C++ DP-GBDT Side-channel Analysis

Contents

1 Secrecy

1.1 Dataset and parameters

entity	secret
content of X	✓
X_cols_size	×
X_rows_size	×
content of y	✓
y_rows_size	×

parameter	secret
nb_trees	×
learning_rate	×
privacy_budget	×
task	×
max_depth	×
min_samples_split	×
balance_partition	×
gradient_filtering	×
leaf_clipping	×
scale_y	×
use_decay	×
l2_threshold	×
l2_lambda	×
cat_idx	×
num_idx	×

inferrable from those	secret
nb_samples per tree	×

While building a single tree

entity	secret	
X_subset	✓	
X_subset_cols_size	×	
X_subset_rows_size	✓	
y_subset	✓	
y_subset_rows_size	✓	
gradients	✓	
gradients_size	✓	

1.2 DP imperfections of the algorithm

- `init_score`, (=mean in regression, =most common feature in classification) leaks information about which feature values are in the dataset. Would need to add noise.
- `compute_gain` is done on the real data points. This was not adressed by the DPBoost paper. Would also need to add noise there.
- GDF is also problematic. changing one data point could have an impact on 2 trees.

2 main

No data dependent operations are done, therefore no side channel leakage. Securely getting the model parameters and dataset into and the resulting model out of the enclave is not among the challenges of this thesis.

Pseudocode 1: main

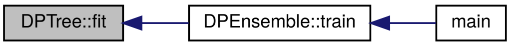
```
function main()
    // get parameters and dataset
    parameters = get_params()
    dataset = get_dataset()
    // create 5 train/test splits for cross validation
    cv_splits = create_cross_val_inputs(dataset, 5)
    // do cross validation
    for split in cv_splits do
        ensemble = DPEnsemble(parameters)
        ensemble.train(split.train)
        // predict using the test set
        y_pred = ensemble.predict(split.test.X)
        // compute score
        score = compute_score(split.test.y, y_pred)
```

3 `class DPTree`

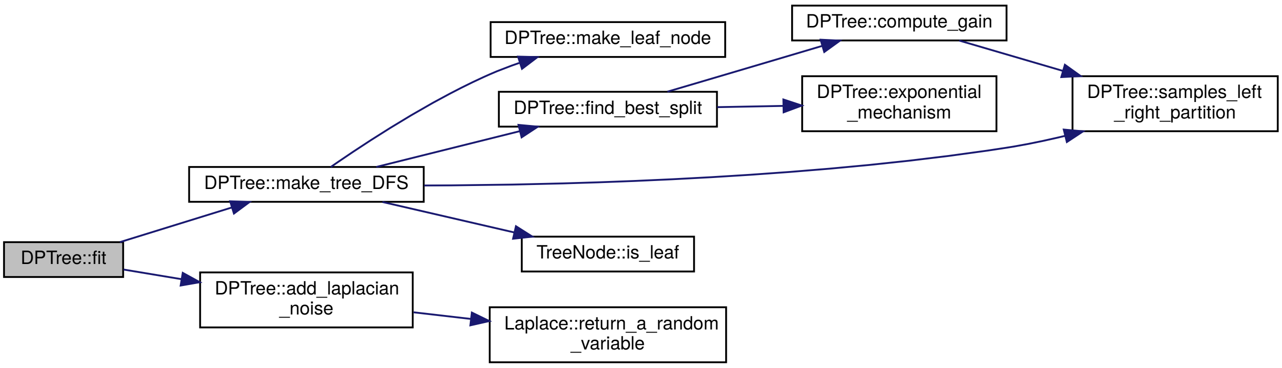
3.1 Methods

3.1.1 `fit`

Caller graph



Call graph



Variables

- must not leak:

`dataset, leaves`

- can leak:

`params.*`

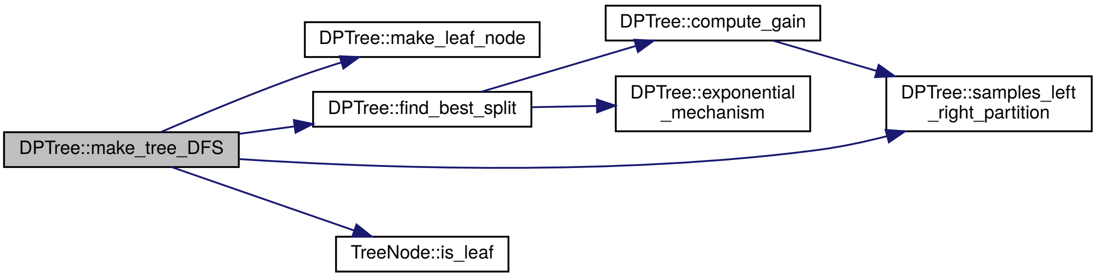
Pseudocode 2: DPTree::fit

```
function fit()
    // all samples are live at the start
    live_samples = [1,2,3,...,dataset.length]
    // build tree
    this→root_node = make_tree_dfs(live_samples, 0)
    // leaf_clipping
    if params.leaf_clipping or !params.gradient_filtering then        ▷ params
        threshold = params.l2 * (1 − η)^tree_index
        for leaf in this→leaves do        ▷ number of leaves, which nodes are leaves
            leaf.prediction = clamp(leaf.prediction, -threshold, threshold)
    // add laplace noise to leaf values
    privacy_budget_for_leaf_nodes = tree_privacy_budget / 2
    laplace_scale = params.Δv / privacy_budget_for_leaf_nodes
    add_laplacian_noise(laplace_scale)
```

3.1.2 `make_tree_dfs`

Caller graph

Call graph



Arguments / variables

- must not leak:

`dataset, X_transposed, live_samples, gradients`

- can leak:

`params.min_samples_split, params.max_depth, curr_depth`

Pseudocode 3: DPTree::make_tree_dfs

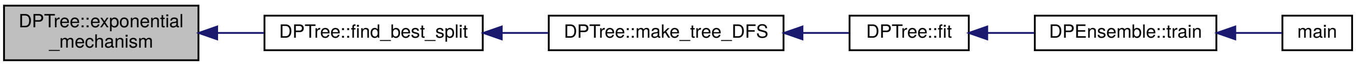
```
function make_tree_dfs(live_samples, curr_depth)                         ▷ size of live_samples
    // max depth reached or not enough samples -> leaf node
    if curr_depth == params.max_depth or len(live_samples) < params.min_samples_split then
        TreeNode *leaf = make_leaf_node(curr_depth, live_samples)        ▷ both branch conditions
        return leaf
    // find best split
    TreeNode *node = find_best_split(X, gradients, live_samples, curr_depth)
    // no split found
    if node.is_leaf then                                                 ▷ number of leaves
        return node
    // prepare the new live samples to continue recursion
    lhs,rhs = samples_left_right_partition(X, node.feature_index, node.feature_value)    ▷ sizes
    for sample in live_samples do                                        ▷ size of live_samples
        if lhs.contains(sample) then                                     ▷ which samples go left/right
            lhs_live_samples.insert(sample)
        else
            rhs_live_samples.insert(sample)
    // recurse
    node→left = make_tree_dfs(lhs_live_samples, curr_depth+1)
    node→right = make_tree_dfs(rhs_live_samples, curr_depth+1)
    return node
```

Recursion leakage

- number of splits in the tree
- number of splits/leaves observable by watching memory allocations

3.1.3 exponential_mechanism

Caller graph



Arguments / variables

- must not leak:

candidates

- can leak:

-

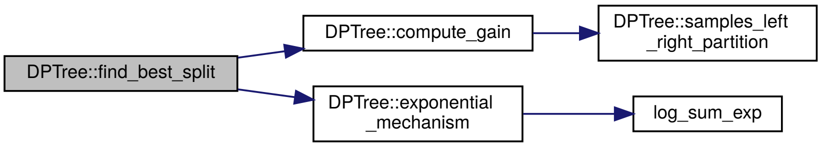
Pseudocode 4: DPTree::exponential_mechanism

```
function exponential_mechanism(candidates)                      ▷ number of candidates
    // if no split with positive gain, return, node will become a leaf
    if cand.gain <= 0 forall cand in candidates then            ▷ no good split exists, leaf creation
        return-1
    // calculate probabilities from the gains
    for candidate in candidates do                              ▷ number of candidates
        gains.append(candidate.gain)
        if candidate.gain <= 0 then                             ▷ number of candidates with viable splits
            probabilities.append(0)
        else
            lse = log ∑_i exp(gains_i)
            probabilities.append(exp(candidate.gain − lse))
    // create a cumulative distribution from the probabilities, its values add up to 1
    partials = std::partial_sum(probabilities)
    // choose random value in [0,1]
    rand_val = std::rand()
    // return the corresponding split
    for i = 0 to i = partials.size() - 1 do
        if partials[i] >= rand_val then
            return i
    return -1
```

