RGB Technology Guide

Scalable & confidential Bitcoin/LN smart contracts built with client-side validation paradigm

LNP/BP Standards Association
Prepared & supervised by **Dr Maxim Orlovsky**, **Pandora Core AG**
Created with support from **Bitfinex**, **Fulgur Ventures** and **Poseidon Group**

RGB is:

- Smart-contract system that is able to manage rich state
- Uses client-side validation paradigm by Peter Todd:
  the data are held by a “state owner” (like asset owner) and not by public consensus
- Operates on top of Bitcoin transaction graph, either from
  - Bitcoin blockchain, or
  - Lightning channel (or any other kind of state channel)
- Can be scripted with Turing-complete formally-verified Simplicity scripting language by Blockstream (once its released)

RGB properties

- Confidentiality
- Safety
- Scalability
- No bitcoin blockchain congestion
- Future-ready without hardforks

RGB properties, part I

- Confidentiality
  - Data is known only to owners, not the whole world
  - Amounts are confidential with Pedersen commitments and Bulletproofs, combining best from Liquid & Grin
  - Merklization and partial data reveal keeps a lot of past history private even from future owners
  - No RGB-specific data can be extracted from Bitcoin blockchain or Lightning channel transactions
- Safety
  - State isolation: state is isolated and contracts can interact only through special protocols (Spectrum) inside channels
  - Formal verification: contract properties can be proven with formal models

RGB properties, part II

- Scalability
  - Not limited by blockchain scalability:
    works on top of Lightning and any other channel
  - Amount of data kept by clients for full validation are significantly lower that in case of blockchain-based smart contract systems
  - Smart-contract-level sharding: multiple contracts keep independent history
- No congestion
  - Transactions keep only homomorphic commitments which require no additional storage
- Future-ready: Taproot, Schnorr, eltoo, multi-party LN channels, DLCs,…

What’s possible with RGB?

- Fungible assets & securities (options, futures)
  - Centrally or federation-issued
  - Issued anonymously or publicly
  - With possible secondary issuance, demurrage, inflation, ...
- Different forms of bearer rights (voting etc)
- Non-fungible assets (like tokenized art or game collectibles)
- Decentralized digital identity & key management

RGB Advantages

- over Liquid Confidential Assets:
  - Works with Lightning Network
  - Large Borromean signatures range proofs -> modern Bulletproofs
  - No blockchain space consumption!
  - Universal smart contract system
  - Works on Bitcoin mainnet, does not require federation

- over OMNI Layer (+Cointerparty, Colored coins):
  - No blockchain consumption
  - Much higher privacy
  - Works with LN without it’s modifications

- over Ethereum, EOS, ... “corporate blockchain”s:
  - Not a blockchain!
  - Works on and with Bitcoin: the only censorship-resistant unconfiscatable hard money

RGB Architecture

Paradigm-based approach

- Layer isolation via abstraction
- Layer interaction via strictly-defined interfaces
- No future hardforks, just a single release

Paradigms

- Strict encoding: LNPBP-6
- Single-use seals: LNPBP-7
- Cryptographic commitments: LNPBP-8
- Client-side verification: LNPBP-9

Can be found in
github:/LNP-BP/rust-lnpbp/tree/refactor-structure/src/paradigms

RGB is this paradigms applied to Bitcoin

- Strict encoding: RGB consensus encoding (LNPBP-10; lnpbp::rgb::*)
- Single-use seals: transaction output-based seals (LNPBP-11; lnpbp::bp::txo_seals)
- Cryptographic commitments:
  - deterministic bitcoin commitments (LNPBP-1, -2, -3; lnpbp::bp::dbc)
  - multi-contract commitments (LNPBP-4; lnpbp::lnpbps::lnpbp4)
- Client-side verification:
  - RGB schema (LNPBP-12; lnpbp::rgb::schema)
  - RGB contracts (LNPBP-13; lnpbp::rgb::{transition, stash, ancor, consignment})

