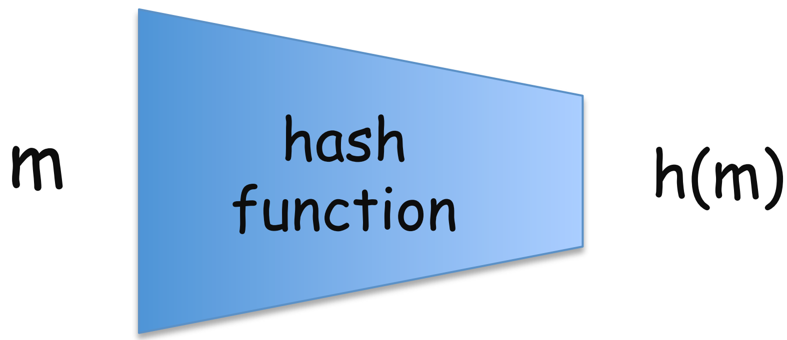
Cryptographic Foundations

Murat Osmanoglu

Hash Functions

- maps inputs of some length to short, fixed-length output in deterministic

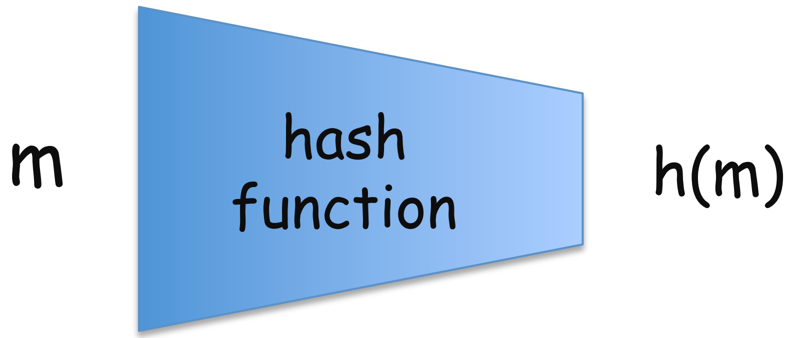


sometimes called digest

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

Hash Functions

- maps inputs of some length to short, fixed-length output in deterministic



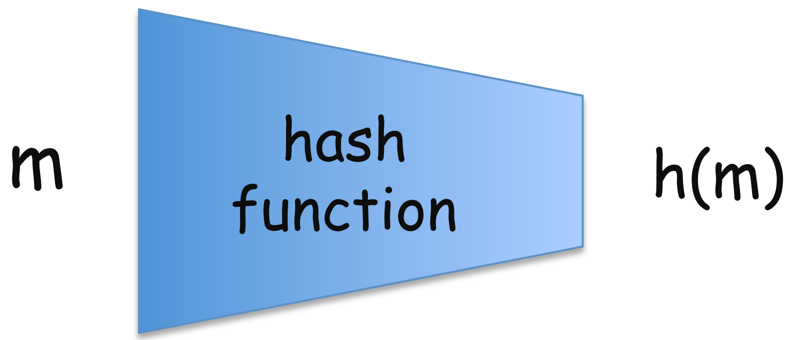
sometimes called digest

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

- originally proposed to provide input to digital signature schemes, by Diffie-Hellman in 1976

Hash Functions

- maps inputs of some length to short, fixed-length output in deterministic



sometimes called digest

$$h : \{0,1\}^* \rightarrow \{0,1\}^n$$

- originally proposed to provide input to digital signature schemes, by Diffie-Hellman in 1976
- security features for hash functions

  pre-image resistance,

  weak collusion resistance,

  collusion resistance

Hash Functions

- **pre-image resistance;** given d, it should be hard to find a message m such that h(m) = d

  it is required in Proof of Work algorithm in Bitcoin, i.e. if the underlying hash functions does not satisfy that feature, it would be much easier to solve the cryptographic puzzle to create blocks

Hash Functions

- **pre-image resistance;** given d, it should be hard to find a message m such that $h(m) = d$

  it is required in Proof of Work algorithm in Bitcoin, i.e. if the underlying hash functions does not satisfy that feature, it would be much easier to solve the cryptographic puzzle to create blocks

- **weak collision resistance;** given m_1, it should be hard to find m_2 such that $h(m_1) = h(m_2)$

Hash Functions

- **pre-image resistance;** given d, it should be hard to find a message m such that h(m) = d

  it is required in Proof of Work algorithm in Bitcoin, i.e. if the underlying hash functions does not satisfy that feature, it would be much easier to solve the cryptographic puzzle to create blocks

- **weak collision resistance;** given m_1, it should be hard to find m_2 such that $h(m_1) = h(m_2)$

- **strong collision resistance;** it should be hard to find $m_1 \neq m_2$ such that $h(m_1) = h(m_2)$

  it is required for a digital signature scheme to provide non-repudiation, i.e. the signer can produce two messages m_1 and m_2, and signs one of them. Later he can deny his signature and claim he signed the other one
  it is required for an immutable distributed ledger

Hash Functions

- **pre-image resistance;** given d, it should be hard to find a message m such that $h(m) = d$

  it is required in Proof of Work algorithm in Bitcoin, i.e. if the underlying hash functions does not satisfy that feature, it would be much easier to solve the cryptographic puzzle to create blocks

- **weak collision resistance;** given m_1, it should be hard to find m_2 such that $h(m_1) = h(m_2)$

- **strong collision resistance;** it should be hard to find $m_1 \neq m_2$ such that $h(m_1) = h(m_2)$

  it … re … , ar … n he si …

> - since the domain is larger than the range, the collision must exist
> - but, if the range is large enough, it is computationally hard to find collisions

Applications

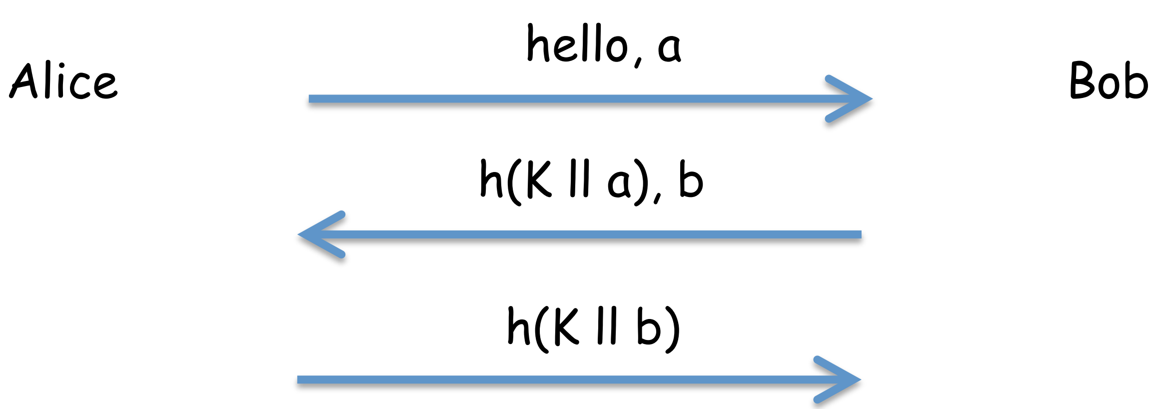
- **Virus fingerprinting**
  - keep a database containing the hashes of known viruses
  - look up the hash of a downloaded application or an email attachment in the database to detect a virus
  - for each virus, a short string needs to be stored, thus the overhead is feasible

Applications

- **Virus fingerprinting**
  - keep a database containing the hashes of known viruses
  - look up the hash of a downloaded application or an email attachment in the database to detect a virus
  - for each virus, a short string needs to be stored, thus the overhead is feasible
- **Password Protection**
  - store the hash of the password instead of password itself in a file
  - when users enter the passwords, check whether the hash equals the value stored in the corresponding file before granting the access

Applications

- **Virus fingerprinting**
  - keep a database containing the hashes of known viruses
  - look up the hash of a downloaded application or an email attachment in the database to detect a virus
  - for each virus, a short string needs to be stored, thus the overhead is feasible
- **Password Protection**
  - store the hash of the password instead of password itself in a file
  - when users enter the passwords, check whether the hash equals the value stored in the corresponding file before granting the access
- **Authentication Protocol**

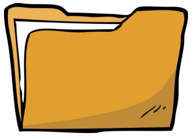


Applications

Merkle Tree

- check the integrity of a file using hash function

Client **Server**

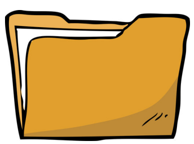


Applications

Merkle Tree

- check the integrity of a file using hash function

Client **Server**



H(F) = d

keep d

Applications

Merkle Tree

- check the integrity of a file using hash function

Client **Server**

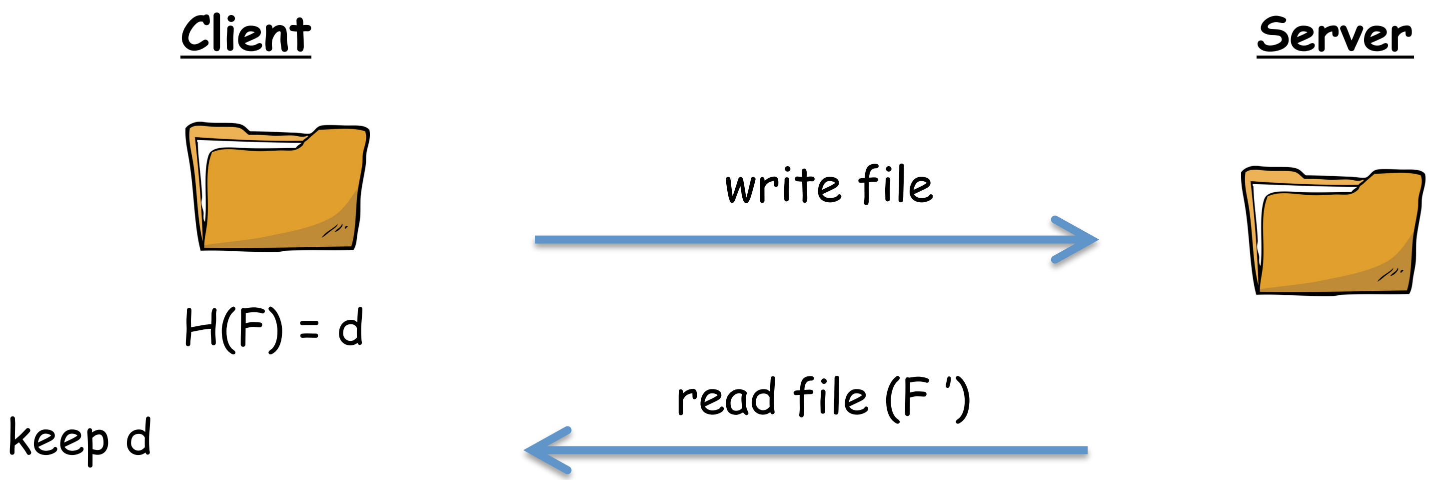
H(F) = d

keep d

Applications

Merkle Tree

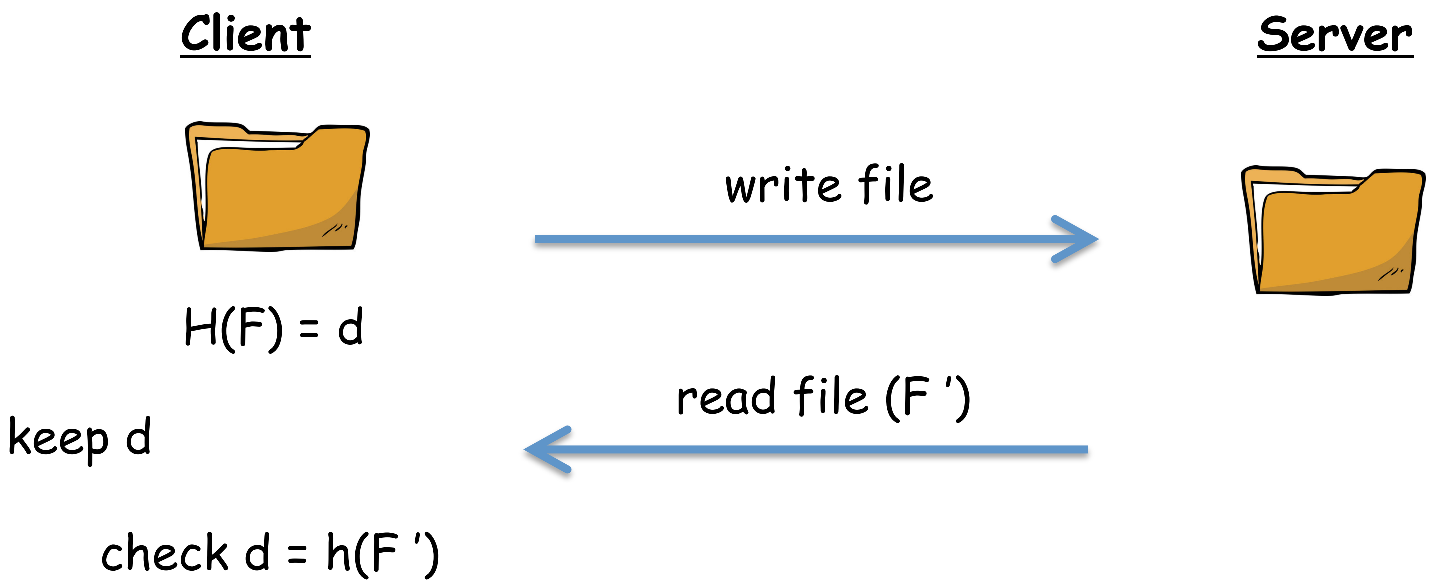
- check the integrity of a file using hash function



