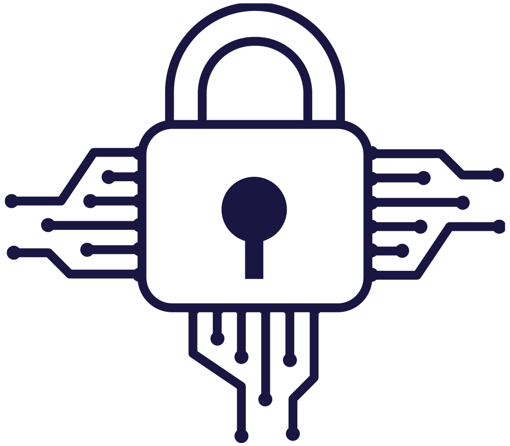
KINGSLAND
SCHOOL OF BLOCKCHAIN

BLOCKCHAIN CRYPTOGRAPHY 2

ECC, ECDSA, EdDSA,Cryptography in Blockchain

1. **Elliptic Curves** and ECDSA: Key Generation, Sign, Verify
2. **Blockchain** Cryptography: Wallets, Keys, Addresses, Signatures
4. **Quantum-Safe** Cryptography: Hashes, Encryption, Signatures

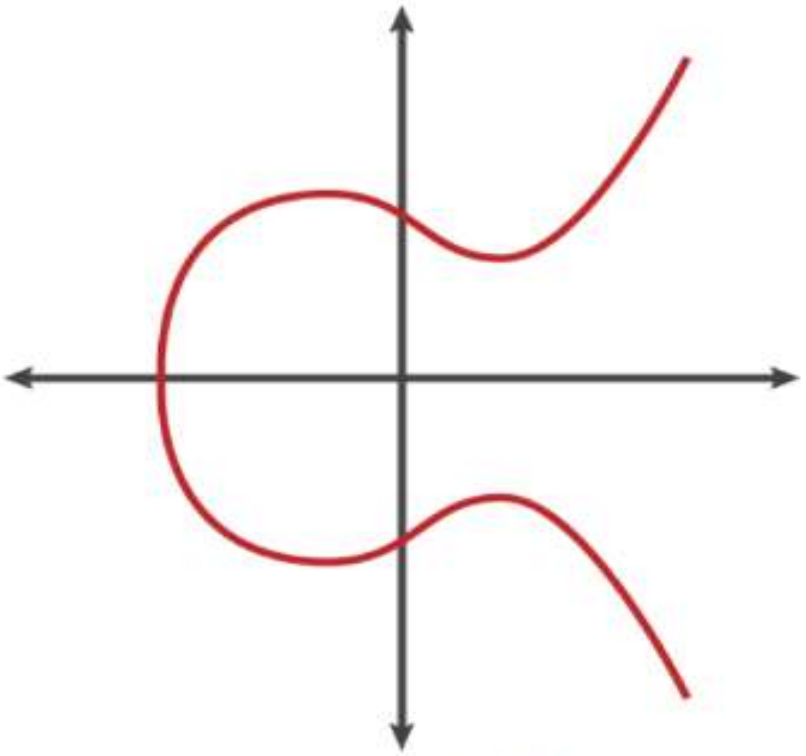


ELLIPTIC CURVE CRYPTOGRAPHY (ECC)

Elliptic Curves, ECC and ECDSA, Sign, Verify

Elliptic Curve Cryptography (ECC)

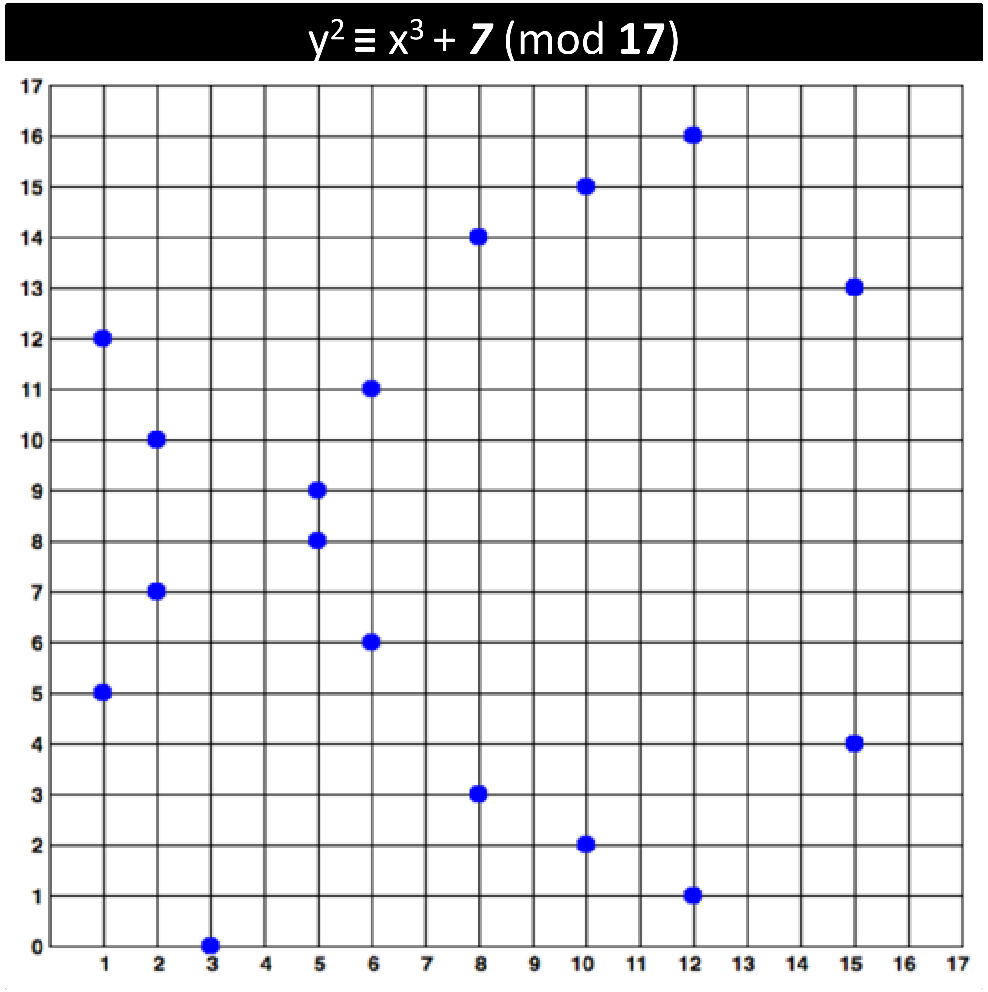
- Public / private key cryptography based on the algebraic structure of **elliptic curves** over finite fields
  - Requires **smaller key-size** than RSA for the same security strength
- **Elliptic curves** == set of points {**x**, **y**} such that:
  - $\mathbf{y^2 = x^3 + \mathit{a}x + \mathit{b}}$
- Example – the Bitcoin elliptic curve:
  - $\mathbf{y^2 = x^3 + 7}$ ($\boldsymbol{a} = 0$; $\boldsymbol{b} = 7$)



