The Universal Cryptographic Adapter

Justin Glibert

0xPARC

0xPARC's vision for Data Systems

Thesis: Programmable Cryptography enables the Universal Data Adapter. The "Turing Machine" moment for Data.

Cryptography as a way to ***get more*** out of data

- Take data out of their Data Prison
- Enable more interoperability and usage of data
- Break down silos

Goal

⇒ Create **more total value** out of data

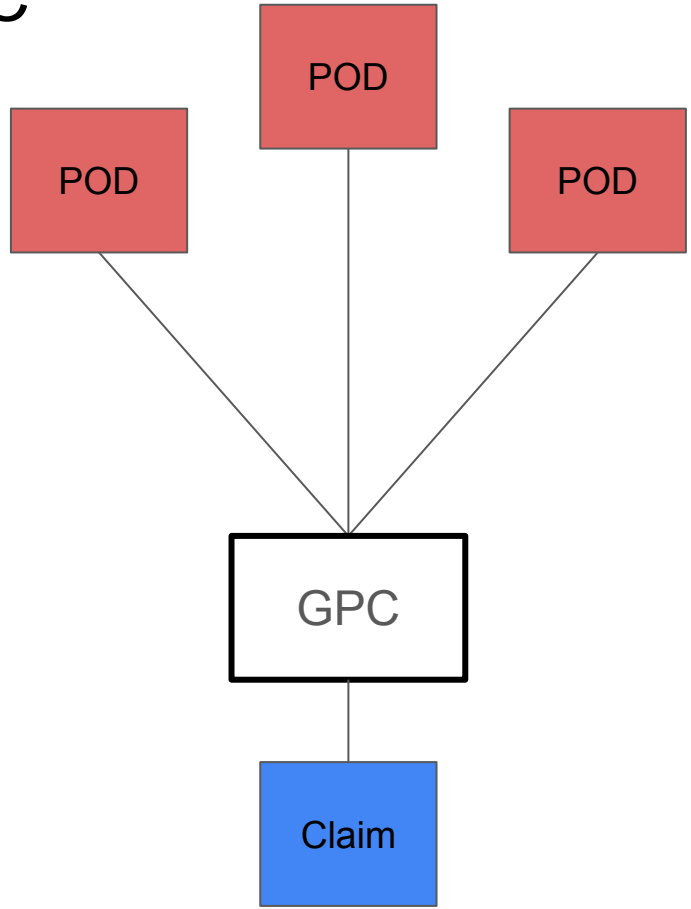
⇒ Enable a pragmatic route towards data ownership in a world where **data keeps increasing in value.**

0xPARC's Data Stack: POD1 & GPC

The system:

- POD are *specially issued* packets of cryptographic data
- GPCs ingest PODs and make a claim using ZKPs
- Performant enough to run on mobile browsers

Close to production ready



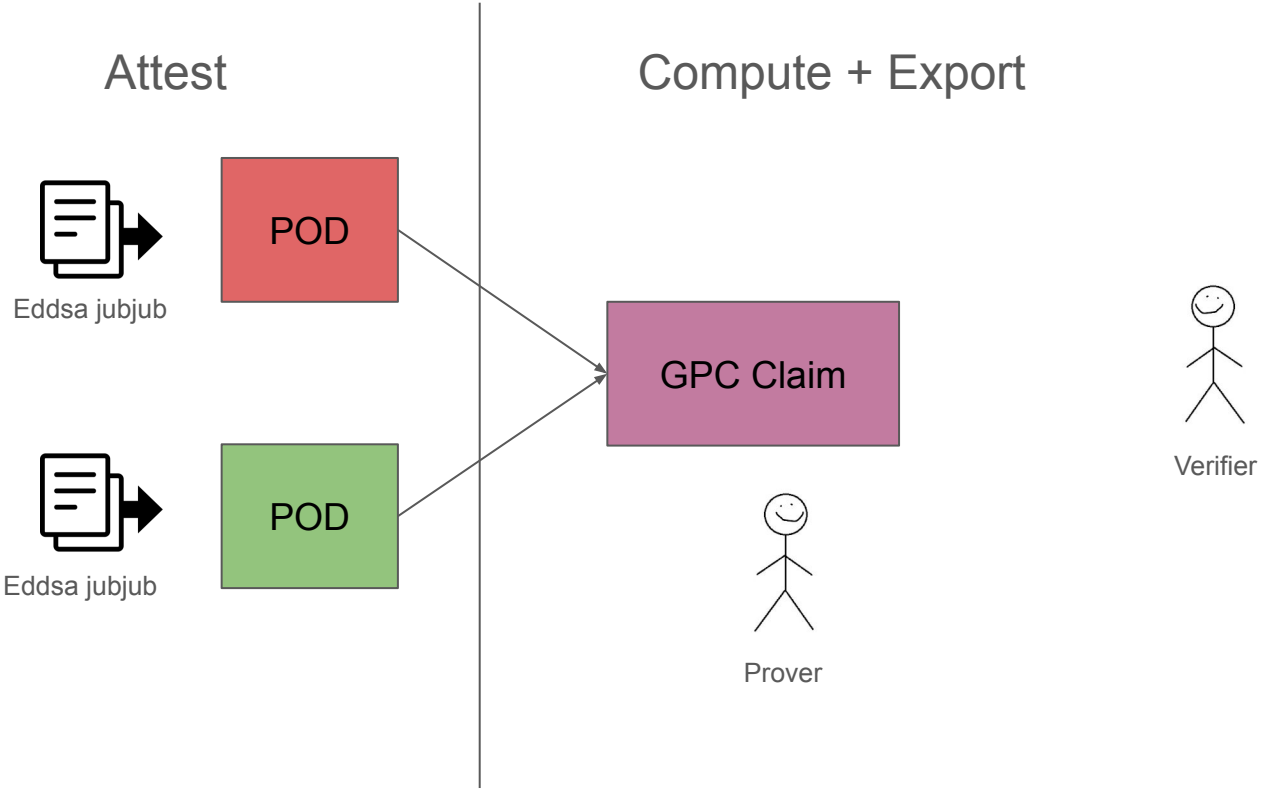
Users: all of you!

- [POD] Devcon ticket
- [POD] Frogs
- [GPC] Login into TG group
- [GPC] Entering FrogZone
- [GPC] Zupoll
- etc

Third-party apps

- Zupoll
- Zucast
- Zustamps
- ZKTG
- Zucat/Zurat
- Meerkat
- Social Layer
- Zuzagora
- Zuzalu QF
- Lemonade
- Zuzalu.city
- FROGCRYPTO
- Protocol week
- Zuum
- and more!

POD & GPC: TLDR


Attest
Compute + Export
Eddsa jubjub
POD
Eddsa jubjub
POD
GPC Claim
Prover
Verifier

What it looks like:

Issuer: POD.sign()

User: GPC Config

Verifier: GPC Claim

→ No new ZK circuits needed.

→ **Extremely** performant.

What it looks like:

Issuer: POD.sign()

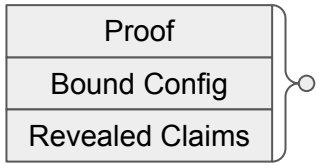
User: GPC Config

Verifier: GPC Claim

→ No new ZK circuits needed.

→ **Extremely** performant.

```
{
  "pods": {
    "idcard": {
      "entries": {
        "date_of_birth": {
          "isRevealed": false,
          "inRange": { "min": 0, "max":
1068104558283 }
        },
        "cardholder": { "isRevealed":
false, "isOwnerID": "SemaphoreV4" }
      }
    }
  }
}
```



What it looks like:

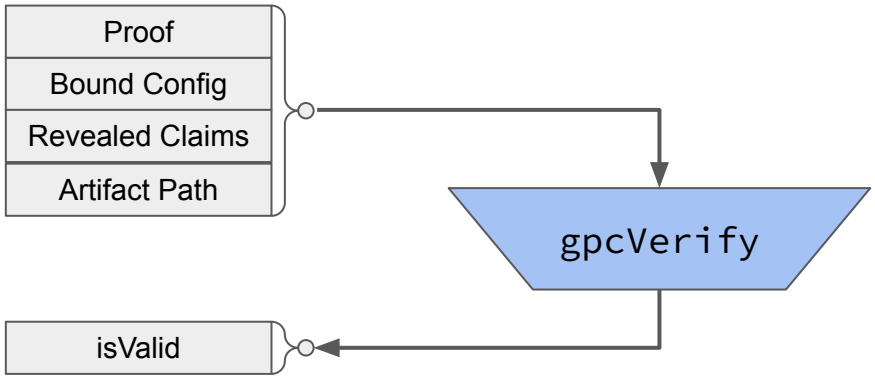
Issuer: POD.sign()

User: GPC Config

Verifier: GPC Claim

→ No new ZK circuits needed.

