LNP/BP Standards Association

RGB

Programming RGB smart contracts

Dr Maxim Orlovsky,
Pandora Core AG

Previous talk re-cap

What is client-side-validation?

- **Trustless distributed computing systems** operating on layer 2/3 using layer 1 for state verification
  (any layer above 1 uses consensus from layer 1)
- … in which **user is responsible for keeping own data** (and not the network doing that “for free”)
- Both an alternative (as layer 2) or an extension (as layer 3) to state channels (lightning channels etc)
- Coined by Peter Todd as a logical extension of his work on OpenTimeStamps

Not client-side-validation

<u>Client-side data, but not client-side-validation:</u>

- Bitcoin P2SH and P2TR requiring keeping script source on client side
- MuSig2 requiring client-side state
- State channels (LN, DLCs etc)

Client-side-validation is when you **validate** client-side data against certain rules, which **must include blockchain-based commitments** (timestamps or single-use-seals)

State channels

Client-side-validation

similarities and differences: neutral negative positive

State channels

- Works with blockchain and, sometimes, other state channels
- Synchronous
- May require routing (lightning channels)
- Uses client-side data (signatures or transactions)
- Tiny size of client-side data
- Security requires watchtowers
- No state validation (outside of blockchain-based mining & transaction validation scopes)

Client-side-validation

- Works with both blockchain and any other state channels
- Asynchronous (network fault tolerant)
- May require storage providers
- Uses client-side data (any form fo complex state)
- Huge size of client-side-data
- No watchtowers required (*when not on top of state channels)
- Performs state validation additional to blockchain-based mining & transaction validation

Why client-side-validation?

- More confidentiality *(than in blockchain)*
- More scalability *(than in blockchain)*
- More programmability *(than in both blockchain and state channels)*
- Richer state *(than in both blockchain and state channels)*

Client-side-validated systems

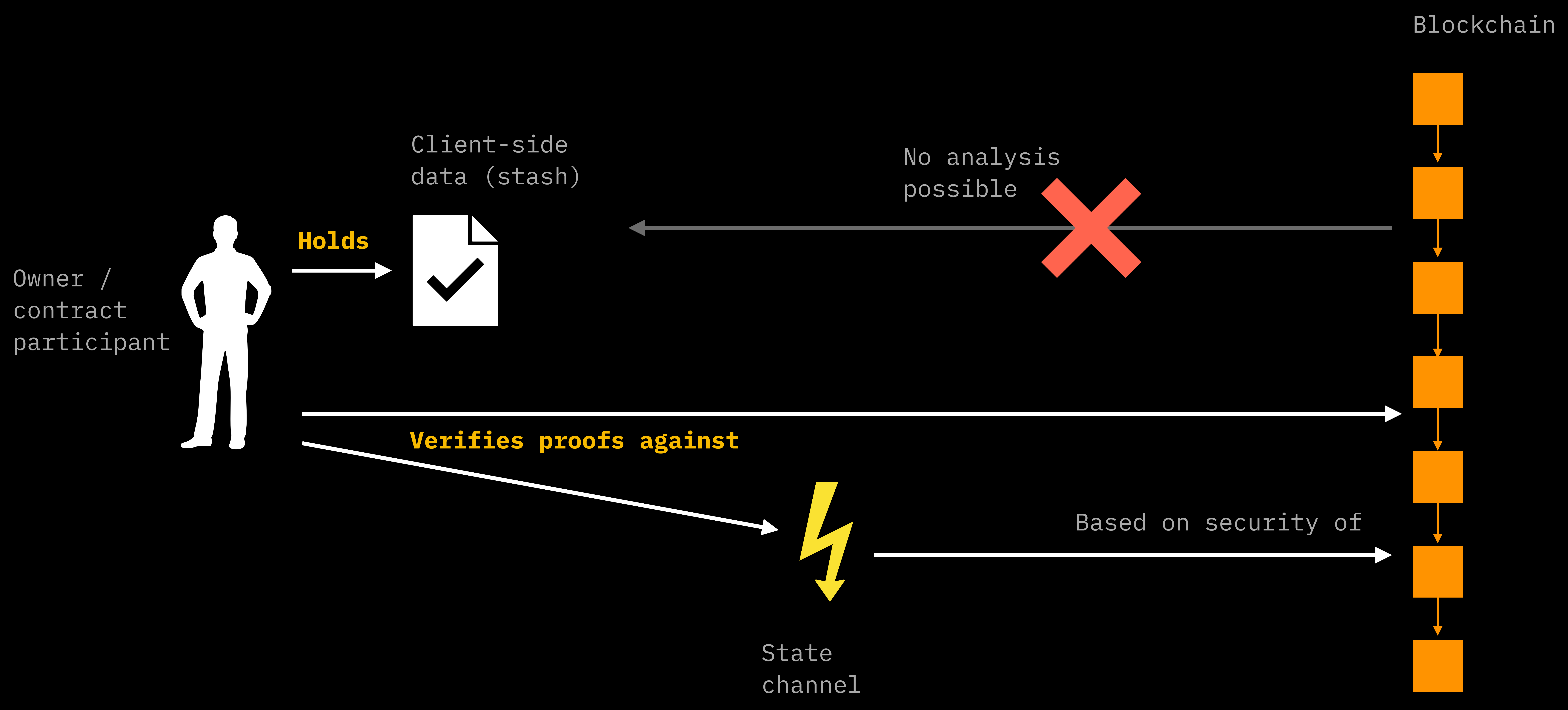
Timestamp based

- OpenTimeStamps

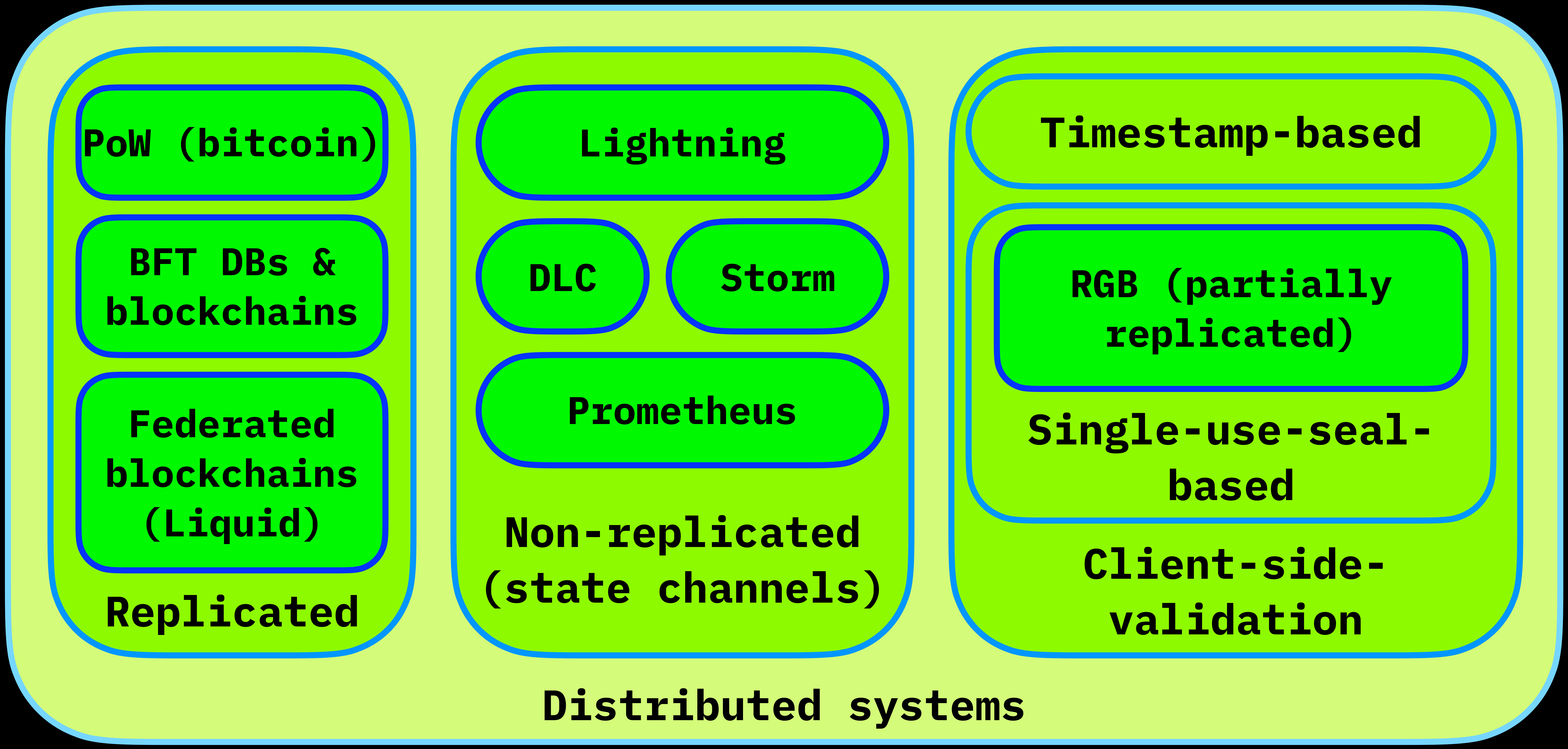
Single-use-seal based

- RGB (smart contracts)
- BIP32/43-based standard for Schnorr signatures & decentralized identity
  https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2021-February/018381.html
- …more to come?