Example of an Elliptic Curve

Elliptic Curve over a Finite Field

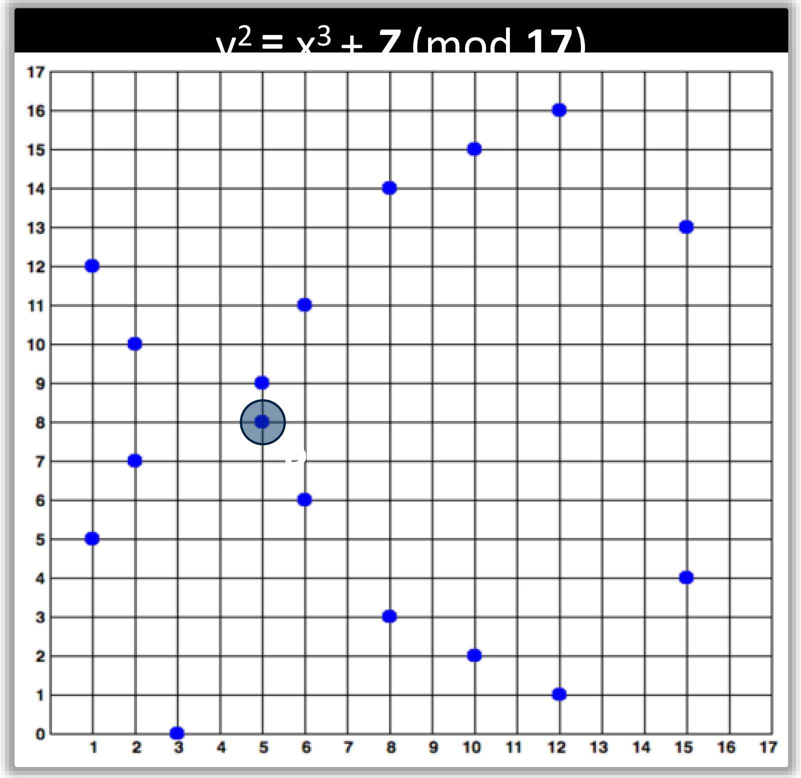
- Elliptic curve cryptography (**ECC**)
  - Uses ecliptic curve over a finite field $\mathbf{F_p}$ ($\mathbf{p}$ is prime, $\mathbf{p} > 3$)
  - A set of integer coordinates $\{\mathbf{x}, \mathbf{y}\}$, such that $\mathbf{0} \leq \mathbf{x}, \mathbf{y} < \mathbf{p}$
  - Staying on the elliptic curve: $y^2 \equiv x^3 + \boldsymbol{a}x + \boldsymbol{b} \pmod{\mathbf{p}}$
- Example of elliptic curve over $\mathbf{F_{17}}$:
  - $y^2 \equiv x^3 + \boldsymbol{7} \pmod{\mathbf{17}}$



Calculating Elliptic Curves over Finite Fields

- The point **P(5, 8)** is on the curve **$y^2 = x^3 + 7$** over the finite field **F_{17}**
  - $(x^3 + 7 - y^2) \equiv \mathbf{0} \pmod{17}$
  - (**5****3 + 7 - **8****2) % 17 == 0

```
Command Prompt - python
>python
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40)
Type "help", "copyright", "credits" or "license" for more information.
>>> (5**3 + 7 - 8**2) % 17 == 0
True
>>>
```



Exercise: Elliptic Curves over Finite Fields

- Problem: True or False?

  Point is on the $y^2 = x^3 + 7$ curve over F_{223}

  1. (192, 105)
  2. (17, 56)
  3. (200, 119)
  4. (1, 193)
  5. (42, 99)

- Solution of $y^2=x^3+7$ curve over F_{223}
  - Just calculate the expression:

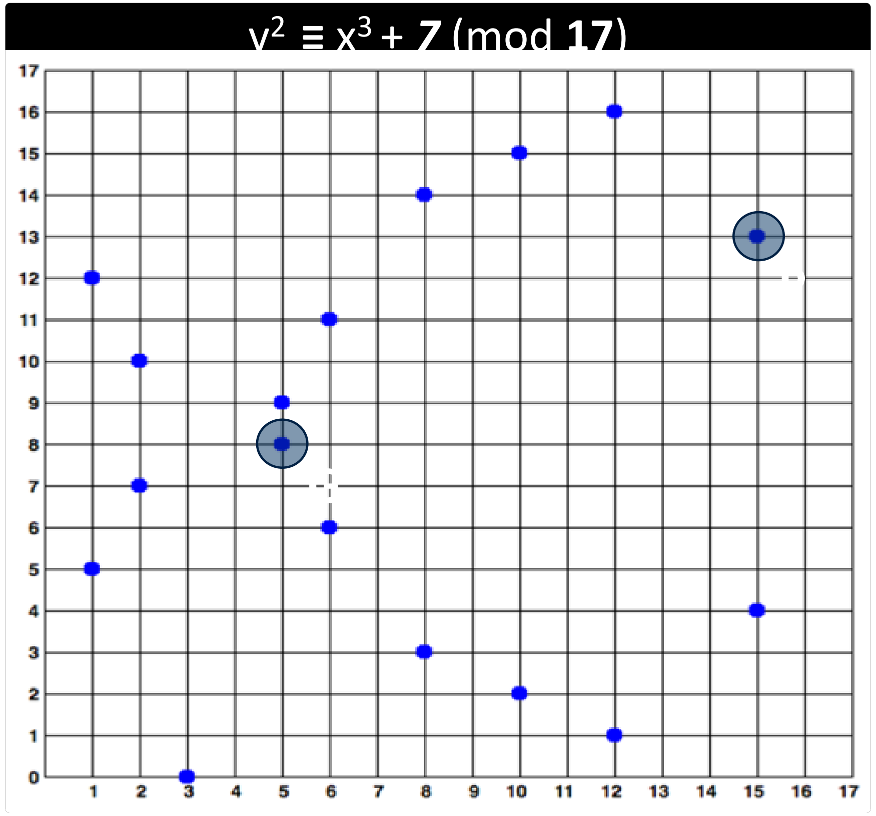
```
(192**3 + 7 - 105**2) % 223 == 0
```

Command Prompt - python

```
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40)
Type "help", "copyright", "credits" or "license" for
>>> (192**3 + 7 - 105**2) % 223 == 0
True
>>>
```

