

Bitcoin and Blockchain Technology

Cryptography

ferdinando@ametrano.net 

<https://github.com/fametrano> 

<https://twitter.com/Ferdinando1970> 

<https://speakerdeck.com/nando1970> 

<https://www.reddit.com/user/Nando1970/> 

<https://www.slideshare.net/Ferdinando1970> 

<https://it.linkedin.com/in/ferdinandoametrano> 

<https://www.youtube.com/c/FerdinandoMAmetrano> 



POLITECNICO
MILANO 1863

Table of Contents

1. **Hash Functions**
2. Modular Arithmetic and Algebra of Sets
3. Elliptic Curves
4. Asymmetric Cryptography and Signature Algorithms
5. Elliptic Curve Signature Algorithms
6. **Public/Private** Keys and Bitcoin **Address/WIF**

Hash Functions

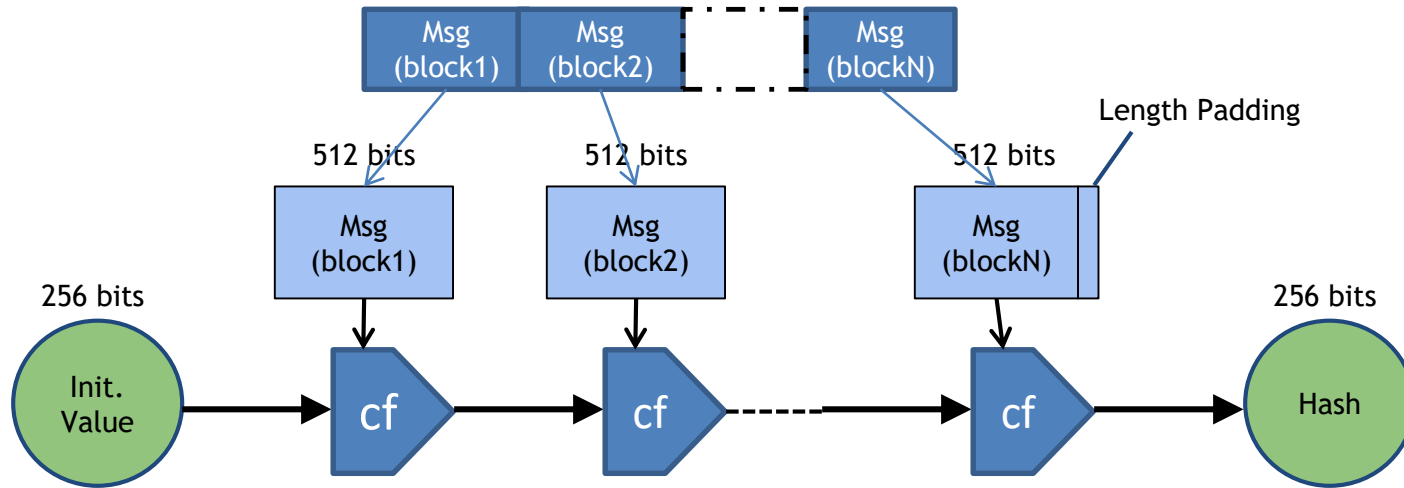
- A hash function is a map from the set of input data (of arbitrary length) to the output set of hash values (bit-string of fixed length)
- Small differences in the input data produce large differences in the result

Properties of Hash Functions

1. Arbitrary message size: $h(x)$ can be applied to message x of any size
2. Fixed output lengths: $h(x)$ produces a hash value of fixed length (SHA256 uses 256 bits)
3. Efficiency: $h(x)$ is relatively easy to compute

Merkle–Damgård hash construction

Hash function is based on a one-way *compression function*; it is as resistant to collisions as its compression function



Cryptographic Hash Functions

4. One-wayness (*preimage resistance*): Given $h(x)$, it is computationally infeasible to find x .
 5. Weak collision resistance (*second pre-image resistance*): Given x it is computationally infeasible to find $y \neq x$ such that $h(y)=h(x)$
 6. (Strong) Collision resistance: it is computationally infeasible to find $(x, y \neq x)$ such that $h(y)=h(x)$
- Collision resistance implies second preimage resistance
 - Second preimage resistance implies preimage resistance
- Hierarchical properties: the converse is not true in general

Computationally Infeasible Collisions

- The size of the possible hash values is smaller than the size of possible input data. Therefore, many input data points will share a single hash value output: collisions do exist...
- Try 2^{130} randomly chosen inputs: 99.8% chance that two of them will collide. This works, but it takes too long
- Is there a faster way to find collisions? For some hash functions, yes; for the “good” functions, we don’t know of one.

Application: Hash as message digest

- If we know $h(x) = h(y)$, it is safe to assume that $x = y$
- Useful because $h(x)$ can be much smaller than x

Puzzle friendliness

Given x and a target set Y , to find r from high min-entropy distribution such that

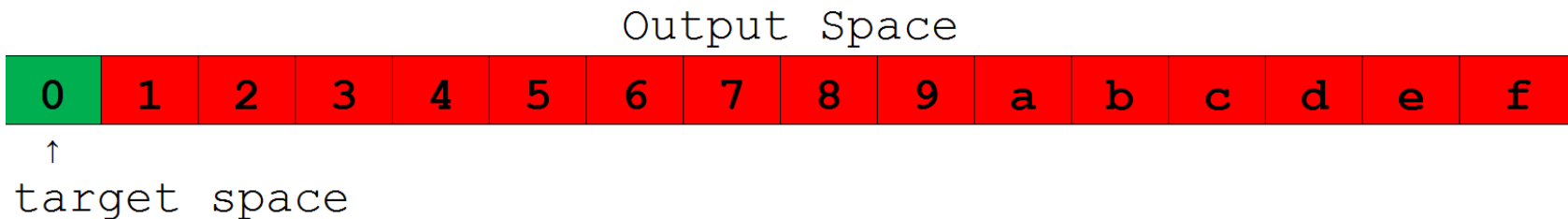
$$h(x|r) \in Y$$

there is no solving strategy better than trying random values of r

Min-entropy measures how likely you are to guess a value on your first try. If this probability is p , then the min-entropy is defined as $-\log_2 p$. For example, for a fair coin toss, you'd have $p = 0.5$, giving a min entropy of 1 bit. A uniformly random 256-bit string would have $-\log_2 2^{-256} = 256$ bits of min entropy

Partial Hash Inversion

- Find *nonce* so that $h(x|nonce)$ is small, e.g. starts with zero



- The smaller the target space, the harder to find a solution
- For good hash functions, the problem can only be solved by brute force trials
- It is always trivial to verify the solution: it is just one hash function evaluation

