RGBCON 0

LNP/BP Standards Association

Non-conference for client-validated state (RGB)

Organised by **LNP/BP Standards Association**, using donations from
Bitfinex & Tether, Fulgur Ventures, Poseidon Group

Structure

- 3 days
- 6 sections per day
- Q&A session after each section
- Workshops/brainstorm sessions during the sections
- Contributions and questions are invited
- Lunch break

1st day: Client-validated state paradigm. Commitment protocols.

2nd day: Single-use seals. State-related protocols. Zero knowledge.

3rd day: Lightning Network. Spectrum. LNP/BP architecture.

Each day include:

- *code samples and their analysis*
- *introduction of the core cryptography concepts*
- *review of new and future developments in Bitcoin and Lightning Network*

About me

- *Secretary* @ **LNP/BP Standards Association**
- *Chief Engineering Officer* @ **Pandora Core AG**
- Leading developer of **RGB & Spectrum** since July 2019
- Author of **The #FreeAI Manifesto** manifesto.ai,
  cypherpunk & cryptoanarchist
- Neuroscientist (PhD)
- Machine learning, complexity science
- E-mail: **dr.orlovsky@me.com**
- Twitter: **@dr_orlovsky**
- GitHub: **dr-orlovsky**
- IRC: **dr-orlovsky** (#bitcoin-wizards, #lnp-bp, #rust-bitcoin)

LNP/BP Standards Association

LNP/BP Standards: **https://github.com/lnp-bp/lnpbps**

- RGB & Spectrum (digital assets & DEX): https://github.com/rgb-org
- Storm (storage & messaging): https://github.com/storm-org/storm-spec
- Prometheus (high-load computing): https://github.com/pandoracore/prometheus-spec

Pandora Core AG, works on implementation of these technologies, aiming to bring scalability and trustlessness into distributed storage and computing

Other participants of these protocols development & sponsorship include:
Giacomo Zucco, Peter Todd, John Carvalho, inbitcoin, Chainside, Bitfinex, Poseidon Group, Fulgur Ventures, Hyperdivision and many others

We are welcoming you to join the work on these projects! IRC **#lnp-bp** @FreeNode

LNP/BP Standards Association

Day I: RGB Intro. Commitment protocols

Dr Maxim Orlovsky, CEO **Pandora Core**, Secretary of LNP/BP Standards Association

Part I: Overview

On blockchains and where we can move out of them – or “what we do and why?”

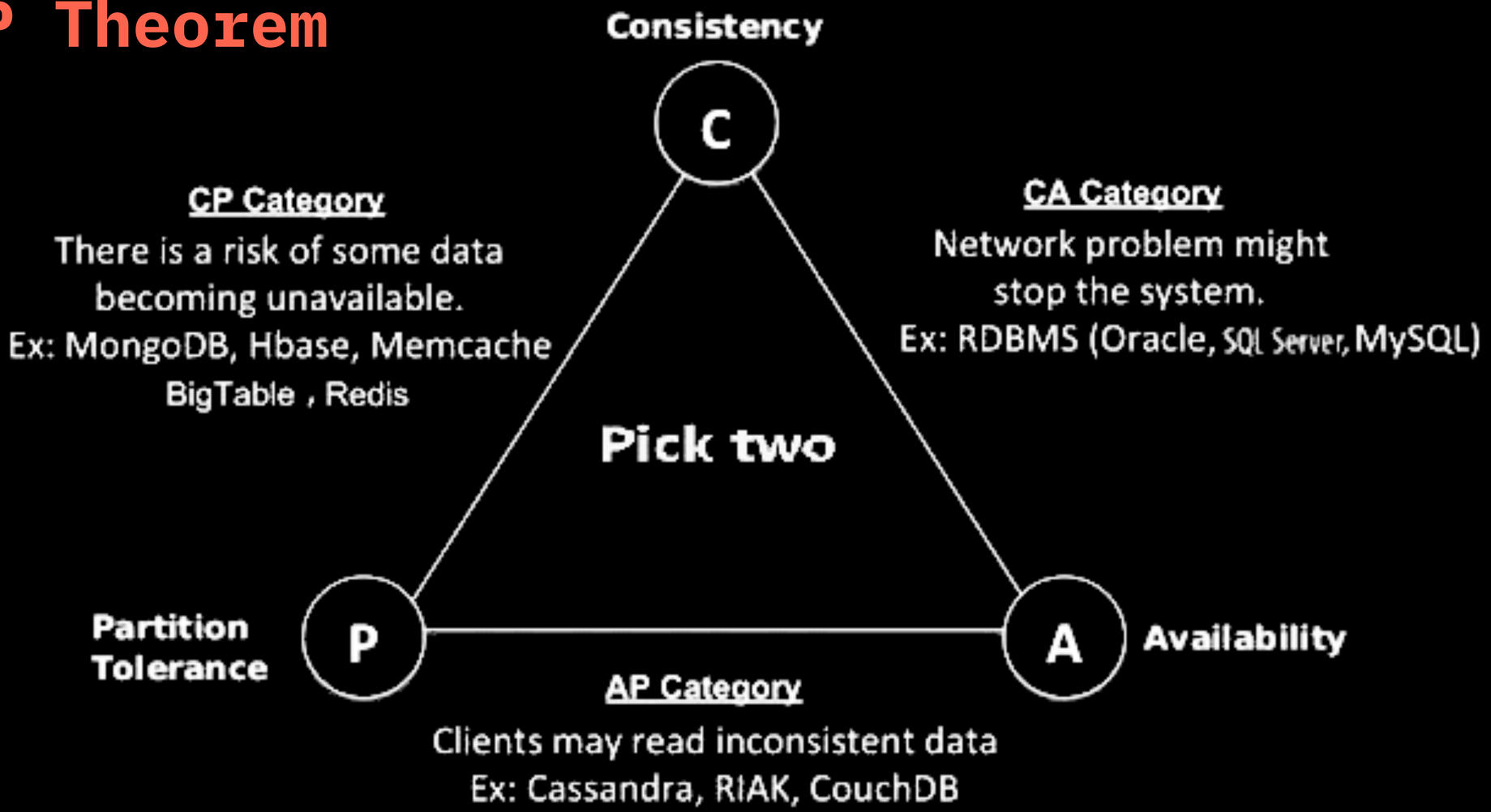
Before designing a protocol we need to understand

Why are we designing it?

Which tradeoffs we should select?

Why existing protocols do not satisfy us?

CAP Theorem


Consistency
C
CP Category
There is a risk of some data becoming unavailable.
Ex: MongoDB, Hbase, Memcache BigTable , Redis
CA Category
Network problem might stop the system.
Ex: RDBMS (Oracle, SQL Server, MySQL)
Pick two
Partition Tolerance
P
A
Availability
AP Category
Clients may read inconsistent data
Ex: Cassandra, RIAK, CouchDB

Why not to use existing DBs?

- Censorship

Who to solve the problem?

- Proof of work: permissionless decentralised system

Timechain (not blockchain)

A case of distributed (not necessarily decentralized) database with

- replicated state
- provable history of state transition
- consensus for the state

Unscalable. Low privacy.

State channels

A case of distributed database with

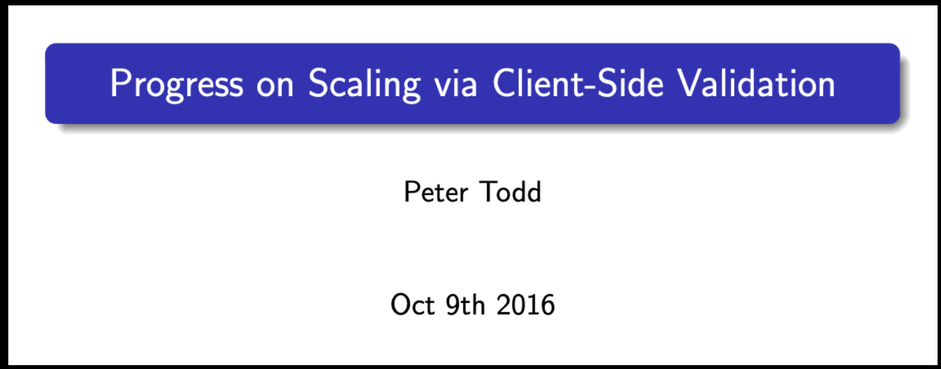
- replicated state
- over a fixed set of participants/replicas
- may not contain the history of state transition
- no consensus protocol

Scalable for internal state transitions (transaction)

Not scalable for the number of nodes and external coordinated state transitions

Intermediate privacy

Fixing blockchain



Client-side validation by Peter Todd:

Let’s put the actual data outside of bitcoin blockchain, since we already have a global consensus layer

https://scalingbitcoin.org/milan2016/presentations/D2 - A - Peter Todd.pdf

https://diyhpl.us/wiki/transcripts/scalingbitcoin/milan/client-side-validation/

Owned state *vs* replicated state

- Replicated state *vs* owned state
- Global consensus is required when we have a replicated state and need to share the same world view on a state across all of the agents
- With the owned state we do not need a global consensus;
  we need proofs of the state validity

Client-validated state (CVS)

A case of distributed database with

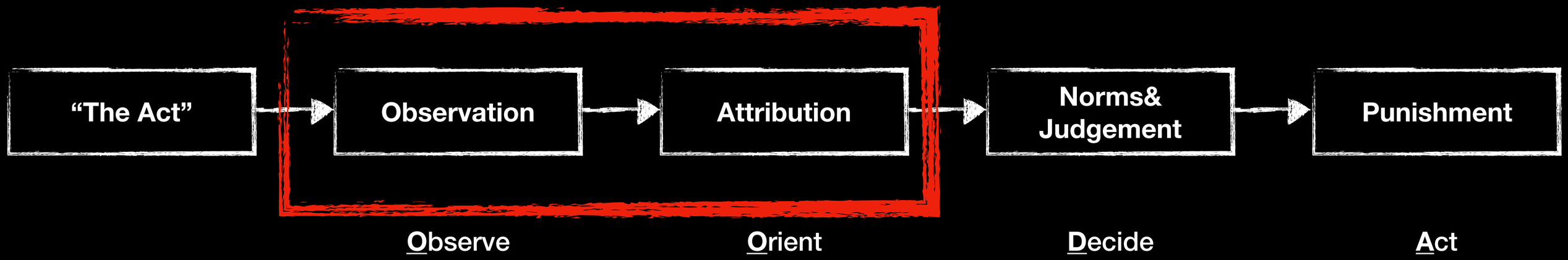
- no shared state (i.e. state is owned)
- full history of state transition (state proofs)
- no consensus protocol; validation protocols instead

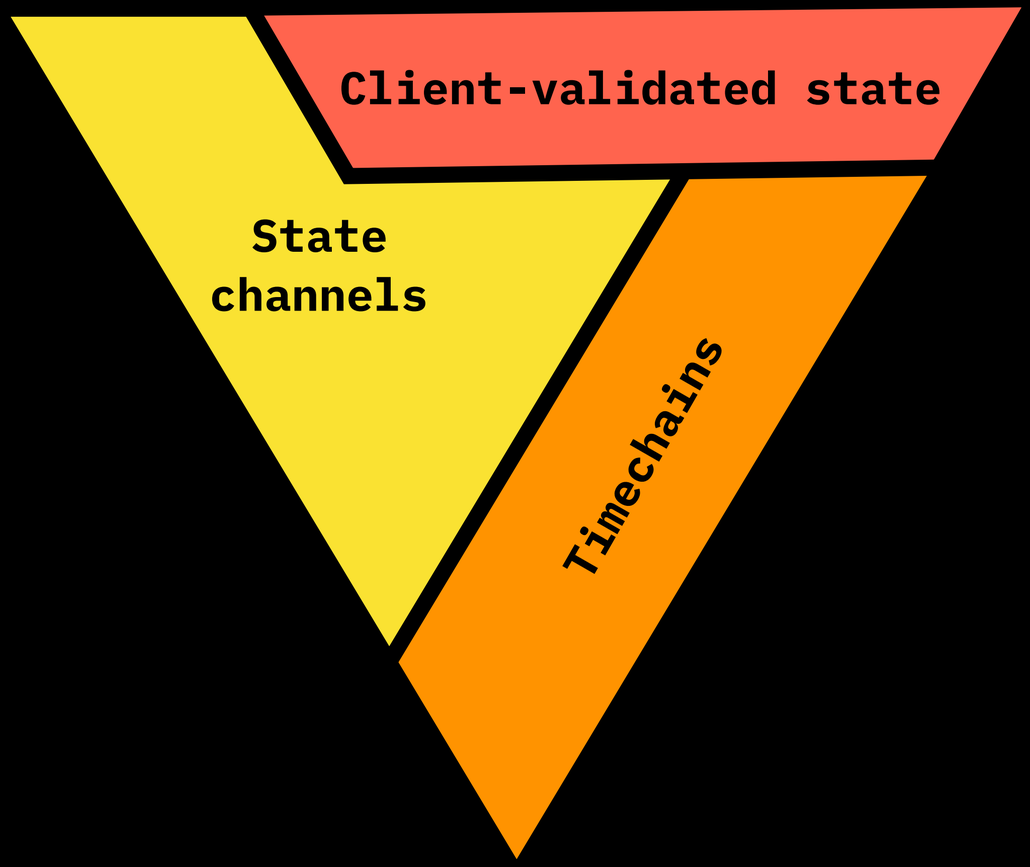
State update scalability depends on the underlying layers

