Lecture #10: Cryptographic Hash Functions

COMS 4995-001:
The Science of Blockchains
URL: https://timroughgarden.org/s25/

Tim Roughgarden

Goals for Lecture #10

1. Short, unique names that require no coordination.
   – e.g., for transactions or blocks

2. Cryptographic hash functions as “random oracles.”
   – ideal hash function = random function, though still has collisions

3. What do cryptographic hash functions actually look like?
   – case study: SHA-256 and length-extension attacks

4. Cryptographic commitments.
   – reconstructing blocks from hashes; binding and hiding

Short, Unique Names

Cryptography in blockchain protocols: two unavoidable primitives: digital signatures (lec #5) and cryptographic hash functions (now).

Short, Unique Names

Cryptography in blockchain protocols: two unavoidable primitives: digital signatures (lec #5) and cryptographic hash functions (now).

Recall: in a blockchain, each block (except 1^{st}) has a *predecessor*.

Question: how should a block specify its predecessor?

Short, Unique Names

Cryptography in blockchain protocols: two unavoidable primitives: digital signatures (lec #5) and cryptographic hash functions (now).

Recall: in a blockchain, each block (except 1st) has a *predecessor*.

Question: how should a block specify its predecessor?

Ideally: use some "naming function" h(.) such that:

- h is easy to evaluate
- the output of h is short
- never have ambiguous/non-unique names: $x \neq x' \Rightarrow f(x) \neq f(x')$

Hash Functions and Collisions

Definition: a hash function h maps each finite-length string x to an element h(x) of some range Y. [canonical example: $Y = \{0,1\}^{256}$]

– length of x can be as long as you want (e.g., text of *War and Peace*)

Hash Functions and Collisions

Definition: a hash function h maps each finite-length string x to an element h(x) of some range Y. [canonical example: $Y = \{0,1\}^{256}$]

– length of x can be as long as you want (e.g., text of *War and Peace*)

By Pigeonhole Principle: no matter what h is, will have *collisions*.

- collision: pair $x \neq x'$ for which $f(x)=f(x')$

Hash Functions and Collisions

Definition: a hash function h maps each finite-length string x to an element h(x) of some range Y. [canonical example: $Y = \{0,1\}^{256}$]

– length of x can be as long as you want (e.g., text of *War and Peace*)

By Pigeonhole Principle: no matter what h is, will have *collisions*.

- collision: pair $x \neq x'$ for which $f(x)=f(x')$
  – Pigeonhole: put n+1 pigeons in n holes, some hole has ≥ 2 pigeons

Hash Functions and Collisions

Definition: a hash function h maps each finite-length string x to an element h(x) of some range Y. [canonical example: $Y = \{0,1\}^{256}$]

- length of x can be as long as you want (e.g., text of *War and Peace*)

By Pigeonhole Principle: no matter what h is, will have *collisions*.

- collision: pair $x \neq x'$ for which $f(x)=f(x')$
  - Pigeonhole: put n+1 pigeons in n holes, some hole has ≥ 2 pigeons

Best-case scenario: a function h for which we'll never encounter a collision in practice (no matter how hard an adversary might try).

Ideal Hash Function: A Random Function

Ideal cryptographic hash function: a uniformly random function h:

Ideal Hash Function: A Random Function

Ideal cryptographic hash function: a uniformly random function h:

- [a gnome in a box, with a big book and some dice]

- on input x:

Ideal Hash Function: A Random Function

Ideal cryptographic hash function: a uniformly random function h:

- [a gnome in a box, with a big book and some dice]

- on input x:
  - if h(x) has never been evaluated before:
    - flip 256 new random coins and return the result

Ideal Hash Function: A Random Function

Ideal cryptographic hash function: a uniformly random function h:

– [a gnome in a box, with a big book and some dice]

- on input x:
  - if h(x) has never been evaluated before:
    - flip 256 new random coins and return the result
  - else
    - return the same output as previous evaluations of h at x

Ideal Hash Function: A Random Function

Ideal cryptographic hash function: a uniformly random function h:

- on input x:
  - if h(x) has never been evaluated before:
    - flip 256 new random coins and return the result
  - else
    - return the same output as previous evaluations of h at x

Ideal Hash Function: A Random Function

Ideal cryptographic hash function: a uniformly random function h:

- on input x:
  - if h(x) has never been evaluated before:
    - flip 256 new random coins and return the result
  - else
    - return the same output as previous evaluations of h at x

Fact: for such a function, don't expect to see any collisions until it's been evaluated $\approx 2^{128}$ times.

