Mastic: Private Weighted Heavy-Hitters and Attribute-Based Metrics

Tech Talk @ Google – 2024/10/3

Dimitris Mouris (Nillion)
Christopher Patton (Cloudflare)
Hannah Davis (Seagate)
Pratik Sarkar (Supra Research)
Nektarios G. Tsoutsos (University of Delaware)

Work at IETF

- Privacy Preserving Measurement (PPM)
  - Use cases
    - aggregation (à la Prio): average, variance, histograms, Bloom filters, linear regression, step of gradient descent
    - heavy hitters (à la Poplar): compute the set of the most frequent inputs (e.g., the most visited websites)
  - Privacy: distribute computation among multiple servers so that no one server sees any measurement in plaintext
  - Robustness: detect invalid measurements uploaded by misbehaving clients
- Verifiable Distributed Aggregation Functions (VDAFs): Specifications of Prio and Poplar
- Distributed Aggregation Protocol (DAP): Execution of a VDAF over HTTP

Where does Mastic fit?

- There are things we'd like Poplar or Prio to do:
  - *attribute-based* aggregation: average, variance, histograms, Bloom filters, linear regression, step of gradient descent *grouped by client attributes (e.g., user agent or geolocation)*
  - *weighted* heavy-hitters: compute the set of the ~~most frequent~~*heaviest weight* inputs (e.g., the ~~most visited websites~~*websites with the highest engagement*)
- Poplar has rough edges
  - Two rounds of communication for input validation (MPC multiplication w/ Beaver triples provided by clients)
  - Different field for leaf versus inner nodes of the IDPF tree, unsafe to use intermediate outputs

Design goals

- Ease of implementation: simplify design, reuse existing components where possible
- Deployability: one round of communication between aggregation servers instead of two
- Squeeze more functionality out of less code: *attribute-based* aggregation and *weighted* heavy-hitters

[MPD+25] Mouris et al. "Mastic: Private Weighted Heavy-Hitters and Attribute-Based Metrics." PETS 2025.

Agenda

1. Setting (functionality, architecture, security goals) ~10 minutes
2. Primitive #1: VIDPF ~15 minutes
3. Primitive #2: FLP ~3 minutes
4. Putting it all together ~3 minutes
5. Next steps ~5 minutes

Setting

Functionality

- Each client generates a ***measurement*** consisting of an ***input*** (an element of $\{0,1\}^n$) and its ***weight*** (an element of $\mathbb{F}^m$)
- For each ***prefix***, compute the ***total weight*** of all inputs beginning with the prefix

```
def mastic_func(measurements: list[tuple[Index, Weight]],
                prefixes: list[Index]) -> dict[Index, Weight]:
    '''
    Compute the total weight for each prefix for the set of
    measurements.
    '''
    r: dict[Index, Weight] = {}
    for (α, β) in measurements:
        for x in prefixes:
            if x.is_prefix(α):
                total_weight = r.setdefault(x, Weight(0))
                total_weight += β
    return r
```

Functionality

```python
def weighted_heavy_hitters(measurements: list[tuple[Index, Weight]],
                           threshold: Weight,
                           bit_len: int) -> list[Index]:
    '''
    Compute the weighted heavy hitters for the given threshold.
    '''
    prefixes = [Index(0), Index(1)]
    for level in range(bit_len):
        next_prefixes = []
        for (x, total_weight) in mastic_func(measurements, prefixes).items():
            if total_weight >= threshold:
                if level < bit_len - 1:
                    next_prefixes.append(x.left_child())
                    next_prefixes.append(x.right_child())
                else:
                    next_prefixes.append(x)
        prefixes = next_prefixes
    return sorted(prefixes)
```

Weighted heavy hitters uses Mastic as a subroutine over multiple rounds of aggregation

Use case #1: Ad attribution

- ***On-device attribution:*** browser keeps track of which ads are shown to the user: when the user makes a purchase, the browser determines which ad, if any, deserves credit.
  - Input: a unique identifier for the ad campaign (e.g., `"Bartlet for America"`)
  - Weight: whether a purchase was made (`Count`); how much the user spent (`Sum`); which category of product was purchased (`Histogram`); etc.

Use case #2: Browser telemetry

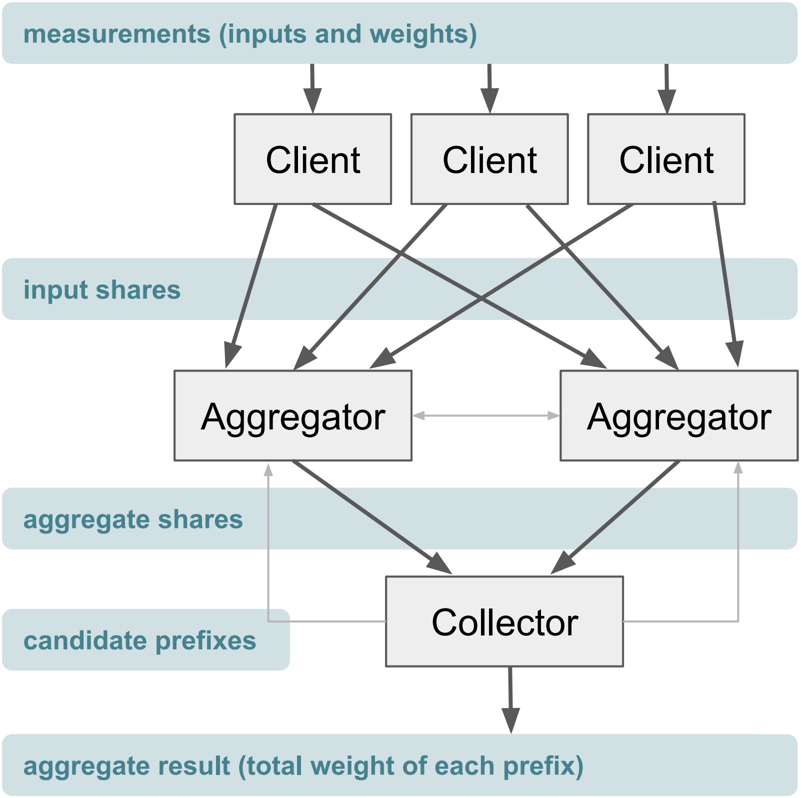
- Many use cases for Prio-style metrics in browsers, e.g.: page load metrics for a fixed set of "benchmark" websites (e.g., `google.com` or `zombo.com`)
  - Input: encodes whatever attributes we might want to break down by, eg., software version and geographic location
  - Weight: average (`Sum`) or distribution (`Histogram`) of load time

Use case #3: Network error logging

- NEL: To detect configuration issues, content-delivery networks like Cloudflare want to know the types of errors preventing eyeballs from connecting to their customers (DNS, TCP, TLS, etc.)
    - Input: the customer identified by a DNS name
    - Weight: indication of which error occurred (`Histogram`)

