Orchard

The Orchard Transaction

[

1837×1332 520 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/da118f9e148cbe03ca546d1dbe6087c6bd1a7179.jpeg)

Relationship between the Orchard parameters

[

2673×2044 540 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/3822de5af45cd182fdf8956f3529ca0e0987edb1.png)

Notes

Sapling

Orchard

rcm/rseed

rcm

is generated randomly[1]

rseed = rcm

rseed

is generated randomly

rcm = PRF_{rseed}(5, \rho)

note plaintext[2]

(leadByte, d, v, memo)

[1:1]

(leadByte, d,v,

rseed,

memo)

[3]

note

(d, pk_d, v, rcm)

(d, pk_d, v,

\rho, \psi

, rcm)

\rho

computed only for positioned notes

, i.e. notes that have a position in the commitment tree, \rho = H(cm, pos) can be computed for any notes: \rho = nf^{old}

from the same Action description (or \rho = []

if no spent notes in the Action description)

note commitment

cm = Commit_{rcm}(g_d, pk_d, v)

cm = Commit_{rcm}(g_d, pk_d, v,

\rho, \psi)

derive nf

nf = PRF_{nk}(\rho)

\rho

recipient’s nk

nf = [PRF_{nk}(\rho) + \psi]*K + cm, K

is a point on the Pallas curve\rho,

\psi = PRF_{rseed}(9, \rho),

cm,

recipient’s nk

spend

disclose nf

ZKP of \rho, ak,

nsk

spend auth signature (PoK of ask

)

disclose nf

ZKP of \rho, ak,

nk

spend auth signature (PoK of ask

)

leadByte

0x01[1:2]

or 0x02[3:1]

0x02

Curves

Sapling

Orchard

Application circuit EC (key agreement, signatures, etc)

Jubjub

Pallas

Proof system EC

BLS12-381

Vesta

RedDSA (SpendAuthSig, BindingSig)

RedJubjub

RedPallas

Misc

Sapling

Orchard

Circuit

Spend/Output

Action: to allow a specific action, set the corresponding flag (enableSpends, enableOutputs)

MerkleCRH

PedersenHash

SinsemillaHash

NoteCommit[4]

PedersenCommit

SinsemillaCommit

Derive ivk

Blake2s

SinsemillaShortCommit

PRF_{nk}(\rho)

(used to compute nullifiers)

Blake2s

PoseidonHash

PRP (used to generate diversifiers)

-

FF1-AES256

Proving system

Groth16

Halo 2

Address encoding

Bech32

Bech32m

Keys (see Sapling keys)

id

name

full name

type

lifetime

description

1.

sk

spending key

scalar

address

A private key associated with a shielded address

Authorizes spending of a note

Generated randomly

Used to generate other keys → enough to perform any action

Nullifier

id

name

full name

type

lifetime

description

2.

nsk

nullifier private key

scalar

-

Doesn't exist in Orchard

3.

nk

nullifier deriving key

scalar

address

nk = PRF_{sk}(7)

Used to derive note nullifiers: nf = [PRF_{nk}(\rho) + \psi]*K+ cm,

K

is a point on the Pallas curve

Spend authorization signature

id

name

full name

type

lifetime

description

4.

ask

spend authorizing key

scalar

address

Used to derive ak

, rsk

ask = PRF_{sk}(6)

5.

ak

spend validating key

point

address

ak = [ask]*P_{\mathbb{G}}, P_{\mathbb{G}}

is a subgroup generator

Used to derive dk, ovk, ivk

Private input to the Action proof: check that rk

is a randomization of ak

(spend authority

condition)

6.

rsk

-

scalar

transaction

Used to sign the hash of a transaction (proof of spend authority)

Randomization of ask

, rsk = ask + \alpha

, \alpha

is a randomness

7.

rk

validating key

point

transaction

Used to validate SpendAuthSig

$rk = [rsk]P_G = [ask + \alpha]P_G = ak + \alpha*P_\mathbb{G}$

, $P_\mathbb{G}$

is a group generator

Public input to the Action proof

Binding signature

id

name

full name

type

lifetime

description

8.

bsk

binding signing key

scalar

transaction

Used to sign the transaction hash

Computed from value commitment randomnesses rcv_i

9.

bvk

binding validating key

point

transaction

Used to validate the BindingSig

Not encoded in the transaction explicitly, must be recalculated

Computed from value commitments cv_i

$bvk = [bsk]*R$

, R

is a generator (it is not how the key is computed in practice, but the relationship should be checked by the signer)

$bvk = ValueCommit_{bsk}(0)$

Encryption

id

name

full name

type

lifetime

description

10.

