JuLenny Data Privacy

Decision Tree Inference

How a company scores another's data with a private model

The scenario

A bank has spent a decade refining a credit-risk model: a decision tree that takes an applicant's financial features and returns a risk classification. A fintech partner holds exactly the customer records that model should be scoring.

Both sides would profit from running the bank's model on the fintech's data. But the bank will not hand over the model (it is the business), and the fintech will not hand over customer records (regulation and trust forbid it). So the deal stalls.

The problem

A decision tree is built out of decisions: *if income is above this threshold, go right, otherwise go left.* Two things in that sentence are secret. The fintech's **income value** is private data. The bank's **threshold** (and which feature it even looks at) is private model IP. Normally, to take the branch, someone has to see one or the other.

How Decision Tree Inference solves this

Both the model and the data are encrypted under a **joint key split between the two parties**, and the platform evaluates the tree entirely on ciphertext. The platform never sees the data, never sees the model, and never even learns which branch was taken.

Step 1: The model owner encrypts the tree

Using the JuLenny toolkit on their own machine, the model owner encrypts every part of the tree: which feature each node tests, each threshold, and each leaf's class scores. They upload only ciphertext. Even JuLenny cannot read the model.

```
Node: test feature [encrypted], threshold [encrypted]

Leaf: class scores [encrypted]
```

Step 2: The data owner encrypts the record

The data owner encrypts the applicant's features locally and uploads only ciphertext.

```
Features (encrypted): [income, debt, age, ...]
```

Step 3: The platform takes every branch at once

Since the computation cannot look at a value to decide a branch, it does something cleverer: at each node it evaluates **both** branches and blends them, weighted by an encrypted indicator of which way the comparison went.

```
indicator = soft_compare(feature - threshold)    # ~1 if feature >
threshold, ~0 if not

result    = indicator * right_branch + (1 - indicator) * left_branch
```

The comparison is done with a polynomial that smoothly approximates a step function, so it never has to branch on a secret. The wrong paths get an encrypted weight near zero and contribute almost nothing; the right path dominates. The math flows through every path of the tree, and at no point does any machine learn which path was real.

This technique comes from peer-reviewed cryptography research, and it runs as a **fixed, auditable circuit** like every function in JuLenny's signed registry: the same operations execute on every run, so the logic cannot quietly change.

Step 4: Only the prediction comes out

The result is an encrypted vector of class scores. It is released only when the parties approve, to whoever the collaboration's visibility rules name. They decrypt it locally and read the prediction (the highest-scoring class).

```
Decrypted result: [0.17, 0.83]  ->  class 1
```

What each side learns

- The **data owner** never reveals a single record.
- The **model owner** never reveals its thresholds, its features, or its logic.
- **JuLenny** sees ciphertext throughout. Because the joint key is split between the two parties, there is nothing the platform could read even if it wanted to.

The only thing produced is the classification.

Bringing your own model

You do not retrain anything. Models come in from the tools data teams already use, scikit-learn and XGBoost among them: a converter turns a trained tree into the encrypted form the toolkit uploads.

Where this lands

- **Credit scoring**: a bank scores a partner's customers without the partner exposing records, and without the bank exposing the model.
- **Fraud detection**: a payment processor's model runs over a merchant's transactions; the merchant keeps its data, the processor keeps its rules.
- **Medical triage**: one hospital's diagnostic model classifies another hospital's patient symptoms, with no patient data leaving the second hospital's control.
- **Insurance underwriting**: an insurer scores applicant data held by a broker, without the broker seeing the model or the insurer seeing raw records.

JuLenny Data Privacy

Why this one is different

JuLenny's other launch functions are symmetric: two parties contribute the same kind of input and learn something about the overlap. Decision Tree Inference is asymmetric. One side has intelligence (a model), the other has information (data), and until now combining them meant somebody had to surrender something. A model owner can now offer scoring as a service without ever shipping the model; a data owner can buy that scoring without ever shipping the data. The math is the contract.

Technical details

- **Scheme:** CKKS (approximate real-number arithmetic, for the feature values and the soft-comparison polynomial)
- **What's encrypted:** the model (per-node feature selector + threshold, per-leaf class scores) *and* the data (feature vector) — both under the joint key. Only the tree's shape (a fixed complete tree of a given height) is public.
- **Soft comparison:** a degree-16 minimax polynomial approximating the step function (degrees 8 / 16 / 32 available — higher is sharper but deeper)
- **Multiplicative depth:** ~11-12 per inference (encrypted feature selection + soft-if + branch blend, accumulated over the tree levels); a depth-13 context handles height-4 trees with no bootstrapping
- **Tree size:** height ≤ 4 (up to 16 leaves) in the current context; deeper trees require bootstrapping
- **Relinearization keys:** Yes (ciphertext × ciphertext at every node)
- **Rotation (sum) keys:** Yes — power-of-two keys, for the encrypted feature selection
- **Engines:** CPU (OpenFHE) and GPU (FIDESlib), validated to agree to ~1e-8
- **Precision:** ~30+ bits per slot
- **Output:** an encrypted per-class score vector; the recipient decrypts locally and takes the highest-scoring class
- **Throughput:** one sample scored per execution (batched multi-sample scoring is a planned optimization)