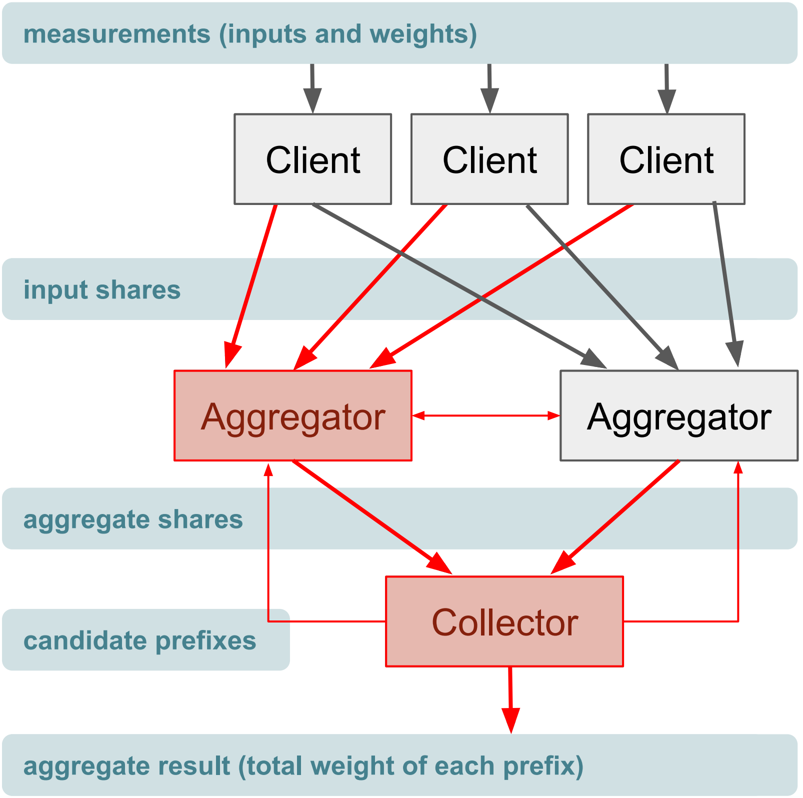
Architecture

- Clients ***shard*** their measurements
- Aggregators interact to validate the measurements, then compute shares of the total weight for each candidate prefix
- Collector ***unshards*** the aggregate result



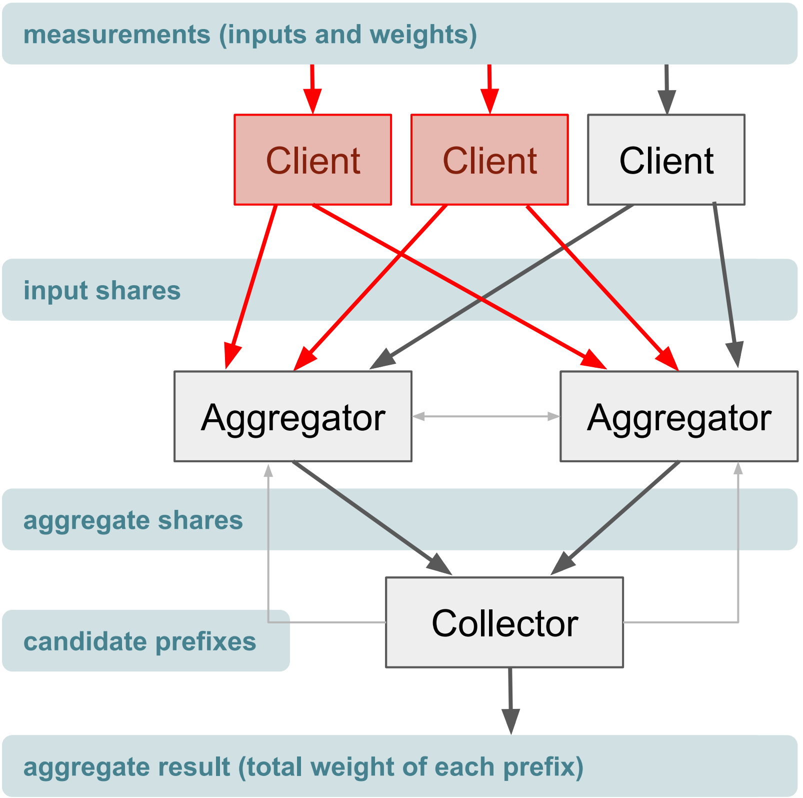
Security goal #1: Privacy

- Attacker controls one of the Aggregators and the Collector
- Attacker's view is efficiently ***simulatable*** given only aggregate results



Security goal #2: Robustness

- Attacker controls a subset of the clients and eavesdrops on the network
- Aggregate result computed by the collector is efficiently ***extractable*** from the input shares

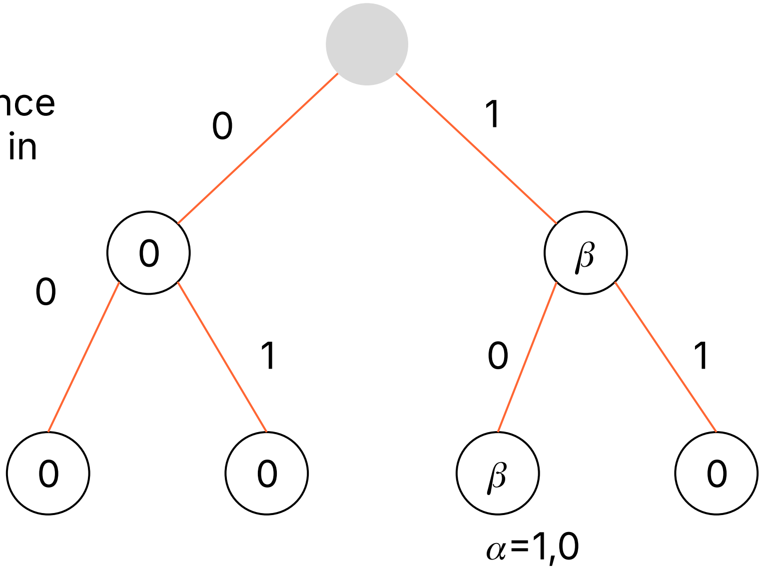


Primitive #1: VIDPF

CLOUDFLARE

Verifiable Incremental Distributed Point Function (VIDPF)

- ***point function:*** `F(x)=`β` if x==`α` else 0`
- ***incremental:*** `F(x)=`β` if x.is_prefix(`α`) else 0`
- ***distributed:*** client generates secret shares of `F` such that each aggregator computes a secret share of `F(x)` for any `x` without learning α or β
- ***verifiable:*** aggregators verify that, for any sequence of prefixes `prefixes`, `F(x)!=0` for at most one `x` in `prefixes` and the others are `0` (***onehot***)



