BTP Report

Project Title: Hardware Security using AI/ML Techniques

Project Mentor: Dr Binod Kumar

By: Praneet Thakur (B19CSE066)

Introduction:

Globalization of hardware design and fabrication processes have raised serious concerns about hardware-based attacks. Hardware has been always assumed to be the guarantee of trustworthiness in cryptographic algorithms and security protocols. However, several backdoors have been reported in the last years, especially in military contexts.

Hardware Trojans (HTs) are malicious changes made to integrated circuits in order to disrupt their functional behaviour. They are made up of two major components: the trigger, which activates the malicious behaviour under certain conditions, and the payload, which performs the malicious tasks.

The triggers can include
(i) functional based conditions, e.g., a specific value or a sequence of values, which activates the payload once
it has been observed on a certain register or port,
(ii) physical-based conditions, e.g., reaching a
value of temperature or power,
(iii) time-based conditions, e.g., a certain number of cycles or
operations that must be counted.
Payloads typically exhibit even more diversity, e.g., leakage of

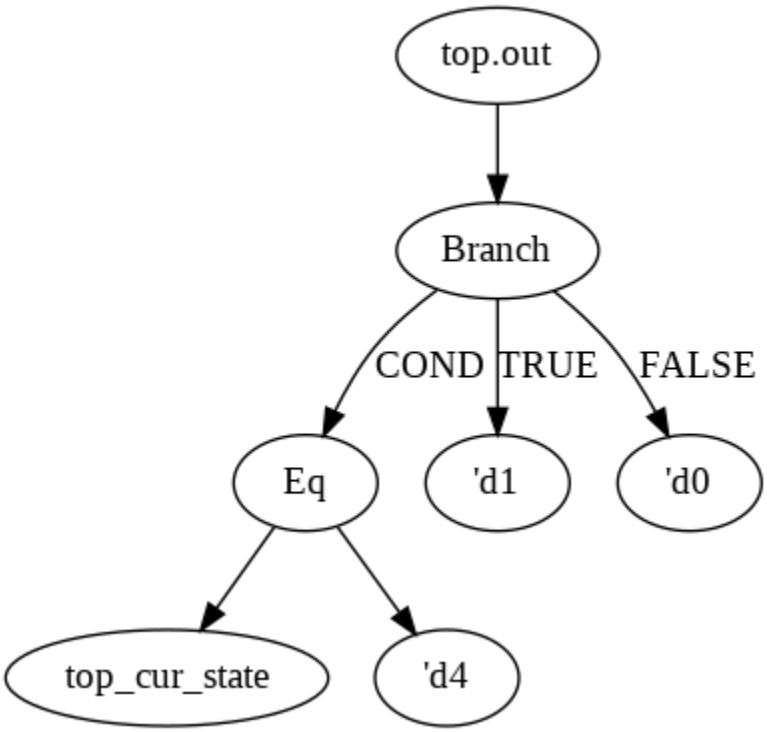
information, data corruptions, performance loss, etc

HTs can be added during every phase of the fabrication process, e.g., design or synthesis, and they are designed to remain silent during the whole verification and testing phase, thus causing the failure of the standard verification approaches. HTs are more and more inserted at RTL because, at this level of abstraction, attackers have high flexibility to implement any malicious function. In this project we have focused on detection of HTs inserted in the RTL phase.

Introduction:

Model the circuit using graph neural networks(GNN) to detect hardware Trojan.
Verilog Code⇒ Data Flow Graph(DFG) and Abstract Systax Tree(AST) => GNN =>Hardware Trojan detection
DFG example: Code and DFG

```
module top ( input clk,
             input rstn,
             input in,
             output out );

  parameter IDLE    = 0,
            S1      = 1,
            S10     = 2,
            S101    = 3,
            S1011   = 4;

  reg [2:0] cur_state, next_state;

  assign out = cur_state == S1011 ? 1 : 0;

  always @ (posedge clk) begin
    if (!rstn)
       cur_state <= IDLE;
    else
       cur_state <= next_state;
  end

  always @ (cur_state or in) begin
    case (cur_state)
      IDLE : begin
        if (in) next_state = S1;
        else next_state = IDLE;
      end

      S1: begin
        if (in) next_state = IDLE;
        else    next_state = S10;
      end

      S10 : begin
        if (in) next_state = S101;
        else    next_state = IDLE;
      end

      S101 : begin
        if (in) next_state = S1011;
        else    next_state = IDLE;
      end

      S1011: begin
        next_state = IDLE;
      end
    endcase
```