rivk

ivk commitment randomness

scalar

address

rivk = PRF_{sk}(8)

Used to derive dk and ovk

Used as a randomness in ivk computation

11.

ivk

-

point

address

Used to derive a diversified key pk_d

ivk = Commit_{rivk}(ak, nk)

12.

ovk

outgoing viewing key

scalar

address

Encryption/decryption of outgoing notes

ovk = PRF_{rivk}(ak, nk)[-l_{ovk}/8:]

[5]

13.

dk

diversifier key

scalar

address

dk = PRF_{rivk}(ak, nk)[:l_{dk}/8]

[6]

Used to derive diversified address (d, pk_d

)

14.

pk_d

diversified transmission key

point

note<

Used to derive a note encryption key

Is a part of a diversified (shielded) payment address (d, pk_d)

pk_d = [ivk] * g_d = [ivk]* H(d),

d = PRP_{dk}(idx)

The diversified payment address derived from idx = 0

is called the default diversified payment address

15.

esk

ephemeral secret key

scalar

note

esk = PRF_{rseed}(4)

Used to derive K_{enc}

16.

epk

ephemeral public key

point

note

epk = [esk]*g_d

[7]

Used to derive K_{enc}

17.

ock

outgoing cipher key

scalar

note

Symmetric encryption key used to encrypt C_{enc}

(pk_d

and esk

)[8]

ock = PRF_{ovk}(cv, cm, epk)

18.

K_{enc}

-

scalar

note

Symmetric encryption key used to encrypt np

K_{enc}= KDF([esk]*pk_d, epk)

19.

(nk, ivk)

receiving key

-

address

Allows scanning of the blockchain for incoming notes and decrypt them

20.

fvk (ak, nk, rivk)

full viewing key

-

address

Enough to both encrypt & decrypt notes (to derive the corresponding keys), but not enough to spend a note

Can be used to derive incoming viewing key, outgoing viewing key, and a set of diversified addresses

21.

(dk, ivk)

incoming viewing key

-

address

Can be used to decrypt a note

dk

is required because it is used to compute g_d

(pk_d

is a part the decryption output)

Encrypt(np

, pk_d

, ovk

):

1. Generate esk
2. epk = [esk]*g_d
3. K_{enc} = KDF([esk]*pk_d, epk)
4. $C_{enc} = E_{K_{enc}}(np)$

5. ock = PRF_{ovk}(cv, cm, epk)
6. C_{out} = E_{ock}(pk_d || esk)

(if ovk

is None

, ovk

is generated randomly, C_{out}

is garbage encrypted on garbage → not used for decryption)

→ ct = (epk, C_{enc}, C_{out})

Decrypt

If the user has ivk

, they decrypt the note directly deriving K_{enc}

from ivk

:

1. K_{enc} = KDF([ivk]*epk, epk)

[7:1]

1. np = D_{K_{enc}}(C_{enc})
2. pk_d = [ivk]*g_d

If a user has the full viewing key

(though we only use the ovk

component of it), they use it to decrypt the keys C_{out}

and then use the decrypted keys to decrypt the note

1. ock = PRF_{ovk}(cv, cm, epk)

(cv

and cm

are parts of the Output description)

1. pk_d, esk = D_{ock}(C_{out})
2. K_{enc} = KDF([esk]*pk_d, epk)
3. np = D_{K_{enc}}(C_{enc})

Misc

id

name

full name

type

description

22.

(ask, nsk, ovk)

expanded spending key

-

Enough to spend a note

23.

(ask, nsk, ovk, pk_d, c)

extended spending key

-

ZIP-32

24.

(ak, nsk)

proof authorizing key

-

As a part of the spending action, one has to prove the knowledge of (\rho, ak, nsk)

and disclose the nullifier

[

2144×860 173 KB

](https://europe1.discourse-cdn.com/standard20/uploads/anoma1/original/1X/f6bd8336fd031f31c7bce11fc4df0e04d0a79bb3.jpeg)

1. Pre-Canopy ↩ ↩ ↩

2. The data needed to spend a note↩

3. Canopy onward ↩ ↩

4. ValueCommit uses PedersenHash in both Sapling and Orchard↩

5. last l_{ovk}/8

bytes (python slice style)↩

1. first l_{dk}/8

bytes (python slice style)↩

1. [esk] * pk_d = [esk] * [ivk] * g_d = [ivk] * epk

↩ ↩

1. If the receiver doesn't have ivk

, they use ovk

to decrypt the keys used to encrypt the note plaintext, and then decrypts the note plaintext. If the receiver has ivk

, they can decrypt the note plaintext directly↩