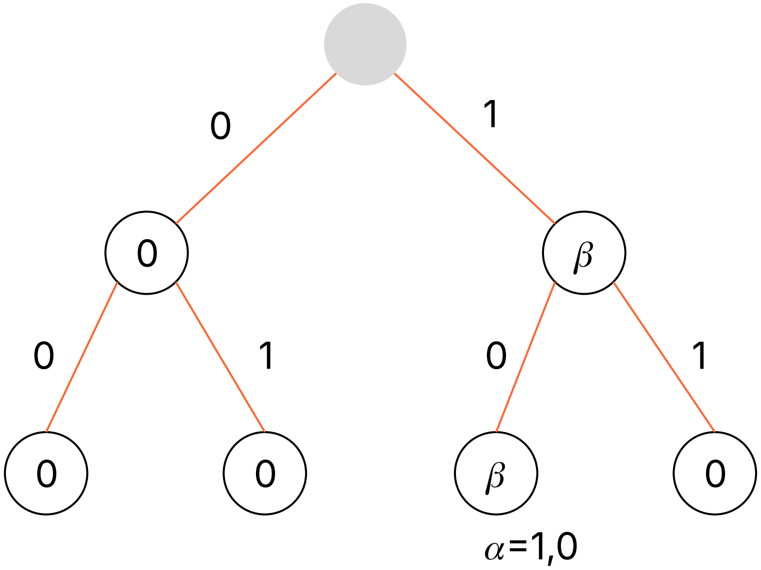
Verifiable Incremental Distributed Point Function (VIDPF)

Client: `(correction_words, [key0, key1]) = vidpf.gen(`α`, `β`)`

Aggregator 0: `(out0, proof0) = vidpf.eval(0, correction_words, key0, prefixes)`

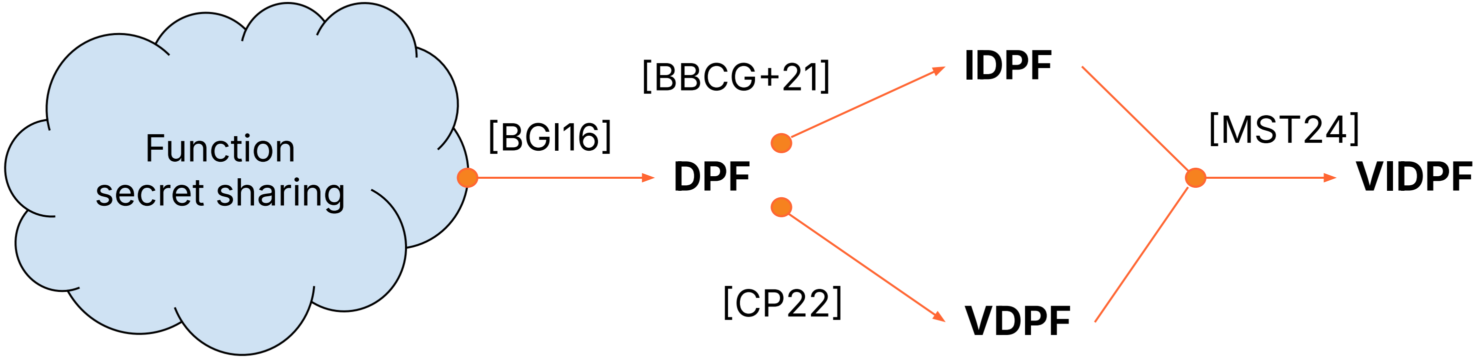
Aggregator 1: `(out1, proof1) = vidpf.eval(1, correction_words, key1, prefixes)`

verifiability: It's computationally infeasible to find `prefixes` for which `proof0 == proof1` but `out0 + out1` is not onehot.



CLOUDFLARE

A bit of history



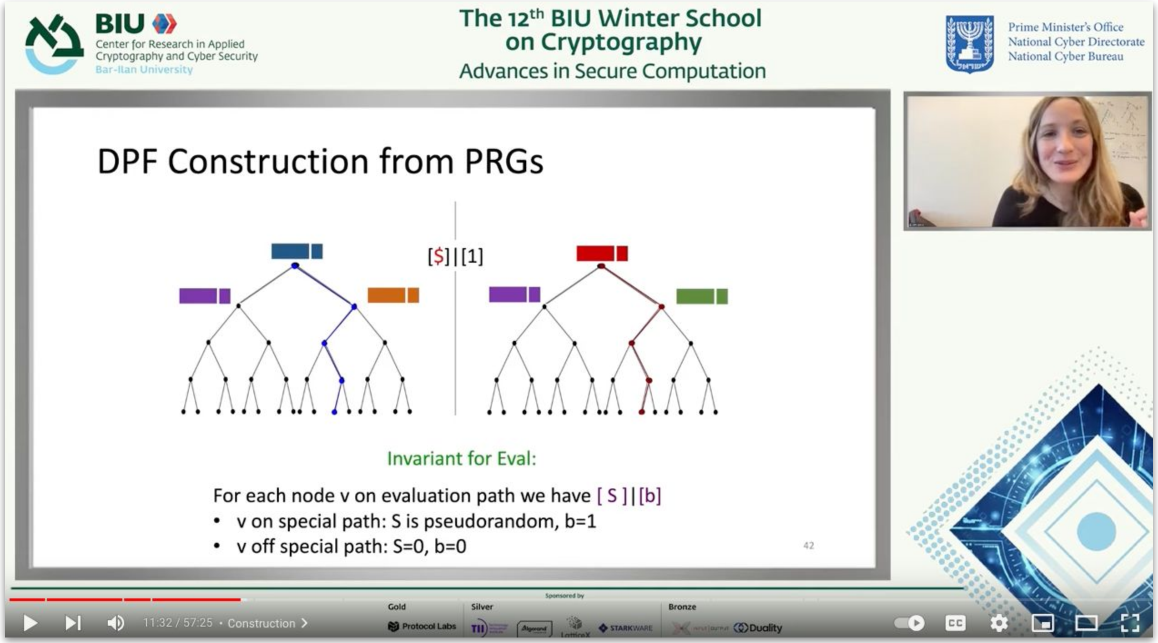
[BGI16] Boyle et al. "Function Secret Sharing: Improvements and Extensions." CCS 2016.

[BBCG+21] Boneh et al. "Lightweight Techniques for Private Heavy Hitters." IEEE S&P 2021.

[CP22] de Castro and Polychroniadou. "Lightweight, Maliciously Secure Verifiable Function Secret Sharing." Eurocrypt 2022.

[MST24] Mouris et al. "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries." PETS 2024.

The DPF of [BGI16]

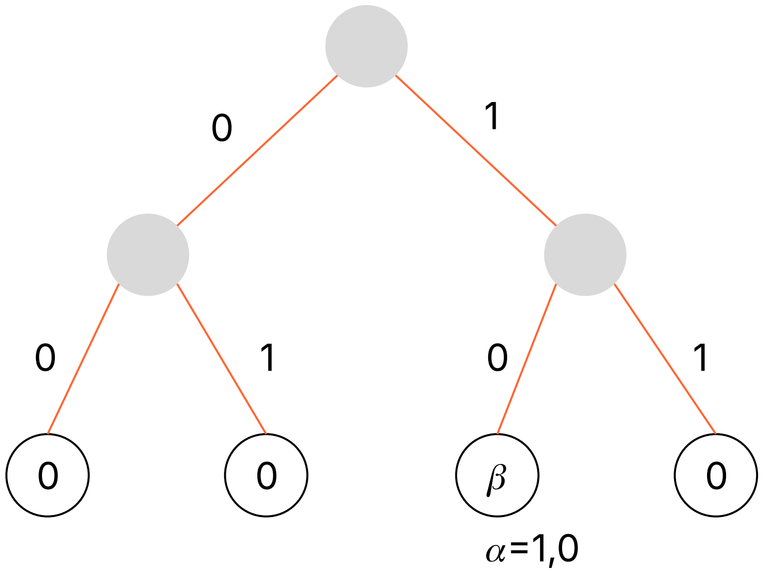


For a complete tutorial, check out Elette Boyle's series on function secret sharing for the 12th BIU Winter School

[BGI16] Boyle et al. "Function Secret Sharing: Improvements and Extensions." CCS 2016.

