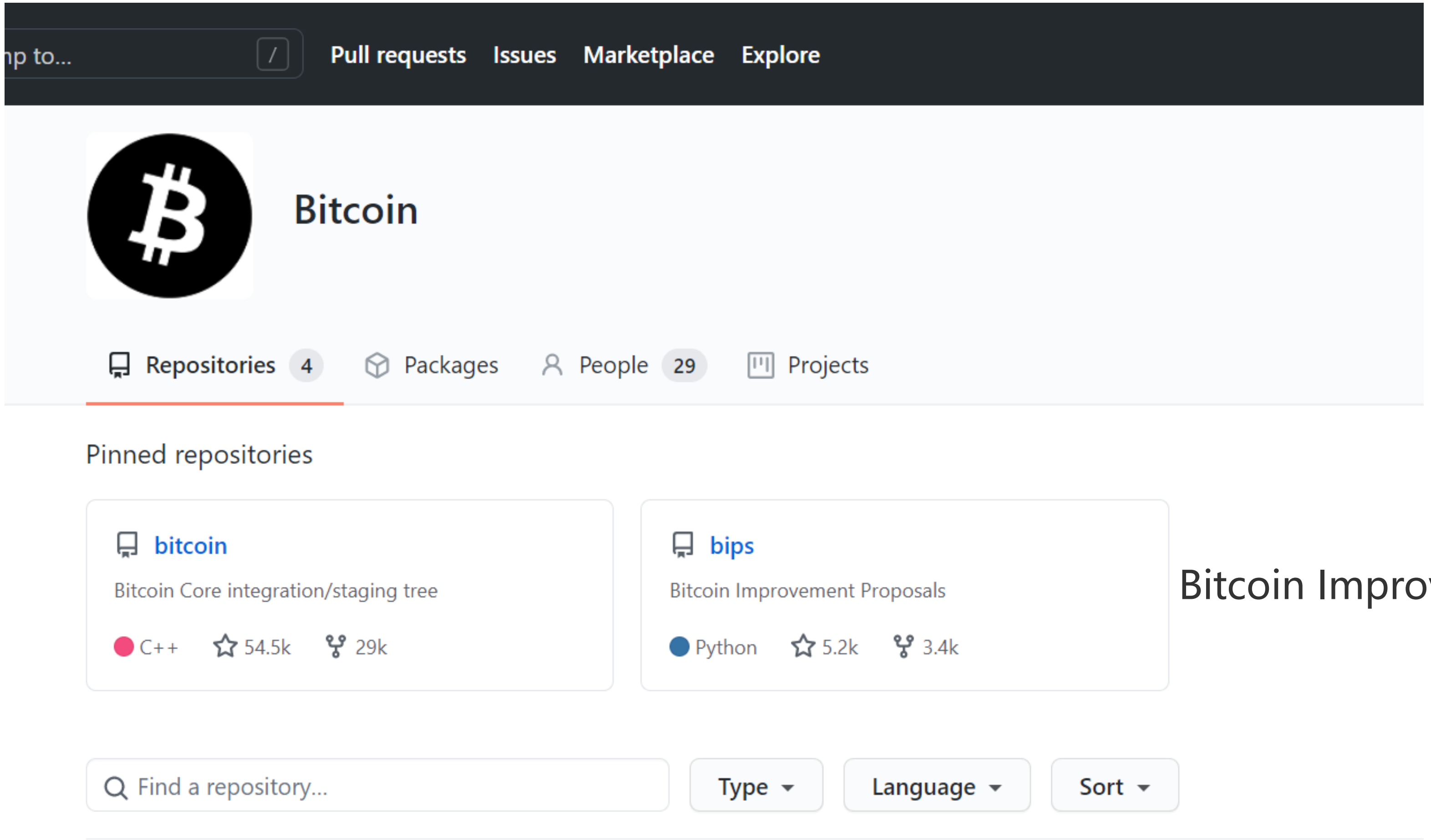
Signature

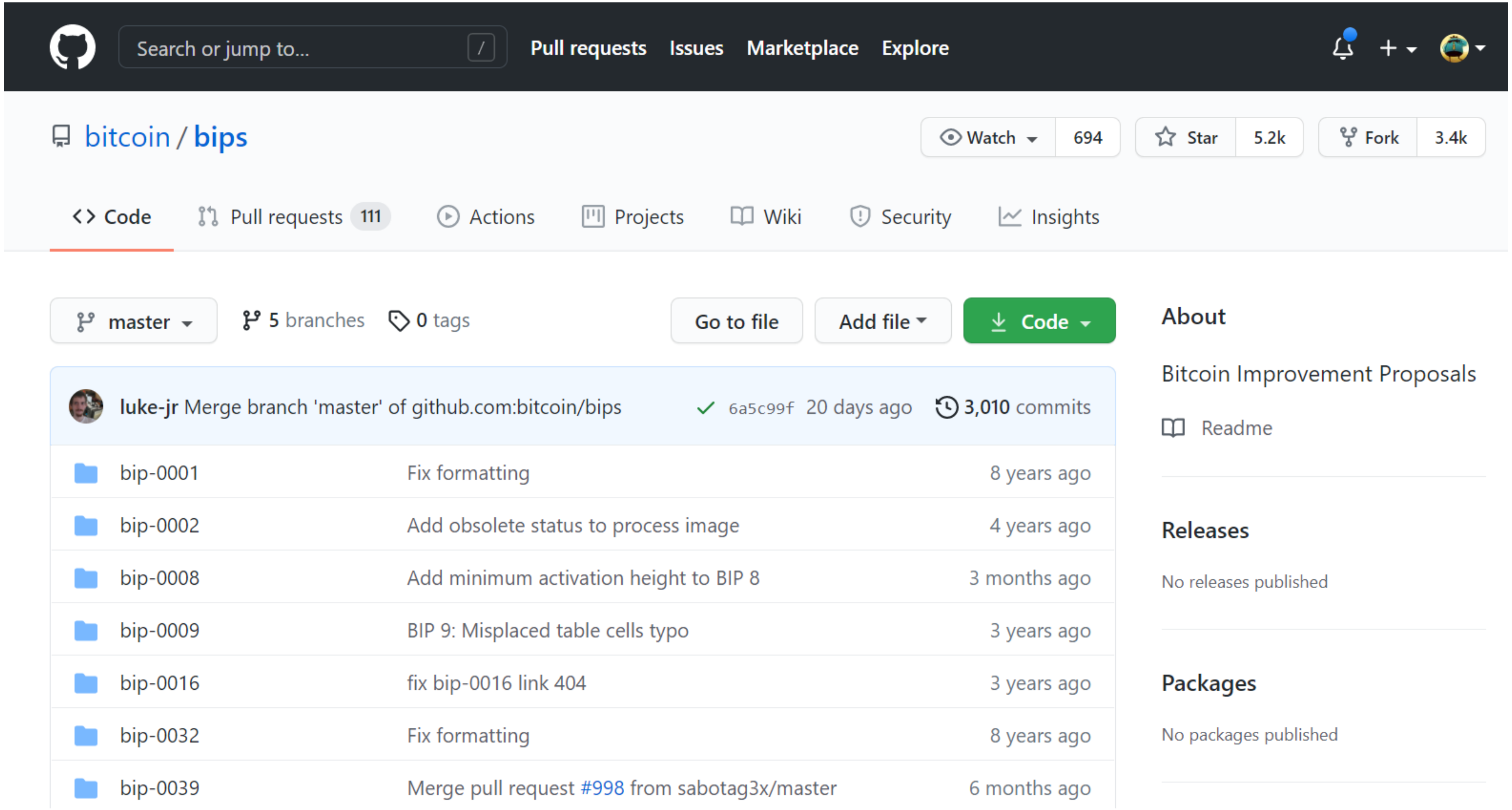
Huang

Start From BIPs



Bitcoin Improvement Proposals

Start From BIPs



Start From BIPs

bip-0300.mediawiki	BIP 300: Fix preamble	2 years ago
bip-0301.mediawiki	Fix preamble in BIP 301	2 years ago
bip-0310.mediawiki	BIP 310: Fix preamble; add to README	3 years ago
bip-0320.mediawiki	Assign BIP 320 to nVersion bits for general purpose use	3 years ago
bip-0322.mediawiki	bip-0322: remove the 'to_spend' transaction from serializ...	6 months ago
bip-0325.mediawiki	BIP 325: Remove empty section "Acknowledgement"	2 months ago
bip-0330.mediawiki	Add comments links and created date.	2 years ago
bip-0338.mediawiki	Assign BIP 338 for Disable transaction relay message	4 months ago
bip-0339.mediawiki	BIP339: clarify fetching	10 months ago
bip-0340.mediawiki	BIP340: remove batch speedup graph and link to it instead	21 days ago
bip-0341.mediawiki	Merge pull request #1104 from ajtowns/202103-bip341-...	last month
bip-0342.mediawiki	Merge pull request #1104 from ajtowns/202103-bip341-...	last month
bip-0343.mediawiki	Fix formatting for BIP 343	20 days ago
bip-0350.mediawiki	Merge pull request #1066 from SomberNight/202002_bi...	4 months ago
bip-0370.mediawiki	Fix Comments-URI for BIP 370	3 months ago

