



*CS1699: Blockchain Technology and Cryptocurrency*

---

## 5. Digital Signatures And Centralized Ledgers

Bill Laboon

---

---

# Merkle-Damgård Transforms

---

- ❖ Recall that hash functions should compress an arbitrary-length string into a fixed-size output
- ❖ Also recall our initial attempt at this, BadHash
  - ❖ Convert each character into a corresponding value, sum them up modulo size of output
  - ❖ Variety of problems with this scheme

---

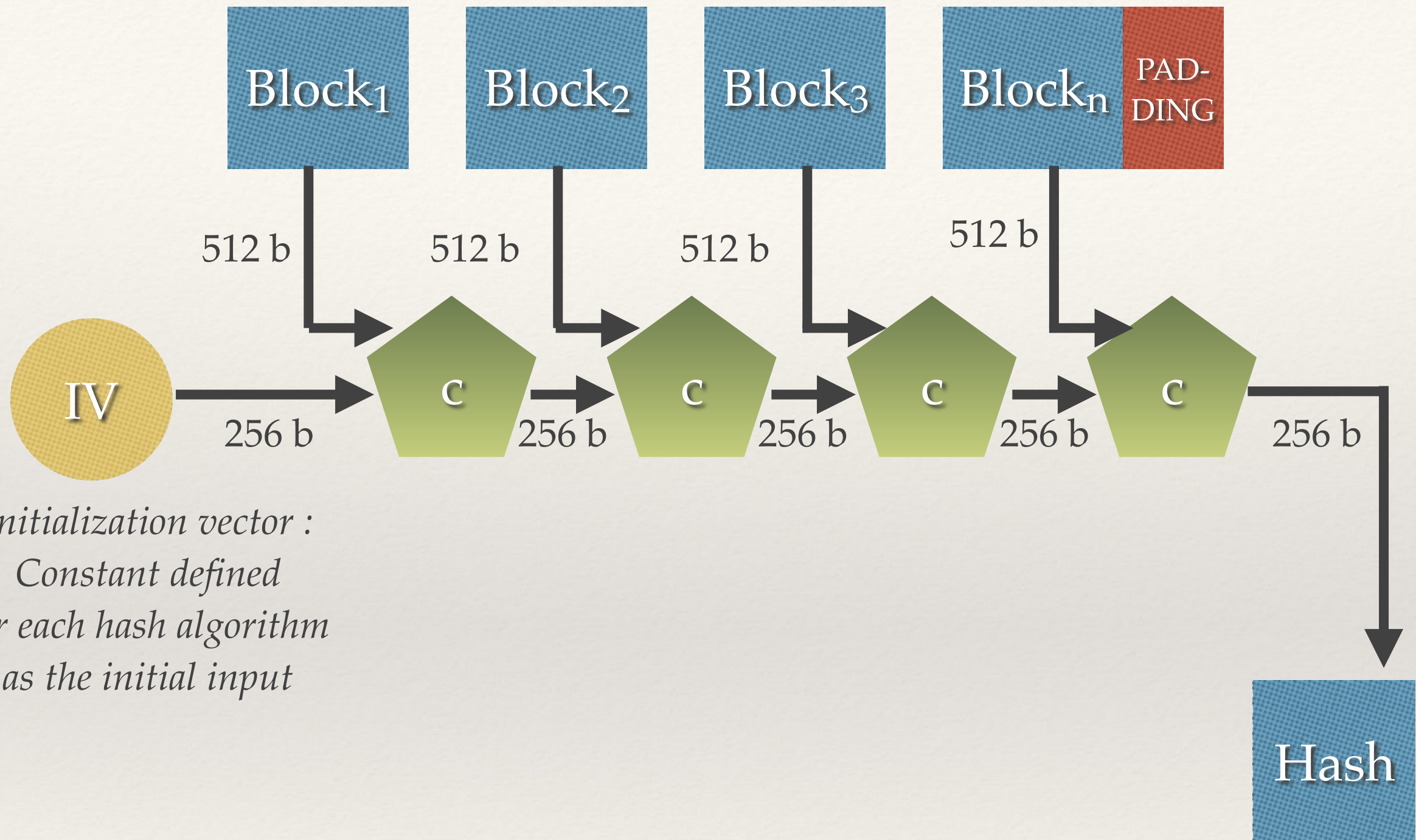
# Merkle-Damgård Transforms

---

- ❖ Merkle-Damgård transforms solve some of the problems converting arbitrary “input to fixed output” using a very similar process to a blockchain!
- ❖ “Chunks” data into blocks (padding if necessary)
- ❖ Accepts results of previous blocks along with current block to produce a new output
  - ❖ compression algorithm accepts two arguments: current block (size  $m$ ) and previous result (size  $n$ )
  - ❖ Outputs result of size  $n$  (where  $n < m$ )
- ❖ Can repeat as many times as needed