The DPF of [BGI16]

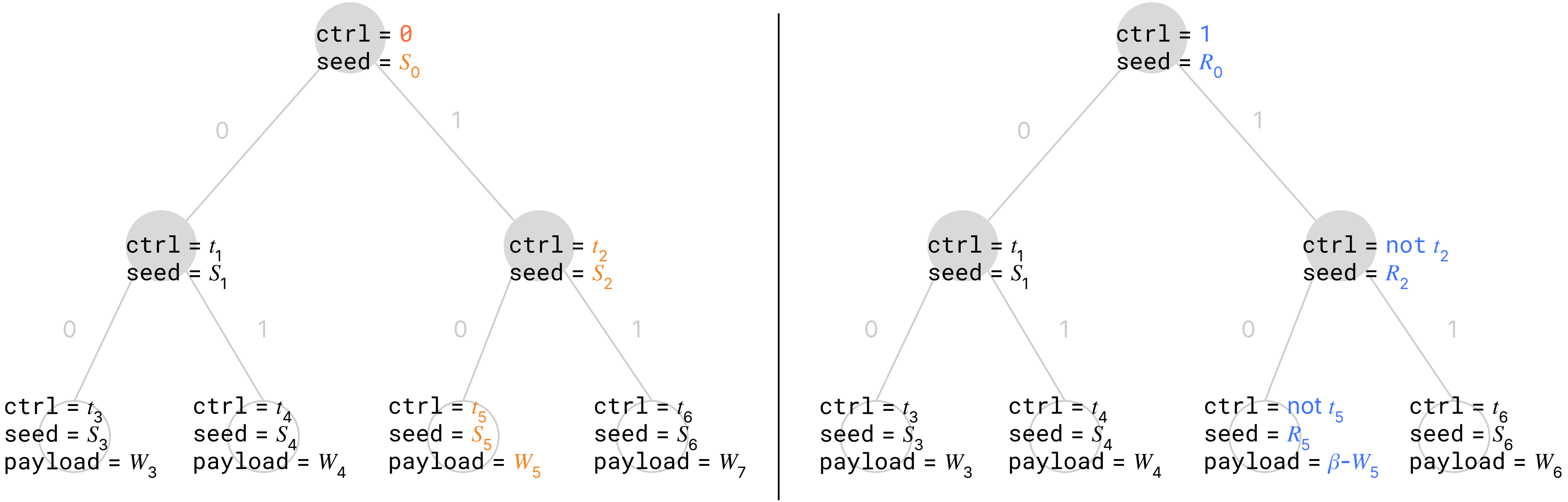
- Evaluation of input `x`:
    - Initialization:
        - `ctrl` = aggregator's ID (`0` or `1`)
        - `seed` = aggregator's key share
    - for each level `i`:
        - Extend the current `seed` into a `ctrl`/`seed` for each child node
        - If `x[i]==0` then select the left child; otherwise select the right child.
    - Share of the output is derived from the `seed` for the last level



[BGI16] Boyle et al. "Function Secret Sharing: Improvements and Extensions." CCS 2016.

The DPF of [BGI16]

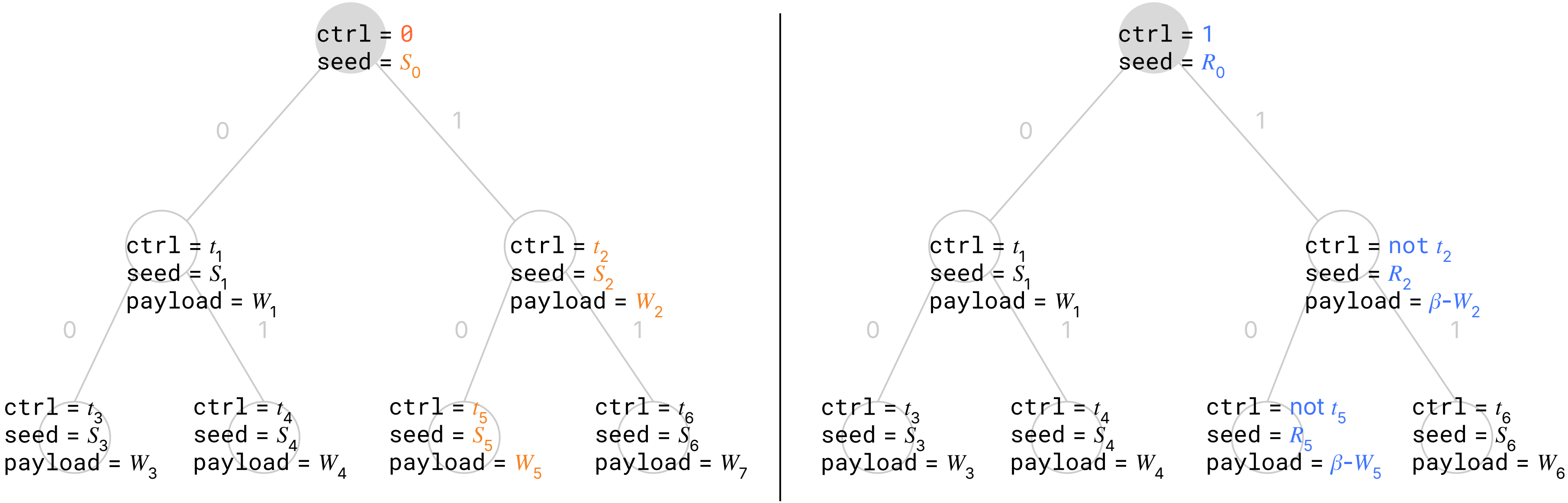
DPF invariant: if `x[:i].is_prefix(`α`)`, then the `ctrls` are shares of `1` and the `seeds` are distinct; otherwise, the `ctrls` are shares of `0` and the `seeds` are the same. (If `ctrl==1`, then "correct" with the level's ***correction word***.)



[BGI16] Boyle et al. "Function Secret Sharing: Improvements and Extensions." CCS 2016.

DPF → IDPF [BBCG+21]

Generate payloads for intermediate levels: use seed to derive the next seed and a share of the payload (after correction).