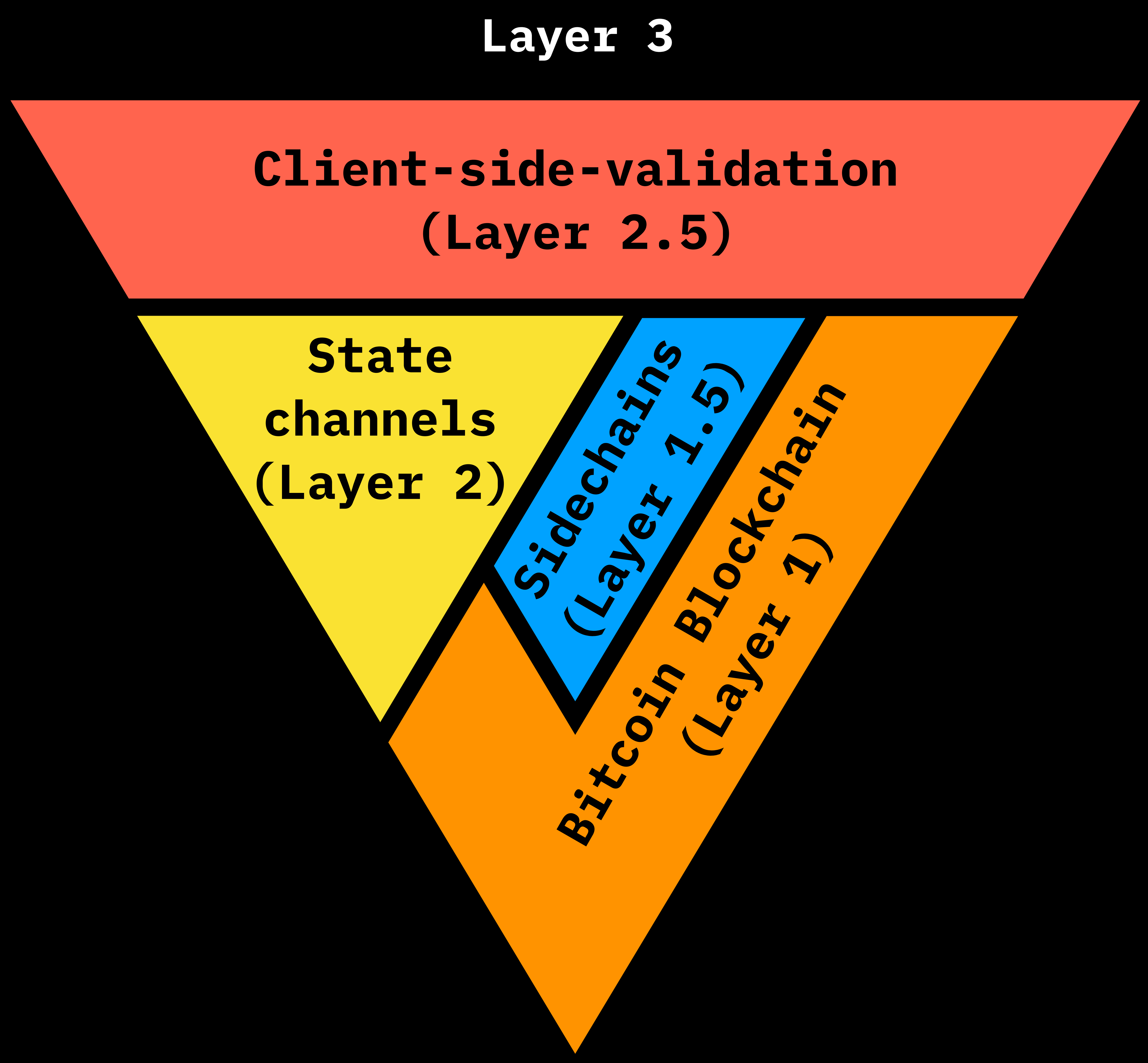
RGB: smart contracts using client-side validation



