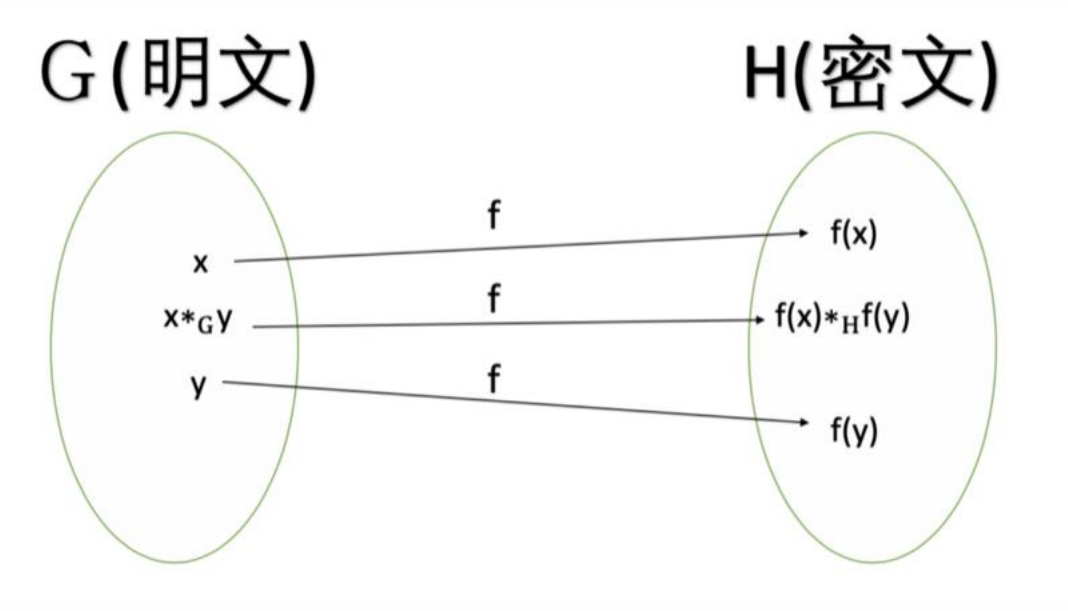
同態加密在門檻式簽章的應用

21/09/2022
CYC

Homomorphic Encryption



- **Homomorphic encryption** is a form of encryption that allows computation on ciphertexts, generating an encrypted result which, when decrypted, matches the result of the operations as if they had been performed on the plaintext.

A simple example (unpadded RSA):

A public key $\mathrm{pk} \coloneqq (n, e)$. For any two messages m_1, and m_2:

$$E(m_1, pk) \cdot E(m_2, pk) = {m_1}^e \cdot {m_2}^e = (m_1 \cdot m_2)^2 \bmod n = E(m_1 \cdot m_2, pk)$$

A security property

A homomorphic encryption cryptosystem should provide semantic security against chosen-plaintext attacks (IND-CPA).

Generates a key pair *PK*, *SK*

Generates a key pair *PK*, *SK*

Generates a key pair *PK*, *SK*

selects a bit $b \in \{0, 1\}$ uniformly at random

$C = E(PK, M_b)$

Generates a key pair *PK*, *SK*

M0, M1 M0, M1

selects a bit $b \in \{0, 1\}$ uniformly at random

$C = E(PK, M_b)$

? Guess b = ?

The history of homomorphic encryption:

- (1978) ON DATA BANKS AND PRIVACY HOMOMORPHISMS (Rivest, Michael, Dertouzos)
- **(1984) Goldwasser–Micali Encryption Scheme (**Goldwasser and Micali **)**
- **(1999)** Public key cryptosystems based on composite degree residue classes (Paillier)
- **(2015) Linearly Homomorphic Encryption from DDH (**Castagno, Laguillaumie**)**
- **(2009)** Fully homomorphic encryption using ideal lattices (Gentry)