Paradigms

Bitcoin tx graph

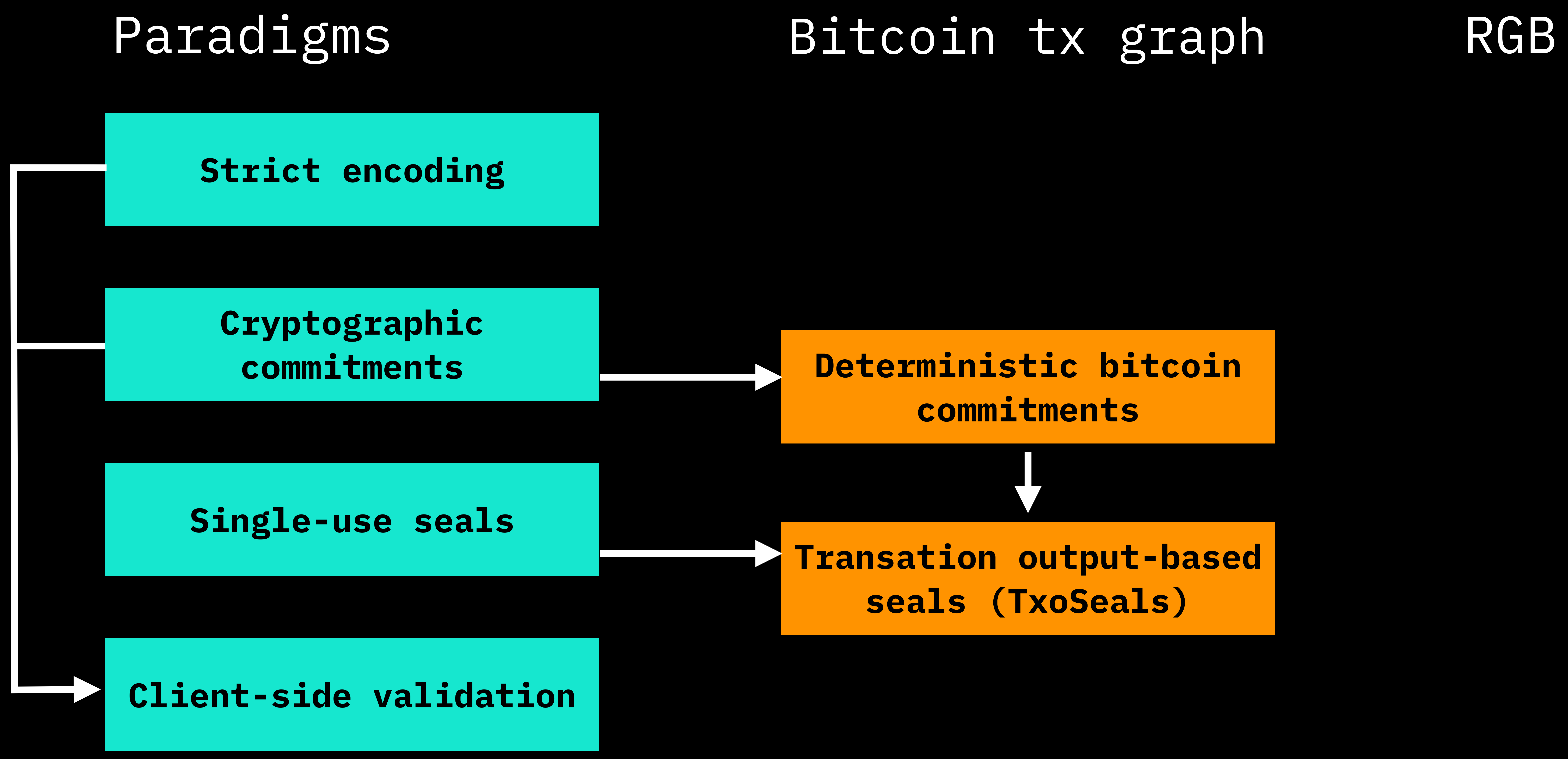
RGB

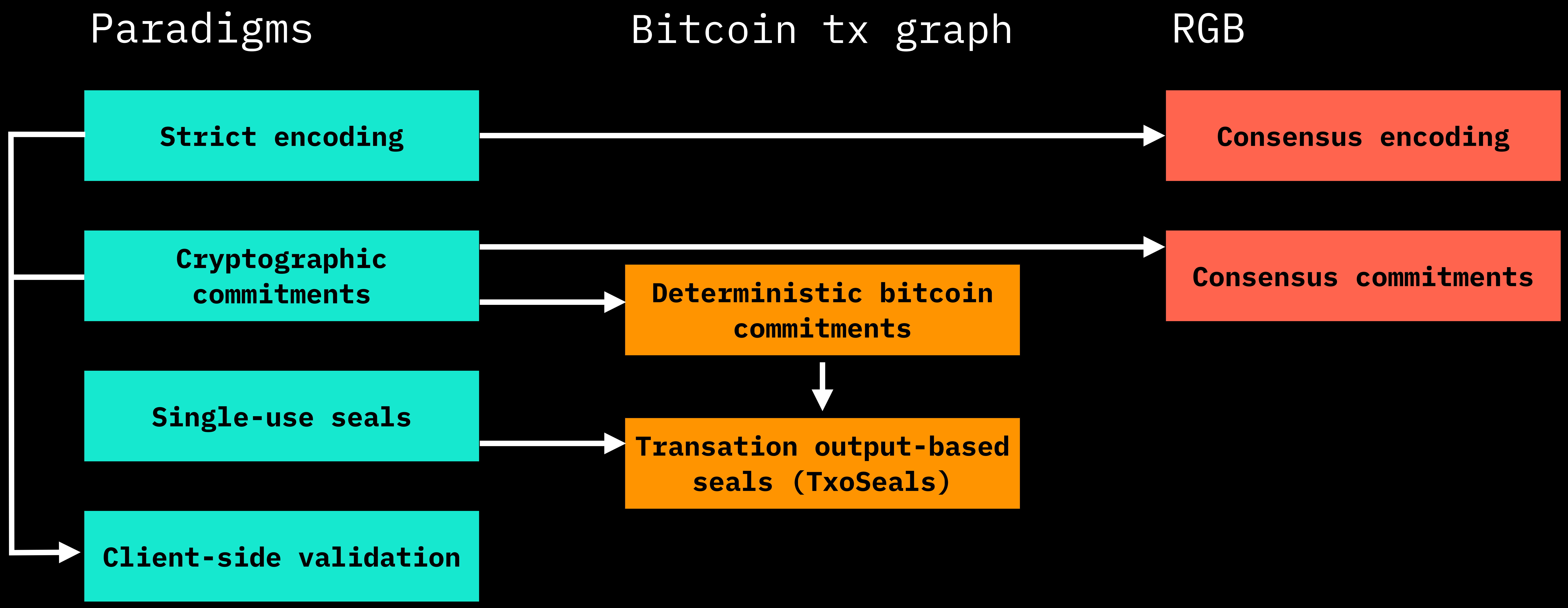
Strict encoding

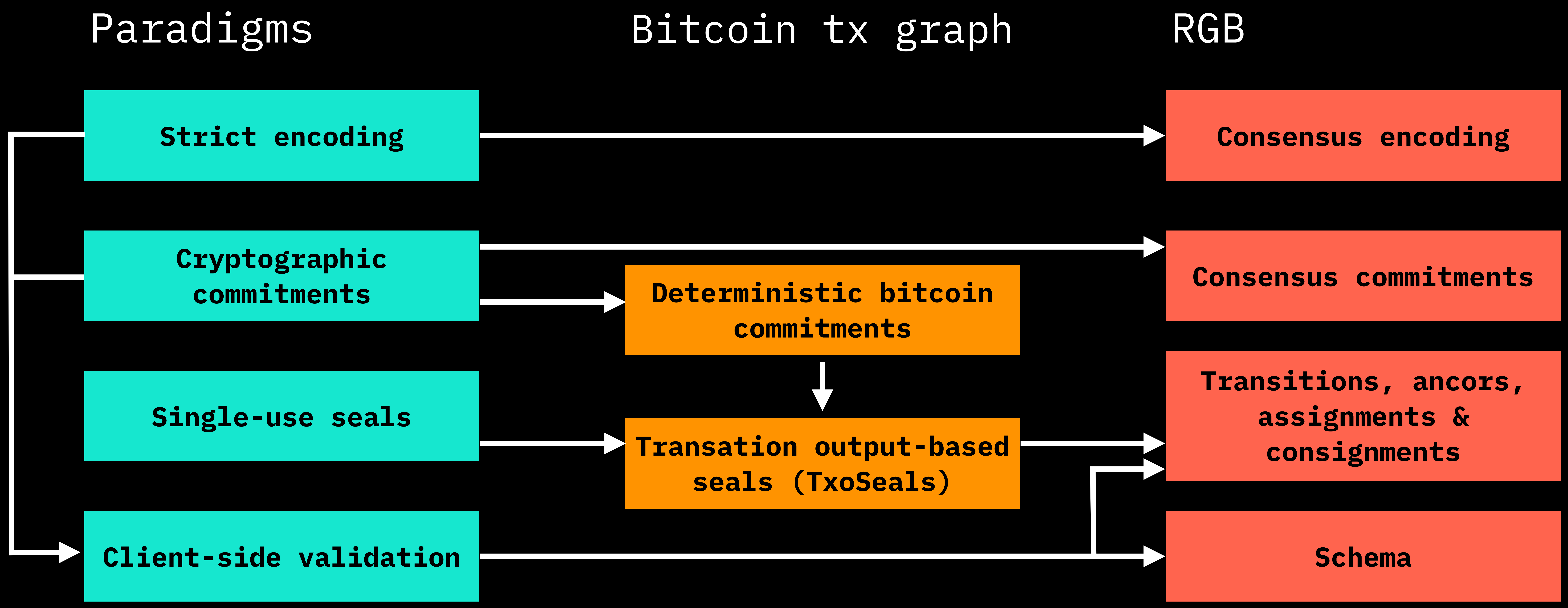
Cryptographic commitments

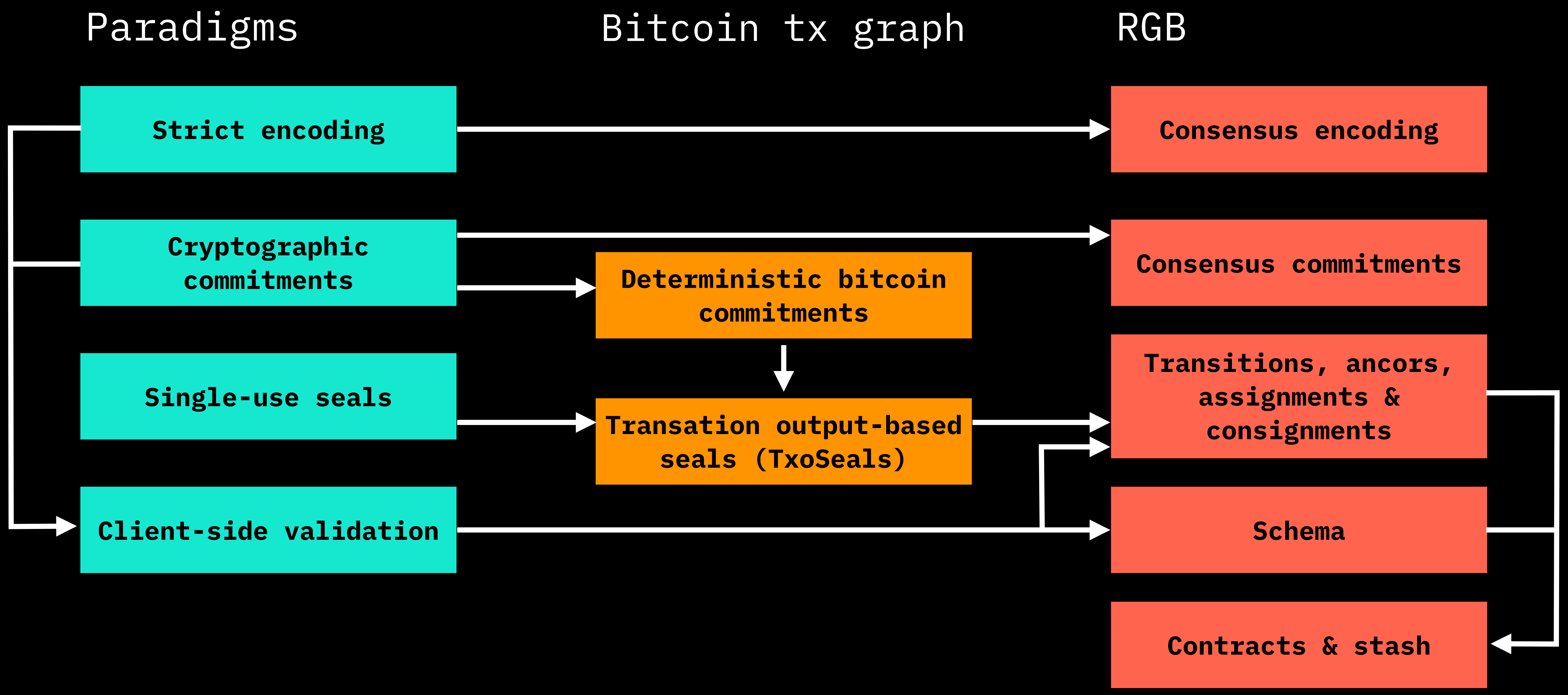
Single-use seals

Client-side validation


Paradigms
Bitcoin tx graph
RGB
Strict encoding
Cryptographic commitments
Single-use seals
Client-side validation
Deterministic bitcoin commitments
Transation output-based seals (TxoSeals)


Paradigms
Bitcoin tx graph
RGB
Strict encoding
Consensus encoding
Cryptographic commitments
Consensus commitments
Deterministic bitcoin commitments
Single-use seals
Transation output-based seals (TxoSeals)
Client-side validation


Paradigms
Bitcoin tx graph
RGB
Strict encoding
Consensus encoding
Cryptographic commitments
Consensus commitments
Deterministic bitcoin commitments
Transitions, ancors, assignments & consignments
Single-use seals
Transation output-based seals (TxoSeals)
Client-side validation
Schema


Paradigms
Bitcoin tx graph
RGB
Strict encoding
Consensus encoding
Cryptographic commitments
Consensus commitments
Deterministic bitcoin commitments
Single-use seals
Transations output-based seals (TxoSeals)
Transitions, ancors, assignments & consignments
Client-side validation
Schema
Contracts & stash

RGB Schema

- “Blueprints”/standards for constructing RGB contracts may think as of “ERC* of RGB”
- “Fungible asset” or “collectible” is a schema
- Issuer defines issuance contract, but for being supported by wallets/exchanges it must stick to (“validate against”) particular schema
- Actual wallets or exchanges will always use schema-based libraries (like “RGB fungible assets”, “RGB collectibles”), and not complex & universal core RGB library

Libraries for RGB

- **LNP/BP Core Library:** *LNP-BP/rust-lnpbp*
  Common components covering LNP/BP Standards for BP, LNP and RGB
  - Low-level
  - Conservative (consensus-critical, every change is a hard fork)
  - WASM/FFI bindings are not required

- **RGB Standard Contracts:** *rgb-org/rust-rgb*
  Implementations of main types of RGB contracts (fungible assets, collectibles, identity)
  - Mid-level
  - Conservative, but not consensus-critical
  - WASM/FFI provided
  - Can be used in developing advanced wallet functionality

- **RGB Wallet Library:** *rgb-org/rust-rgb-wallet*
  Implementations of main types of RGB contracts (fungible assets, collectibles, identity)
  - High-level
  - Not conservative
  - WASM/FFI provided
  - Easy wallet development

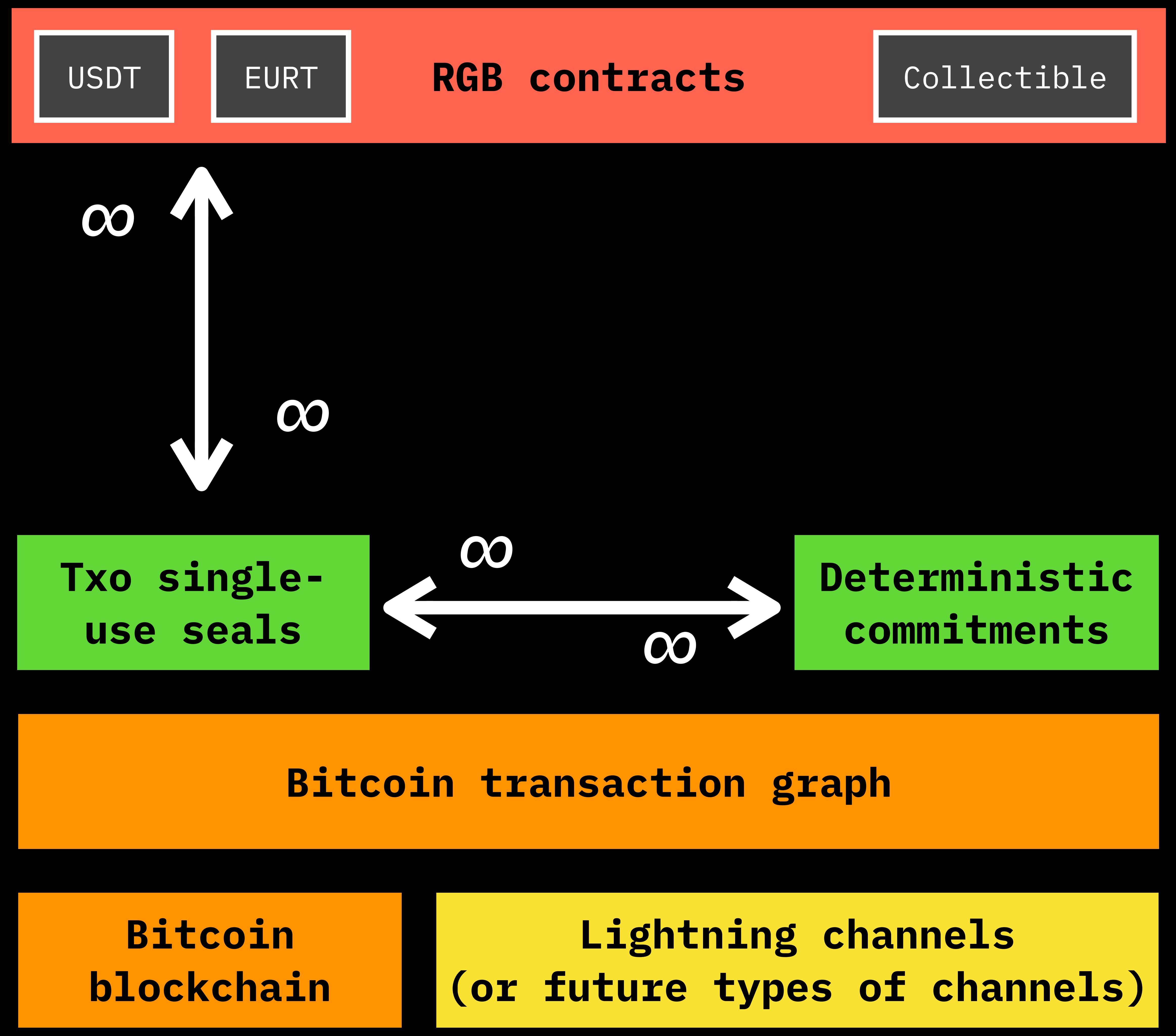
… and SDKs:

- **RGB SDK:** *rgb-org/rgb-sdk*
  - JavaScript, Swift & Kotlin libs based on WASM/FFI
  - Token issuance tools
  - Command-line wallet (kaleidoscope)