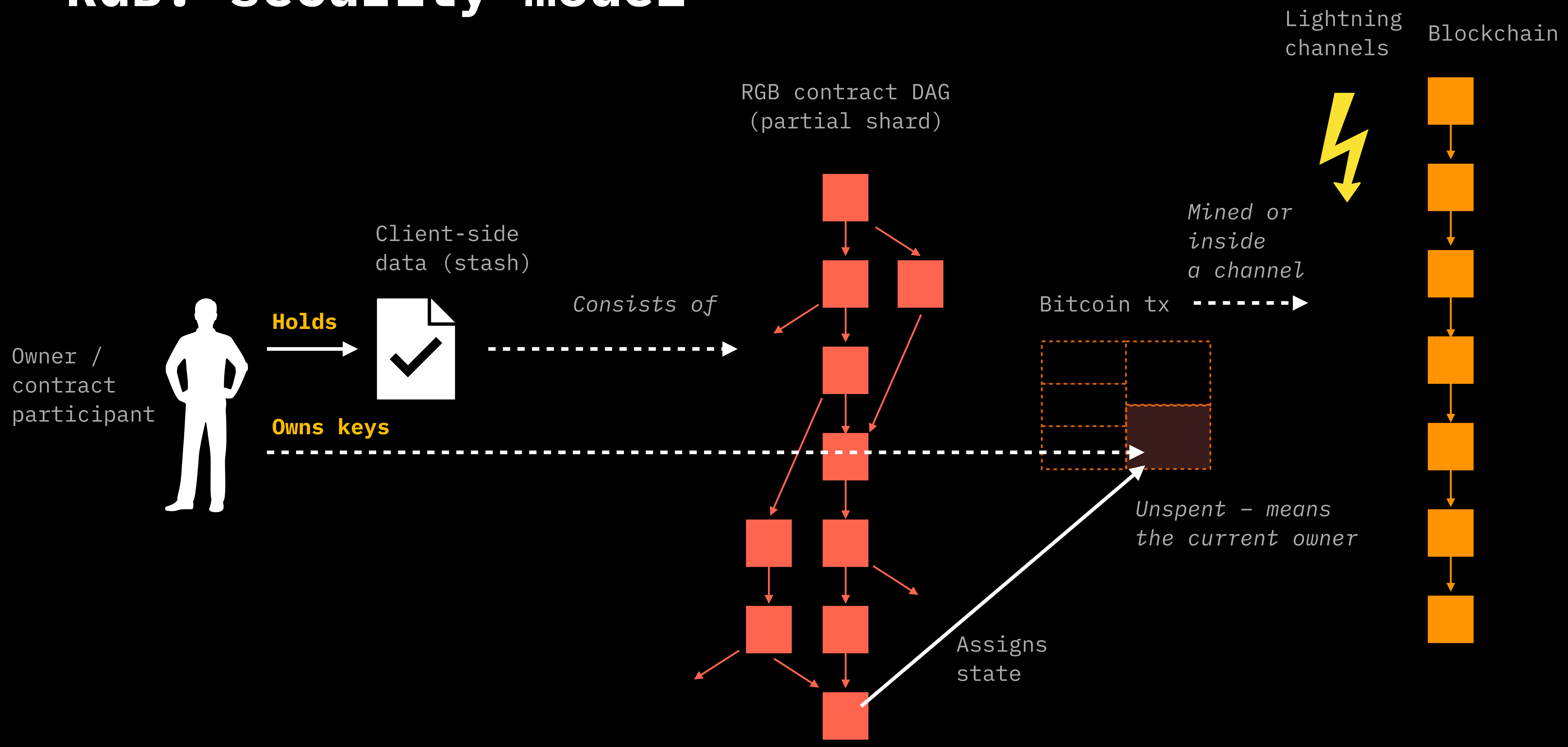
Putting it all together



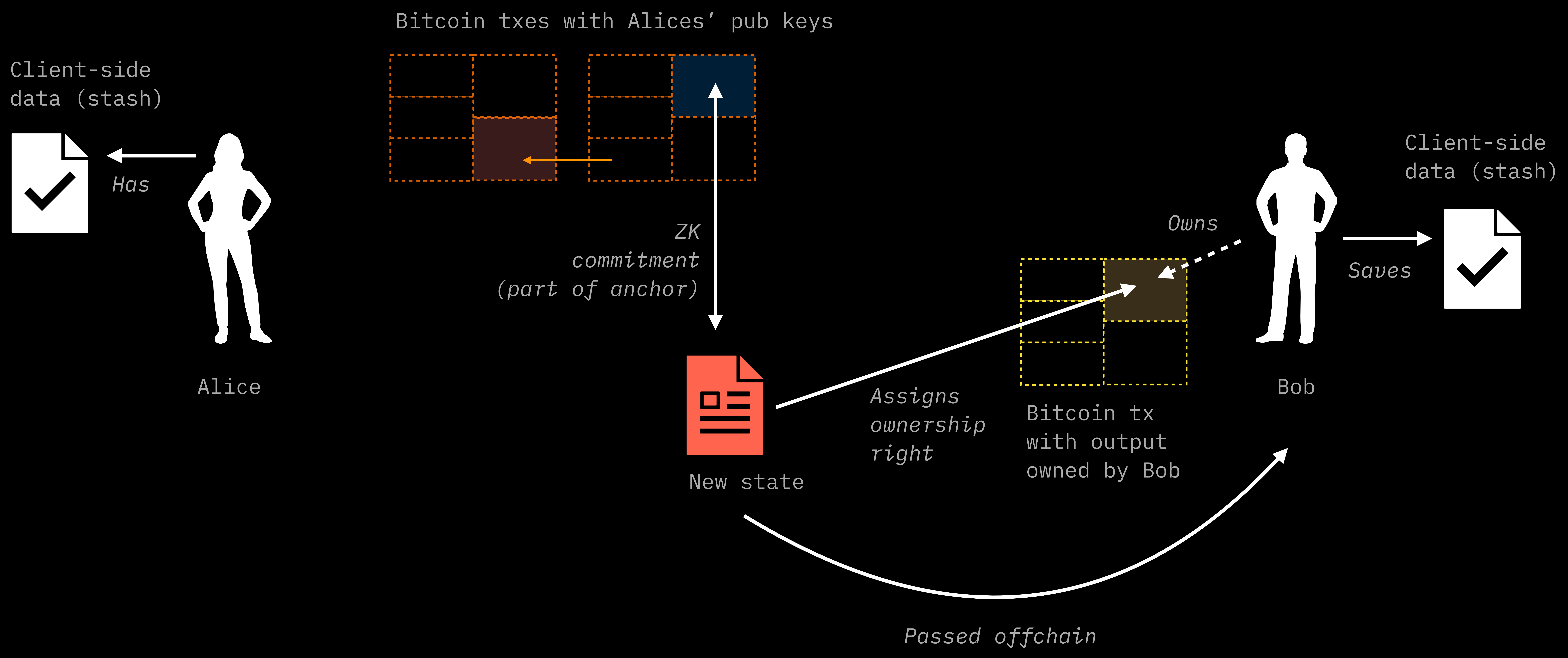
What is Layer 2 and Layer 3?



RGB: security model



RGB: how state transfer works



Client-side validation with RGB

- Ownership & “double-spent” prevention: re-using **Bitcoin script** combined with **single-use-seal validation**
- Consistency & completeness: **Schema**
- Changes in state: **AluVM**
  you can learn more about AluVM and its use in RGB at
  - *https://github.com/LNP-BP/presentations/blob/master/Presentation%20slides/Single-use-seals.pdf*
  - *https://youtu.be/brfWta7XXFQ*