→ **Extremely** performant.




```
const isValid = await gpcVerify(
  proof,
  boundConfig,
  revealedClaims,
  GPC_ARTIFACTS_PATH
);
```

What it looks like:

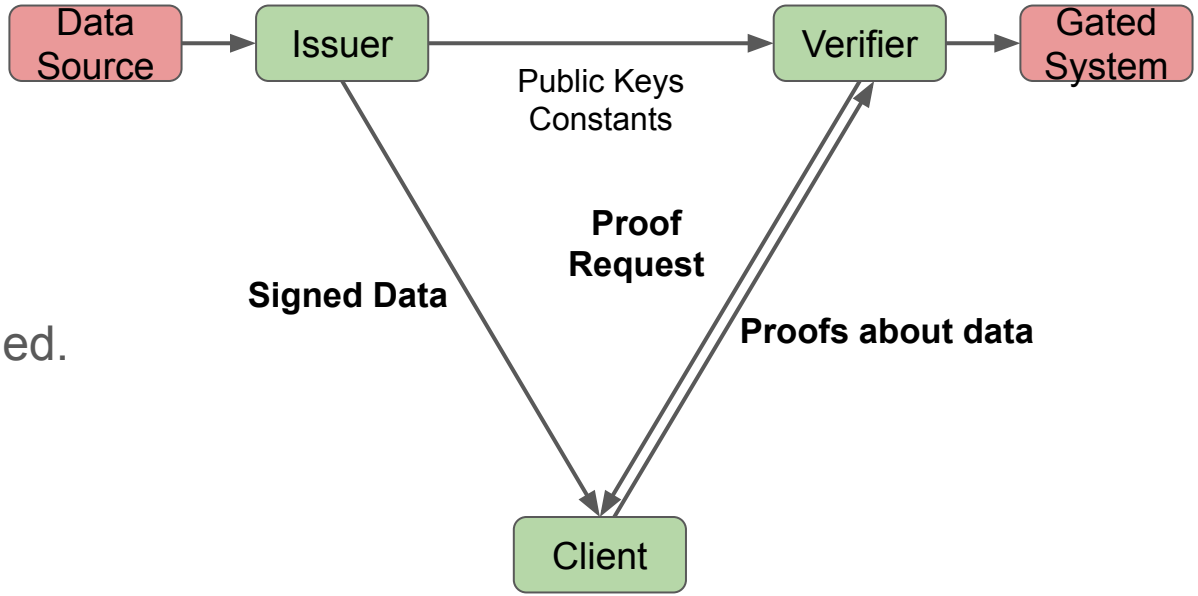
Issuer: POD.sign()

User: GPC Config

Verifier: GPC Claim

→ No new ZK circuits needed.

→ **Extremely** performant.

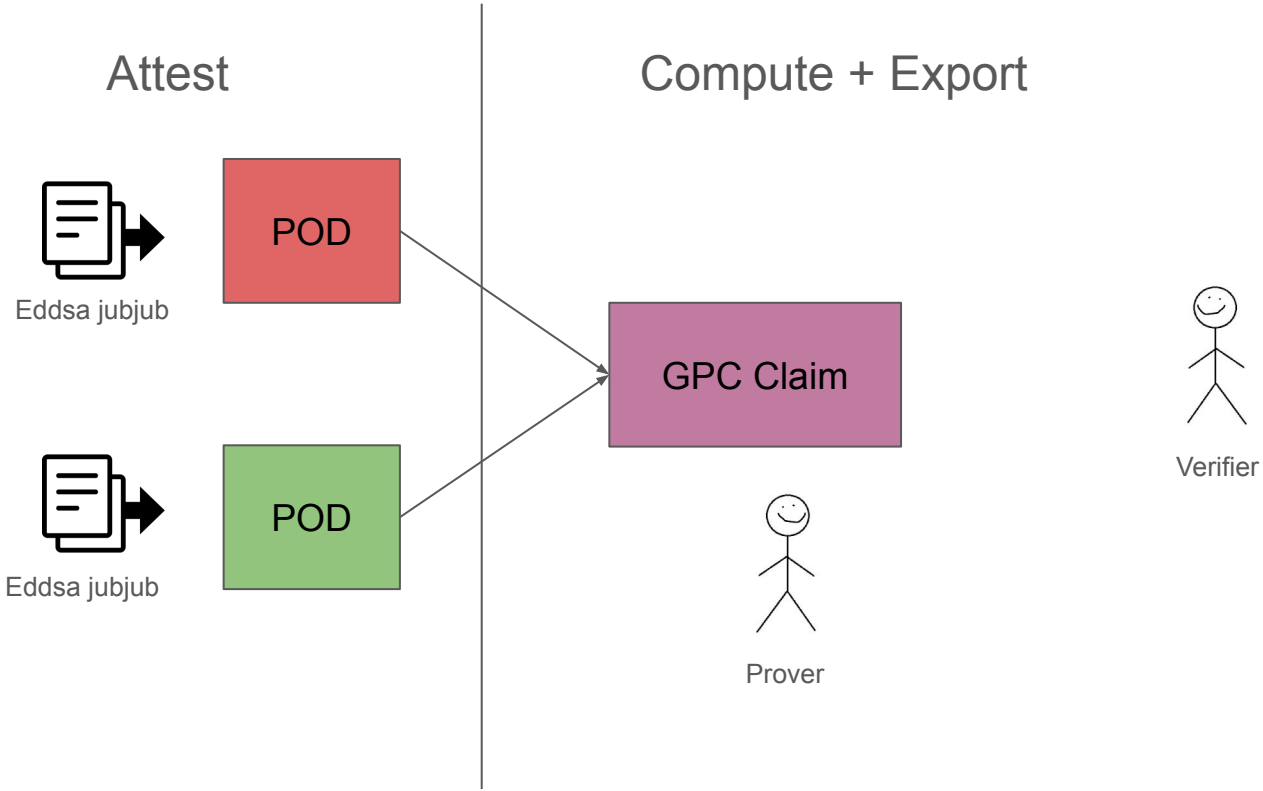


If you want to learn more about POD1

2:30 PM **Tomorrow** Classroom B

Summer 2024:

We wanted more out of our system


Attest
Compute + Export
POD
Eddsa jubjub
POD
Eddsa jubjub
GPC Claim
Prover
Verifier

Limitation of the POD1 Stack

1. Need specifically issued data
2. Limited to one round of computation
3. Single-player use-cases

   (not designed for confidentiality over multiple inputs)

The world today

- Lots of cryptography data, yet very heterogeneous

  TEE, blockchains, email, TLS, national ID system, JWTs, Canon verification, etc

- Basic use-cases often require two rounds of transformation; and a whole world might open up if going more than two rounds is possible.
- **Lots of work on domain-specific systems, yet no interoperability**

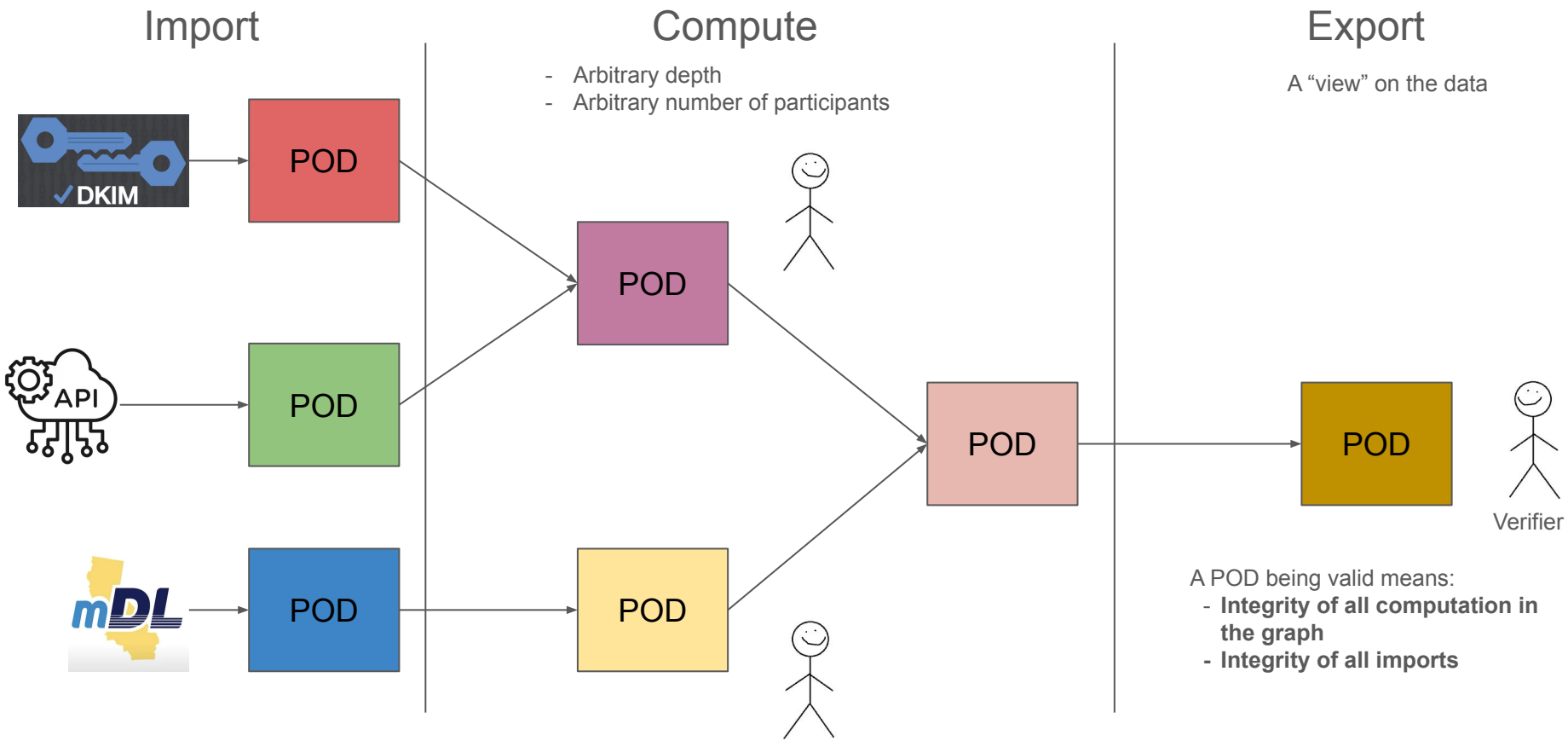
Programmable cryptography:

- Unlock of ZK: **trustless refactoring and import**

  ⇒ Can now create a universal system for making claim about the world's data

- Unlock of FHE / co-SNARK / MPC: **combine data confidentially**.