Applications

Merkle Tree

- check the integrity of a file using hash function

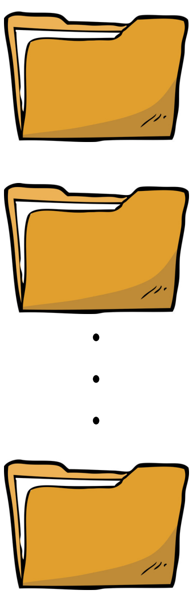


Applications

Merkle Tree

- check the integrity of multiple files using hash function

Client **Server**

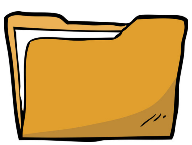


Applications

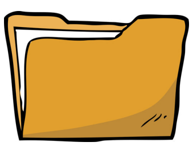
Merkle Tree

- check the integrity of multiple files using hash function

Client **Server**

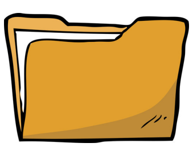


$H(F_1) = d_1$



$H(F_2) = d_2$

⋮ ⋮



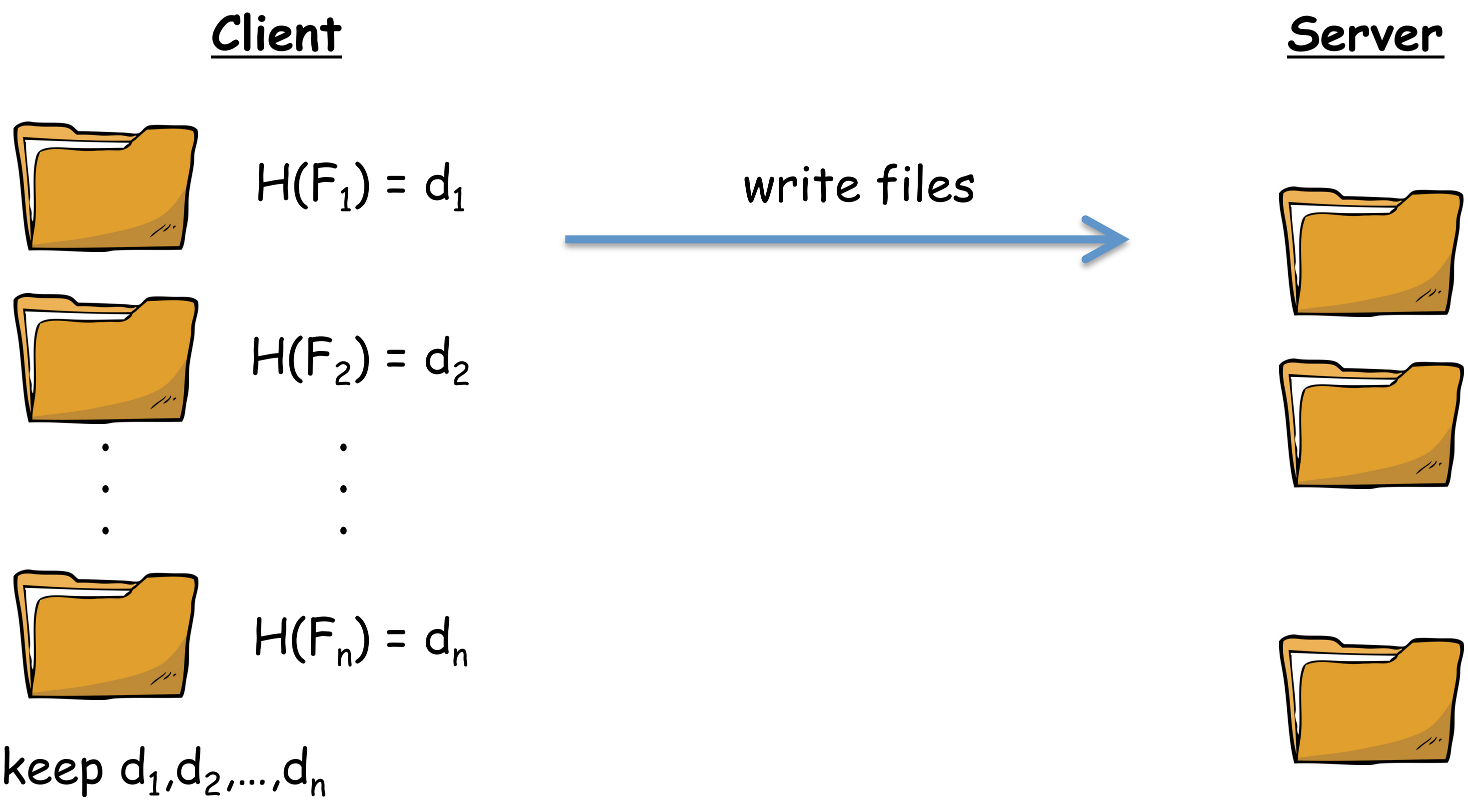
$H(F_n) = d_n$

keep $d_1, d_2, \ldots, d_n$

Applications

Merkle Tree

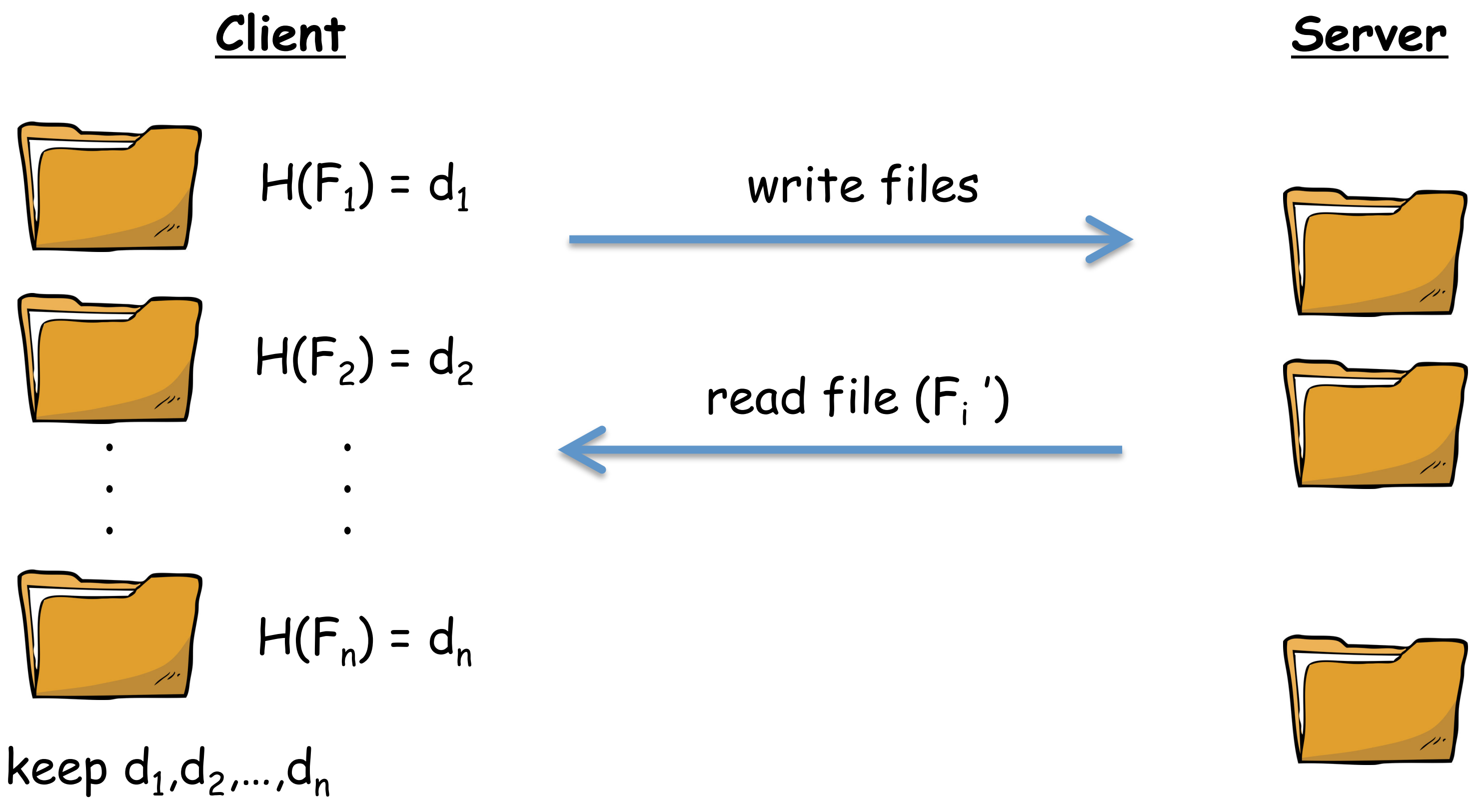
- check the integrity of multiple files using hash function