RGB: topdown approach

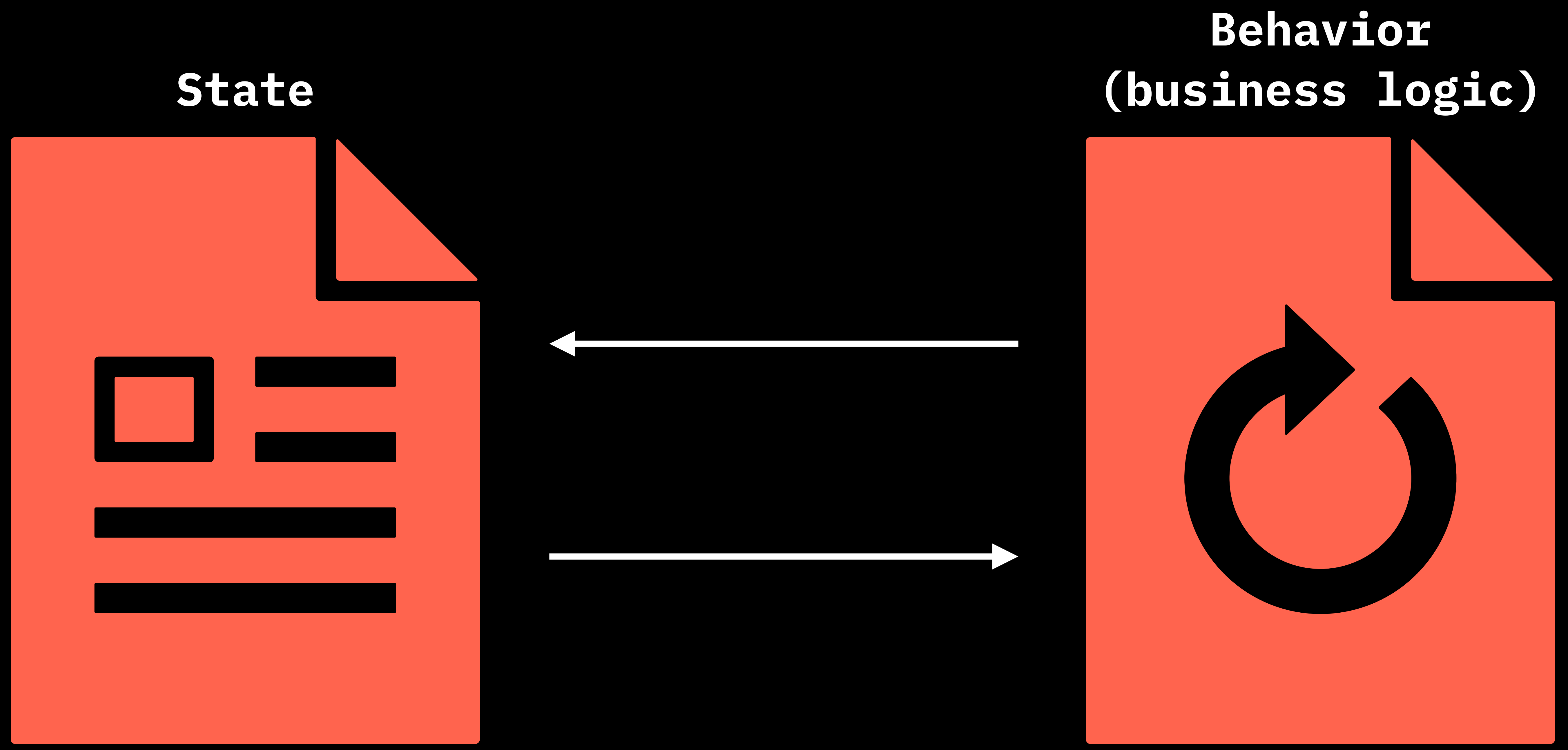
Smart contract

Automatically* enforced* agreement between parties

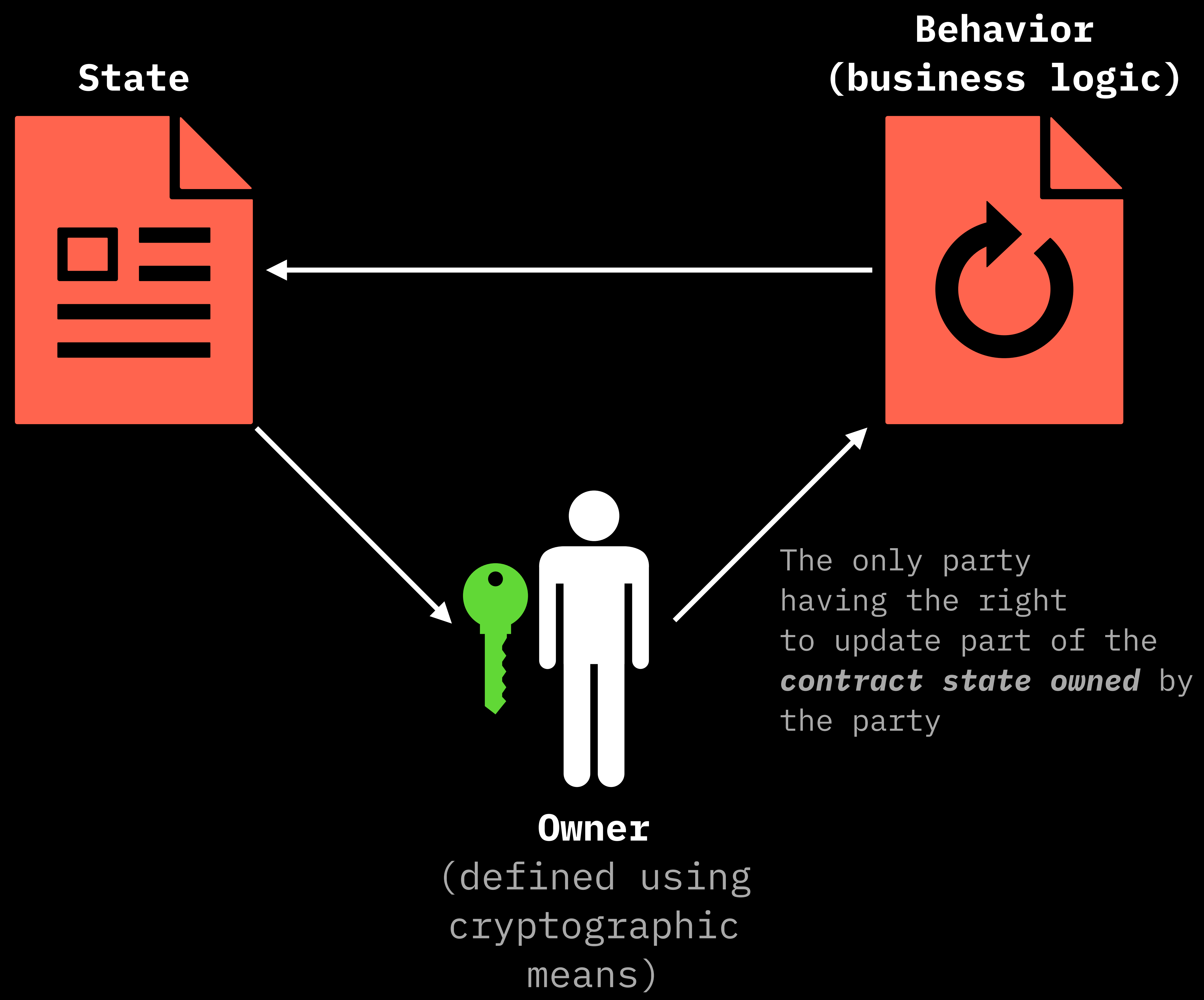
* "Automatic" means without direct human involvement or subjective factors

** Enforced by economic and mathematic (computing) means, without any form of physical violence

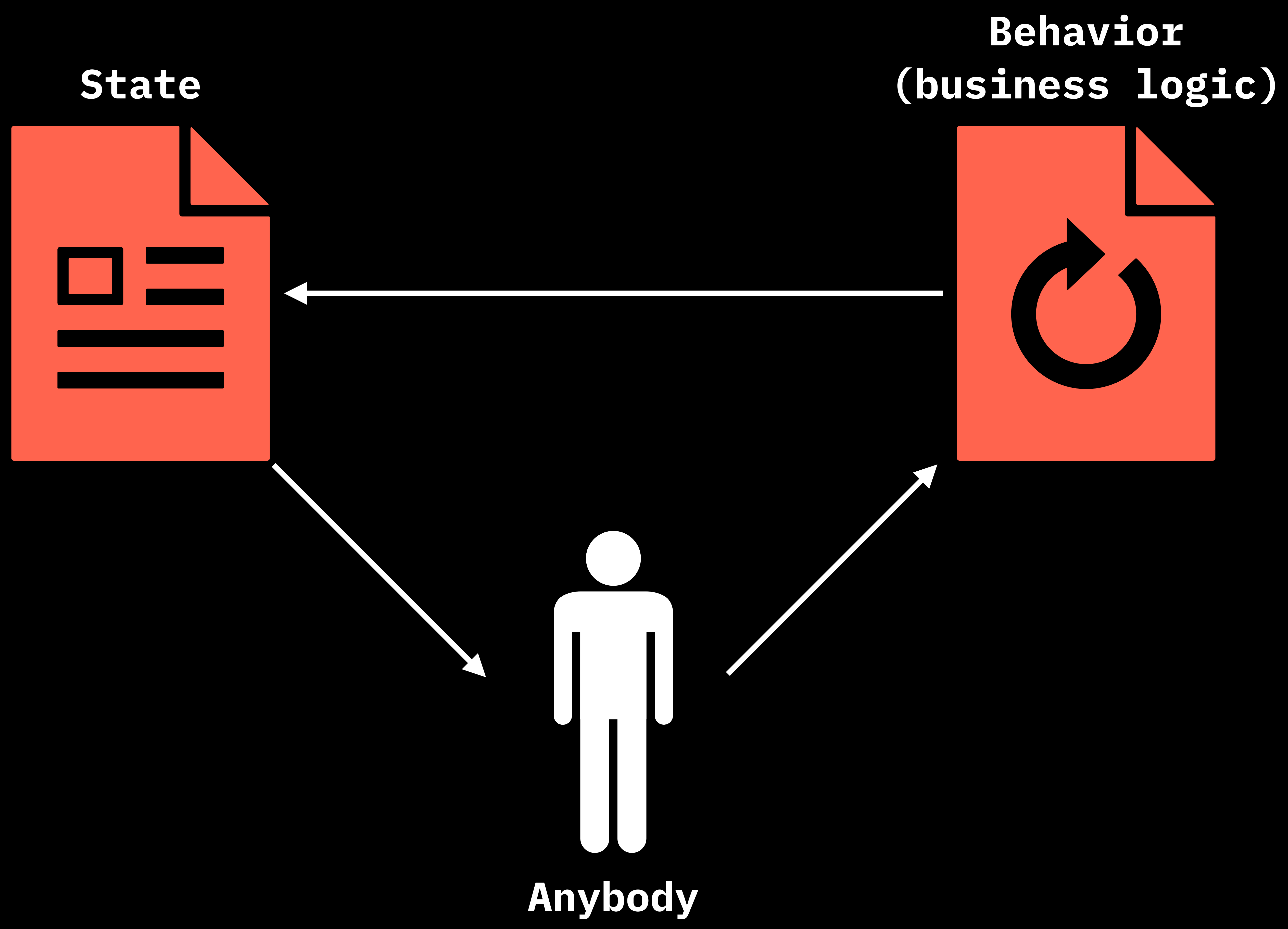
Smart contract



RGB smart contracts: owned state



RGB smart contracts: unowned* state



* we need a better term

Single-use-seals

- Presentation: https://github.com/LNP-BP/presentations/blob/master/Presentation%20slides/Single-use-seals.pdf
- Video recording: https://www.youtube.com/watch?v=gGPLYfWOb_8