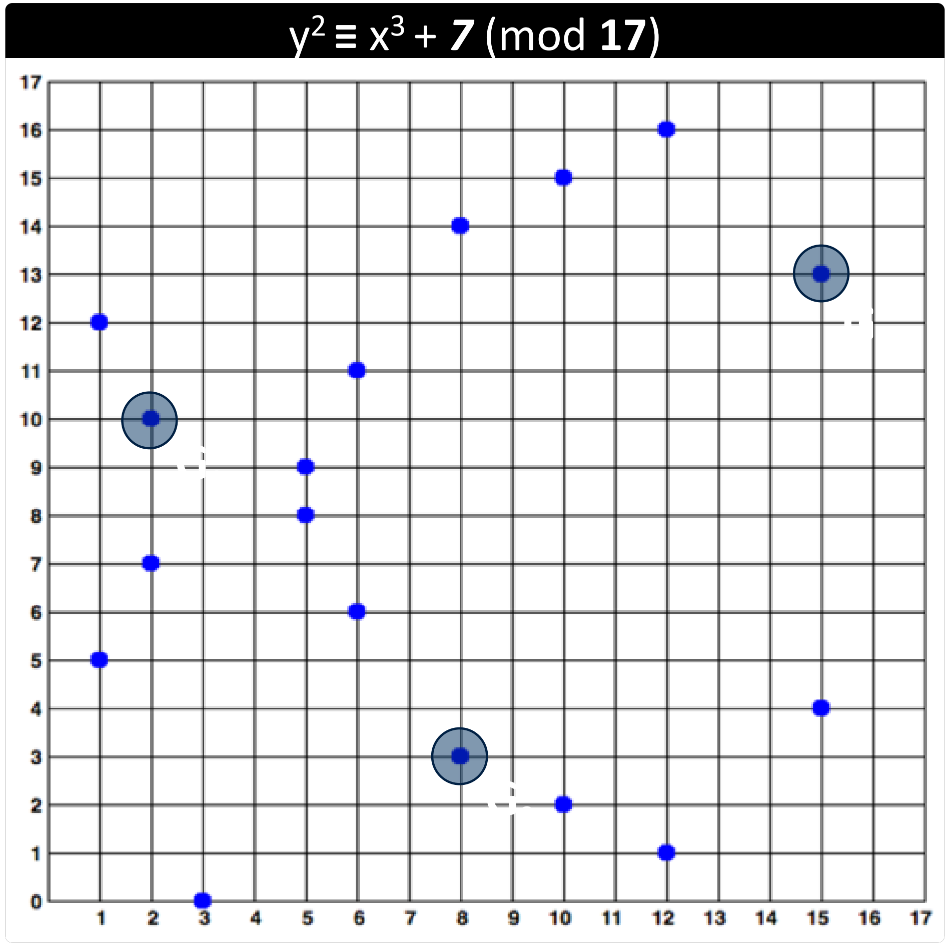
Multiply a Point Over an Elliptic Curve

- A point **G** over the curve can be multiplied by an integer **k**
  - **P** = **k** * **G**
  - The result is another point **P** staying on the same curve
  - **k == private key** (integer)
  - **P == public key** (point)
  - Very **fast** to calculate **P = k * G**
  - Extremely **slow** (considered infeasible) to calculate **k = P / G**



Example: Multiply Points Over Elliptic Curves

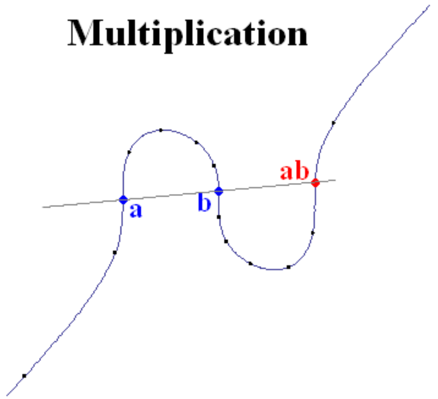
- Elliptic curve point multiplication is done by [well-known algorithms]
- Sample elliptic curve:
  - $y^2 \equiv x^3 + \boldsymbol{7} \pmod{\mathbf{17}}$
  - **a** = 0; **b** = 7; **p** = 17
- Let **G** = (15, 13)
  - $G_2 = 2 * \mathbf{G} = (2, 10)$
  - $G_3 = 3 * \mathbf{G} = (8, 3)$



Elliptic Curves Multiplication in Python

```
pip install pycoin
```

```
from pycoin.ecdsa import Point
from pycoin.ecdsa import Curve

curve = Curve.Curve(17, 0, 7)
print("Curve = " + str(curve))

G = Point.Point(15, 13, curve)
print("G = " + str(G))

for k in range(0, 6) :
  print(str(k) + " * G = " + str(k * G))
```



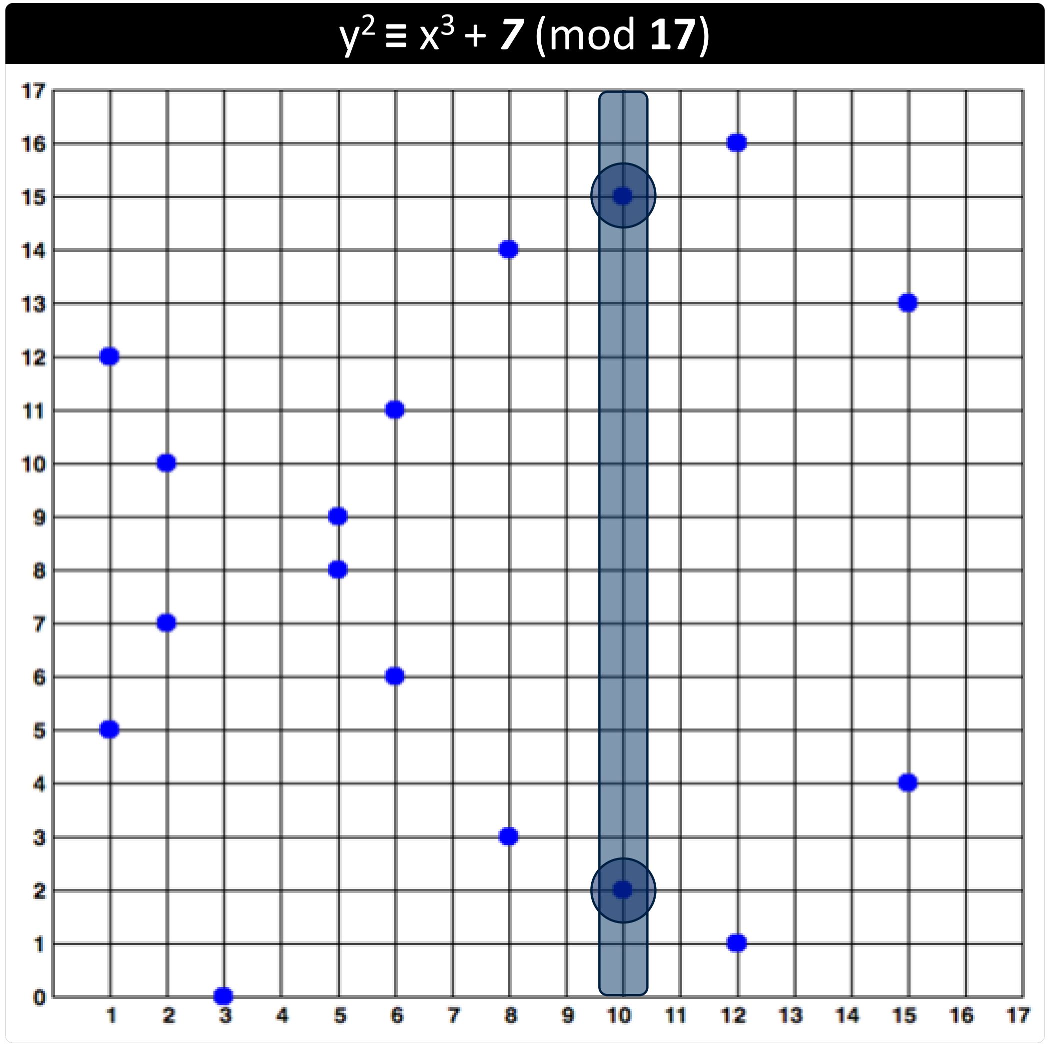
Command Prompt - python

```
>>> for k in range(0, 6) :
...   print(str(k) + " * G = " + str(k * G))
...
0 * G = infinity
1 * G = (15,13)
2 * G = (2,10)
3 * G = (8,3)
4 * G = (12,1)
5 * G = (6,6)
>>>
```