- **RGB Wallet SDK:** *rgb-org/rgb-wallet-sdk*
  - JavaScript, Swift & Kotlin libs based on WASM/FFI
  - Standard storage providers
  - Self-hosted/easy-deploy servers required for RG operations

RGB & Bitcoin multidimensional relations

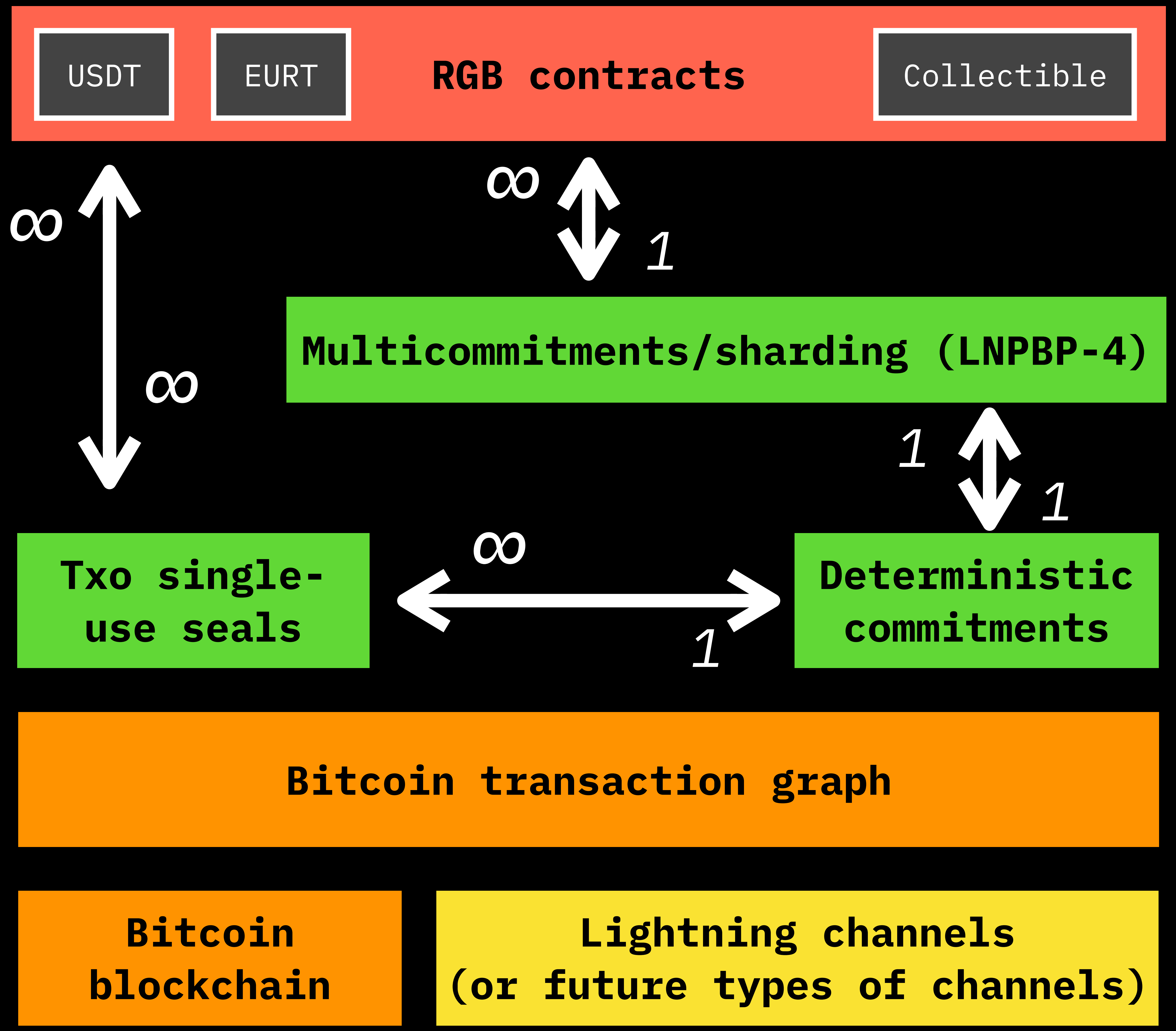
- There may be many RGB contracts issuing many different tokens, identity, collectibles…
- Each asset can be allocated to multiple transaction outputs owned by the same party
- Many different assets may be allocated to the same output
- Some asset may be allocated to the same output many times under different transfer operations...



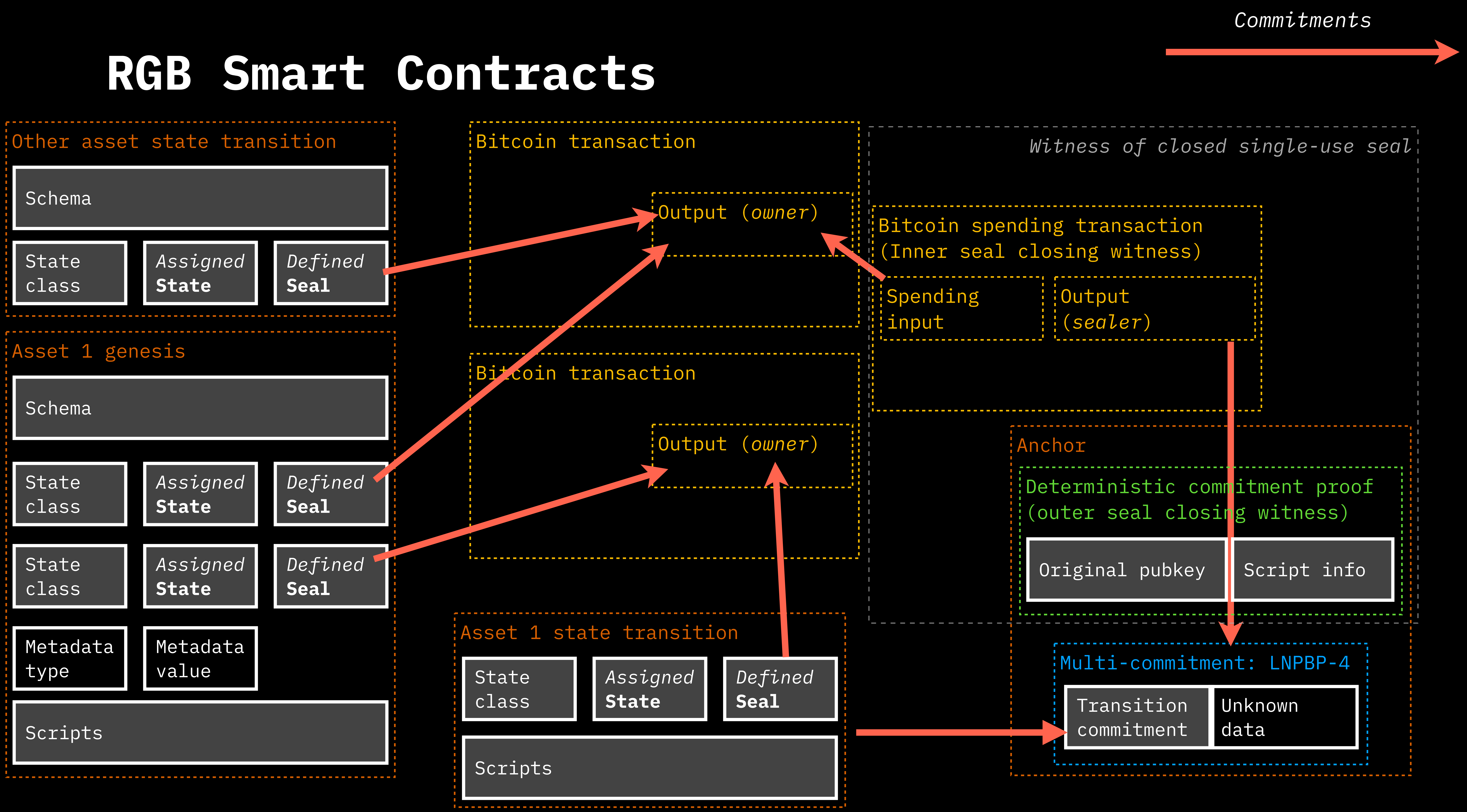
RGB & Bitcoin multidimensional relations

- Contract sharding:
  - Isolates histories of different contract without the risk of double-spending

- Requires introduction of
  - “Anchors”, linking many transitions to the same single commitment, closing some set of seals over multiple messages
  - “Stash”: a combination of all contracts with their histories and inter-contract anchors kept by an owner (wallet)



RGB Smart Contracts



Solving the complexity

Eating large cake one piece at a time:

- Deterministic bitcoin commitments:
  zero-footprint commitments within bitcoin transactions
- Single-use seals and their application to bitcoin transactions
- Client-side validation, trust & security model

DBC: Deterministic bitcoin commitments

Zero-footprint provably unique commitments embedded into transactions or their components

Deterministic Bitcoin Commitments

- Defined in **LNPBP-1**, **LNPBP-2** and **LNPBP-3** standards
  - Commitments in public key: LNPBP-1
  - Commitments in Bitcoin script: LNPBP-2
  - Commitments in transaction output (trivial)
  - Commitments in transaction: LNPBP-3
- Based on **commit-embed-verify scheme**
  paradigms::commit_verify module of *LNP/BP Core Library*
- Implemented in bp::dbc module of *LNP/BP Core Library*
- Rely on bp::scripts module for automating work with complex bitcoin scripts

Properties

- **Provably unique:** you can embed only a single commitment within (sub)transaction data
- **Hiding:** the original message can't be restored from the transactional data
- **Zero-footprint:** commitment does not increase the size of transaction or its components and does not introduces new components into the transaction
- **Client-side verified:** creation and verification of the commitment require extra-transactional information

DBC Terms & Definitions

- **Container:** data structure providing all necessary information to hold the commitment
  - **Host:** target transaction or its component for embedding commitment
  - **External data:** additional extra-transaction information required for proper embedding (like fee amount or full script source). Can be deconstructed into or reconstructed from
    - **Proof:** an important part of the external data that must be persisted for verification procedure
    - **Supplement:** re-computible part of the external data
- **Commitment:** resulting transaction or it's part containing the embedded commitment to the message

Extra-transaction information (external data)

Persistent (Proof)