To start with a smart contract we need to start with state

What is state

Arbitrary rich data (fitting certain size restrictions)

Rich means:

- Typed (strongly typed)
- May be nested (one types are composed of other types)
- May be organized into collections: **lists**, **sets** and **maps**

Smart contract state is split into “state atoms”, making ownership atomic

Each “state atom” must have a well-defined data type

Thus, to define state you must declare data types first

Size restrictions

Any data of any specific data type must not exceed 64kb

- prevents unlimited growth of client-side validated data
- ensures that any state will always fit into AluVM register

If you need more, either:

- introduce more data types and split state into multiple state entries
- use data containers, which are not part of the validation procedures (keeping NFT data etc)

Number of elements in any collection type must not exceed 2^16 (65536)

Introducing Contractum

- Haskell-inspired language for RGB smart contracts (but not subset of Haskell itself)
- Avoids visual clutter
- Focus on validation procedures
- Avoids foot guns as much as possible
- Designed with composability & category theory in mind, made for formal verification

Defining state datatypes

- Contractum language `datapkg` statement
- Data type definitions may be shared across multiple RGB contracts and be re-used
- LNP/BP Standards Association will provide data type libraries for the most important applications and protocols (Bitcoin data structures, lightning-network related data structures etc)
- The same data types work not just for RGB, but for any other AluVM-based system

```
datapkg BP

alias Hash256 :: U8^32
alias Sha256 :: Hash256
alias Sha256d :: Hash256
alias Sha256t :: Hash256
alias Ripemd160 :: U8^20

alias BlockId :: Sha256d
alias Txid :: Sha256d

alias Amount :: U64

data OutPoint :: Txid, vout U16

data ScriptPubkey :: U8^..10_000
data StackByteStr :: U8^..520
data Witness :: StackByteStr^..1_000
data SigScript :: StackByteStr^..1_000

data TxOut :: Amount,
              ScriptPubkey
data TxIn :: OutPoint,
             nSeq U16,
             SigScript,
             Witness?

data Transaction :: ver U8,
                    inputs TxIn*,
                    outputs TxOut*,
                    lockTime U32
```

Data types are

- Primitive types, covering
  - unsigned integers
  - signed integers
  - float numbers

  of multiple bit size (U8, I32, F64)

- Aliases for existing primitive types via `alias` keyword

- New data types composed of other data types via `data` keyword

- New types list their “fields” via comma, optionally giving them name (name defaults to type name otherwise)

```
datapkg BP

alias Hash256 :: U8^32
alias Sha256 :: Hash256
alias Sha256d :: Hash256
alias Sha256t :: Hash256
alias Ripemd160 :: U8^20

alias BlockId :: Sha256d
alias Txid :: Sha256d

alias Amount :: U64

data OutPoint :: Txid, vout U16

data ScriptPubkey :: U8^..10_000
data StackByteStr :: U8^..520
data Witness :: StackByteStr^..1_000
data SigScript :: StackByteStr^..1_000

data TxOut :: Amount,
              ScriptPubkey
data TxIn :: OutPoint,
             nSeq U16,
             SigScript,
             Witness?

data Transaction :: ver U8,
                    inputs TxIn*,
                    outputs TxOut*,
                    lockTime U32
```

Collections

- **Lists**: just specify size restrictions for a list after the type name
  - `*` for lists from 0 to 2^16 elements
  - `+` for lists from 1 to 2^16 elements
  - `?` for “lists” with either 0 or 1 element (*optionals*)
  - `^N..M` for lists which may contain from N to M elements (N or M may be omitted, defaulting to 0 and 2^16)
- Sets
- Maps

```
datapkg BP

alias Hash256 :: U8^32
alias Sha256 :: Hash256
alias Sha256d :: Hash256
alias Sha256t :: Hash256
alias Ripemd160 :: U8^20

alias BlockId :: Sha256d
alias Txid :: Sha256d

alias Amount :: U64

data OutPoint :: Txid, vout U16

data ScriptPubkey :: U8^..10_000
data StackByteStr :: U8^..520
data Witness :: StackByteStr^..1_000
data SigScript :: StackByteStr^..1_000

data TxOut :: Amount,
              ScriptPubkey
data TxIn :: OutPoint,
             nSeq U16,
             SigScript,
             Witness?

data Transaction :: ver U8,
                    inputs TxIn*,
                    outputs TxOut*,
                    lockTime U32

```

Collections

- **Lists**: just specify size restrictions for a list after the type name
- **Sets**: the same as lists, but the type name is enclosed into braces
  `data Seals :: {OutPoint}*`
  in sets, all members are unique and deterministically lexicographically ordered
- Maps

```
datapkg BP

alias Hash256 :: U8^32
alias Sha256 :: Hash256
alias Sha256d :: Hash256
alias Sha256t :: Hash256
alias Ripemd160 :: U8^20

alias BlockId :: Sha256d
alias Txid :: Sha256d

alias Amount :: U64

data OutPoint :: Txid, vout U16

data ScriptPubkey :: U8^..10_000
data StackByteStr :: U8^..520
data Witness :: StackByteStr^..1_000
data SigScript :: StackByteStr^..1_000

data TxOut :: Amount,
              ScriptPubkey
data TxIn :: OutPoint,
             nSeq U16,
             SigScript,
             Witness?

data Transaction :: ver U8,
                    inputs TxIn*,
                    outputs TxOut*,
                    lockTime U32
```

Collections

- **Lists**: just specify size restrictions for a list after the type name

- **Sets**: the same as lists, but the type name is enclosed into braces

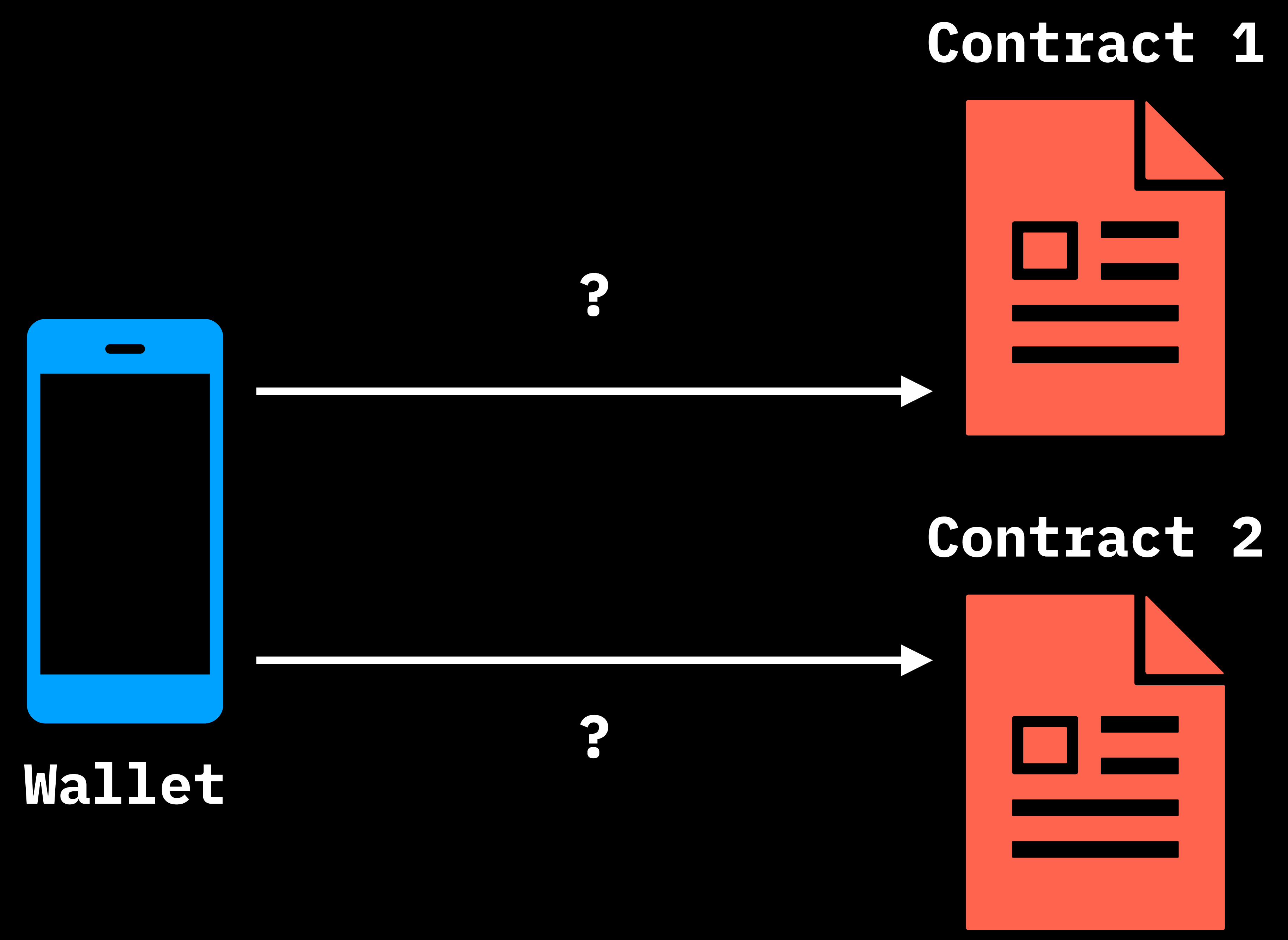
- **Maps**: key-value maps
  `data OwnedAmounts :: {OutPoint -> U64}*`
  keys in maps, like elements in sets, are unique and deterministically lexicographically ordered

```
datapkg BP

alias Hash256 :: U8^32
alias Sha256 :: Hash256
alias Sha256d :: Hash256
alias Sha256t :: Hash256
alias Ripemd160 :: U8^20

alias BlockId :: Sha256d
alias Txid :: Sha256d

alias Amount :: U64

data OutPoint :: Txid, vout U16

data ScriptPubkey :: U8^..10_000
data StackByteStr :: U8^..520
data Witness :: StackByteStr^..1_000
data SigScript :: StackByteStr^..1_000

data TxOut :: Amount,
              ScriptPubkey
data TxIn :: OutPoint,
             nSeq U16,
             SigScript,
             Witness?

data Transaction :: ver U8,
                    inputs TxIn*,
                    outputs TxOut*,
                    lockTime U32
```