– i.e., for all practical purposes, never!

Ideal Hash Function: A Random Function

Ideal cryptographic hash function: a uniformly random function h:

- on input x:
  - if h(x) has never been evaluated before:
    - flip 256 new random coins and return the result
  - else
    - return the same output as previous evaluations of h at x

Fact: for such a function, don't expect to see any collisions until it's been evaluated $\approx 2^{128}$ times.

– i.e., for all practical purposes, never!

Reason: the "birthday paradox."

The Birthday Paradox

- suppose h a uniformly random function with range Y
- suppose h evaluated at N different points

The Birthday Paradox

- suppose h a uniformly random function with range Y
- suppose h evaluated at N different points
  - N(N-1)/2 opportunities for a collision (one per pair of points)

The Birthday Paradox

- suppose h a uniformly random function with range Y
- suppose h evaluated at N different points
  - N(N-1)/2 opportunities for a collision (one per pair of points)
  - probability that a given pair of points collide = 1/|Y|

The Birthday Paradox

- suppose h a uniformly random function with range Y
- suppose h evaluated at N different points
  - N(N-1)/2 opportunities for a collision (one per pair of points)
  - probability that a given pair of points collide = 1/|Y|
  - expect one collision when |Y| ≈ N(N-1)/2, or N ≈ $\sqrt{2|Y|}$

The Birthday Paradox

- suppose h a uniformly random function with range Y
- suppose h evaluated at N different points
  - N(N-1)/2 opportunities for a collision (one per pair of points)
  - probability that a given pair of points collide = 1/|Y|
  - expect one collision when |Y| ≈ N(N-1)/2, or N ≈ $\sqrt{2|Y|}$
  - if Y = 256-bit strings, need N ≈ $2^{256/2} = 2^{128}$

The Birthday Paradox

- suppose h a uniformly random function with range Y
- suppose h evaluated at N different points
  - N(N-1)/2 opportunities for a collision (one per pair of points)
  - probability that a given pair of points collide = 1/|Y|
  - expect one collision when |Y| ≈ N(N-1)/2, or N ≈ $\sqrt{2|Y|}$
  - if Y = 256-bit strings, need N ≈ $2^{256/2} = 2^{128}$

Also: (more detailed but elementary probability calculations)

- # of evaluations << 2^{128} ➔ almost no chance of a collision
- # of evaluations >> 2^{128} ➔ almost no chance of no collisions

Cryptographic Hash Functions

Informal definition: a function is *collision-resistant* if it's computationally infeasible to find collisions.

Cryptographic Hash Functions

Informal definition: a function is *collision-resistant* if it's computationally infeasible to find collisions.

- intuitively, as hard as for a random function with the same range
- like signatures, precise security statement involve many nuances
- more common property desired of "cryptographic" hash functions

Cryptographic Hash Functions

Informal definition: a function is *collision-resistant* if it's computationally infeasible to find collisions.

- intuitively, as hard as for a random function with the same range
- like signatures, precise security statement involve many nuances
- more common property desired of "cryptographic" hash functions

Birthday paradox: size of range = twice the desired bits of security.