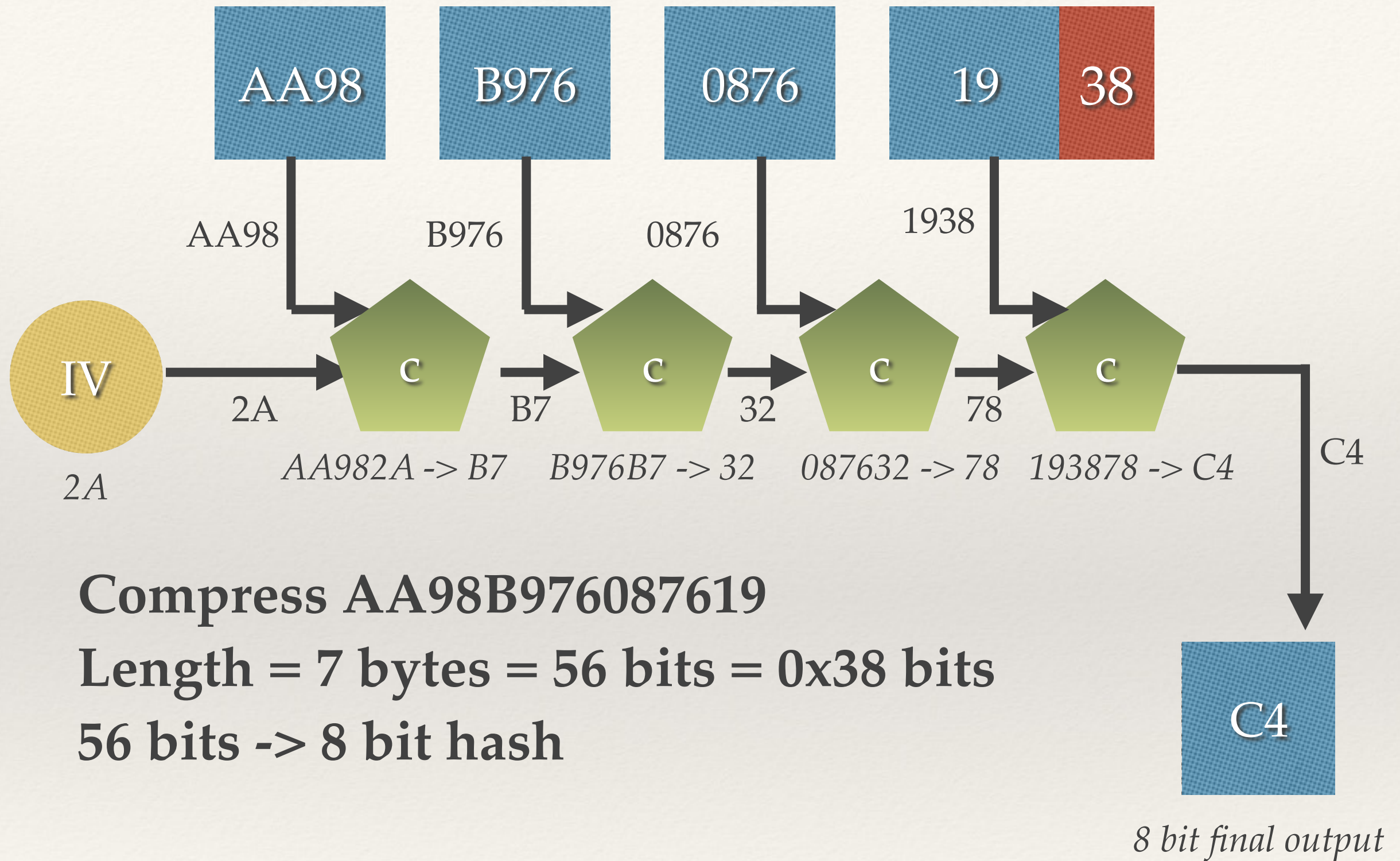
*Block size must remain fixed!*



*Initialization vector :  
Constant defined  
for each hash algorithm  
as the initial input*

*256 bit final output*

# Example w/ $m = 16\text{ b}$ , $n = 8\text{ b}$





---

# Digital Signatures

---

- ❖ We have already discussed how cryptographic hashes can be used to hide information
- ❖ We can also use them to prove our identity using digital signatures

---

# Characteristics of Digital Signatures

---

- ❖ Only I can make a signature, but anyone can verify its validity
- ❖ The signature cannot be re-used - it is tied to a specific document (= set of bits)
- ❖ Signature creation does NOT have to be deterministic!
- ❖ Note this is more powerful than a hand-written signature which can easily be forged, cannot easily be validated, and can be re-used!

---

# Digital Signature Scheme = Three Algorithms

---

**(sk, pk) = generateKeys(keysize)**

*Given a key size keysize, return a “keypair” - a public key used for verification and a secret key for signing*

**sig = sign(sk, message)**

*Given a secret key sk and a message, return a signature for that message*

**isValid = verify(pk, message, sig)**

*Given a public key pk, a message, and a signature, return a Boolean value indicating whether or not the message was properly signed*