A verifiable and interoperable computational graph



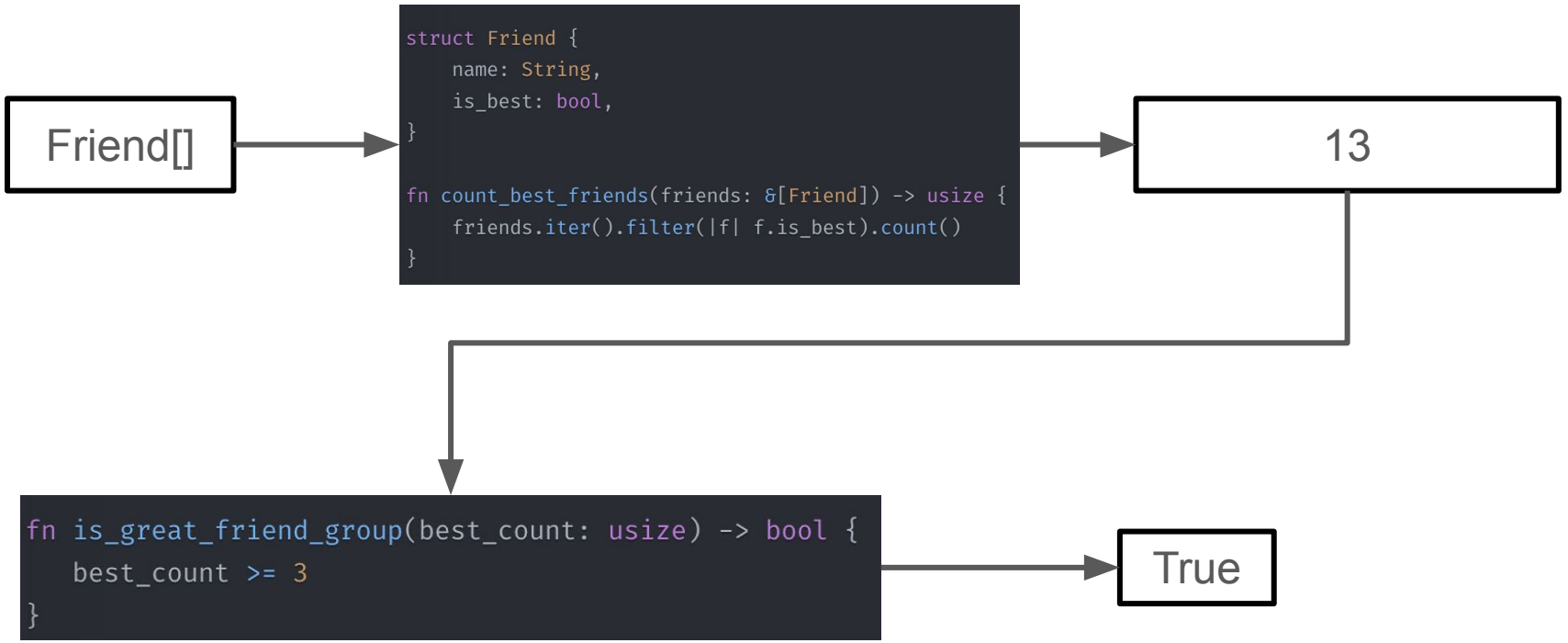
Goals for a second version of our stack

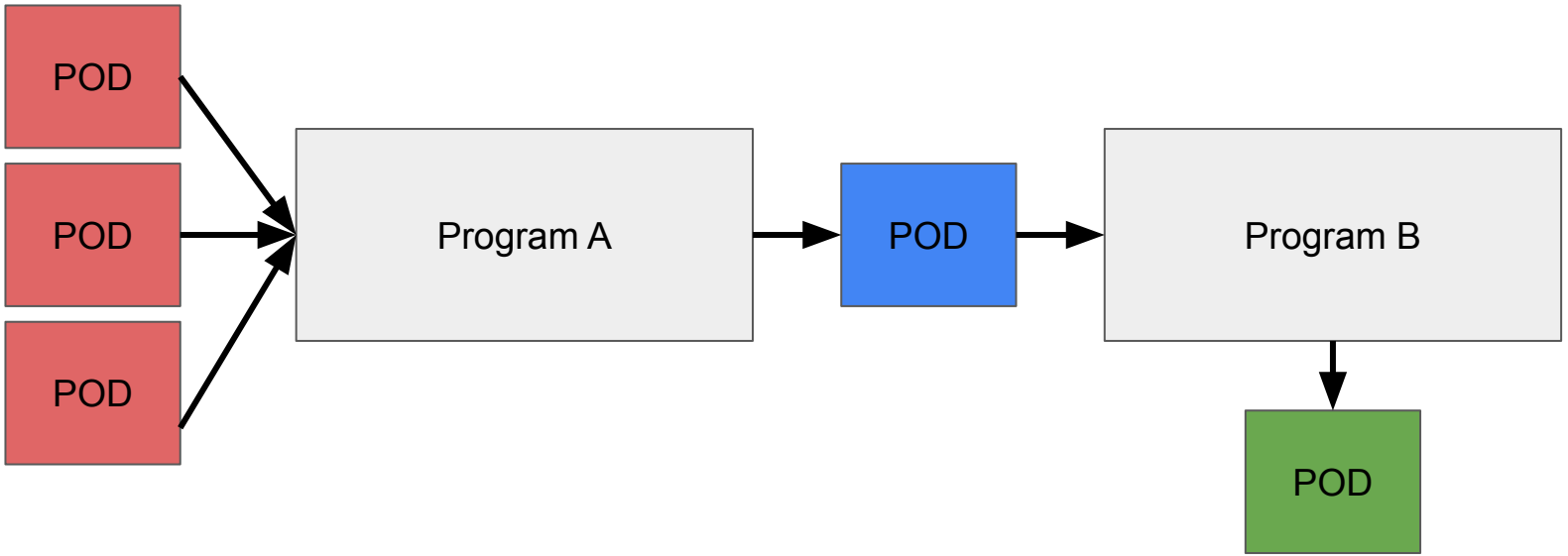
Introducing *any* cryptographic data in the system trustlessly (= scalable)

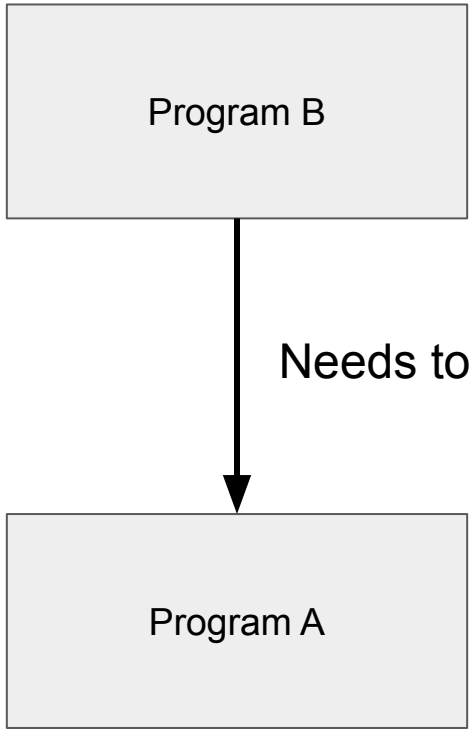
Deep computation graphs

Multi-party POD creation while maintaining confidentiality

Challenge 1: Program composability


Friend[]
struct Friend {
name: String,
is_best: bool,
}
fn count_best_friends(friends: &[Friend]) -> usize {
friends.iter().filter(|f| f.is_best).count()
}
13
fn is_great_friend_group(best_count: usize) -> bool {
best_count >= 3
}
True


POD
POD
POD
Program A
POD
Program B
POD



Via:

- Static analysis
- Formal proof
- etc

CPU → Logic

OPCODE → Logical operation

The Logic Virtual Machine

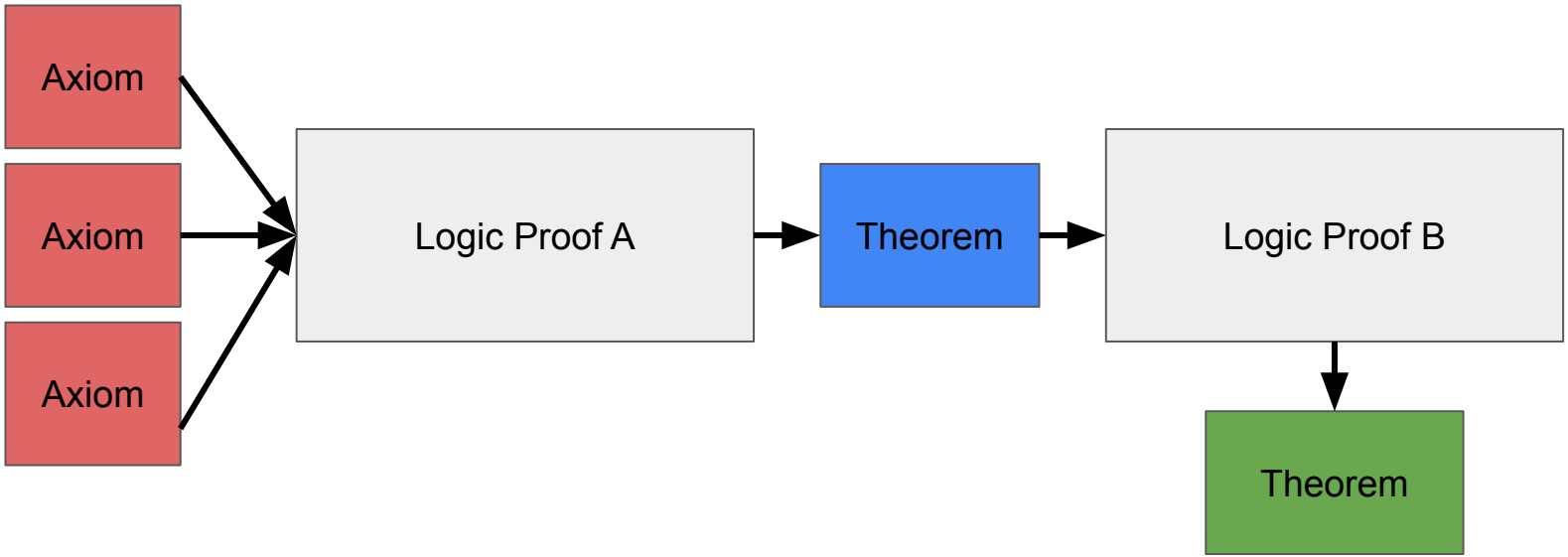
- Replace memory slots with **statements**

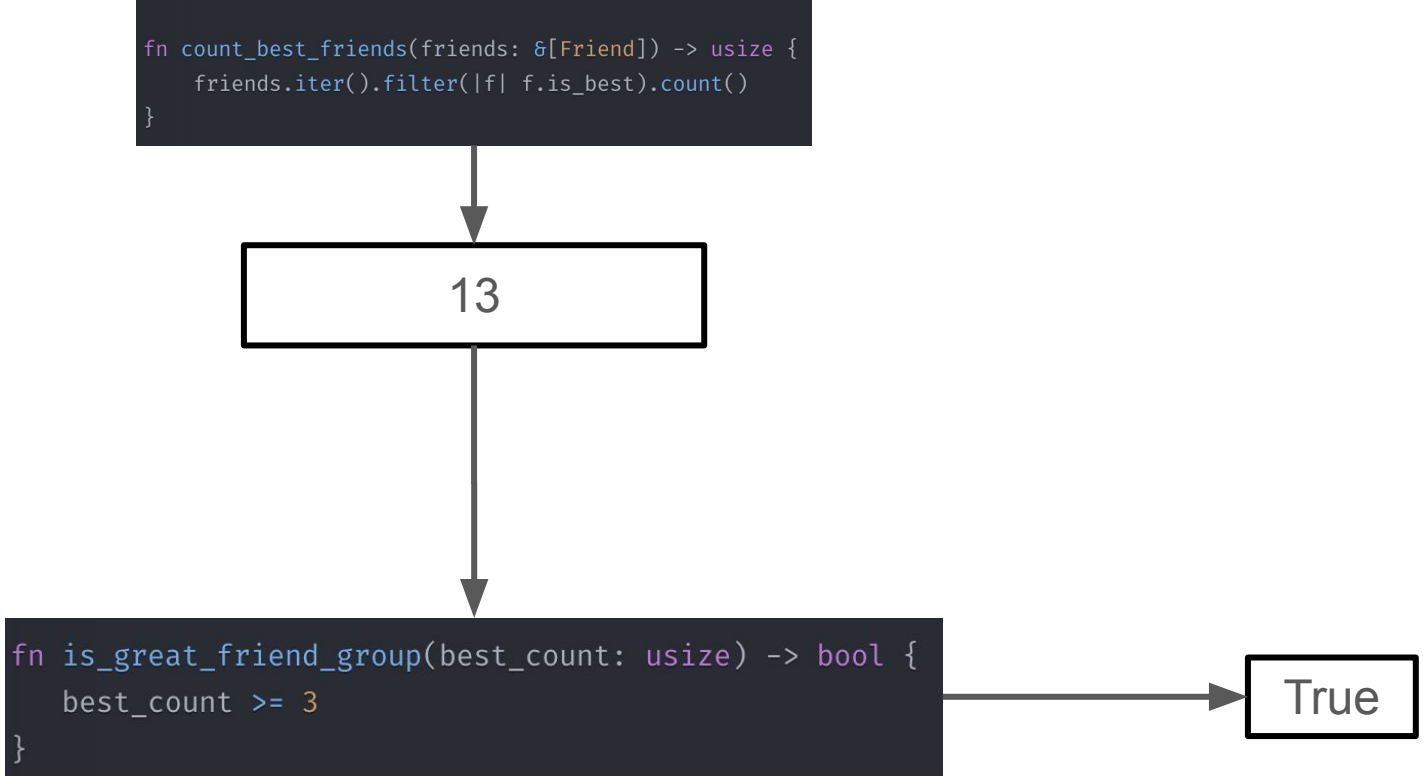
  Eg: “A is greater than B”; “There exists C such that $C \subseteq D$”

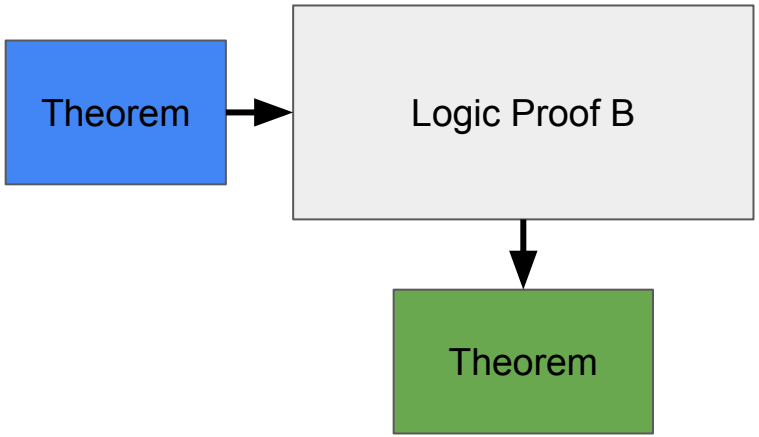
- Replace opcode with **logical operations**

  Eg: “$A < B \mapsto A \neq B$”; “$A := 10, B := 10 \mapsto A = B$”

- Data becomes **theorems**
- Programs become **proofs**


Axiom
Axiom
Axiom
Logic Proof A
Theorem
Logic Proof B
Theorem


fn count_best_friends(friends: &[Friend]) -> usize {
friends.iter().filter(|f| f.is_best).count()
}
13
fn is_great_friend_group(best_count: usize) -> bool {
best_count >= 3
}
True



In plain english:

Theorem

- I have 12 best friends

Implies

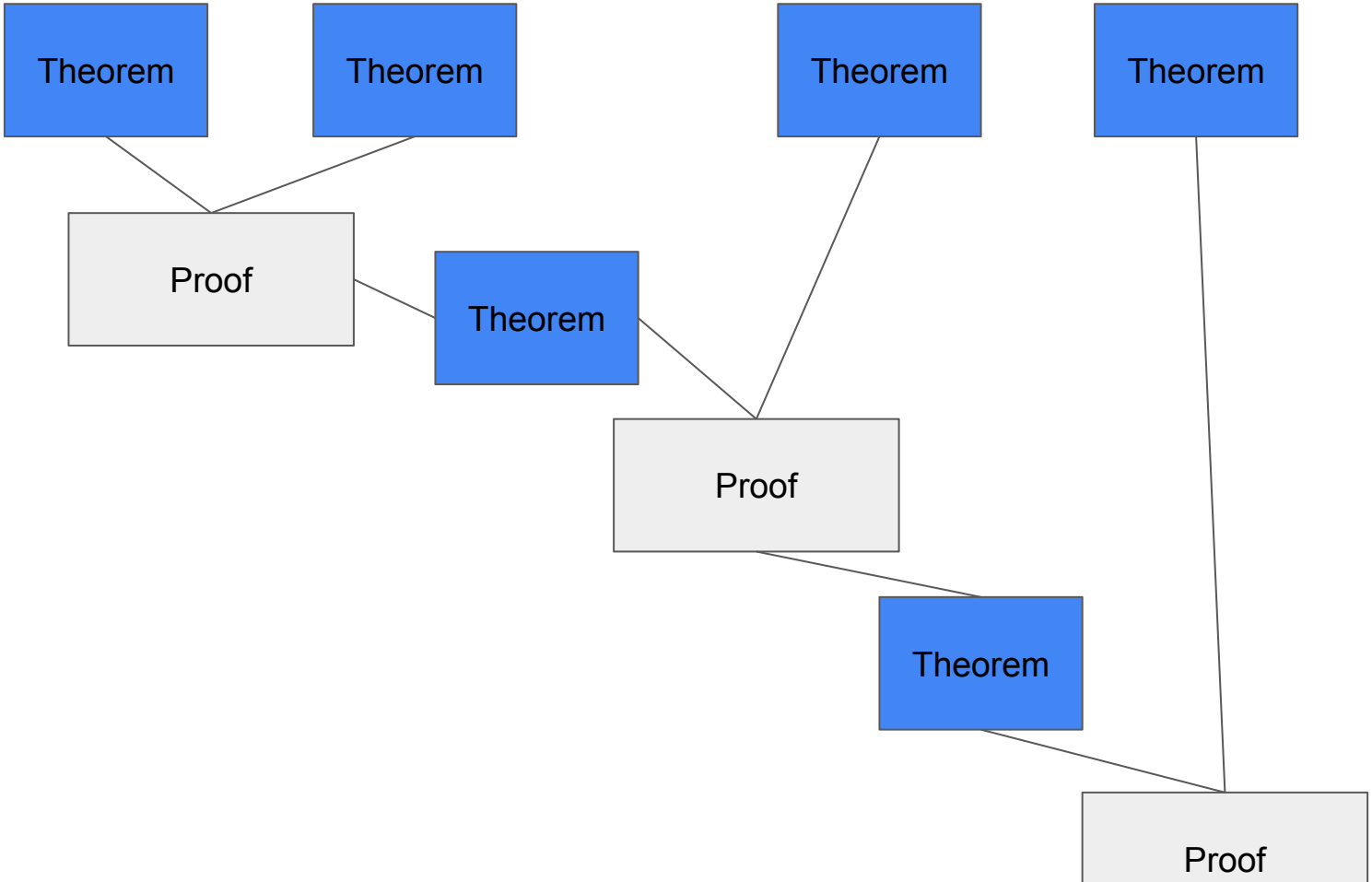
Theorem

- I have more than 3 best friends

Two kinds of proof:

- Formal Proof
- ZK Proof

  (to show formal proof was computed correctly)


Theorem
Theorem
Theorem
Theorem
Proof
Theorem
Proof
Theorem
Proof

```rust
#[derive(Clone, Debug, Serialize, Deserialize, PartialEq)]
pub enum Statement {
    None,
    ValueOf(AnchoredKey, ScalarOrVec),
    Equal(AnchoredKey, AnchoredKey),
    NotEqual(AnchoredKey, AnchoredKey),
    Gt(AnchoredKey, AnchoredKey),
    Lt(AnchoredKey, AnchoredKey),
    Contains(AnchoredKey, AnchoredKey),
    SumOf(AnchoredKey, AnchoredKey, AnchoredKey),
    ProductOf(AnchoredKey, AnchoredKey, AnchoredKey),
    MaxOf(AnchoredKey, AnchoredKey, AnchoredKey),
}
```

```rust
#[derive(Clone, Debug, Serialize, Deserialize, PartialEq)]
pub enum Statement {
    None,
    ValueOf(AnchoredKey, ScalarOrVec),
    Equal(AnchoredKey, AnchoredKey),
    NotEqual(AnchoredKey, AnchoredKey),
    Gt(AnchoredKey, AnchoredKey),
    Lt(AnchoredKey, AnchoredKey),
    Contains(AnchoredKey, AnchoredKey),
    SumOf(AnchoredKey, AnchoredKey, AnchoredKey),
    ProductOf(AnchoredKey, AnchoredKey, AnchoredKey),
    MaxOf(AnchoredKey, AnchoredKey, AnchoredKey),
}
```

```rust
#[derive(Clone, Debug, Serialize, Deserialize, PartialEq)]
pub enum Statement {
    None,
    ValueOf(AnchoredKey, ScalarOrVec),
    Equal(AnchoredKey, AnchoredKey),
    NotEqual(AnchoredKey, AnchoredKey),
    Gt(AnchoredKey, AnchoredKey),
    Lt(AnchoredKey, AnchoredKey),
    Contains(AnchoredKey, AnchoredKey),
    SumOf(AnchoredKey, AnchoredKey, AnchoredKey),
    ProductOf(AnchoredKey, AnchoredKey, AnchoredKey),
    MaxOf(AnchoredKey, AnchoredKey, AnchoredKey),
}
```

```rust
#[derive(Clone, Debug, Serialize, Deserialize, PartialEq)]
pub enum Statement {
    None,
    ValueOf(AnchoredKey, ScalarOrVec),
    Equal(AnchoredKey, AnchoredKey),
    NotEqual(AnchoredKey, AnchoredKey),
    Gt(AnchoredKey, AnchoredKey),
    Lt(AnchoredKey, AnchoredKey),
    Contains(AnchoredKey, AnchoredKey),
    SumOf(AnchoredKey, AnchoredKey, AnchoredKey),
    ProductOf(AnchoredKey, AnchoredKey, AnchoredKey),
    MaxOf(AnchoredKey, AnchoredKey, AnchoredKey),
}
```

```rust
#[derive(Clone, Debug)]
pub enum Operation<S: StatementOrRef> {
    None,
    NewEntry(Entry),
    CopyStatement(S),
    EqualityFromEntries(S, S),
    NonequalityFromEntries(S, S),
    GtFromEntries(S, S),
    LtFromEntries(S, S),
    TransitiveEqualityFromStatements(S, S),
    GtToNonequality(S),
    LtToNonequality(S),
    ContainsFromEntries(S, S),
    RenameContainedBy(S, S),
    SumOf(S, S, S),
    ProductOf(S, S, S),
    MaxOf(S, S, S),
}
```

```
#[derive(Clone, Debug)]
pub enum Operation<S: StatementOrRef> {
    None,
    NewEntry(Entry),
    CopyStatement(S),
    EqualityFromEntries(S, S),
    NonequalityFromEntries(S, S),
    GtFromEntries(S, S),
    LtFromEntries(S, S),
    TransitiveEqualityFromStatements(S, S),
    GtToNonequality(S),
    LtToNonequality(S),
    ContainsFromEntries(S, S),
    RenameContainedBy(S, S),
    SumOf(S, S, S),
    ProductOf(S, S, S),
    MaxOf(S, S, S),
}
```

Entry1, Entry2 *=EqFromEntries⇒* Eq(Entry1, Entry2)

Iff Entry1 == Entry2

ZKP for POD2

A ZKVM has circuits proving the state transition of a CPU architecture (RISC-V, WASM, etc)

This ZK Logic VM has **circuits proving the logical operation on statements is done correctly**.

A cryptographically-verified theorem prover if you want!

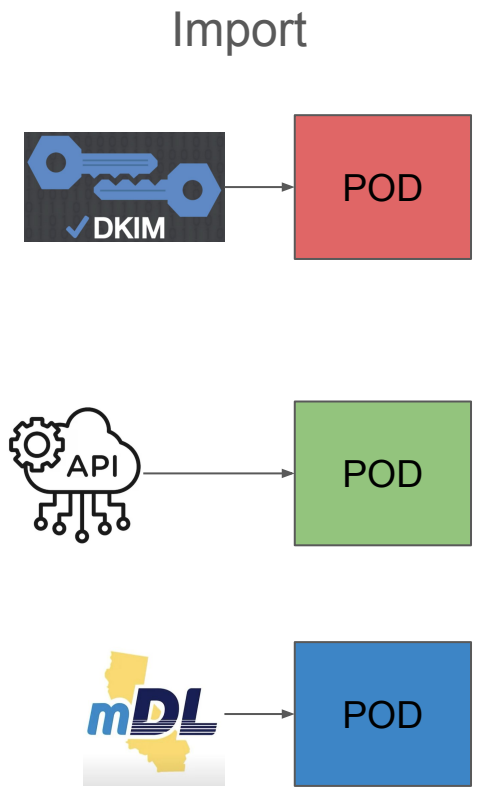
Challenge 2: Import

Introducing external cryptographic data:

- Turn any verifiable data into `ValueOf` statements

```
ValueOf(AnchoredKey, ScalarOrVec),
```

- With special metadata (`_signer`, `_url`, `_timestamp`, etc)
- Which can then be further combined like any other POD



TODD

JMP	VIB	SPD	INT	BTY
03	SADG	00	00	09

Collapse

Todd the weathered fisherman spends his days in quiet solitude, angling on the Ribbington Docks. Unbeknownst to many, he was once a great general whose campaign helped quell the Salamander Uprising. With each cast of his rod into the Sea, Todd's memories resurface, intertwining the tranquility of the present with the guilt of his past — leaving him a silent sentinel of a time long gone.

Signed at **11/13/2024**

View as POD

What if I want to compute with one of my Frog?

TODD

JMP	VIB	SPD	INT	BTY
03	SADG	00	00	09

Collapse


```
{
  "name": "Todd",
  "description": "Todd the weathered fisher…
  "imageUrl": "https://api.zupass.org/frogc…
  "frogId": 54,
  "biome": 4,
  "rarity": 3,
  "temperament": 16,
  "jump": 3,
  "speed": 0,
  "intelligence": 0,
  "beauty": 9,
  "timestampSigned": 1731478558016,
  "ownerSemaphoreId": "JBhIIHjbIGcZVKZFgz+B…
  "ownerEddsaPublicKey": null,
  "signature": "Hnt4sQFDbY9U3UQaJW9AIMThEdm…
  "signerPublicKey": "4sGycsjU8rG8FzKyvZd9h…
}
```


View as Frog

Frogs are POD1, foreign cryptographic data from the perspective of the POD2 system (different curve, signing algorithm, merklelization, etc).