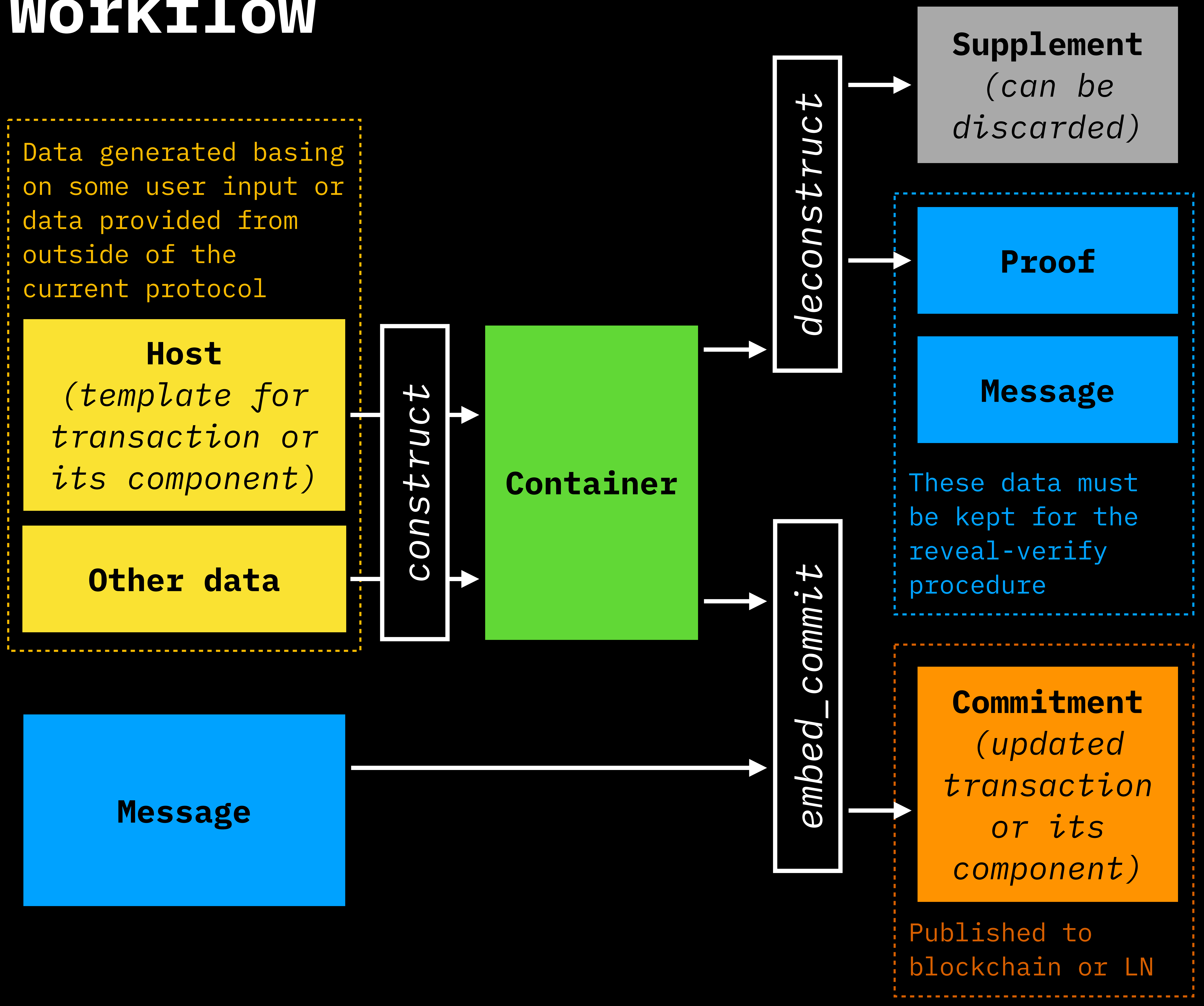
- Must be created before the commitment by wallet or other software using DBCs
- Have to be persisted; otherwise commitment verification will be impossible even if the message was revealed
- Consists of:
  - Original value of the public key
  - Script Info structure consisting of either
    - script for non-trivial script-based commitments
    - hash of tapscript root for Taproot-based outputs

Re-computable (Supplement)

- Can be reconstructed from information that is usually persisted outside of DBC scope:
  - Transaction graph stored in
    - Bitcoin blockchain
    - Lightning channel
  - RGB genesis data
- Reconstruction may take significant time + require external services (Bitcoin Core, LN node, Electrum service etc), so the supplement can be persisted as well for optimization reasons
- Structure of supplement is host-specific

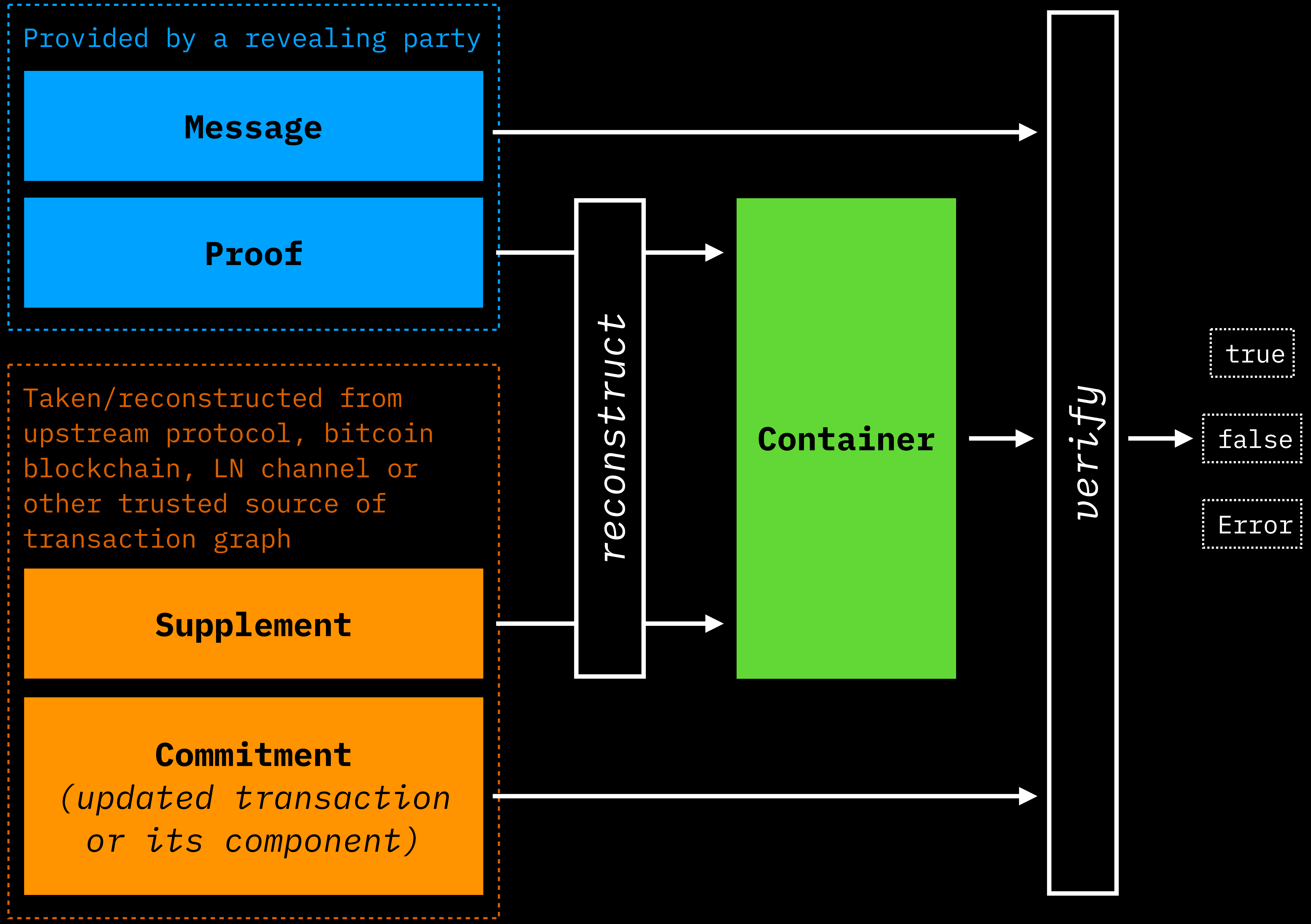
DBC Embed-Commit Workflow

- Procedure may fail with a negligible probability b/c of elliptic curve tweak procedure collisions (more details on LNPBP-1 screen)
- Procedure may fail due to incorrect Container construction



DBC Verification Workflow

- Must succeed only for the original message and correct proof
- Must fail with proper error on incorrect proof data
- Must fail with “false” on message that is different from the original
- If multiple commitments were applied consequently, must succeed only for the last message



Interfaces

- paradigms::commit_verify::**EmbedCommitVerify**
  - **embed_commit**(<u>Container</u>, Message) -> <u>Commitment</u> *(can fail)*
  - **verify**(<u>Commitment</u>, <u>Container</u>, Message) -> true/false
- bp::dbc::types::**Container**
  - **reconstruct**(Proof, <u>Supplement</u>, [<u>Host</u>]) -> Container
  - **deconstruct**(Container) -> (Proof, <u>Supplement</u>)
  - **to_proof**(Container) -> Proof
  - *container construction is a custom process depending on which part of the transaction is constructed*

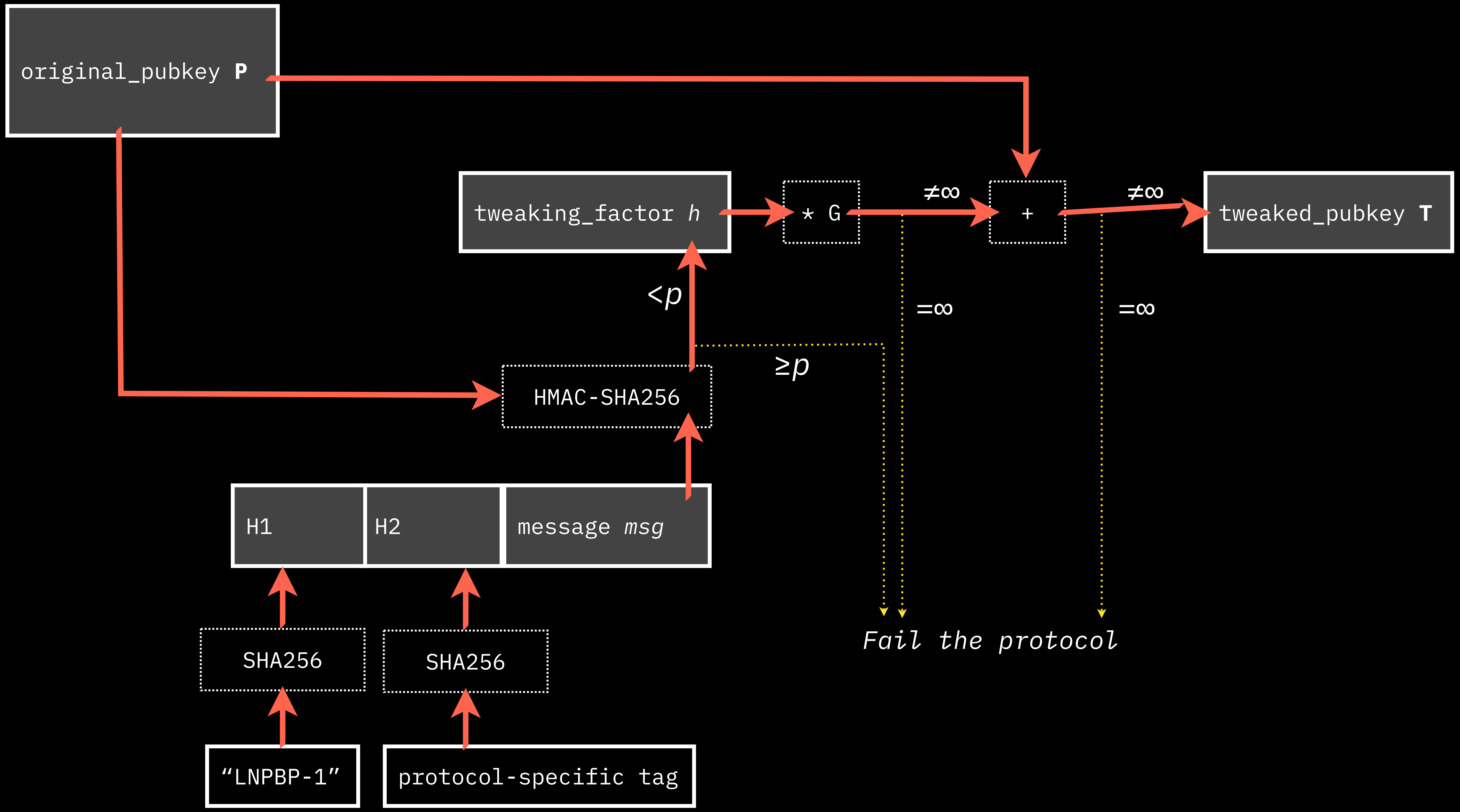
Legend:

- Type – *concrete type*
- <u>Type</u> – *generic type*

Important data structures

- bp::dbc::types::**Proof**
  - **original_pubkey**: secp256k1::PublicKey
    original public key before any commitment applied
  - **script_info**: **ScriptInfo** (enum)
    information required to detect the original public key within scriptPubkey of the transaction output
    - None
    - RedeemScript(bitcoin::Script)
    - Taproot(sha256::Hash) *// tapscript root hash*