Applications

Merkle Tree

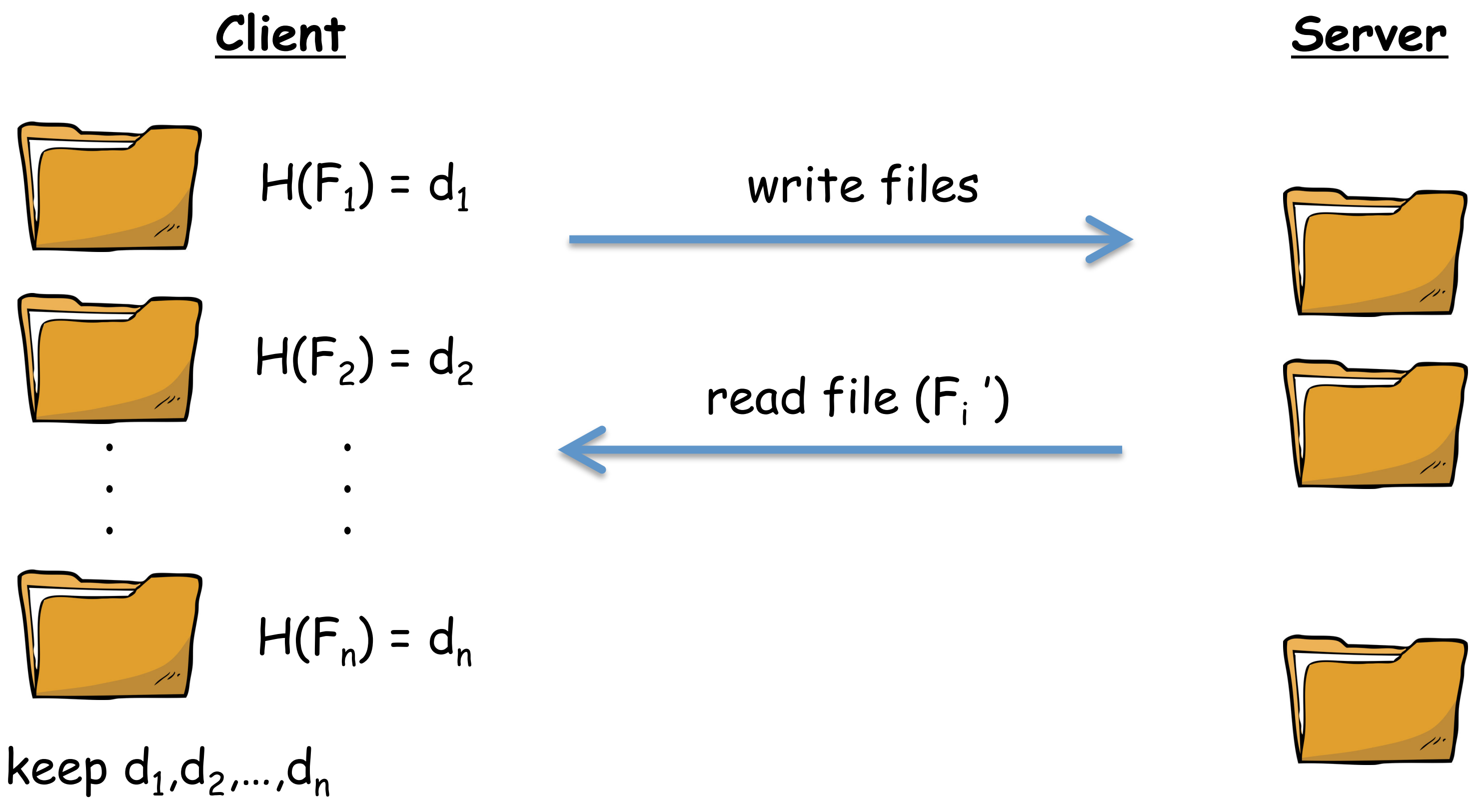
- check the integrity of multiple files using hash function