Steps:

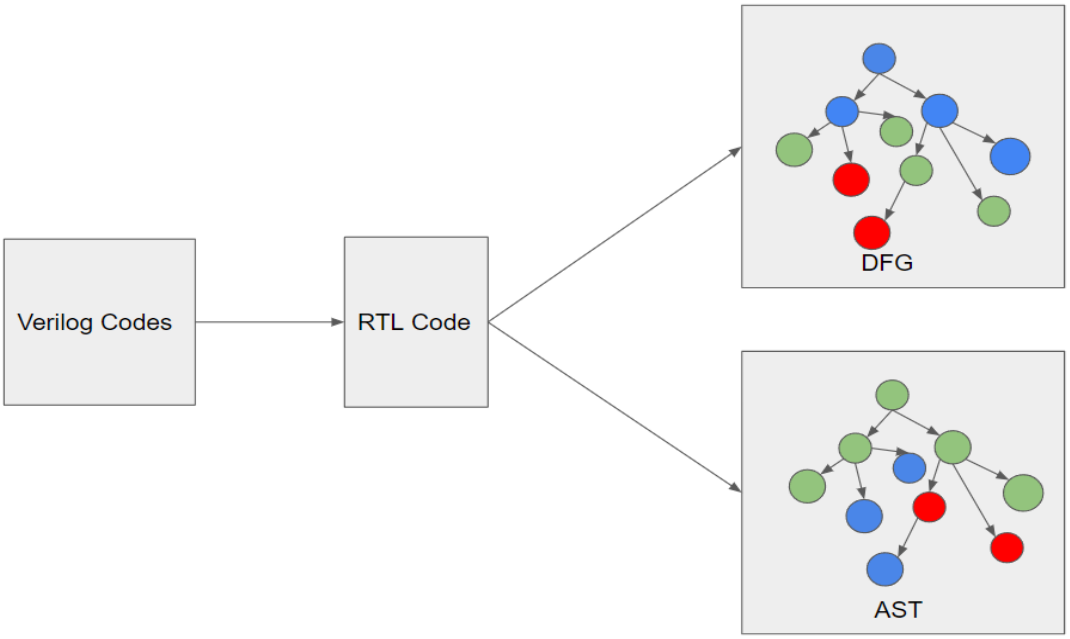
i) Extracting DFG and AST from the hardware designs.