Homework #1

- Find the nonce that appended to your name obtains a hash value starting with 7 zeros
- How many numbers in the range $[0, N]$ result into hash values starting with zeros?
- Produce the histogram with the frequency of hash values starting with 1, 2, 3, 4, 5, 6, and 7 zeros in the $[0, N]$ range

Check scripts/sha256.py at <https://github.com/fametrano/BitcoinBlockchainTechnology>

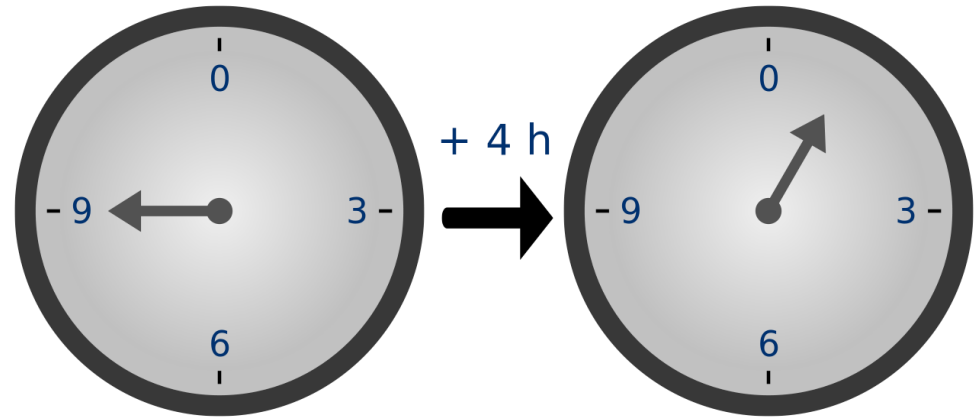
Table of Contents

1. Hash Functions
- 2. Modular Arithmetic and Algebra of Sets**
3. Elliptic Curves
4. Asymmetric Cryptography and Signature Algorithms
5. Elliptic Curve Signature Algorithms
6. **Public/Private** Keys and Bitcoin **Address/WIF**

Modular Arithmetic

arithmetic for integers: numbers "wrap around"
upon reaching the *modulus* value

$$9 + 4 = 1 \pmod{12}$$



Source: https://commons.wikimedia.org/wiki/File:Clock_group.svg

Congruence and Remainders

If $a_1 \equiv b_1 \pmod{n}$ and $a_2 \equiv b_2 \pmod{n}$ then

- $a_1 + a_2 \equiv b_1 + b_2 \pmod{n}$
- $a_1 - a_2 \equiv b_1 - b_2 \pmod{n}$
- $a_1 a_2 \equiv b_1 b_2 \pmod{n}$
- $(a \bmod n)(b \bmod n) \equiv ab \pmod{n}$
- $((a \bmod n)(b \bmod n)) \bmod n = (ab) \bmod n$

Group $(G, +)$

A set G together with an operation $+$ that combines any two elements a and b to form another element $a+b$ is a group if:

- Closure: for all a and b in G , $a+b$ is also in G
- Associativity: for all a , b and c in G , $(a+b)+c=a+(b+c)$
- Identity: there exists an (unique) element 0 in G , such that for every element a in G , the equation $0+a=a+0=a$
- Invertibility: for each a in G , there exists an element b in G , commonly denoted $-a$, such that $a+b=b+a=0$

A group is commutative if for all a and b in G , $a+b=b+a$

e.g. integers under addition $(\mathbb{Z}, +)$ are a commutative group, integers under multiplication (\mathbb{Z}, \bullet) are not a group (the multiplicative inverse of 2 is not an integer)

Ring and Field $(G, +, \bullet)$

- A ring is a group with a second operation that is associative and the distributive properties make the two operations “compatible”
- A field is a ring such that the second operation satisfies all the group properties, after throwing out the identity element of the first operation

Modular Addition and Multiplication

For any modulus p , $([0, p-1], +)$ is a commutative group

- 0 is the identity element
- The inverse of any element a is $p-a$

For any **prime number** p , $([1, p-1], \bullet)$ is a commutative group

- 1 is the identity element
- For any element a there exist its inverse $ab=1 \pmod{p}$. It is crucial here that p is prime

The Finite Field F_p

e.g. F_7

- $(\{0, 1, 2, 3, 4, 5, 6\}, +)$ is a commutative group
- $4+3 \%7 = 0 \rightarrow 3$ is the additive opposite of 4
- $(\{1, 2, 3, 4, 5, 6\}, \bullet)$ is a commutative group
- $4*2 \%7 = 1 \rightarrow 2$ is the multiplicative inverse of 4
- $4 = 2*2 \%7 \rightarrow 2$ is a (even) square root of 4
- $4 = 5*5 \%7 \rightarrow 5$ is a (odd) square root of 4

Division must be interpreted as multiplication by the inverse

$$-2 \%7 = 5$$

$$\text{odd root} + \text{even root} = 7$$

The Finite Field F_7

	opposite	inverse	odd sqrt	even sqrt
0	0	#N/A	0	0
1	6	1	1	6
2	5	4	3	4
3	4	5	#N/A	#N/A
4	3	2	5	2
5	2	3	#N/A	#N/A
6	1	6	#N/A	#N/A

Homework #2

- Calculate for F_{19} and F_{23} the table of opposites, inverses, and square roots