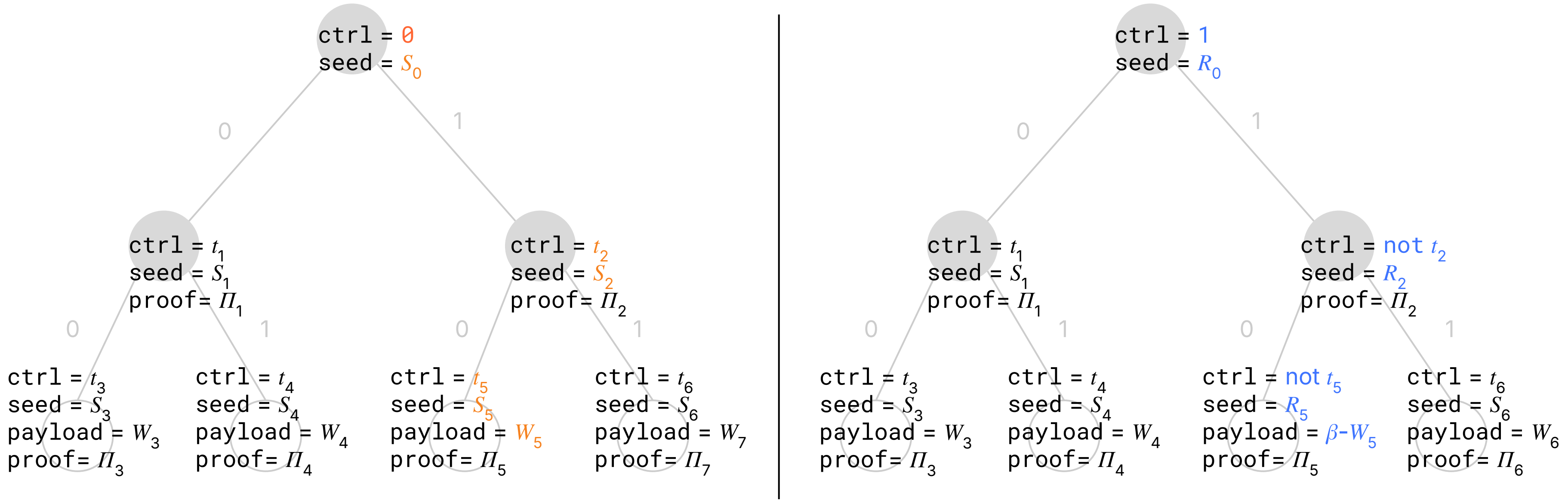


[BBCG+21] Boneh et al. "Lightweight Techniques for Private Heavy Hitters." IEEE S&P 2021.

CLOUDFLARE

DPF → VDPF [CP22]

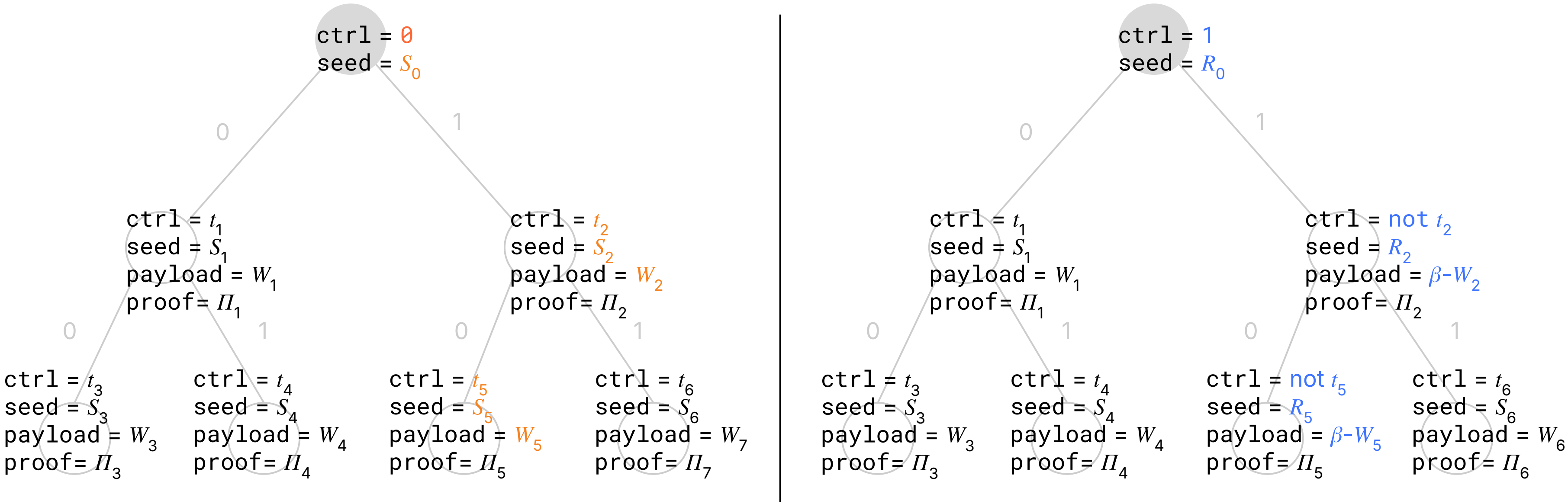
Verify the DPF invariant: let `proof = hash(seed,x[:i])` be the ***node proof;*** if the DPF invariant holds, then each aggregator computes the same node proof (after correction).



[CP22] de Castro and Polychroniadou. "Lightweight, Maliciously Secure Verifiable Function Secret Sharing." Eurocrypt 2022.

IDPF, VDPF → VIDPF [MST24]

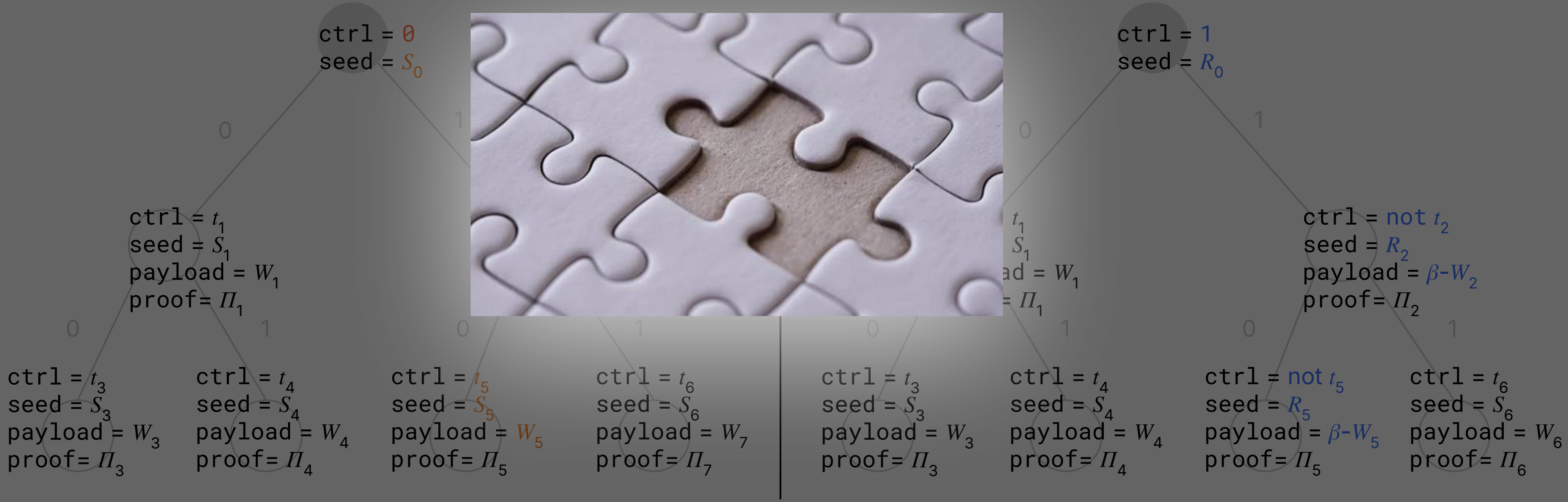
Generate intermediate payloads, verify the DPF invariant, and verify ***path consistency:*** the payload of each node traversed is equal to the sum of its children ⇒ β is the output for each prefix of α.