Paillier homomorphic encryption：

```go
paillierParameter, _ := NewPaillier(128)
message := big.NewInt(8011313)
ciphertext1, _ := paillierParameter.Encrypt(message)
ciphertext2, _ := paillierParameter.Encrypt(message)
ciphertext3, _ := paillierParameter.Encrypt(message)
ciphertext4, _ := paillierParameter.Encrypt(message)

plaintext1, _ := paillierParameter.Decrypt(ciphertext1)
plaintext2, _ := paillierParameter.Decrypt(ciphertext2)
plaintext3, _ := paillierParameter.Decrypt(ciphertext3)
plaintext4, _ := paillierParameter.Decrypt(ciphertext4)
fmt.Println("ciphertext1:", ciphertext1)
fmt.Println("ciphertext2:", ciphertext2)
fmt.Println("ciphertext3:", ciphertext3)
fmt.Println("ciphertext4:", ciphertext4)

fmt.Println("plaintext1:", plaintext1)
fmt.Println("plaintext2:", plaintext2)
fmt.Println("plaintext3:", plaintext3)
fmt.Println("plaintext4:", plaintext4)
```

```
API server listening at: 127.0.0.1:35649
ciphertext1: 341033116921558987290115030091506652312939759982304066006940067384638323817 65
ciphertext2: 710272507380863541422607449216950420226559969574068350127384184452481268707 77
ciphertext3: 715871074493577535377545929186647256663836590910638147319255253116816557581 39
ciphertext4: 626062385404840658076766287646445925666211472671027374730716478933248355961 13
plaintext1: 8011313
plaintext2: 8011313
plaintext3: 8011313
plaintext4: 8011313
```

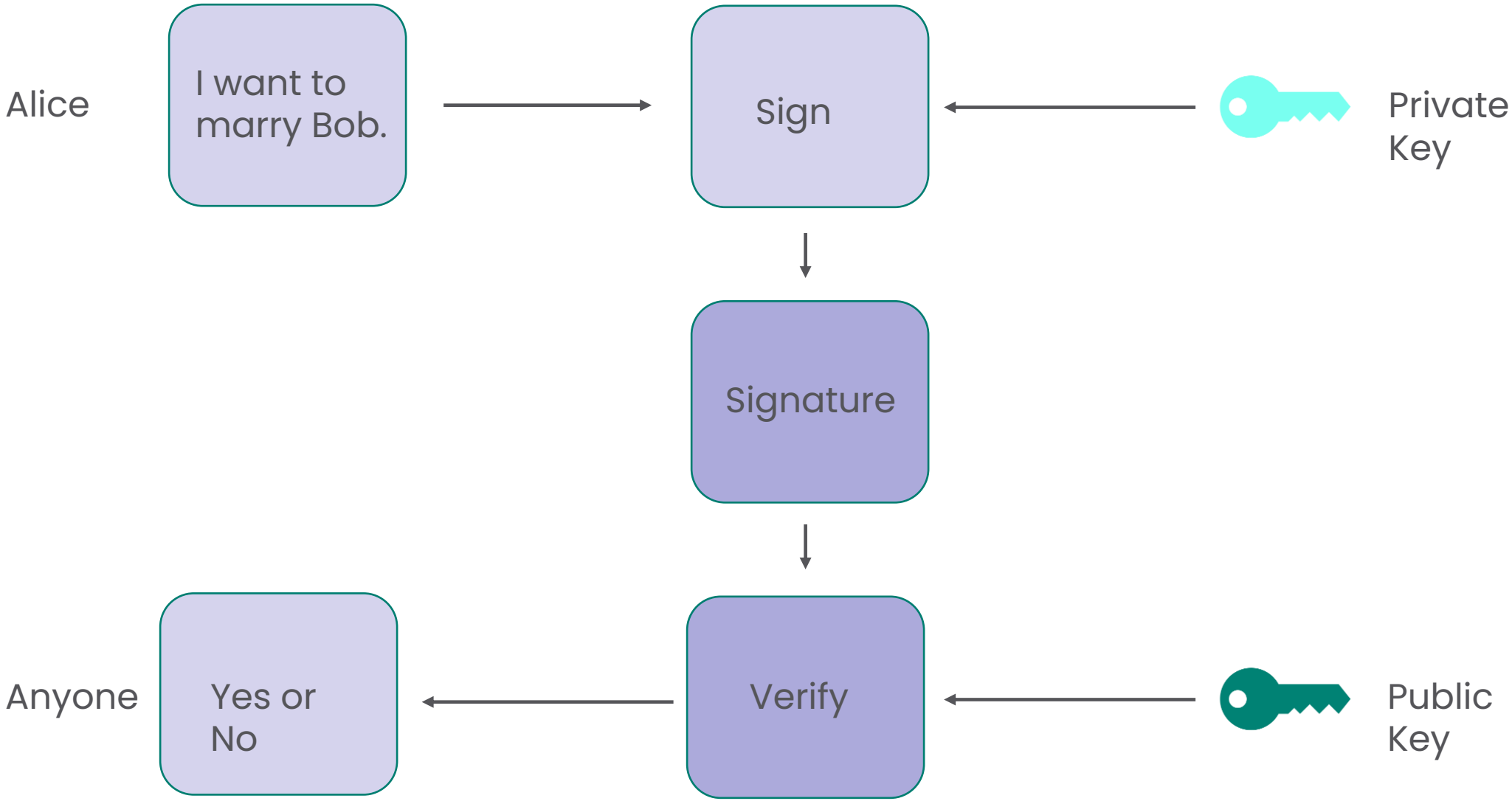
system	RSA	Class group (imaginary quadratic order)	Elliptic curve cryptography
Hard problem	factoring problem	Discrete logarithm problem …..	Discrete logarithm problem ….
history	Very old	Gauss (~1800)	Koblitz, Miller(wide use 2004~2005)
The length of private key	Long	Medium	short
speed of encryption	fast	medium	fast