Scalable in terms of space requirements comparing to blockchain

Best privacy across three

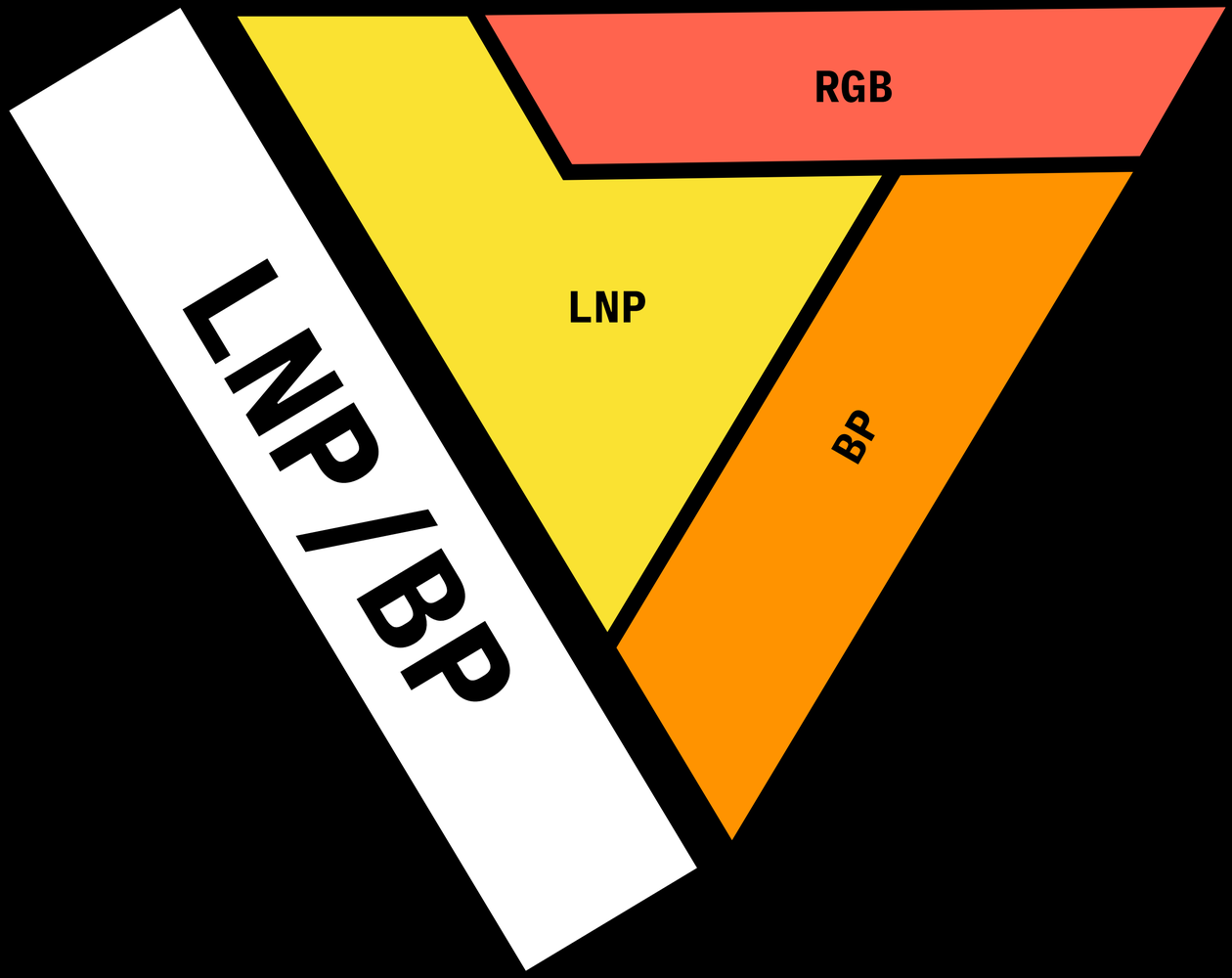
The loop of attribution

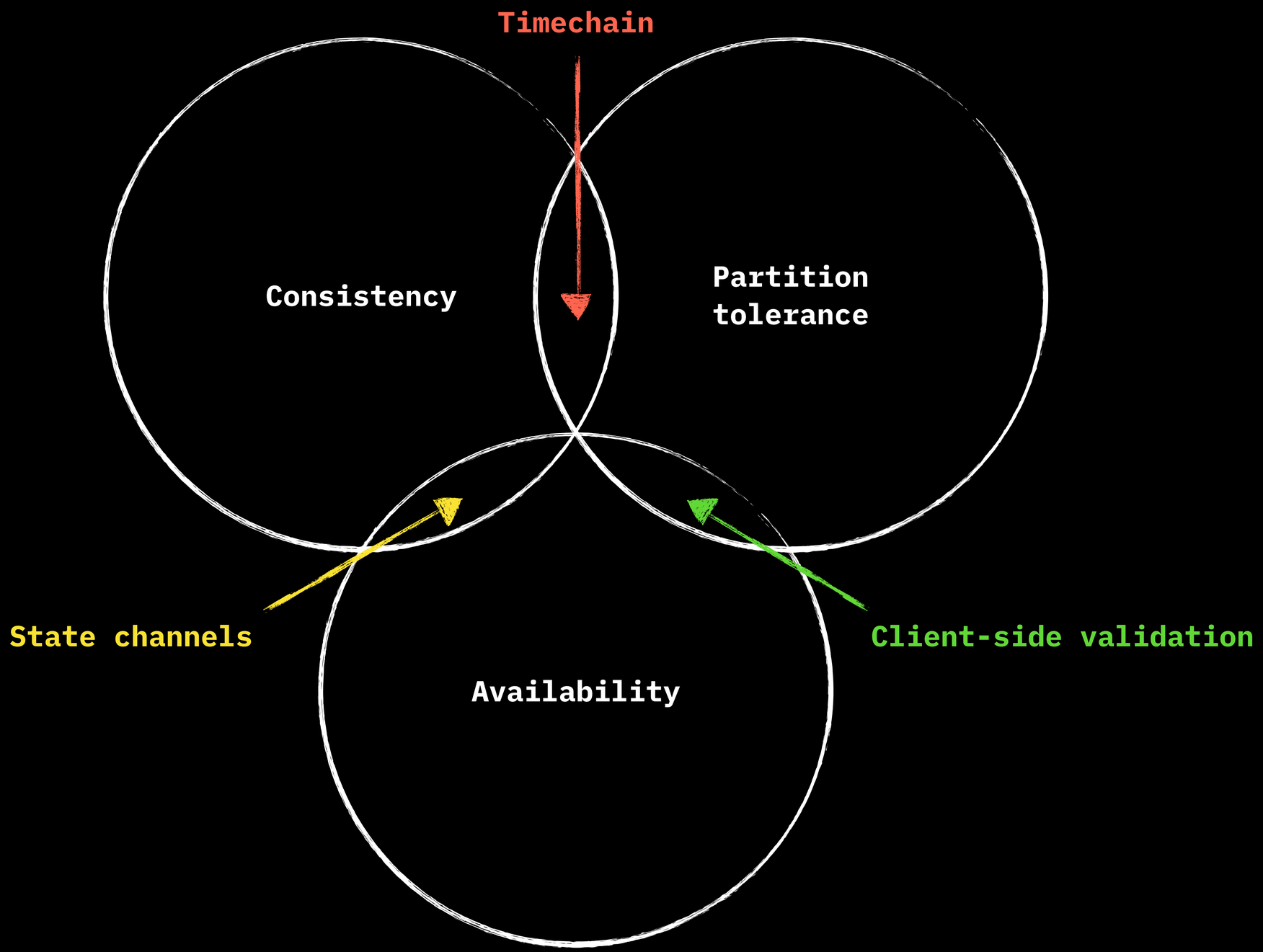



Client-validated state
State channels
Timechains

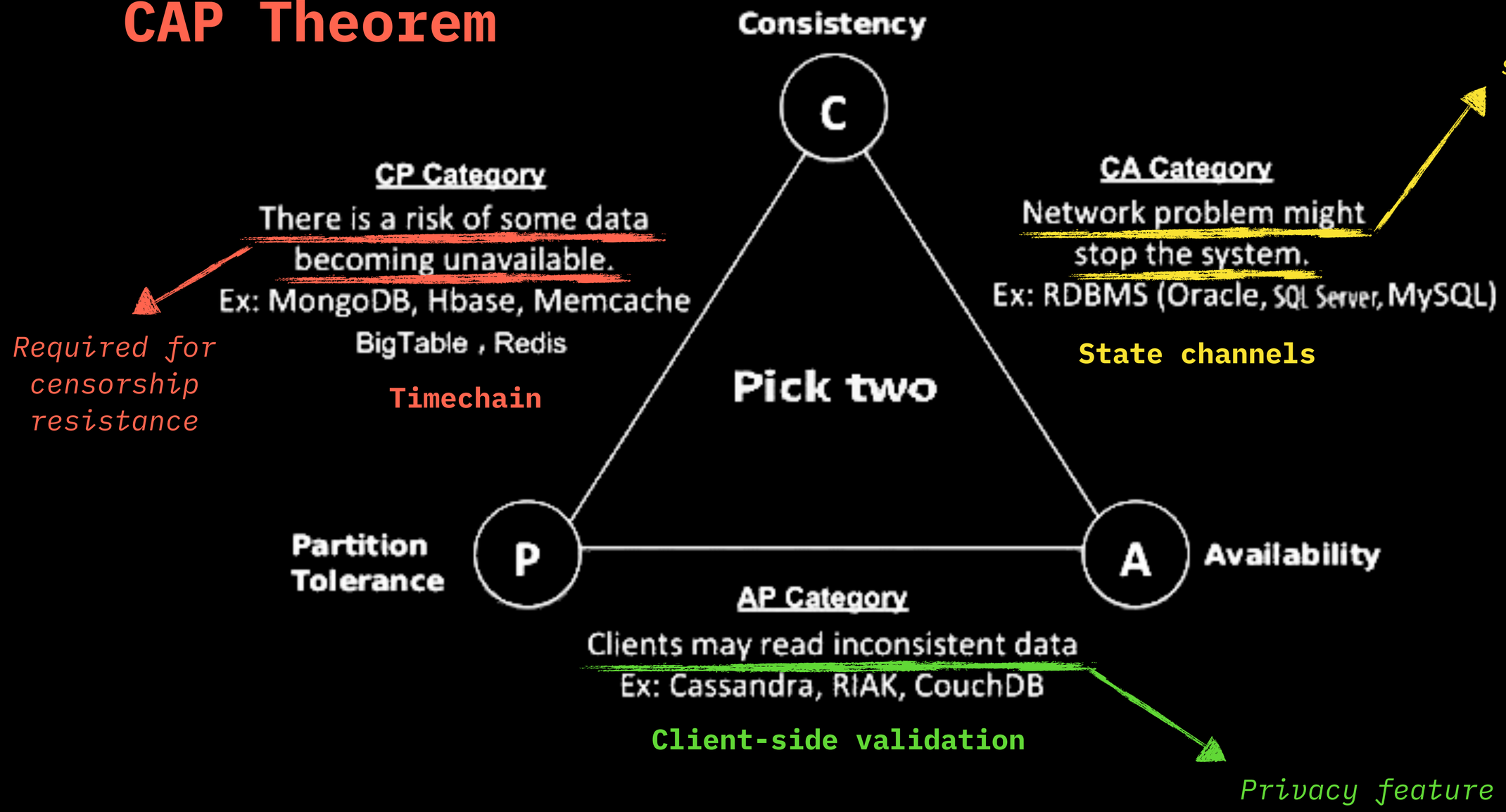
Who is who

- Timechain: Bitcoin protocol and bitcoin network
- State channels: Lightning network protocol and Lightning network
- Client-validated state: RGB and RGB network


RGB
LNP
BP
LNP/BP


Timechain
Consistency
Partition tolerance
State channels
Client-side validation
Availability

CAP Theorem



Price of scalability

Required for censorship resistance

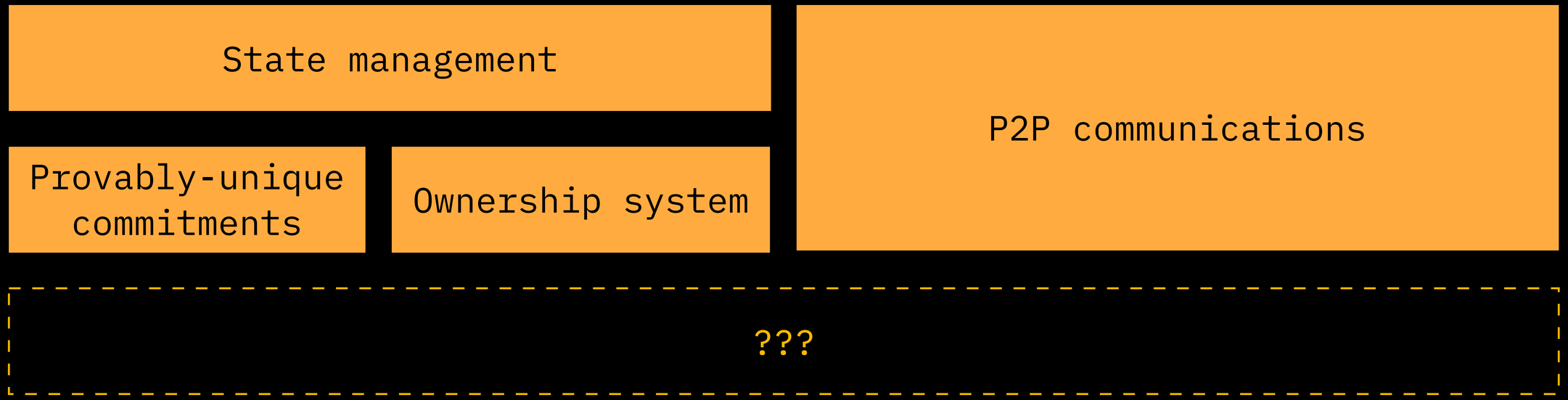
Privacy feature

Why we need CVS?

	Blockchain: ***Bitcoin protocol***	**State channels:** ***Lightning network protocol***	**Client-validated state:** ***RGB***
Censorship resistance	Strong	Strong	Strong
Privacy	Poor	Intermediate	Best (even better with zk)
Scalability	Low	Relatively high	Depends on the layer underneath

Part II: Layering CVS out

Client-validated state architecture



Cryptographic commitments

- Another name for Reveal-verify scheme
- Commit to some data (message) without revealing the message
- Prove that this message was the source for the commitment later, at reveal phase
- Cryptographic hash functions are the simplest form of the commitment

Provably-unique commitments

- **Provable:** The fact of the commitment can be proven publicly
- **Unique:** If the commitment is made, no other valid alternative state commitment can be created
- **Private:** Without the knowledge of some secret data the fact of the commitment can’t be determined
- **Deterministic:** for a given secret, the fact of the valid commitment can be automatically revealed and verified in a deterministic way

Provably-unique commitments and state

- PUC are used to define a state and build state transition history outside of blockchain
- Blockchain becomes a commitment layer; state channels – scalability layer
- Such state, unlike blockchain and state channel-based state, is *owned*
- Blockchain also defines the rules for ownership

Single-use seals: Peter’s Todd primitive for unique commitments

“Commitment to commit” (a promise of future commitment):

- Alice deterministically defines a point (*single-use seal*) at which a commitment to the future (non-existing yet) message will be made
- Alice commits to that point (“commitment to commit”)

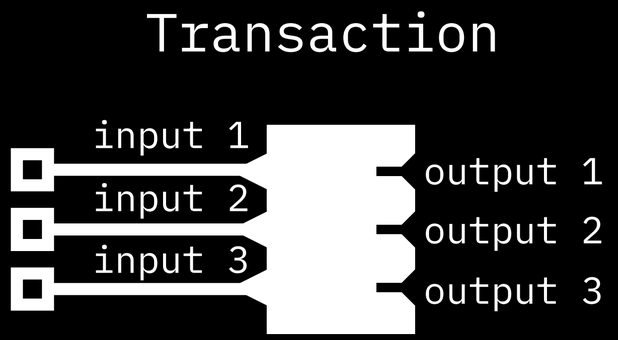
define

- Later on, Alice creates a message and commits to it at the defined point (*closes the seal over a message*)
- The fact of the message commitment can be proved by some private data block (*witness*)

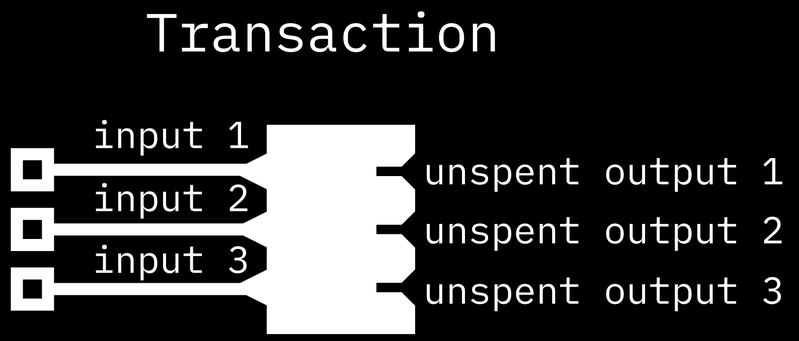
close

verify

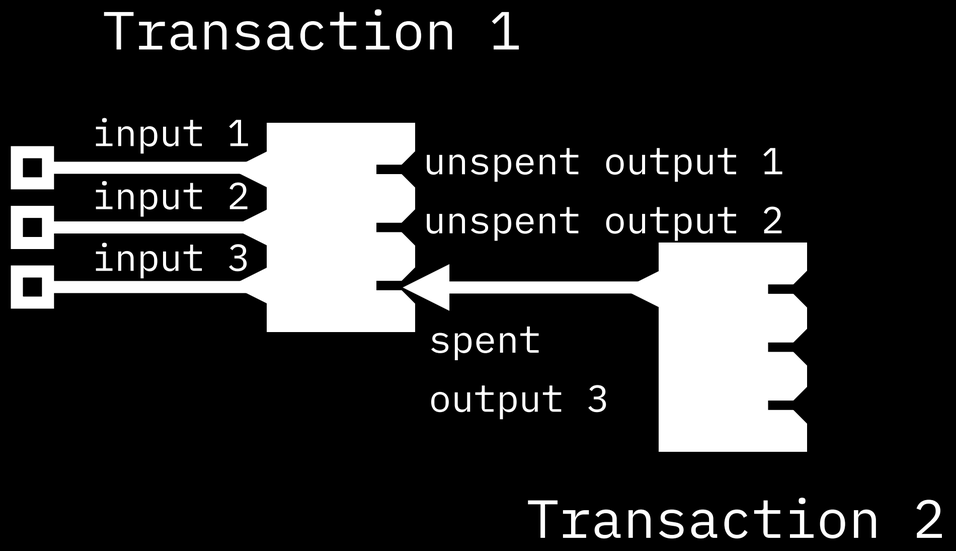
Bitcoin as single-use seal commitments medium