LNPBP-1: public key-based commitments



LNPBP-2: Script-based commitments

Challenge with bitcoin script in transaction:

- transaction outputs (the main target for commitment embedding) in many cases does not contain information sufficient to reconstruct public keys: information required for LNPBP-1 commitment

We have made an re-think of Bitcoin script hierarchy related to transaction structures and have designed a robust way for embedding script-based commitments into different types of transaction outputs

Side note #1

Bitcoin Script type hierarchy & transformations

guide into **bp::scripts** module of *LNP/BP Core Library*

Life of Bitcoin Script

We start with either of these

- Single public key
- Public keys and hash preimages
- Script template (miniscript etc)

We decide on scriptPubkey representation type

- Explicit
- Key hash
- Script hash
- Taproot

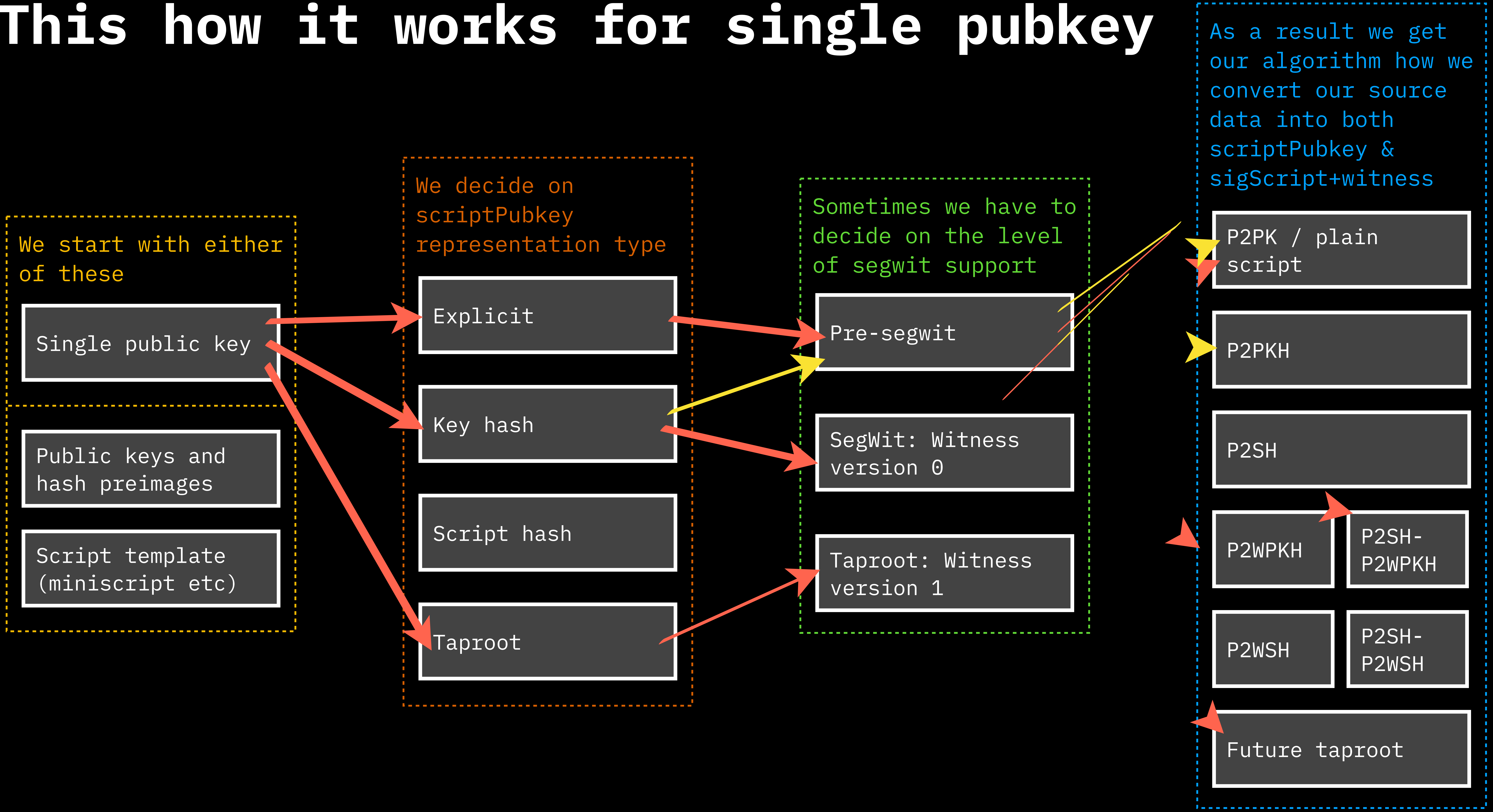
Sometimes we have to decide on the level of segwit support

- Pre-segwit
- SegWit: Witness version 0
- Taproot: Witness version 1

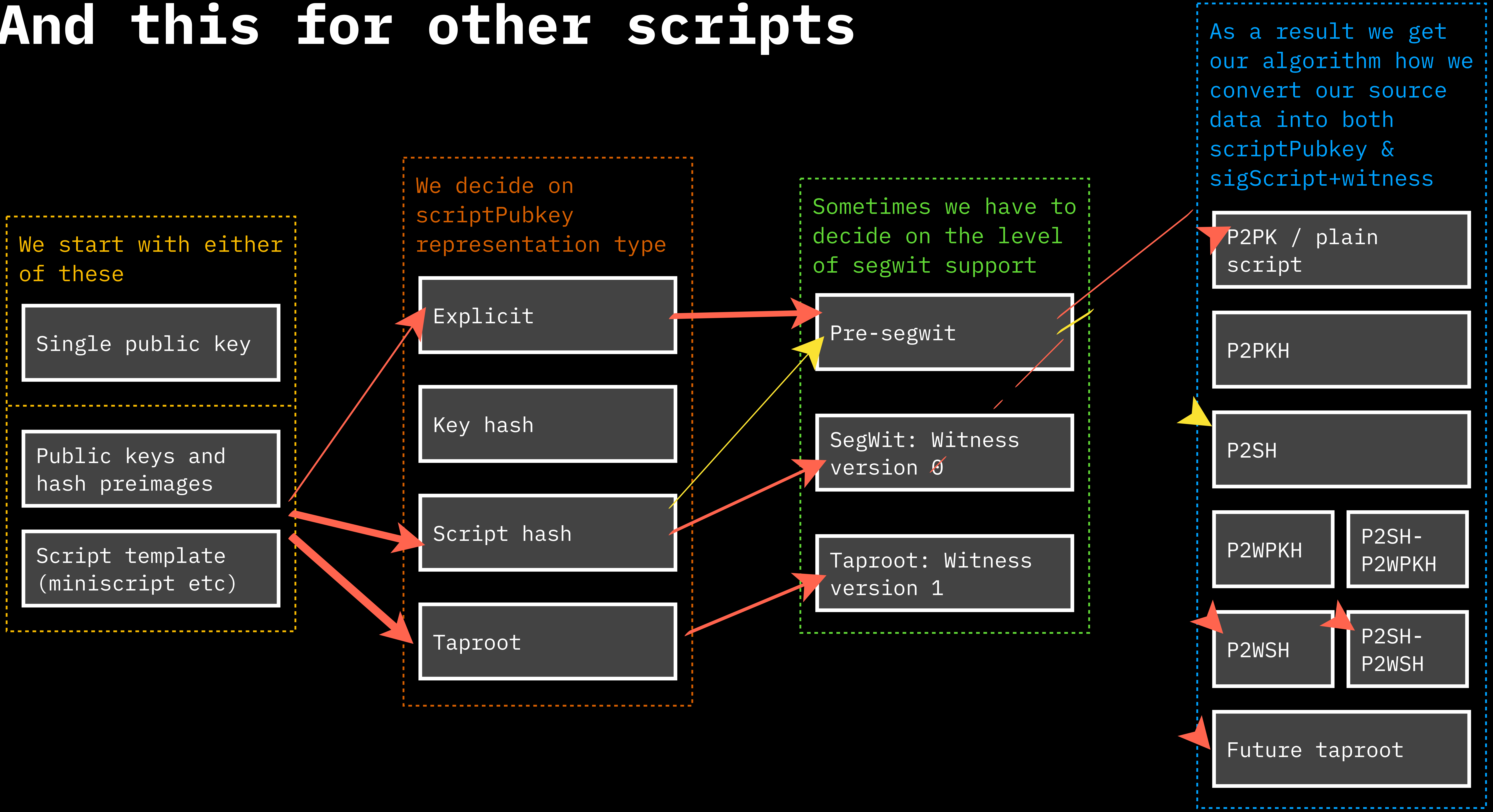
As a result we get our algorithm how we convert our source data into both scriptPubkey & sigScript+witness

- P2PK / plain script
- P2PKH
- P2SH
- P2WPKH
- P2SH-P2WPKH
- P2WSH
- P2SH-P2WSH
- Future taproot

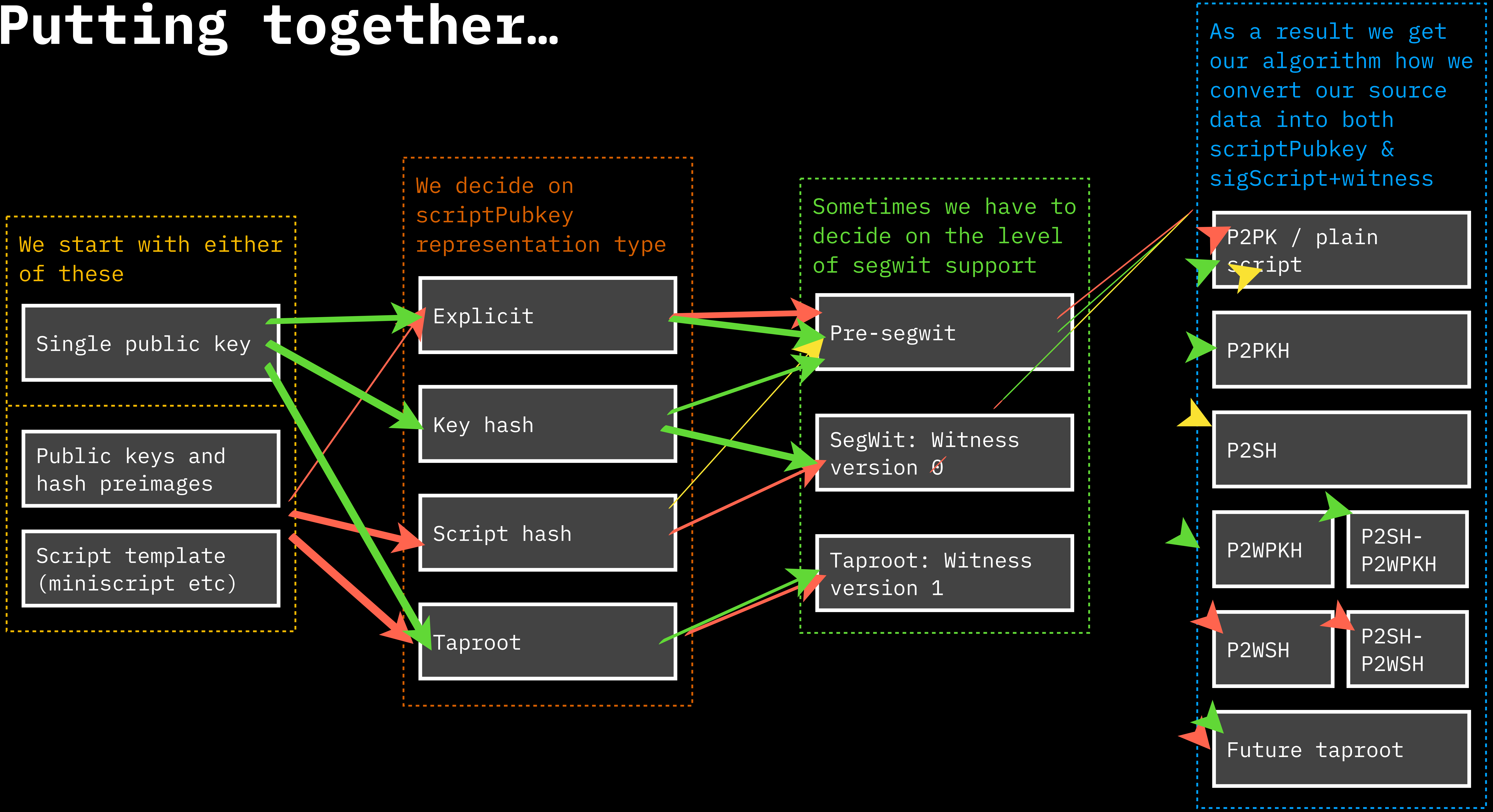
This how it works for single pubkey



And this for other scripts



Putting together…



- To do DBC we had to sort out this mess
- This will also simplify life for wallet developers
- We need strong type system and algorithm workflow