Yet, they can be imported via an *Introduction Gadget* for POD1

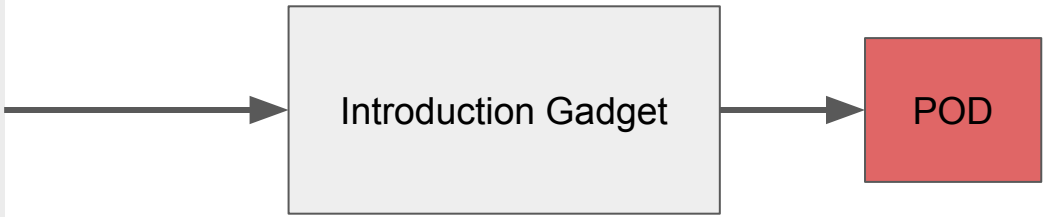
TODD

JMP	VIB	SPD	INT	BTY
03	SADG	00	00	09

Collapse


```
{
  "name": "Todd",
  "description": "Todd the weathered fisher…
  "imageUrl": "https://api.zupass.org/frogc…
  "frogId": 54,
  "biome": 4,
  "rarity": 3,
  "temperament": 16,
  "jump": 3,
  "speed": 0,
  "intelligence": 0,
  "beauty": 9,
  "timestampSigned": 1731478558016,
  "ownerSemaphoreId": "JBhIIHjbIGcZVKZFgz+B…
  "ownerEddsaPublicKey": null,
  "signature": "Hnt4sQFDbY9U3UQaJW9AIMThEdm…
  "signerPublicKey": "4sGycsjU8rG8FzKyvZd9h…
}
```


View as Frog



- Verify the Eddsa sig is correct.
  - Store the signer into a ValueOf statement with key `_signer`
- Copy each POD entry (via merkle proof) and create a ValueOf statement matching the key and value.

EMEI MOUSTACHE TOAD

JMP	VIB	SPD	INT	BTY
04	CALM	00	06	07

TODD

JMP	VIB	SPD	INT	BTY
03	SADG	00	00	09

Introduction Gadget

Introduction Gadget

POD

POD

Proof

POD

`GT(Todd.Bty, Moustache.Bty)`

“Todd is more beautiful than a Emei Moustache Toad

Bank Account

Receipts

Challenge 3: Making building easy

```
OpCmd::new(Op::NewEntry(entry9.clone()), "entry for claimed sum"),
OpCmd::new(
    Op::SumOf(
        StatementRef::new("_SELF", "VALUEOF:entry for claimed sum"),
        StatementRef::new("oraclePODParent", "VALUEOF:p1-apple"),
        StatementRef::new("oraclePODParent", "VALUEOF:a scalar entry"),
    ),
    "sumof entry",
),
OpCmd::new(
    Op::CopyStatement(StatementRef::new(
        "oraclePODParent",
        "VALUEOF:schnorrPOD2-signer",
    )),
    "p2-signer",
),
OpCmd::new(
    Op::GtToNonequality(StatementRef::new(
        "oraclePODParent",
        "GT:apple banana comparison",
    )),
    "apple banana nonequality with gt",
),
OpCmd::new(
    Op::LtToNonequality(StatementRef::new(
        "oraclePODParent",
        "LT:banana apple comparison",
    )),
    "apple banana nonequality with lt",
),
```

Making sense of PODs or crafting them is tedious and easy to get wrong

Users of the system need to:

- Craft a sequence of Logic OPCODE to make new PODs

```
Dependencies
-------------------------------------------------------------------------------
  From origin_4 (unknown)
    └─ EQUAL:from_pod_3474489270196213218_statement_7: id (value not found)
  From origin_3 (unknown)
    └─ EQUAL:from_pod_3474489270196213218_statement_7: id (value not found)
  From pod_3474489270196213218 (PLONKY)
    └─ in EQUAL:statement_3: timestamp-issued → ValueOf(timestamp-issued = Scalar(12219120042))
    └─ in EQUAL:statement_1: zip-code → ValueOf(zip-code = Scalar(12001))
    └─ in VALUEOF:from_pod_3474489270196213218_constant_18: 18 → ValueOf(18 = Scalar(18))

POD Statements (PLONKY)
-------------------------------------------------------------------------------
  └─ EQUAL:from_pod_3474489270196213218_statement_7 origin_4:id = origin_3:id
  └─ VALUEOF:statement_2 timestamp-issued = Scalar(12219120042)
  └─ EQUAL:statement_3 pod_3474489270196213218:timestamp-issued = timestamp-issued
  └─ VALUEOF:constant_100000 100000 = Scalar(100000)
  └─ VALUEOF:statement_0 zip-code = Scalar(12001)
  └─ GT:from_pod_3474489270196213218_statement_6 origin_4:age > pod_3474489270196213218:18
  └─ EQUAL:statement_1 pod_3474489270196213218:zip-code = zip-code
  └─ GT:statement_4 pod_3474489270196213218:salary > 100000
  └─ VALUEOF:from_pod_3474489270196213218_constant_18 pod_3474489270196213218:18 = Scalar(18)
```

Making sense of PODs or crafting them is tedious and easy to get wrong

Users of the system need to:

- Craft a sequence of Logic OPCODE to make new PODs
- Make sure statements in PODs match what they are asking to be proven when they receive a POD.
- Etc.

PEX

- A logic programming language to create PODs and verify them
- Currently a LISP, to make our life easier.
- Interactive development loop.

```
>⟩ [createpod id-card
::: age 26
::: zip-code 12001
::: id 1847202750]

Created new POD:

POD Statements (SCHNORR16)
```

```
  └─ VALUEOF:id id = Scalar(1847202750)
  └─ VALUEOF:age age = Scalar(26)
  └─ VALUEOF:_signer _signer = Scalar(15682816468093782297)
  └─ VALUEOF:zip-code zip-code = Scalar(12001)
```

```
>) [createpod citizen-salary
::: [define [age zip-code id-card-id] [pod? [age] [zip-code] [id]]]
::: [define [salary timestamp-issued paystub-id] [pod? [salary] [timestamp-issued] [id]]]
::: zip-code zip-code
::: salary salary
::: timestamp-issued timestamp-issued
::: [> age 18]
::: [= id-card-id paystub-id]]

Dependencies
────────────────────────────────
  From pod_11547658291100730763 (SCHNORR16)
    └─ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
    └─ in EQUAL:statement_1: zip-code → ValueOf(zip-code = Scalar(12001))
  From pod_1091298580668582930 (SCHNORR16)
    └─ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
    └─ in EQUAL:statement_5: timestamp-issued → ValueOf(timestamp-issued = Scalar(1219120042))
    └─ in EQUAL:statement_3: salary → ValueOf(salary = Scalar(175000))

POD Statements (PLONKY)
────────────────────────────────
   └─ GT:statement_6 pod_11547658291100730763:age > 18
   └─ VALUEOF:statement_0 zip-code = Scalar(12001)
   └─ EQUAL:statement_7 pod_11547658291100730763:id = pod_1091298580668582930:id
   └─ VALUEOF:constant_18 18 = Scalar(18)
   └─ EQUAL:statement_5 pod_1091298580668582930:timestamp-issued = timestamp-issued
   └─ EQUAL:statement_3 pod_1091298580668582930:salary = salary
   └─ VALUEOF:statement_2 salary = Scalar(175000)
   └─ VALUEOF:statement_4 timestamp-issued = Scalar(1219120042)
   └─ EQUAL:statement_1 pod_11547658291100730763:zip-code = zip-code
```

```
>⟩ [createpod citizen-salary
::: [define [age zip-code id-card-id] [pod? [age] [zip-code] [id]]]
::: [define [salary timestamp-issued paystub-id] [pod? [salary] [timestamp-issued] [id]]]
::: zip-code zip-code
::: salary salary
::: timestamp-issued timestamp-issued
::: [> age 18]
::: [= id-card-id paystub-id]]

Dependencies
------------

  From pod_11547658291100730763 (SCHNORR16)
    └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
    └ in EQUAL:statement_1: zip-code → ValueOf(zip-code = Scalar(12001))
  From pod_1091298580668582930 (SCHNORR16)
    └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
    └ in EQUAL:statement_5: timestamp-issued → ValueOf(timestamp-issued = Scalar(12219120042))
    └ in EQUAL:statement_3: salary → ValueOf(salary = Scalar(175000))

POD Statements (PLONKY)
-----------------------

  └ GT:statement_6 pod_11547658291100730763:age > 18
  └ VALUEOF:statement_0 zip-code = Scalar(12001)
  └ EQUAL:statement_7 pod_11547658291100730763:id = pod_1091298580668582930:id
  └ VALUEOF:constant_18 18 = Scalar(18)
  └ EQUAL:statement_5 pod_1091298580668582930:timestamp-issued = timestamp-issued
  └ EQUAL:statement_3 pod_1091298580668582930:salary = salary
  └ VALUEOF:statement_2 salary = Scalar(175000)
  └ VALUEOF:statement_4 timestamp-issued = Scalar(12219120042)
  └ EQUAL:statement_1 pod_11547658291100730763:zip-code = zip-code
```

```
>) [createpod citizen-salary
::: [define [age zip-code id-card-id] [pod? [age] [zip-code] [id]]]
::: [define [salary timestamp-issued paystub-id] [pod? [salary] [timestamp-issued] [id]]]
::: zip-code zip-code
::: salary salary
::: timestamp-issued timestamp-issued
::: [> age 18]
::: [= id-card-id paystub-id]]

Dependencies
------------

  From pod_11547658291100730763 (SCHNORR16)
    └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
    └ in EQUAL:statement_1: zip-code → ValueOf(zip-code = Scalar(12001))
  From pod_1091298580668582930 (SCHNORR16)
    └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
    └ in EQUAL:statement_5: timestamp-issued → ValueOf(timestamp-issued = Scalar(12219120042))
    └ in EQUAL:statement_3: salary → ValueOf(salary = Scalar(175000))

POD Statements (PLONKY)
-----------------------

  └ GT:statement_6 pod_11547658291100730763:age > 18
  └ VALUEOF:statement_0 zip-code = Scalar(12001)
  └ EQUAL:statement_7 pod_11547658291100730763:id = pod_1091298580668582930:id
  └ VALUEOF:constant_18 18 = Scalar(18)
  └ EQUAL:statement_5 pod_1091298580668582930:timestamp-issued = timestamp-issued
  └ EQUAL:statement_3 pod_1091298580668582930:salary = salary
  └ VALUEOF:statement_2 salary = Scalar(175000)
  └ VALUEOF:statement_4 timestamp-issued = Scalar(12219120042)
  └ EQUAL:statement_1 pod_11547658291100730763:zip-code = zip-code
```

```
>) [createpod citizen-salary
::: [define [age zip-code id-card-id] [pod? [age] [zip-code] [id]]]
::: [define [salary timestamp-issued paystub-id] [pod? [salary] [timestamp-issued] [id]]]
::: zip-code zip-code
::: salary salary
::: timestamp-issued timestamp-issued
::: [> age 18]
::: [= id-card-id paystub-id]]
```