Bitcoin as single-use seal commitments medium

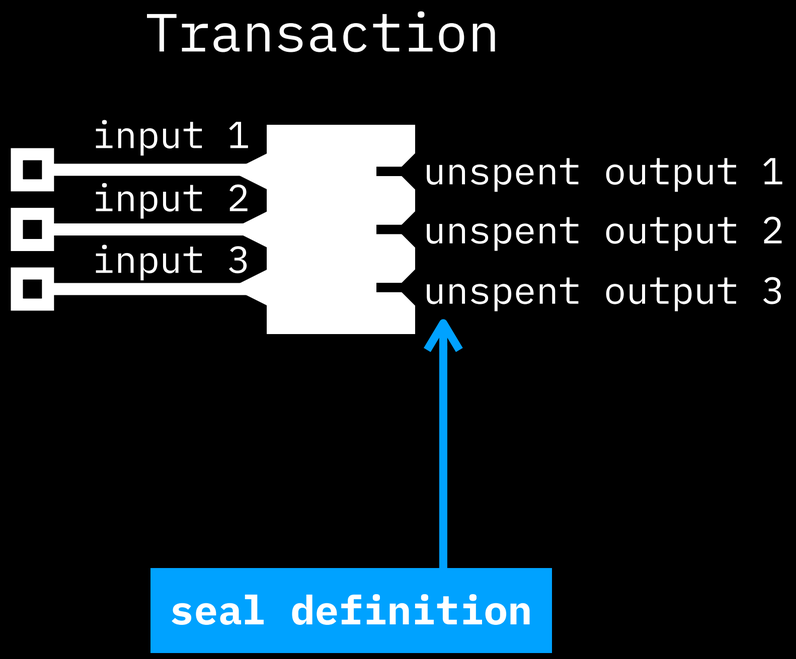


Bitcoin as single-use seal commitments medium

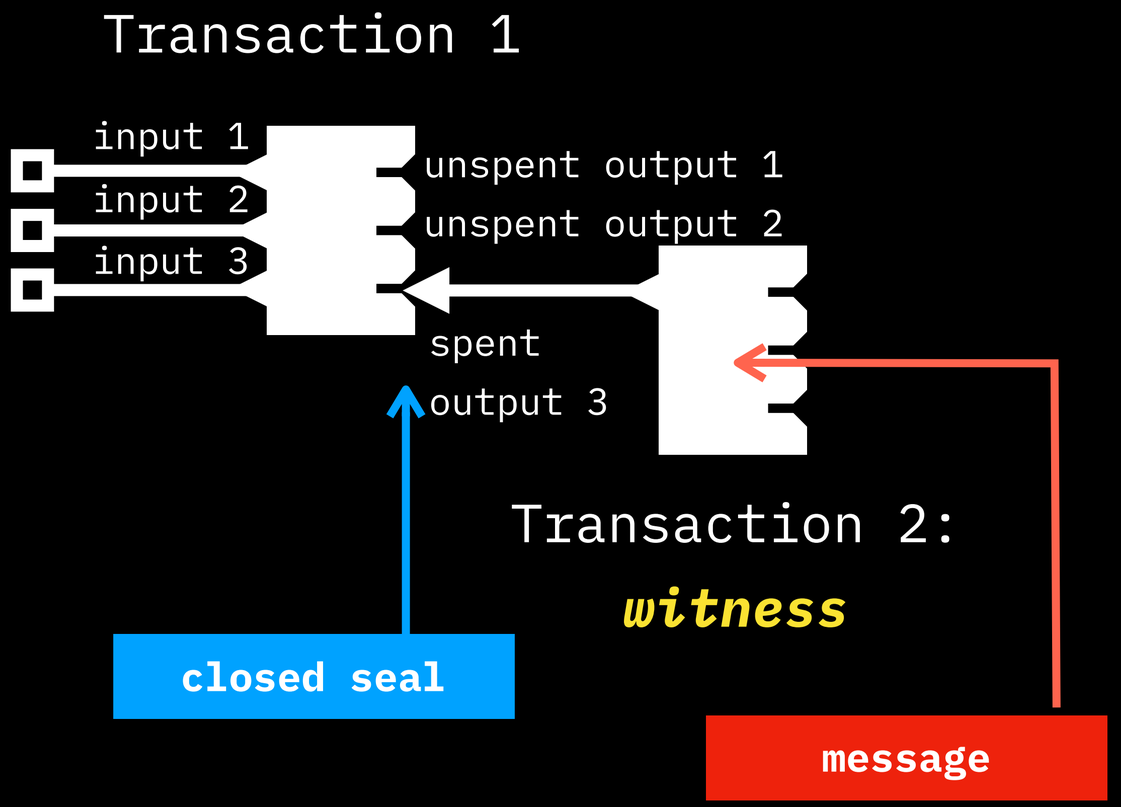


A unique event

Bitcoin as single-use seal commitments medium



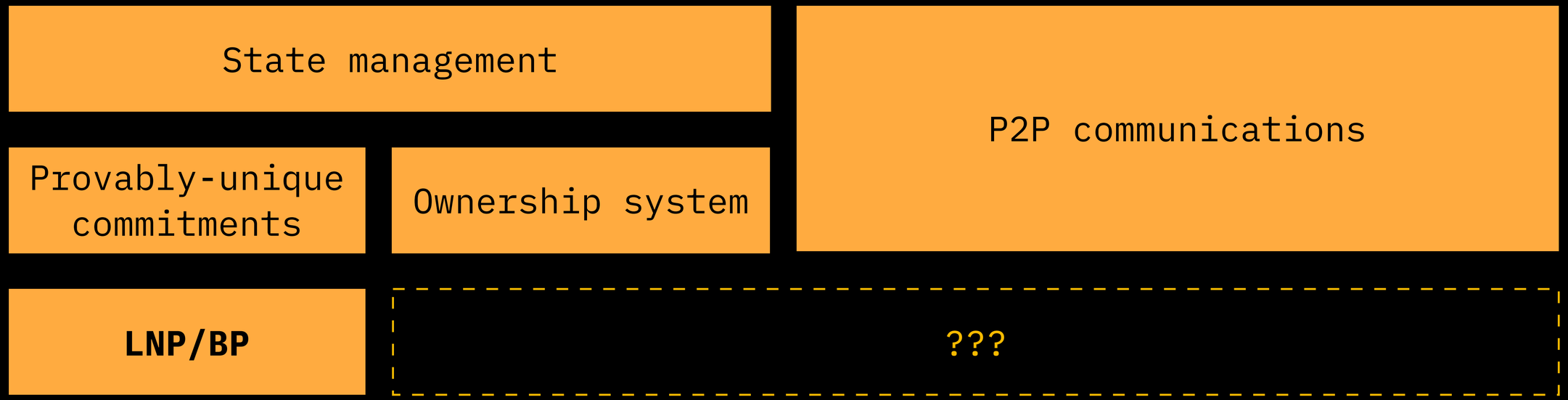
Bitcoin as single-use seal commitments medium



Bitcoin as single-use seal commitments medium

	Define:	Close, verify:
Public *(onchain or a part of state channel)*:	**Seal: Tx OutPoint**	**Witness: spending tx**
Private *(offchain, owned)*:	**Seal definition data**	**Message**

Client-validated state architecture



Part III: Layering CVS out

Challenges

- How we can commit to the seal definition?
- How we can put commitment to the message into the spending transaction (witness)?
- ... and make it such that no double-commitment is possible
- ... and at the same time preserve privacy properties for client-validated state
- ... maintaining Lightning network tx structure compatibility and future Taproot compatibility

Solving the challenges

- **Privacy:** embed all commitments into public key using EC homomorphic properties (“public key tweaking”)
- **Uniqueness:** find a deterministic way to define where the tweaked public key with the commitment resides inside the given transaction
- **Interoperability:** make this working for any possible tx outputs, including P2WSH (Lightning network), OP_RETURN (hardware wallets) and P2TR (future LNP/BP after Taproot)

LNPBP Standards

- Layered with proper abstractions: separation of concerns
  - Upgradability
  - Security
  - Interoperability
  - Code sharing
- Peer-reviewed

LNP/BP Standards (outside of BIP scope)

GitHub, Inc. (US) | https://github.com/lnp-bp/lnpbps

LNP-BP / lnpbps

Unwatch 2 | Unstar 3 | Fork 0

Code | Issues 2 | Pull requests 0 | Projects 0 | Wiki | Security | Insights | Settings

LNP/BP Specifications

Edit

bitcoin | lightning-network | decentralization | distributed-systems | privacy | cryptography | Manage topics

3 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

dr-orlovsky LNPBPs-0001 (early draft): Cryptographic commitments with public key ... Latest commit 55b573c 17 days ago

assets	README: Project description, inclusion criteria, layers and initial l...	17 days ago
.gitignore	README: Project description, inclusion criteria, layers and initial l...	17 days ago
README.md	LNPBPs-0001 (early draft): Cryptographic commitments with public key ...	17 days ago
lnpbps-0001.md	LNPBPs-0001 (early draft): Cryptographic commitments with public key ...	17 days ago

README.md

LNP/BP Specifications

LNP/BP stands for "Bitcoin Protocol / Lightning Network Protocol". This set of specifications covers standards & best practices for Layer 2, 3 solutions (and above) in cases when they do not require soft- or hard-forks on the Bitcoin blockchain level and are not directly related to issues covered in Lightning Network RFCs (BOLTs).

Number	Layer	Field	Title	Owner	Type	Status
1	Transaction (1)	Cryptographic primitives	Cryptographic commitments with public key tweaking	n/a	Standard	Draft
2	Transaction DAG (2)	Client-side validation	Simple single-use seal for LNP/BP	n/a	Standard	Draft
3	Offchain data (3)	Consensus rules	State history directed acyclic graphs on Bitcoin	n/a	Standard	Draft
4	Offchain data (3)	Serialization	Serialization for state history DAGs, LNPBPs-4	n/a	Standard	Draft
5	Offchain metadata (4)	Serialization	Schemata for rich state	n/a	Standard	Draft
6	Application (5)	Assets	RGB, part 1: Fungible centrally-issued assets with client-side validation	n/a	Standard	Draft
7	Offchain data & metadata (3-4)	P2P messaging	State announcements for Lightning Network gossip protocol	n/a	Standard	Draft
8	Offchain data & metadata (3-4)	P2P messaging	State updates over Lightning Network onion messaging	n/a	Standard	Draft
9	Application (5)	DEX/DMP	Spectrum: decentralized market / exchange over Lightning Network	n/a	Standard	Draft
10	Offchain metadata (4)	Assets	RGB, part 2: Zero-knowledge proofs for asset transfers	n/a	Standard	Draft

Layering commitments

Embedding tweaked public key

Into Outpoint

Into Bitcoin Script

Public key tweaking

Verifying commitment:
which off chain data we have
to provide in order to verify?

Steps for building the commitments

- We have a repeated pattern of layered & nested commitments: leverage a generic API
- Define a deterministic & unique (collision-resistant) way to commit on each of the layers

API for commitments: draft

```rust
pub trait Commitment<MSG> where MSG: Committable<Self>
{
}

pub trait Committable<CMT> where CMT: Commitment<Self>
{
    fn commit(&self) -> CMT;
    fn verify(&self, commitment: &CMT) -> bool;
}
```

- We can commit to some message type under multiple commitment protocols
- A commitment protocol may create commitments to different types of messages

Problem

- How to embed commitments (in particular - public key tweaks)?
- This will be required for many situations: script embeddings, transaction outpoint embeddings and many other, which will be disclosed in future
- Do a “embeddable commitment” API!

API for commitments with embeddings

```rust
pub trait Verifiable<CMT>
    where CMT: Commitment
{
    fn verify(&self, commitment: &CMT) -> bool;
}

pub trait Committable<CMT>: Verifiable<CMT>
     where CMT: StandaloneCommitment<Self>
{
    fn commit(&self) -> CMT;
}

pub trait EmbedCommittable<CMT>: Verifiable<CMT>
     where CMT: EmbeddedCommitment<Self>
{
    fn commit_embed(&self, container: &CMT::Container) -> Result<CMT, CMT::Error>;
}
```

“The code is the best formal spec”

– Peter Todd

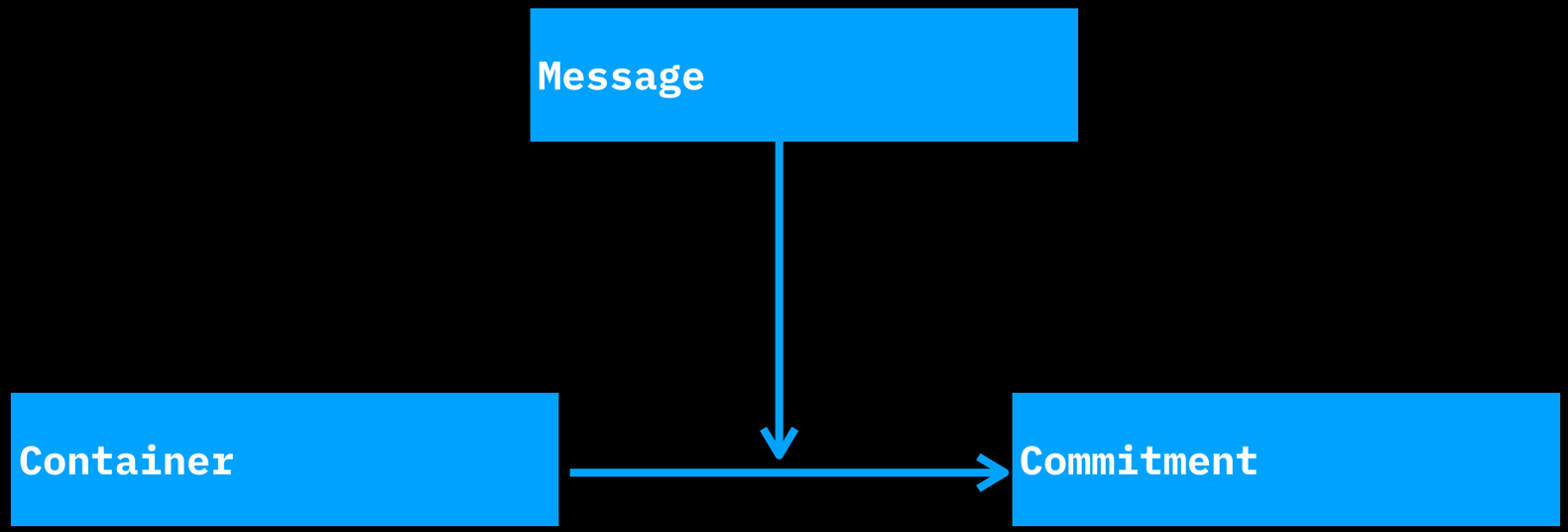
Rust language

- Deterministic execution of the compiled program
- No garbage collection
- Generics system most close to C++
- C interoperability (bindings for C, Python, Swift, Java/Kotlin, Go)
- Most advanced WASM toolchain

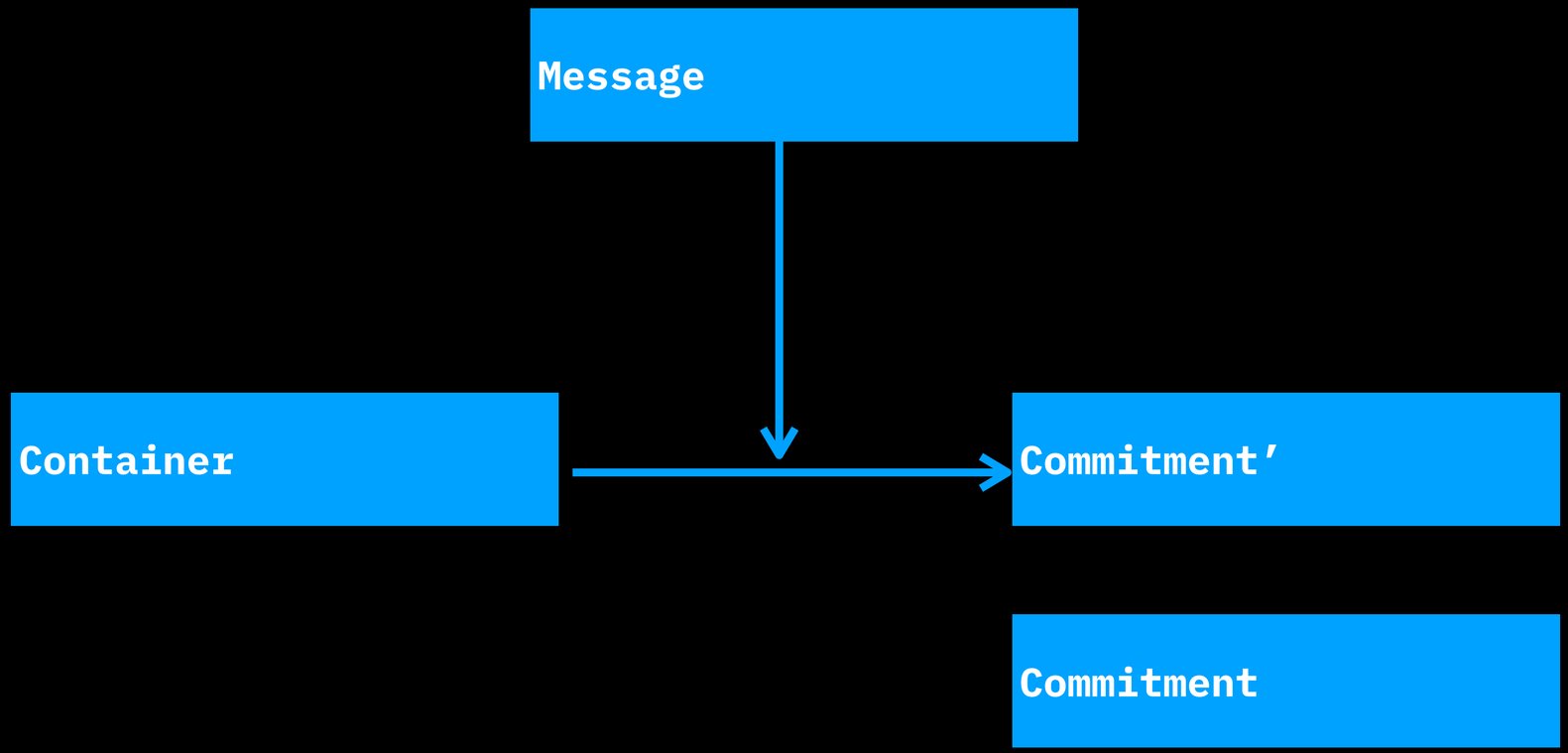
Rust language *vs* other

- C/C++: non-deterministic execution
- Go: garbage collection, poor generics
- Swift: garbage collection, poor generics
- Kotlin: JVM
- JavaScript: non-deterministic behaviour, not a formal language
- Python: non-deterministic behaviour, poor binary data support