Type system

Hash (and related) types

- **bitcoin::PubkeyHash**: bitcoin HASH160 (SHA256 followed by RIPEMD160) of public key
- **bitcoin::ScriptHash**: bitcoin HASH160 (SHA256 followed by RIPEMD160) of bitcoin script
- **bitcoin::WPubkeyHash**: variant of public key bitcoin HASH160 (SHA256 followed by RIPEMD160) designed for WitnessProgram
- **bitcoin::WScriptHash**: double SHA256 hash of bitcoin script designed for WitnessProgramm

Script types

- **bp::PubkeyScript**: anything that we can put into or take from *scriptPubkey* field of transaction output
- **bp::SigScript**: anything that we can put into or take from *sigScript* field of transaction input
- **bp::Witness**: anything that we can put into or take from *witness* field of transaction input (SegWit-only)
- **bp::RedeemScript**: reconstructed from/used for *sigScript* (non-SegWit) or *witness* (SegWit v0) fields of transaction input
- **bp::WitnessScript**: a part of the *witness* field containing bitcoin script, en equivalent of *redeemScript*, however hashed with double SHA256 hash
- **bp::TapScript**: any branch of Tapscript; can't be put or reconstructed from any part of bitcoin transaction directly
- **bp::LockScript**: a script which contains complete/explicit form of public keys used to control the spending; equal to either RedeemScript (if present) or PubkeyScript (for P2PK outputs)

Type system

Other types

- `bp::scripts::WitnessProgram`: (not necessary a hash) content of 2-40 byte push in SegWit-enabled transaction outputs that follows SegWit version 1-byte push opcode
  - Equals to either *WPubkeyHash* or WScriptHash for V0 of SegWit
  - Equals to the public key serialized according to BIP-Schnorr in V1 SegWit outputs (Taproot)
- `bp::scripts::WitnessVersion`: enum that covers possible SegWit versions, from 0 to 16

`bp::scripts::ConversionStrategy`

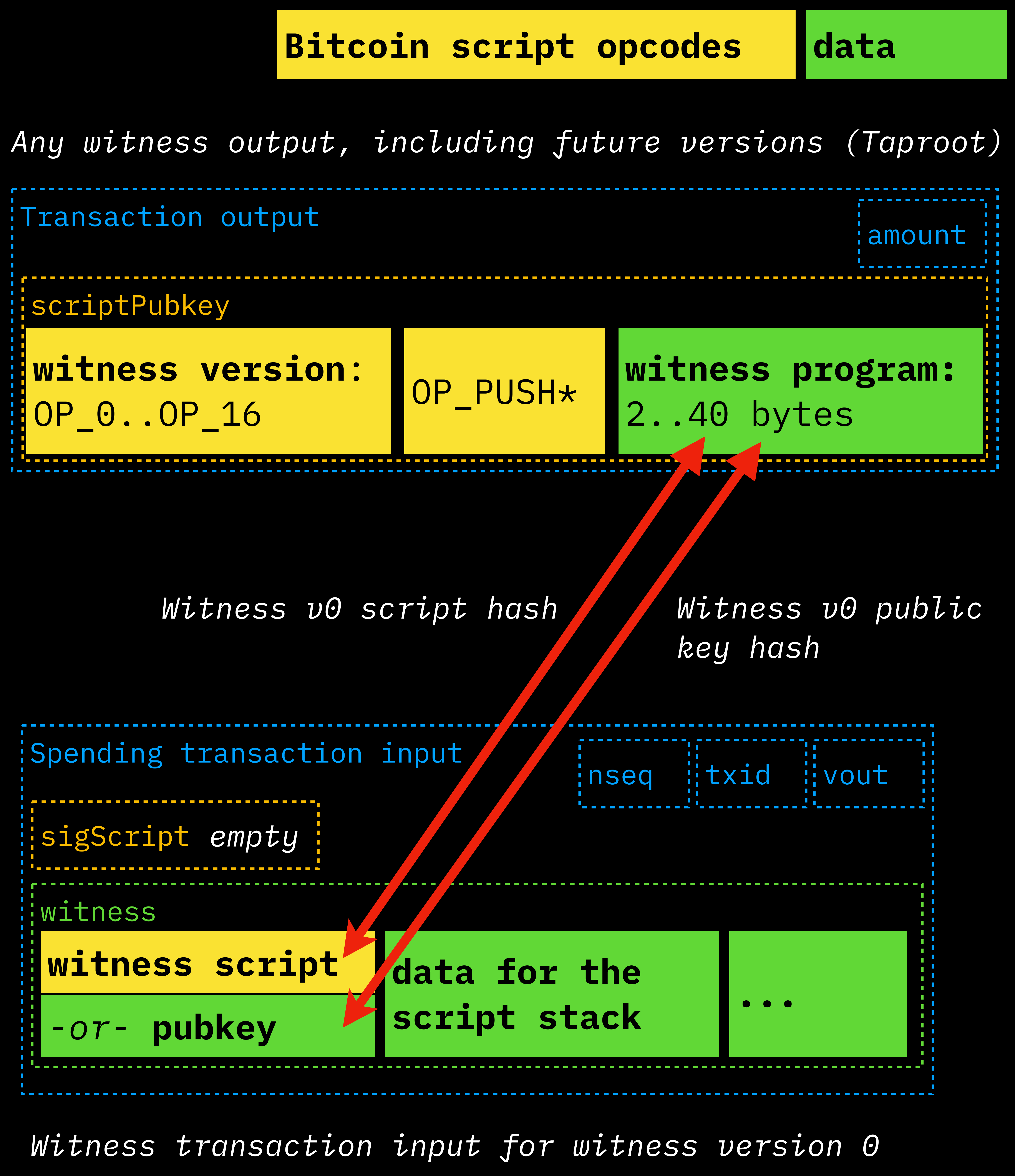
Defines strategy for converting some source Bitcoin script (i.e. *LockScript*) into both *scriptPubkey* and *sigScript/witness* fields

- *Exposed*: the script or public key gets right into *scriptPubkey*, i.e. as **P2PK** (for a public key) or as custom script (mostly used for **OP_RETURN**)
- *LegacyHashed*: we hash public key or script and use non-SegWit *scriptPubkey* encoding, i.e. **P2PKH** or **P2SH** with corresponding non-segwit transaction input sigScript containing copy of *LockScript* in *redeemScript* field
- *SegWitV0*: we produce either **P2WPKH** or **P2WSH** output and use *witness* field in transaction input to store the original LockScript or the public key
- *SegWitScriptHash*: SegWit version and program become *redeemScript* in *witness* field of transaction input, which is encoded as **P2SH** in *scriptPubkey* (**P2SH-P2WPKH** and **P2SH-P2WSH** variants)
- *SegWitTaproot*: will be used for Taproot

SegWit-specific data

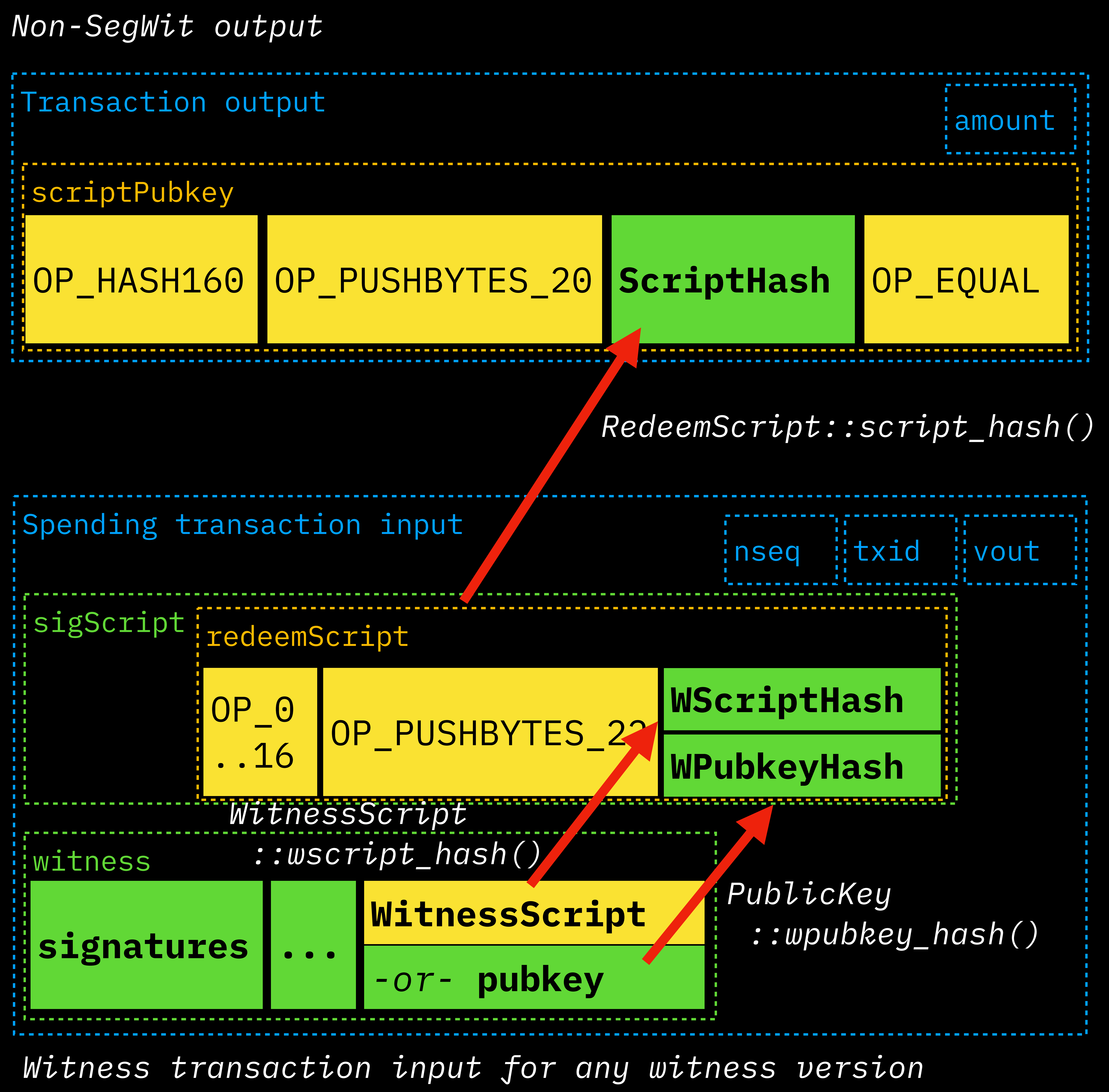
SegWit-related terminology is very confusing