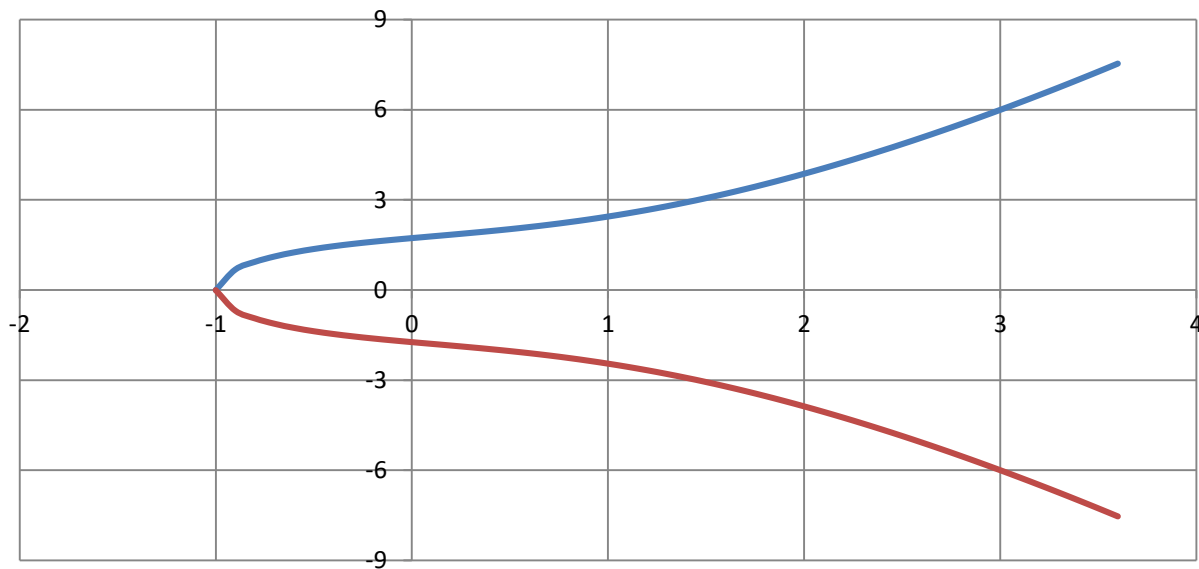
Check scripts/FiniteFields.py at <https://github.com/fametrano/BitcoinBlockchainTechnology>

Table of Contents

1. Hash Functions
2. Modular Arithmetic and Algebra of Sets
- 3. Elliptic Curves**
4. Asymmetric Cryptography and Signature Algorithms
5. Elliptic Curve Signature Algorithms
6. **Public/Private** Keys and Bitcoin **Address/WIF**