Commit/embed

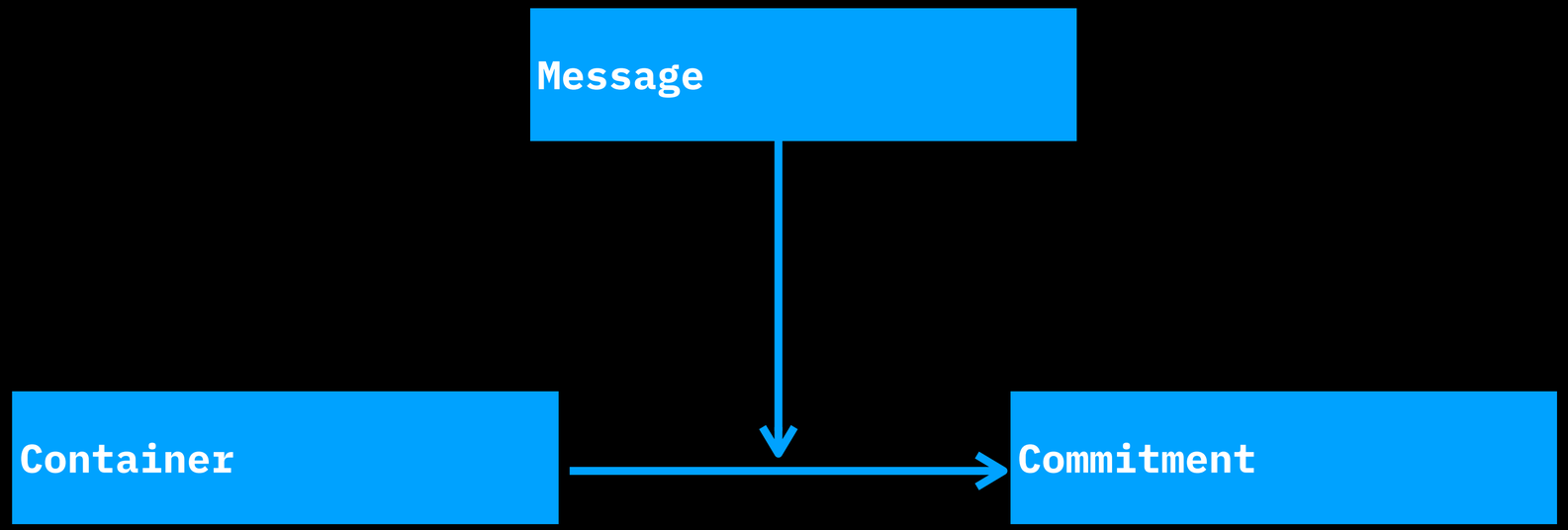


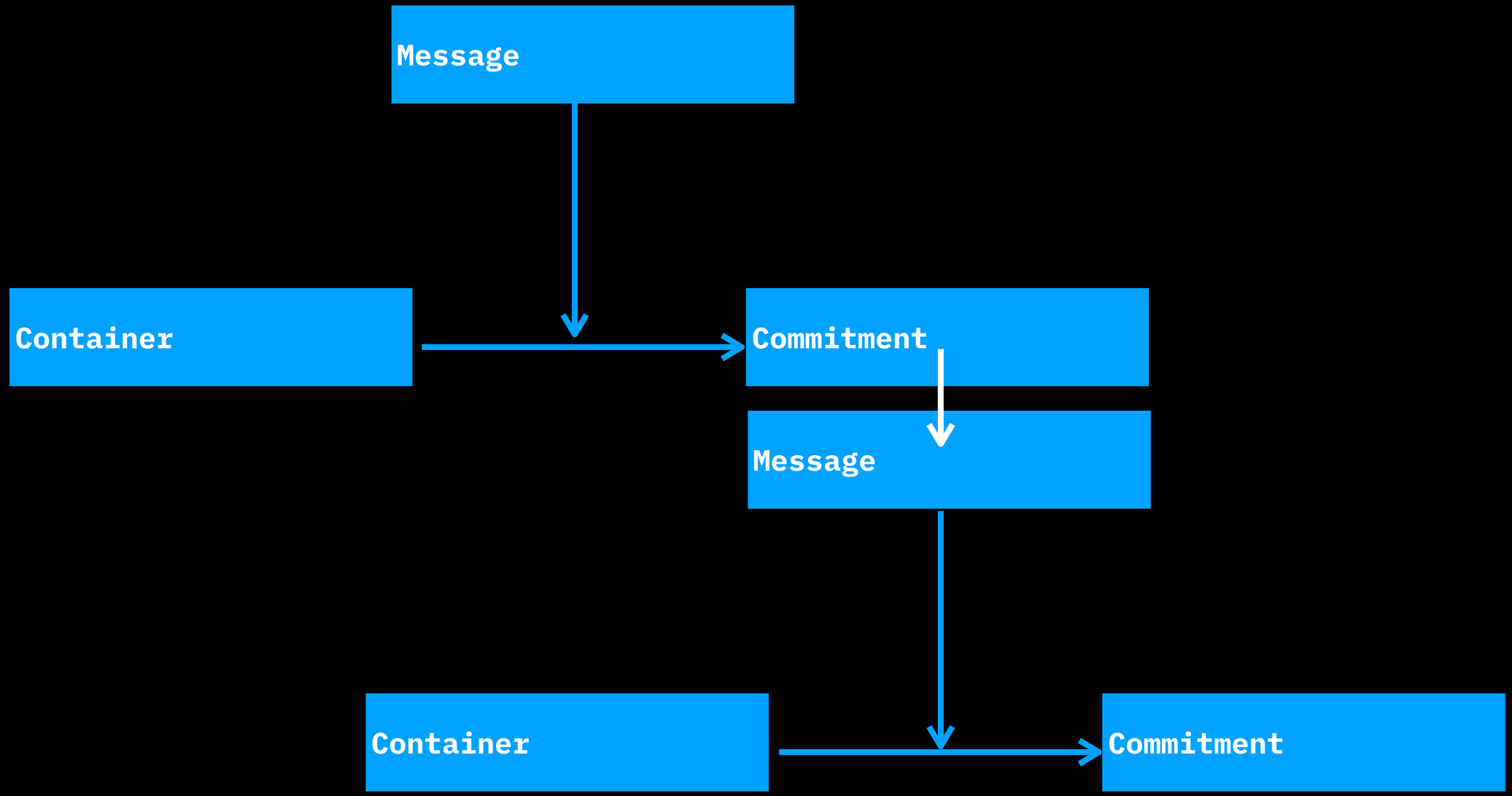
Verify

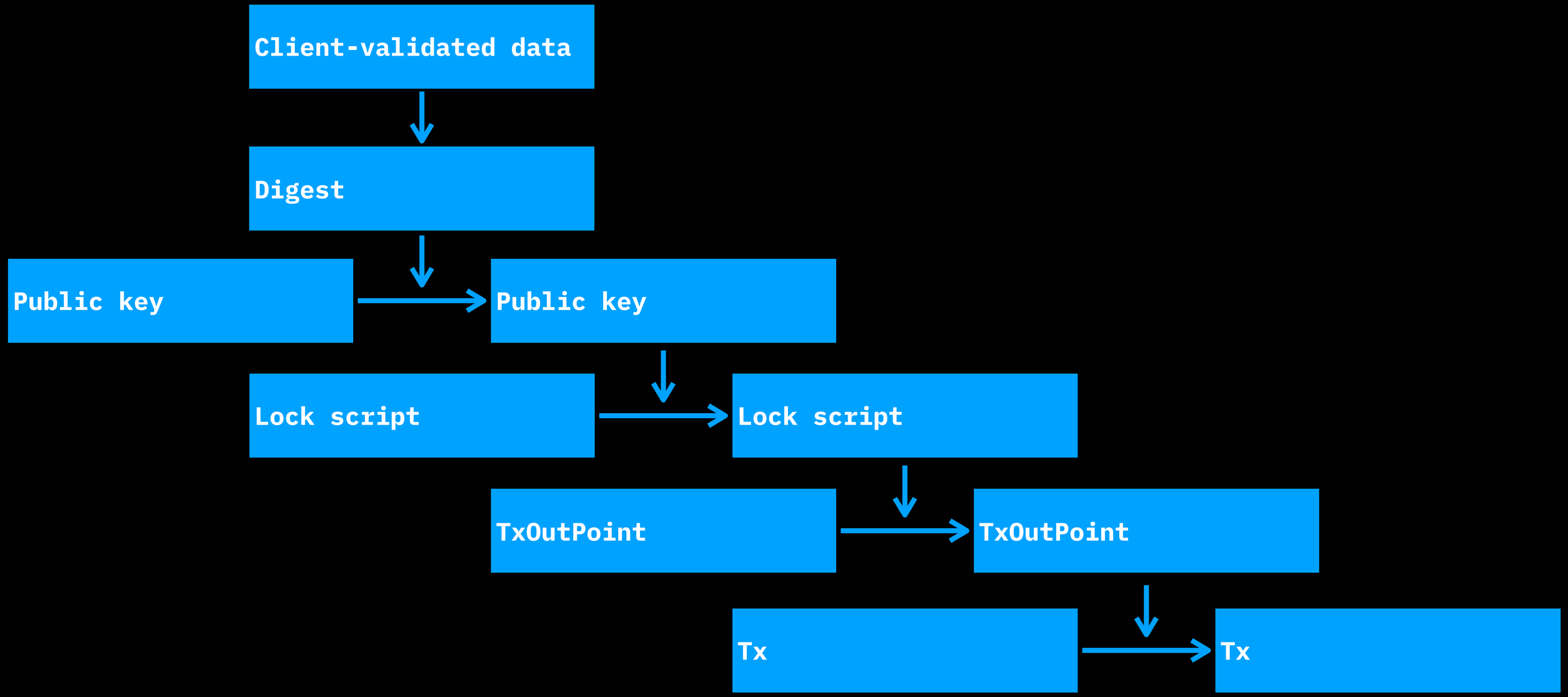


Private *(offchain, owned)*:

Public *(onchain or a part of state channel)*:




Message
Container
Commitment
Message
Container
Commitment


Client-validated data
Digest
Public key
Public key
Lock script
Lock script
TxOutPoint
TxOutPoint
Tx
Tx

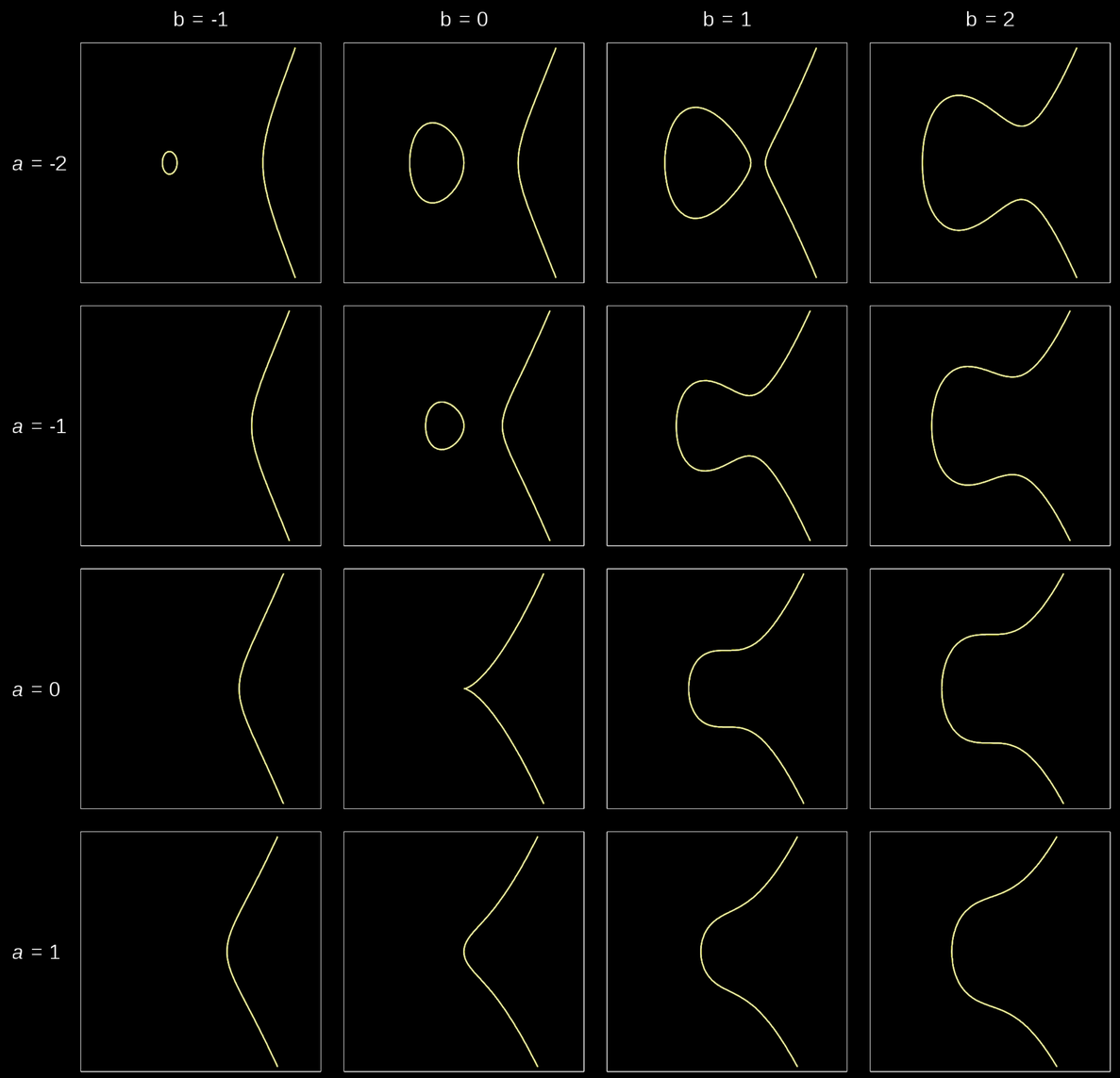
Well, it’s not that simple...