Castagnos-Laguillaumie homomorphic encryption

```
// Generate a private key and the corresponding public key.
var publicKey, privateKey, _ = PubKeygen(bigPrime, SAFE_PARAMETER)

// Encrypt two plaintexts by the public key
plaintext1 := big.NewInt(3)
cipherMessege1 := Encrypt(publicKey, plaintext1)

plaintext2 := big.NewInt(13)
cipherMessege2 := Encrypt(publicKey, plaintext2)

// Do an operation of cipherMessege1 and cipherMessege2.
AddResult := EvalAdd(cipherMessege1, cipherMessege2, publicKey)

// The result should be 3 + 13 = 16
decyptAddResult := Decrypt(AddResult, privateKey)
fmt.Println("The decryption of adding ciphertext1 and ciphertext2 to be", decyptAddResult)
```

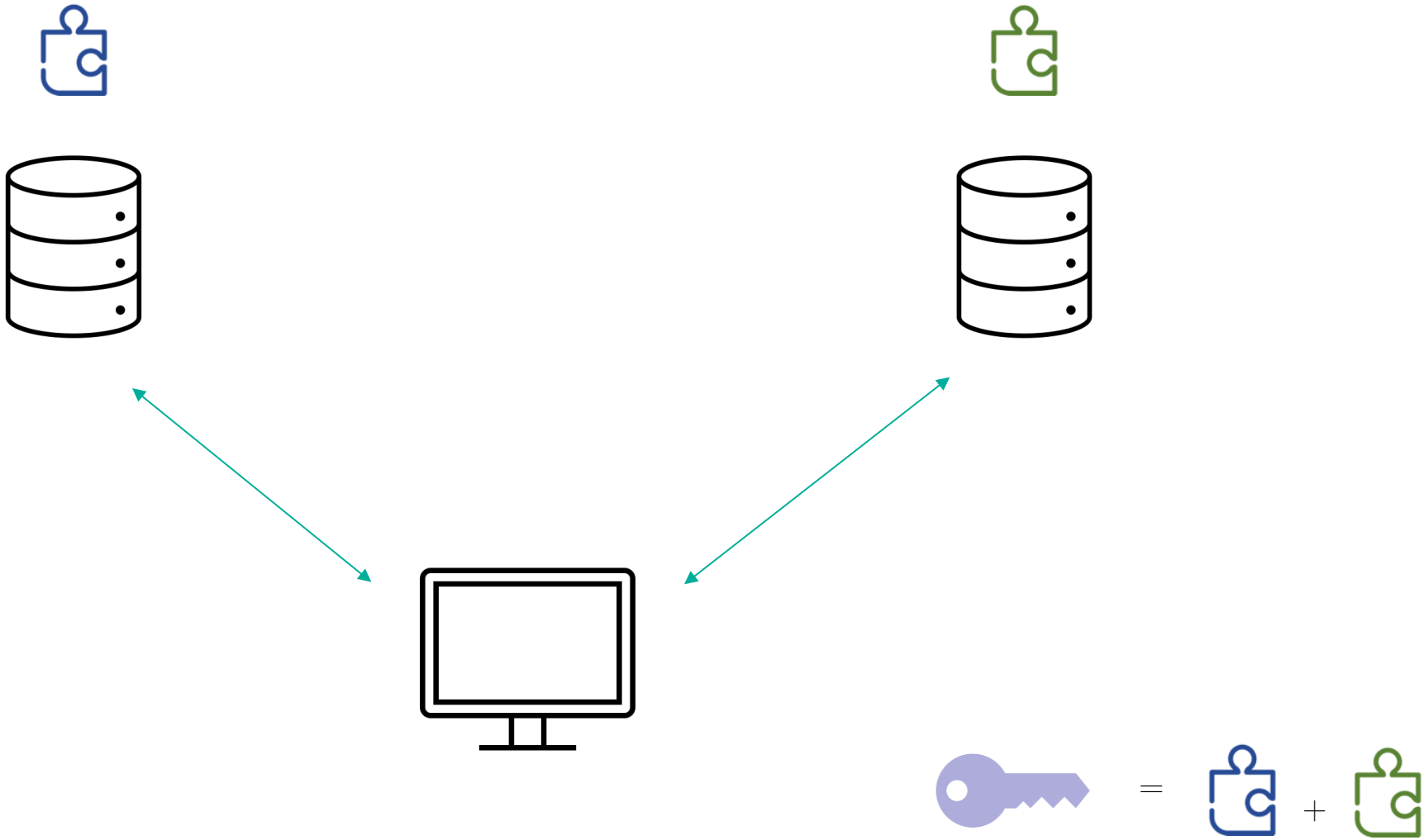
```
API server listening at: 127.0.0.1:22310
The decryption of adding ciphertext1 and ciphertext2 to be 16
```

Digital Signature



Private key management

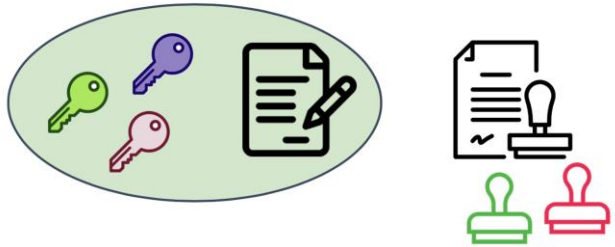
An idea:



Two solutions

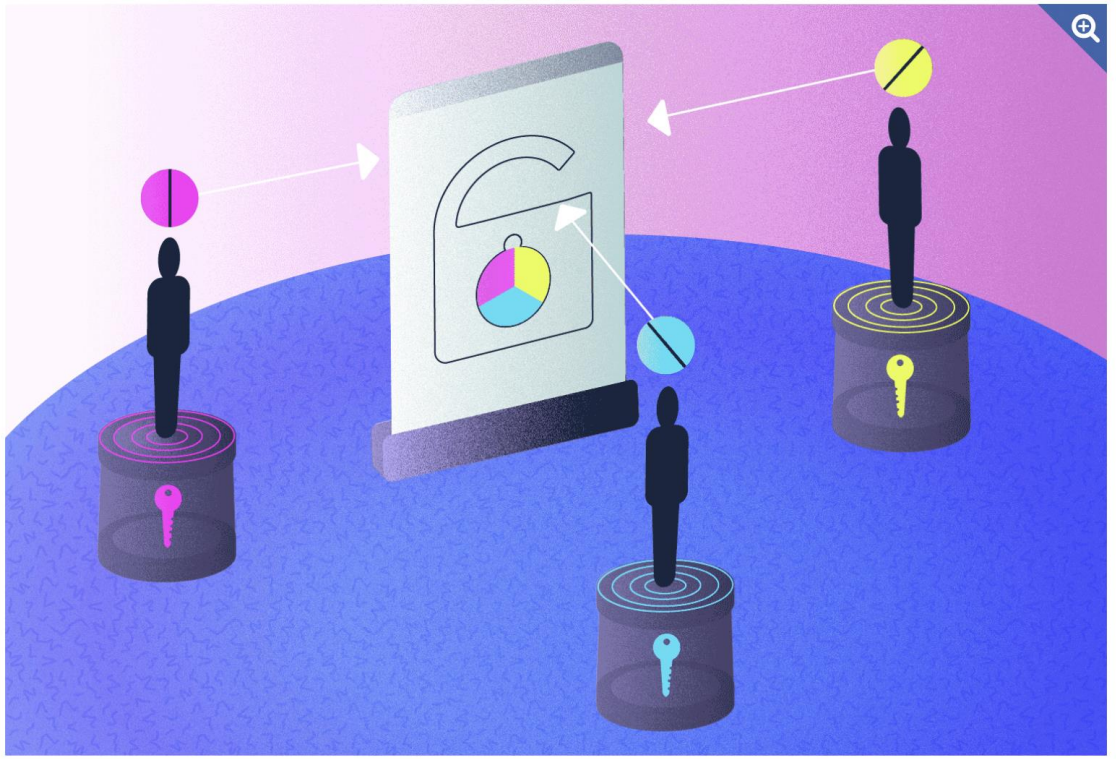
- Multi-Signature
- Threshold Signature scheme

(t,n)-Multi-Signature



- **Multi-signature** is a digital signature scheme which allows a group of users to sign a transaction.
- t : the threshold of signatures or keys
- n : the total number of signatures or keys involved in the group.

National Institute of Standards and Technology: Standardization



This artist's conception of threshold cryptography shows a lock that can only be opened by three people working together. When the threshold cryptosystem receives a request to process information with a secret key, it initially splits the key into shares and sends them to the entire group, each share to a different participant. The three people must agree to work together and also perform their own secret operations on the incoming message. From these actions, each person uses their share key — represented by the three colored circles — to process the message, and then sends the result back to the system. Only the combination of all three partial results can open the lock, reducing the likelihood that a single corrupt party could compromise the system.
Credit: N. Hanacek/NIST

Ref: https://www.nist.gov/news-events/news/2020/07/nist-kick-starts-threshold-cryptography-development-effort

Multi-party computation

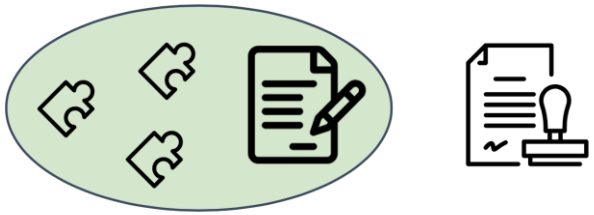
Secure multi-party computation (MPC) can be defined as the problem of n players to compute an agreed function of their inputs in a secure way, where security means guaranteeing the correctness of the output as well as the privacy of the players' inputs, even when some players cheat.

Ref: Multiparty Computation, an Introduction （Cramer et al.）, https://link.springer.com/book/10.1007/3-7643-7394-6

Yao's Millionaires' problem

Who is rich ?