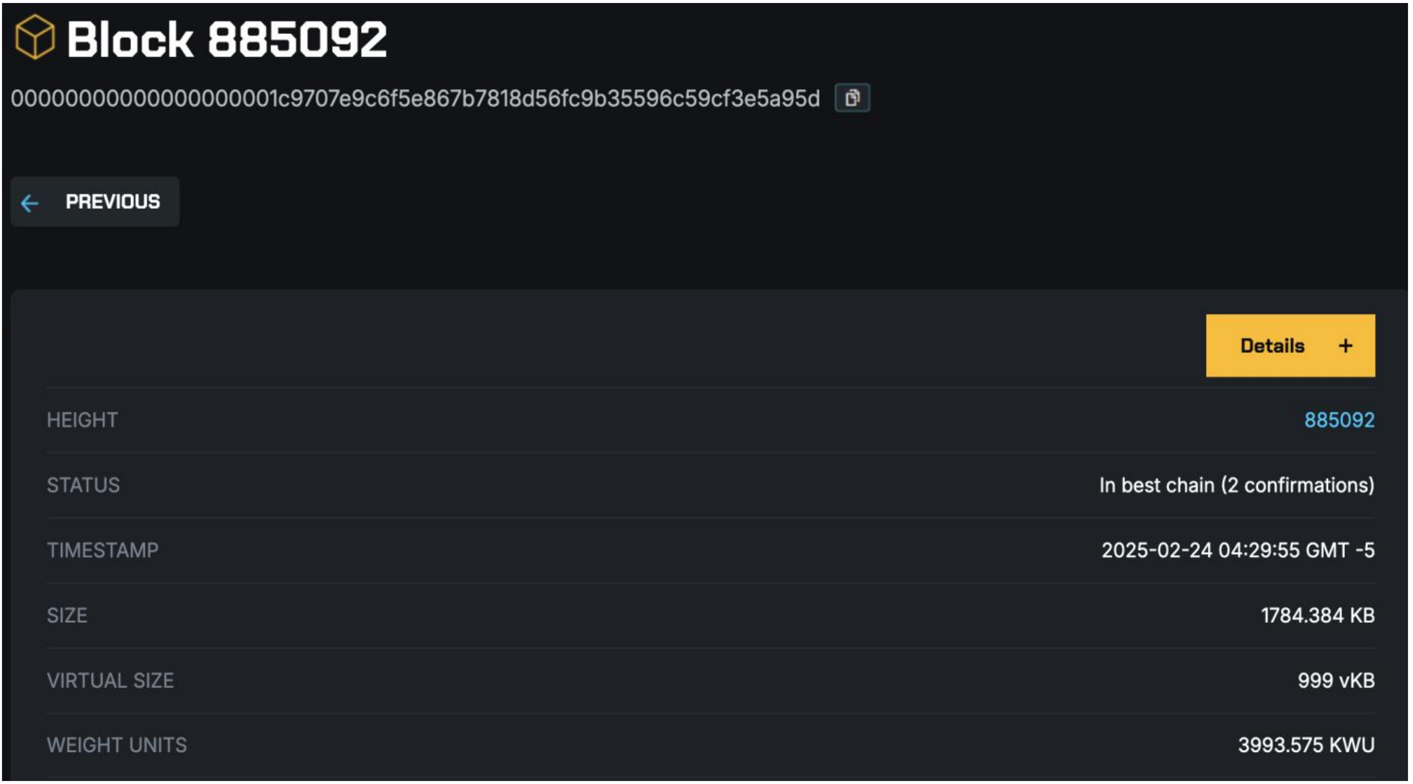
Cryptographic Hash Functions

Informal definition: a function is *collision-resistant* if it's computationally infeasible to find collisions.

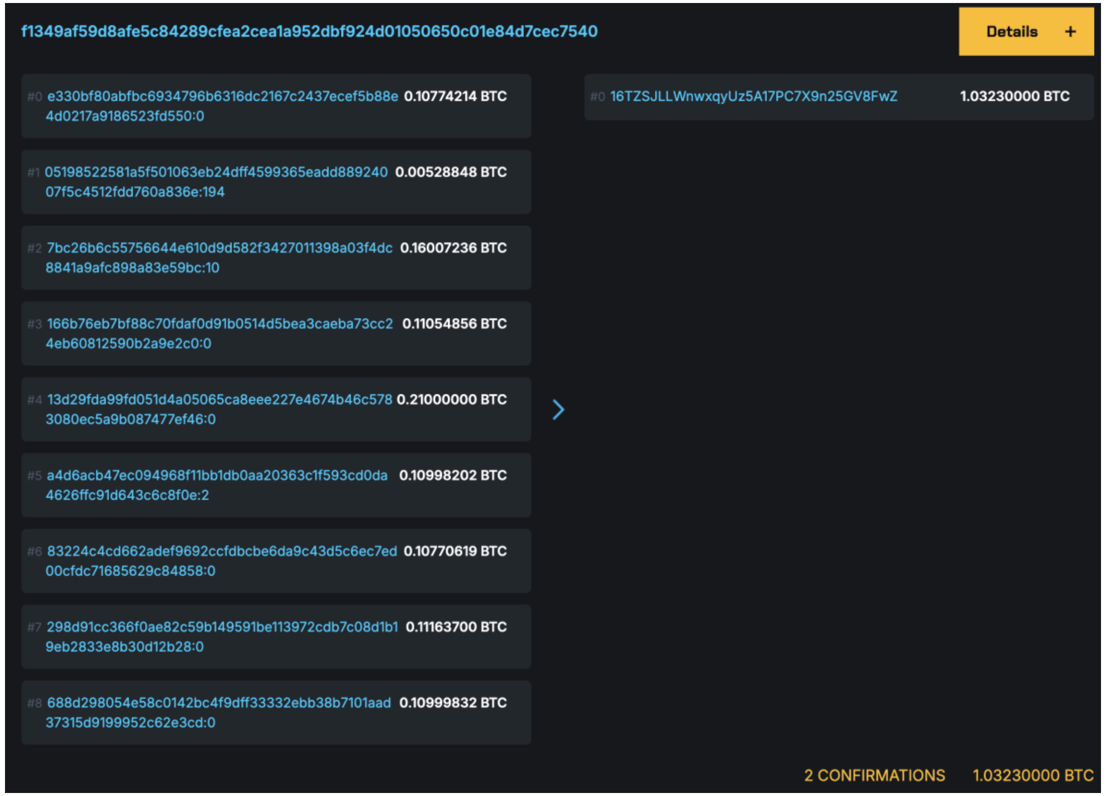
- intuitively, as hard as for a random function with the same range
- like signatures, precise security statement involve many nuances
- more common property desired of "cryptographic" hash functions