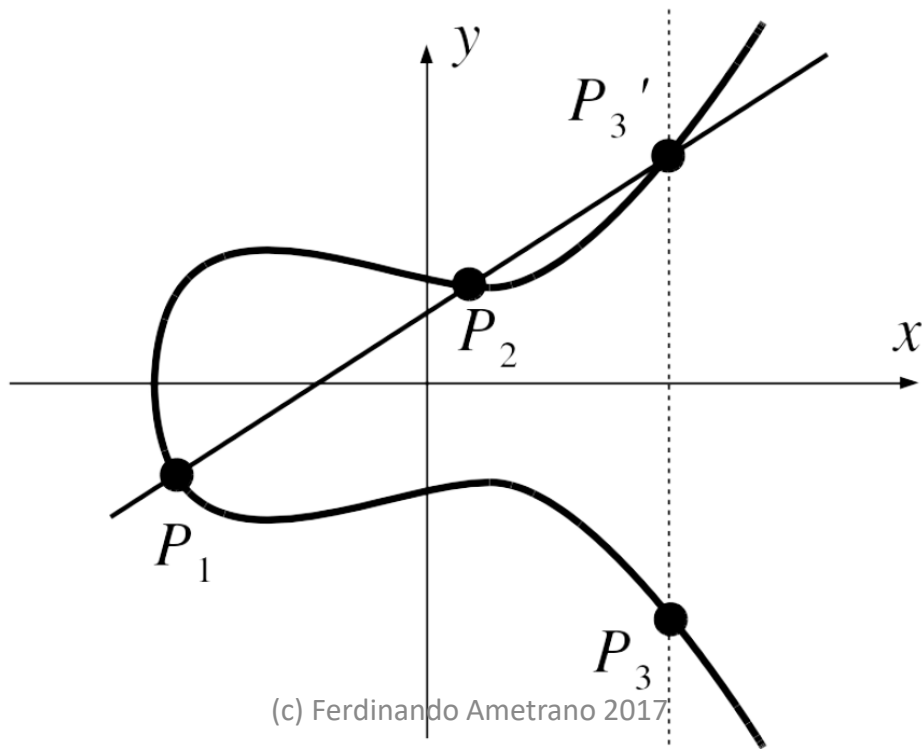
Elliptic Curves $y^2 = x^3 + ax + b$

e.g. $y^2 = x^3 + 2x + 3$



Point Addition $P_1 + P_2 = P_3$

Source: Pedro Franco, "Understanding Bitcoin", Wiley



(c) Ferdinando Ametrano 2017

Point Addition $P_1 + P_2 = P_3$

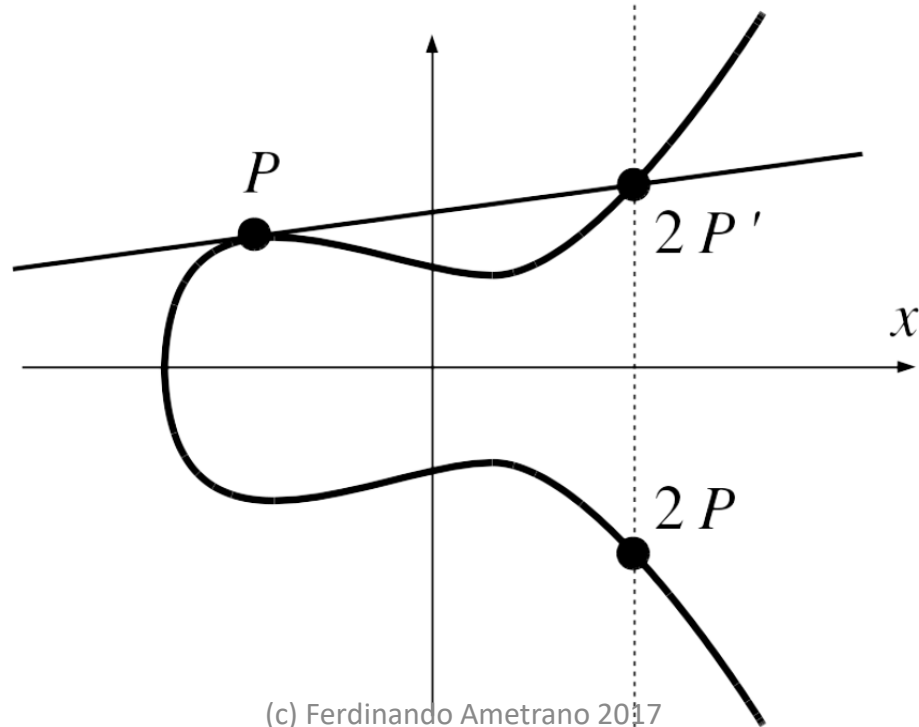
$$P_i = (x_i, y_i)$$

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1} \right)^2 - x_2 - x_1$$

$$y_3 = \frac{y_2 - y_1}{x_2 - x_1} (x_1 - x_3) - y_1$$

Point Doubling

Source: Pedro Franco, "Understanding Bitcoin", Wiley



(c) Ferdinando Ametrano 2017

Point Doubling $P_1 + P_1 = P_3$

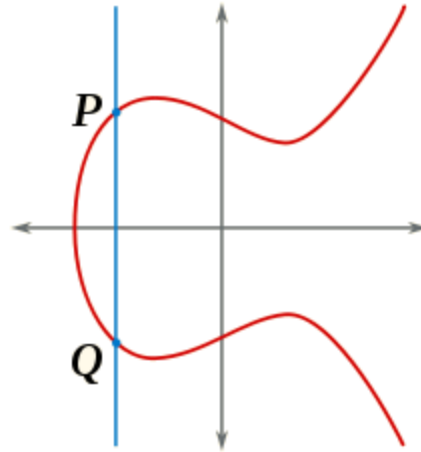
$$P_i = (x_i, y_i)$$

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1} \right)^2 - 2x_1$$

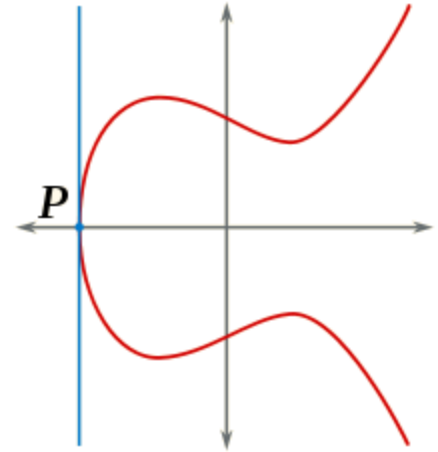
$$y_3 = \frac{3x_1^2 + a}{2y_1} (x_1 - x_3) - y_1$$

Infinity Point (Group Identity, Neutral Element)

- Zero in additive notation
- Identity in multiplicative notation
- Sometime also indicated as ∞
- $P + Q = 0 \rightarrow Q = -P$



$$P + Q = 0$$



$$2P = 0$$

Source:

https://en.wikipedia.org/wiki/Elliptic_curve#/media/File:ECCLines.svg

EC Commutative Group

- Points on an elliptic curve with the *addition operation* form a commutative group
- The point at infinity is the neutral element
- Arbitrarily named addition: it is the *group operation* and could have been called multiplication instead
- In multiplicative notation doubling would have been called squaring

Multiplication $P = nG$

Double and Add Algorithm

$$\text{e.g. } 947 = 2^0 + 2^1 + 2^4 + 2^5 + 2^7 + 2^8 + 2^9$$

$$947G = G + 2G + 16G + 32G + 128G + 256G + 512G$$

9 doublings and **6** additions: polynomial in the number of bits representing n . Much better than 946 additions!

```
def pointMultiply(n, G):  
    if n==1:  
        return G  
    elif n%2==1: # addition when n is odd  
        return pointAdd(G, pointMultiply(n-1, G))  
    else:        # doubling when n is even  
        return pointMultiply(n/2, pointDouble(G))
```

https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication

Discrete Logarithm

One Way Function

- With a known n it is easy and fast to compute

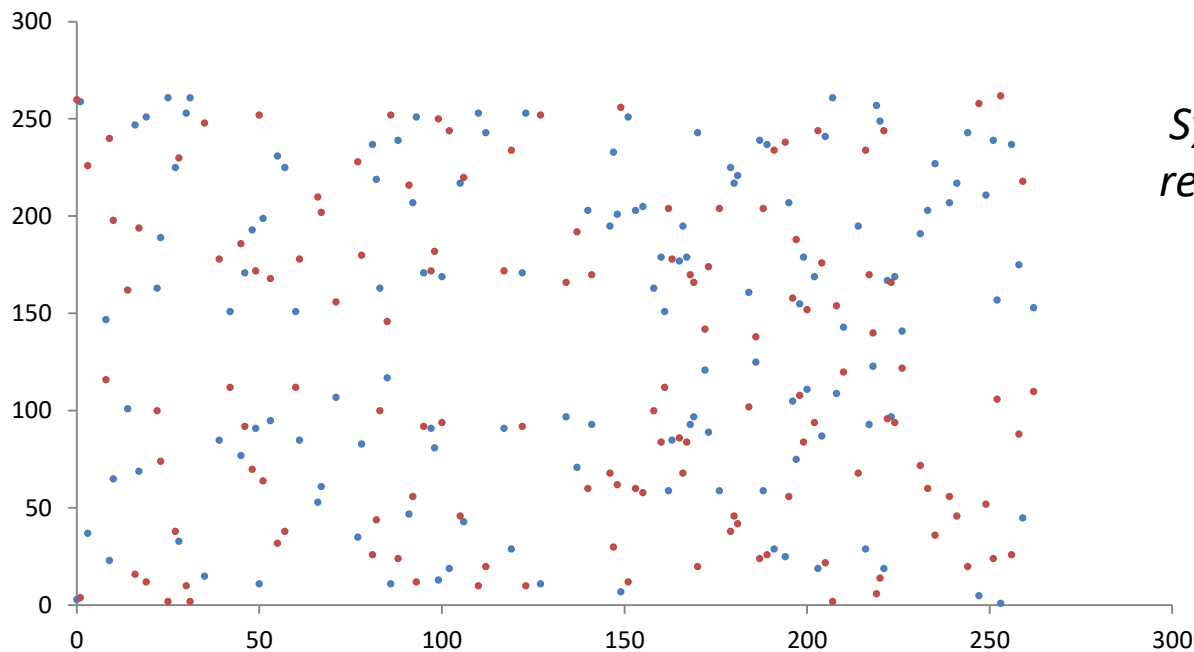
$$P = nG$$

- For $n \in [1, N - 1]$, to infer n from (P, G) is exponential in the number of bits representing N
- This inverse problem is known as **discrete logarithm**, computationally unfeasible for large N
- Multiplicative notation would have been $P = G^n$, leading to usual logarithm definition: $n = \log_G P$