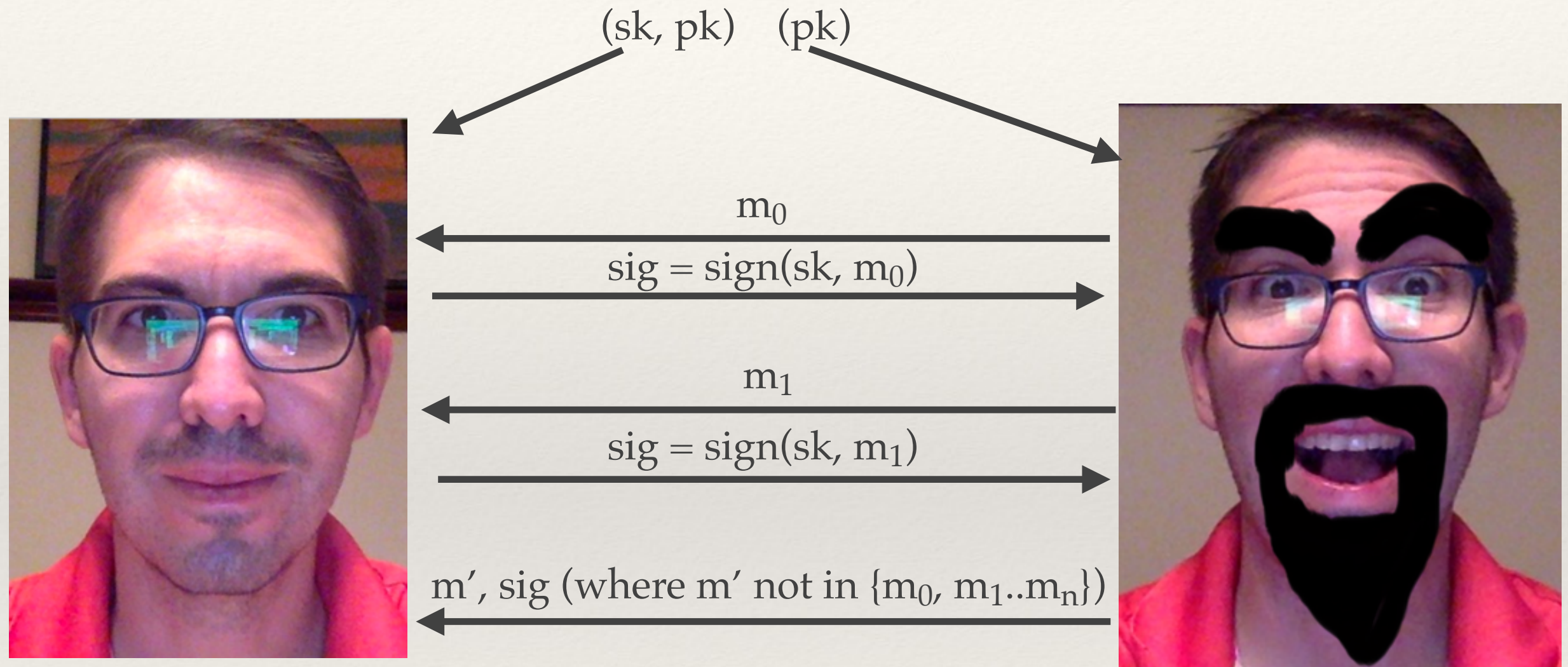
---

# Unforgeability Game

---



# Unforgeability Game



$\text{verify}(pk, m', sig)$

TRUE = Attacker wins

FALSE = Attacker loses



---

# Key = Identity

---

- ❖ Now that I can always prove that I am me, I have a digital identity
- ❖ If I want a new identity, I can always make a new keypair - I will be just another face in the crowd
  - ❖ But be careful! Could use other ways to link your real identity to a generated keypair
- ❖ If someone wants to impersonate me, it should be computationally infeasible

---

# GoofyCoin

---

- ❖ The first rule of GoofyCoin is that only one entity, Goofy, can create a new coin, and all new coins belong to Goofy
- ❖ The second rule of GoofyCoin is whoever owns a coin can transfer the coin to somebody else
- ❖ Both of these rules can be implemented using cryptographic operations



---

# Generating Money With GoofyCoin

---

- ❖ Goofy generates a unique coin ID  $u$
- ❖ Goofy computes a digital signature  $s$  with his secret key
- ❖  $u||s$  ( $u$  concatenated with  $s$ ) is a coin
- ❖ Anyone can run  $verify(pk, u, s)$  to determine if a coin is valid (i.e., has been signed by Goofy)

---

# Transferring Coins with GoofyCoin

---

- ❖ Goofy creates a new transaction “Give *this* to *Alice*” where *this* is a hash pointer to a coin and *Alice* is Alice’s public key
- ❖ Then Goofy signs that transaction with his private key
- ❖ We can now verify that Goofy generated the coin and that it was transferred to Alice



---

# Transferring Coins with GoofyCoin

---

- ❖ Since Alice's public key is specified in the transaction, if Alice tries to send the coin to someone else, she will need the corresponding secret key
- ❖ Anyone can verify that the secret key she used to sign off sending the coin to someone else matches the public key used to give the coin to her!