Birthday paradox: size of range = twice the desired bits of security.

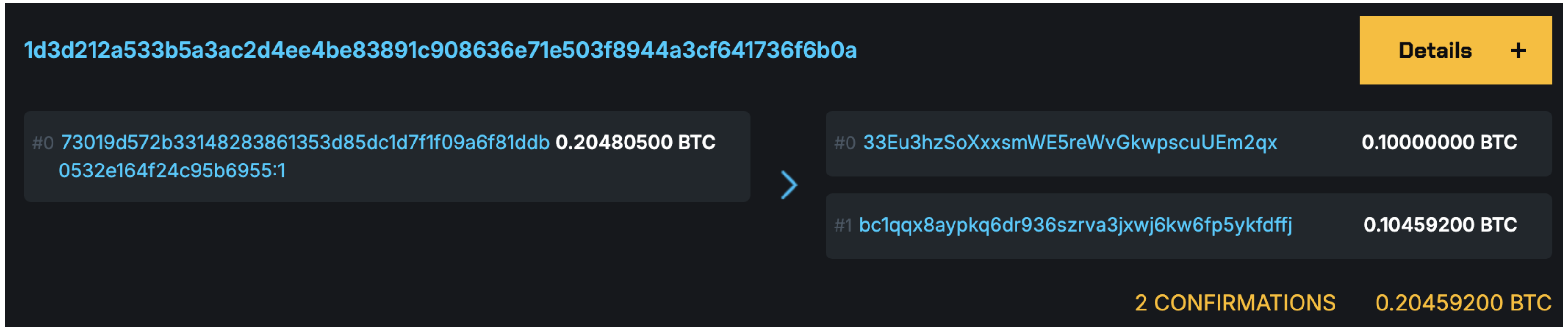
Upshot: can use a collision-resistant hash function (with e.g. 256-bit outputs) to provide objects with names that are short and (for all practical purposes) unique. [no coordination necessary!]



Block 885092
000000000000000000001c9707e9c6f5e867b7818d56fc9b35596c59cf3e5a95d
PREVIOUS
Details +
HEIGHT
885092
STATUS
In best chain (2 confirmations)
TIMESTAMP
2025-02-24 04:29:55 GMT -5
SIZE
1784.384 KB
VIRTUAL SIZE
999 vKB
WEIGHT UNITS
3993.575 KWU



f1349af59d8afe5c84289cfea2cea1a952dbf924d01050650c01e84d7cec7540
Details +
#0 e330bf80abfbc6934796b6316dc2167c2437ecef5b88e4d0217a9186523fd550:0 0.10774214 BTC
#1 05198522581a5f501063eb24dff4599365eadd88924007f5c4512fdd760a836e:194 0.00528848 BTC
#2 7bc26b6c55756644e610d9d582f3427011398a03f4dc8841a9afc898a83e59bc:10 0.16007236 BTC
#3 166b76eb7bf88c70fdaf0d91b0514d5bea3caeba73cc24eb60812590b2a9e2c0:0 0.11054856 BTC
#4 13d29fda99fd051d4a05065ca8eee227e4674b46c5783080ec5a9b087477ef46:0 0.21000000 BTC
#5 a4d6acb47ec094968f11bb1db0aa20363c1f593cd0da4626ffc91d643c6c8f0e:2 0.10998202 BTC
#6 83224c4cd662adef9692ccfdbcbe6da9c43d5c6ec7ed00cfdc71685629c84858:0 0.10770619 BTC
#7 298d91cc366f0ae82c59b149591be113972cdb7c08d1b19eb2833e8b30d12b28:0 0.11163700 BTC
#8 688d298054e58c0142bc4f9dff33332ebb38b7101aad37315d9199952c62e3cd:0 0.10999832 BTC
#0 16TZSJLLWnwxqyUz5A17PC7X9n25GV8FwZ 1.03230000 BTC
2 CONFIRMATIONS
1.03230000 BTC



