C++ DP-GBDT Side-channel Analysis

Contents

1 Severity List

General

entity	secrecy
X	✓
X_cols_size	×
X_rows_size	×
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	secrecy	
X_subset	✓	
X_subset_cols_size	×	
X_subset_rows_size	✓	
y_subset	✓	
y_subset_rows_size	✓	
gradients	✓	
gradients_size	✓	

2 `main`

3 `class DPTree`

3.1 Creation / destruction / global variables

Side channel leakage

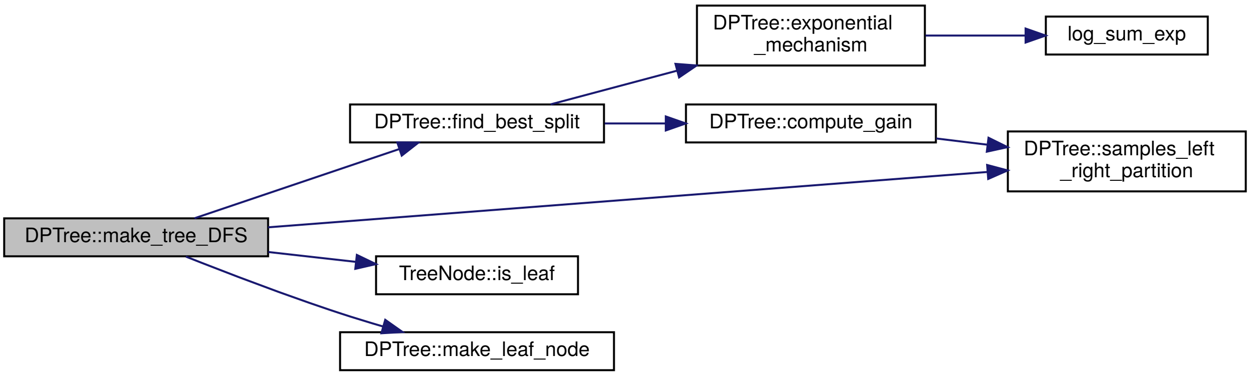
3.2 Methods

3.2.1 `fit`

3.2.2 make_tree_DFS

Caller graph

Call graph



Arguments / used variables

variable	secret
dataset	✓
gradients	✓
curr_depth	?
live_samples	✓

params.min_samples_split	×
params.max_depth	×

Algorithm 1: make_tree_DFS

```
Function make_tree_DFS(live_samples[], curr_depth)
    // max depth reached or not enough samples -> leaf node
    if curr_depth == params.max_depth || len(live_samples) < params.min_samples_split
     then
        TreeNode *leaf = make_leaf_node(curr_depth, live_samples)
        return leaf
    // get actual sample rows and respective gradients from indices in live_samples
    X_live = dataset→X[live_samples]
    gradients_live = dataset→gradients[live_samples]
    // find best split
    TreeNode *node = find_best_split(X_live, gradients_live, curr_depth)
    // no split found
    if node.is_leaf then
        return node
    // prepare the new live samples to continue recursion
    lhs = samples_left_right_partition(X_live, node.feature_index, node.feature_value)
    for sample : live_samples do
        if lhs.contains(sample) then
            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
```

Side channel leakage

- leakage in called methods

From branches/loops:

- params.max_depth
- params.min_samples_split
- curr_depth == max_depth
- number of features (columns of X)

- number of live samples resp. rows in X_live
- whether node becomes a leaf / #leaves
- whether split is done on a categorical feature
- split details (how many samples go left/right)

Recursion leakage: ?

- e.g. #splits in the tree by measuring time
- #nodes observable by watching memory allocations for node

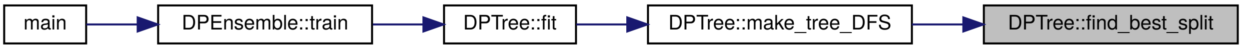
Mitigations TODO

3.2.3 `exponential_mechanism`

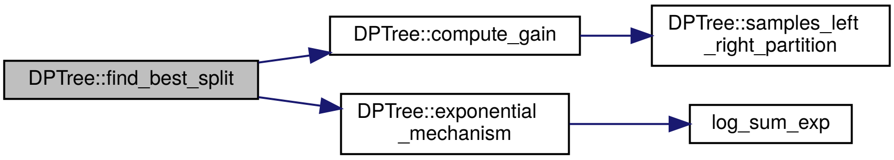
TODO

3.2.4 `find_best_split`

Caller graph



Call graph



Arguments / used variables

variable	secret
X	✓
gradients	✓
curr_depth	?
tree_budget	×
params.use_decay	×
params.Δg	×
params.max_depth	×

Algorithm 2: find_best_split

```
Function find_best_split(X[][], gradients[], curr_depth)
    // determine node privacy budget
    if params.use_decay then
        if curr_depth == 0 then
            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 : features do
        for feature_value : X[feature_index] do
            if "already encountered feature_value" then
                continue
            gain = compute_gain(X, gradients, feature_index, feature_value)
            if gain < 0 then
                continue
            gain = (node_budget*gain) / (2*Δg)
            candidates.insert(Candidate(feature_index, feature_value, gain))
    // choose a split using the exponential mechanism
    index = exponential_mechanism(candidates)
    // construct the node
    TreeNode *node = new TreeNode(candidates[index])
    return node
```

Side channel leakage

- leakage in `compute_gain` and `exponential_mechanism`

From branches/loops:

- params.use_decay
- curr_depth == 0
- number of features (columns of X)
- number of rows in X resp. length of gradients
- number of unique feature values of a feature
- number of splits that don't give any gain
- number of split candidates

Potential arithmetic leakage: ?

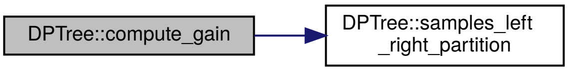
- Not sure about this in SGX though
- edge cases of variables appearing in formulas → `tree_budget` and `curr_depth` and Δg and `gain`

Mitigations TODO

3.2.5 compute_gain

Caller graph

Call graph



Arguments / used variables

variable	secret	
X	✓	
gradients	✓	
feature_index	×	
feature_value	×	
params.l2_lambda	×	

Algorithm 3: compute_gain

```
Function compute_gain(X[][], gradients[], feature_index, feature_value)
    // // partition into lhs/rhs
    lhs, rhs = samples_left_right_partition(X, feature_index, feature_value)
    lhs_size = lhs.size()
    rhs_size = rhs.size()
    // return on useless split
    if lhs_size == 0 || rhs_size == 0 then
        return -1
    // sums of lhs/rhs gains
    lhs_gain = sum(gradients[lhs])
    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 = lhs_gain + rhs_gain
    total_gain = max(total_gain, 0)
    return total_gain
```

Side channel leakage

- leakage in `samples_left_right_partition`

From branches/loops/function calls:

- size (#rows) of X/gradients
- lhs/rhs size
- whether it's a useless split
- memory access pattern of left/right gradients
- max function might leak whether `total_gain` < 0

Potential arithmetic leakage: ?

- edge cases of variables appearing in formulas → `lhs_gain` and `lhs_size`, rhs respectively.

Mitigations TODO

3.2.6 make_leaf_node

3.2.7 predict

3.2.8 samples_left_right_partition

3.2.9 predict

4 class DPEnsemble

4.1 Creation / destruction / global variables

Side channel leakage

4.2 Methods

4.2.1 train

4.2.2 predict

4.2.3 update_gradients

4.2.4 add_laplacian_noise

4.2.5 remove_rows

4.2.6 get_subset

5 other classes