3.1.4 `find_best_split`

Caller graph

Call graph



Arguments / variables

- must not leak:

`X, gradients, live_samples`

- can leak:

`params.*, tree_budget, curr_depth`

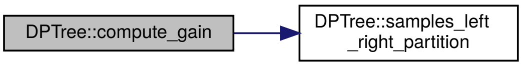
Pseudocode 5: DPTree::find_best_split

```
function find_best_split(X, gradients, live_samples, curr_depth)    ▷ size of X, gradients
    // determine node privacy budget
    if params.use_decay then                                         ▷ params.use_decay
        if curr_depth == 0 then                                      ▷ curr_depth == 0
            node_budget = tree_budget / (2*(2^(max_depth+1) + 2^(curr_depth+1)))
        else
            node_budget = tree_budget / (2*2^(curr_depth+1))
    else
        node_budget = tree_budget / (2*max_depth)
    // iterate over all possible splits
    for feature_index in features do                                 ▷ number of cols in X
        for feature_value in X[feature_index] do                     ▷ number of rows in X
            if "already encountered feature_value" then              ▷ number of unique feature values
                continue
            gain = compute_gain(X, gradients, live_samples, feature_index, feature_value)
            if gain < 0 then                                         ▷ number of splits with no gain
                continue
            gain = (node_budget*gain) / (2*Δg)
            candidates.insert(Candidate(feature_index, feature_value, gain))   ▷ number of candidates
    // choose a split using the exponential mechanism
    index = exponential_mechanism(candidates)
    // construct the node
    TreeNode *node = new TreeNode(candidates[index])                 ▷ internal node vs. leaf
    return node
```

3.1.5 compute_gain

Caller graph

Call graph



Arguments / variables

- must not leak:

`X, gradients, live_samples`

- can leak:

`params.l2_lambda, feature_index, feature_value`

Pseudocode 6: DPTree::compute_gain

```
function compute_gain(X, gradients, live_samples, feature_index, feature_value)      ▷ X, gradients
    // partition into lhs/rhs
    lhs,rhs = samples_l_r_partition(X, live_samples, feature_index, feature_value)    ▷ lhs/rhs size
    lhs_size = lhs.size()
    rhs_size = rhs.size()
    // return on useless split
    if lhs_size == 0 or rhs_size == 0 then                                            ▷ useless split
        return -1
    // sums of lhs/rhs gains
    lhs_gain = sum(gradients[lhs])                        ▷ memory access pattern of left/right gradients
    rhs_gain = sum(gradients[rhs])
    lhs_gain = lhs_gain^2 / (lhs_size+params.l2_lambda)
    rhs_gain = rhs_gain^2 / (rhs_size+params.l2_lambda)
    total_gain = max(lhs_gain + rhs_gain, 0)                  ▷ max might leak whether total_gain < 0
    return total_gain
```

3.1.6 make_leaf_node

3.1.7 samples_left_right_partition

3.1.8 predict

4 class DPEnsemble

4.1 Methods

4.1.1 train

Caller graph

TODO

Call graph

TODO

Variables

- must not leak:

TODO

- can leak:

params.*, tree_params.*

Pseudocode 7: DPEnsemble::train

```
function train(dataset)
    // compute initial prediction
    init_score = compute_init_score(dataset.y)
    // each tree gets the full budget since they train on distinct data
    tree_privacy_budget = params.privacy_budget
    // train all trees
    for tree_index = 0 to tree_index = nb_trees - 1 do                ▷ bla
        // init/update gradients
        update_gradients(dataset.gradients, tree_index)
        // sensitivity for internal nodes
        tree_params.Δg = 3 * (params.l2_threshold)^2
        // sensitivity for leaf nodes
        if params.gradient_filtering or !params.leaf_clipping then
            tree_params.Δv = params.l2_threshold / (1 + params.l2_lambda)
        else
            tree_params.Δv = min(params.l2_threshold / (1 + params.l2_lambda), 2 * params.l2_threshold * (1 - η)^tree_index)
        // determine number of rows
        if params.balance_partition then
            number_of_rows = |D| / (nb_trees - tree_index)
        else
            number_of_rows = |D| η (1 - η)^tree_index / (1 - (1 - η)^nb_trees)
```

4.1.2 `predict`

4.1.3 `update_gradients`

4.1.4 `add_laplacian_noise`

4.1.5 `remove_rows`

4.1.6 `get_subset`

5 other classes