Compressing the Public Key

- The elliptic curves over $\mathbf{F_p}$
  - Have at most 2 points per **x** coordinate (odd **y** and even **y**)
- A public key **P**(**x**, **y**) can be **compressed** as **C**(**x**, **odd/even**)
  - At the curve $y^2 \equiv x^3 + 7 \pmod{17}$

    P(**10**, **15**) == **C**(**10**, **odd**)
  - mod_sqrt($\mathbf{x}^3$ + *7*, ***17***) == **y** || **17** - **y**



Compressing a Public Key in Python

```
pip install nummaster
```

```
from pycoin.ecdsa import Curve, Point
from nummaster.basic import sqrtmod

def compress_key_pair(key_pair):
  return (key_pair[0], key_pair[1] % 2)

def uncompress_key(curve, compressed_key):
  x, is_odd = compressed_key
  p, a, b = curve._p, curve._a, curve._b
  y = sqrtmod(pow(x, 3, p) + a * x + b, p)
  if bool(is_odd) == bool(y & 1):
    return (x, y)
  return (x, p - y)
```

Compressing a Public Key in Python (2)

```python
curve = Curve.Curve(17, 0, 7)

p = Point.Point(10, 15, curve)
print(f"original key = {p}")

compressed_p = compress_key_pair(p)
print(f"compressed = {compressed_p}")

restored_p = uncompress_key(curve, compressed_p)
print(f"uncompressed = {restored_p}")
```

Command Prompt

```
>python compress-key.py
original key = (10,15)
compressed = (10, 1)
uncompressed = (10, 15)

>
```

ECC Parameters and secp256k1

- ECC operates with a set of **EC domain parameters**:
  - **T** = (**p**, **a**, **b**, **G**, **n**, **h**)
  - Prime field (prime **p**), elliptic equation (a, b), base point **G**($\mathbf{x_G}$, $\mathbf{y_G}$), order of **G** (prime **n**), cofactor (**h**)
- The **secp256k1** standard (used in **Bitcoin**) defines 256-bit elliptic-curves cryptosystem:
  - **Prime field** (**p**) = $2^{256} - 2^{32} - 977$; **Equation**: $y^2 = x^3 + 7$ (a = **0**, b = **7**)
  - **G** = 0x79BE667E …; **n** = 0xFFF…D0364141; **h** = 1

Learn more at: http://www.secg.org/sec2-v2.pdf, https://en.bitcoin.it/wiki/Secp256k1

ENSURING THE FUTURE OF BLOCKCHAIN

What is a Digital Signature?

- Digital signature
  - Cryptographic proof for message **authenticity**
  - **Authentication**
    - **Signed by certain private key**
    - **Verified by the corresponding public key**
  - Non-repudiation
    - **The sender cannot deny the signing later**
  - Integrity
    - **The message cannot be altered after signing**

ECDSA: Sign Messages and Verify Signatures

- The Elliptic-Curves Digital Signature Algorithm (**ECDSA**) provides **signing** by private key + **verifying** signature by public key

function(**parameters**) → **output**

- **Signing** a message
  - **sign**(**private_key**, **msg**) → **signature** (the signature is a pair of numbers [r, s])
  - Performs some math, based on elliptic curve calculations
- **Verifying** a message signature
  - **verify**(**public_key**, **msg**, **signature**) → **true** / **false**
  - Performs some math, based on elliptic curve calculations

Example: ECDSA in Python – Sign a Message

```
from pycoin.ecdsa import secp256k1
import hashlib

def sha3_hash(msg) :
  hash_bytes = hashlib.sha3_256(msg.encode("utf8")).digest()
  return int.from_bytes(hash_bytes, byteorder="big")

msg = "some message"
msg_hash = sha3_hash(msg)
private_key = 99999999999999999999999999999999999999999999
signature = secp256k1.Generator.sign(self=secp256k1.secp256k1_generator,
secret_exponent=private_key, val=msg_hash)
print("signature = " + str(signature))
```

Example: ECDSA in Python – Verify a Signature

```
public_key = (secp256k1.secp256k1_generator * private_key)
print("public key: " + str(public_key))

valid = secp256k1.Generator.verify(self=secp256k1.secp256k1_generator,
  public_pair=public_key, val=msg_hash, sig=signature)
print("Signature valid? " + str(valid))

tampered_msg_hash = sha3_hash("tampered msg")
valid = secp256k1.Generator.verify(self=secp256k1.secp256k1_generator,
  public_pair=public_key, val=tampered_msg_hash, sig=signature)
print("Signature (tampered msg) valid? " + str(valid))
```

Select Command Prompt - python

```
>>> from pycoin.ecdsa import generator_secp256k1, sign, verify
>>> import hashlib
>>>
>>> def keccak_hash(msg) :
...   hash_bytes = hashlib.sha3_256(msg.encode("utf8")).digest()
...   return int.from_bytes(hash_bytes, byteorder="big")
...
>>> msg = "some message"
>>> msg_hash = keccak_hash(msg)
>>> private_key = 9999999999999999999999999999999999999999999
>>> signature = sign(generator_secp256k1, private_key, msg_hash)
>>> print("signature = " + str(signature))
signature = (1925212439601568054285967497901230258858462955656896
8901492225663695506442908, 560797074966800541568893068080735474561
78875133099061833359366964281484733604)
>>>
```

Command Prompt - python

```
>>> public_key = (generator_secp256k1 * private_key).pair()
>>> print("public key: " + str(public_key))
public key: (9097862414606533331137147129398896310883556996345088
3730382388108303588651424, 110364029629519587207785954936721854056
293761724232019418865066994981555675473)
>>>
>>> valid = verify(generator_secp256k1,
...   public_key, msg_hash, signature)
>>> print("Signature valid? " + str(valid))
Signature valid? True
>>>
>>> tampered_msg_hash = keccak_hash("tampered msg")
>>> valid = verify(generator_secp256k1, public_key,
...   tampered_msg_hash, signature)
>>> print("Signature (tampered msg) valid? " + str(valid))
Signature (tampered msg) valid? False
>>>
```

ECDSA:
SIGN / VERIFY SIGNATURE

Live Demo

Ethereum Addresses and secp256k1

- The **private key** in secp256k1 is **256-bit integer** (32 bytes)
  - Example of **Ethereum private key** (encoded as 64 hex digits)

```
97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a
```

- The **public key** is a EC point (2 * 256 bits == **64 bytes**)

```
7b83ad6afb1209f3c82ebeb08c0c5fa9bf6724548506f2fb4f991e2287a77090
177316ca82b0bdf70cd9dee145c3002c0da1d92626449875972a27807b73b42e
```