1d3d212a533b5a3ac2d4ee4be83891c908636e71e503f8944a3cf641736f6b0a
Details +
#0 73019d572b33148283861353d85dc1d7f1f09a6f81ddb0532e164f24c95b6955:1 0.20480500 BTC
#0 33Eu3hzSoXxxsmWE5reWvGkwpscuUEm2qx 0.10000000 BTC
#1 bc1qqx8aypkq6dr936szrva3jxwj6kw6fp5ykfdffj 0.10459200 BTC
2 CONFIRMATIONS
0.20459200 BTC

Case Study: SHA-256

Note: infeasible to use a random function in practice.

- describing such a function requires an infinite number of bits

Case Study: SHA-256

Note: infeasible to use a random function in practice.

- describing such a function requires an infinite number of bits

In practice: use function that is easy to describe + evaluate, but as unpredictable (for practical purposes) as a random function.

Code for SHA-256

```
Pre-processing (Padding):
begin with the original message of length L bits
append a single '1' bit
append K '0' bits, where K is the minimum number >= 0 such that L + 1 + K + 64 is a
multiple of 512
append L as a 64-bit big-endian integer, making the total post-processed length a multip
of 512 bits

Process the message in successive 512-bit chunks:
break message into 512-bit chunks
for each chunk
    create a 64-entry message schedule array w[0..63] of 32-bit words
    (The initial values in w[0..63] don't matter, so many implementations zero them here
    copy chunk into first 16 words w[0..15] of the message schedule array

    Extend the first 16 words into the remaining 48 words w[16..63] of the message sched
array:
    for i from 16 to 63
        s0 := (w[i-15] rightrotate 7) xor (w[i-15] rightrotate 18) xor (w[i-15] rightshi
3)
        s1 := (w[i-2] rightrotate 17) xor (w[i-2] rightrotate 19) xor (w[i-2] rightshift
10)
        w[i] := w[i-16] + s0 + w[i-7] + s1

    Initialize working variables to current hash value:
    a := h0
    b := h1
    c := h2
    d := h3
    e := h4
    f := h5
    g := h6
    h := h7
```

```
Initialize hash values:
(first 32 bits of the fractional parts of the square roots of the first 8 primes 2..19):
h0 := 0x6a09e667
h1 := 0xbb67ae85
h2 := 0x3c6ef372
h3 := 0xa54ff53a
h4 := 0x510e527f
h5 := 0x9b05688c
h6 := 0x1f83d9ab
h7 := 0x5be0cd19

Initialize array of round constants:
(first 32 bits of the fractional parts of the cube roots of the first 64 primes 2..311):
k[0..63] :=
   0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b, 0x59f111f1, 0x923f82a4,
0xab1c5ed5,
   0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,
0xc19bf174,
   0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc, 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc,
0x76f988da,
   0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147, 0x06ca6351,
0x14292967,
   0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e,
0x92722c85,
   0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819, 0xd6990624, 0xf40e3585,
0x106aa070,
   0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,
0x682e6ff3,
   0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208, 0x90befffa, 0xa4506ceb, 0xbef9a3f7,
0xc67178f2
```

```
Compression function main loop:
for i from 0 to 63
    S1 := (e rightrotate 6) xor (e rightrotate 11) xor (e rightrotate 25)
    ch := (e and f) xor ((not e) and g)
    temp1 := h + S1 + ch + k[i] + w[i]
    S0 := (a rightrotate 2) xor (a rightrotate 13) xor (a rightrotate 22)
    maj := (a and b) xor (a and c) xor (b and c)
    temp2 := S0 + maj

    h := g
    g := f
    f := e
    e := d + temp1
    d := c
    c := b
    b := a
    a := temp1 + temp2

Add the compressed chunk to the current hash value:
h0 := h0 + a
h1 := h1 + b
h2 := h2 + c
h3 := h3 + d
h4 := h4 + e
h5 := h5 + f
h6 := h6 + g
h7 := h7 + h

Produce the final hash value (big-endian):
digest := hash := h0 append h1 append h2 append h3 append h4 append h5 append h6 append h7
```

Case Study: SHA-256

Note: infeasible to use a random function in practice.