Applications

Merkle Tree

- check the integrity of multiple files using hash function

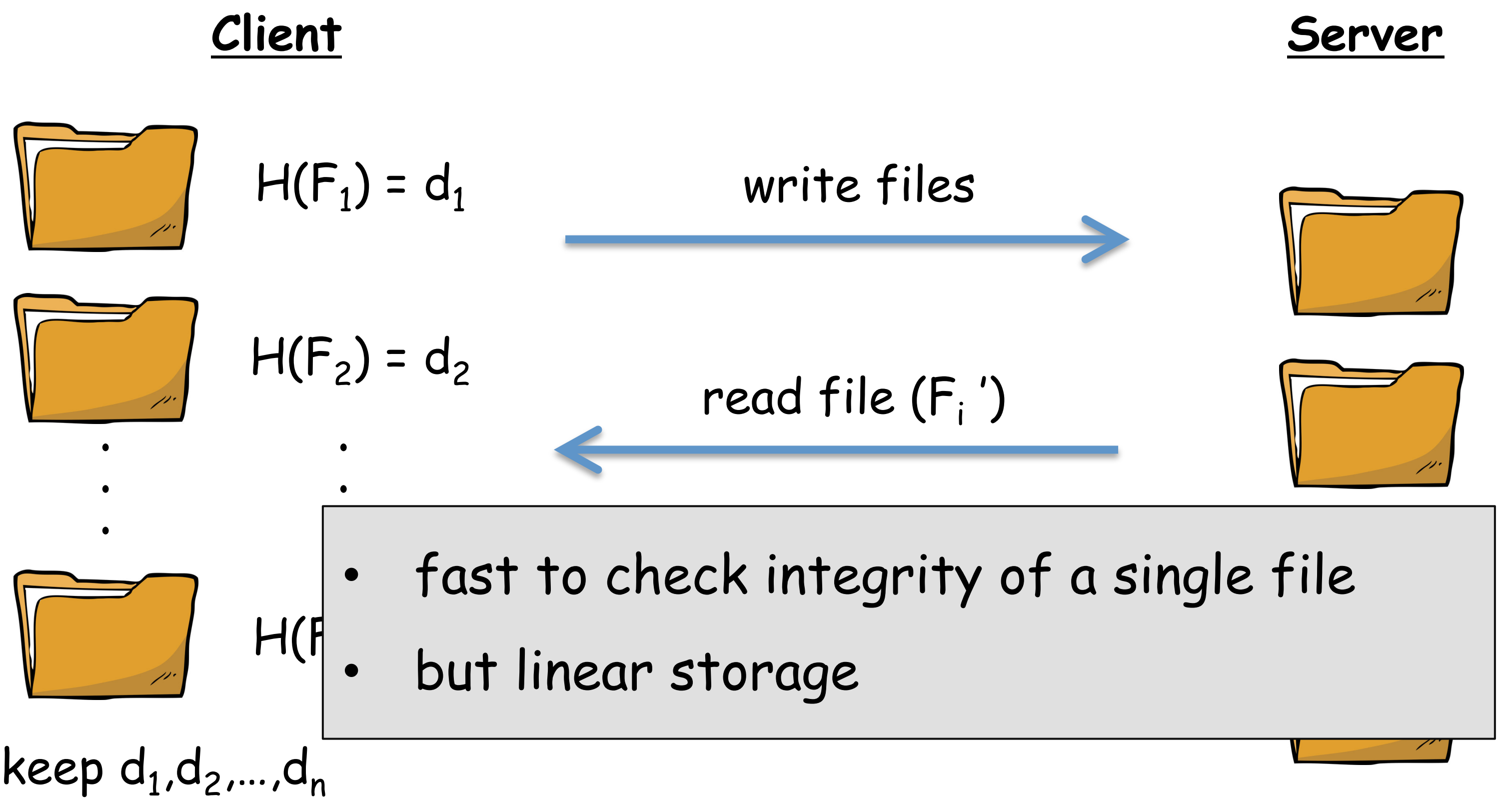


check $d_i = h(F_i')$ for each i

Applications

Merkle Tree

- check the integrity of multiple files using hash function



- fast to check integrity of a single file
- but linear storage

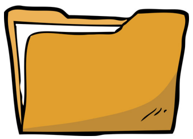
check $d_i = h(F_i')$ for each i

Applications

Merkle Tree

- check the integrity of multiple files using hash function

Client **Server**



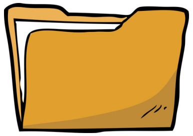
Applications

Merkle Tree

- check the integrity of multiple files using hash function

Client **Server**

keep D

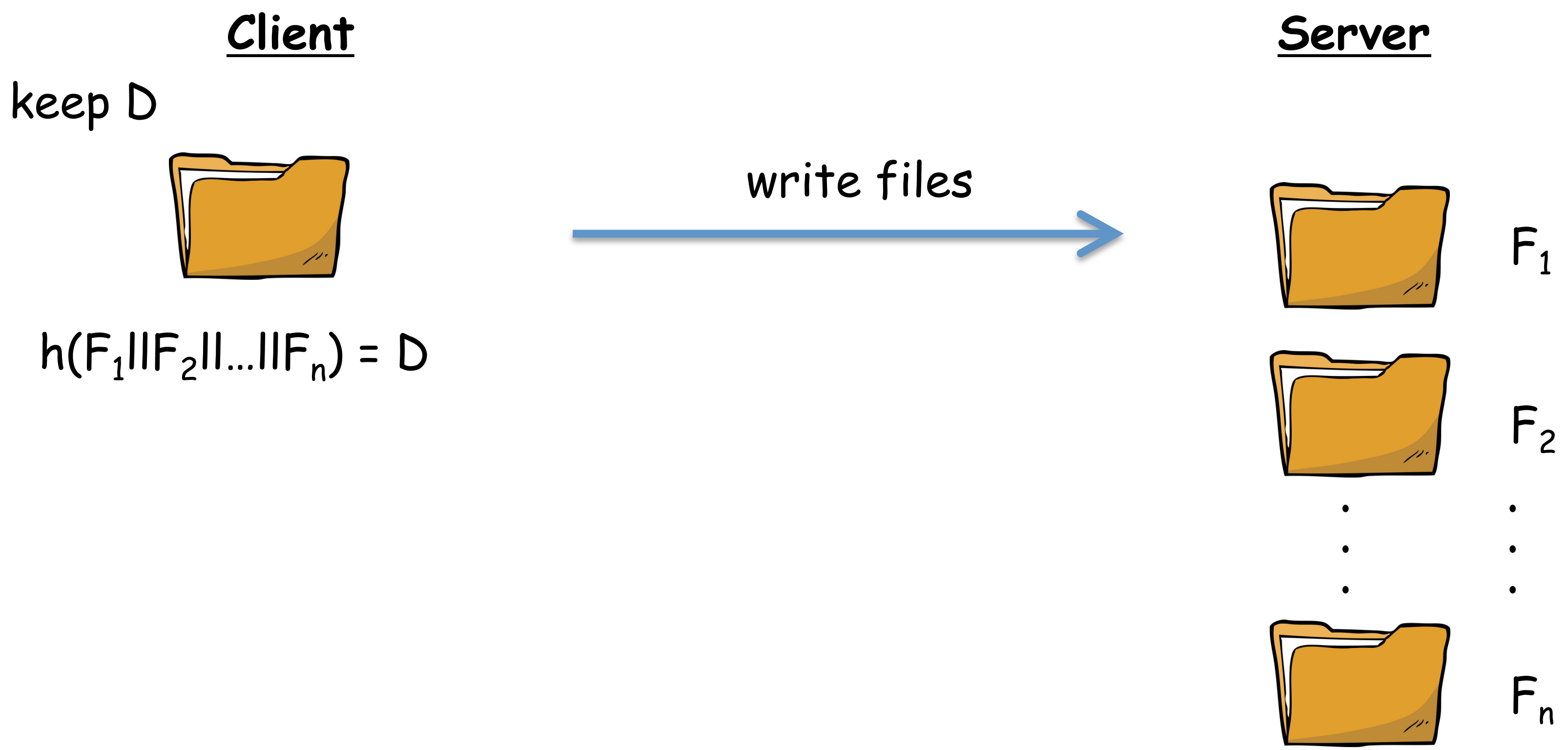


$h(F_1||F_2||...||F_n) = D$

Applications

Merkle Tree

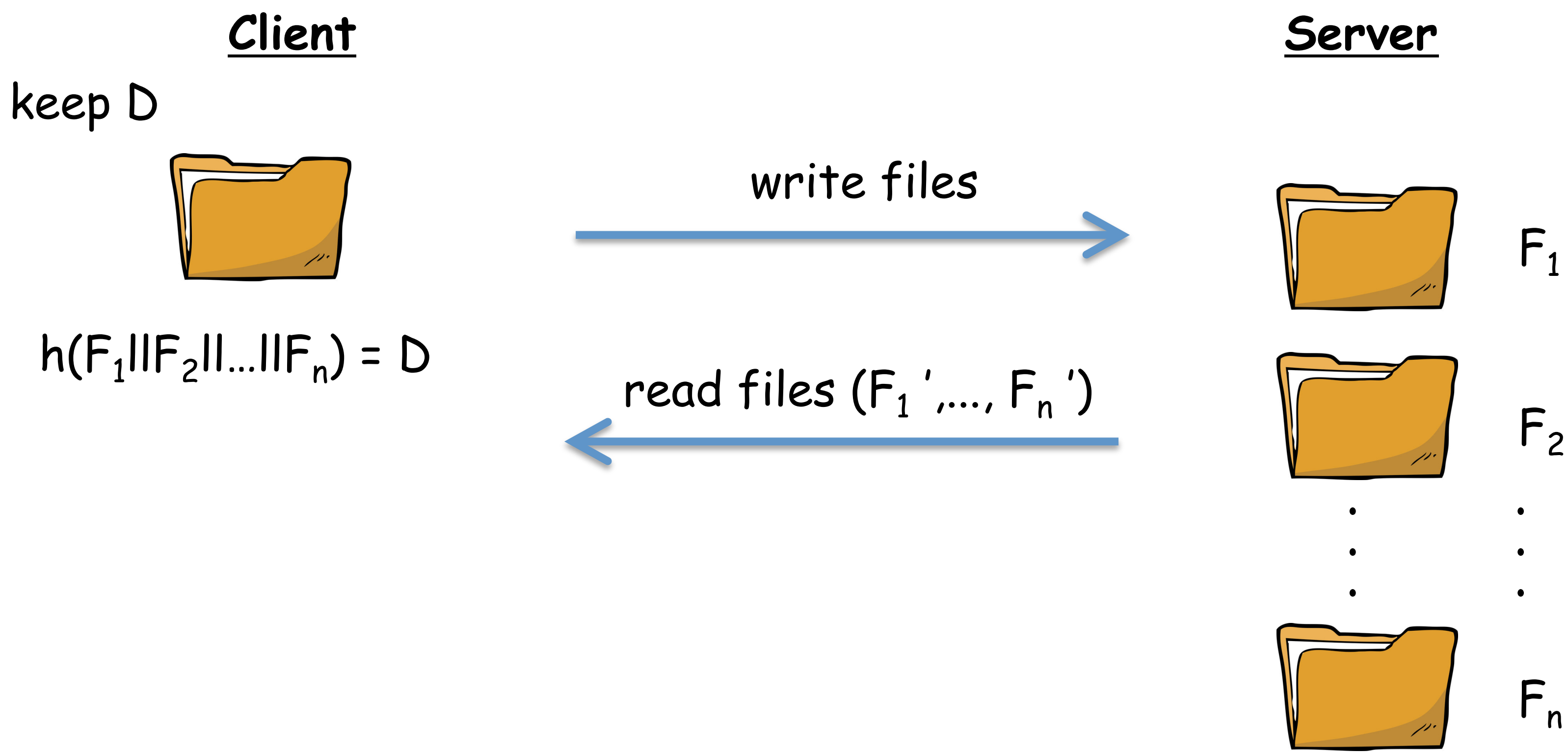
- check the integrity of multiple files using hash function



Applications

Merkle Tree

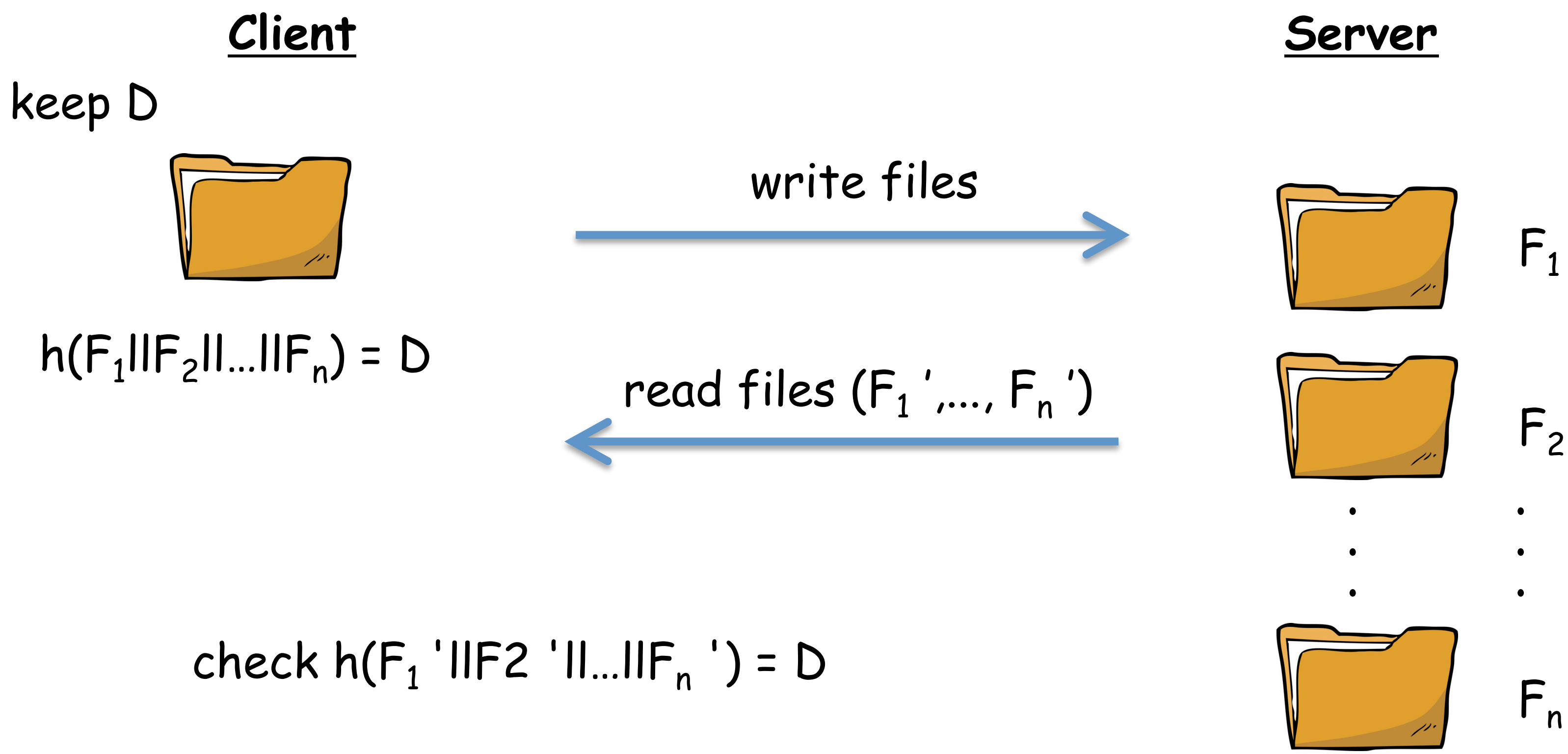
- check the integrity of multiple files using hash function



Applications

Merkle Tree

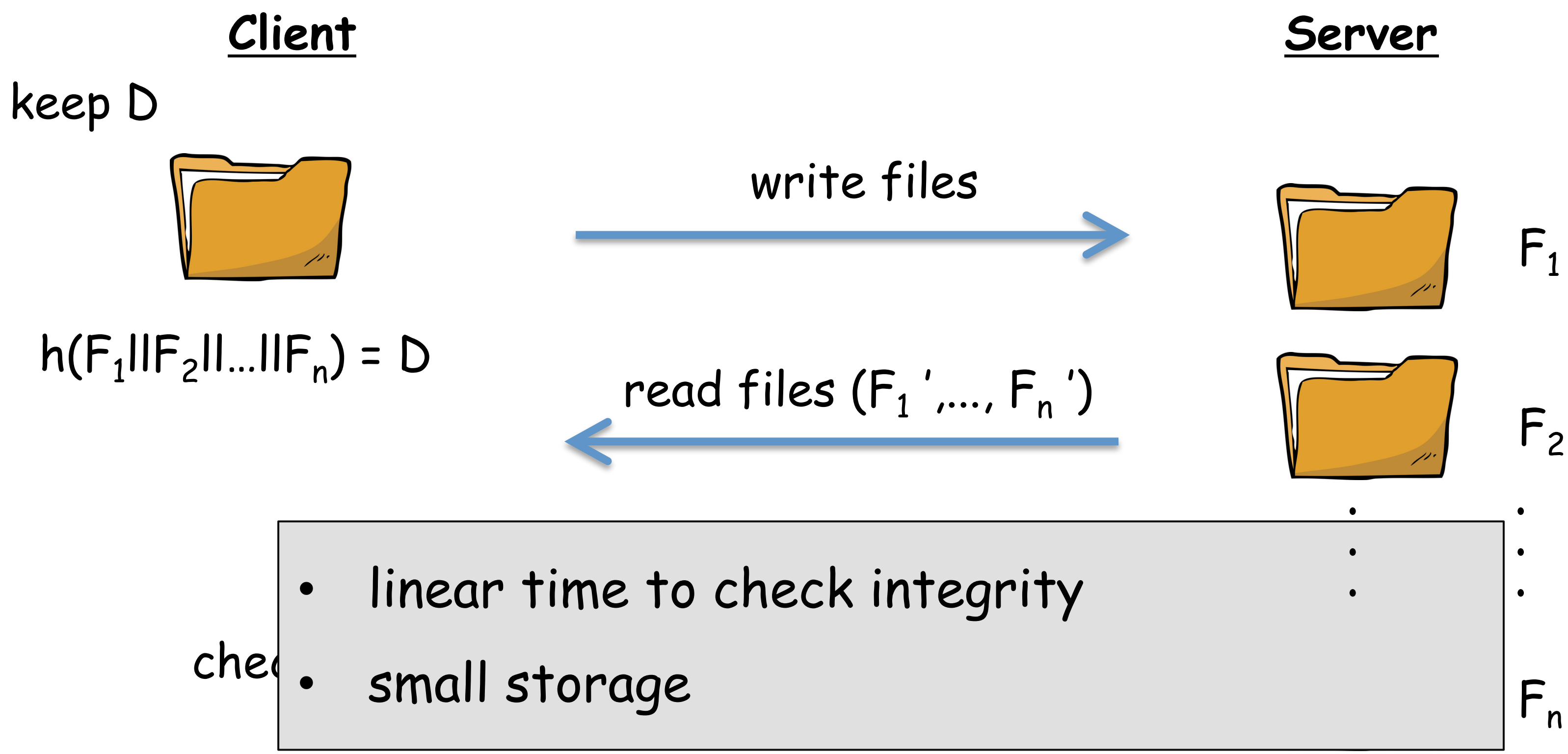
- check the integrity of multiple files using hash function



Applications

Merkle Tree

- check the integrity of multiple files using hash function



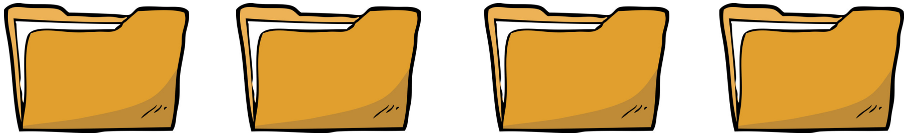
- linear time to check integrity
- small storage