- **Witness version:** first opcode in *scriptPubkey* when it has value<=16 (**bp::WitnessVersion**)
- **Witness program:** data pushed to the stack by the second instruction in *scriptPubkey*; must be from 2 to 40 bytes long (**bp::WitnessProgram**)
- **Witness:** data structure (not Script!) in transaction input; a vector of variable-length byte strings (**bp::Witness**)
- **Witness script:** equivalent of redeemScript in non-SegWit transactions; contained as one of witness records preceding signature data (**bp::WitnessScript**)
- **Witness v0 script hash:** variant of witness program for P2WSH outputs (**bitcoin::WScriptHash**)
- **Witness v0 public key hash:** variant of witness program for P2WPKH outputs (**bitcoin::WPubkeyHash**)

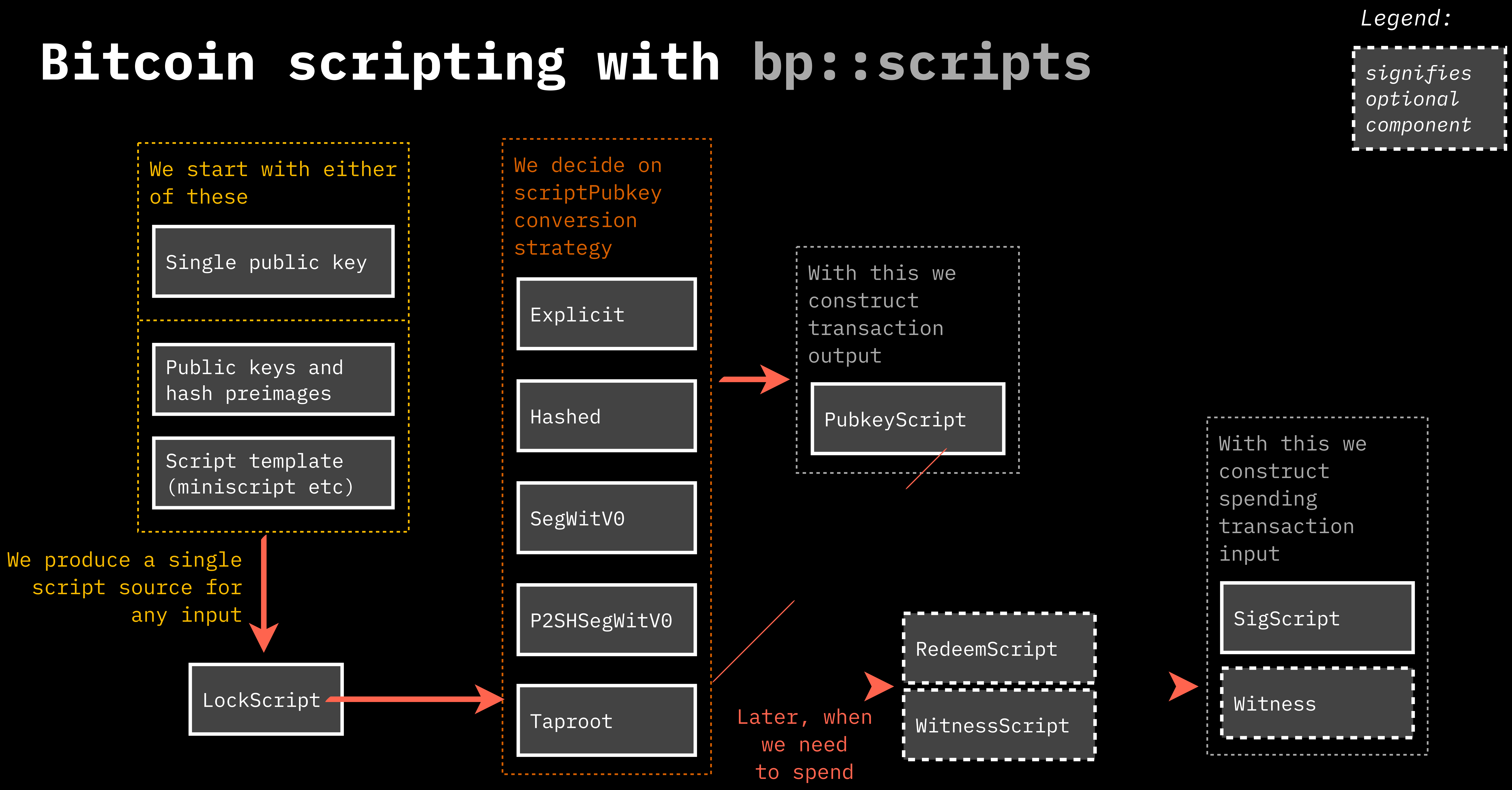


P2W*H in P2SH

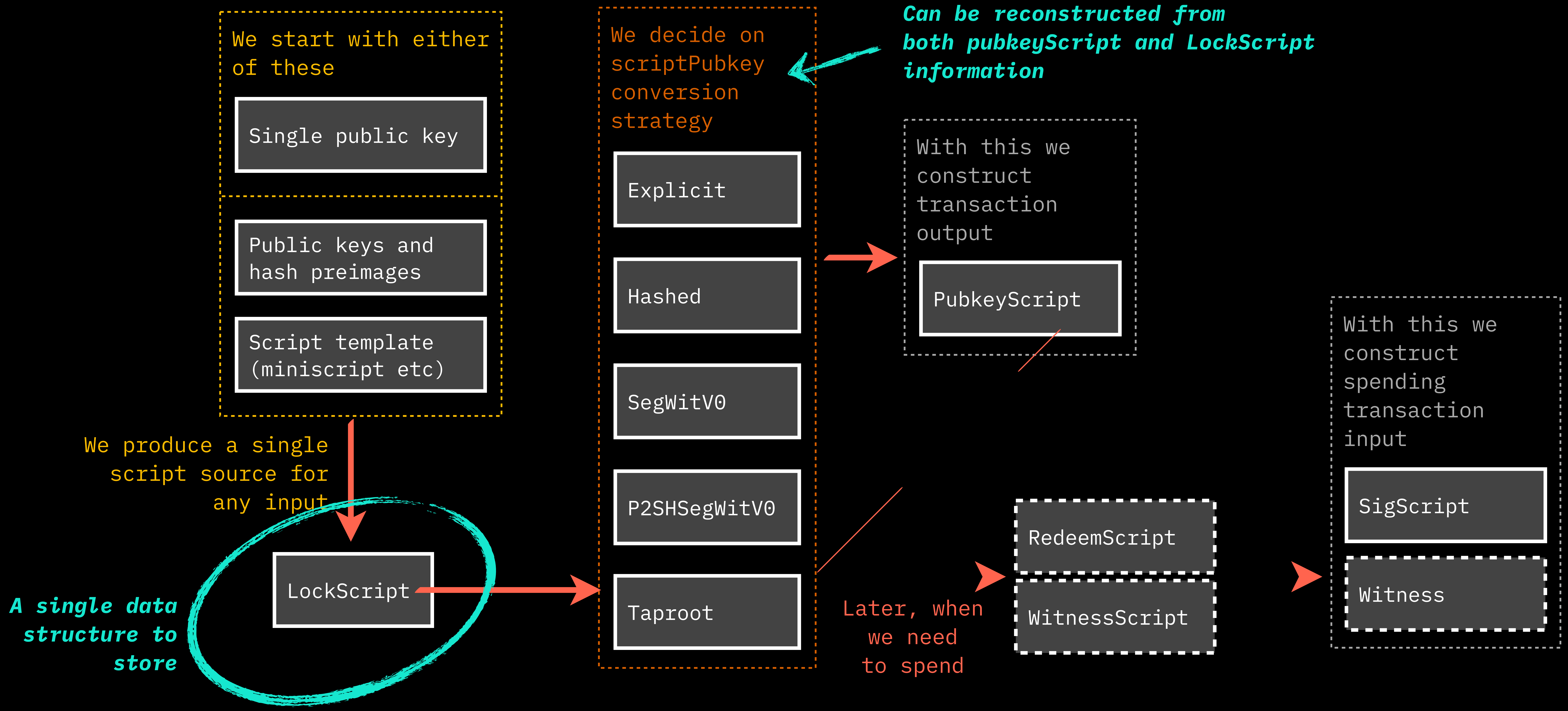
- Both RedeemScript and WitnessScript are present
- Script is fact hashed twice: once in sigScript field, and the second time in scriptPubkey