Dependencies

```
From pod_11547658291100730763 (SCHNORR16)
  └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
  └ in EQUAL:statement_1: zip-code → ValueOf(zip-code = Scalar(12001))
From pod_1091298580668582930 (SCHNORR16)
  └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
  └ in EQUAL:statement_5: timestamp-issued → ValueOf(timestamp-issued = Scalar(12219120042))
  └ in EQUAL:statement_3: salary → ValueOf(salary = Scalar(175000))
```

POD Statements (PLONKY)

```
└ GT:statement_6 pod_11547658291100730763:age > 18
└ VALUEOF:statement_0 zip-code = Scalar(12001)
└ EQUAL:statement_7 pod_11547658291100730763:id = pod_1091298580668582930:id
└ VALUEOF:constant_18 18 = Scalar(18)
└ EQUAL:statement_5 pod_1091298580668582930:timestamp-issued = timestamp-issued
└ EQUAL:statement_3 pod_1091298580668582930:salary = salary
└ VALUEOF:statement_2 salary = Scalar(175000)
└ VALUEOF:statement_4 timestamp-issued = Scalar(12219120042)
└ EQUAL:statement_1 pod_11547658291100730763:zip-code = zip-code
```

```
>⟩ [createpod citizen-salary
::: [define [age zip-code id-card-id] [pod? [age] [zip-code] [id]]]
::: [define [salary timestamp-issued paystub-id] [pod? [salary] [timestamp-issued] [id]]]
::: zip-code zip-code
::: salary salary
::: timestamp-issued timestamp-issued
::: [> age 18]
::: [= id-card-id paystub-id]]
```

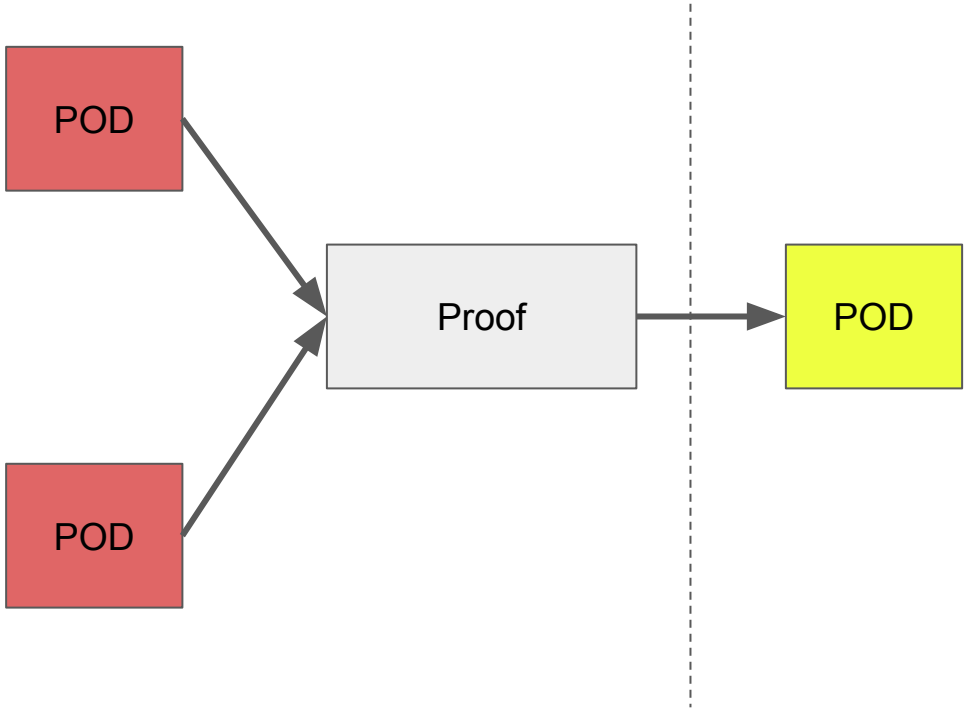
Dependencies

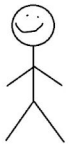
```
From pod_11547658291100730763 (SCHNORR16)
  └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
  └ in EQUAL:statement_1: zip-code → ValueOf(zip-code = Scalar(12001))
From pod_1091298580668582930 (SCHNORR16)
  └ in EQUAL:statement_7: id → ValueOf(id = Scalar(1847202750))
  └ in EQUAL:statement_5: timestamp-issued → ValueOf(timestamp-issued = Scalar(12219120042))
  └ in EQUAL:statement_3: salary → ValueOf(salary = Scalar(175000))
```

POD Statements (PLONKY)

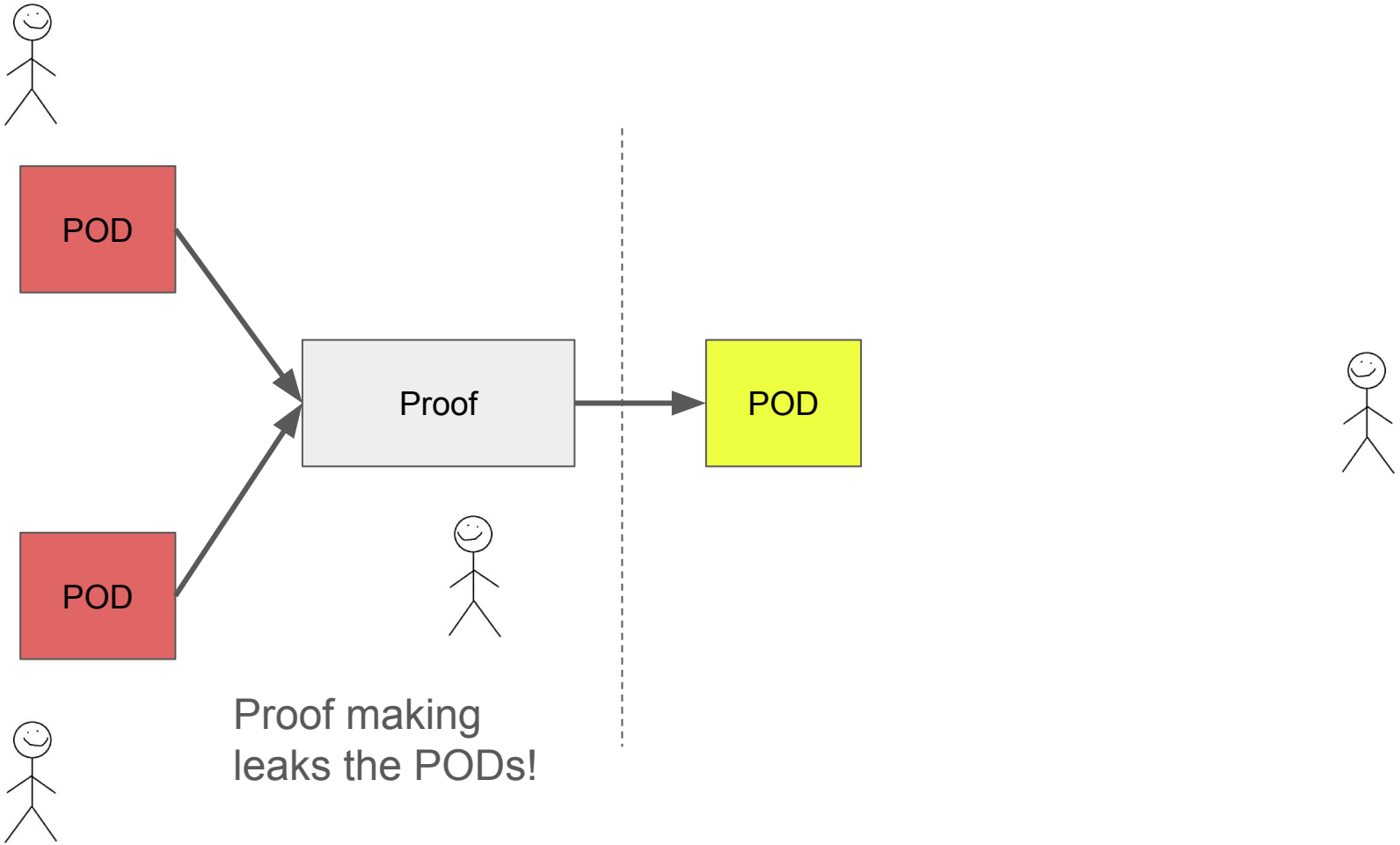
```
└ GT:statement_6 pod_11547658291100730763:age > 18
└ VALUEOF:statement_0 zip-code = Scalar(12001)
└ EQUAL:statement_7 pod_11547658291100730763:id = pod_1091298580668582930:id
└ VALUEOF:constant_18 18 = Scalar(18)
└ EQUAL:statement_5 pod_1091298580668582930:timestamp-issued = timestamp-issued
└ EQUAL:statement_3 pod_1091298580668582930:salary = salary
└ VALUEOF:statement_2 salary = Scalar(175000)
└ VALUEOF:statement_4 timestamp-issued = Scalar(12219120042)
└ EQUAL:statement_1 pod_11547658291100730763:zip-code = zip-code
```