Applications

Merkle Tree

- check the integrity of multiple files using hash function

Client **Server**



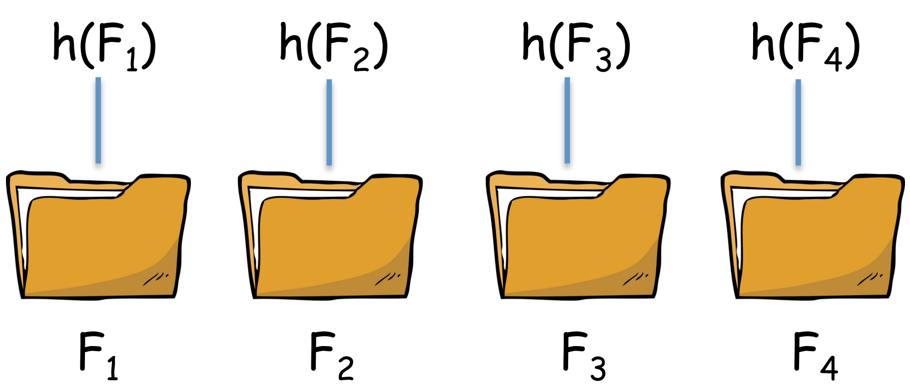
F_1 F_2 F_3 F_4

Applications

Merkle Tree

- check the integrity of multiple files using hash function

Client **Server**

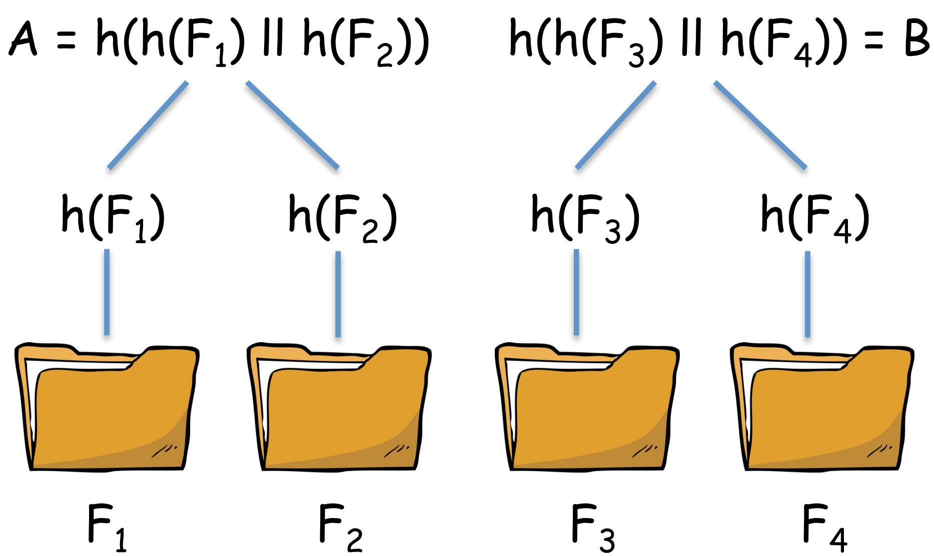


Applications

Merkle Tree

- check the integrity of multiple files using hash function

Client **Server**



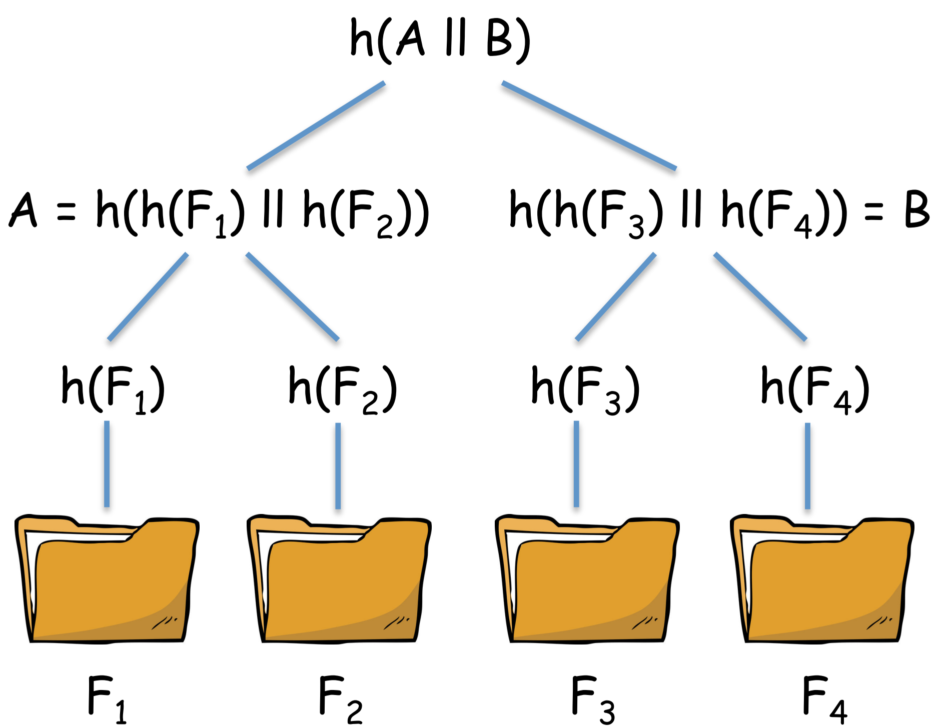
Applications

Merkle Tree

- check the integrity of multiple files using hash function

Client | **Server**

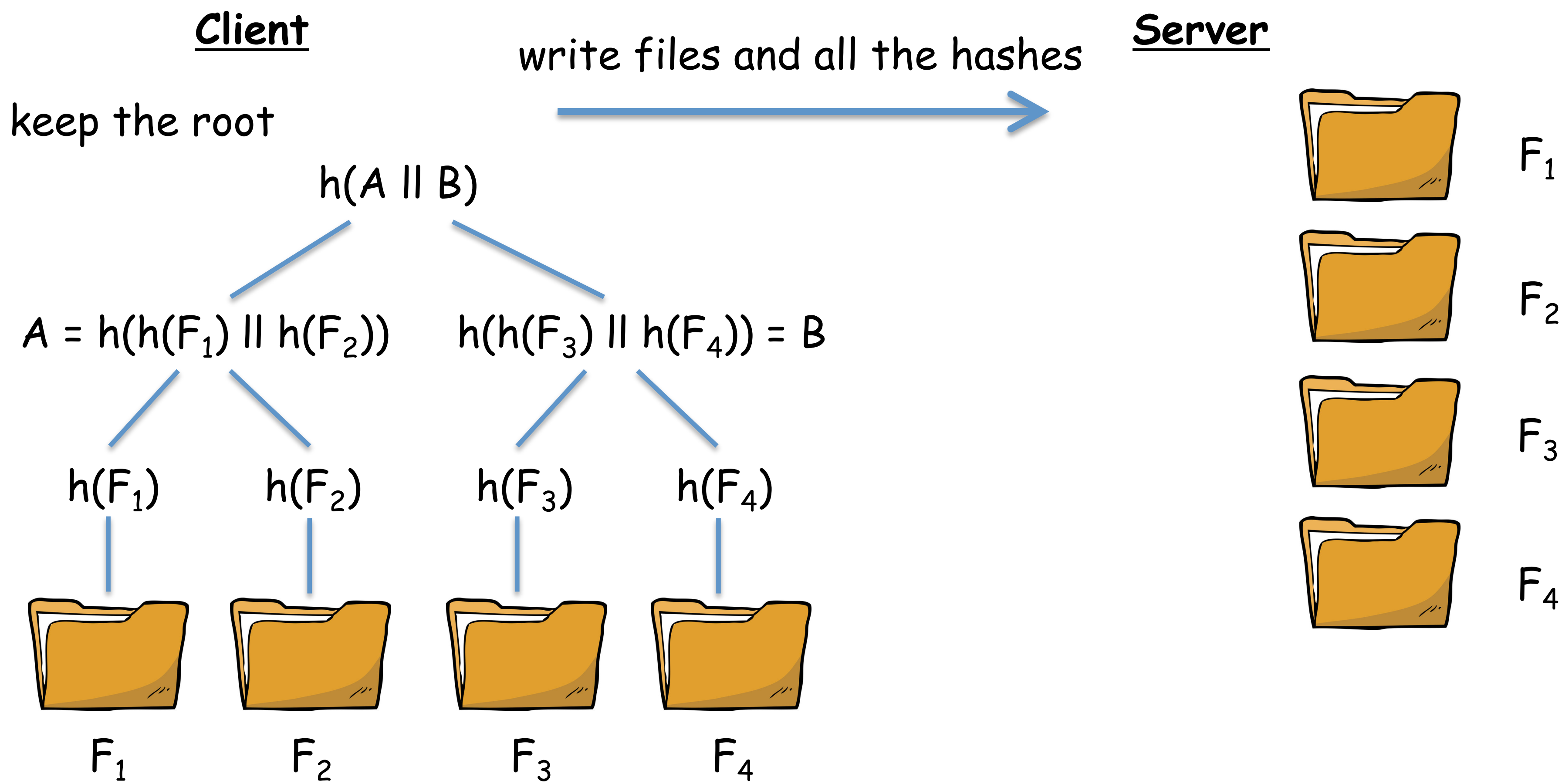
keep the root



Applications

Merkle Tree

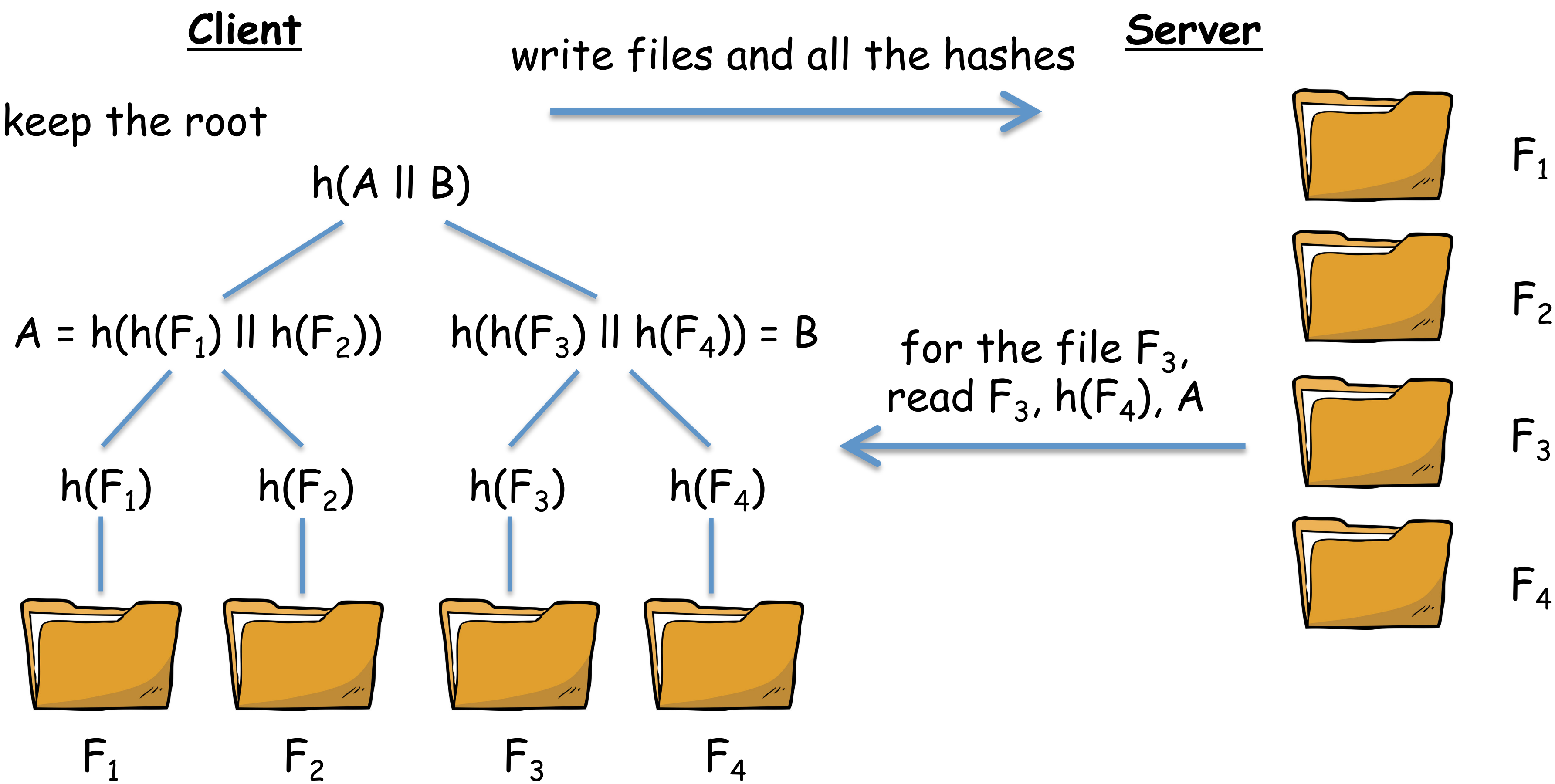
- check the integrity of multiple files using hash function