[MST24] Mouris et al. "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries." PETS 2024.

IDPF, VDPF → VIDPF [MST24]

Generate intermediate payloads, verify the DPF invariant, and verify ***path consistency:*** the payload of each node traversed is equal to the sum of its children ⇒ β is the output for each prefix of α.



[MST24] Mouris et al. "PLASMA: Private, Lightweight Aggregated Statistics against Malicious Adversaries." PETS 2024.

Primitive #2: FLP

Fully Linear Proof (FLP)

Zero-knowledge proof system on distributed data [BBCG+19]. Syntax here is as presented in [draft-irtf-cfrg-vdaf]:

Client: `proof = flp.prove(`β`)`; secret share `proof` into `proof0 + proof1` and β into `beta0 + beta1`

Aggregator 0: `verifier0 = flp.query(beta0, proof0)`

Aggregator 1: `verifier1 = flp.query(beta1, proof1)`

> ***proof verification:*** if `flp.decide(verifier0 + verifier1)` then the weight is valid with high probability.

[BBCG+19] Boneh et al. "Zero-Knowledge Proofs on Secret-Shared Data via Fully Linear PCPs". Crypto 2019.

[draft-irtf-cfrg-vdaf] Barnes et al. "Verifiable Distributed Aggregation Functions." CFRG draft, version 11.

Putting it all together

VIDPF + FLP = 

CLOUDFLARE

Weight type is determined by the FLP:

- `Count`: weight is `0` or `1` (same functionality as Poplar)
- `Sum`: weight is an integer in a bounded range
- `Histogram`: weight is a histogram with a fixed number of buckets

Client runs the sharding algorithm to get the ***public share*** (the correction words) sent to both aggregators and one ***input share*** for each aggregator.

```
def shard(self,
          measurement: tuple[Index, Weight],
          nonce: bytes):
    # Encode the weight according to the FLP.
    (α, weight) = measurement
    β = self.flp.encode(weight)

    # Generate VIDPF keys (depends on the nonce).
    (correction_words, keys) = self.vidpf.gen(α, β, nonce)

    # Generate FLP and split it into shares.
    proof = self.flp.prove(β)
    helper_seed = gen_rand(32)
    helper_proof_share = self.helper_proof_share(helper_seed)
    leader_proof_share = vec_sub(proof, helper_proof_share)

    return (correction_words,
            [(keys[0], leader_proof_share), # aggregator 0
             (keys[1], helper_seed)])       # aggregator 1
```

VIDPF + FLP = 

Aggregator, on input of the report from the client (correcton words, VIDPF key share, FLP proof share) and `prefixes` from the collector:

- Evaluate the VIDPF at `prefixes` to get output `output_share`, the secret shares of `F(x)` for each `x` in `prefixes`
- Interact with the co-Aggregator to check *(one roundtrip over the network)*
  - ***onehotness:*** Check that DPF invariant holds for `prefixes` ⇒ at most one of the outputs is β (the rest are `0`)
  - ***path consistency:*** For each node traversed in the prefix tree, check that the payload is equal to the sum of its children ⇒ `F(x)=`β for each prefix `x` of α
  - **weight validity**: Verify the FLP ⇒ β encodes a valid weight
- Aggregate `output_share`.

Next steps

CLOUDFLARE

Planned work

Status	Task
Done ✅	Initial design and analysis [MPD+25].
In progress	Implementation (github.com/divviup/libprio-rs) in collaboration with ISRG (Divvi Up), who maintains open source implementations of Prio and Poplar (Rust).
In progress	Specification [draft-mouris-cfrg-mastic]. Many improvements are being considered (github.com/jimouris/draft-mouris-cfrg-mastic/issues).
Planned	End-to-end analysis of the final specification, especially concrete security of VIDPF.

[MPD+25] Mouris et al. "Mastic: Private Weighted Heavy-Hitters and Attribute-Based Metrics." PETS 2025.

[draft-mouris-cfrg-mastic] Mouris et al. "The Mastic VDAF." Individual draft, version 03.

Status of Mastic at IETF

- [draft-mouris-cfrg-mastic] is an ***individual draft*** and has not been adopted by a working group
- We will pursue adoption if/when there is sufficient interest to deploy Mastic. Currently in a holding pattern:
    - Top priority right now is completing the core documents [draft-ietf-ppm-dap, draft-irtf-cfrg-vdaf]
    - CFRG currently has low bandwidth for new work
    - PPM is chartered to do protocol design, not develop new cryptographic algorithms (this is meant to be delegated to CFRG)
    - We considered replacing Poplar with Mastic, but the current consensus is to keep Poplar in the core VDAF draft.

[draft-mouris-cfrg-mastic] Mouris et al. "The Mastic VDAF." Individual draft, version 03.

[draft-ietf-ppm-dap] Geoghegan et al. "Distributed Aggregation Protocol for Privacy Preserving Measurement." PPM working group draft, version 11.

[draft-irtf-cfrg-vdaf] Barnes et al. "Verifiable Distributed Aggregation Functions." CFRG draft, version 11.