Elliptic Curves Over a Finite Field F_p

$$y^2 = x^3 + ax + b \pmod{p}$$

e.g. $y^2 = x^3 + 6x + 9 \pmod{263}$



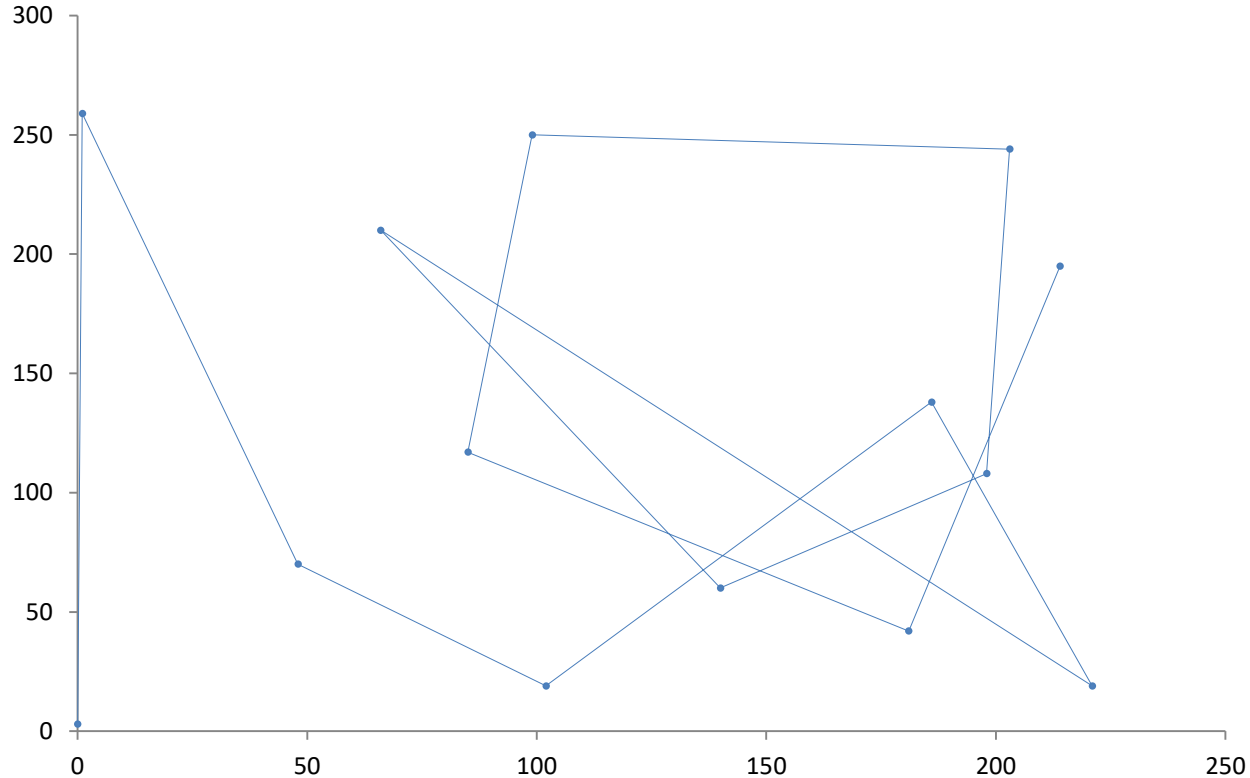
*Symmetric with
respect to $y=131$*

Finite Cyclic Group

- An elliptic curve defined *over a finite field* F_p is also a finite cyclic group
- Starting with an initial generator point G in the curve and adding this point successively, all the N points in the group are recovered. N is the group order
- Any point can be reached quickly from a predecessor if the number of steps is known
- $y^2 = x^3 + 9x + 9$ over F_{263} has 269 points: 268 affine points the point at infinity

$$y^2 = x^3 + 6x + 9 \text{ over } F_{263}$$

First 14 points starting with (0, 3)



Primality of the Field Order

The primality of the field order is crucial for:

- the modular inverse to exists
- the cyclic field not to have subgroups, e.g.
 $y^2 = x^3 + 2x + 3$ over F_{263}
 - with $G=(200, 39)$ has order 270
 - with $G=(72, 120)$ has order 90

Is the order is even, it does exist a point $P=(x,0)$ with $2P=0$, e.g. for $y^2 = x^3 + 2x + 3$ over F_{263} with $G=(200, 39)$, $P=(262,0)$

Elliptic Koblitz Curve secp256k1

Domain Parameters

- F_p is defined by $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$, i.e.
 $p = \text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFC2F}$
- The elliptic curve defined over F_p is $y^2 = x^3 + 7$
- The generation point $G =$
 $(79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798,$
 $483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8)$
- Finally the order of G :
 $\text{FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141}$

SECG, SEC 2: Recommended Elliptic Curve Domain Parameters, <http://www.secg.org/sec2-v2.pdf>
<https://en.bitcoin.it/wiki/Secp256k1>

Elliptic Curve Private/Public Key

- A **public key** is one point P on the elliptic curve
- A **private key** is the number p of additive steps from the generator point G to arrive at point P

$$P = p G$$

- In multiplicative notation p is called *secret exponent*

$$P = G^p$$

https://en.wikipedia.org/wiki/Elliptic_curve_cryptography

Homework #3

- Create a spreadsheet for

$$y^2 = x^3 + 2x + 2 \bmod 17$$

- List all its points
- It does not have subgroups, why?

- Create a spreadsheet for

$$y^2 = x^3 + 4x + 20 \bmod 29$$

- List all its points
- What is the order of the group with generator (8,10)?

Check scripts/F11_EC1_6_G5_9_N13.py, scripts/F263_EC6_9_G0_3_N269.py, and scripts/secp256k1.py at <https://github.com/fametrano/BitcoinBlockchainTechnology>

Table of Contents

1. Hash Functions
2. Modular Arithmetic and Algebra of Sets
3. Elliptic Curves
4. **Asymmetric Cryptography and Signature Algorithms**
5. Elliptic Curve Signature Algorithms
6. **Public/Private** Keys and Bitcoin **Address/WIF**

Asymmetric Cryptography Families

Key Generation Algorithms

- Integer factorization (1977), based on the difficulty of factoring large integers (e.g. RSA)
- Discrete Logarithm (1976), based on the intractability of the discrete logarithm problem on finite cyclic groups (e.g. Diffie and Hellman)
- Elliptic Curve (1985), based on the difficulty of computing the generalized logarithm problem on an elliptic curve (e.g. Bitcoin)

Break Elliptic Curve Cryptography

- The best known algorithms to break the EC discrete logarithm problem take steps proportional to $\sqrt{2^m}$ where m is the number of bits of the key
- secp256k1 uses 256bit keys: 2^{128} steps are needed to break it
- An EC computation takes 1 million CPU cycles. A 3GHz CPU is able to process $2^{11.55}$ EC computations per second
- A CPU can break the EC in $2^{116.45}$ seconds, or about $2^{91.54}$ years, i.e. about 3599861590422752583114293248 years
- Throwing a million CPUs at the problem would reduce the time by a million, leaving it at 3599861590422752583114 years, roughly 260,859,535,537 times the age of the universe