– describing such a function requires an infinite number of bits

In practice: use function that is easy to describe + evaluate, but as unpredictable (for practical purposes) as a random function.

Canonical example: SHA-256 (used in Bitcoin, Solana, etc.).

- based on the *Merkle-Damgard approach* of iteratively applying a ”compression function” to chunks of the input

The Merkle-Damgard Approach

  - [assuming target output length = 256 bits]
- break input into chunks $x_1,x_2,\ldots,x_m$ of 512 bits each
- for i = 1,2,…,m:

The Merkle-Damgard Approach

- [assuming target output length = 256 bits]
- break input into chunks $x_1,x_2,\ldots,x_m$ of 512 bits each
- for i = 1,2,…,m:
  - $h_i := f(x_i,h_{i-1})$ [f = "compression function," h_0 = fixed default value (IV)]

The Merkle-Damgard Approach

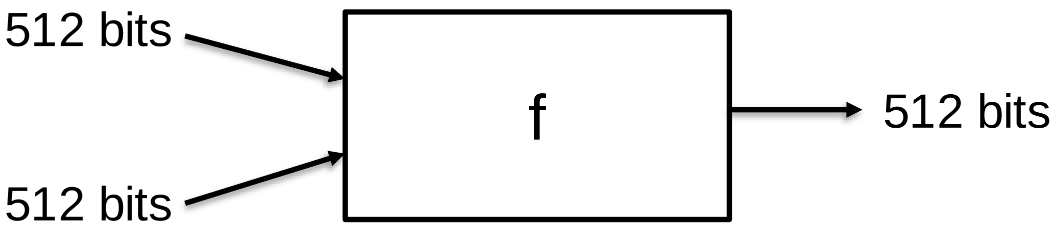
  – [assuming target output length = 256 bits]
- break input into chunks $x_1,x_2,\ldots,x_m$ of 512 bits each
- for i = 1,2,…,m:
  – $h_i := f(x_i,h_{i-1})$ [f = “compression function,” h_0 = fixed default value (IV)]
- return last 256 bits of h_m

The Merkle-Damgard Approach

- [assuming target output length = 256 bits]

- break input into chunks $x_1, x_2, \ldots, x_m$ of 512 bits each
- for i = 1,2,…,m:
  - $h_i := f(x_i, h_{i-1})$ [f = "compression function," h_0 = fixed default value (IV)]
- return last 256 bits of h_m

Compression function: resembles a block cipher (like AES).



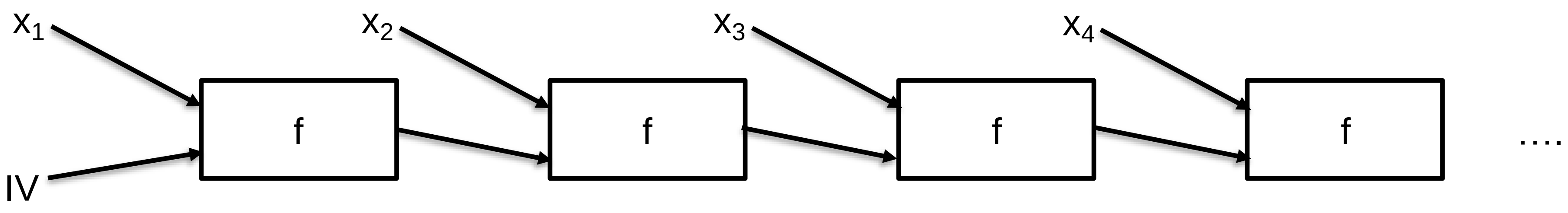
Case Study: SHA-256

Note: infeasible to use a random function in practice.

In practice: use function that is easy to describe + evaluate, but as unpredictable (for practical purposes) as a random function.

Canonical example: SHA-256 (used in Bitcoin, Solana, etc.).

- based on the *Merkle-Damgard approach* of iteratively applying a ”compression function” to chunks of the input



Cryptographic vs. Random Functions

Warning: cryptographic hash functions are *not* random functions.

- deterministic, often < 100 lines of code
- have detectable properties that random functions don't have
  - does not necessarily contradict collision-resistance

Cryptographic vs. Random Functions

Warning: cryptographic hash functions are *not* random functions.