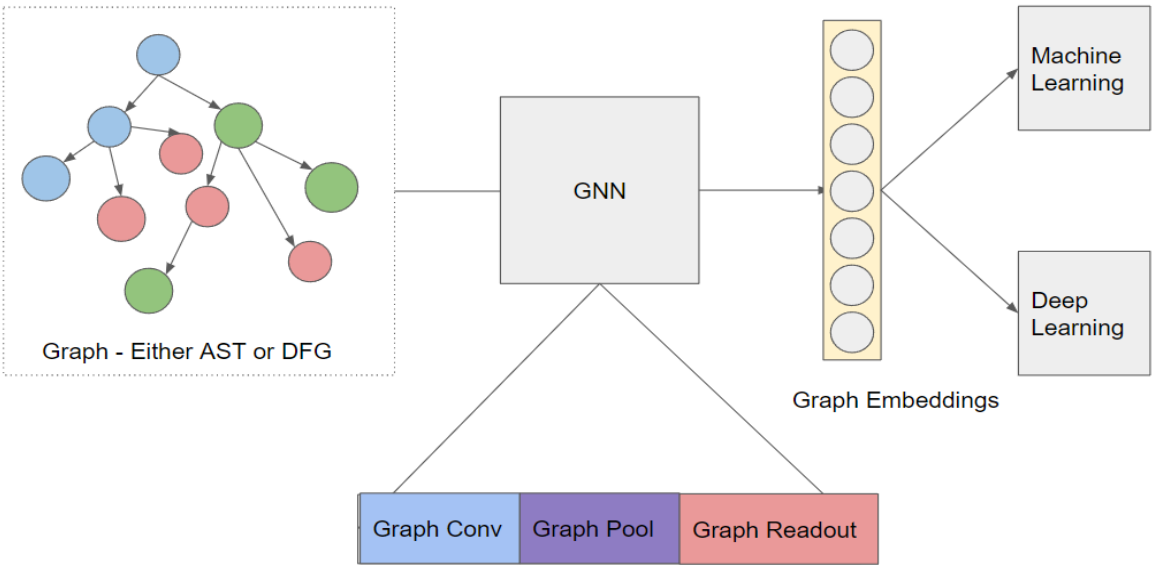
ii) The DFGs and ASTs passed to GNN to generate graph embeddings.

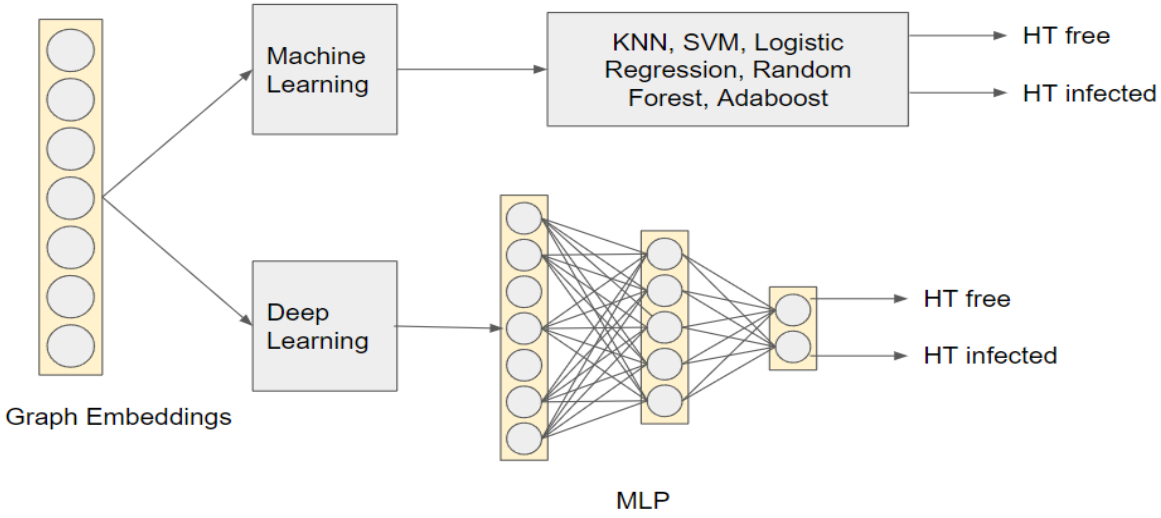
iii) Machine learning models on the generated graph embeddings to detect wheter trojan infected or not.(AST, DFG)

iv) MLP trained on the embeddings to detect whether trojan infected or not.(DFG, AST, AST+DFG)

Workflow Diagram:







Difficulties faced:

There were some difficulties faced. Tried changing environments multiple times but nothing worked. So, I had to patch some files in pytorch and pytorch-geomertic in order to get embeddings.