(t,n)-Threshold Signature (an application of homomorphic encryption)



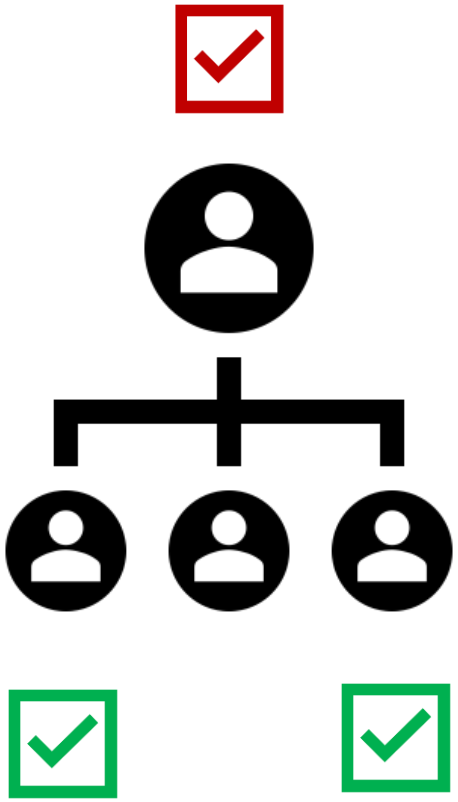
- A (*t*,*n*)-threshold signature scheme is a digital signature scheme where any *t* or more signers of a group of *n* signers can produce signatures on behalf of the group.

Comparation

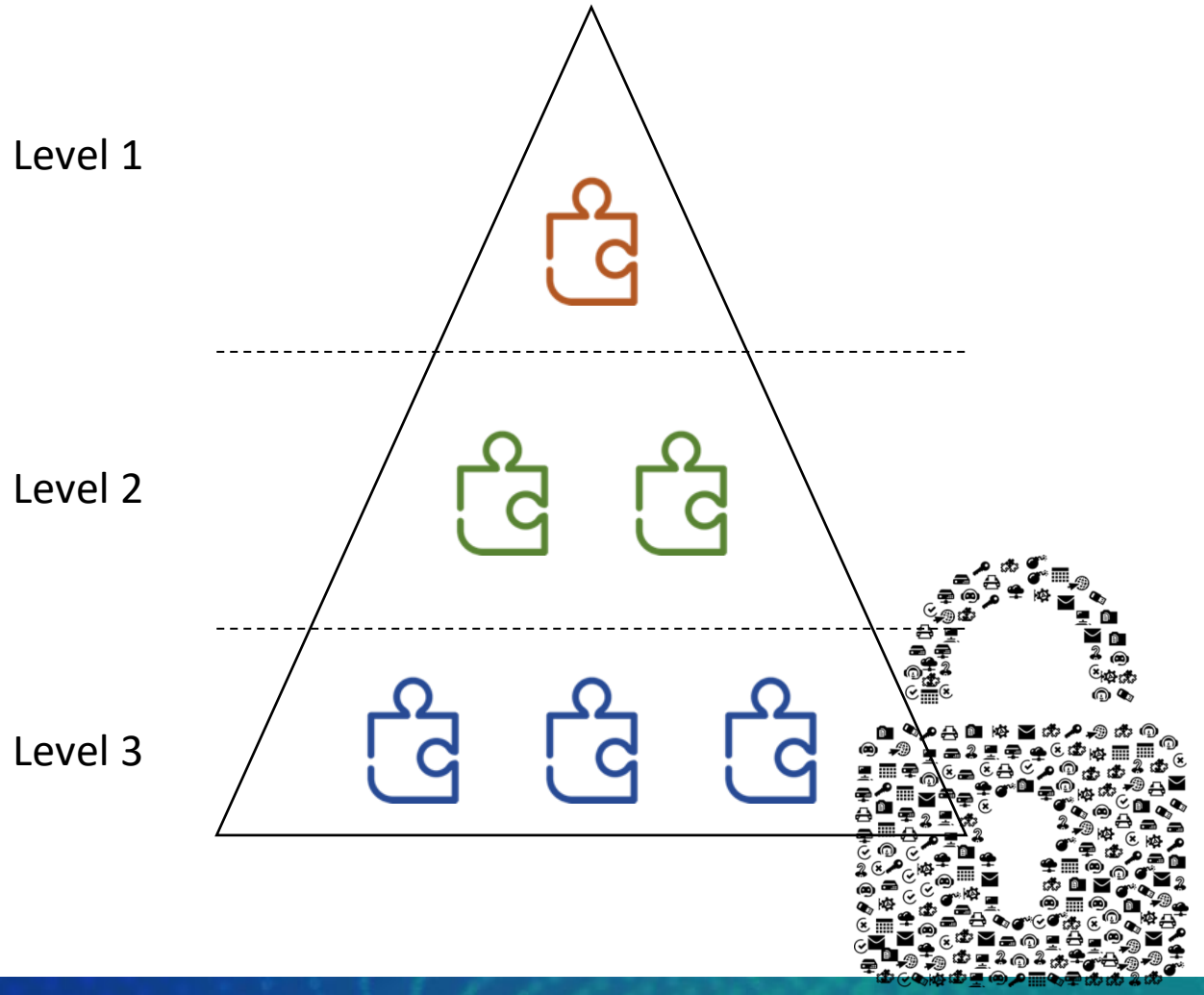
	Threshold Signature	Multi-Signature
verification cost	low	high
maintenance cost	low	high
implementation complexity	high	low
secret management	easy	hard
support of blockchains	Native	no
accountability	no	yes
interactive	synchronous	asynchronous