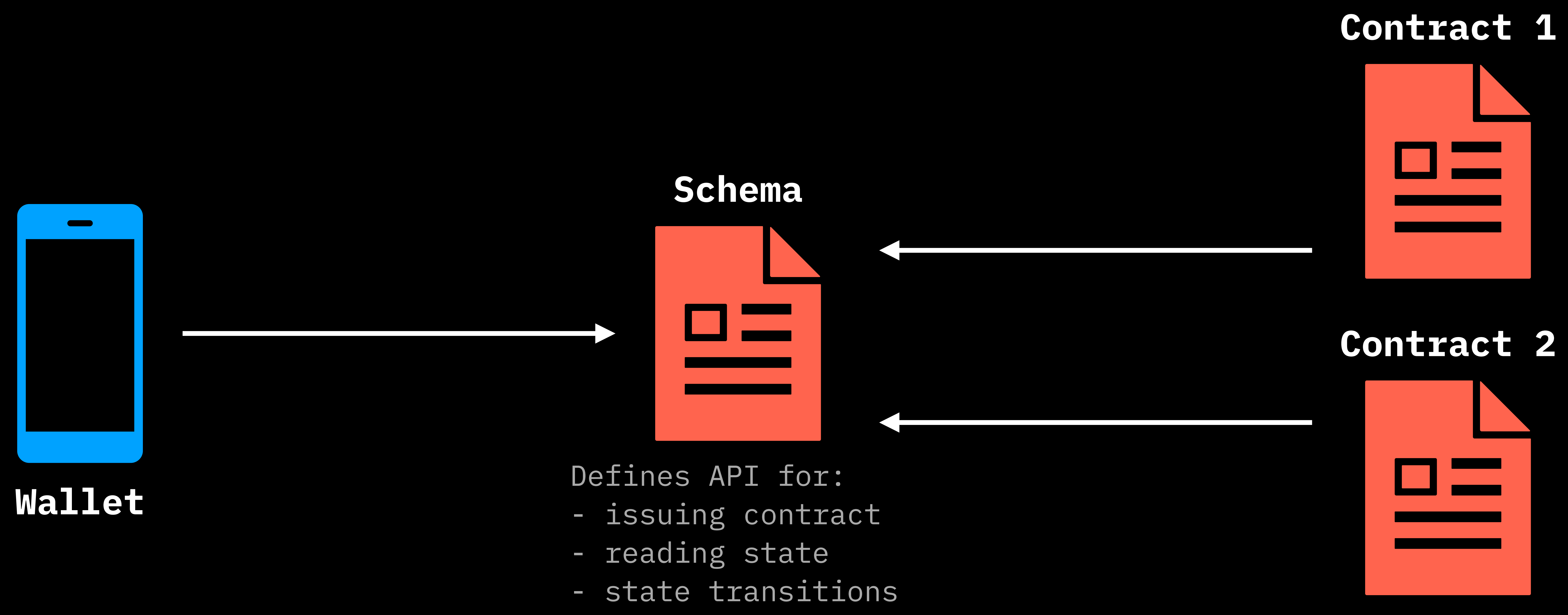
Ok, we have state data defined.

What’s next?

From state to contracts: Schema is in-between



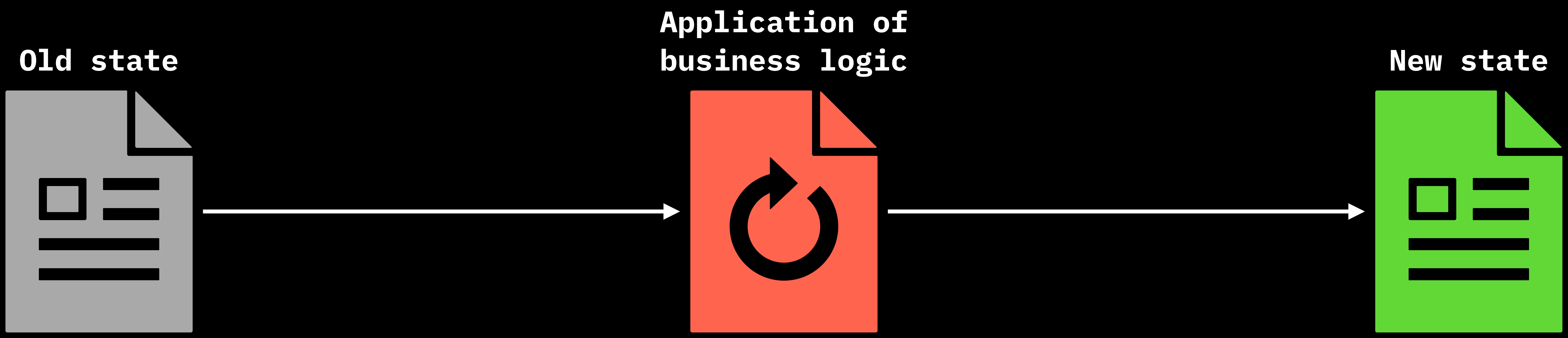
From state to contracts: Schema is in-between



RGB Schema

- Defines which specific state smart contract should have
  - of which data types
  - which state should be owned and unowned
  - and how it should be made confidential
- Defines how state should be validated
- Provides convenience methods for reading state aggregated data from the history
- Defines which state transitions are possible and how they should be validated ("business logic" of a smart contract)
- Embeds strict encoding schema on data types from the previous sections