The patched portions are:

i)usr/local/lib/torch-geometric/nn/Dense/linear.py

```
    def _lazy_load_hook(self, state_dict, prefix, local_metadata, strict,
                        missing_keys, unexpected_keys, error_msgs):
        temp=state_dict
        d2 = OrderedDict([('layers.0.graph_conv.lin.weight', v) if k == 'layers.0.graph_conv.weight' else (k, v) for k, v in state_dict.items()])
#         print(d2)
        d3 = OrderedDict([('layers.0.graph_conv.lin.bias', v) if k == 'layers.0.graph_conv.bias' else (k, v) for k, v in d2.items()])
#         print(d3)
        d4 = OrderedDict([('layers.1.graph_conv.lin.weight', v) if k == 'layers.1.graph_conv.weight' else (k, v) for k, v in d3.items()])
        #print(d4)
        d5 = OrderedDict([('layers.1.graph_conv.lin.bias', v) if k == 'layers.1.graph_conv.bias' else (k, v) for k, v in d4.items()])
#         print(d5)
        d6 = OrderedDict([('pool1.graph_pool.gnn.lin_rel.weight', v) if k == 'pool1.graph_pool.gnn.lin_l.weight' else (k, v) for k, v in d5.items()])
#         print(d6)
        d7 = OrderedDict([('pool1.graph_pool.gnn.lin_rel.bias', v) if k == 'pool1.graph_pool.gnn.lin_l.bias' else (k, v) for k, v in d6.items()])
        #print(d7)
        d8 = OrderedDict([('pool1.graph_pool.gnn.lin_root.weight', v) if k == 'pool1.graph_pool.gnn.lin_r.weight' else (k, v) for k, v in d7.items()])
#         print(d8)
        state_dict=d8
```