- deterministic, often < 100 lines of code
- have detectable properties that random functions don't have
  - does not necessarily contradict collision-resistance

Example: *length-extension attacks.*

- applies to hash functions that use the Merkle-Damgard approach
- given x and h(x), can construct a y:=xz that extends x so that can compute h(y) from h(x) (rather than recomputing h(y) from scratch)

Cryptographic vs. Random Functions

Warning: cryptographic hash functions are *not* random functions.

- deterministic, often < 100 lines of code
- have detectable properties that random functions don't have
  - does not necessarily contradict collision-resistance

Example: *length-extension attacks.*

- applies to hash functions that use the Merkle-Damgard approach
- given x and h(x), can construct a y:=xz that extends x so that can compute h(y) from h(x) (rather than recomputing h(y) from scratch)
  - lethal for HMACs, not obviously relevant to blockchain protocols
  - reason why SHA-256 often applied twice in the Bitcoin protocol?

Cryptographic vs. Random Functions

Warning: cryptographic hash functions are *not* random functions.

Example: *length-extension attacks.*

- applies to hash functions that use the Merkle-Damgard approach
- given x and h(x), can construct a y:=xz that extends x so that can compute h(y) from h(x) (rather than recomputing h(y) from scratch)

Tl;dr: practitioners treat cryptographic hash functions like SHA-256 as random functions, even though they're not.

- typically can get away with it, though beware of important edge cases

Compression via Hash Functions

Recall: in e.g. Tendermint, validators pass around entire chains.

Protocol D (≈ Tendermint)

- at time $4\Delta \cdot v$:
  - each validator i sends its current chain A_i to v's leader ℓ
- at time $4\Delta \cdot v + \Delta$:
  - let A = of the A_i's received, the most recently created one; let B := all not-yet-included (in A) valid txs ℓ knows about
  - ℓ sends proposal (A,B) to all other validators
- at time $4\Delta \cdot v + 2\Delta$:
  - if validator i receives a proposal (A,B) from ℓ with A = A_i or with A more recent than A_i by this time:
    - send "(A,B) is up-to-date" message to all validators
- at time $4\Delta \cdot v + 3\Delta$:
  - if validator i has heard > 2n/3 "up-to-date" msgs for (A,B) by this time (a *read quorum)*:
    - package these messages into a quorum certificate (QC), Q
    - send "ack (A,B,Q)" message to all validators and reset A_i := (A,B,Q)
- at time $4\Delta \cdot v + 4\Delta$:
  - if validator i has received > 2n/3 "ack (A,B,Q)" messages (a *write quorum)*:
    - reset C_i := (A,B,Q) (and also A_i := (A,B,Q), if necessary)

Compression via Hash Functions

Recall: in e.g. Tendermint, validators pass around entire chains.

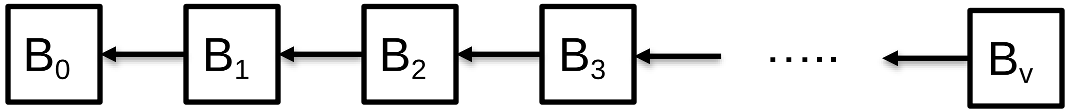
To make practical:

Compression via Hash Functions

Recall: in e.g. Tendermint, validators pass around entire chains.

To make practical:

- every block specifies a predecessor via hash of latter
  - – additional metadata, required for block to be viewed as valid



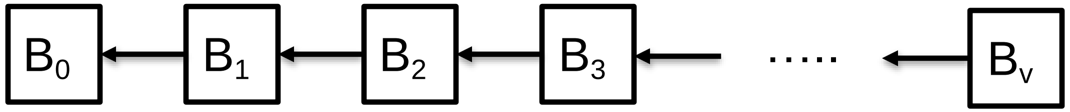
“hash pointers”

Compression via Hash Functions

Recall: in e.g. Tendermint, validators pass around entire chains.

To make practical:

- every block specifies a predecessor via hash of latter
  - additional metadata, required for block to be viewed as valid



"hash pointers"

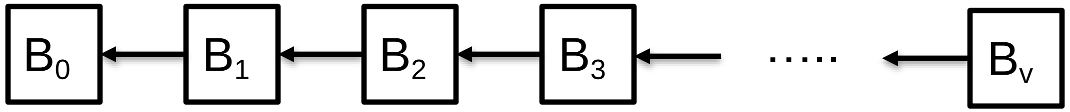
- leader proposes a block B, not a chain

Compression via Hash Functions

Recall: in e.g. Tendermint, validators pass around entire chains.