Start From BIPs

bip-0340.mediawiki	BIP340: remove batch speedup graph and link to it instead	21 days ago

```
BIP: 340
Title: Schnorr Signatures for secp256k1
Author: Pieter Wuille <pieter.wuille@gmail.com>
        Jonas Nick <jonasd.nick@gmail.com>
        Tim Ruffing <crypto@timruffing.de>
Comments-Summary: No comments yet.
Comments-URI: https://github.com/bitcoin/bips/wiki/Comments:BIP-0340
Status: Draft
Type: Standards Track
License: BSD-2-Clause
Created: 2020-01-19
Post-History: 2018-07-06: Schnorr signatures BIP
```

BIP-340

Motivation

Bitcoin has traditionally used ECDSA signatures over the secp256k1 curve with SHA256 hashes for authenticating transactions. These are standardized, but have a number of downsides compared to Schnorr signatures over the same curve

ECDSA

The **Elliptic Curve Digital Signature Algorithm** (ECDSA) offers a variant of the **Digital Signature Algorithm** (DSA) which uses elliptic curve cryptography.

The Elliptic Curve Digital Signature Algorithm (ECDSA)

Johnson, D., Menezes, A. & Vanstone, S. 2001

Cited by: 1850, [link]

ECDSA

ECDSA signature generation. To sign a message m, an entity A with domain parameters $D = (q, \mathrm{FR}, a, b, G, n, h)$ and associated key pair (d, Q) does the following:

1. Select a random or pseudorandom integer k, $1 \le k \le n-1$.
2. Compute $kG = (x_1, y_1)$ and convert x_1 to an integer $\overline{x}_1$.
3. Compute $r = x_1 \bmod n$. If $r = 0$ then go to step 1.
4. Compute $k^{-1} \bmod n$.
5. Compute SHA-1(m) and convert this bit string to an integer e.
6. Compute $s = k^{-1}(e + dr) \bmod n$. If $s = 0$ then go to step 1.
7. A's signature for the message m is (r, s).

Signer : key pair (d, Q), $dG \rightarrow Q$

Public : G, Q, m

Random pick : k

Calc : $kG \rightarrow (x, y)$, $x \rightarrow r$

Calc : SHA-1$(m) \rightarrow e$

Calc : $s = k^{-1}(e + dr)$ mod n

Signature : (r, s)

ECDSA

ECDSA signature verification. To verify A's signature (r, s) on m, B obtains an authentic copy of A's domain parameters $D = (q, \mathrm{FR}, a, b, G, n, h)$ and associated public key Q. It is recommended that B also validates D and Q (see Sects. 5.4 and 6.2). B then does the following:

1. Verify that r and s are integers in the interval $[1, n-1]$.
2. Compute SHA-1(m) and convert this bit string to an integer e.
3. Compute $w = s^{-1} \bmod n$.
4. Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$.
5. Compute $X = u_1 G + u_2 Q$.
6. If $X = \mathcal{O}$, then reject the signature. Otherwise, convert the x coordinate x_1 of X to an integer $\overline{x}_1$, and compute $v = \overline{x}_1 \bmod n$.
7. Accept the signature if and only if $v = r$.

Verifier : Signature : (r, s)

Public : G, Q, m

Calc : SHA-1$(m) \rightarrow e$

If $s = k^{-1}(e + dr)$ mod n and $kG \rightarrow (x, y)$, $x \rightarrow r$

$\Rightarrow k = s^{-1}e + s^{-1}dr$

$\Rightarrow kG = s^{-1}eG + s^{-1}drG = s^{-1}eG + s^{-1}rQ = (x_1, y_1)$

$\Rightarrow x_1 \ mod\ n = r$

Verify : $(s^{-1}eG + s^{-1}rQ)\ mod\ n =?= $ (r, _)

Schnorr

In cryptography, a **Schnorr signature** is a digital signature produced by the Schnorr signature algorithm that was described by Claus Schnorr. It is a digital signature scheme known for its simplicity, among the first whose security is based on the intractability of certain discrete logarithm problems.[1] It is efficient and generates short signatures. It was covered by U.S. Patent 4,995,082 which expired in February 2008.

Efficient Signature Generation by Smart Cards*

C.P. Schnorr. 1991

Cited by: 3339, [link]

Schnorr

The user's private und public key. A user generates by himself a private key s which is a random number in $\{1, 2, \ldots, q\}$. The corresponding public key v is the number $v = \alpha^{-s} \pmod p$.

Protocol for signature generation.

To sign message m with the private key s perform the following steps:

1. *Preprocessing* (see section 3). Pick a random number $r \in \{1, \ldots, q\}$ and compute $x := \alpha^r \pmod p$.
2. Compute $e := h(x, m) \in \{0, \ldots, 2^t - 1\}$.
3. Compute $y := r + se \pmod q$ and output the *signature* (e, y).

Signer

Key pair : (s, v), $\alpha^{-s} \rightarrow v$

Random pick : r, $\alpha^r (mod\ p) \rightarrow x$

Calc : $h(x,\ m) \rightarrow e$

Calc : r $+ se \rightarrow y$

Signature : $(e,\ y)$

Schnorr

Protocol for signature verification.

To verify the signature (e, y) for message m with public key v compute $\overline{x} = \alpha^y v^e \pmod{p}$ and check that $e = h(\overline{x}, m)$.

A signature (e, y) is accepted if it withstands verification. A signature generated according to the protocol is always accepted since we have