- Can be **compressed** to 257 bits (Ethereum uses prefix 02 or 03)
  - Example of **compressed public key** (33 bytes / 66 hex digits):

```
027b83ad6afb1209f3c82ebeb08c0c5fa9bf6724548506f2fb4f991e2287a77090
```

ECDSA, secp256k1 and Ethereum (2)

- The **blockchain address** in Ethereum is **20** bytes
  - Calculated as: **`last20bytes(keccak256(publicKeyFull))`**
  - Example of **Ethereum address** (encoded as 40 hex digits):

```
a44f70834a711F0DF388ab016465f2eEb255dEd0
```

  - Note: some letters are capital to incorporate a checksum (EIP55)
- Digital **signatures** in secp256k1 are **64 bytes** (2 * 32 bytes)
  - A pair of two 256-bit numbers: [**r**, **s**]
  - Calculated by the well-known ECDSA formulas (see RFC6979)

Ethereum Key to Addresses – Example

```
pip install eth_keys
```

To simplify **eth_keys** installation, you may use Anaconda: https://anaconda.com/download

```
import eth_keys, binascii

privKey = eth_keys.keys.PrivateKey(binascii.unhexlify(
  '97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a'))
print('Private key (64 hex digits):', privKey)

pubKey = privKey.public_key
print('Public key (plain, 128 hex digits):', pubKey)

pubKeyCompr = '0' + str(2 + int(pubKey) % 2) + str(pubKey)[2:66]
print('Public key (compressed, 66 hex digits):', pubKeyCompr)

address = pubKey.to_checksum_address()
print('Ethereum address:', address)
```

Verifying an Ethereum Signature

- Ethereum uses secp256k1-based ECDSA signatures
  - ECDSA generates deterministically a random point **R** (see RFC6979)
- **Ethereum signatures** consists of 3 numbers: [**v**, **r**, **s**]
  - **v** – the compressed **Y** coordinate of the point **R** (1 byte: 00 or 01)
  - **r** – the **X** coordinate of the point **R** (256-bit integer, 32 bytes)
  - **s** – 256-bit integer (32 bytes), calculated from the signer's **private key** + **message hash** (Ethereum uses keccak256)
  - Typically encoded as **130 hex digits** (65 bytes), e.g. **0x…465c5cf4be401**
- Given an Ethereum signature [**v**, **r**, **s**], the public key can be **recovered** from [**R**, **s**, **msgHash**] → also the signer's Ethereum **address**

ENSURING THE FUTURE OF BLOCKCHAIN

Sign Message in Ethereum – Example

```python
import eth_keys, binascii

privKey = eth_keys.keys.PrivateKey(binascii.unhexlify(
  '97ddae0f3a25b92268175400149d65d6887b9cefaf28ea2c078e05cdc15a3c0a'))
print('Private key (64 hex digits):', privKey)

signature = privKey.sign_msg(b'Message for signing')

print('Signature: [v = {0}, r = {1}, s = {2}]'.format(
  hex(signature.v), hex(signature.r), hex(signature.s)))
print('Signature (130 hex digits):', signature)
```

Select Command Prompt - python

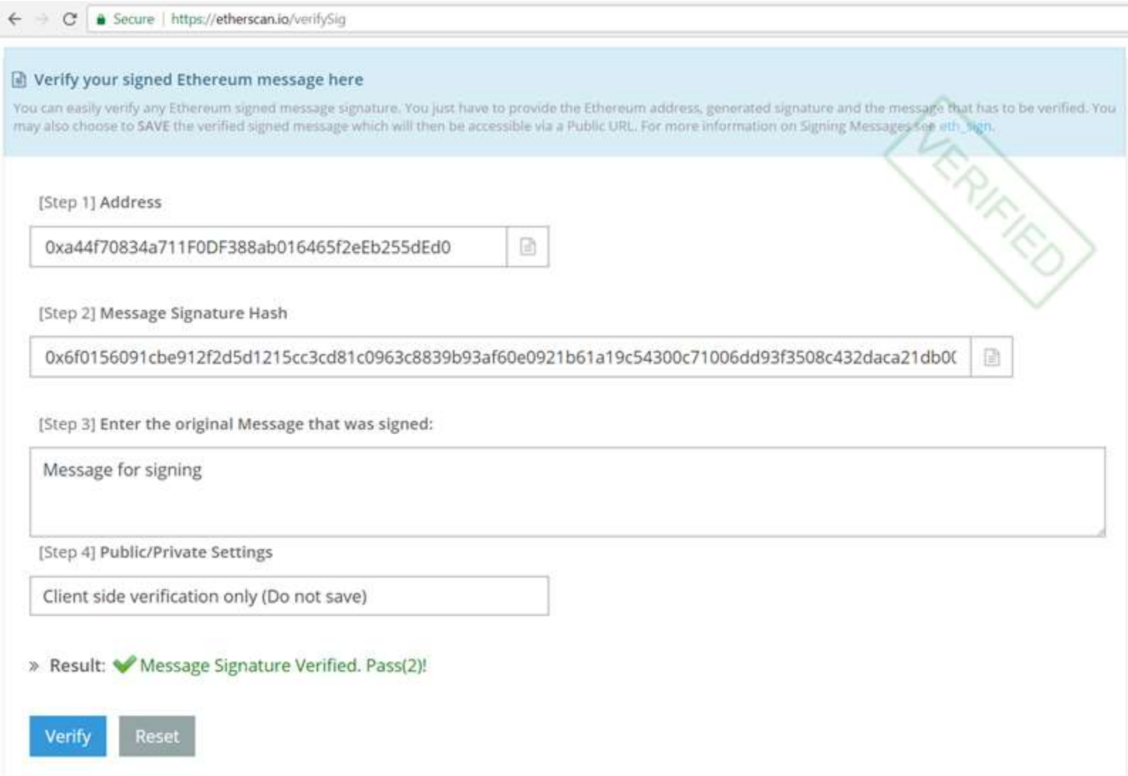
```
Signature: [v = 0x1, r = 0x6f0156091cbe912f2d5d1215cc3cd81c0963c8839b93af60e0921b61a19c54
30, s = 0xc71006dd93f3508c432daca21db0095f4b16542782b7986f48a5d0ae3c583d4
>>> print('Signature (130 hex digits):', signature)
Signature (130 hex digits): 0x6f0156091cbe912f2d5d1215cc3cd81c0963c8839b93af60e0921b61a19
c54300c71006dd93f3508c432daca21db0095f4b16542782b7986f48a5d0ae3c583d401
```

Verify Message Signature in Etherscan

Verify message signature at https://etherscan.io/verifySig by:

- **signer address** (40 hex digits)
- **signature** (130 hex digits)
- original **message** text