State transitions: nodes of contract evolution DAG



State transition

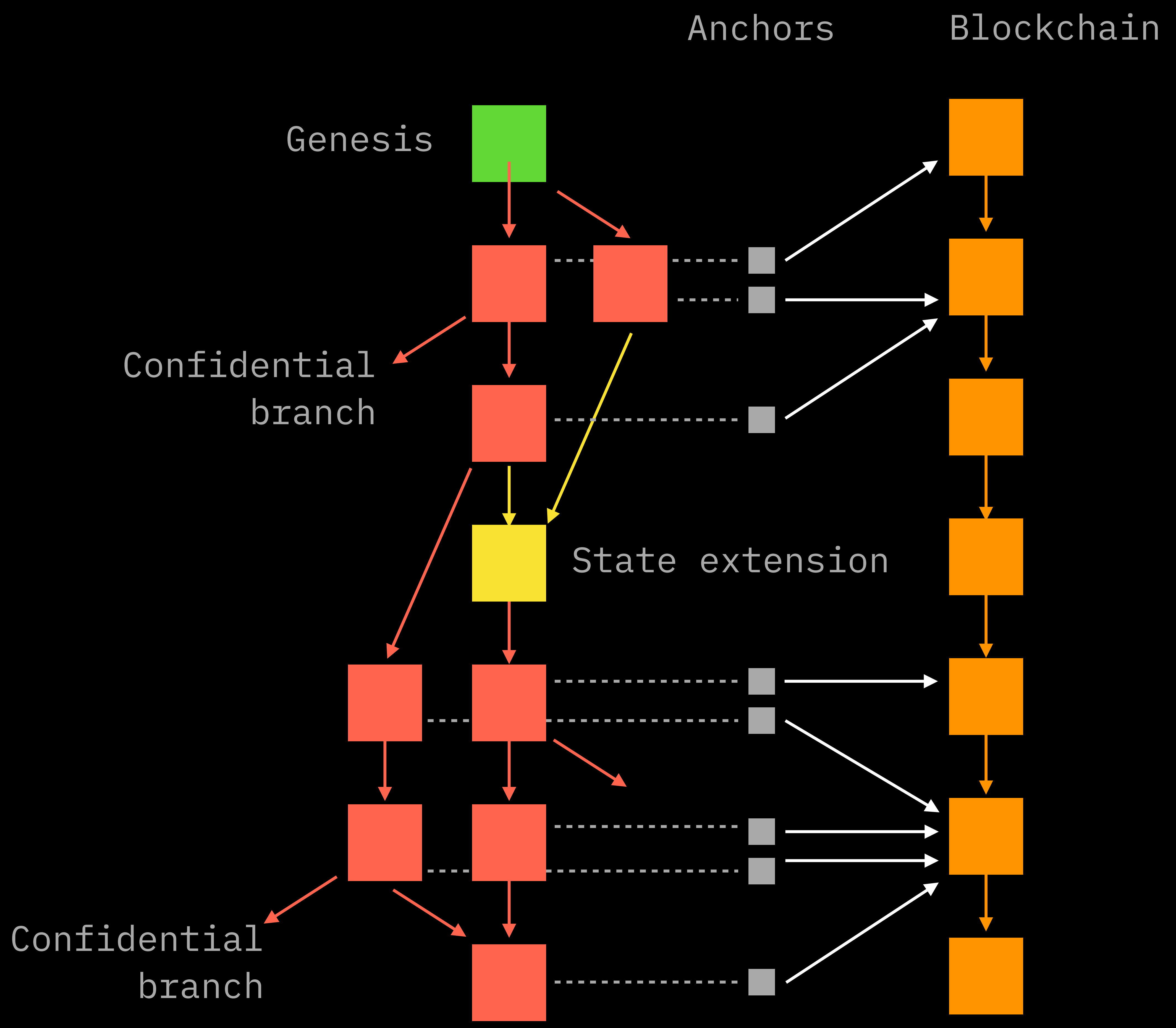
Forms of state transitions

- **Genesis**: from nothing to initial contract state
- “Normal” **state transition**: from owned and other types of state to a new owned state
- **State extension**: adds state to smart contract (owned and unowned) without discarding any existing state.

  State extensions opens smart contract for public participation (like decentralized issuance backed by some peg)

State transitions: nodes of contract DAG

- Each RGB contract is an isolated DAG shard
- Only tips of the DAG contains active state; the rest is history
- A single view on contract by a node is “partial shard” covering history from the known state up to the state genesis in all branches
- Anchors may link state transitions from many shards (but always one-per-shard) to a single blockchain



Schema with Contractum

Video demo

```
library FungibleAssets
    data AssetAmount :: U64

    abstract isCoin :: () -> Bool

    abstract totalIssued :: () -> AssetAmount
    abstract knownIssuance :: () -> AssetAmount

    method isShitcoin :: () -> Bool
        return isCoin

    method isTrustless :: () -> Bool
        return false


schema RGB20Simple using Assets
    field Ticker :: UTF8^3..8
    field Name :: UTF8^8..32
    field ContractText :: UTF8*
    field DecFractions :: U8 <- precision U8
        -- here we must validate the value range,
        -- so we are taking U8 value with `<-`
        -- and making sure it fits into our requirements
        assert precision in 0..=18
    field IssuedAmount :: AssetAmount

    -- if this was NFT, we can add something like
    -- container Painting :: image/png

    homomorph AssignedAmount AssetAmount
    right Renomination
    -- hashed accumulating Engraving -- used in NFTs

    genesis :: Ticker, Name, ContractText?, IssuedAmount, allocation AssignedAmount*, Renomination?
        assert sum! *allocation == issuedAmount

    transition transfer :: spent AssignedAmount+ -> sent AssignedAmount+
        assert sum! #spent == sum #sent

    transition renominate :: Renomination -> Renomination?, Ticker?, Name?, ContractText?
        assert ticker? or name? or contractText?

    method totalIssued :: () -> AssetAmount
        return @self.issuedAmount
```

Schema & contracts re-cap

State

- Unowned state:
  - **fields**: public unowned data
  - **valencies**: public rights to extend state
- Owned state: always confidential (assigned to single-use-seals):
  - **rights**: no state data (“void state”)
  - **homomorph**: homomorphically-encrypted state (only integer types are supported)
  - **hashed**: hash-encrypted state
- **methods**: state reading convenience functions

State transitions

- **genesis**: initial node and its validation
- **state transitions** (“normal”): can update owned state, always anchored to layer 1
- **state extensions**: can use valencies and extend contract state (owned and unowned), but can’t change previous state

State can be

- **Mutable**: each state transition *discards* previous state and assigns a new one

- **Accumulating**: each state transition *adds* to previous state a new state

Smart contract state

	Genesis	State extension	State transition
Adds fields *(metadata previously)*	+	+	+
Mutates fields	n/a	No	+
Adds owned state	+	+	+
Mutates owned state	n/a	No	+
Adds valencies	+	+	+

“+” means “if allowed by the used schema”

Sneak peak into RGB/2:
smart contract composability

- Single-use-seals can be defined not only as UTXO at layer 1, but also as a “unspent” state at RGB layer
- This opens door into smart contract composability

Caveats:

- A lot of research into security
- This “links” independent contract “shards”, providing increased requirements for client-side storage capacity

How smart contracts state & scripts are encoded: Strict Encoding

- Schema-based encoding for client-side-validation (used in RGB, AluRE)
- Deterministic ordering of elements in collections
- All data fields are byte-aligned
- Strict requirements to a number of items in any collection
- Best for creating cryptographic commitments to immutable data
- Described with a simple language,
  kept in binary form,
  able to auto-generate code for main existing languages:
  C, Rust, Go, Python, Kotlin, Swift, JavaScript

Why Strict Encoding and not protobuf, …

- Bounded data sizes allow high portability and determinism required for client-side-validation and AluVM
- Deterministic ordering of elements, required for creation of cryptographic commitments to the data in client-size-validation
- Native support for primitives widely used in cryptogrpahy (fixed-length byte arrays for hashes, keys