Part IV: Elliptic curve cryptography

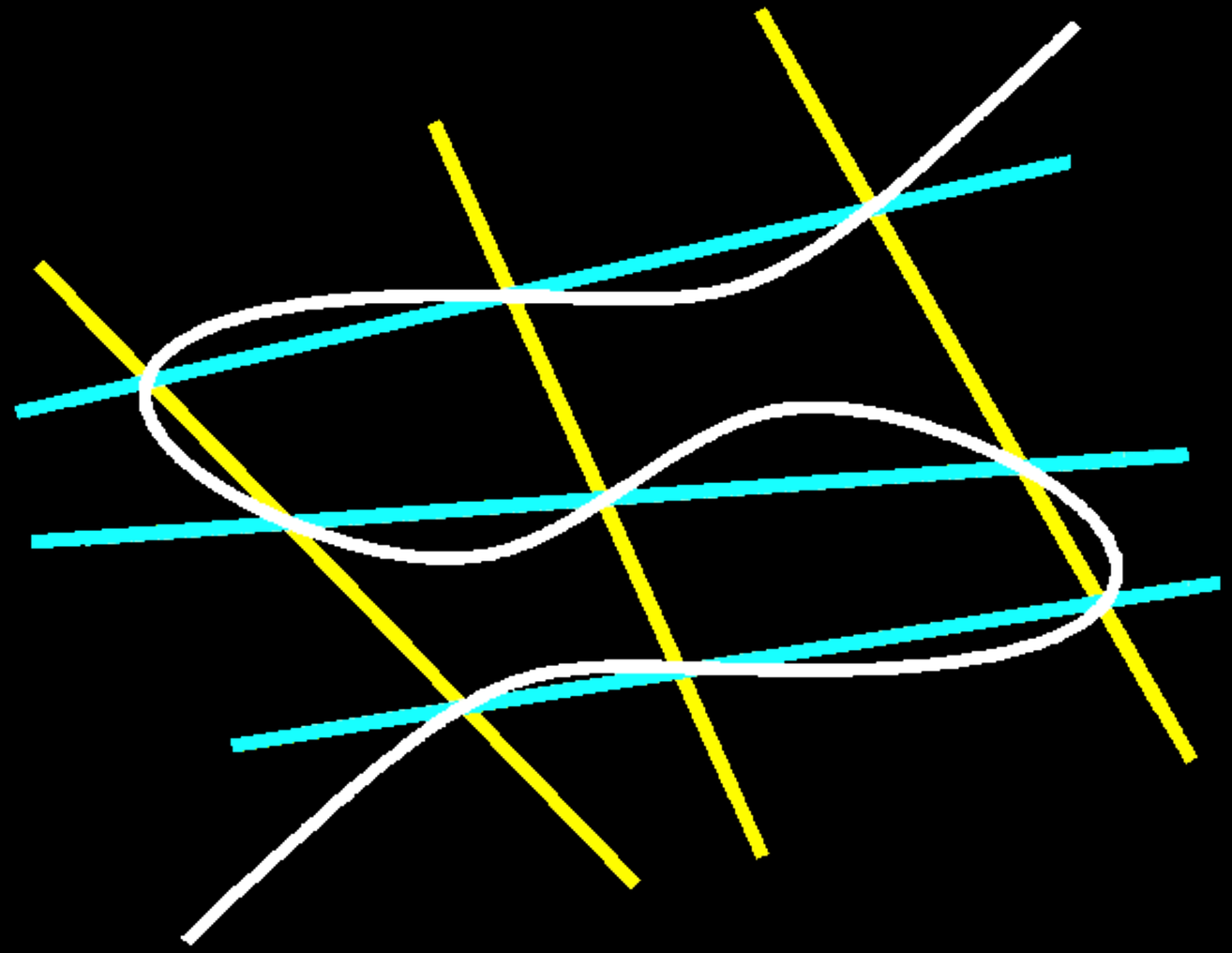
Elliptic curve

All pairs of (x, y) points that satisfy the equation

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



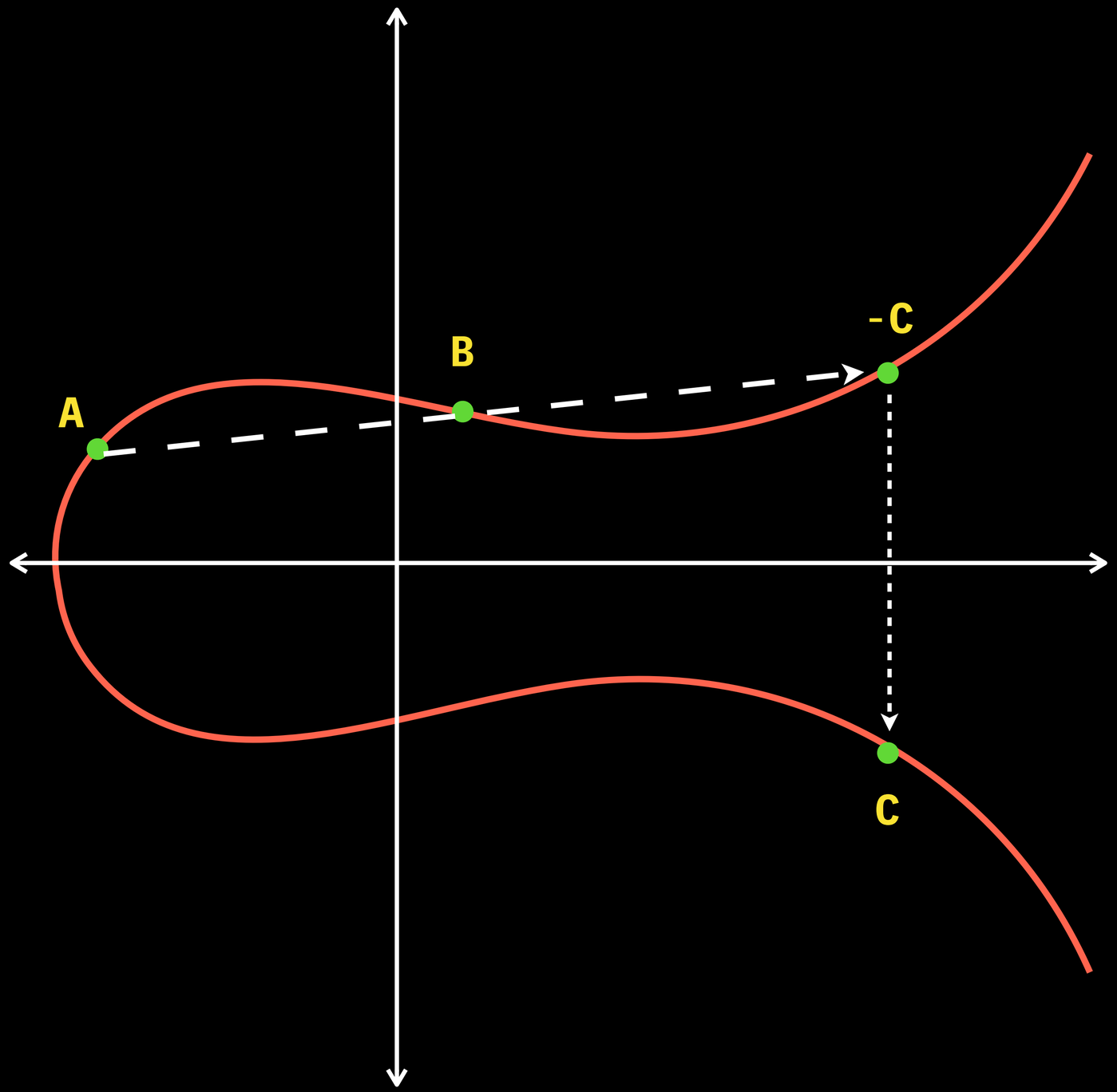
But why elliptic curve?



Cayley–Bacharach theorem

Elliptic curve arithmetics

- Points **A** = (x_g, y_g)
- Point addition: **C** = **A** + **B**
- Scalars ***h***
- Point multiplication on a scalar:
  C = **A** + **h** * **B**



Fields in math

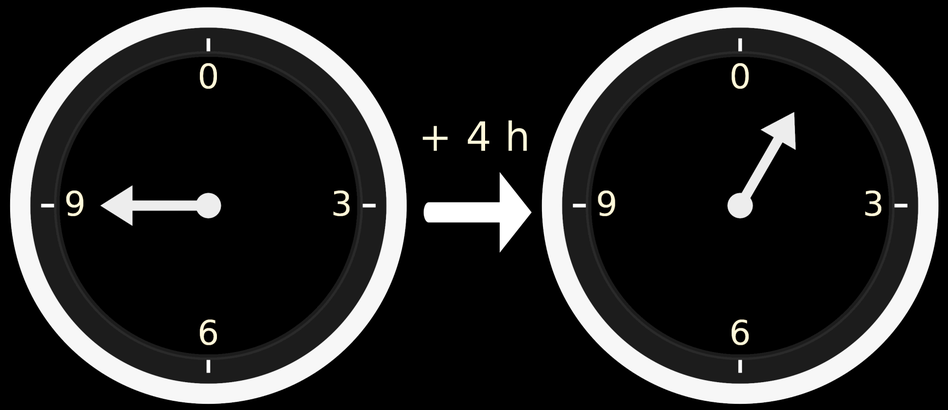
Field: a set with **+ − * /** ops defined working like the arithmetics on usual numbers

However, this operations MUST result in one of the element of the field

Hint: overflow with modulo division (**%** op)

- Finite fields vs infinite fields

- Discrete fields vs continuous fields



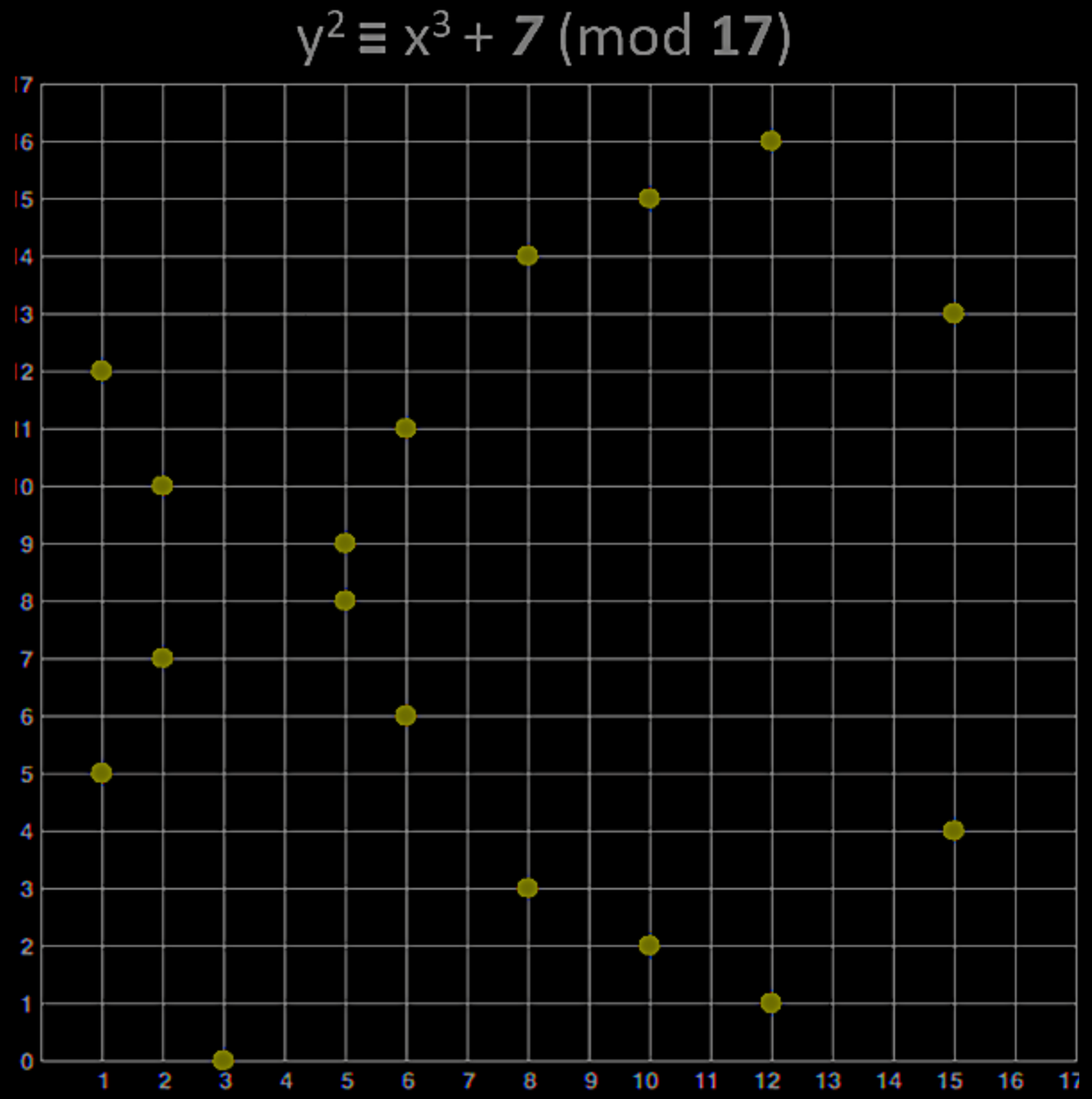
Prime order fields

- Galois field
- GF(p), F_p, Z_p, $\mathbb{F}_p$, $\mathbb{Z}_p$
- p is the order, i.e. number of elements in the field
- In prime order fields, the p is always a prime number
- The larger p, the more values we can have
- So we just need to pick the closest prime number to 2^{256}

Elliptic curve over prime order field

All pairs of (x, y) points from $\mathbb{Z}_p$ that satisfy the equation

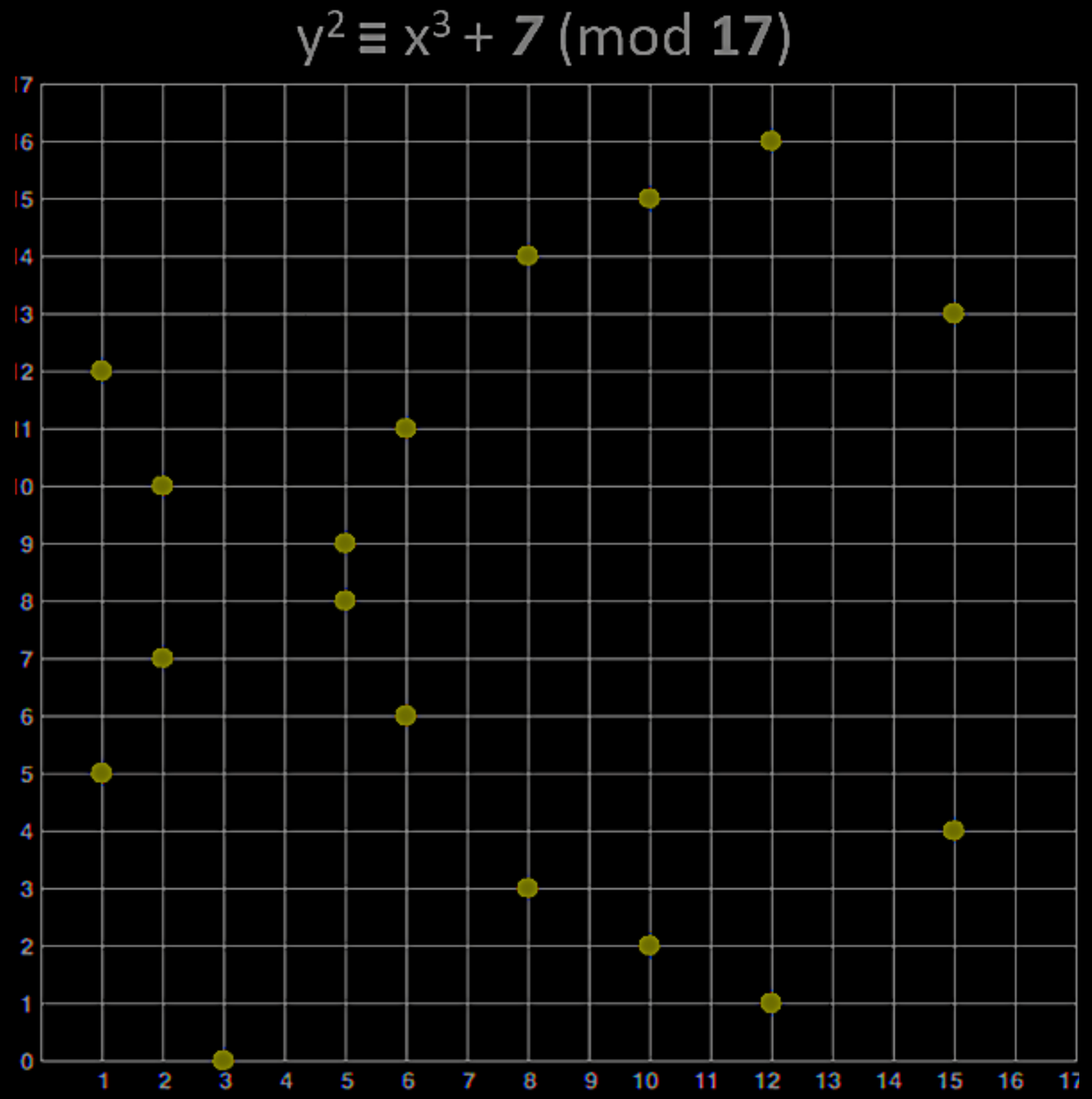
$$y^2 = x^3 + ax^2 + b$$



Elliptic curve parameters

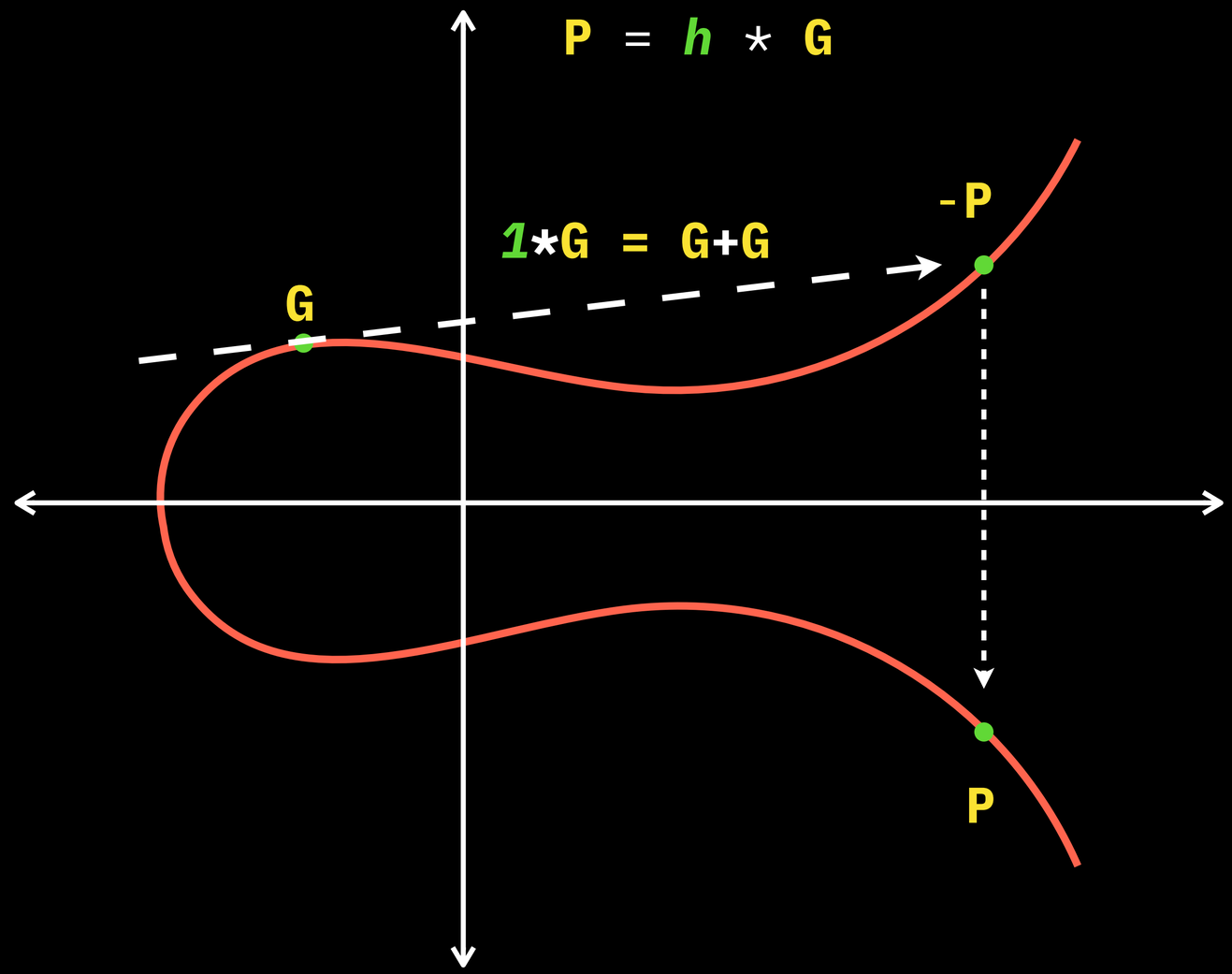
$$y^2 = x^3 + ax^2 + b$$

- Coefficients: a, b
- $\mathbb{Z}_p$ order: p
  values for x and y MUST be in $[0, p)$ range
- Curve order: n
  Number of points on a curve (for a given generator point)