The result is: **valid** / **invalid**



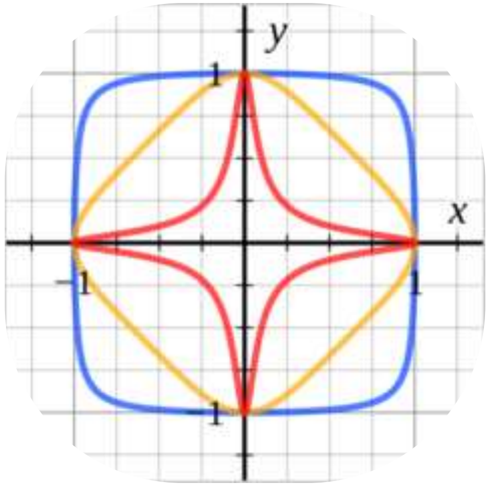
Verify Ethereum Signature – Example

```python
import eth_keys, binascii

msg = b'Message for signing'
msgSigner = '0xa44f70834a711F0DF388ab016465f2eEb255dEd0'
signature = eth_keys.keys.Signature(binascii.unhexlify(
'6f0156091cbe912f2d5d1215cc3cd81c0963c8839b93af60e0921b61a19c54300c71006d
d93f3508c432daca21db0095f4b16542782b7986f48a5d0ae3c583d401'))

signerPubKey = signature.recover_public_key_from_msg(msg)
print('Signer public key (recovered):', signerPubKey)

signerAddress = signerPubKey.to_checksum_address()
print('Signer address:', signerAddress)
print('Signature valid?:', signerAddress == msgSigner)
```

EdDSA and Ed25519

- Edwards-curve Digital Signature Algorithm (**EdDSA**) uses twisted Edwards curves, designed by D. Bernstein and others
- **Ed25519** is a cryptosystem, based on **ECC** (elliptic-curve cryptographic) – http://ed25519.cr.yp.to
  - Uses the Edwards's curve Curve25519 (RFC7748)

$$x^2 + y^2 = 1 + dx^2y^2$$



  - **EdDSA** (using curve25519) is **faster** than **ECDSA** (using secp256k1) at similar level of security (even slightly better)

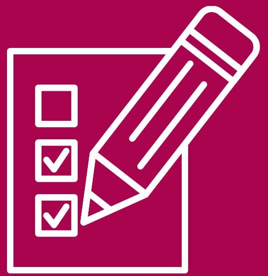
ENSURING THE FUTURE OF BLOCKCHAIN

Example: Ed25519 in Python

```
from pure25519 import ed25519_oop

privKey, pubKey = ed25519_oop.create_keypair()
print("Private key (32 bytes):", privKey.to_ascii(encoding='hex'))
print("Public key (32 bytes): ", pubKey.to_ascii(encoding='hex'))

msg = b'Message for signing'
signature = privKey.sign(msg, encoding='hex')
print("Signature (64 bytes):", signature)

try:
    pubKey.verify(signature, msg, encoding='hex')
    print("The signature is valid.")
except:
    print("Invalid signature!")
```

```
pip install pure25519
```

ENSURING THE FUTURE OF BLOCKCHAIN

EXERCISES

Ethereum: Private Key to Address, Signing Messages and Verifying Signatures

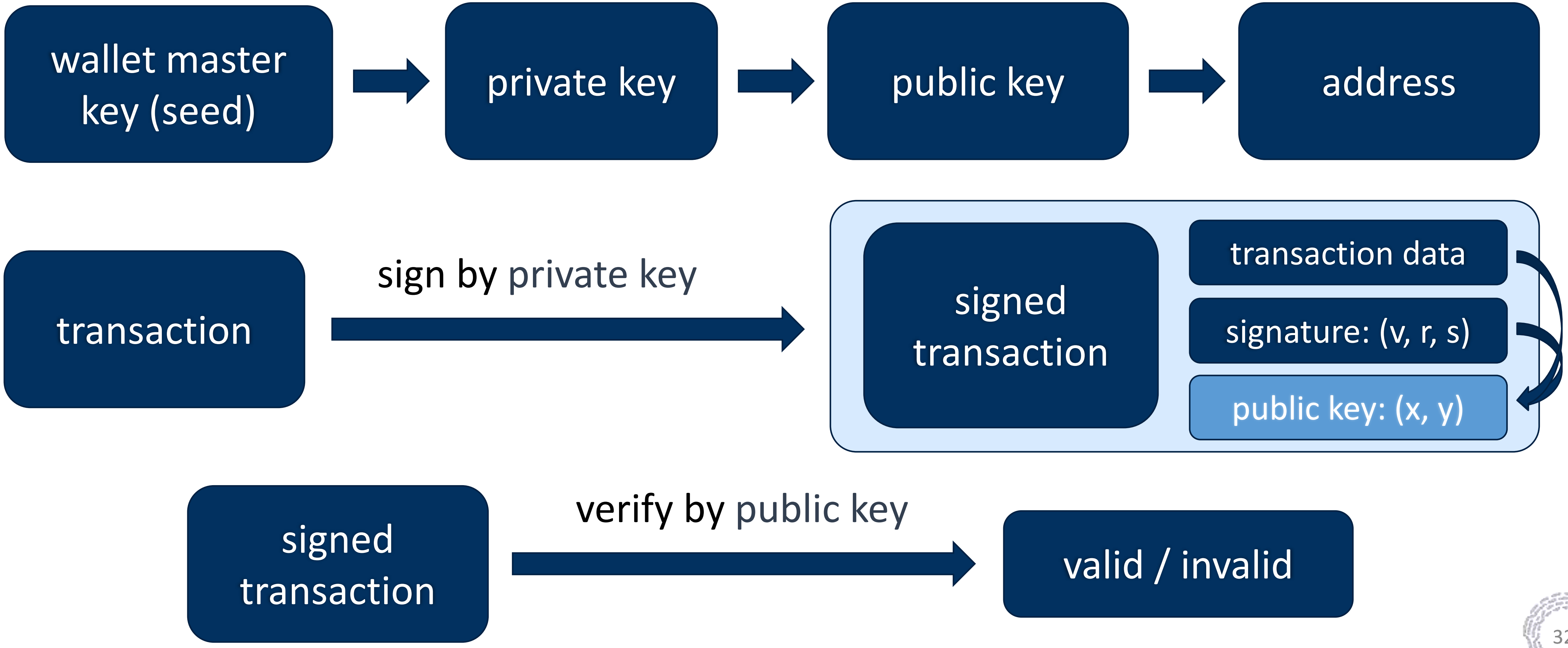
BLOCKCHAIN CRYPTOGRAPHY

Keys, Addresses, Signatures, Wallets

Public Blockchains and Cryptography

- Public blockchain networks (like Bitcoin and Ethereum) are mainly based on **ECC** (**elliptic curve cryptography**)
- **Bitcoin**, **Ethereum**, **EOS**, **Tron** use ECC with the **secp256k1** curve
- **Ripple**, **Cardano**, **Stellar** use the **Ed25519** ECC cryptosystem
- **NEO** uses ECC with curve **secp256r1** (it is similar to secp256k1)
- **IOTA** uses **Winternitz hash-based cryptography** (quantum-safe)
- **Monero** / **CryptoNote** use **Ed25519** + unique ring signatures
- **Dash** uses ECC with **secp256k1** + coin-mixing of transactions