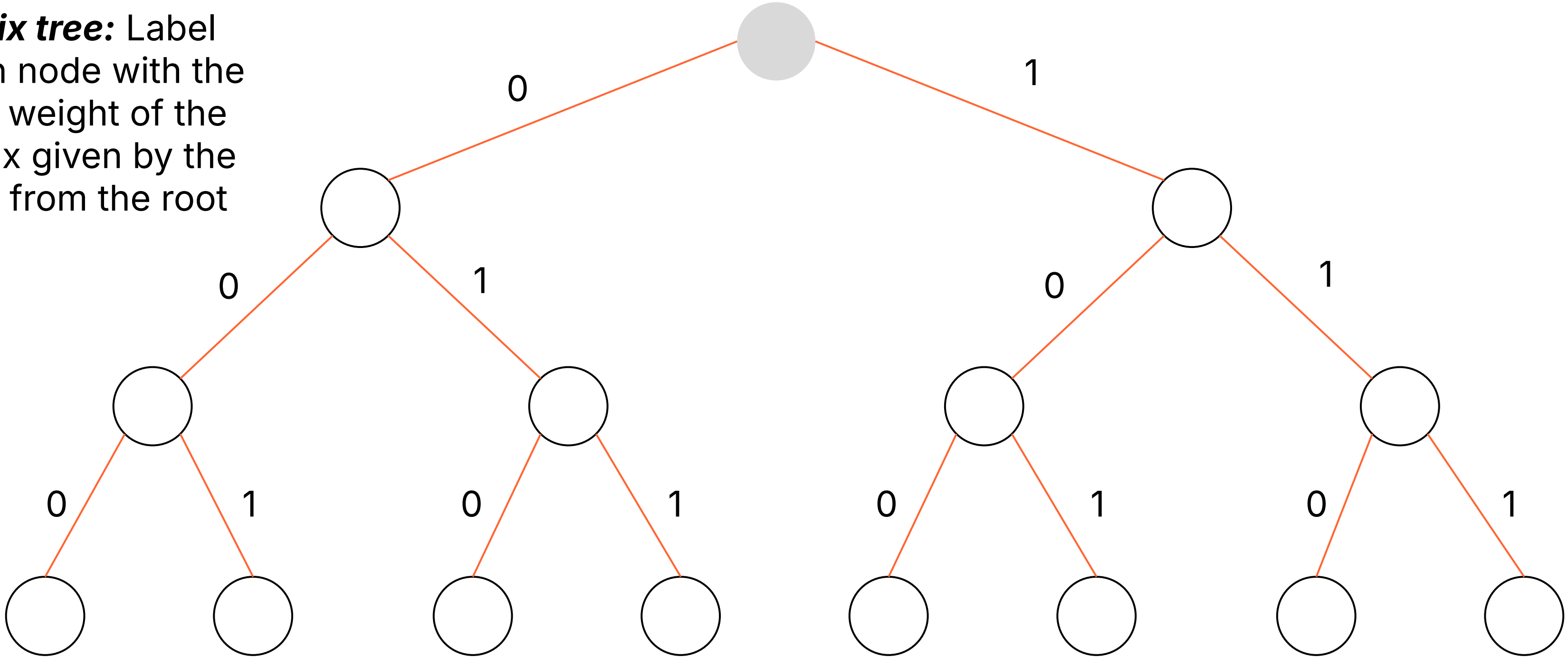
Thank you

cpatton@cloudflare.com

datatracker.ietf.org/doc/draft-mouris-cfrg-mastic

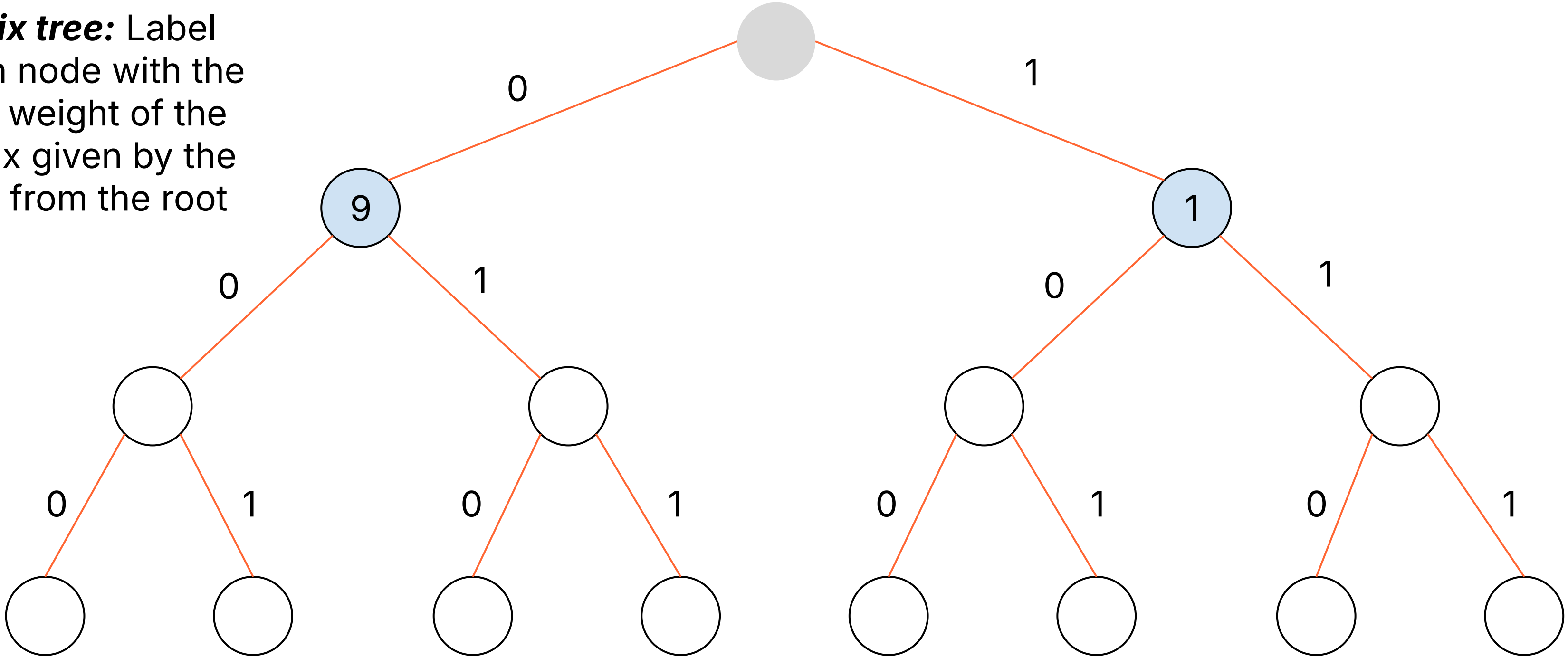
Functionality

prefix tree: Label each node with the total weight of the prefix given by the path from the root