Key Size At Comparable Security Levels

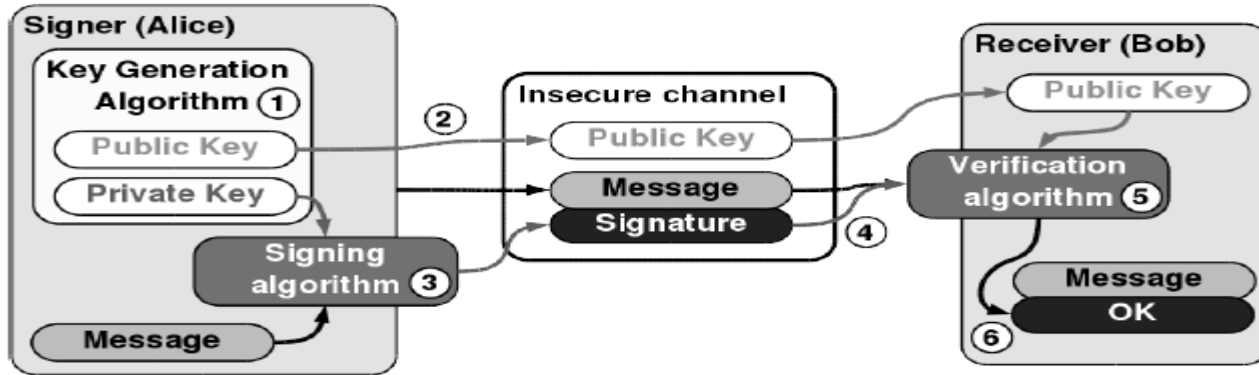
Source: Pedro Franco, "Understanding Bitcoin", Wiley

	Security level (bits)			
	80	128	192	256
RSA	1024	3072	7680	15360
DL	1024	3072	7680	15360
ECC	160	256	384	512
Symmetric	80	128	192	256

Digital Signature Protocol

- Public-key algorithm + digital signature scheme
- Message is only authenticated, not encrypted

Source: Pedro Franco, "Understanding Bitcoin", Wiley

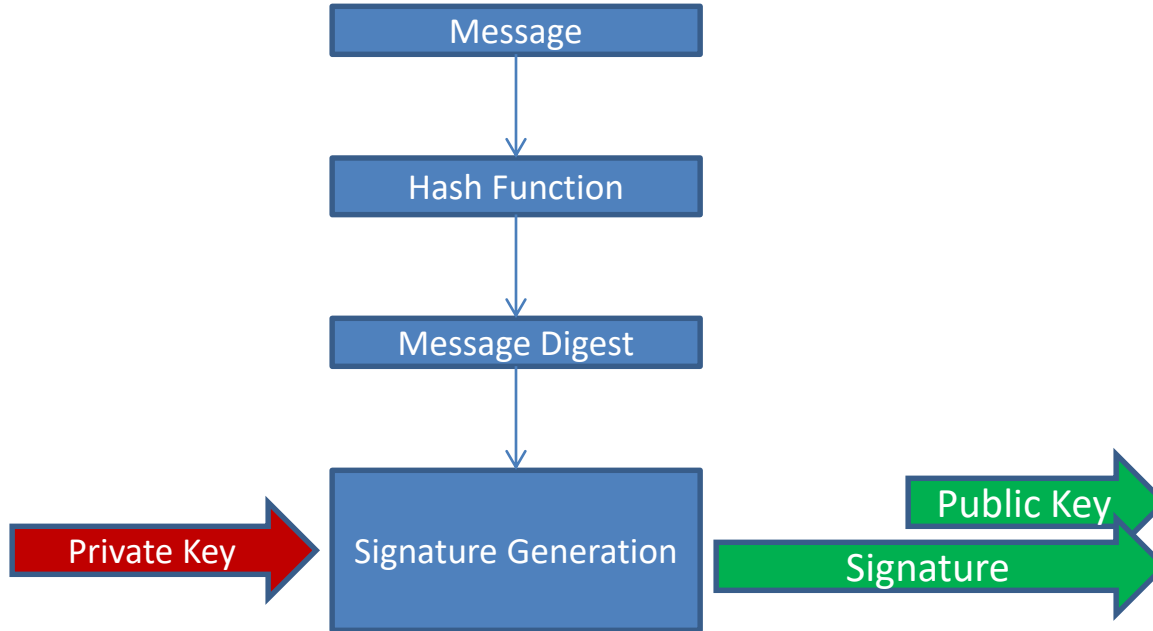


Signing The Message Digest

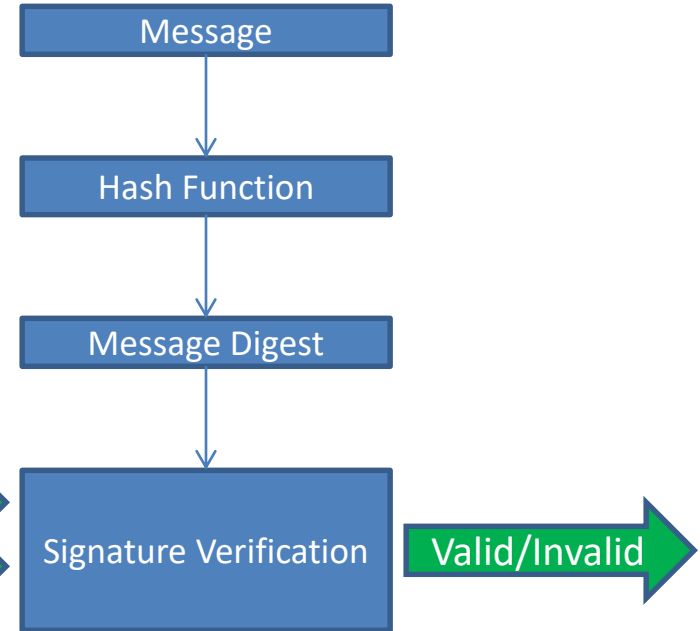
- Problem: signature generation/verification is quite slow: message length can be a problem
- Solution: sign the hash value of the message $h(msg)$, whose length is independent from the message's size
- This is also useful if, for some reason, the msg is to be revealed only after signature verification
 - If the msg can take only few values (e.g. {tail, head}), it can be concealed using an ephemeral number k : sign and reveal $h(msg|k)$