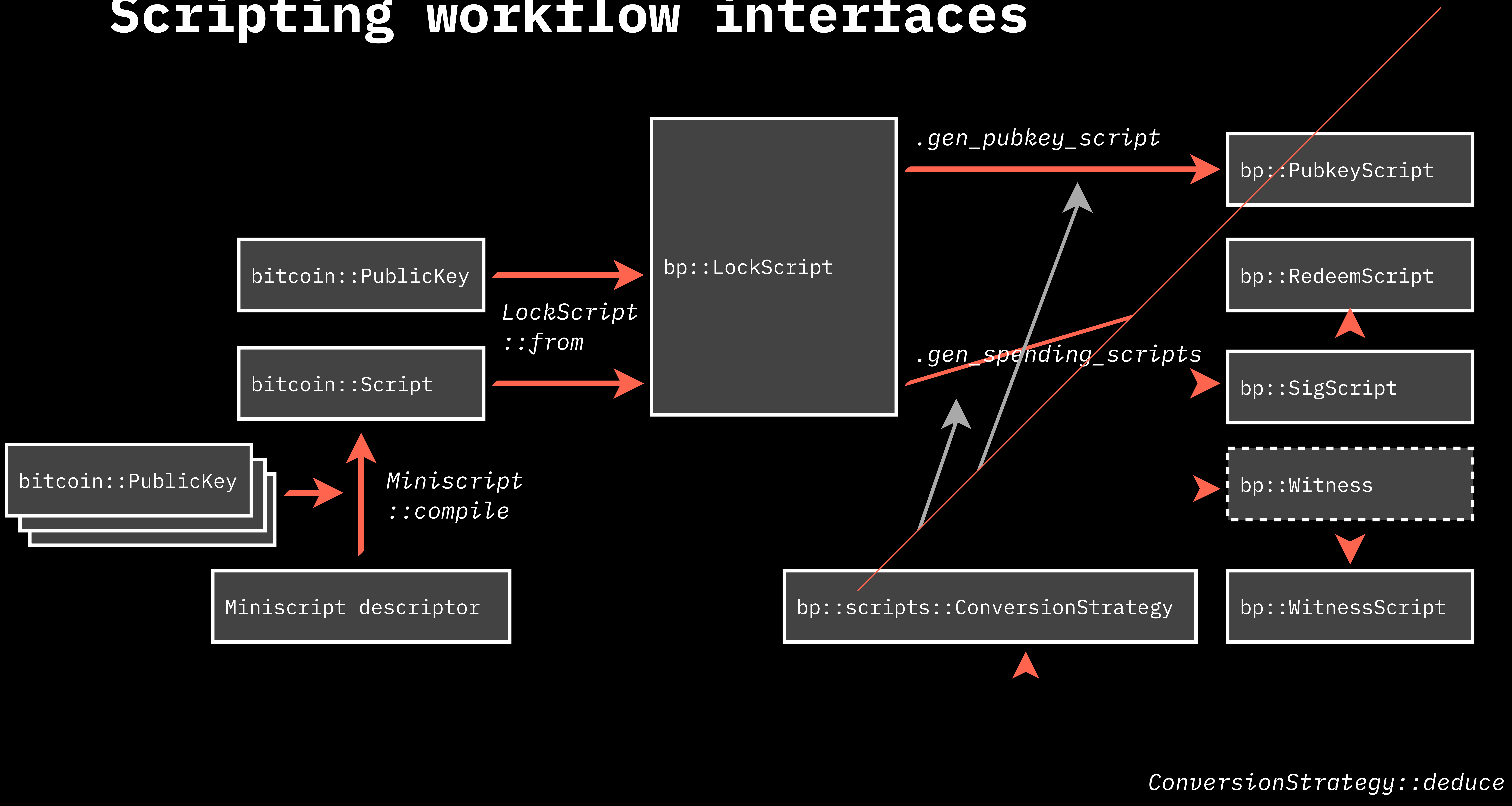
Bitcoin scripting with bp::scripts



Bitcoin scripting with bp::scripts



Scripting workflow interfaces



LNPBP-2: Script Embed-Commit procedure

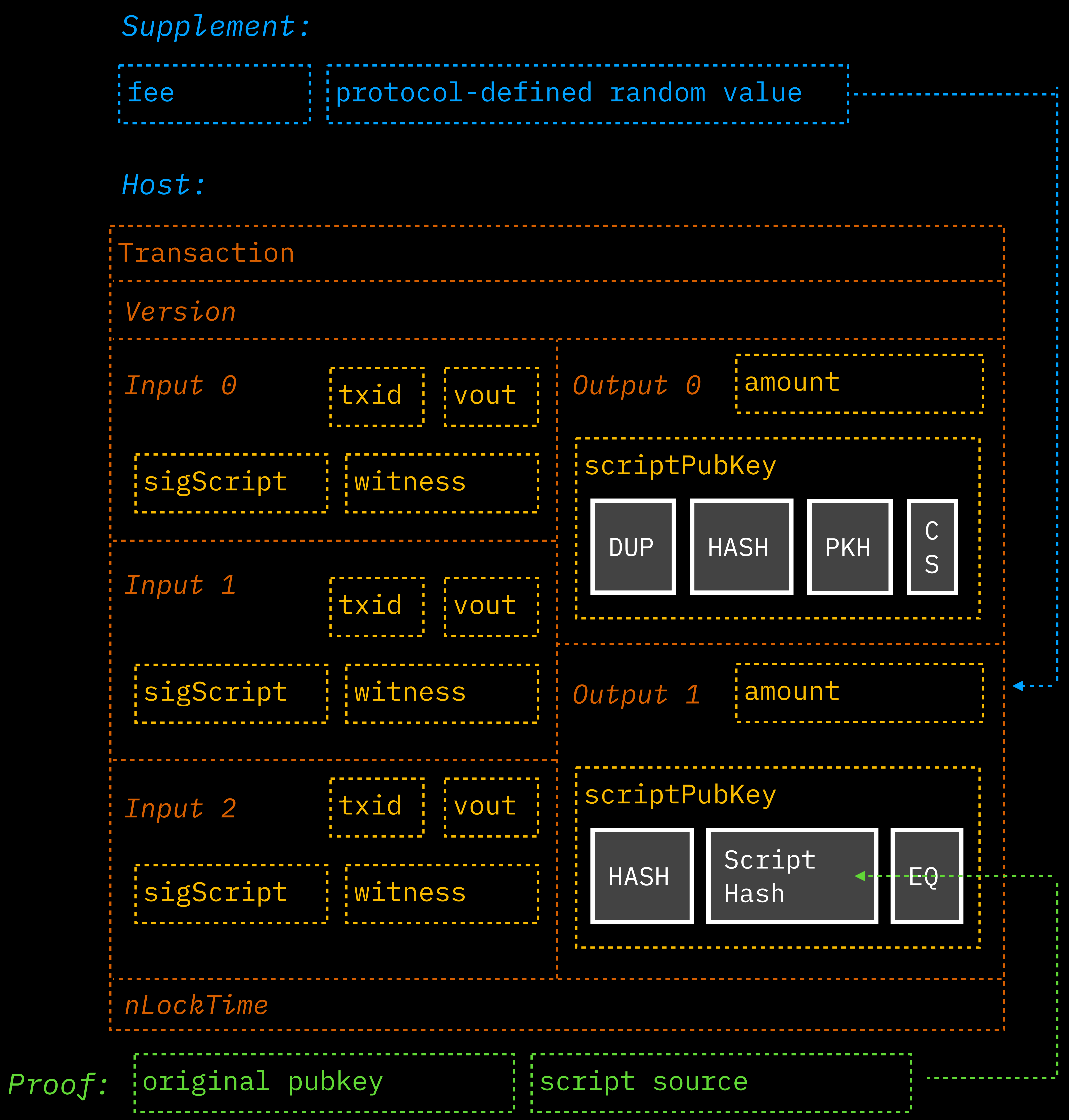
- All public keys within the script should be known for both embed-commit and verify procedures
- Only a single public key is tweaked with modified LNPBP-1 procedure (in all it's instances, both hashed and un-hashed)
- In LNPBP-1 procedure modification we commit to the sum of all untweaked public keys (each unique key is counted once; no mater how many times it appears in the script)
- LockScript type makes procedure easy (since it contains an explicit form of all the keys)

LNPBP-3: Put embed-commit TxOut into Tx

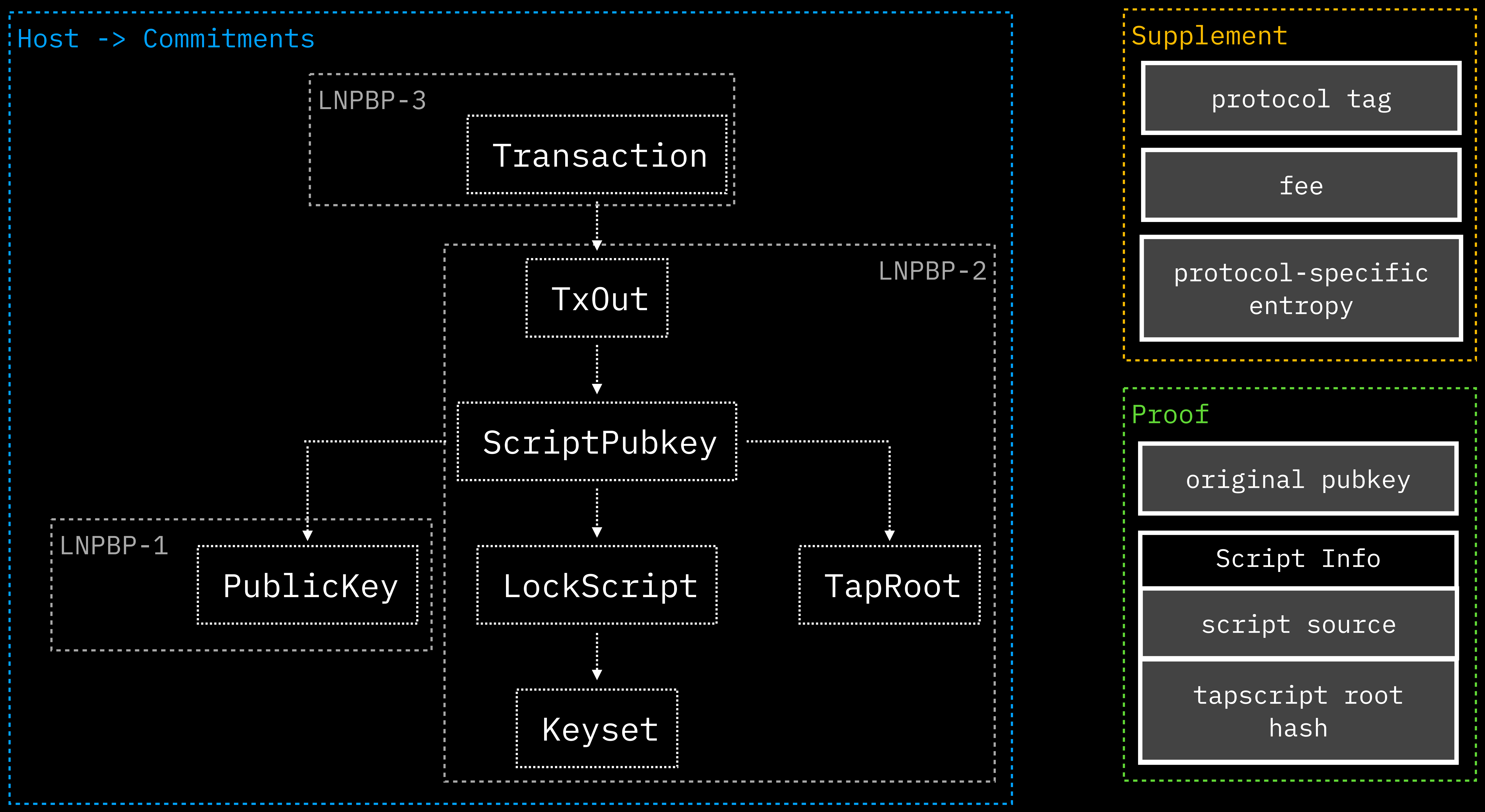
- We need deterministic and private way to define which transaction output is an embed-commit output
- Such output is defined as (transaction_fee + protocol_specific_value) mod num_outputs
- Transaction fee is publicly known, but since each protocol defines its own value, and there may be endless number of such protocols (like different asset types), potentially each output may contain embed-commit for some protocol, making onchain analysis inefficient

Putting DBCs together

- Construct original transaction with required source scripts
- Find which output to use
- Take public keys involved in that output; define which one of them will be tweaked
- Tweak public key (if there are many, commit to the sum of all public keys)
- Re-construct the script hierarchy
- Update output; construct and keep proof
- Sign and publish transaction



Core DBC workflow & components



A code example

```
// Done by comitting party
let container :TxContainer = TxContainer::construct(
    protocol_factor: 0,
    protocol_tag: &sha256::Hash::hash( data: b"RGB"),
    fee: 100, // tx fee
    tx: transaction,
    pubkey: keys[2],
    script_info: ScriptInfo::LockScript(lockscript_from_miniscript!(
        "or(thresh(3,pk({}),pk({}),pk({})),and(thresh(2,pk({}),pk({})),older(10000)))",
        keys[0],
        keys[1],
        keys[2],
        keys[3],
        keys[4],
    )),
    scriptpubkey_composition: ScriptPubkeyComposition::SHWScriptHash,
);

let msg :&str = "message to commit to";

let commitment :TxCommitment = TxCommitment::embed_commit(&container, &msg)?;
// We publish witness transaction from the commitment
let witness_tx = commitment.into();

// We discard supplement; keep the proof / transfer it to the verifying party
let (proof :Proof , supplement :Supplement ) = container.deconstruct();
```

- Bitcoin network / LN channel
- Keep & send other party

A code example

We know this information from blockchain / LN channel & protocol information

Proof we kept / received from the other party

```
// Later at verification stage
let supplement = TxSupplement {
    protocol_factor: 0,
    fee: 1000,
    tag: sha256::Hash::hash( data: b"RGB")
};
let container :TxContainer = TxContainer::reconstruct(&proof, &supplement)?;
let commitment :TxCommitment = TxCommitment::from( t: tx); // Tx from blockchain
commitment.verify(&container, &msg)?;
```

Verification result

Bitcoin network / LN channel

Glossary

Generic cryptographic commitments

- Container: structure used to embed commitment
- Commitment: structure holding the commitment

Deterministic bitcoin commitments

- Host: transaction or part of it in which we will keep commitment
- Proof: important sensitive data required for commitment verification; must be kept by client
- Supplement: other data needed for commitment verification; reconstructable from transaction graph and publicly-known protocol information

To be continued in Part II:

- Single-use seals and their application to bitcoin (transaction output-based seals) explained
- Client-side validation explained;
  how it is related to single-use seals and other commitment technologies
- How RGB is structured and why it is structured in such way:
  details on
  - contract genesis
  - state assignments
  - state transitions
  - anchors
  - consignments
  - seals allocations
  - stash
- How confidentiality in RGB works: fully-disclosed and partially-revealed data for transitions, seals & consignments

Seems like we will need Part III as well:

- RGB-enabled Lightning channel specifics
- P2P protocols:
  - RGB wire protocol
  - LN protocol integration
  - Spectrum: LN multi-hop payment and inter-contract protocol
- Fungible Assets API
- Presentation of RGB SDK:
  - Kaleidoscope: first RGB-enabled command-line wallet & tool
  - BP Node: indexing bitcoin node for RGB operations
  - LNP Node: lightning node able to work with RGB
  - RGB Node: backend for RGB transfers
  - RGB Wallet SDK: early prototype