Elliptic curve cryptography

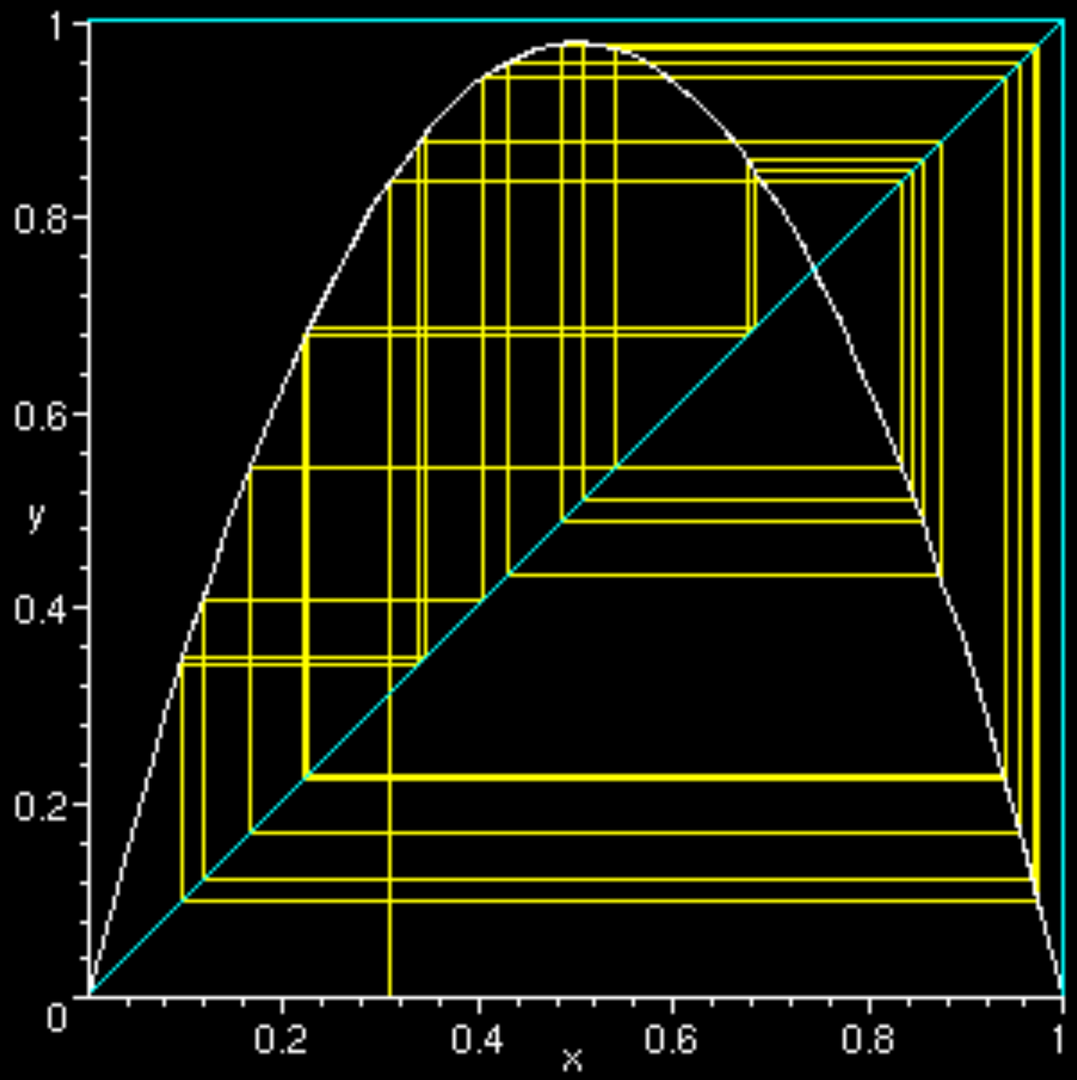
- We define a dedicated point
  G = (x_g, y_g)
- We take a random number ***h*** from $\mathbb{Z}_p$
- We add **G** to itself ***h*** times which gives us another point on the curve
  P
- This is our public key!
  P = (x_p, y_p)



But why it is cryptography?

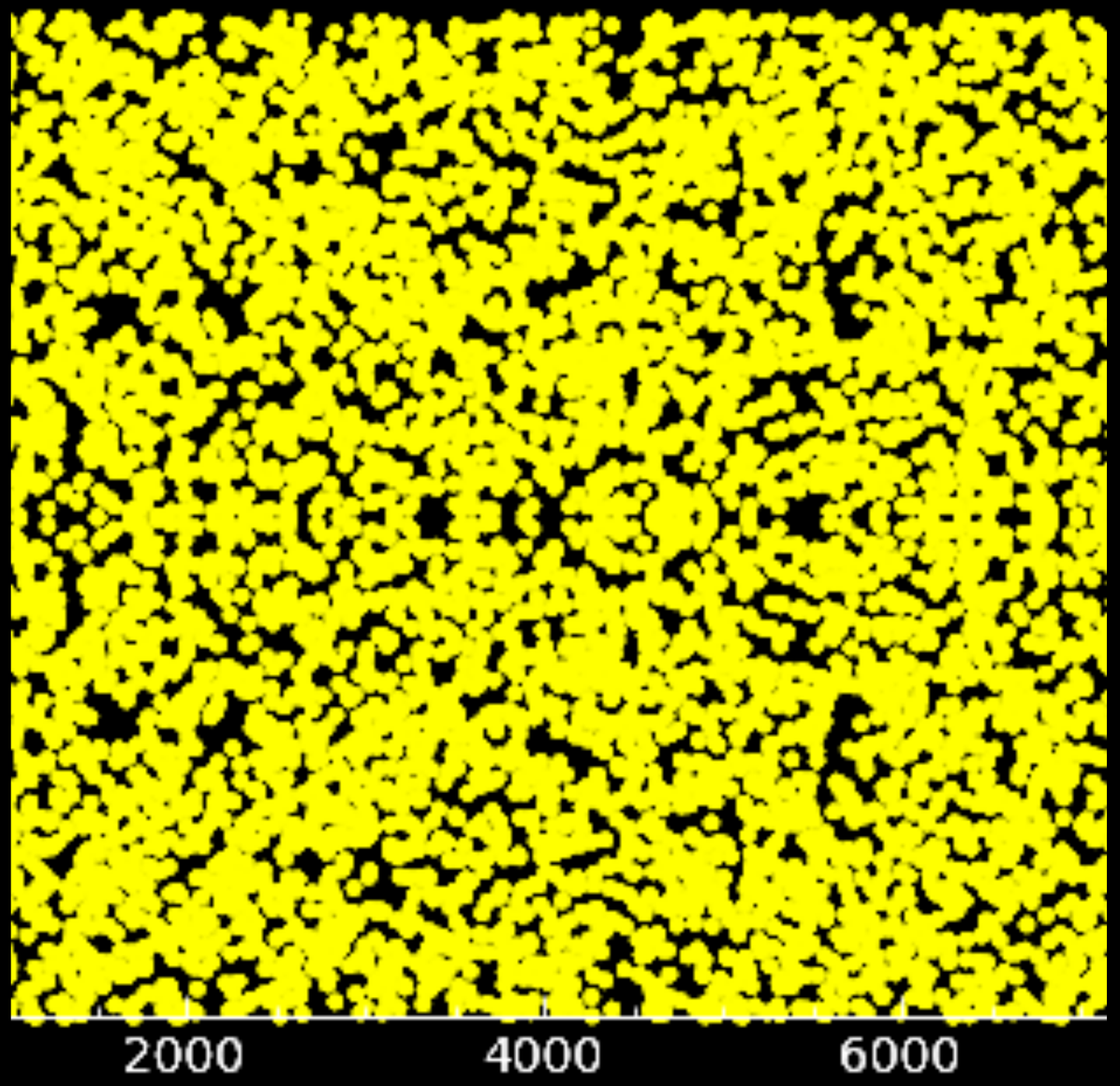
Why we can do a crypto on elliptic curves?

Unpredictability of simple systems



But why prime order field?

- They result in a ***dense*** elliptic curve point fitting
- Required by Cayley–Bacharach theorem
- Proven to have a discrete log problem



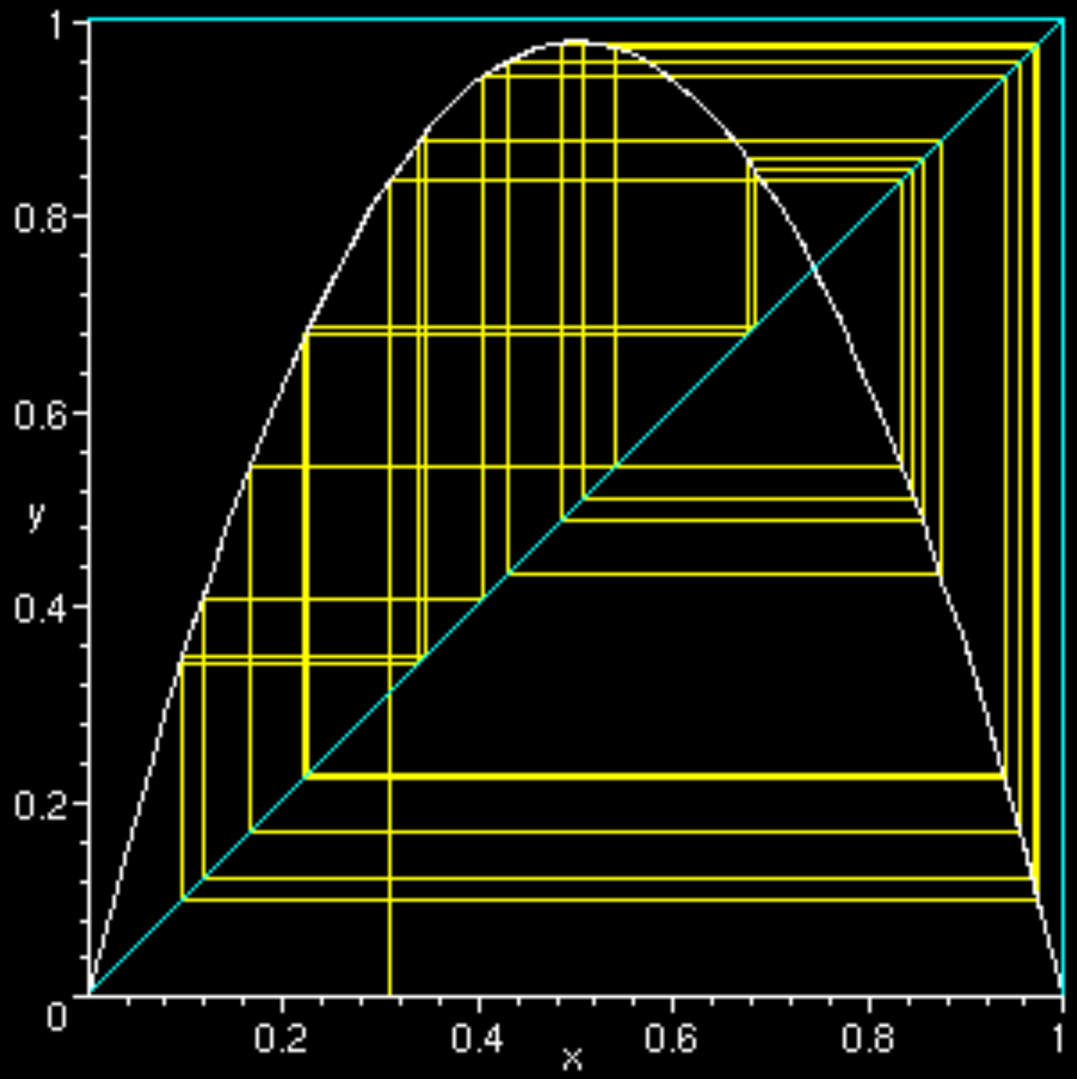
Discrete log problem

We can’t reverse scalar multiplication of elliptic curve points

Knowing **P** = ***h*** ⋆ **G**

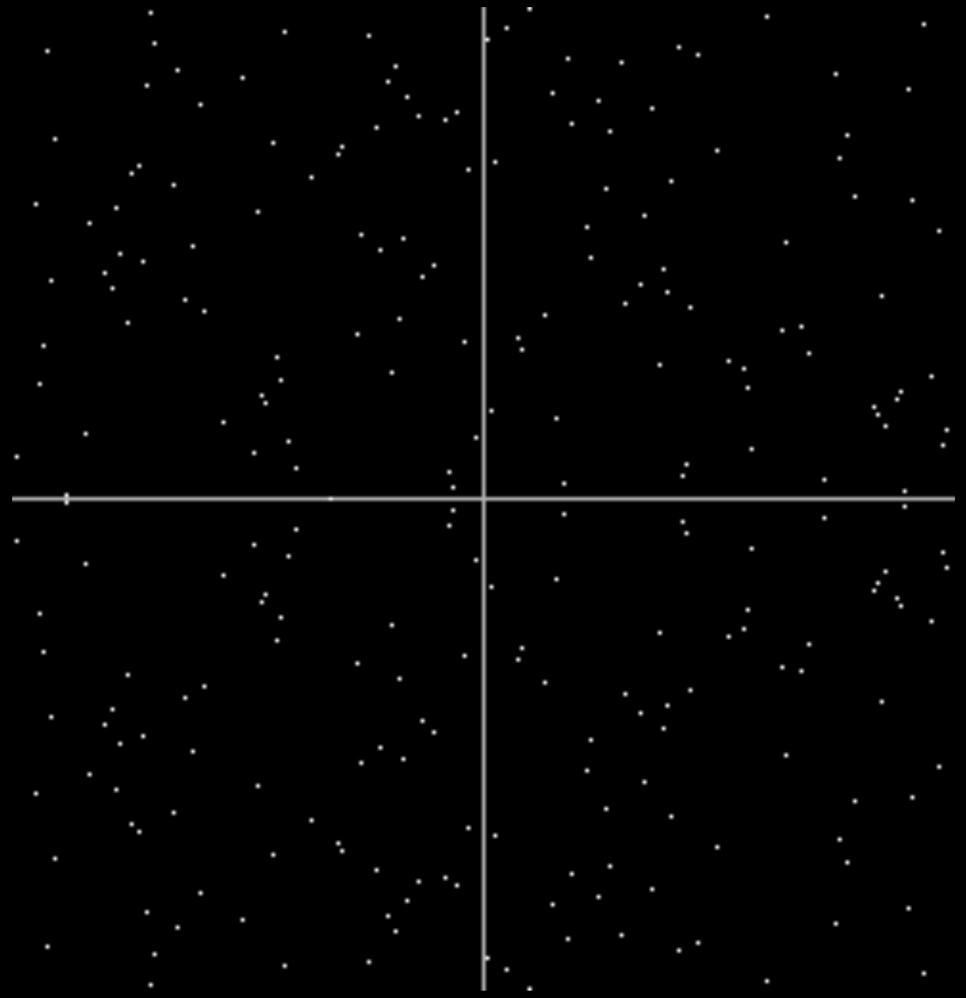
we would not be able to guess ***h***

without trying ***a*** ⋆ **G** for all numbers ***a*** ∈ $\mathbb{Z}_p$



Sec256k1

- p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
  FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F
  = 2256 - 232 - 29 - 28 - 27 - 26 - 24 - 1
- a = 0
- b = 7
- G.x = 79BE667E F9DCBBAC 55A06295 CE870B07
  029BFCDB 2DCE28D9 59F2815B 16F81798
- n = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE
  BAAEDCE6 AF48A03B BFD25E8C D0364141
  ~ 2^{256} / 2



Part V: Commitments rationale

Collision-resistant public key tweaking: LNPBP-1

- No double commitment
- Differentiable from other public key tweaking schemes
  - Pay-to-contracts
  - Taproot
  - …
- Protected from different forms of commitment attacks