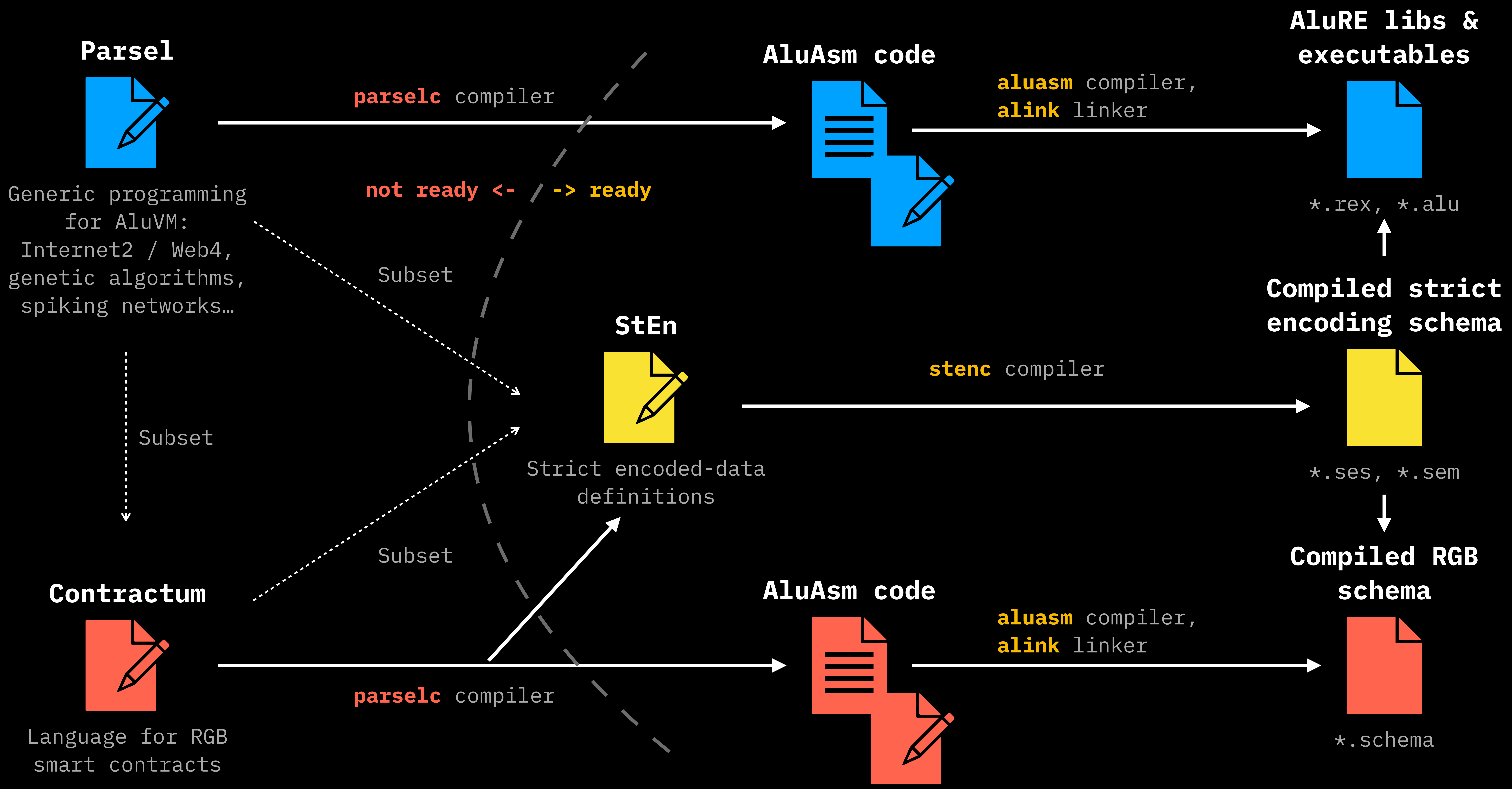
Disclaimer

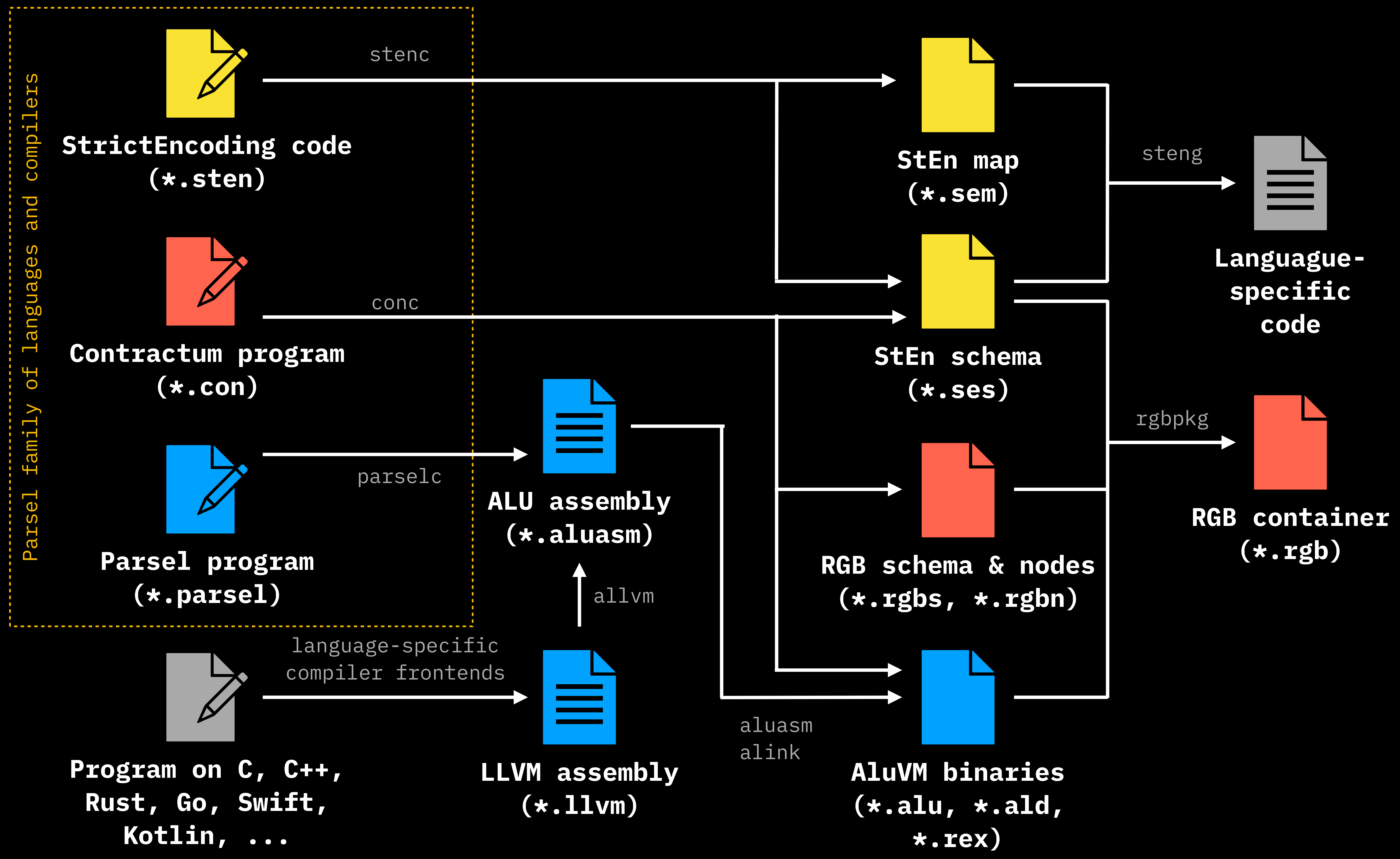
What we describe in this presentation regarding smart contract language Contractum is a preview of how RGB programming will work in the future _after_ RGB release.

To program the RGB (*once its released*) before Contractum, one need to use AluVM assembly and parts of Contractum language working today (type definitions) + rust-based DSL languages, i.e. that is not that easy.

Today we collect feedback on the Contractum language so we can improve its design

Language family tree & toolchain related to RGB




Parsel family of languages and compilers
StrictEncoding code
(*.sten)
stenc
Contractum program
(*.con)
conc
Parsel program
(*.parsel)
parselc
Program on C, C++,
Rust, Go, Swift,
Kotlin, ...
language-specific
compiler frontends
LLVM assembly
(*.llvm)
allvm
ALU assembly
(*.aluasm)
aluasm
alink
StEn map
(*.sem)
StEn schema
(*.ses)
RGB schema & nodes
(*.rgbs, *.rgbn)
AluVM binaries
(*.alu, *.ald,
*.rex)
steng
Languague-
specific
code
rgbpkg
RGB container
(*.rgb)