---

# The GoofyCoin Ecosystem

---

- ❖ Goofy can generate an infinite amount of new coins (as long as he can think up new unique strings)
- ❖ Whoever owns a coin can transfer it by saying “Give this to the person with public key X”
- ❖ Anybody can verify by following the “chain of ownership” back to Goofy (the originator of all coins)





**Goofy**  
(knows  $pk_{\text{Goofy}}$ ,  $sk_{\text{Goofy}}$ )

"I am creating coin A763BA.  
**`sig = sign(skGoofy, msg)`**  
Signature is 98"



**Alice**  
(knows  $pk_{\text{Goofy}}$ )

"Coin A763BA, signed 98..  
**`isValid = verify(pkGoofy, msg, sig)`**  
I can verify Goofy made this coin"





**Goofy**  
(knows  $pk_{\text{Goofy}}$ ,  $sk_{\text{Goofy}}$ ,  $pk_{\text{Alice}}$ )

"I am giving coin A763BA to  $pk_{\text{Alice}}$ .  
**`sig = sign(skGoofy, msg)`**  
Signature is 3A"  
Note msg includes  $pk_{\text{Alice}}$ .



**Alice**  
(knows  $pk_{\text{Goofy}}$ )

"Coin A763BA to  $pk_{\text{Alice}}$ , signed 3A..  
**`isValid = verify(pkGoofy, msg, sig)`**  
I can verify Goofy gave me this coin"





**Alice**  
**(knows  $sk_{Alice}$ ,  $pk_{Bob}$ )**

"I am giving coin A763BA to  $pk_{Bob}$ .

**$sig = sign(sk_{Alice}, msg)$**

Signature is 7B"

Note msg includes  $pk_{Bob}$ .



**Bob**  
**(knows  $pk_{Alice}$ ,  $pk_{Goofy}$ )**

"Coin A763BA to  $pk_{Alice}$ , signed 3A..

Coin A763BA to  $pk_{Bob}$ , signed 7B..

**$isValid = verify(pk_{Goofy}, msg, sig)$**

**$isValid = verify(pk_{Alice}, msg, sig)$**

I can verify Goofy gave this to Alice  
and Alice is giving it to me"

---

# Problem!

---

- ❖ What if Alice gave that coin to Bob and neither Bob nor Alice have told anyone?
- ❖ Alice can also give the coin to Carol and Bob would be none the wiser!
- ❖ Double-spending attack





## Alice

(knows  $sk_{\text{Alice}}$ ,  $pk_{\text{Carol}}$ )

Note: I already gave coin A763BA to  $pk_{\text{Bob}}$ !  
I am giving coin A763BA to  $pk_{\text{Carol}}$ .

**$\text{sig} = \text{sign}(sk_{\text{Alice}}, \text{msg})$**

Signature is DF"

Note msg includes  $pk_{\text{Carol}}$ .



## Carol

(knows  $pk_{\text{Alice}}$ ,  $pk_{\text{Goofy}}$ )

"Coin A763BA to  $pk_{\text{Alice}}$ , signed 3A..

Coin A763BA to  $pk_{\text{Carol}}$ , signed DF..

**$\text{isValid} = \text{verify}(pk_{\text{Goofy}}, \text{msg}, \text{sig})$**

**$\text{isValid} = \text{verify}(pk_{\text{Carol}}, \text{msg}, \text{sig})$**

I can verify Goofy gave this to Alice  
and Alice is giving it to me"