Public key homomorphic properties

if **P** and **p** are public and private keys

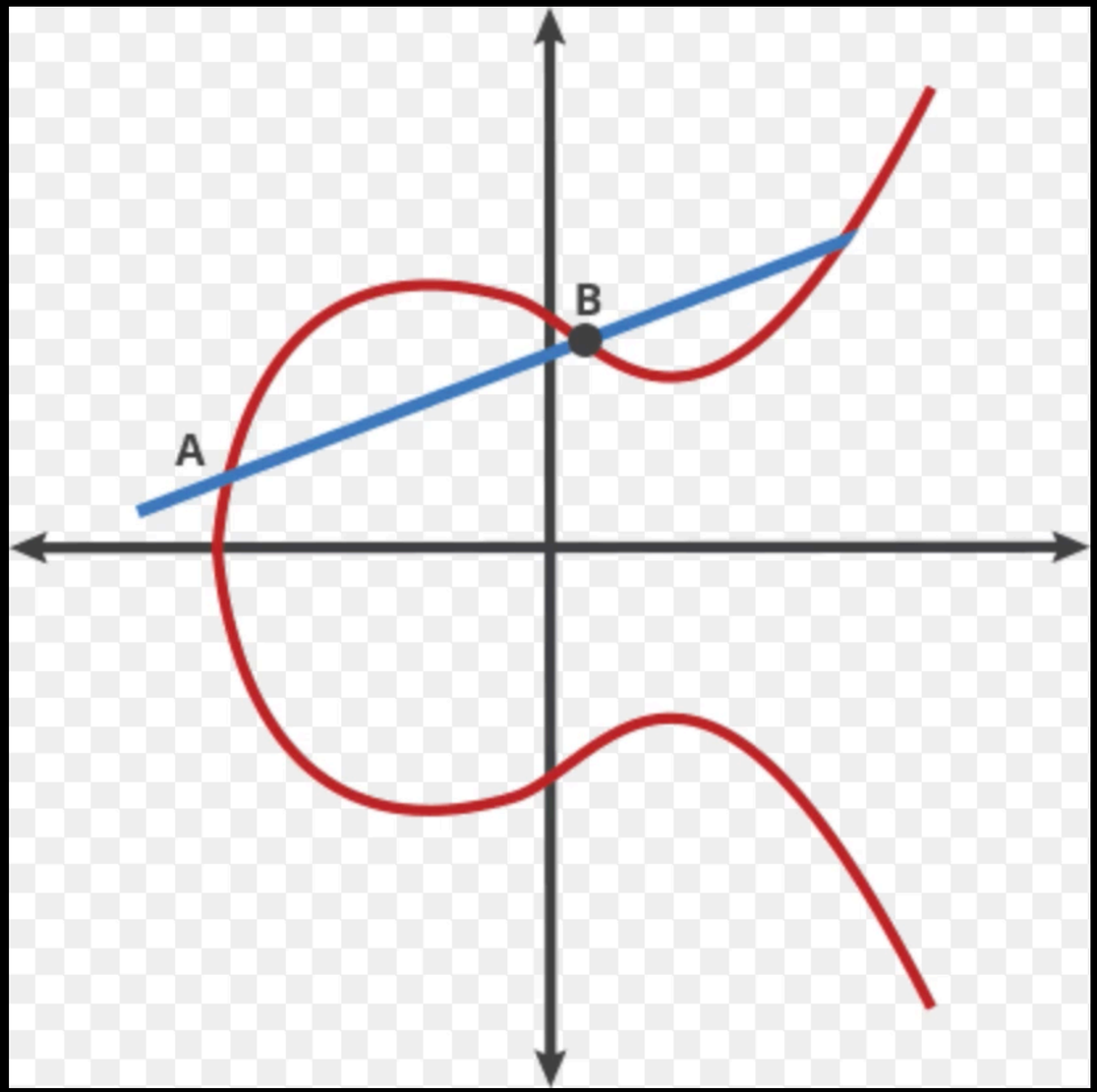
h = SHA256(data)

T = **P** + **h** * **G**

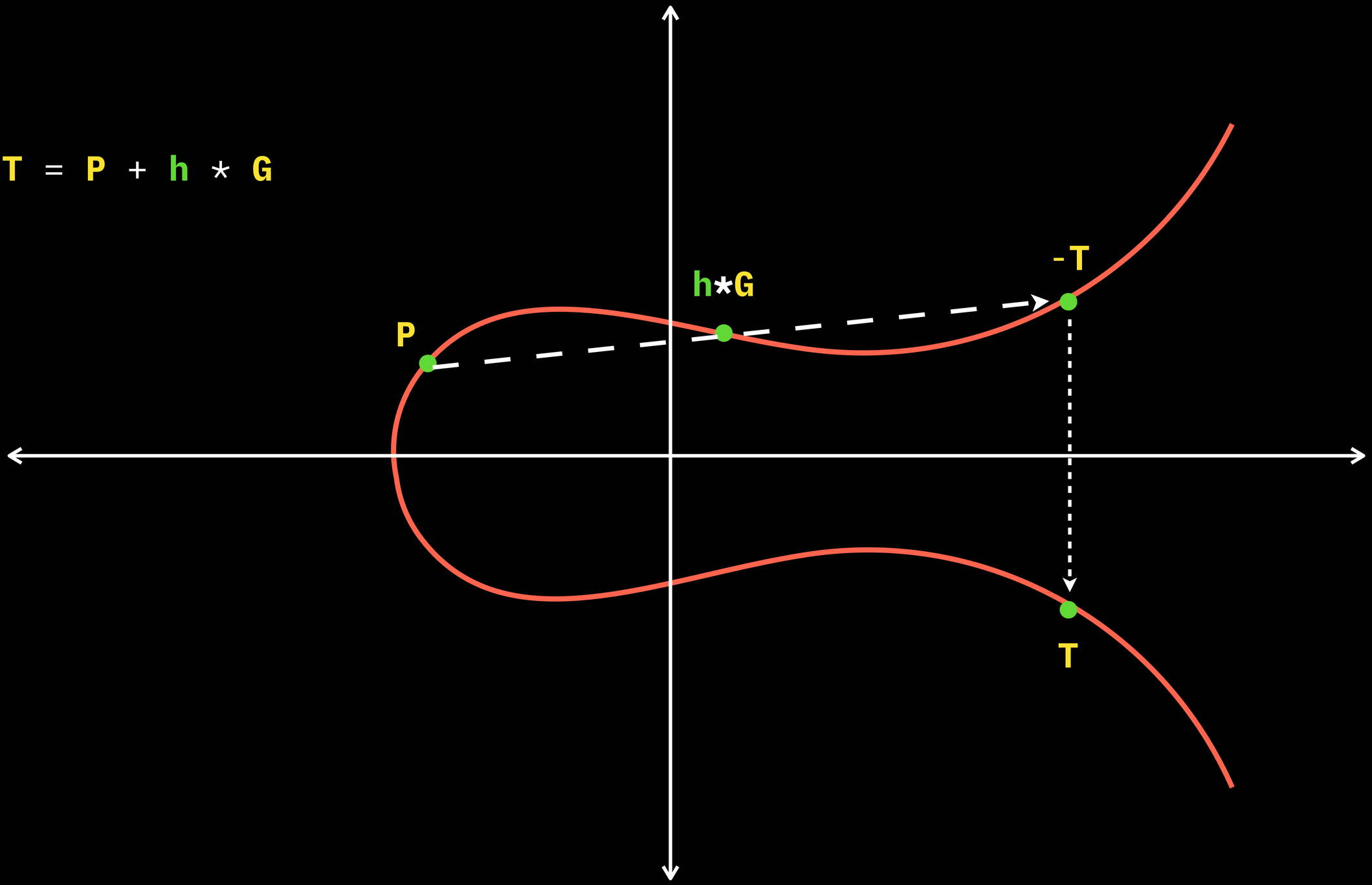
use **T** instead of **P**

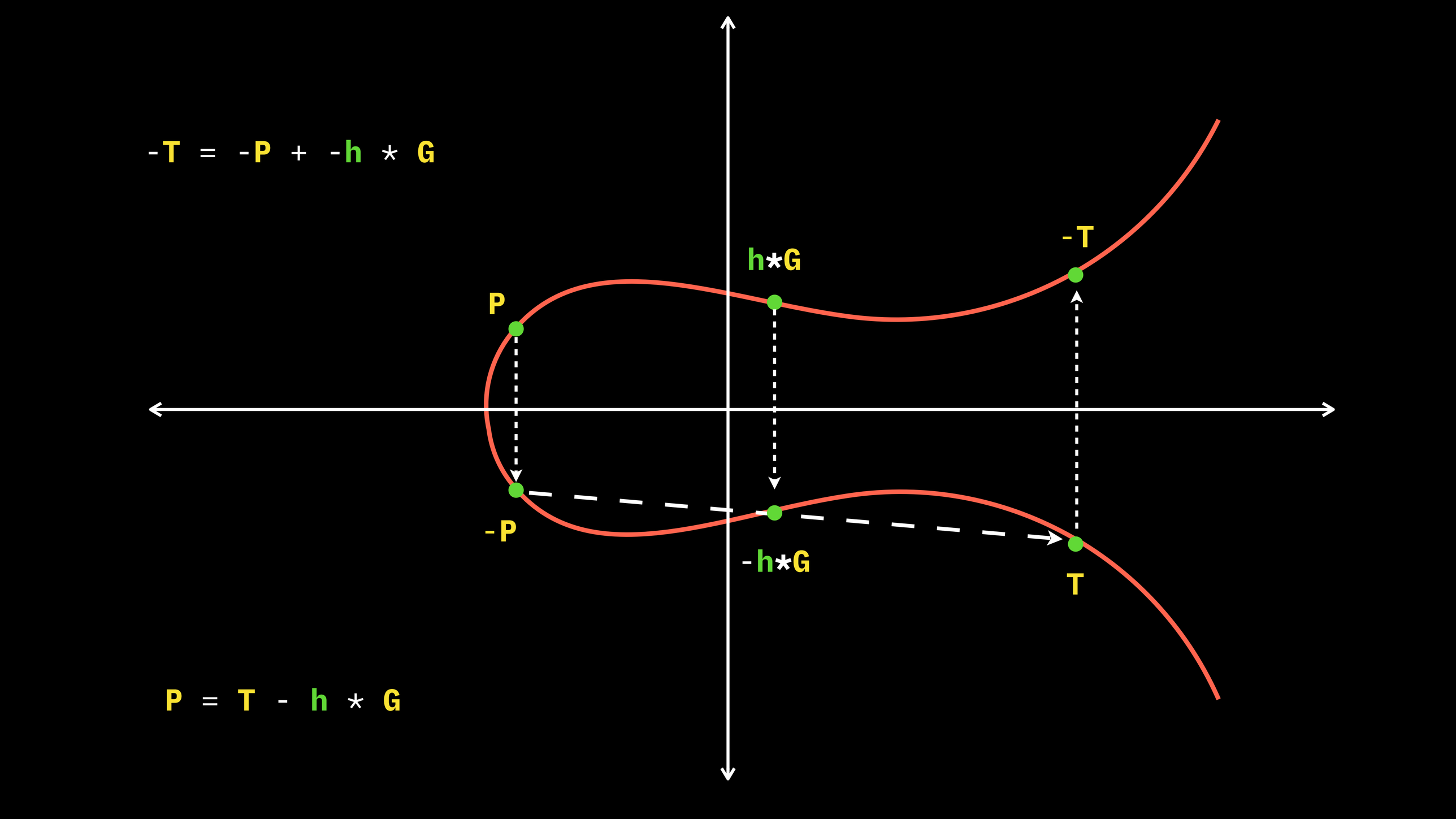
to spend, use

t = **p** + **h**



$$T = P + h * G$$




-T = -P + -h * G
h*G
-T
P
-P
-h*G
T
P = T - h * G

Demo #1

Public key tweaking reversibility

Introducing LBX

lbx is a command-line utility for working with LNP/BP standards

- Inspired by:
  bx from libbitcoin by *Eric Voskuil*
  hal from rust-bitcoin by *Steven Roose*

- Covers much more specific LNP/BP stuff absent in **bx** and **hal**

- Based on **rust-lnpbp** library and implements its functionality in CLI

$ git clone https://github.com/lnp-bp/lbx

Demo

Step 1: let’s add two random public keys

```
$ lbx ec-add
02d1d80235fa5bba42e9612a7fe7cd74c6b2bf400c92d866f28d429846c679cceb
021bab84e687e36514eeaf5a017c30d32c1f59dd4ea6629da7970ca374513dd006
```

Demo

Step 1: lets add two random public keys **P** and **F** (containing tweaking factor) to obtain resulting tweaked key **T**

```
$ lbx ec-add
02d1d80235fa5bba42e9612a7fe7cd74c6b2bf400c92d866f28d429846c679cceb
021bab84e687e36514eeaf5a017c30d32c1f59dd4ea6629da7970ca374513dd006

031db1a5db2f68323f57434e5edc4a714d9f040b3191eeb2c339a9d8d86081cc9a
```

Demo

Step 2: lets add the initial key **P** to the inverse of **T**

03**1**db1a5db2f68323f57434e5edc4a714d9f040b3191eeb2c339a9d8d86081cc9a
->
02**1**db1a5db2f68323f57434e5edc4a714d9f040b3191eeb2c339a9d8d86081cc9a

$ lbx ec-add
02d1d80235fa5bba42e9612a7fe7cd74c6b2bf400c92d866f28d429846c679cceb
021db1a5db2f68323f57434e5edc4a714d9f040b3191eeb2c339a9d8d86081cc9a

Demo

Step 2: lets add the initial key **P** to the inverse of **T**

```
$ lbx ec-add
02d1d80235fa5bba42e9612a7fe7cd74c6b2bf400c92d866f28d429846c679cceb
021db1a5db2f68323f57434e5edc4a714d9f040b3191eeb2c339a9d8d86081cc9a

031bab84e687e36514eeaf5a017c30d32c1f59dd4ea6629da7970ca374513dd006
```

What do we have now? The inverse of F!

```
021bab84e687e36514eeaf5a017c30d32c1f59dd4ea6629da7970ca374513dd006
```

P + F = T <=> **P + (-T) = -F**

Is this a problem?

No, because:

- We do not rely on EC point addition irreversibility to hide the message digest nor the original public key
- For a given tweaked key **T** there exists an infinity of messages and original public keys **P** that can result in **T**
- We need either (**T**, **P**, **msg**) or (**T**, **F**, **msg**) to proof that **T** contains commitment to **msg**

Workshop #1

Selecting which information to provide for the commitment proofs with public key tweaking

Selection criteria

	Original public key	Tweaking factor
Privacy		+
Computational efficiency, commit phase	=	=
Computational efficiency, reveal phase	+	
Storage		+

Tagged hashes

- Proposed by Pieter Wuille as a part of the original Taproot proposal, and later, Schnorr signatures
- Prefixing message by a two SHA256 hashes of some protocol-specific tag

To prevent possibility of double tweak
we have to commit to the initial
value of the public key

$$T = P + h(P, msg) * G$$

$$T = P + h(P, msg1) * G + h(P, msg2) * G$$

$$T = P1 + h(P1, msg2) * G$$

$$T = P2 + h(P2, msg1) * G$$

$$\text{if } msg1 \neq msg2: P1 \neq P2$$

$$P2 = P1 + h(P2, msg2) * G$$

$$P1 = P2 + h(P1, msg1) * G$$

Hash function attack

- Length extension
- Birthday
- Message brute forcing

Workshop #2

Designing the collision-resistant commitment function for a given message

LNPBP-1 Primer, hashing part

Selection criteria

	SHA256	Double SHA256	HMAC-SHA256
Privacy			
Computational efficiency, commit phase			
Computational efficiency, reveal phase			
Security			

What can go wrong with public key tweaking?

- Scalar overflow for hash function value
- Tweaking, resulting in point-at-infinity
- Tweaking, resulting in zero point
- Tweaking, resulting in the original point

What can go wrong with public key tweaking?

- **Scalar overflow for hash function value**
  - ➡ when **h=SHA256d(msg)** exceeds $\mathbb{Z}_p$ order p
  - ➡ $P\{h>p\} = (2^{256} - p) / p = (2^{32} + 2^9 + 2^8 + 2^7 + 2^6 + 2^4 + 1) / (2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1) = \mathbf{3.7 * 10^{-66}\%}$
- Tweaking, resulting in point-at-infinity
- Tweaking, resulting in zero point
- Tweaking, resulting in the original point

What can go wrong with public key tweaking?

- Scalar overflow for hash function value
- **Tweaking, resulting in point-at-infinity**
  - ➡ when **P** = -(**h** * **G**)
    due to DLP we can’t know which h may satisfy this beforehand
  - ➡ P{**P**=-(**h***G)|**P**} = 1 / **n** =~ 2 / 2^{256} = **1.73 * $10^{-72\%}$**
- Tweaking, resulting in zero point
- Tweaking, resulting in the original point

What can go wrong with public key tweaking?

- Scalar overflow for hash function value
- Tweaking, resulting in point-at-infinity
- **Tweaking, resulting in zero point**
- **Tweaking, resulting in the original point**

➡ not a problem: operation succeeds and still contains a provable commitment

Workshop #3

Designing the deterministic public key tweaking algorithm

LNPBP-1 Primer

Selection criteria: scalar overflow

	Fail the procedure	Wrap scalar	Introduce randomness
Privacy			
Computational efficiency, commit phase			
Computational efficiency, reveal phase			
Security			

Selection criteria: point at infinity

	Fail the procedure	Wrap scalar	Introduce randomness
Privacy			
Computational efficiency, commit phase			
Computational efficiency, reveal phase			
Security			

Workshop #4

Finalizing LNPBP-1 standard

Part VI: Script & Transaction commitments

Problems with Bitcoin scripts

- Sometimes we have a complete script in scriptPubKey: P2OR, P2PK
- Sometimes we have just a hash of a public key with commitment: P2(W)PKH
- Sometimes we have a hash of the script: P2(W)SH
- And in a future we will have just a plain public key committed to a script (P2TR)

How to provide a consistent and coherent logic for commitment verification and off-chain data?