$$x = \alpha^r = \alpha^{r+se} v^e = \alpha^y v^e \pmod{p} \ .$$

Verifier : Signature : $(e, \ y)$

Public : α, v, m

If $\alpha^r (mod\ p) \rightarrow x$ and $\mathrm{r} + se \rightarrow y$

$$\alpha^r = \alpha^{y-se}$$

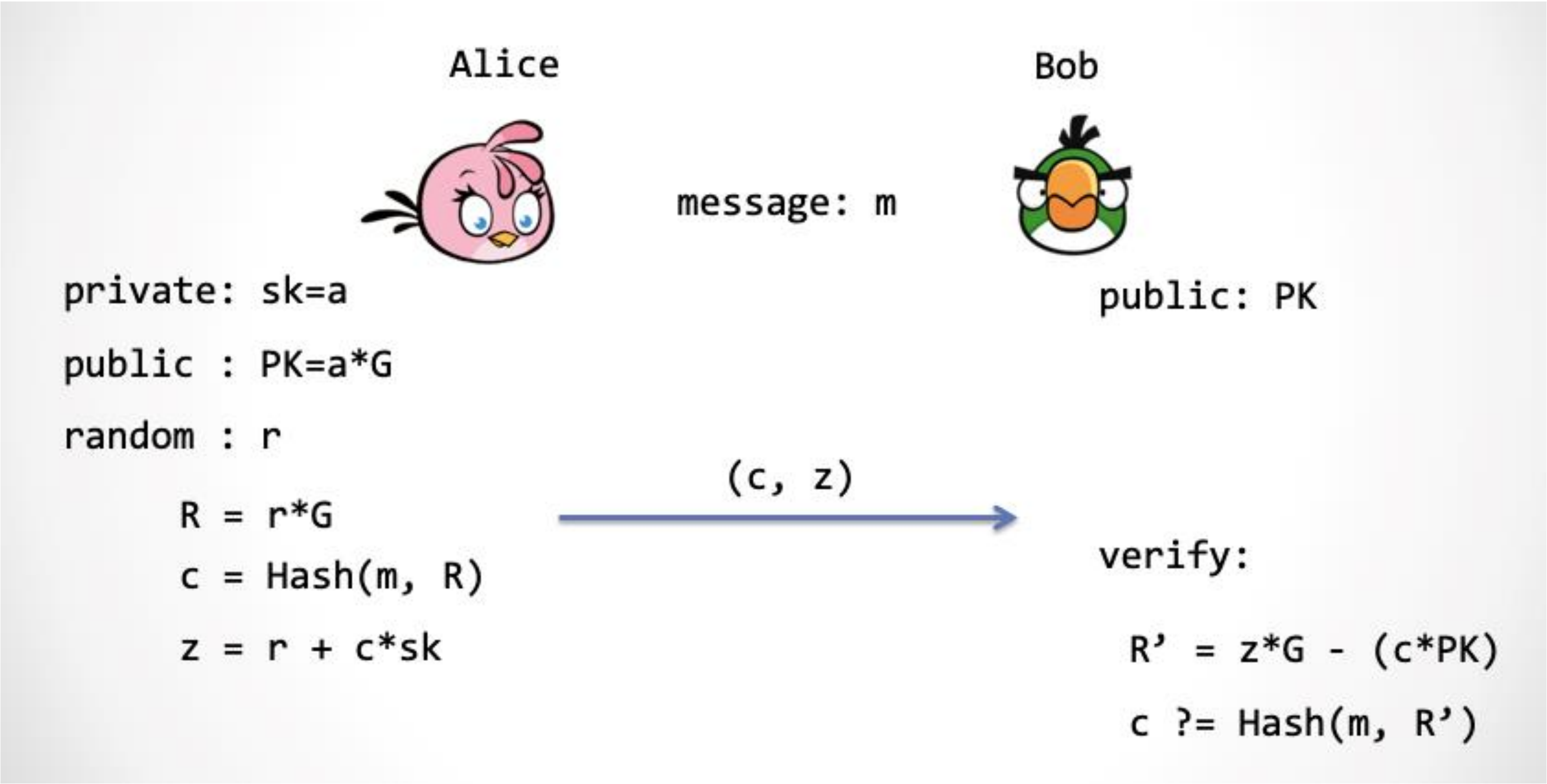
$$= \alpha^y \alpha^{-se}$$

$$= \alpha^y v^e$$

$$= x$$

Verify : $e =?= hash(x, m) = hash(\alpha^y v^e, m)$

Schnorr + Elliptic Curve



Randomness

Every time a ECDSA signature is created, signer calculate :

$s = k^{-1}(e + dr)$. keep k, d secret, but s, e, r public

We can't get two unknowns by one equation

So what if we get two equations ? (two signatures on different message)

$$\begin{cases} s_1 = {k_1}^{-1}(e_1 + dr_1) \\ s_2 = {k_2}^{-1}(e_2 + dr_2) \end{cases}$$

There are three unknowns (k_1, k_2, d) , because k is picked by signer randomly for each signature, so $k_1 \neq k_2$

Randomness

If random number are reused.

ECDSA k

1. $s_1 = k^{-1}(e_1 + dr)$
2. $s_2 = k^{-1}(e_2 + dr)$

$$\Longrightarrow \quad k = \frac{e_1 - e_2}{s_1 - s_2} \qquad d = \frac{s_1 k - e_1}{r}$$

Schnorr r

1. $z_1 = r + c_1 \times sk$
2. $z_2 = r + c_2 \times sk$

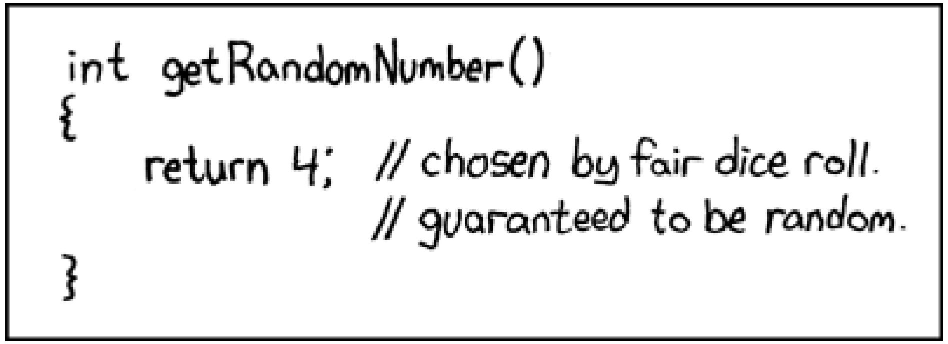
$$\Longrightarrow \quad sk = \frac{z_1 - z_2}{c_1 - c_2}$$

Randomness is important

One of Bitcoin vulnerabilities is caused by ECDSA weak randomness. A random number is not cryptographically secure, which leads to private key leakage and even fund theft.

ECDSA weak randomness in Bitcoin

Ziyu Wang, Hui Yu, Zongyang Zhang, Jiaming Piao, Jianwei Liu. 2020 [link]



Signature in Bitcoin

Signature in **Transaction**

A **transaction** is a <u>transfer of bitcoin</u>

Example : Transfer 9 bitcoins from Alice to Bob

Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

Transaction : Alice -> Bob

Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

- Prove that Alice owns at least 9 bitcoins

Transaction : Alice -> Bob

Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

- Prove that Alice owns at least 9 bitcoins

Transaction : Charlie -> Alice	$\Longrightarrow$	Transaction : Alice -> Bob

$\Uparrow$

Alice's UTXO (Unspent Transaction output)

Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

➢ Prove that Alice owns at least 9 bitcoins

Transaction : Charlie -> Alice ⟹ Transaction : Alice -> Bob

Alice's sig

⇧

Alice's UTXO (Unspent Transaction output)

Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

- Prove that Alice owns at least 9 bitcoins
- After this transaction, the 9 bitcoins can only be used by Bob

Transaction : Charlie -> Alice

Transaction : Alice -> Bob

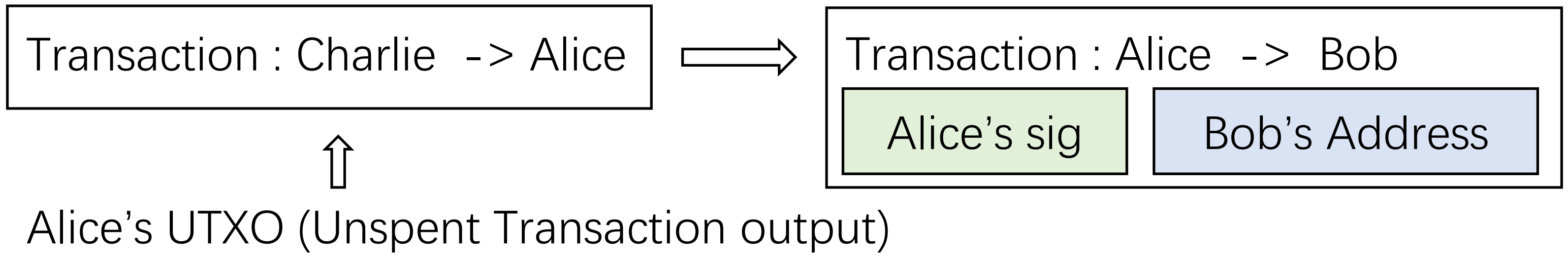
Alice's sig

Alice's UTXO (Unspent Transaction output)

Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

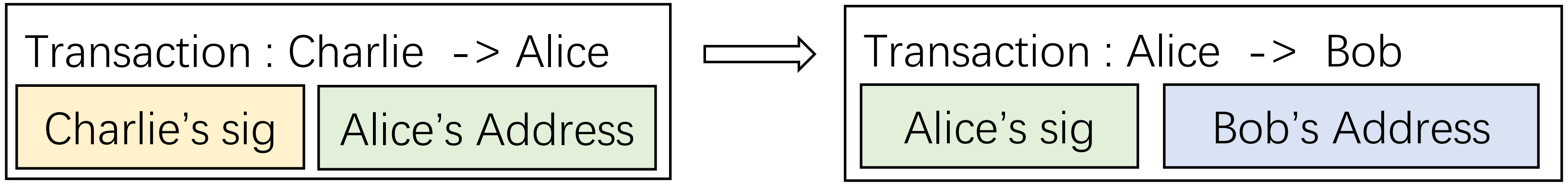
- Prove that Alice owns at least 9 bitcoins
- After this transaction, the 9 bitcoins can only be used by Bob



Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

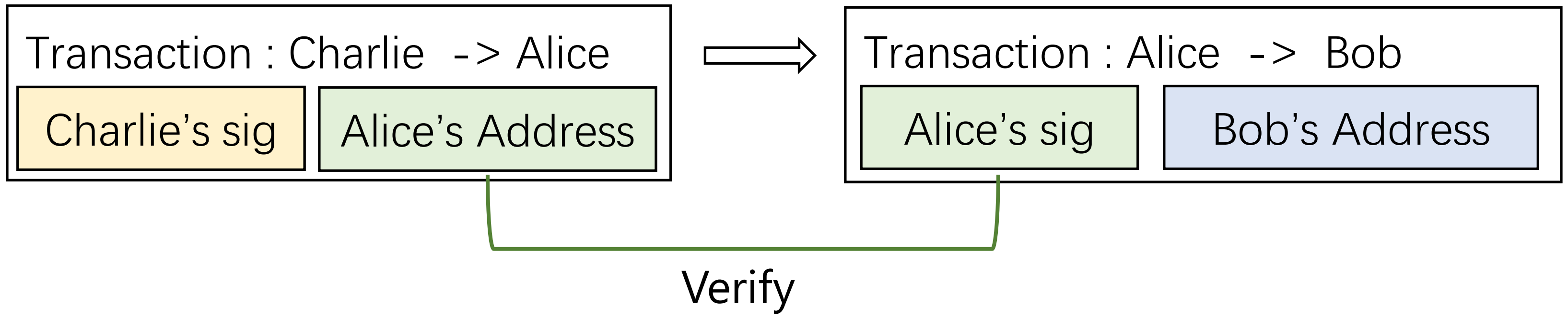
- Prove that Alice owns at least 9 bitcoins
- After this transaction, the 9 bitcoins can only be used by Bob



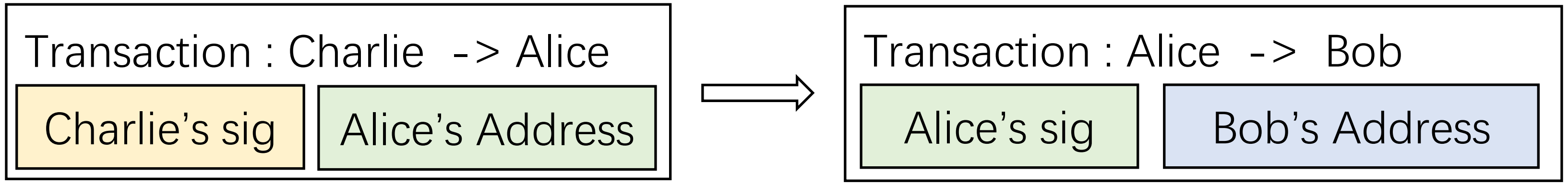
Signature in Bitcoin

Example : Transfer 9 bitcoins from Alice to Bob

- Prove that Alice owns at least 9 bitcoins
- After this transaction, the 9 bitcoins can only be used by Bob



Signature in Bitcoin



Inputs

Index	0	Details	Output
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb 5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101 03662ae5be14405447f08a037f9b5b951f1717af4ddcb9a8e5e71e81d333efb8c5		
Witness			

Outputs

Index	0	Details	Unsp...
Address	14ZCxwXGd6ung2BUw75kMbGuZSezeUihZr	Value	0.00...
Pkscript	OP_DUP OP_HASH160 26ffacf291820b281bcc4a0af5ea0e64b2d59f7e OP_EQUALVERIFY OP_CHECKSIG		

Index	1	Details	Unsp...
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		

Signature in Bitcoin

Transaction : Alice -> Bob

Alice’s sig | Bob’s Address

Inputs

Index	0	Details	Output
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb 5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101 03662ae5be14405447f08a037f9b5b951f1717af4ddcb9a8e5e71e81d333efb8c5		
Witness			

Transaction : Charlie -> Alice

Alice’s Address

Outputs

Index	0	Details	Unsp...
Address	14ZCxwXGd6ung2BUw75kMbGuZSezeUihZr	Value	0.00...
Pkscript	OP_DUP OP_HASH160 26ffacf291820b281bcc4a0af5ea0e64b2d59f7e OP_EQUALVERIFY OP_CHECKSIG		
Index	1	Details	Unsp...
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		

Signature in Bitcoin

Transaction : Alice -> Bob

Bob’s Address

Inputs

Index	0	Details	Output
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb 5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101 03662ae5be14405447f08[illegible]af4ddcb9a8e5e71e81d333efb8c5		
Witness			

Transaction : Charlie -> Alice

Alice’s Address

Alice’s sig

Outputs

Index	0	Details	Unsp...
Address	14ZCxwXGd6ung2BUw75kMbGuZSezeUihZr	Value	0.00...
Pkscript	OP_DUP OP_HASH160 26ffacf291820b281bcc4a0af5ea0e64b2d59f7e OP_EQUALVERIFY OP_CHECKSIG		
Index	1	Details	Unsp...
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		

Signature in Bitcoin

Transaction : Alice -> Bob

Inputs

Index	0	Details	Output
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb 5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101 03662ae5be14405447f08 [illegible] af4ddcb9a8e5e71e81d333efb8c5		
Witness			

Transaction : Charlie -> Alice

Alice's Address

Alice's sig

Outputs

Index	0	Details	Unsp...
Address	14ZCxwXGd6ung2BUw75kMbGuZSezeUihZr	Value	0.00...
Pkscript	OP_DUP OP_HASH160 26ffacf291820b281bcc4a0af5ea0e64b2d59f7e OP_EQUALVERIFY OP_CHECKSIG		

Bob's Address

Index	1	Details	Unsp...
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		

Signature in Bitcoin

Transaction : Alice -> Bob

Inputs

Index 0 Details Output

Address 1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa Value 0.26...

Pkscript
OP_DUP
OP_HASH160
77f4a478955a93f4d0cad1e2aabb356641b68b9f
OP_EQUALVERIFY
OP_CHECKSIG

Transaction : Charlie -> Alice

Alice’s Address

Sigscript 3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb
5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101
03662ae5be14405447f08[illegible]af4ddcb9a8e5e71e81d333efb8c5

Alice’s sig

Witness

Outputs

Index 0 Details Unsp...

Address 14ZCxwXGd6ung2BUw75kMbGuZSezeUihZr Value 0.00...

Pkscript
OP_DUP
OP_HASH160
26ffacf291820b281bcc4a0af5ea0e64b2d59f7e
OP_EQUALVERIFY
OP_CHECKSIG

Bob’s Address

Index 1 Details Unsp...

Address 1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa Value 0.26...

Pkscript
OP_DUP
OP_HASH160
77f4a478955a93f4d0cad1e2aabb356641b68b9f
OP_EQUALVERIFY
OP_CHECKSIG

Signature in Bitcoin

Inputs

Index	0	Details	Output
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb 5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101 03662ae5be14405447f08a037f9b5b951f1717af4ddcb9a8e5e71e81d333efb8c5		
Witness			

Alice’s Address

Alice’s sig

Signature in Bitcoin

Inputs

Index	0	Details	Output
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...
Pkscript	OP_DUP OP_HASH160 77f4a478955a93f4d0cad1e2aabb356641b68b9f OP_EQUALVERIFY OP_CHECKSIG		
Sigscript	3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb 5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101 03662ae5be14405447f08a037f9b5b951f1717af4ddcb9a8e5e71e81d333efb8c5		
Witness			

Alice's public address

Signature in Bitcoin

Inputs

Index	0	Details	Output	
Address	1BwGUtHirPQ4dKusbCTcxMbxBTu24yRkHa	Value	0.26...	
Pkscript	OP_DUP			
	OP_HASH160			
	77f4a478955a93f4d0cad1e2aabb356641b68b9f			Alice’s public address
	OP_EQUALVERIFY			
	OP_CHECKSIG			
Sigscript	3045022100c79badb56ec34afe3c3e46b1adddfe2a6ca681d7c27679c4eabfcff1eb3ebfb 5022031d4106bf6399c5afcc8e2161e7a5f9939aa2877be181da33fe5c4d233ffd47101			Alice’s signature
	03662ae5be14405447f08a037f9b5b951f1717af4ddcb9a8e5e71e81d333efb8c5			Alice’s public key
Witness				

Signature in Bitcoin

Pkscript	OP_DUP OP_HASH160 Alice's public address OP_EQUALVERIFY OP_CHECKSIG
Sigscript	Alice's signature Alice's public key

Signature in Bitcoin

Sigscript	Alice's signature	⇐ Unlock Script	
	Alice's public key		
Pkscript	OP_DUP		⇐ Bitcoin Script
	OP_HASH160		
	Alice's public address	⇐ Lock Script	
	OP_EQUALVERIFY		
	OP_CHECKSIG		

Signature in Bitcoin

Alice's signature

Alice's public key

OP_DUP

OP_HASH160

Alice's public address

OP_EQUALVERIFY

OP_CHECKSIG

Signature in Bitcoin

Alice’s public key

OP_DUP

OP_HASH160

Alice’s public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice’s signature

Signature in Bitcoin

OP_DUP

OP_HASH160

Alice's public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice's public key
Alice's signature

Signature in Bitcoin

OP_HASH160

Alice’s public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice’s public key	OP_DUP
Alice’s signature	

Signature in Bitcoin

OP_HASH160

Alice's public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice's public key
Alice's public key
Alice's signature