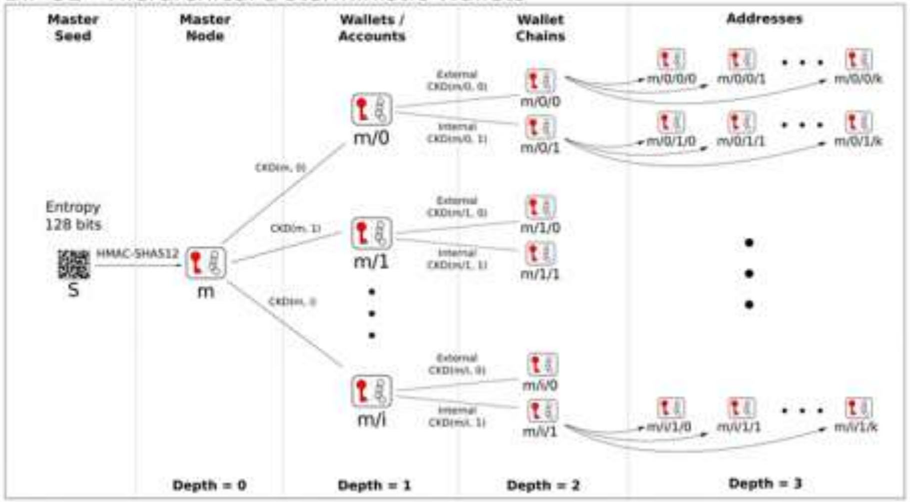
Public / Private Keys, Wallets & Blockchain



ECC and Wallets: BIP32

- The **BIP-32 standard** defines how a crypto-wallet can generate multiple keys + addresses
  - The wallet is initialized by a **256-bit master key** (seed)
  - **HMAC** + **ECC** math used to generate multiple accounts
    - Through a **derivation path**
    - E.g. **m/44'/60'/1'/12**
    - Each account holds **private key → public key → address**

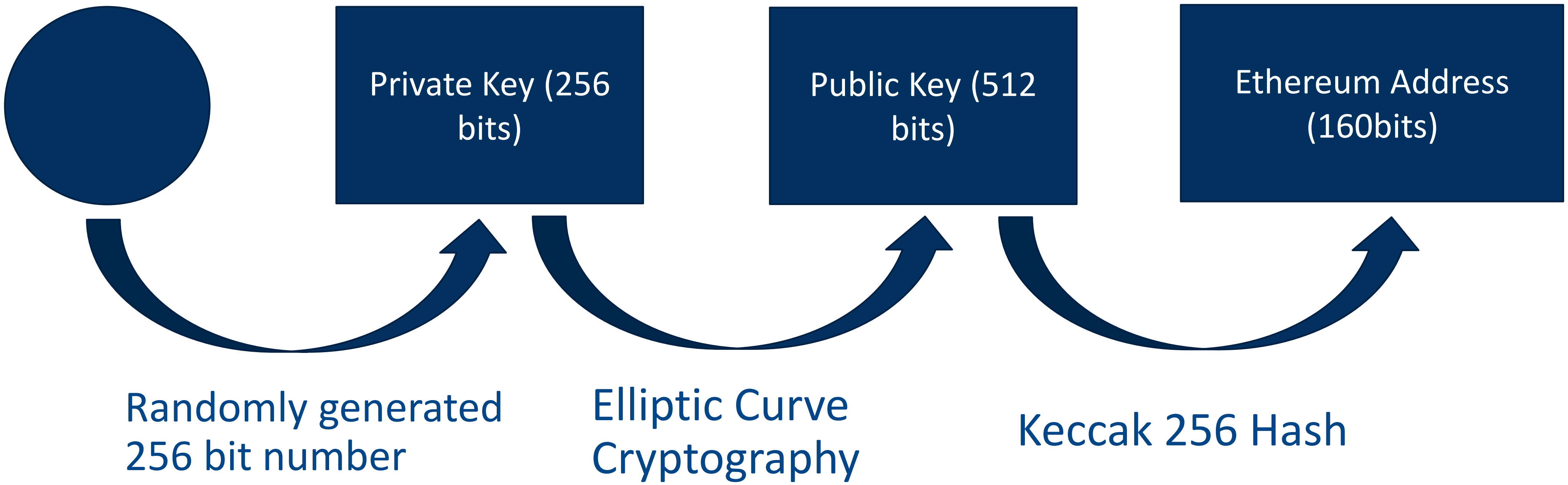
BIP 32 - Hierarchical Deterministic Wallets



Child Key Derivation Function ~ $CKD(x,n) = HMAC\text{-}SHA512(x_{Chain}, x_{PubKey} || n)$

Generating an Ethereum Address

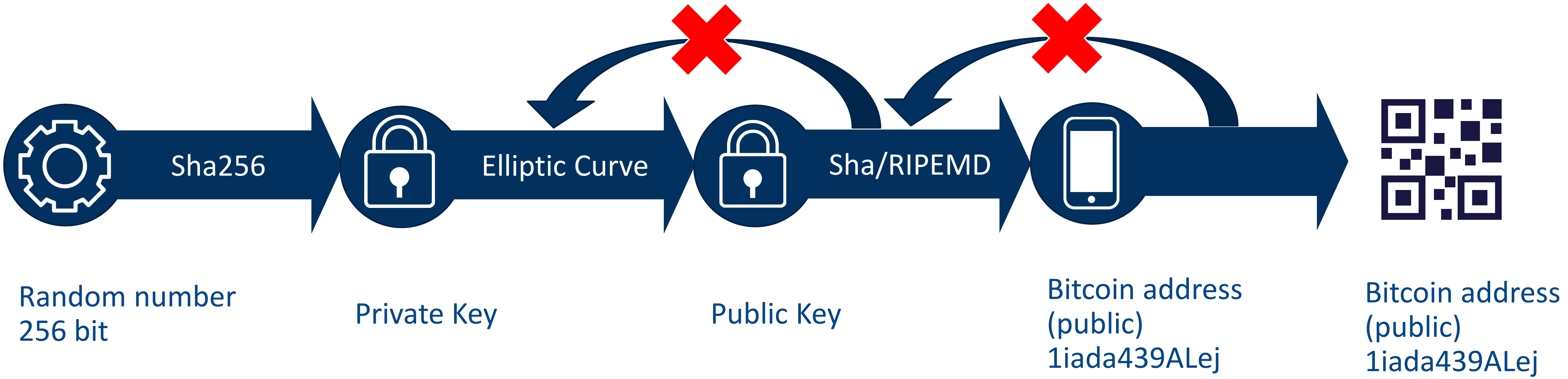
- Wallet seed **S** / CSPRNG → private key **k** → public key **P** → **h** = Keccak256(**P**) → Last160bits(**h**) → address **A**



