need to add 15 dummy-leaves for $2^5=32$
  - Very efficient: O(log n) all ops.
    - E.g., 4096 files, user-server comp./comm. ≈16 hashes
  - User storage-cost O(1) – only rootHash stored