OP_DUP

Signature in Bitcoin

OP_HASH160

Alice’s public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice’s public key
Alice’s public key
Alice’s signature

Signature in Bitcoin

Alice’s public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice’s public key
Alice’s public key
Alice’s signature

OP_HASH160

Signature in Bitcoin

Alice's public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice's public key
Alice's signature

OP_HASH160 $\boxed{\text{Alice's public address}}$

Signature in Bitcoin

Alice's public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice's public address
Alice's public key
Alice's signature

Signature in Bitcoin

OP_EQUALVERIFY

OP_CHECKSIG

Alice’s public address
Alice’s public address
Alice’s public key
Alice’s signature

Signature in Bitcoin

OP_CHECKSIG

Alice’s public address
Alice’s public address
Alice’s public key
Alice’s signature

OP_EQUALVERIFY

Signature in Bitcoin

Alice's public address == Alice's public address

OP_EQUALVERIFY

OP_CHECKSIG

Alice's public key
Alice's signature

Signature in Bitcoin

OP_CHECKSIG

Alice’s public key
Alice’s signature

Signature in Bitcoin

Alice’s public key	OP_CHECKSIG
Alice’s signature	

Signature in Bitcoin

ECDSAVerify (Alice’s public key , Alice’s signature)

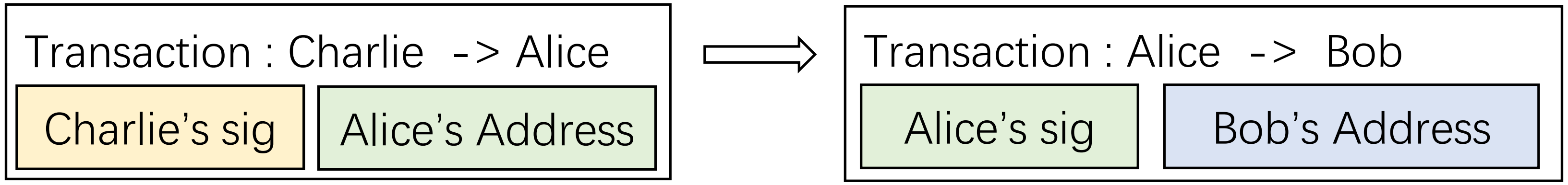
OP_CHECKSIG

Signature in Bitcoin

Finish

Signature in Bitcoin

Finish

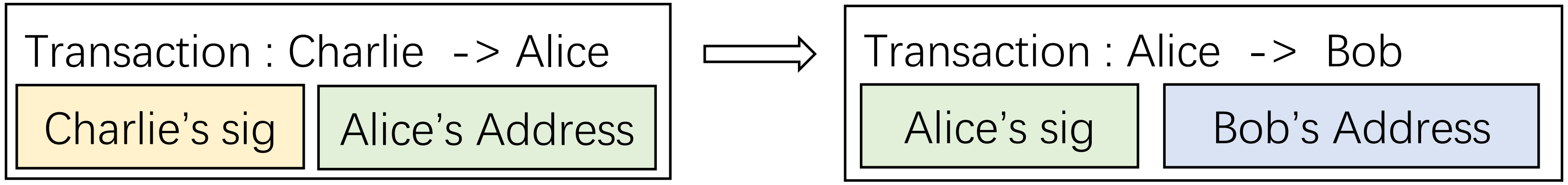


Pkscript : OP_DUP OP_HASH160 <address> OP_EQUALVERIFY OP_CHECKSIG

Sigscript : <signature> <public key>

Signature in Bitcoin

In the context of Bitcoin, standard transactions on the Bitcoin network could be called "single-signature transactions," because transfers require only one signature — from the owner of the private key associated with the Bitcoin address.



Multi-Signature

Multiple signers (each with their own private/public key) jointly sign a single message, resulting in a single signature. This single signature can then be verified by anyone who also knows the message and the public keys of the signers.

These are often referred to as m-of-n transactions.

- 1-of-2：the signature of either is sufficient to spend the funds.
- 2-of-2：both signatures are required to spend the funds
- 2-of-3：two out of three people need to verify the transaction
- …

Multi-Signature in Bitcoin

Suppose we are sending money to a company headed by 3 people (James, Alice and Curry) and two out of those three people need to verify the transaction for it to go through.

Transaction : Charlie -> Company

Charlie's sig	Company's Address

Transaction : Company -> Bob

James' sig Curry's sig	Bob's Address

- P2PKH，Pay-to-Public Key Hash： start with "1"： 14qViLJfdGaP4EeHnDyJbEGQysnCpwn1gd
- P2SH，Pay-to-Script Hash： start with "3"： 3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy

Multi-Signature in Bitcoin

Suppose we are sending money to a company headed by 3 people (James, Alice and Curry) and two out of those three people need to verify the transaction for it to go through.

Transaction : Charlie -> Company

Charlie's sig	(P2SH) Company's Address

Transaction : Company -> Bob

James' sig Curry's sig	(P2PKH) Bob's Address

- P2PKH，Pay-to-Public Key Hash： start with "1"：14qViLJfdGaP4EeHnDyJbEGQysnCpwn1gd
- P2SH，Pay-to-Script Hash： start with "3"：3J98t1WpEZ73CNmQviecrnyiWrnqRhWNLy

Multi-Signature in Bitcoin

Company’s Address : Hash of <Redeem Script>

Redeem Script : 2 <James’ public key> <Alice’s public key> <Curry’s public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Transaction : Company -> Bob

James’ sig Curry’s sig	(P2PKH) Bob’s Address

Multi-Signature in Bitcoin

Company's Address : Hash of <Redeem Script>

Redeem Script : 2 <James' public key> <Alice's public key> <Curry's public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Unlock Script : James' sig Curry's sig <Redeem Script>

Multi-Signature in Bitcoin

Company's Address : Hash of <Redeem Script>

Redeem Script : 2 <James' public key> <Alice's public key> <Curry's public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Unlock Script : James' sig Curry's sig <Redeem Script>

EQUAL
Hash of Redeem Script
HASH160
Redeem Script
Curry's sig
James' sig

Multi-Signature in Bitcoin

Company's Address : Hash of <Redeem Script>

Redeem Script : 2 <James' public key> <Alice's public key> <Curry's public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Unlock Script : James' sig Curry's sig <Redeem Script>

EQUAL

Hash of Redeem Script
HASH160
Redeem Script
Curry's sig
James' sig

Multi-Signature in Bitcoin

Company’s Address : Hash of <Redeem Script>

Redeem Script : 2 <James’ public key> <Alice’s public key> <Curry’s public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Unlock Script : James’ sig Curry’s sig <Redeem Script>

EQUAL (Hash of Redeem Script ,

HASH160
Redeem Script
Curry’s sig
James’ sig

Multi-Signature in Bitcoin

Company's Address : Hash of <Redeem Script>

Redeem Script : 2 <James' public key> <Alice's public key> <Curry's public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Unlock Script : James' sig Curry's sig <Redeem Script>

EQUAL (Hash of Redeem Script , HASH160

Redeem Script
Curry's sig
James' sig

Multi-Signature in Bitcoin

Company’s Address : Hash of <Redeem Script>

Redeem Script : 2 <James’ public key> <Alice’s public key> <Curry’s public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Unlock Script : James’ sig Curry’s sig <Redeem Script>

EQUAL (Hash of Redeem Script , HASH160 Redeem Script)

Curry’s sig
James’ sig

Multi-Signature in Bitcoin

Company's Address : Hash of <Redeem Script>

Redeem Script : 2 <James' public key> <Alice's public key> <Curry's public key> 3 CHECKMULTISIG

Lock Script : HASH160 <Hash of Redeem Script> EQUAL

Unlock Script : James' sig Curry's sig <Redeem Script>

EQUAL (Hash of Redeem Script , HASH160 Redeem Script)

CHECKMULTISIG
3
Curry's public key
Alice's public key
James' public key
2
Curry's sig
James' sig