---

# ScroogeCoin

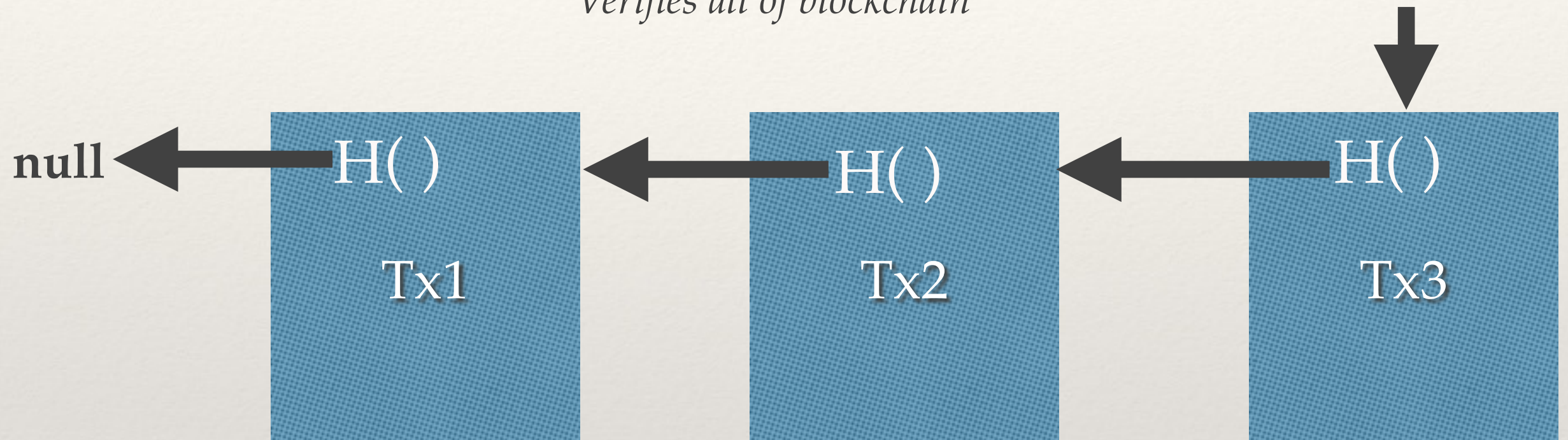
---

- ❖ Transactions occur just as in GoofyCoin BUT..
- ❖ Scrooge also creates an append-only ledger (blockchain) where people can verify that a coin transfer is “official”
- ❖ Scrooge determines that a transaction is valid (i.e. signed correctly and no double-spend) and signs the block
- ❖ Transactions that are not recorded on Scrooge’s blockchain (and signed with Scrooge’s digital signature) are not official



# ScroogeCoin Blockchain

*Root hash signed by Scrooge and published -* **Head:  $H()$**   
*Verifies all of blockchain*



*Note: for simplicity, only showing one transaction per block. As an optimization, multiple transactions can be in a block.*





**Alice**  
**(knows  $sk_{Alice}$ ,  $pk_{Carol}$ )**

Note: I already gave coin A763BA to p  
I am giving coin A763BA to  $pk_{Carol}$ .

**$sig = \text{sign}(sk_{Alice}, msg)$**

Signature is DF"

Note msg includes  $pk_{Carol}$ .



**Carol**  
**(knows  $pk_{Alice}$ ,  $pk_{Scrooge}$ )**

"Coin A763BA to  $pk_{Alice}$ , signed 3A..

Coin A763BA to  $pk_{Carol}$ , signed DF..

**$isValid = \text{verify}(pk_{Scrooge}, msg, sig)$**

**$isValid = \text{verify}(pk_{Carol}, msg, sig)$**

BUT I check Scrooge's blockchain and  
see Alice already gave this coin to Bob!"



---

# ScroogeCoin works... If We Trust Scrooge!

---

- ❖ Scrooge can't steal coins (as he does not know secret keys of individual account holders)
- ❖ But he can:
  - ❖ Blacklist users or coins
  - ❖ Create new coins for himself
  - ❖ Stop updating the blockchain (holding the entire system hostage)

---

# Centralization

---

- ❖ So far, technical challenges have been minimal - earlier cryptocurrencies got to approximately this far, but all relied on some centralized intermediary
- ❖ Central technical challenge and breakthrough of Bitcoin: how do we come to a consensus of valid transactions without a coordinating entity? We need to figure out a way:
  - ❖ For users to agree on a single, published, authoritative blockchain
  - ❖ For users to agree on what transactions are valid and which occurred
  - ❖ IDs to be assigned in a decentralized way
  - ❖ Minting of coins to be done in a decentralized way