- Address **collision** probability = 1 / 2^{160}

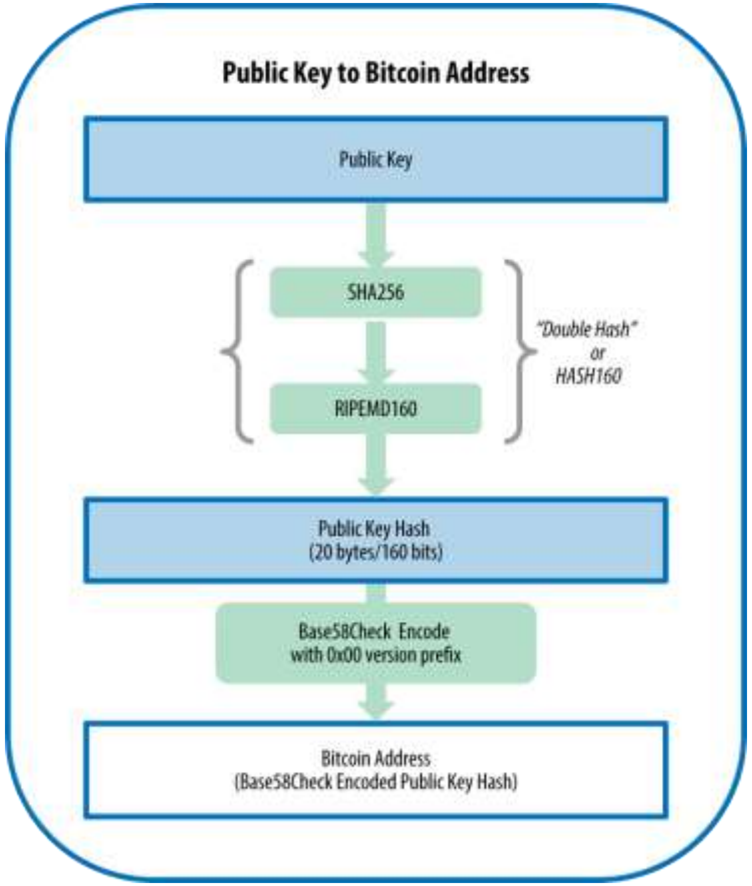
Generating a Bitcoin Address

- Wallet seed **S** / CSPRNG → private key **k** → public key **P** → WIF compressed public key **W** → RIPEMD160(SHA256(**W**)) → address **A**



Public Key → Bitcoin Address

1. Generate random private key **k**
2. Derive the **public key P** from it
3. **Pc = CompressPublicKey(P)**
4. **hash = RIPEMD160(SHA256(Pc))**
5. **Base58CheckEncode(hash, network)**
   - Calculate **checksum = SHA256(SHA256(network + hash))** [:4]
   - **Base58Encode (network + hash + checksum)**



Generate Bitcoin Address in Python

```
pip install bitcoin
```

```
import bitcoin, hashlib, binascii

def private_key_to_public_key(privKeyHex: str) -> (int, int):
  privateKey = int(privKeyHex, 16)
  return bitcoin.fast_multiply(bitcoin.G, privateKey)

def pubkey_to_address(pubKey: str, magic_byte = 0) -> str:
  pubKeyBytes = binascii.unhexlify(pubKey)
  sha256val = hashlib.sha256(pubKeyBytes).digest()
  ripemd160val = hashlib.new('ripemd160', sha256val).digest()
  return bitcoin.bin_to_b58check(ripemd160val, magic_byte)
```

Generate Bitcoin Address in Python (2)

```
private_key = bitcoin.random_key()
print("Private key (hex):", private_key)

public_key = private_key_to_public_key(private_key)
print("Public key (x,y) coordinates:", public_key)

compressed_public_key = bitcoin.compress(public_key)
print("Public key (hex compressed):", compressed_public_key)

address = pubkey_to_address(compressed_public_key)
print("Compressed Bitcoin address (base58check):", address)
```

Command Prompt - python

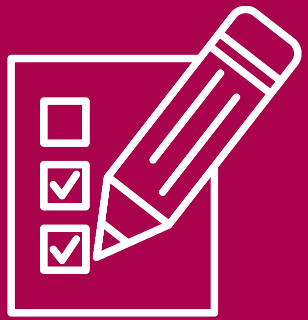
```
>>> address = pubkey_to_address(compressed_public_key)
>>> print("Compressed Bitcoin address (base58check):", address)
Compressed Bitcoin address (base58check): 1NFrYp9xyD9JPUVRazjP4qWGSqomeFEJE
>>>
```

ENSURING THE FUTURE OF BLOCKCHAIN

LIVE DEMO:

GENERATE A BITCOIN ADDRESS

https://www.bitaddress.org

EXERCISES

Calculating Hashes and Bitcoin Addresses

POST-QUANTUM CRYPTOGRAPHY

Quantum-Resistant Crypto Algorithm

ECC Cryptography is Quantum Unsafe!

- A **k**-bit number can be factored in time of order O(**k**^3) using a quantum computer of 5**k**+1 **qubits** (using Shor's algorithm)
  - See http://www.theory.caltech.edu/~preskill/pubs/preskill-1996-networks.pdf
  - **256-bit number** (e.g. Bitcoin public key) can be factorized using 1281 qubits in 72*256^3 quantum operations
    - ~ **1.2 billion operations** == ~ **less than 1 second** using good machine
  - **ECDSA**, **DSA**, **RSA**, **ElGamal** cryptosystems are all **quantum-broken**
- **Conclusion**: publishing signed transactions (like Ethereum does) is not quantum safe → avoid revealing the ECC public key

Hashes are Quantum Safe

- Cryptographic **hashes** (SHA256 / SHA3) are **quantum-safe**:
  - On traditional computer, finding a **collision** takes $\sqrt{N}$ **steps** (due to the birthday paradox) → SHA256 has 2^{128} crypto-strength
  - Quantum computers might find hash collisions in $\sqrt[3]{N}$ **operations** (the BHT algorithm), but this is disputed (see [Bernstein 2009])
  - It might take $\mathbf{2^{85}}$ **quantum operations** to find **SHA256 / SHA3** collision, but in practice it will cost significantly more
- **Conclusion**: SHA256/SHA3-256 are most probably quantum-safe
  - SHA384, SHA512 and SHA3-384, SHA3-512 are quantum-safe

Symmetric Ciphers are Quantum Safe

- AES encryption / most **symmetric ciphers** are quantum-safe:
  - Grover's algorithm finds AES secret key in $\sqrt{N}$ quantum operations
  - Quantum era will **double the key size** of the symmetric ciphers (see http://cr.yp.to/codes/grovercode-20100303.pdf)
- AES-256 in the post-quantum era is like AES-128 before
  - 128-bits or less symmetric ciphers are quantum-attackable
- **Conclusion: 256-bit** symmetric ciphers are **quantum safe**
  - AES-256, Twofish-256, Camellia-256 are considered quantum-safe

Post-Quantum Cryptography

- **Hashes** (like SHA256 / SHA3), **HMAC**, **Bcrypt**, **Scrypt** are basically **quantum-safe** (only slightly affected by quantum computing)
  - Use 384-bits or more to be quantum-safe (256-bits should be enough for long time)
- **Symmetric ciphers** (like AES-256, Twofish-256) are **quantum-safe**
  - Use 256-bits or more as key length (don't use 128-bit AES)
- **RSA**, **DSA**, **ECDSA**, **DHKE** are **quantum-broken**!
  - Use **quantum-safe signatures** (e.g. lattice-based or hash-based)
  - See https://en.wikipedia.org/wiki/Post-quantum_cryptography

Summary

- **Public** / **private key cryptography** is widely used in the blockchain technologies
- **ECDSA** is elliptic curve cryptosystem: **sign** / **verify**
- **Elliptic curves** are not quantum safe

Blockchain Dev Course: Welcome

Questions?

KINGSLAND SCHOOL OF BLOCKCHAIN

THANK YOU

ENSURING THE FUTURE OF BLOCKCHAIN