Distributed key generation, sign, reshare

A scenario



Level of shares:



AMIS' solution: Hierarchical Threshold Signature Scheme

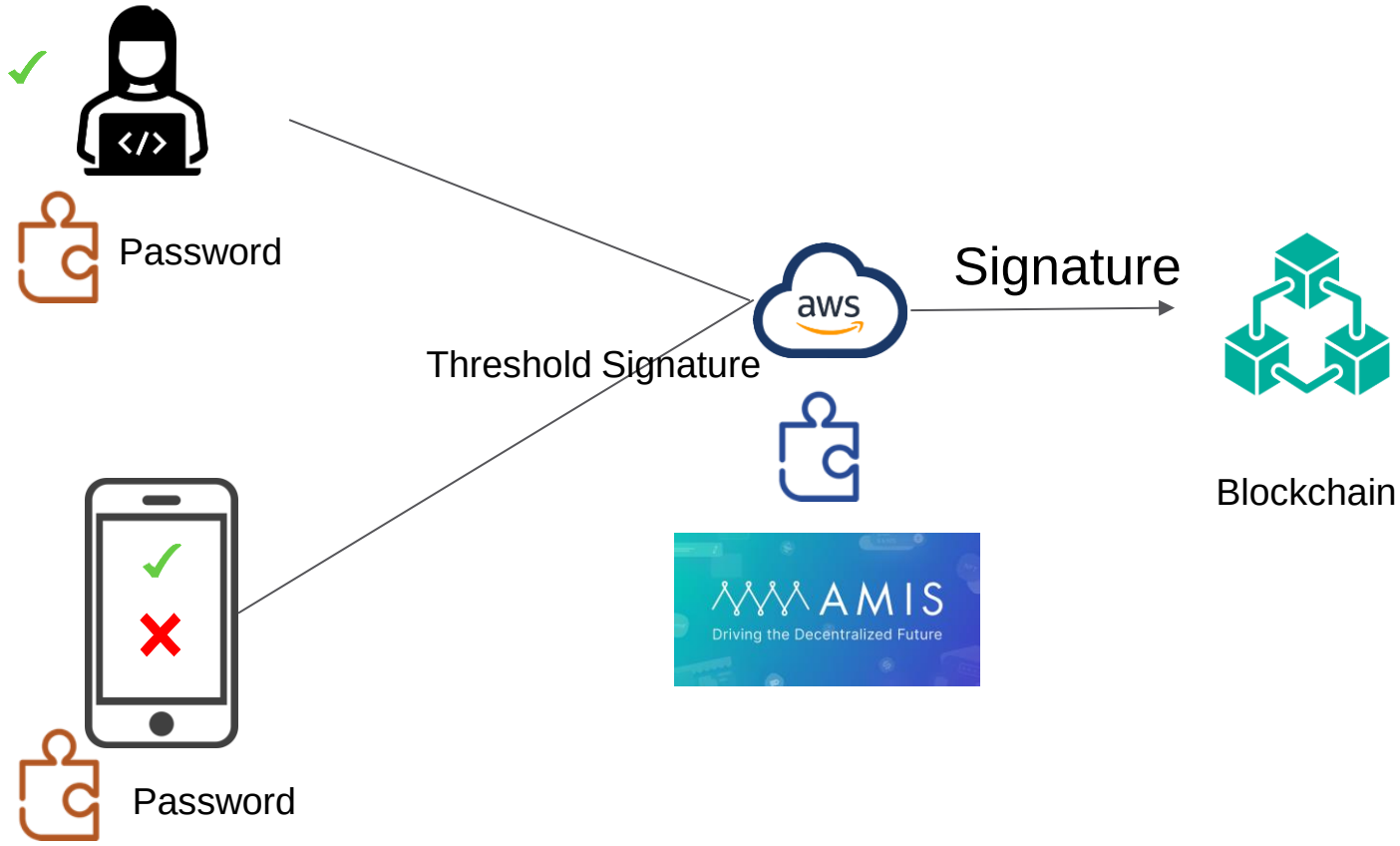
- Support ECDSA, and EdDSA (Audited, In progress)
- Support DKG, Sign, Reshare, and Add share
- Open Library