Digital Signature Process

Signature Generation



- Signature Verification



Digital Signature Algorithms

- **RSA**, the most widely used
- **Elgamal** signature. It has little use being computationally intensive and having large signature
- Digital Signature Algorithm (**DSA**), quicker and smaller than RSA: widely used
- **Schnorr** signature, the simplest scheme. Signing and verification are computationally efficient, signature is small. Not used so far because of patents

Table of Contents

1. Hash Functions
2. Modular Arithmetic and Algebra of Sets
3. Elliptic Curves
4. Asymmetric Cryptography and Signature Algorithms
- 5. Elliptic Curve Signature Algorithms**
6. **Public/Private** Keys and Bitcoin **Address/WIF**

EC DSA: Generation

1. Choose a nonce as **secret** ephemeral key
 $0 < k < \text{order}$
2. Compute $K = (x_K, y_K) = kG$
3. $s = (h + x_K p)k^{-1} \bmod \text{order}$
If $x_K = 0$ or $s = 0$ (extremely unlikely) then restart with another k

The signature of h is (x_K, s)

k must be secret: if revealed, $p = (sk - h)x_K^{-1} \bmod \text{order}$

EC DSA: Verification

Steps for the (x_K, s) verification of h

1. $u = h s^{-1} \bmod \text{order}$
2. $v = x_K s^{-1} \bmod \text{order}$
3. $(x, y) = uG + vP$
4. The signature is valid if $x = x_K \bmod \text{order}$

Malleability: if (x_K, s) is a valid signature of h , then also $(x_K, \text{order} - s)$ is a valid signature

Note that assuming a valid signature, $P = ((x_K, y_{\text{odd or even}}) - uG)/v$

EC DSA: Correctness Proof

$$x = x_K \bmod \text{order if } uG + vP = kG$$

1. $(u + v p)G = kG$
from public key definition
2. $(h s^{-1} + x_K s^{-1} p)G = kG$
from signature verification [2] and [3]
3. $(h + x_K p)s^{-1}G = kG$
4. $(h + x_K p)(h + x_K p)^{-1}kG = kG$
from signature generation [5]
5. $kG = kG$

Ephemeral Key Used for Signing

The ephemeral key k must remain secret: therefore, it is **to be used only once** per p . Reusing the nonce k would reveal it:

$$\begin{aligned}s_1 &= (h_1 + x_K p) k^{-1} \bmod \text{order} \\ s_2 &= (h_2 + x_K p) k^{-1} \bmod \text{order} \\ k &= (h_1 - h_2) / (s_1 - s_2) \bmod \text{order}\end{aligned}$$

Ask Sony PS3 developers and bitcoin owners using Android Wallet in 2013...

<http://www.bbc.com/news/technology-12116051>

<https://bitcoin.org/en/alert/2013-08-11-android>

Solution: use a deterministically different k for each msg , with k remaining secret because of p salting (RFC6979):

$$k = h(msg|p)$$

EC Schnorr SA: Generation

1. Choose a nonce as **secret** ephemeral key

$$0 < k < \textit{order}$$

2. Compute $K = kG$

3. $s = k - h p \bmod \textit{order}$

If $s = 0$ (extremely unlikely) then restart with another k

The signature of h is (K, s)

k must be secret: if revealed, $p = (k - s)h^{-1} \bmod \textit{order}$

EC SSA: Verification

Steps for the (K, s) verification of h

1. $V = K - h P$
2. The signature is valid if $sG = V$

Schnorr signature is **not** malleable

Note that assuming a valid signature, $P = (K - sG)/h$

EC SSS: Correctness Proof

$$sG = V$$

$$(k - h p)G = K - h P$$

$$(k - h p)G = kG - h pG$$

$$(k - h p)G = (k - h p)G$$

Ephemeral Key Used for Signing

For Schnorr too, the ephemeral key k must remain secret: therefore, it is **to be used only once** per p . Reusing the nonce k would reveal it:

$$\begin{aligned}s_1 &= k - h_1 p \text{ mod order} \\ s_2 &= k - h_2 p \text{ mod order} \\ p &= (s_1 - s_2)(h_2 - h_1)^{-1} \text{ mod order}\end{aligned}$$

Again: for each msg use a deterministically different k that remains secret because of p salting (RFC6979):

$$k = h(msg|p)$$

Homework #4

Calculate the public key(s) from this valid signature:

```
*** message hash and its signature
h: 9788fd27b3aafd1bd1591a1158ce2d8bdc37ab4040dddb64e64d17616e69ce2b
x: 2ab2a733eae4e67e06611aba01345b85cdd4f5ad44f72e369ef0dd640424dbfa
s: 27598b74fc77ee8aaa6f56d3f976949ac2c2f5849c98412d10ce02c170262be8
```

A second signature is computed in error using the same ephemeral key. Calculate the private key:

```
*** another message hash and its signature
h2: 7adb91982ec03ef87efcae7f0199aefa231d8855e0bd03319460e58c0bd18049
x: 2ab2a733eae4e67e06611aba01345b85cdd4f5ad44f72e369ef0dd640424dbfa
s2: 691d6fb6b4b90a8358a3b1c241bbc53b5be5bf52196561dbe5270ba1f54815a2
```

Check scripts/secp256k1 DSA.py and scripts/secp256k1 SchnorrSA.py (also F29_EC4_20_G8_10_N37 DSA.py and F29_EC4_20_G8_10_N37 SchnorrSA.py) at <https://github.com/fametrano/BitcoinBlockchainTechnology>

Table of Contents

1. Hash Functions
2. Modular Arithmetic and Algebra of Sets
3. Elliptic Curves
4. Asymmetric Cryptography and Signature Algorithms
5. Elliptic Curve Signature Algorithms
6. **Public/Private** Keys and Bitcoin **Address/WIF**

Public Key Compression

$$pG = P = (x, y)$$

- **Uncompressed** P representation is $04\ x\ y$

For every x , two y roots are possible:

$$y^2 = x^3 + 7$$

- Even y root, compressed P representation is $02\ x$
- Odd y root, compressed P representation is $03\ x$

Bitcoin Uses Base58 Digits

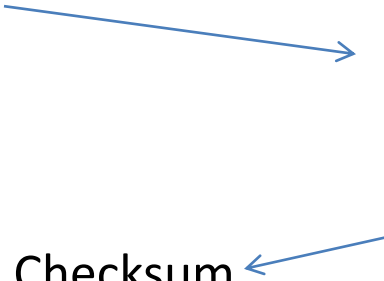
- Binary digits: 01
- Decimal digits: 0123456789
- Hexadecimal digits: 0123456789ABCDEF
- Base58 digits: all alphanumeric characters (numbers, uppercase, and lowercase) omitting 0 (zero), O (capital o), I (capital i) and l (lower case L)
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