ii)usr/local/lib/torch/nn/Module/module.py

```
def load_state_dict(self, state_dict: 'OrderedDict[str, Tensor]',
                        strict: bool = True):
    r"""Copies parameters and buffers from :attr:`state_dict` into
    this module and its descendants. If :attr:`strict` is ``True``, then
    the keys of :attr:`state_dict` must exactly match the keys returned
    by this module's :meth:`~torch.nn.Module.state_dict` function.

    Args:
        state_dict (dict): a dict containing parameters and
            persistent buffers.
        strict (bool, optional): whether to strictly enforce that the keys
            in :attr:`state_dict` match the keys returned by this module's
            :meth:`~torch.nn.Module.state_dict` function. Default: ``True``

    Returns:
        ``NamedTuple`` with ``missing_keys`` and ``unexpected_keys`` fields:
            * **missing_keys** is a list of str containing the missing keys
            * **unexpected_keys** is a list of str containing the unexpected keys

    Note:
        If a parameter or buffer is registered as ``None`` and its corresponding key
        exists in :attr:`state_dict`, :meth:`load_state_dict` will raise a
        ``RuntimeError``.
    """
    missing_keys: List[str] = []
    unexpected_keys: List[str] = []
    error_msgs: List[str] = []

    # copy state_dict so _load_from_state_dict can modify it
    metadata = getattr(state_dict, '_metadata', None)
    state_dict = state_dict.copy()
    temp=state_dict
    d2 = OrderedDict([('layers.0.graph_conv.lin.weight', v) if k == 'layers.0.graph_conv.weight' else (k, v) for k, v in state_dict.items()])
#     print(d2)
#     d3 = OrderedDict([('layers.0.graph_conv.lin.bias', v) if k == 'layers.0.graph_conv.bias' else (k, v) for k, v in d2.items()])
#     print(d3)
    d4 = OrderedDict([('layers.1.graph_conv.lin.weight', v) if k == 'layers.1.graph_conv.weight' else (k, v) for k, v in d2.items()])
#     print(d4)
#     d5 = OrderedDict([('layers.1.graph_conv.lin.bias', v) if k == 'layers.1.graph_conv.bias' else (k, v) for k, v in d4.items()])
#     print(d5)
    d6 = OrderedDict([('pool1.graph_pool.gnn.lin_rel.weight', v) if k == 'pool1.graph_pool.gnn.lin_l.weight' else (k, v) for k, v in d4.items()])
#     print(d6)
    d7 = OrderedDict([('pool1.graph_pool.gnn.lin_rel.bias', v) if k == 'pool1.graph_pool.gnn.lin_l.bias' else (k, v) for k, v in d6.items()])
    #print(d7)
    d8 = OrderedDict([('pool1.graph_pool.gnn.lin_root.weight', v) if k == 'pool1.graph_pool.gnn.lin_r.weight' else (k, v) for k, v in d7.items()])
#     print(d8)
    state_dict=d8
```

```
def _load_from_state_dict(self, state_dict, prefix, local_metadata, strict,
                          missing_keys, unexpected_keys, error_msgs):
    r"""Copies parameters and buffers from :attr:`state_dict` into only
    this module, but not its descendants. This is called on every submodule
    in :meth:`~torch.nn.Module.load_state_dict`. Metadata saved for this
    module in input :attr:`state_dict` is provided as :attr:`local_metadata`.
    For state dicts without metadata, :attr:`local_metadata` is empty.
    Subclasses can achieve class-specific backward compatible loading using
    the version number at `local_metadata.get("version", None)`.

    .. note::
        :attr:`state_dict` is not the same object as the input
        :attr:`state_dict` to :meth:`~torch.nn.Module.load_state_dict`. So
        it can be modified.

    Args:
        state_dict (dict): a dict containing parameters and
            persistent buffers.
        prefix (str): the prefix for parameters and buffers used in this
            module
        local_metadata (dict): a dict containing the metadata for this module.
            See
        strict (bool): whether to strictly enforce that the keys in
            :attr:`state_dict` with :attr:`prefix` match the names of
            parameters and buffers in this module
        missing_keys (list of str): if ``strict=True``, add missing keys to
            this list
        unexpected_keys (list of str): if ``strict=True``, add unexpected
            keys to this list
        error_msgs (list of str): error messages should be added to this
            list, and will be reported together in
            :meth:`~torch.nn.Module.load_state_dict`
    """


    for hook in self._load_state_dict_pre_hooks.values():
        hook(state_dict, prefix, local_metadata, strict, missing_keys, unexpected_keys, error_msgs)

    persistent_buffers = {k: v for k, v in self._buffers.items() if k not in self._non_persistent_buffers_set}
    local_name_params = itertools.chain(self._parameters.items(), persistent_buffers.items())
    local_state = {k: v for k, v in local_name_params if v is not None}

    for name, param in local_state.items():
        key = prefix + name
        if key in state_dict:
            input_param = state_dict[key]
            print(key)
            print(input_param.shape)
            if(not key=="pool1.graph_pool.gnn.lin_rel.weight" and not key=="pool1.graph_pool.gnn.lin_root.weight" and not key=="fc.weight" and len(input_param.shape)>1):
              print("Entered")
              input_param=torch.t(input_param)
            print(input_param.shape)
            if not torch.overrides.is_tensor_like(input_param):
                error_msgs.append('While copying the parameter named "{}", '
```

Example Embedding:

Hardware: det_1011

```
module top ( input clk,
             input rstn,
             input in,
             output out );

  parameter IDLE    = 0,
            S1      = 1,
            S10     = 2,
            S101    = 3,
            S1011   = 4;

  reg [2:0] cur_state, next_state;

  assign out = cur_state == S1011 ? 1 : 0;

  always @ (posedge clk) begin
    if (!rstn)
      cur_state <= IDLE;
    else
      cur_state <= next_state;
  end

  always @ (cur_state or in) begin
    case (cur_state)
      IDLE : begin
        if (in) next_state = S1;
        else next_state = IDLE;
      end

      S1: begin
        if (in) next_state = IDLE;
        else    next_state = S10;
      end

      S10 : begin
        if (in) next_state = S101;
        else    next_state = IDLE;
      end

      S101 : begin
        if (in) next_state = S1011;
        else    next_state = IDLE;
      end

      S1011: begin
        next_state = IDLE;
      end
    endcase
  end
endmodule
```

```
tensor([[1.3948e-02, 1.2979e-01, 0.0000e+00, 2.4929e-02, 2.9985e-03, 4.1728e-02,
         2.9751e-02, 1.6098e-01, 4.9910e-02, 6.4285e-02, 1.8172e-02, 1.2490e-01,
         6.1642e-02, 2.5481e-01, 1.3095e-02, 2.7890e-01, 9.4345e-03, 1.9104e-02,
         5.7868e-02, 2.9018e-02, 1.4692e-01, 6.6350e-03, 8.3674e-02, 1.2552e-01,
         1.3926e-01, 0.0000e+00, 6.1905e-02, 1.8589e-01, 1.0695e-02, 8.7878e-02,
         4.8211e-03, 9.0990e-02, 9.4179e-02, 0.0000e+00, 0.0000e+00, 8.1620e-02,
         7.3958e-02, 1.3948e-01, 1.0907e-02, 0.0000e+00, 6.5199e-02, 2.9032e-02,
         5.6405e-03, 5.4433e-02, 4.7946e-02, 9.2855e-02, 3.8569e-02, 2.6426e-02,
         2.4809e-02, 2.2096e-02, 5.5625e-02, 3.2019e-02, 3.3750e-02, 0.0000e+00,
         1.2897e-03, 4.8722e-02, 4.0722e-02, 0.0000e+00, 2.9506e-02, 0.0000e+00,
         1.1650e-02, 6.8441e-02, 1.2338e-01, 4.6509e-03, 4.5510e-02, 7.8622e-02,
         2.4183e-02, 0.0000e+00, 9.9947e-03, 6.6222e-02, 4.9974e-02, 8.7760e-02,
         1.0573e-02, 0.0000e+00, 0.0000e+00, 0.0000e+00, 9.1501e-02, 4.3337e-02,
         9.6938e-02, 0.0000e+00, 0.0000e+00, 0.0000e+00, 4.2152e-02, 5.8777e-03,
         1.3552e-02, 1.5872e-01, 0.0000e+00, 0.0000e+00, 1.9055e-01, 0.0000e+00,
         5.6886e-02, 7.3342e-02, 0.0000e+00, 1.3330e-01, 5.8647e-02, 5.6083e-02,
         3.5284e-02, 0.0000e+00, 0.0000e+00, 0.0000e+00, 1.1944e-02, 0.0000e+00,
         1.7427e-02, 3.8086e-03, 7.6280e-02, 3.7846e-03, 3.4469e-02, 3.9759e-02,
         0.0000e+00, 2.9256e-02, 0.0000e+00, 0.0000e+00, 0.0000e+00, 1.4103e-02,
         6.1886e-02, 1.0444e-01, 7.8638e-02, 4.4072e-02, 2.1238e-01, 2.9323e-02,
         4.9623e-03, 7.0244e-03, 5.0709e-02, 0.0000e+00, 0.0000e+00, 7.9893e-02,
         1.0898e-01, 2.0652e-02, 2.0869e-01, 8.5429e-02, 5.8493e-02, 0.0000e+00,
         6.4630e-02, 6.4834e-02, 2.2735e-01, 1.1476e-01, 7.4748e-03, 9.0293e-03,
         0.0000e+00, 1.3093e-02, 8.2761e-02, 2.9584e-02, 5.2660e-02, 2.2517e-02,
         1.4587e-01, 2.5117e-04, 1.2272e-01, 0.0000e+00, 3.8560e-02, 7.5219e-02,
         3.4223e-02, 9.7445e-02, 3.3781e-02, 1.3148e-01, 0.0000e+00, 2.4623e-01,
         3.8362e-02, 4.0974e-02, 4.8491e-02, 1.6494e-02, 4.6291e-02, 3.5212e-02,
         1.5864e-02, 3.2648e-02, 2.3950e-02, 1.0302e-01, 3.4302e-01, 0.0000e+00,
         2.5280e-02, 1.1299e-01, 0.0000e+00, 1.2514e-01, 3.2592e-02, 1.3427e-01,
         6.6011e-02, 4.2720e-02, 1.0140e-02, 1.5393e-02, 1.0258e-01, 3.8331e-02,
         4.3766e-02, 1.7193e-01, 1.4044e-01, 0.0000e+00, 0.0000e+00, 3.2624e-02,
         1.6002e-02, 2.2024e-04, 1.5550e-01, 1.0395e-01, 2.7653e-02, 0.0000e+00,
         0.0000e+00, 1.5260e-01, 0.0000e+00, 6.2729e-05, 1.1490e-01, 0.0000e+00,
         6.8117e-02, 6.2184e-02]], grad_fn=<CppNode<ScatterMax>>)
```