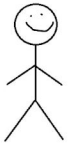
Challenge 4: Multi-party POD creation

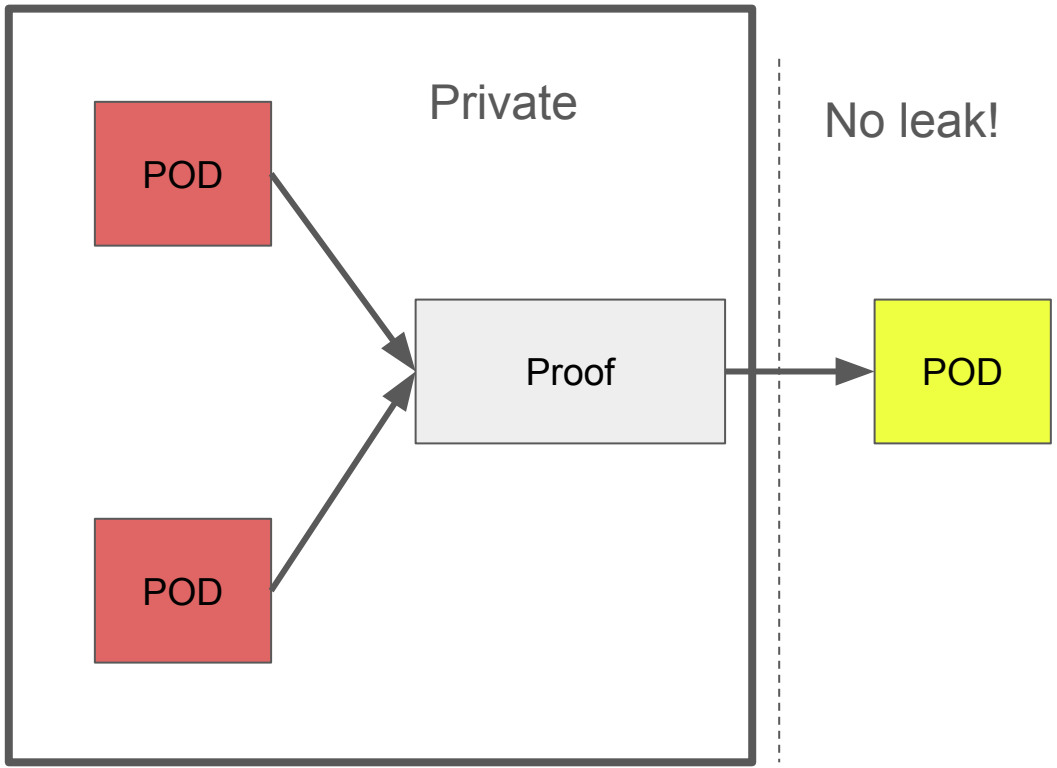


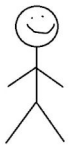


Red PODs are private


POD
POD
Proof
POD
Proof making
leaks the PODs!



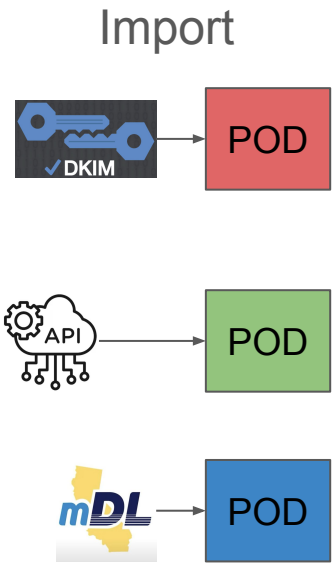




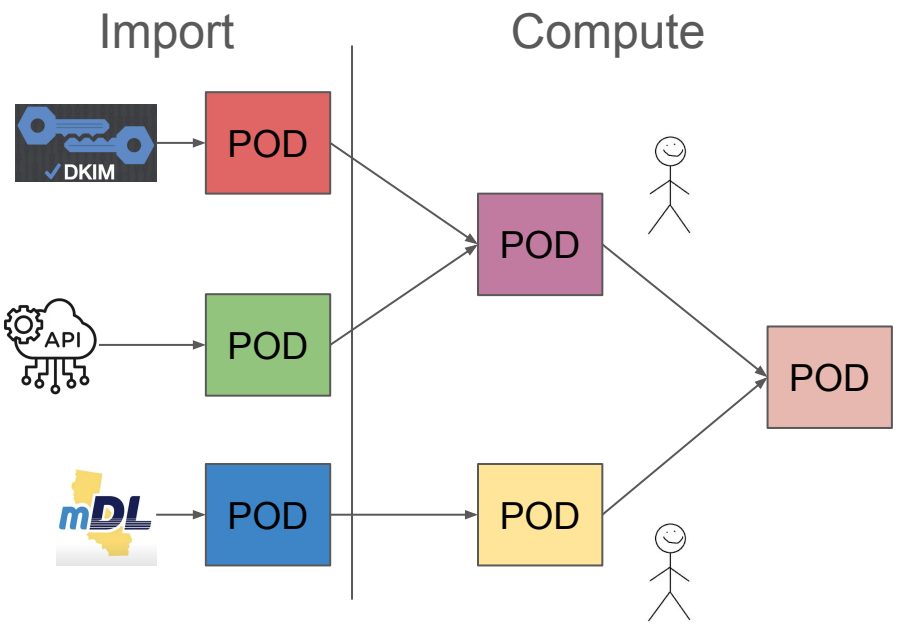
- Executing Logical operation in multi-party FHE or verifiable Garbled Circuit
- We want to have **verifiable FHE** without multiplicative overhead
- Very much an area of research. Other possible solutions: co-SNARK, etc

POD2

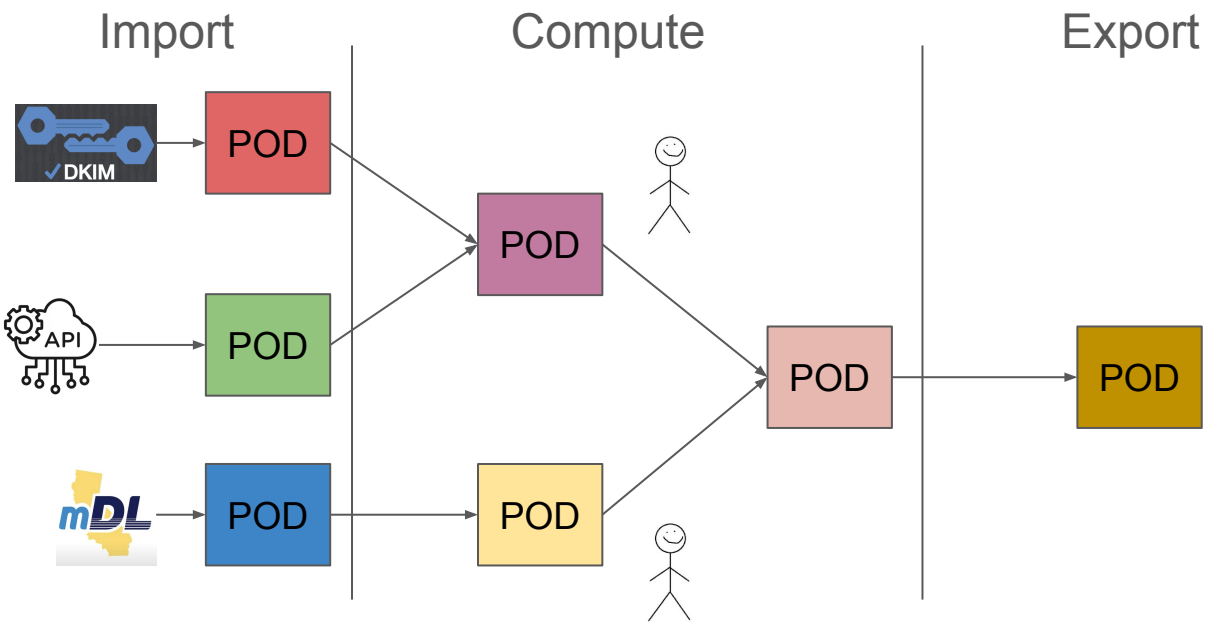
POD2



POD2



POD2



POD2

1. Logic-based claims and operations
2. Introduction system for importing cryptographic data
3. Language for creating and verifying PODs
4. Confidential execution of the logic for high-stake use-cases

Active development in 2025. Aiming for production readiness by end of next year, with pilots throughout the year.

If you want to learn more

POD1

2:30 PM **Tomorrow** in Classroom B

POD2 & PEX

Friday 12.30pm at Workshop - Breakout 2