Multi-Signature in Bitcoin

CHECKMULTISIG https://github.com/bitcoin/bitcoin/blob/master/src/script/interpreter.cpp#L1177-L1205

```
1176                 bool fSuccess = true;
1177                 while (fSuccess && nSigsCount > 0)
1178                 {
1179                     valtype& vchSig    = stacktop(-isig);
1180                     valtype& vchPubKey = stacktop(-ikey);
1181
1182                     // Note how this makes the exact order of pubkey/signature evaluation
1183                     // distinguishable by CHECKMULTISIG NOT if the STRICTENC flag is set.
1184                     // See the script_(in)valid tests for details.
1185                     if (!CheckSignatureEncoding(vchSig, flags, serror) || !CheckPubKeyEncoding(vchPubKey, flags, sigversion, serror)) {
1186                         // serror is set
1187                         return false;
1188                     }
1189
1190                     // Check signature
1191                     bool fOk = checker.CheckECDSASignature(vchSig, vchPubKey, scriptCode, sigversion);
1192
1193                     if (fOk) {
1194                         isig++;
1195                         nSigsCount--;
1196                     }
1197                     ikey++;
1198                     nKeysCount--;
1199
1200                     // If there are more signatures left than keys left,
1201                     // then too many signatures have failed. Exit early,
1202                     // without checking any further signatures.
1203                     if (nSigsCount > nKeysCount)
1204                         fSuccess = false;
1205                 }
```

3
Curry's public key
Alice's public key
James' public key
2
Curry's sig
James' sig

Multi-Signature in Bitcoin

CHECKMULTISIG https://github.com/bitcoin/bitcoin/blob/master/src/script/interpreter.cpp#L1177-L1205

```
1176                 bool fSuccess = true;
1177                 while (fSuccess && nSigsCount > 0)
1178                 {
1179                     valtype& vchSig    = stacktop(-isig);
1180                     valtype& vchPubKey = stacktop(-ikey);
1181 
1182                     // Note how this makes the exact order of pubkey/signature evaluation
1183                     // distinguishable by CHECKMULTISIG NOT if the STRICTENC flag is set.
1184                     // See the script_(in)valid tests for details.
1185                     if (!CheckSignatureEncoding(vchSig, flags, serror) || !CheckPubKeyEncoding(vchPubKey, flags, sigversion, serror)) {
1186                         // serror is set
1187                         return false;
1188                     }
1189 
1190                     // Check signature
1191                     bool fOk = checker.CheckECDSASignature(vchSig, vchPubKey, scriptCode, sigversion);
1192 
1193                     if (fOk) {
1194                         isig++;
1195                         nSigsCount--;
1196                     }
1197                     ikey++;
1198                     nKeysCount--;
1199 
1200                     // If there are more signatures left than keys left,
1201                     // then too many signatures have failed. Exit early,
1202                     // without checking any further signatures.
1203                     if (nSigsCount > nKeysCount)
1204                         fSuccess = false;
1205                 }
```

3
Curry's public key
Alice's public key
James' public key
2
Curry's sig
James' sig

Multi-Signature in Bitcoin

CHECKMULTISIG https://github.com/bitcoin/bitcoin/blob/master/src/script/interpreter.cpp#L1177-L1205

```
        bool fSuccess = true;
        while (fSuccess && nSigsCount > 0)
        {
            valtype& vchSig    = stacktop(-isig);
            valtype& vchPubKey = stacktop(-ikey);

            // Note how this makes the exact order of pubkey/signature evaluation
            // distinguishable by CHECKMULTISIG NOT if the STRICTENC flag is set.
            // See the script_(in)valid tests for details.
            if (!CheckSignatureEncoding(vchSig, flags, serror) || !CheckPubKeyEncoding(vchPubKey, flags, sigversion, serror)) {
                // serror is set
                return false;
            }

            // Check signature
            bool fOk = checker.CheckECDSASignature(vchSig, vchPubKey, scriptCode, sigversion);

            if (fOk) {
                isig++;
                nSigsCount--;
            }
            ikey++;
            nKeysCount--;

            // If there are more signatures left than keys left,
            // then too many signatures have failed. Exit early,
            // without checking any further signatures.
            if (nSigsCount > nKeysCount)
                fSuccess = false;
        }
```

signatures must be placed in the signature script using the same order as their corresponding public keys were placed in the pubkey script or redeem script.

3
Curry's public key
Alice's public key
James' public key
2
Curry's sig
James' sig

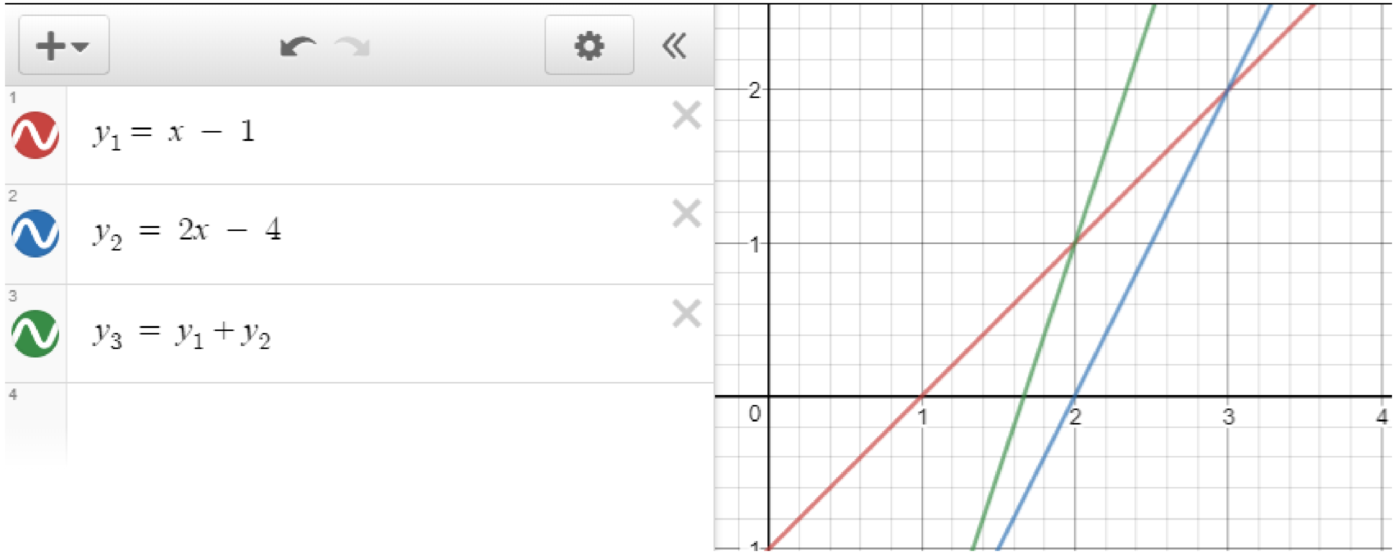
Multi-Signature in Bitcoin

Multi-Signature in bitcoin is implemented as check multiple single signatures

Multi-Signature with Schnorr

The main difference between Schnorr signatures and Bitcoin current signatures (ECDSA) is that Schnorr signatures are **Linear**, Schnorr allows native multi-signature

The most relevant property of **linearity** for our purpose, is that when you add two (or more) Schnorr signatures together, the result is a valid Schnorr signature too!



Multi-Signature with Schnorr

User : secret key : x， random pick : r

Calc : $R = r \times G$, $P = x \times G$

Calc : $s = r + \text{hash}(P, R, msg) \cdot x$

Signature : (s, R)

Verify : $s \times G =?= r \times G + \text{hash}(P, R, msg) \cdot x \times G$

Multi-Signature with Schnorr

User : secret key : x， random pick : r

Calc : $R = r \times G$, $P = x \times G$

Calc : $s = r + \text{hash}(P, R, msg) \cdot x$

Signature : (s, R)

Verify : $s \times G =?= r \times G + \text{hash}(P, R, msg) \cdot x \times G$

User1 : secret key : $x1$， random pick : $r1$

User2 : secret key : $x2$， random pick : $r2$

Multi-Signature with Schnorr

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G,\ P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G,\ P2 = x2 \times G$

Calc : $R = R1 + R2,\ P = P1 + P2$

Calc : $s1 = r1 + \mathrm{hash}(P, R, msg) \cdot x1$

Calc : $s2 = r2 + \mathrm{hash}(P, R, msg) \cdot x2$

Verify1 : $s1 \times G =?= r1 \times G + \mathrm{hash}(P, R, msg) \cdot x1 \times G$

Verify2 : $s2 \times G =?= r2 \times G + \mathrm{hash}(P, R, msg) \cdot x2 \times G$

Multi-Signature with Schnorr

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G,\ P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G,\ P2 = x2 \times G$

Calc : $R = R1 + R2,\ P = P1 + P2$

Calc : $s1 = r1 + \mathrm{hash}(P, R, msg) \cdot x1$

Calc : $s2 = r2 + \mathrm{hash}(P, R, msg) \cdot x2$

Verify1 : $s1 \times G =?= r1 \times G + \mathrm{hash}(P, R, msg) \cdot x1 \times G$

Verify2 : $s2 \times G =?= r2 \times G + \mathrm{hash}(P, R, msg) \cdot x2 \times G$