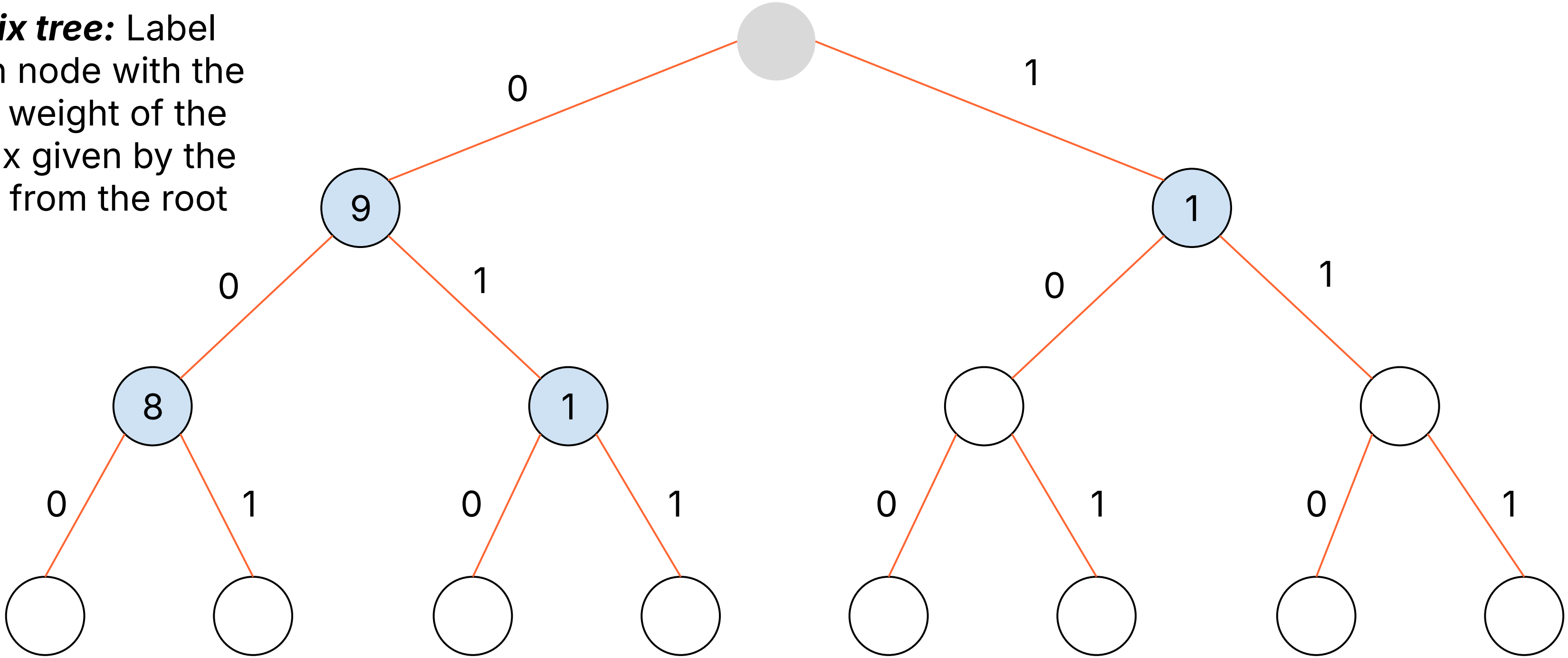
Functionality

prefix tree: Label each node with the total weight of the prefix given by the path from the root



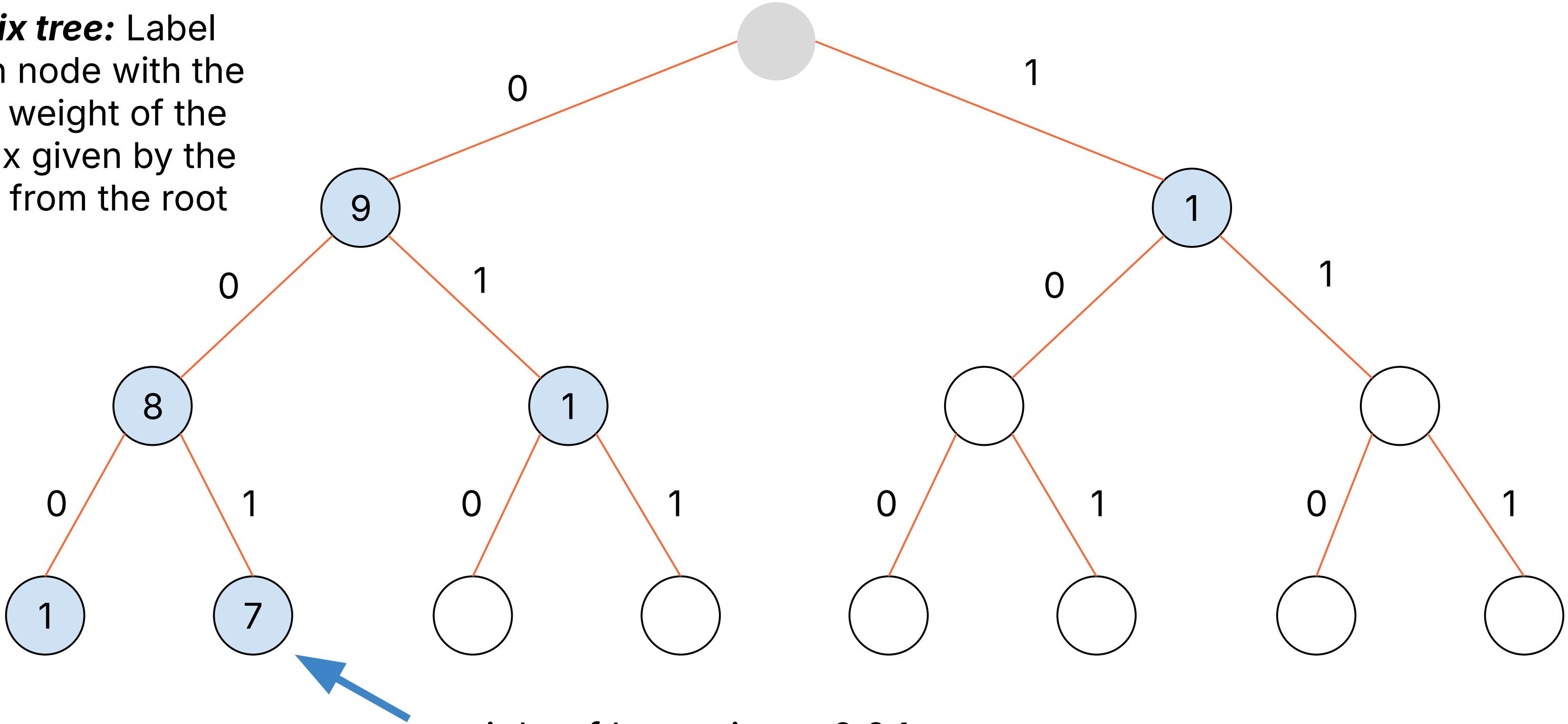
Functionality

prefix tree: Label each node with the total weight of the prefix given by the path from the root



Functionality

prefix tree: Label each node with the total weight of the prefix given by the path from the root



weight of heavy input 0,0,1

```python
def prep_init(self,
              verify_key: bytes,
              agg_id: int,
              agg_param: MasticAggParam,
              nonce: bytes,
              correcton_words: list[CorrectionWord],
              input_share: MasticInputShare,
              ) -> tuple[MasticPrepState, MasticPrepShare]:
    (level, prefixes, do_weight_check) = agg_param
    (key, proof_share) = \
        self.expand_input_share(agg_id, input_share)

    # Evaluate the VIDPF.
    (beta_share, out_share, eval_proof) = \
        self.vidpf.eval(agg_id, correction_words, key,
                        level, prefixes, nonce)

    # Query the FLP if applicable.
    verifier_share: Optional[list[F]] = None
    if do_weight_check:
        query_rand = self.query_rand(verify_key, nonce, level)
        verifier_share = \
        self.flp.query(beta_share, proof_share, query_rand)

    prep_share = (eval_proof, verifier_share)
    return (out_share, prep_share)
```