Introducing Script Type system

- Rust allows to define new types as wrappers

```
pub type LockScript; // the script that has to be known to spend the transaction
pub type PubkeyScript; // the actual content of scriptPubkey
pub type RedeemScript; // content of the stack for P2(W)SH scripts
pub type TapScript; // the script behind taproot scriptPubkey tweak
```

```
$ git clone https://github.com/lnp-bp/rust-lnpbp
```

```
less src/common/types.rs
```

- May become a part for rust-bitcoin

Introducing Script Type system

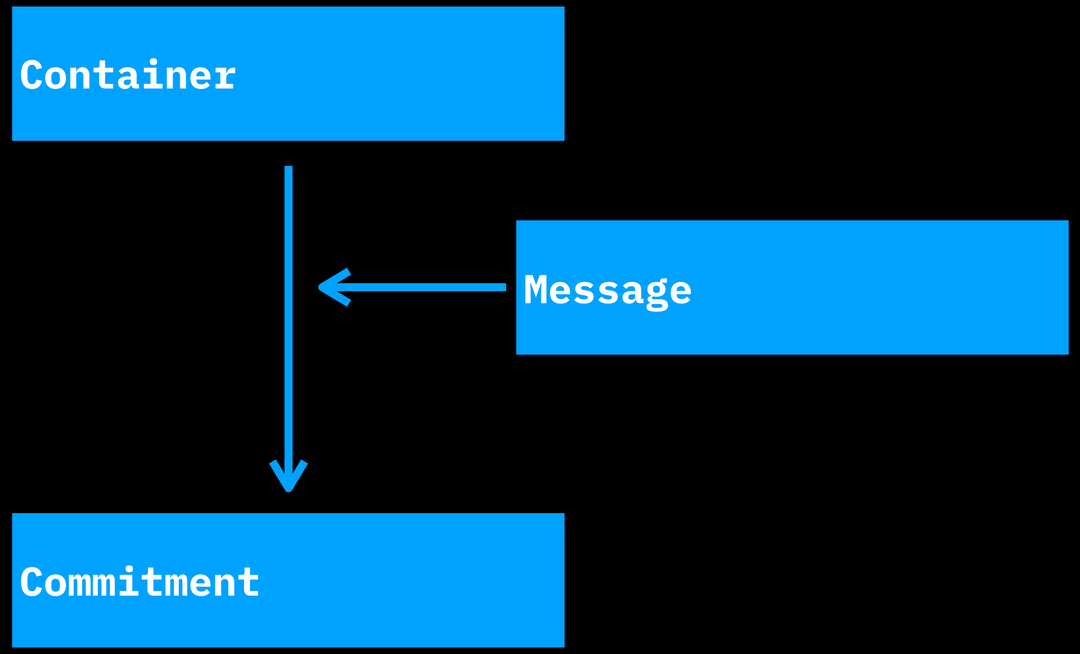
- Now we need to classify all our situations for scriptPubkey value meaning and provide the required data to reconstruct it:

```
pub enum ScriptPubkeyType {
    P2S(Script),
    P2PK(PublicKey),
    P2PKH(PubkeyHash),
    P2SH(ScriptHash),
    P2OR(Box<[u8]>),
    P2WPKH(WPubkeyHash),
    P2WSH(WScriptHash),
    P2TR(PublicKey),
}
```

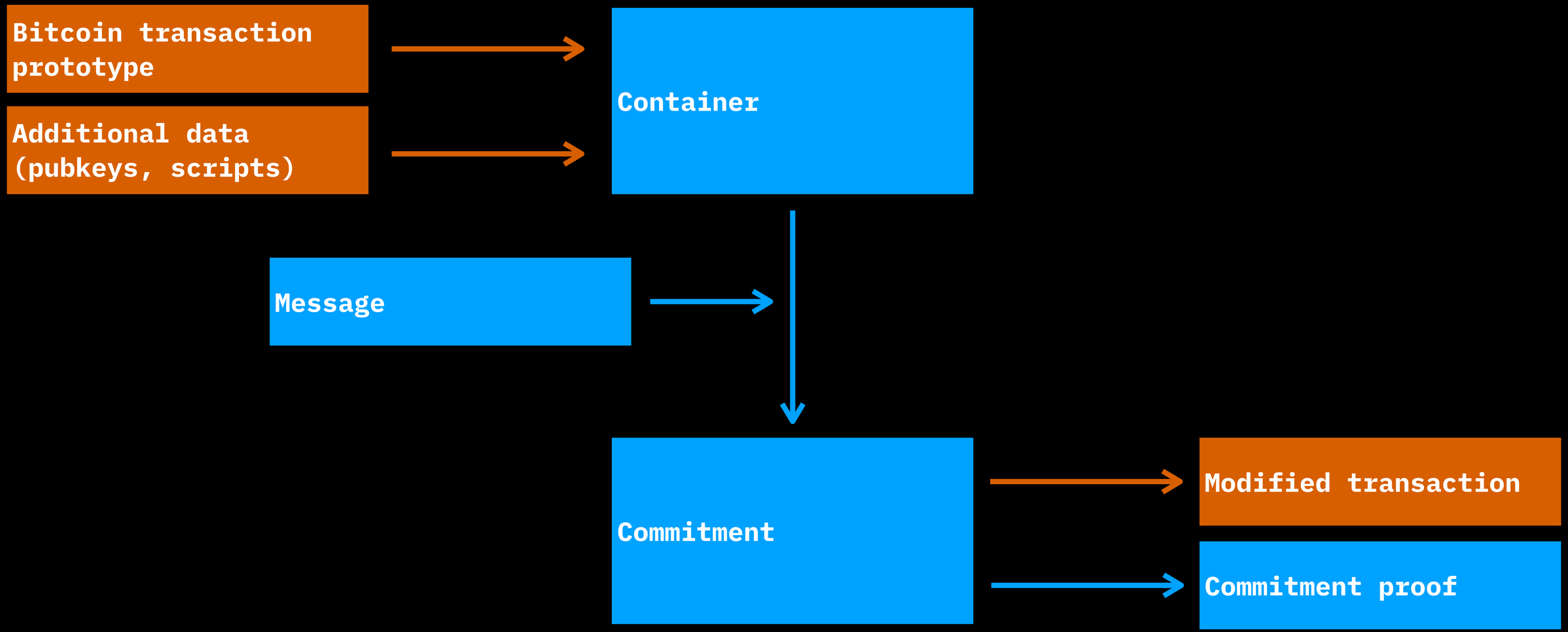
Transaction output commitments

```rust
#[derive(Clone, Eq, PartialEq)]
pub enum TxoutContainer {
    PublicKey,
    PubkeyHash(PublicKey),
    ScriptHash(LockScript),
    TapRoot(TaprootContainer),
    OpReturn(PublicKey),
    OtherScript,
}

#[derive(Clone, Eq, PartialEq)]
pub enum TxoutCommitment {
    PublicKey(PubkeyCommitment),
    LockScript(LockscriptCommitment),
    TapRoot(TaprootCommitment),
}
```



Transaction commitment system: *commit*

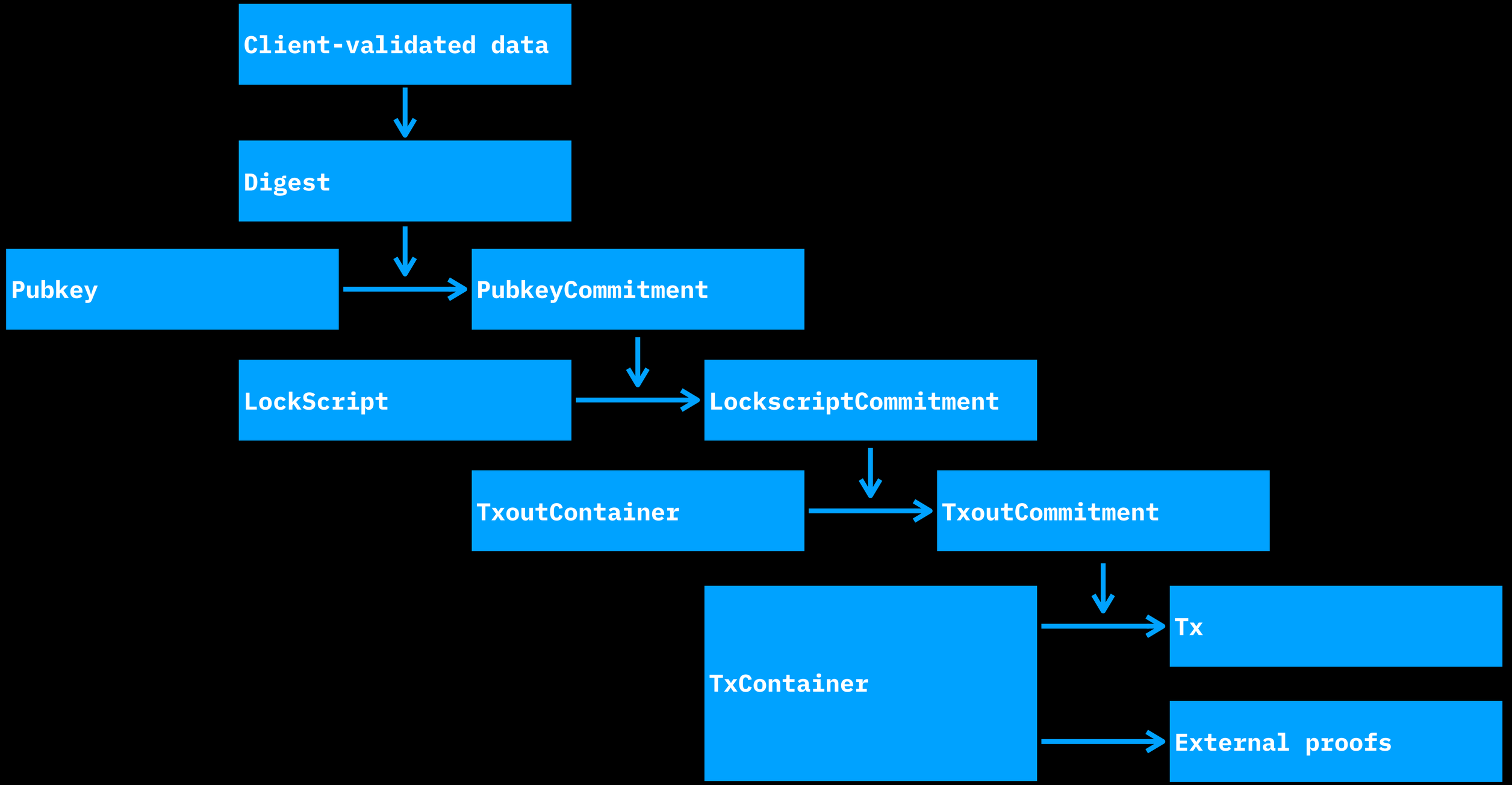


Embedded commitment splitting

```
pub trait EmbeddedCommitment<MSG, EXT, INT>
    where MSG: EmbedCommittable
{
    fn external_part(&self) -> EXT;
    fn internal_part(&self) -> INT;
    fn parts(&self) -> (INT, EXT);
}
```

Transaction commitments summary

	Container	Commitment	External	Internal
Public key	PublicKey	PublicKeyCommitment	tweaked PublicKey	original PublicKey
Script (LockScript)	LockScript	LockScriptCommitment	scriptPubKey	original LockScript
Transaction Output	TxoutContainer: *enum with possible script type-specific information*	TxoutCommitment: *one of* PubkeyCommitment, LockscriptCommitment or TaprootCommitment	TxOut	*none*
Transaction	TxContainer: *tx draft, entropy and txout_container*	TxCommitment	Transaction	entropy + original TxoutContainer


Client-validated data
Digest
Pubkey
PubkeyCommitment
LockScript
LockscriptCommitment
TxoutContainer
TxoutCommitment
TxContainer
Tx
External proofs

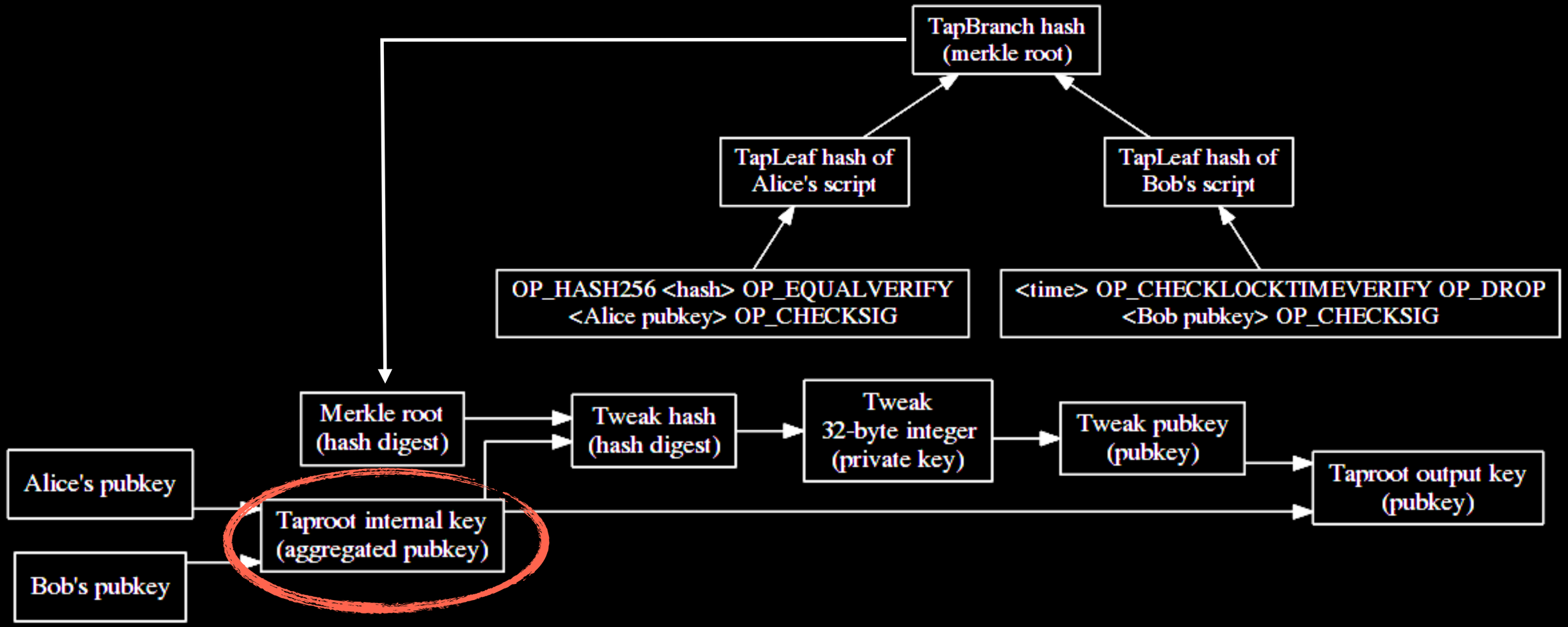
LNPBP-2: How to detect public keys in LockScript?

1. Start with an empty set of script-extracted public key hashes `H`
2. Scan the script for the presence of `OP_HASH160 OP_PUSH32 <32-bytes of data> OP_EQUALVERIFY OP_CHECKSIG[VERIFY]` template, extract the 32-byte sequences from the pattern and put each of them into the set `H`. Splice out each of the found script patterns from the script before the further processing.
3. Scan the script for the presence of `OP_PUSH32 <33-bytes of data> [OP_EQUALVERIFY] OP_CHECKSIG[VERIFY]` and `OP_PUSH <n> [<33-bytes of data>]+ OP_PUSH <m> OP_CHECKMULTISIG[VERIFY]` template, extract the 33-byte sequences from the pattern and make sure that they represent a valid x-coordinates of Secp256k1 cure points. Compute a double SHA-256 hash of them and put each of the resulting values into the set `H`. Splice out each of the found script patterns from the script before further processing.

LNPBP-2: How to detect public keys in LockScript?

4. Scan the script for the presence of `OP_PUSH32 <65-bytes of data> [OP_EQUALVERIFY] OP_CHECKSIG[VERIFY]` and `OP_PUSH <n> [<65-bytes of data>]+ OP_PUSH <m> OP_CHECKMULTISIG[VERIFY]` template, extract the 65-byte sequences from the pattern and make sure that they represent valid uncompressed public keys for the Secp256k1 curve points. Take their x-coordinates and compute a double SHA-256 hash of them and put each of the resulting values into the set `H`. Splice out each of the found script patterns from the script before further processing.
5. Compute double SHA-256 hash of each of the public keys in set `S` and put it into the set `H'`.
6. Ensure that `H' = H`, i.e. they have the same number of elements and there is exactly one element from the set `H'` that is equal to an element from the set `H`. Otherwise, fail the verification procedure.

Taproot-specific details



Taproot-specific details

- Public key in scriptPubkey commits to the Merkle root of TapScript branches, so it can’t commit to our CVS data
- We can use internal key instead, but all software running RGB and LNPBP-1,2,3-compatible protocols have to understand this fact, since it differs from normal P2PK-type commitment (while being indistinguishable from it)
- A special external proof structure for Taproot has to be constructed

Taproot-specific details

- We have to provide the Merkle root hash

```
#[derive(Clone, Eq, PartialEq)]
pub struct TaprootContainer {
    pub script_root: sha256::Hash,
    pub intermediate_key: PublicKey,
}

#[derive(Clone, Eq, PartialEq)]
pub struct TaprootCommitment {
    pub script_root: sha256::Hash,
    pub pubkey_commitment: PubkeyCommitment,
}
```

LNPBP-3: How to determine tx output with commitment?

- Combination of external to transaction and internal factors
- External factor - entropy - adds protection from chain analysis
- Internal factor - fee - allows to decide which output to use for the commitment at the moment of transaction construction

LNPBP-3: How to determine tx output with commitment?

- We need to include pre-defined entropy into the Container and Commitment
- ... and modify the fee inside the resulting transaction

```rust
#[derive(Clone, Eq, PartialEq)]
pub struct TxContainer {
    pub entropy: u32,
    pub tx: Transaction,
    pub fee_output: u32,
    pub txout_container: TxoutContainer,
}

#[derive(Clone, Eq, PartialEq)]
pub struct TxCommitment {
    pub entropy: u32,
    pub tx: Transaction,
    pub tweaked: TxoutCommitment,
    pub original: TxoutContainer,
}
```

Day 1 Summary:

Now we can embed commitment to client-validated data into any transaction with LN and Taproot compatibility, strong privacy and collision-security guarantees