Verify : $(s1+s2) \times G =?= (r1+r2) \times G + \mathrm{hash}(P, R, msg) \cdot (x1+x2) \times G$

Multi-Signature with Schnorr

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G,\ P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G,\ P2 = x2 \times G$

Calc : $R = R1 + R2,\ P = P1 + P2$

Calc : $s1 = r1 + \text{hash}(P, R, msg) \cdot x1$

Calc : $s2 = r2 + \text{hash}(P, R, msg) \cdot x2$

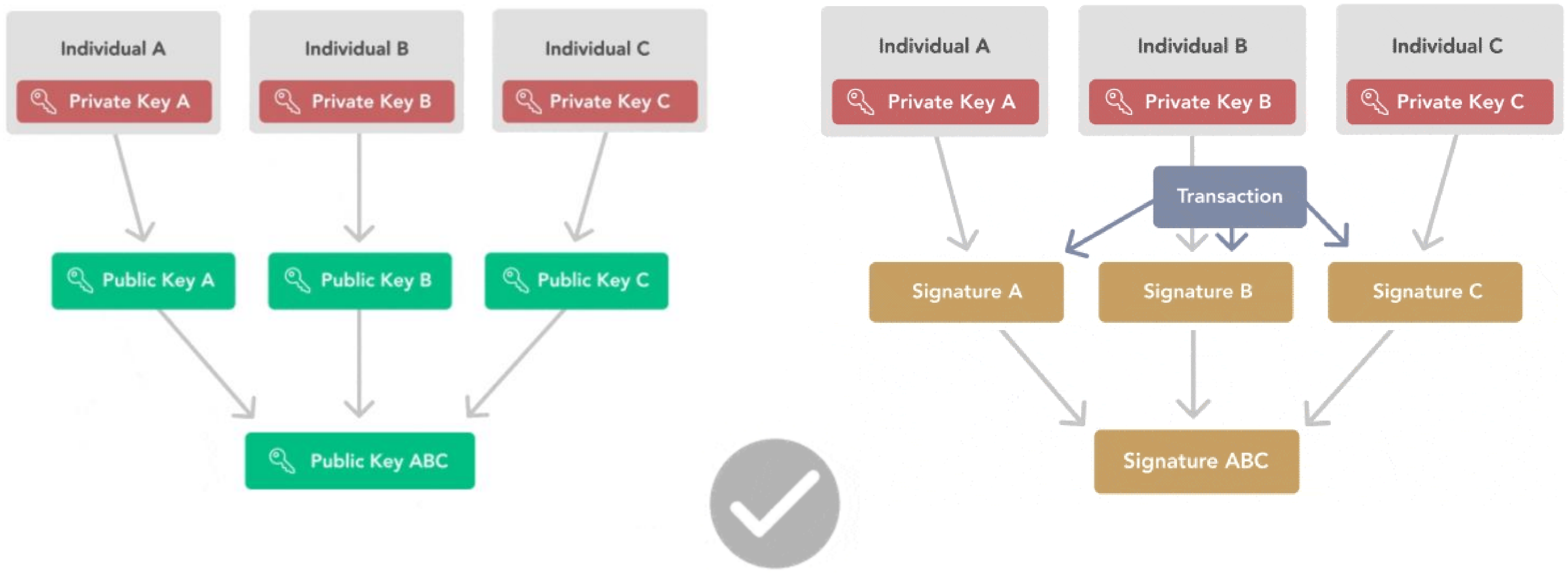
Calc : $s = s1 + s2$

Verify : $(s1+s2) \times G =?= (r1+r2) \times G + \text{hash}(P, R, msg) \cdot (x1+x2) \times G$

Signature : (s, R)

Verify : $s \times G =?= R + \text{hash}(P, R, msg) \cdot P$

Multi-Signature with Schnorr



There is no way to know whether that signature is from one individual or many individuals.

It looks the same. This feature is known as **signature aggregation**.

Multi-Signature with Schnorr(Bug)

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G$, $P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G$, $P2 = x2 \times G$

Calc : $R = R1 + R2$, $P = P1 + P2$

Calc : $s1 = r1 + hash(P, R, msg) \cdot x1$

Calc : $s2 = r2 + hash(P, R, msg) \cdot x2$

Calc : $s = s1 + s2$

Signature : (s, R)

Verify : $s \times G =?= R + hash(P, R, msg) \cdot P$

Multi-Signature with Schnorr(Bug)

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G, P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G, P2 = x2 \times G$

Claim : $R2' = R2 - R1, P2' = P2 - P1$

Calc : $R = R1 + R2$, $P = P1 + P2$

Calc : $s1 = r1 + hash(P, R, msg) \cdot x1$

Calc : $s2 = r2 + hash(P, R, msg) \cdot x2$

Calc : $s = s1 + s2$

Signature : (s, R)

Verify : $s \times G =?= R + hash(P, R, msg) \cdot P$

Multi-Signature with Schnorr(Bug)

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G,\ P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G,\ P2 = x2 \times G$

Claim : $R2' = R2 - R1,\ P2' = P2 - P1$

Calc : $R = R1 + R2' = R1 + R2 - R1 = R2$

$P = P1 + P2' = P1 + P2 - P1 = P2$

Calc : $s1 = r1 + hash(P, R, msg) \cdot x1$

Calc : $s2 = r2 + hash(P, R, msg) \cdot x2$

Calc : $s = s1 + s2$

Signature : (s, R)

Verify : $s \times G =?= R + hash(P, R, msg) \cdot P$

Multi-Signature with Schnorr(Bug)

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G,\ P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G,\ P2 = x2 \times G$

Claim : $R2' = R2 - R1,\ P2' = P2 - P1$

Calc : $R = R1 + R2' = R1 + R2 - R1 = R2$

$P = P1 + P2' = P1 + P2 - P1 = P2$

Calc : $s1 = r1 + \text{hash}(P, R, msg) \cdot x1$

Calc : $s2 = r2 + \text{hash}(P, R, msg) \cdot x2$

Signature : $(s2, R)$

Verify : $s \times G =?= R + \text{hash}(P, R, msg) \cdot P$

Multi-Signature with Schnorr(Bug)

User1 :

secret key : $x1$

random pick : $r1$

Calc : $R1 = r1 \times G,\ P1 = x1 \times G$

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G,\ P2 = x2 \times G$

Claim : $R2' = R2 - R1,\ P2' = P2 - P1$

Calc : $R = R1 + R2' = R1 + R2 - R1 = R2$

$P = P1 + P2' = P1 + P2 - P1 = P2$

Calc : $s1 = r1 + \text{hash}(P, R, msg) \cdot x1$

Calc : $s2 = r2 + \text{hash}(P, R, msg) \cdot x2$

Signature : $(s2, R)$ *always valid*

Verify : $s \times G =?= R + \text{hash}(P, R, msg) \cdot P$ $\Longrightarrow$

$$
\begin{aligned}
& R2 + \text{hash}(P1 + P2', R, msg) \cdot (P1 + P2') \\
&= R2 + \text{hash}(P2, R, msg) \cdot P2 \\
&= s2 \times G
\end{aligned}
$$

Multi-Signature with Schnorr(Bug)

~~User1 :~~

~~secret key : $x1$~~

~~random pick : $r1$~~

~~Calc : $R1 = r1 \times G,\ P1 = x1 \times G$~~

User2 :

secret key : $x2$

random pick : $r2$

Calc : $R2 = r2 \times G,\ P2 = x2 \times G$

Claim : $R2' = R2 - R1,\ P2' = P2 - P1$

Calc : $R = R1 + R2' = R1 + R2 - R1 = R2$

$P = P1 + P2' = P1 + P2 - P1 = P2$

~~Calc : $s1 = r1 + hash(P, R, msg) \cdot x1$~~

Calc : $s2 = r2 + hash(P, R, msg) \cdot x2$

Signature : $(s2, R)$ *always valid*

User2 successfully forged the aggregate signature!
User2 can clearly sign for this by himself.

Multi-Signature - MuSig

The attack described above is called a rogue-key attack, and one way to avoid it is requiring that User1 and User2 prove first that they actually possess the private keys corresponding to their claimed public keys.