Applications

Merkle Tree

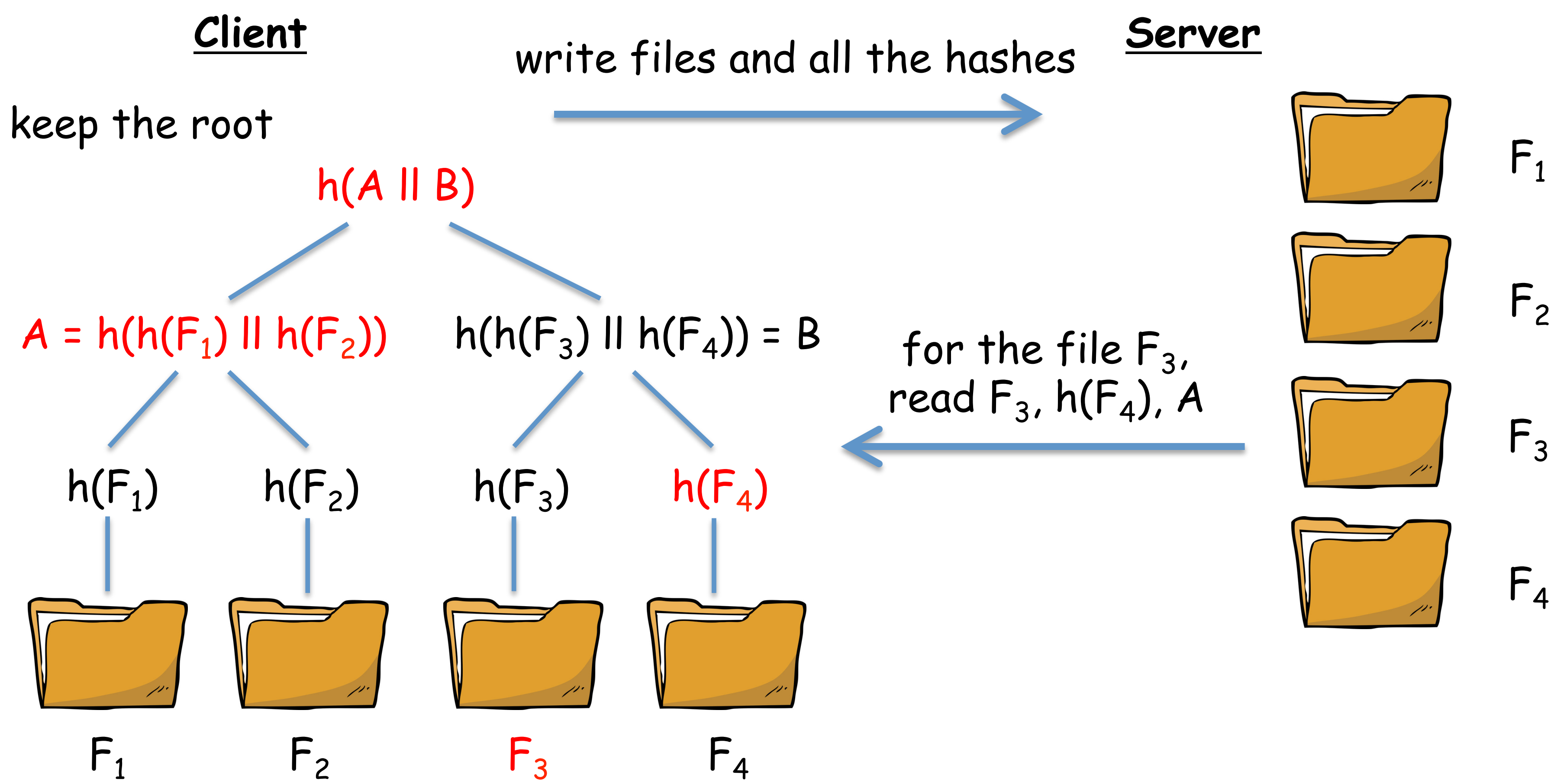
- check the integrity of multiple files using hash function



Applications

Merkle Tree

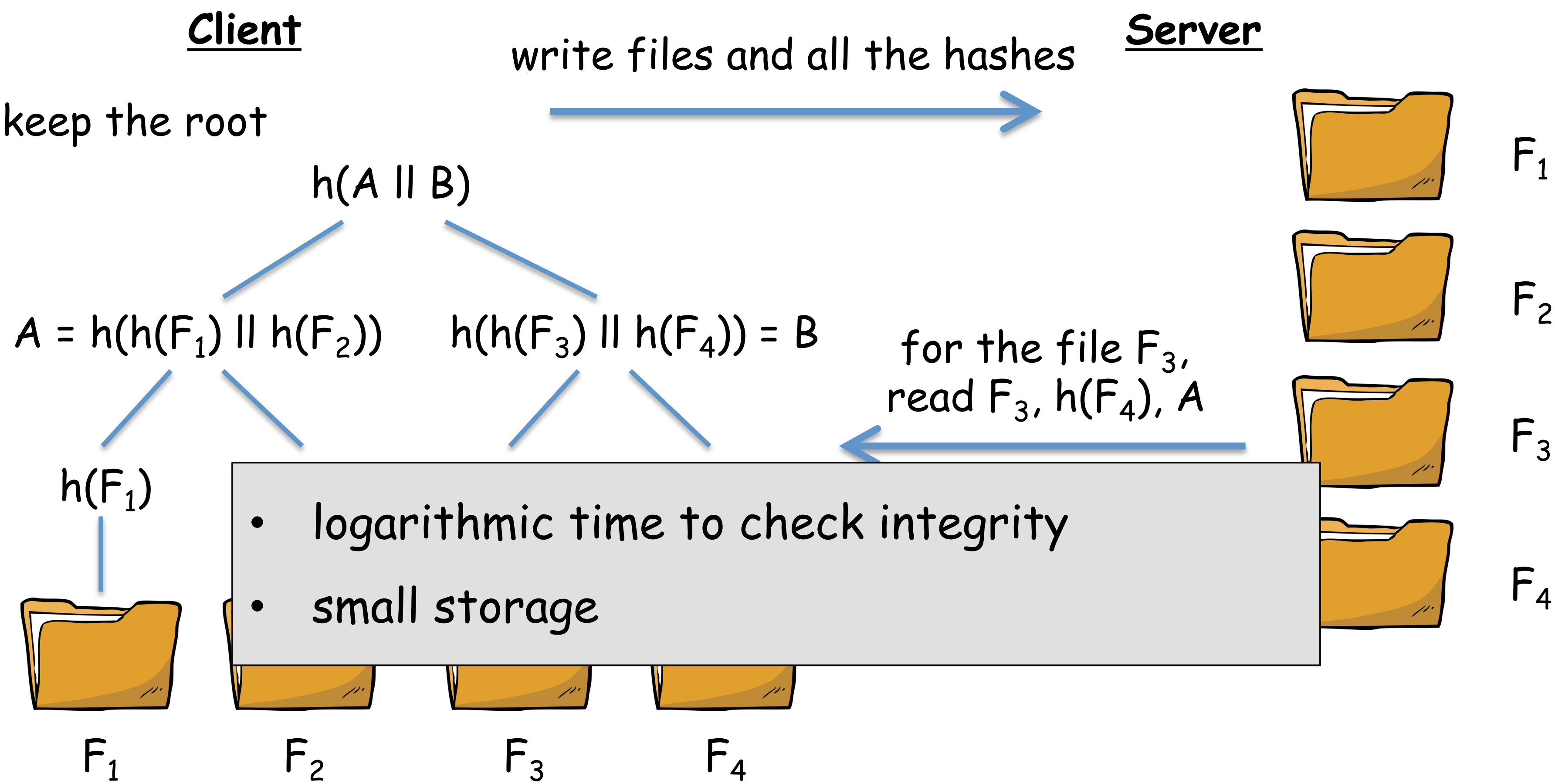
- check the integrity of multiple files using hash function



Applications

Merkle Tree

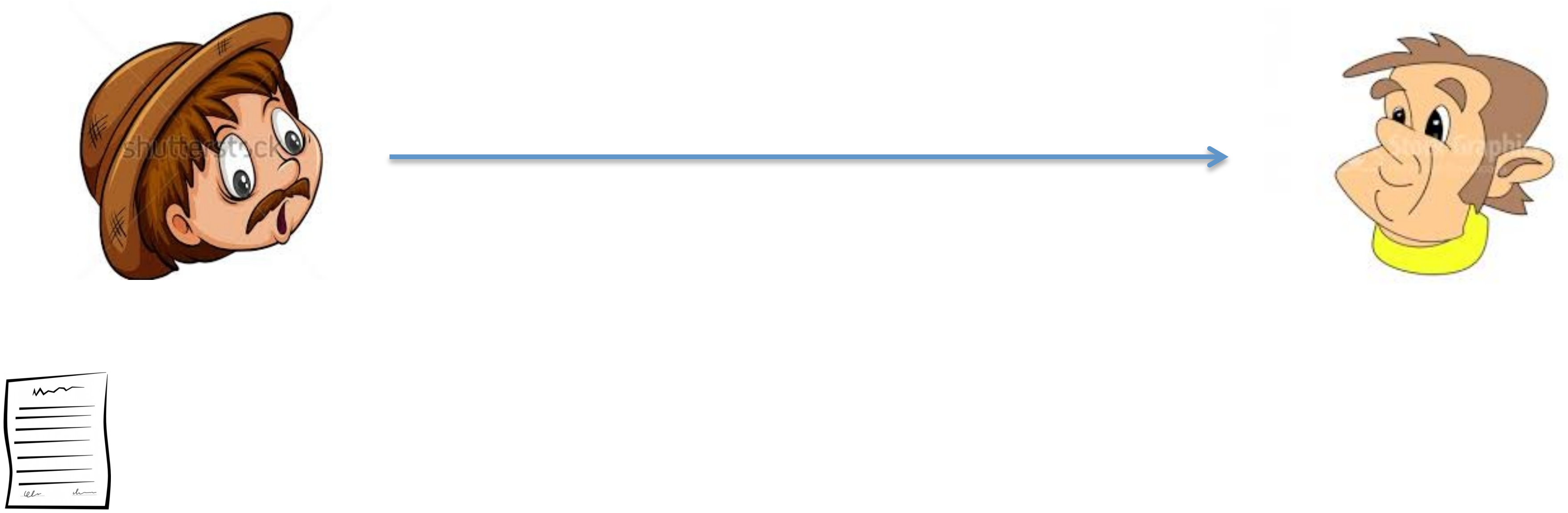
- check the integrity of multiple files using hash function