Ref: https://github.com/getamis/alice

Coinbase **developer grant winners 2022**

Ref: https://blog.coinbase.com/announcing-our-second-developer-grant-winners-ffd3f6e93860

Product: Qubic Wallet



Also support 2 party HD Wallet

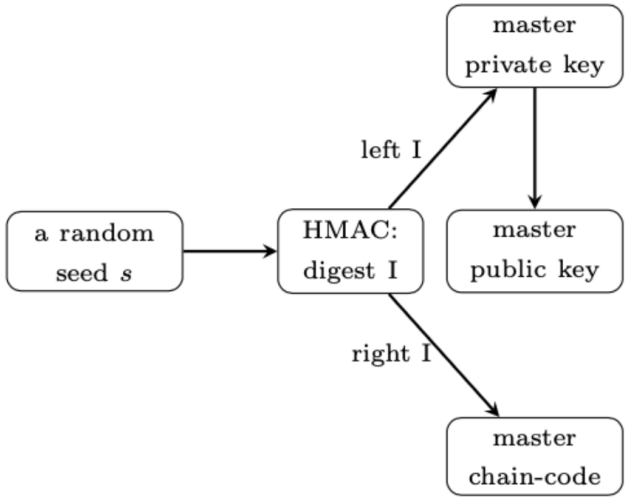


Fig. 1: BIP32: Master Key Generation

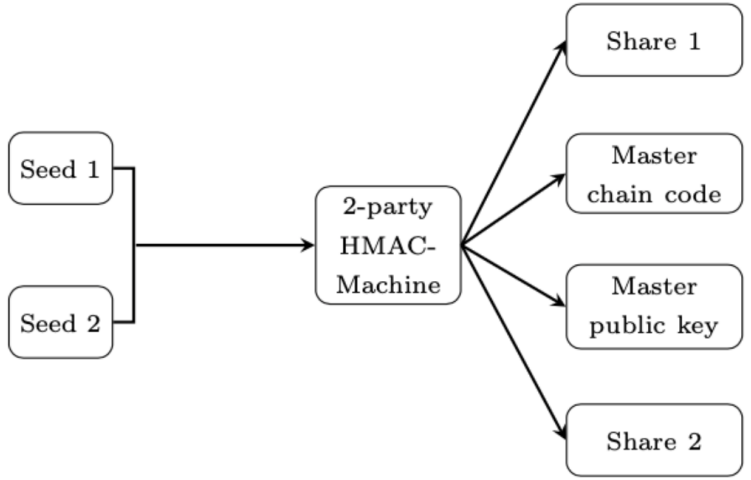


Fig. 2: 2-Party BIP32:Master Key Generation

Paper: Joint work with Yi-Hong Hsu, Ting Fung Lee.

Project: AMIS Team


AMIS