To make practical:

- every block specifies a predecessor via hash of latter
  - additional metadata, required for block to be viewed as valid



"hash pointers"

- leader proposes a block B, not a chain
- "up-to-date" and "ack" messages reference has h(B), not B
  - quorum certificates attest to a blockhash, not a block (or a chain)

Protocol D (≈ Tendermint)

- at time $4\Delta \cdot v$:
  - each validator i sends its current chain A_i to v's leader ℓ
- at time $4\Delta \cdot v + \Delta$:
  - let A = of the A_i's received, the most recently created one; let B := all not-yet-included (in A) valid txs ℓ knows about
  - ℓ sends proposal (A,B) to all other validators
- at time $4\Delta \cdot v + 2\Delta$:
  - if validator i receives a proposal (A,B) from ℓ with A = A_i or with A more recent than A_i by this time:
    - send "(A,B) is up-to-date" message to all validators
- at time $4\Delta \cdot v + 3\Delta$:
  - if validator i has heard > 2n/3 "up-to-date" msgs for (A,B) by this time (a *read quorum)*:
    - package these messages into a quorum certificate (QC), Q
    - send "ack (A,B,Q)" message to all validators and reset A_i := (A,B,Q)
- at time $4\Delta \cdot v + 4\Delta$:
  - if validator i has received > 2n/3 "ack (A,B,Q)" messages (a *write quorum)*:
    - reset C_i := (A,B,Q) (and also A_i := (A,B,Q), if necessary)

Compression via Hash Functions

To make practical:

- every block specifies a predecessor via hash of latter
- leader proposes a block B, not a chain
- “up-to-date” and “ack” messages reference has h(B), not B

Compression via Hash Functions

To make practical:

- every block specifies a predecessor via hash of latter
- leader proposes a block B, not a chain
- "up-to-date" and "ack" messages reference has h(B), not B

Benefit: size of blockhash << size of block << size of chain.

Compression via Hash Functions

To make practical:

- every block specifies a predecessor via hash of latter
- leader proposes a block B, not a chain
- "up-to-date" and "ack" messages reference has h(B), not B

Benefit: size of blockhash << size of block << size of chain.

Issue: validator may know blockhash before corresponding block.

  - e.g., the predecessor blockhash in the current leader's block proposal
  - e.g., the blockhash referenced in "up-to-date" and "ack" messages

Cryptographic Hashes Are Binding

Issue: validator may know blockhash before corresponding block.

Cryptographic Hashes Are Binding

Issue: validator may know blockhash before corresponding block.

Note: collision-resistance of hash fn ➔ can't reverse engineer block from blockhash. [i.e., collision-resistance ➔ one-way function]

- thus: validator must obtain the full block from some other source

Cryptographic Hashes Are Binding

Issue: validator may know blockhash before corresponding block.

Note: collision-resistance of hash fn ➔ can't reverse engineer block from blockhash. [i.e., collision-resistance ➔ one-way function]

- thus: validator must obtain the full block from some other source

Worry: could an untrusted source fabricate the block?

Cryptographic Hashes Are Binding

Issue: validator may know blockhash before corresponding block.

Note: collision-resistance of hash fn ➔ can't reverse engineer block from blockhash. [i.e., collision-resistance ➔ one-way function]

- thus: validator must obtain the full block from some other source

Worry: could an untrusted source fabricate the block?

Good news: no, would contradict collision-resistance of hash fn.

  – collision-resistant ➔ "second pre-image resistant"

Cryptographic Commitments

Terminology: hash h(x) is a *commitment* to x.

- analogy: object in a locked box

Cryptographic Commitments

Terminology: hash h(x) is a *commitment* to x.

- analogy: object in a locked box
- *binding*: after announcing h(x), can't later pretend to have committed to some $y \neq x$ (collision-resistance ➔ $h(y) \neq h(x)$)

Cryptographic Commitments

Terminology: hash h(x) is a *commitment* to x.

- analogy: object in a locked box
- *binding*: after announcing h(x), can't later pretend to have committed to some y ≠ x (collision-resistance ➔ h(y) ≠ h(x))
- *hiding:* knowing only h(x), don't know x until it's revealed
  - can be important e.g. for privacy in some applications

Cryptographic Commitments

Terminology: hash h(x) is a *commitment* to x.

- analogy: object in a locked box
- *binding*: after announcing h(x), can't later pretend to have committed to some y ≠ x (collision-resistance ➔ h(y) ≠ h(x))
- *hiding:* knowing only h(x), don't know x until it's revealed
  - can be important e.g. for privacy in some applications

Worry: what if validator can't find source for block?

- need to ensure this never happens ("data availability")