- logarithmic time to check integrity
- small storage

Digital Signature Scheme

signing by hand

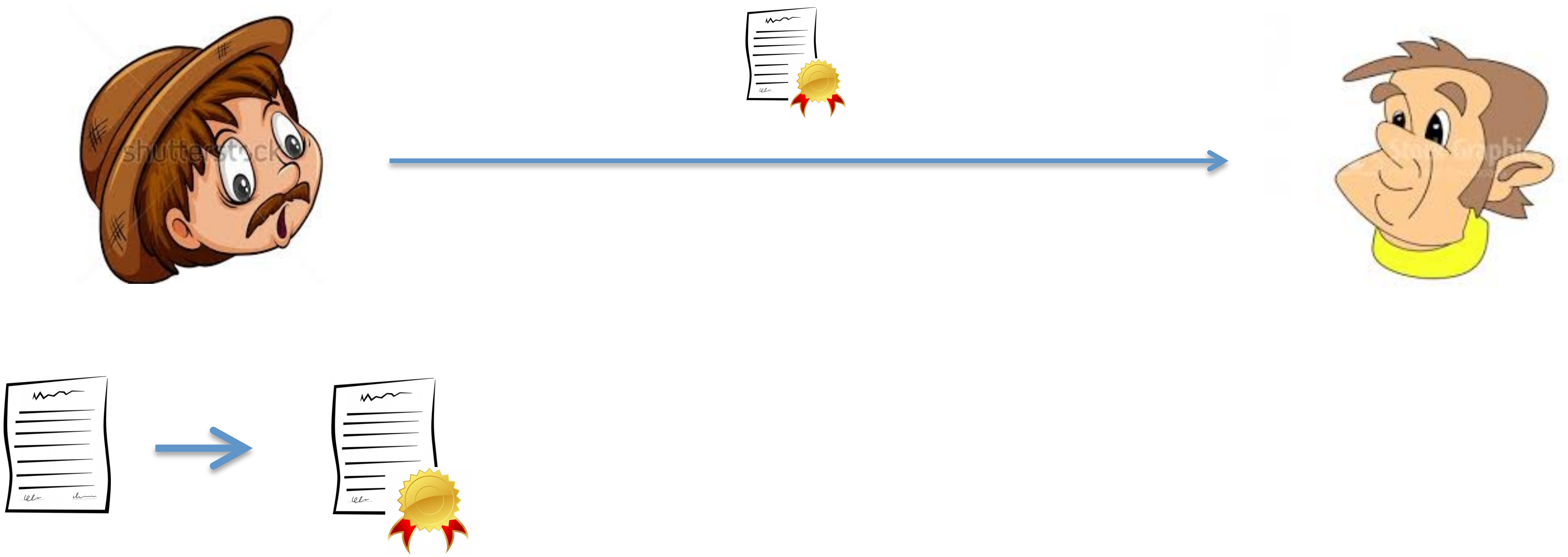


Digital Signature Scheme

signing by hand

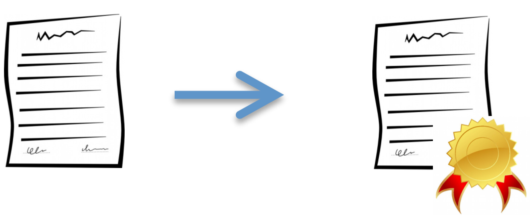
Digital Signature Scheme

signing by hand



Digital Signature Scheme

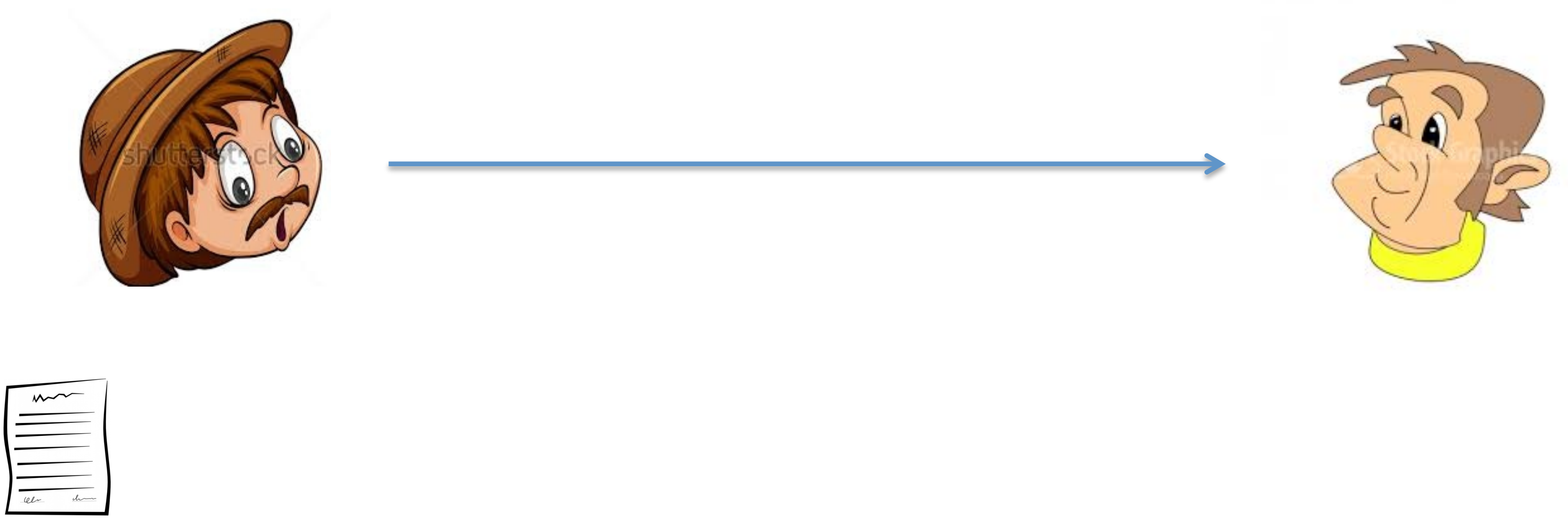
signing by hand



verify the signature

Digital Signature Scheme

signing electronically



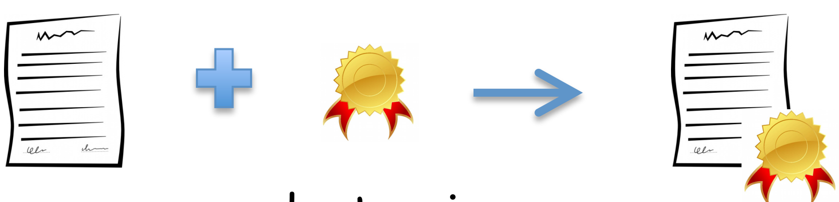
Digital Signature Scheme

signing electronically

electronic
signature

Digital Signature Scheme

signing electronically

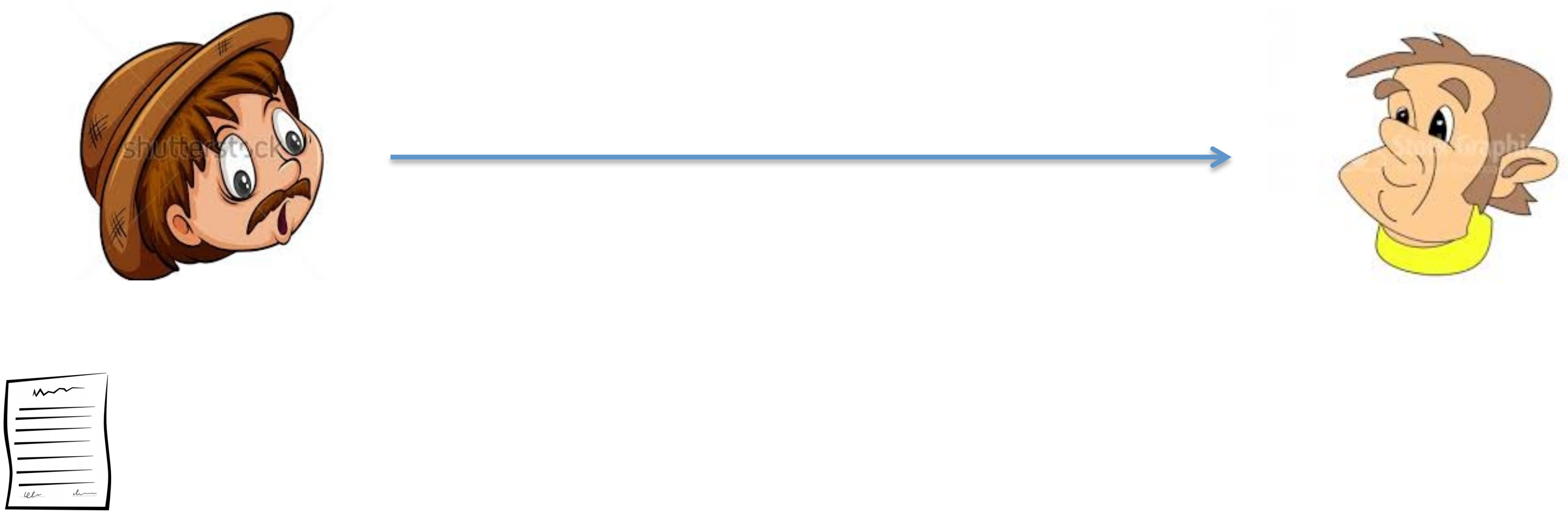


electronic
signature

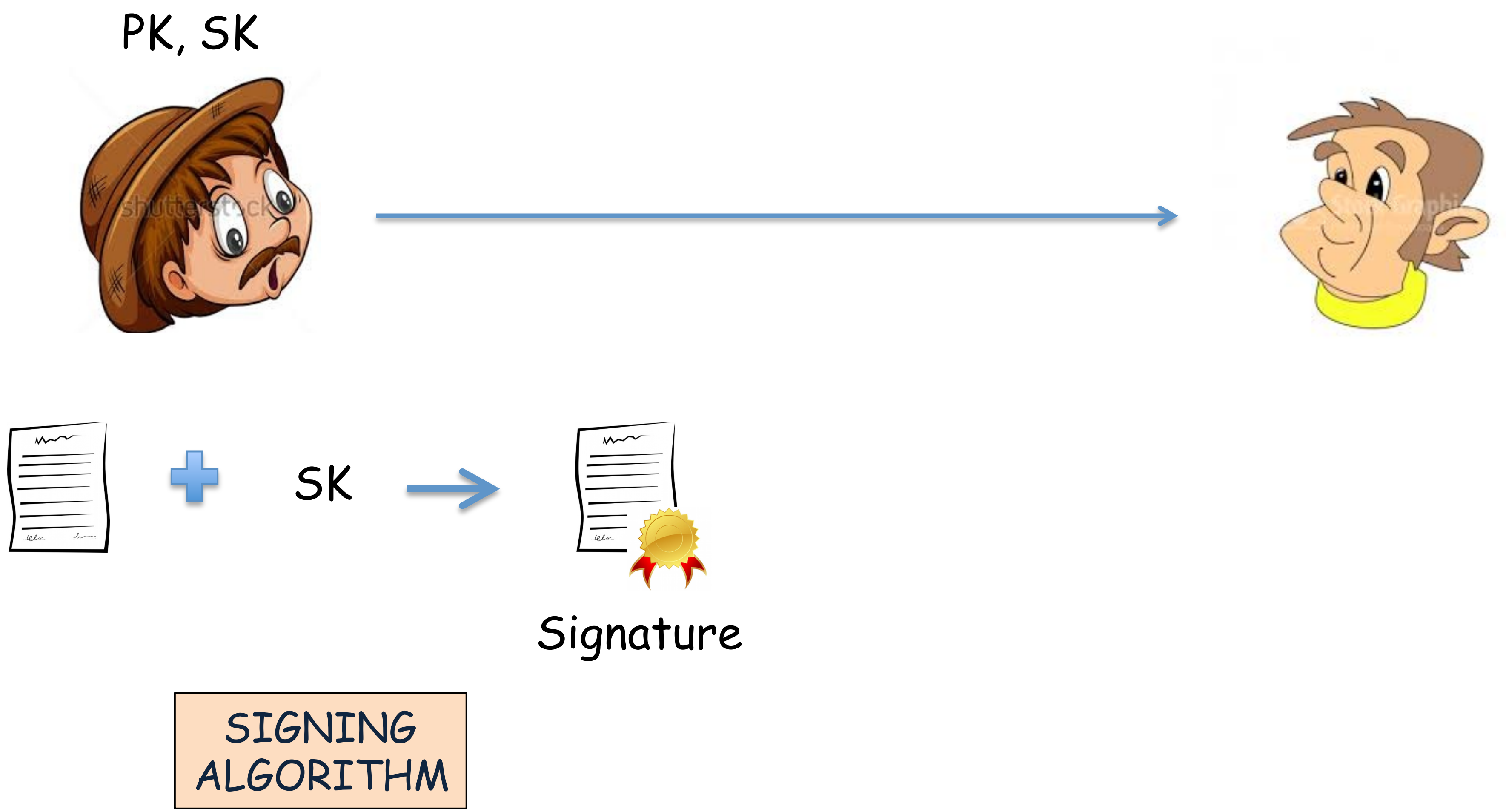
- signature can be easily copied
- it should be a function of the message