Fewer digits to represent large numbers

Base58Check Encoding

includes a checksum algorithm, that helps detecting errors

1. Payload
 2. Prefix + Payload
 3. Prefix + Payload + Checksum
 4. Base58Check encoding
- Checksum Calculation***

 - a. Prefix+Payload
 - b. SHA256(Prefix+Payload)
 - c. SHA256(SHA256(Prefix+Payload))
 - d. Checksum = first 4 bytes of the previous step
- 

Base58 encoding can be decoded

Base58 Representation of Public/Private Keys

Base58 is used for compact representation of Bitcoin Public/Private Keys:

- Address: P in Base58 representation
- Wallet Import Format: p in Base58 representation

From Uncompressed Public Key to Bitcoin Address (1/4)

Having a private EC key

18E14A7B6A307F426A94F8114701E7C8E774E7F9A47E2C2035DB29A206321725

1. Start from the associated public key in **uncompressed** representation (65 bytes: 1 byte x04 prefix, 32 bytes corresponding to X coordinate, 32 bytes corresponding to Y coordinate)

04

50863AD64A87AE8A2FE83C1AF1A8403CB53F53E486D8511DAD8A04887E5B2352
2CD470243453A299FA9E77237716103ABC11A1DF38855ED6F2EE187E9C582BA6

From Uncompressed Public Key to Bitcoin Address (2/4)

2. Perform SHA-256 hashing on the public key

600FFE422B4E00731A59557A5CCA46CC183944191006324A447BDB2D98D4B408

3. Perform RIPEMD-160 hashing on the result of SHA-256

010966776006953D5567439E5E39F86A0D273BEE

4. Add version byte in front of RIPEMD-160 hash (x00 for Main Network) to obtain the extended RIPEMD-160

00010966776006953D5567439E5E39F86A0D273BEE

Note: step 2-3 make the PubKey->Address derivation not invertible

From Uncompressed Public Key to Bitcoin Address (3/4)

Base58Check encoding steps:

5. Perform SHA-256 hash on the extended RIPEMD-160

445C7A8007A93D8733188288BB320A8FE2DEBD2AE1B47F0F50BC10BAE845C094

6. Perform SHA-256 hash on the result of the previous step

D61967F63C7DD183914A4AE452C9F6AD5D462CE3D277798075B107615C1A8A30

7. Take the first 4 bytes of the second SHA-256 hash. This is the address checksum

D61967F6

From Uncompressed Public Key to Bitcoin Address (4/4)

8. Add the 4 checksum bytes from stage 7 at the end of extended RIPEMD-160

00010966776006953D5567439E5E39F86A0D273BEE**D61967F6**

9. Convert into a base58 string using Base58 encoding

16UwLL9Risc3QfPqBUvKofHmBQ7wMtjvM

https://en.bitcoin.it/wiki/Technical_background_of_version_1_Bitcoin_addresses

From Compressed Public Key to Bitcoin Address

Using the same private EC key as in the previous slide

18E14A7B6A307F426A94F8114701E7C8E774E7F9A47E2C2035DB29A206321725

- Start from the associated public key in compressed representation (33 bytes: 1 byte x02 or x03, 32 bytes corresponding to X coordinate)

02

50863AD64A87AE8A2FE83C1AF1A8403CB53F53E486D8511DAD8A04887E5B2352

- Arrive to a Base58 encoded bitcoin address

1PMycacnJaSqwwJqjawXBErnLsZ7RkXUAs

Private Key (Uncompressed) *WIF*

Having a private EC key

0C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D

- **Uncompressed Extended Key (x80 prefix, no suffix)**

800C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D

- **Base58 encode**

5HueCGU8rMjxEXxiPuD5BDku4MkFqeZyd4dZ1jvhTVqvbTLvyTJ

- When an address will be derived, it will be **uncompressed**

Note: the process is invertible

Private Key (Compressed) *WIF*

Using the same private EC key as in the previous slide

0C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D

- Compressed Extended Key (x80 prefix, x01 suffix)

800C28FCA386C7A227600B2FE50B7CAE11EC86D3BF1FBE471BE89827E19D72AA1D**01**

- Base58 encode

KwdMAjGmerYanjeui5SHS7JkmpZvVipYvB2LJGU1ZxJwYvP98617

- When an address will be derived, it will be compressed

Bitcoin Address Version (main net)

version	usage	leading symbol(s)
x00	pubkey hash (P2PKH address)	1
x05	script hash (P2SH address)	3
x80	prvkey (WIF, uncompressed pubkey)	5
x80	prvkey (WIF, compressed pubkey)	K or L
x0488B21E	BIP32 pubkey	xpub
x0488ADE4	BIP32 private key	xprv

Bitcoin Address Version (test net)

version	usage	leading symbol(s)
6F	pubkey hash	m or n
C4	script hash	2
EF	privkey (WIF, uncompressed pubkey)	9
EF	privkey (WIF, compressed pubkey)	c
x043587CF	BIP32 pubkey	tpub
x04358394	BIP32 private key	tprv

Homework #5

With the private key obtained with the previous homework:

1. Calculate the uncompressed WIF
2. Calculate the compressed WIF
3. Calculate the bitcoin address from uncompressed public key
4. Calculate the bitcoin address from compressed public key

Check PrivKeyToWIFCompressed.py, PrivKeyToWIFUncompressed.py, and PubKeyToBitcoinAddress.py in the scripts folder at <https://github.com/fametrano/BitcoinBlockchainTechnology>

Verify the results at www.bitaddress.org (Wallet Details)

Bibliography

- Christof Paar and Jan Pelzl, “Understanding Cryptography”, Springer, chapter 8, 9, 10, 11
- Pedro Franco, “Understanding Bitcoin”, Wiley, chapter 5, 7 “Public Key Cryptography”
- Andreas Antonopoulos, “Mastering Bitcoin”, O'Reilly, chapter 4 “Keys, Addresses, Wallets”
<https://goo.gl/tNBu5w>
- A. Narayanan et al., “Bitcoin and Cryptocurrencies Technologies”, Princeton, chapter 1
- NIST, Digital Signature Standard,
<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>

Homework #6

Please provide few paragraphs of feedback on this lesson: its strongest and weakest points, if and where it would need improvements, what had you struggling the most, what got you most excited, etc.