Simple Schnorr Multi-Signatures with Applications to Bitcoin

Gregory Maxwell, Andrew Poelstra, Yannick Seurin, and Pieter Wuille 2018

Cited by: 130, [link]

Multi-Signature - MuSig

Signing. Let X_1 and x_1 be the public and private key of a specific signer, let m be the message to sign, let $X_2, \ldots, X_n$ be the public keys of other cosigners, and let $L = \{X_1, \ldots, X_n\}$ be the multiset of all public keys involved in the signing process.[7] For $i \in \{1, \ldots, n\}$, the signer computes

$$a_i = H_{\mathrm{agg}}(L, X_i) \tag{1}$$

and then the "aggregated" public key $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$. Then, the signer generates a random $r_1 \leftarrow_{\$} \mathbb{Z}_p$, computes $R_1 = g^{r_1}$, $t_1 = H_{\mathrm{com}}(R_1)$, and sends t_1 to all other cosigners. Upon reception of commitments $t_2, \ldots, t_n$ from other cosigners, it sends R_1. Upon reception of $R_2, \ldots, R_n$ from other cosigners, it checks that $t_i = H_{\mathrm{com}}(R_i)$ for all $i \in \{2, \ldots, n\}$ and aborts the protocol if this is not the case; otherwise, it computes

$$R = \prod_{i=1}^{n} R_i,$$
$$c = H_{\mathrm{sig}}(\widetilde{X}, R, m),$$
$$s_1 = r_1 + c a_1 x_1 \bmod p,$$

and sends s_1 to all other cosigners. Finally, upon reception of $s_2, \ldots, s_n$ from other cosigners, the signer can compute $s = \sum_{i=1}^{n} s_i \bmod p$. The signature is $\sigma = (R, s)$.

User1	User2
x1 -> P1	x2 -> P2

L = Hash(P1, P2)

H(L, P1) · P1 r1 -> R1	H(L, P2) · P2 r2 -> R2

P = H(L, P1)P1 + H(L, P2)P2
R = R1 + R2

c1 = H(P, R, msg) · H(L, P1) s1 = r1 + c1 · x1	c2 = H(P, R, msg) · H(L, P2) s2 = r2 + c2 · x2

s = s1 + s2
Signature : (R, s)

Multi-Signature - MuSig

Signer's view

HPR = H(P, R, msg) ; HLPi = H(L, Pi)

Verify1 : $s1 \times G = r1 \times G + HPR \cdot HLP1 \cdot x1 \times G$

Verify2 : $s2 \times G = r2 \times G + HPR \cdot HLP2 \cdot x2 \times G$

Verify1 + Verify2

linear

$\Rightarrow$ $(s1+s2) \times G = (r1+r2) \times G + HPR \cdot (HLP1 \cdot x1 + HLP2 \cdot x2\ ) \times G$

$\Rightarrow$ $s \times G =?= R + H(P, R, msg) \cdot (HLP1 \cdot P1 + HLP2 \cdot P2\ )$

$\Rightarrow$ $\underline{s} \times G =?= \underline{R} + H(P, R, msg) \cdot P$

Signature : (R, s)

User1	User2
x1 -> P1	x2 -> P2

L = Hash(P1, P2)

User1	User2
$H(L, P1) \cdot P1$ r1 -> R1	$H(L, P2) \cdot P2$ r2 -> R2

P = H(L, P1)P1 + H(L, P2)P2

R = R1 + R2

User1	User2
$c1 = H(P, R, msg) \cdot H(L, P1)$ $s1 = r1 + c1 \cdot x1$	$c2 = H(P, R, msg) \cdot H(L, P2)$ $s2 = r2 + c2 \cdot x2$

s = s1 + s2

Signature : (R, s)

Multi-Signature - MuSig

Verifier's view

Verification. Given a multiset of public keys $L = \{X_1, \ldots, X_n\}$, a message m, and a signature $\sigma = (R, s)$, the verifier computes $a_i = H_{\mathrm{agg}}(L, X_i)$ for $i \in \{1, \ldots, n\}$, $\widetilde{X} = \prod_{i=1}^{n} X_i^{a_i}$, $c = H_{\mathrm{sig}}(\widetilde{X}, R, m)$ and accepts the signature if $g^s = R \prod_{i=1}^{n} X_i^{a_i c} = R\widetilde{X}^c$.

User1

x1 -> P1

User2

x2 -> P2

L = Hash(P1, P2)

H(L, P1) · P1
r1 -> R1

H(L, P2) · P2
r2 -> R2

P = H(L, P1)P1 + H(L, P2)P2
R = R1 + R2

Signature : (R, s)

Public : msg, G, P1, P2, H(), Hash()

Calc : L = Hash(P1, P2) P = H(L, P1) · P1 + H(L, P2) · P2

Verify : s × G =?= R + H(P, R, msg) · P

Multi-Signature - MuSig

Attacker's view (rogue-key attack)

The aggregated public key would be $P = \prod_{i=1}^{n} P_i$.

Attacker reveals his key as $P_n(\prod_{i=1}^{n-1} P_i)^{-1}$, resulting in an aggregated key $P = P_n$, which the last signerclearly can forge signatures for.

Signature : (R, s)

Public：msg, G, P1, P2, H(), Hash()

Calc : L = Hash(P1, P2) P = H(L, P1) · P1 + H(L, P2) · P2

To make the aggregated public key P = H(L, P2) P2. *(notice: revealed public key P2 ≠ real key P2)*

The attacker need to solve : $H(L, P_2)P_2 = H(L, P_1)P_1 + H(L, P_2)P_2$ *hard to solve*

Multi-Signature - MuSig

Attacker's view (rogue-key attack)

User1, User2, aggregated public key

Naive Schnorr $\underline{P} = P1 + P2$

Calc : $L = \mathrm{Hash}(P1, P2)$ $\underline{P} = H(L, P1) \cdot P1 + H(L, P2) \cdot P2$

All we had to do was define $\underline{P}$ not as a simple sum of the individual public keys $\underline{P_i}$, but as a sum of multiples of those keys, where the multiplication factor depends on a hash of all participating keys.

Conclusion

No Conclusion

End With BIPs

BIP-340

Bitcoin has traditionally used ECDSA signatures over the secp256k1 curve with SHA256 hashes for authenticating transactions. These are standardized, but have a number of downsides compared to Schnorr signatures over the same curve

1. **Provable security**: Schnorr signatures are provably secure.
2. **Non-malleability**: The SUF-CMA security of Schnorr signatures implies that they are non-malleable.
3. **Linearity**: Schnorr signatures provide a simple and efficient method that enables multiple collaborating parties to produce a signature that is valid for the sum of their public keys.

End With BIPs

Along with BIP 341 and BIP 342, BIP 340 is an integral part of the Taproot upgrade, which is in the process of being activated.

340		Schnorr Signatures for secp256k1	Pieter Wuille, Jonas Nick, Tim Ruffing	Standard	Draft
341	Consensus (soft fork)	Taproot: SegWit version 1 spending rules	Pieter Wuille, Jonas Nick, Anthony Towns	Standard	Draft
342	Consensus (soft fork)	Validation of Taproot Scripts	Pieter Wuille, Jonas Nick, Anthony Towns	Standard	Draft

References

1. https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki
2. Johnson, D., Menezes, A. & Vanstone, S. The Elliptic Curve Digital Signature Algorithm (ECDSA). *IJIS* **1,** 36–63 (2001). https://doi.org/10.1007/s102070100002
3. Schnorr, C.P. Efficient signature generation by smart cards. *J. Cryptology* **4,** 161–174 (1991). https://doi.org/10.1007/BF00196725
4. Maxwell, G., Poelstra, A., Seurin, Y. *et al.* Simple Schnorr multi-signatures with applications to Bitcoin. *Des. Codes Cryptogr.* **87,** 2139–2164 (2019). https://doi.org/10.1007/s10623-019-00608-x
5. https://medium.com/@blairlmarshall/how-does-ecdsa-work-in-bitcoin-7819d201a3ec
6. https://medium.com/@blairlmarshall/signature-verification-multi-signatures-19886fafe97b
7. https://en.bitcoin.it/wiki/Multi-signature
8. https://en.wikipedia.org/wiki/Schnorr_signature
9. https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm
10. Key Aggregation for Schnorr Signatures https://blockstream.com/2018/01/23/en-musig-key-aggregation-schnorr-signatures/
11. Send coins to a 2-of-3 multisig, then spend them. https://gist.github.com/gavinandresen/3966071
12. 为什么 Schnorr 签名被誉为比特币 Segwit 后的最大技术更新 https://linux.cn/article-12797-1.html
13. A brief intro to Bitcoin Schnorr Multi-signatures https://hackernoon.com/a-brief-intro-to-bitcoin-schnorr-multi-signatures-b9ef052374c5
14. The Best Step-by-Step Bitcoin Script Guide Part 1. https://blockgeeks.com/guides/best-bitcoin-script-guide/
15. The Best Step-by-Step Bitcoin Script Guide Part 2 https://blockgeeks.com/guides/bitcoin-script-guide-part-2/
16. … …

More

1. Attacks
2. Bitcoin Scripts
3. MAST && Taproot
4. EDDSA

Thanks