Digital Signature Scheme

PK, SK

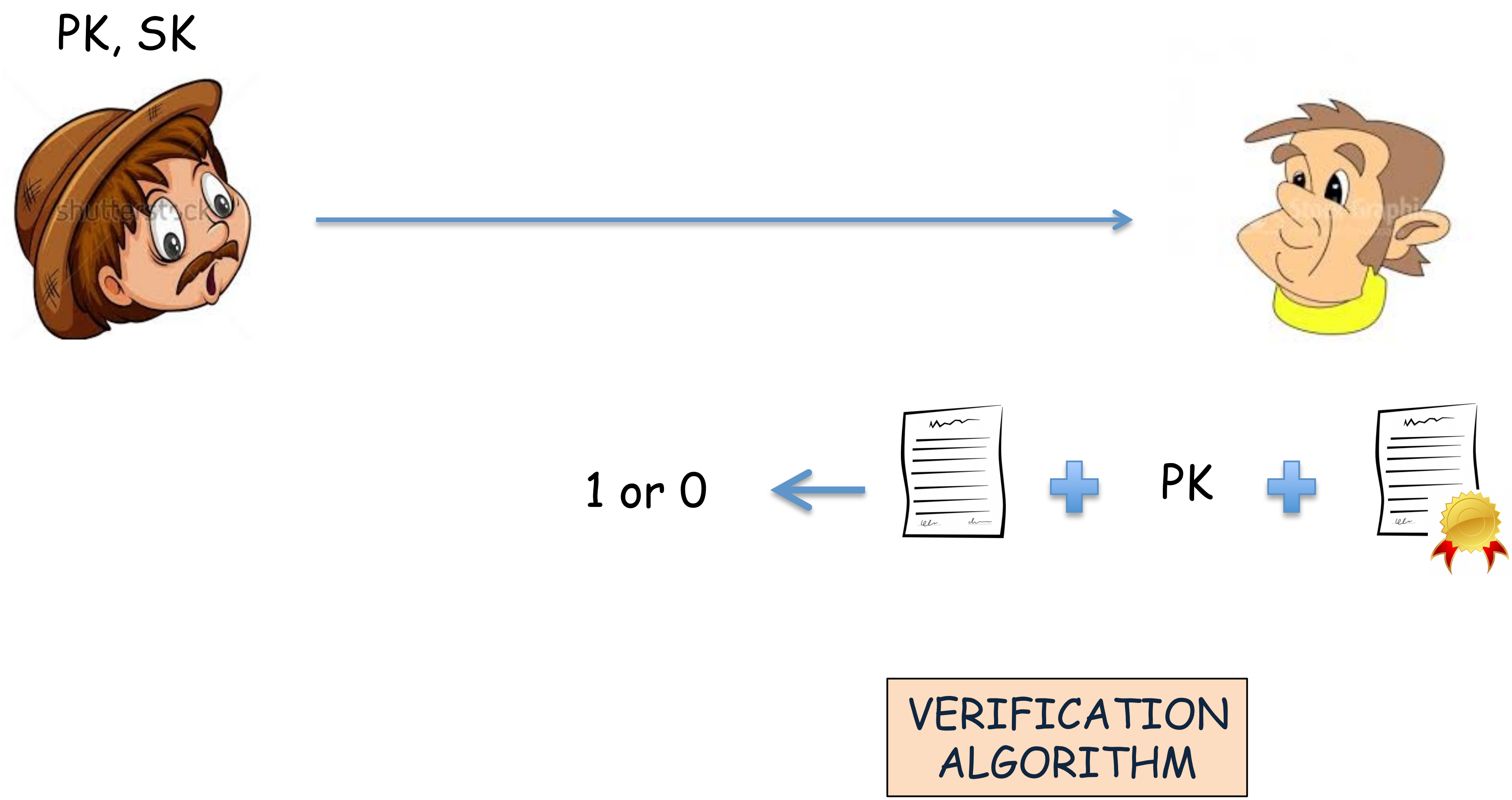


Digital Signature Scheme



Digital Signature Scheme

Digital Signature Scheme



Digital Signature Scheme

A digital signature scheme consists of three algorithms

Gen : outputs a key pair (pk, sk)

Sign :takes a message m in M and the signing key sk as inputs and outputs a signature σ on m

Verify : takes a signature σ, the public key pk, and a message m as inputs and outputs 1 or 0

Correctness

For all (pk, sk) output by Gen and for all m in M

$$\text{Verify}(pk, m, \text{Sign}(sk, m)) = 1$$

Digital Signature Scheme

A digital signature scheme consists of three algorithms

Gen : outputs a key pair (pk, sk)

Sign :takes a message m in M and the signing key sk as inputs and outputs a signature σ on m

Verify : takes a signature σ, the public key pk, and a message m as inputs and outputs 1 or 0

- Integrity
- Authenticity
- Non-repudiation

Correctness

For all (pk, sk) output by Gen and for a

$$\text{Verify}(pk, m, \text{Sign}(sk, m)) = 1$$

RSA Signature

KeyGen

- pick two large primes p and q
- compute N = p.q
- choose an exponent e such that gcd(e,phi(N)) = 1
- choose an exponent d such that e.d = 1 mod phi(N)

RSA Signature

PK=(N,e)

SK=(N, d)

KeyGen

- pick two large primes p and q
- compute N = p.q
- choose an exponent e such that gcd(e,phi(N)) = 1
- choose an exponent d such that e.d = 1 mod phi(N)
- keep (N, d) as secret key, and publish (N, e) as public key

RSA Signature

PK=(N,e)

SK=(N, d)

Signing

$\sigma = m^d \pmod{N}$ where m in $(Z_N)^*$

RSA Signature

PK=(N,e)

$$\xrightarrow{m,\ \sigma}$$

SK=(N, d)

RSA Signature

PK=(N,e)

SK=(N, d)

Verification

if $m = \sigma^e \pmod N$, then output 1;

otherwise, output 0

Attack on RSA Signature

no-message attack

Charlie
(Challenger)

Adversary

Attack on RSA Signature

no-message attack

(N,e)

Charlie
(Challenger)

Adversary

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

Attack on RSA Signature

no-message attack

$\xrightarrow{(N,e)}$

$\xleftarrow{m \text{ in } Q}$

Charlie
(Challenger)

Adversary

(pk,sk) 🡐 Gen(.)
where
pk = (N, e) and sk = (N, d)

Attack on RSA Signature

no-message attack

(N,e) →

m in Q ←

(m, σ) →

Charlie (Challenger)

Adversary

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

Attack on RSA Signature

no-message attack

$(N,e) \longrightarrow$

m in Q $\longleftarrow$

$(m, \sigma) \longrightarrow$

Charlie (Challenger)

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

Adversary

choose σ' in Z_N^*

compute $m' = (\sigma')^e \bmod N$

Attack on RSA Signature

no-message attack

(N,e)

m in Q

Charlie
(Challenger)

Adversary

(m, σ)

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

choose σ′ in Z_N*

compute m′ = $(\sigma')^e$ mod N

(m′, σ′)

Attack on RSA Signature

no-message attack

(N,e) →

Charlie
(Challenger)

← m in Q

(m, σ) →

Adversary

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

choose σ' in Z_N^*

compute $m' = (\sigma')^e \bmod N$

← (m′, σ′)

since $m' = (\sigma')^e \bmod N$, adversary can produce
a valid signature for a message

Attack on RSA Signature

no-message attack

(N,e)

m in Q

Charlie
(Challenger)

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

(m, σ)

Adversary

choose σ' in Z_N^*

compute $m' = (\sigma')^e \bmod N$

(m', σ')

si

the adversary has no control over the message

e game

Attack on RSA Signature

forging a signature on an arbitrary message

Charlie
(Challenger)

Adversary

Attack on RSA Signature

<u>forging a signature on an arbitrary message</u>

$\xrightarrow{(N,e)}$

Charlie
(Challenger)

Adversary

(pk,sk) 🡐 Gen(.)
where
pk = (N, e) and sk = (N, d)

Attack on RSA Signature

forging a signature on an arbitrary message

(N,e) →

← m_1, m_2

Charlie
(Challenger)

Adversary

$(pk,sk) \leftarrow Gen(.)$
where
$pk = (N, e)$ and $sk = (N, d)$

for a message m
choose m_1, m_2 in Z_N^*
s.t. $m = m_1.m_2 \bmod N$

Attack on RSA Signature

forging a signature on an arbitrary message

(N,e)

Charlie
(Challenger)

m_1, m_2

(m_1, σ_1) and (m_2, σ_2)

Adversary

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

for a message m
choose m_1, m_2 in Z_N^*
s.t. $m = m_1.m_2 \bmod N$

Attack on RSA Signature

forging a signature on an arbitrary message

(N,e) →

Charlie
(Challenger)

m_1, m_2 ←

(m_1, σ_1) and (m_2, σ_2) →

Adversary

(pk,sk) ← Gen(.)
where
pk = (N, e) and sk = (N, d)

for a message m
choose m_1, m_2 in Z_N^*
s.t. $m = m_1.m_2 \bmod N$

(m, σ) ←

compute $\sigma = \sigma_1 . \sigma_2 \bmod N$

Attack on RSA Signature

forging a signature on an arbitrary message

Charlie (Challenger)

$(pk, sk) \leftarrow Gen(.)$
where
$pk = (N, e)$ and $sk = (N, d)$

$\xrightarrow{(N,e)}$

$\xleftarrow{m_1,\ m_2}$

$\xrightarrow{(m_1,\ \sigma_1) \text{ and } (m_2,\ \sigma_2)}$

$\xleftarrow{(m,\ \sigma)}$

Adversary

for a message m
choose m_1, m_2 in Z_N^*
s.t. $m = m_1.m_2 \bmod N$

compute $\sigma = \sigma_1.\ \sigma_2 \bmod N$

since $\sigma^e = (\sigma_1.\ \sigma_2)^e = (m_1^d.\ m_2^d)^e$
$= m_1.m_2 \bmod N = m$
adversary wins the game

RSA-FDH

PK=(N, H, e)

SK=(N, H, d)

KeyGen

- pick two large primes p and q
- compute $N = p.q$
- choose an exponent e such that $gcd(e,phi(N)) = 1$
- choose an exponent d such that $e.d = 1 \bmod phi(N)$
- choose a function $H : \{0,1\}^* \rightarrow Z_N^*$
- keep (N, H, d) as secret key, and publish (N, H, e) as public key

RSA-FDH

PK=(N, H, e)

SK=(N, H, d)

Signing

$\sigma = H(m)^d \pmod N$ where m in $\{0,1\}^*$

RSA-FDH

PK=(N, H, e)

σ

SK=(N, H, d)

RSA-FDH

PK=(N, H, e)

SK=(N, H, d)

Verification

if $H(m) = \sigma^e \pmod N$, then output 1;

otherwise, output 0

RSA-FDH

PK=(N, H, e)

- to prevent no-message attack, it should be infeasible for the adversary to invert H

  ---- find m from H(m) ----

SK=(N, H, d)

Verification

if $H(m) = \sigma^e \pmod{N}$, then output 1;

otherwise, output 0

RSA-FDH

PK=(N, H, e)

- to prevent no-message attack, it should be infeasible for the adversary to invert H
  ---- find m from H(m) ----
- to prevent the second attack, it should be hard to find three message m, m_1, m_2 such that $H(m) = H(m_1).H(m_2) \bmod N$

SK=(N, H, d)

Verification

if $H(m) = \sigma^e \pmod N$, then output 1;

otherwise, output 0

RSA-FDH

PK=(N, H, e)

SK=(N, H, d)

- to prevent no-message attack, it should be infeasible for the adversary to invert H
  ---- find m from H(m) ----
- to prevent the second attack, it should be hard to find three message m, m_1, m_2 such that $H(m) = H(m_1).H(m_2) \bmod N$
- also, it should be hard to find collusion:
  ---- find m_1, m_2 s.t. $H(m_1) = H(m_2)$ ----

if $H(m) = \sigma^e \pmod N$, then output 1;

otherwise, output 0