Graph saint:

- GNN requires GraphSAINT to perform node classification . I have used the TensorFlow implementation of GraphSAINT.
- Since the size of dataset is small I have used the graph saint subgraph matching technique to increase the training for the model.
- In GraphSAINT, subgraphs are sampled from the original graph, and a full GNN is constructed for each subgraph.

Results:

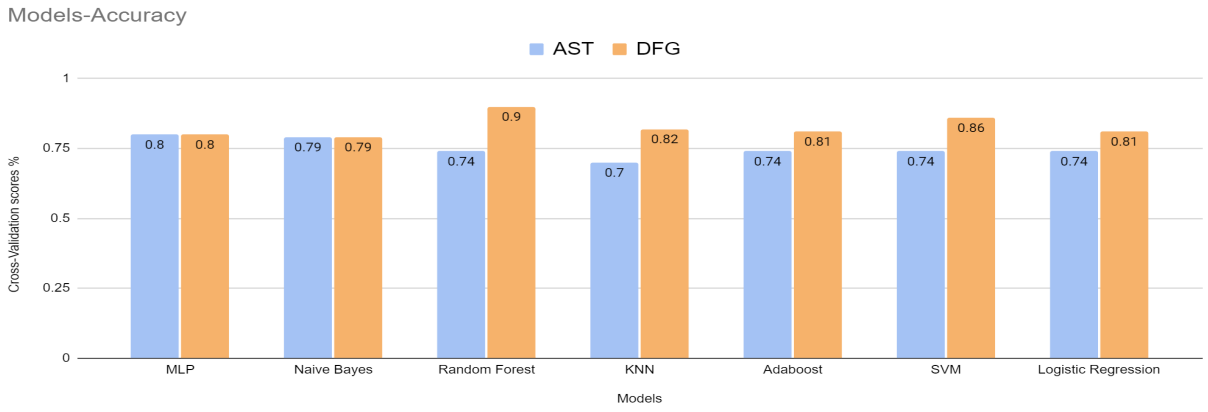
I have tried to optimize the gnn model by trying out different set of hyperparameters. Two such set of parameters are shown ahead with different sets of results. The metrics for evaluating the model are accuracy, precision score, recall and f1 score.

GNN configuration:

Architecture 1	Training
Total number of layers - 2 Pooling type: “topk” graph pooling Readout_type - “max” Activation - Relu	Optimiser - Adam Dropout - 0.5 Learning rate - 0.001 Batch_size - 4 #number of graphs in a batch Epochs - 200

Results for GNN1:

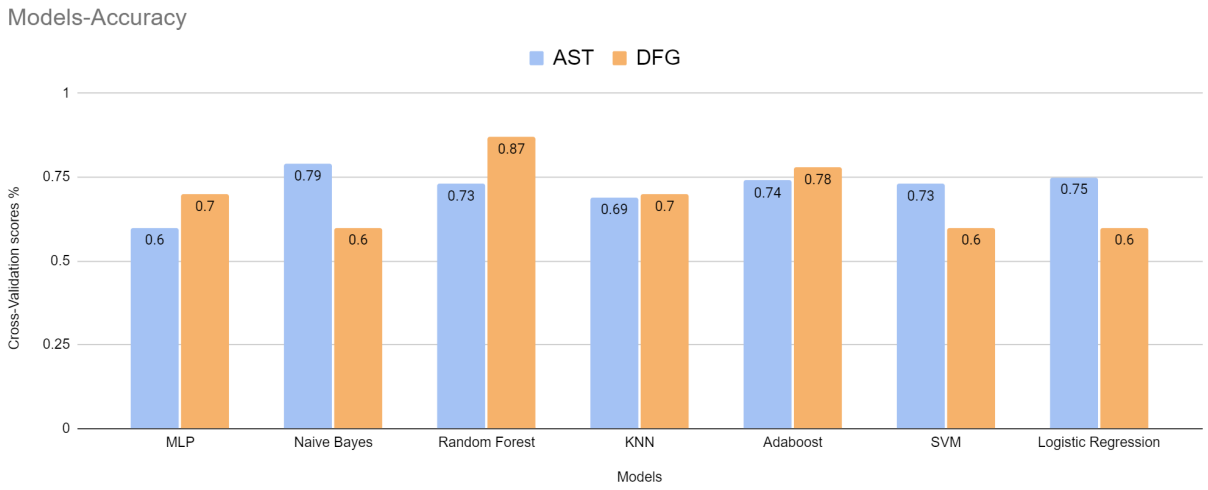
Model	AST Accuracy ,Precision, Recall, F1score	DFG Accuracy, Precision, Recall, F1score
MLP	accuracy:0.80, precision: 0.7500, recall: 1.0000, f1 score: 0.8000	0.80 precision: 0.7500, recall: 1.0000, f1 score: 0.8000
KNN	0.70, 0.6, 1.0, 0.749	0.82, 0.6, 1.0, 0.749
SVM	0.74, 0.75, 1.0, 0.857	0.86, 0.75, 1.0, 0.857
Random Forest	0.74, 0.6, 1.0, 0.749	0.90, 0.75, 1.0, 0.857
Adaboost	0.74, 0.75, 1.0, 0.857	0.81, 0.75, 1.0, 0.857
Logistic Regression	0.74, 1.0, 1.0, 1.0	0.81, 1.0, 1.0, 1.0
Naive Bayes	0.79, 1.0, 1.0, 1.0	0.79, 0.75, 1.0, 0.857



Architecture 2	Training
Total number of layers - 3 Pooling type: “topk” graph pooling Readout_type - “max” Activation - Relu	Optimiser - Adam Dropout - 0.4 Learning rate - 0.001 Batch_size - 4 Epochs - 200

Results for GNN2-

Model	AST Accuracy ,Precision, Recall, F1score	DFG Accuracy, Precision, Recall, F1score
MLP	Accuracy: 0.60, precision: 0.6000,recall: 1.0000,f1 score: 0.7500	0.70 precision: 0.6000,recall: 1.0000, f1 score: 0.7500
KNN	0.69, 0.6, 1.0, 0.749	0.70, 0.75, 1.0, 0.857
SVM	0.73, 0.75, 1.0, 0.857	0.60, 0.6, 1.0, 0.749
Random Forest	0.73, 0.75, 1.0, 0.857	0.87, 0.75, 1.0, 0.857
Adaboost	0.74, 0.75, 1.0, 0.857	0.78, 0.75, 1.0, 0.857
Logistic Regression	0.75, 1.0, 1.0, 1.0	0.60, 0.6, 1.0, 0.749
Naive Bayes	0.79, 0.75, 1.0, 0.857	0.60, 0.6, 1.0, 0.749



Conclusion:

Through this project I have put forward an approach that can be used for the detection of HTs using DFG, GNN and ML/DL models.

Future Works:

One should use the same approach on a bigger dataset. We can also play around with GNN structure to get a more efficient model. One can use ensemble learning and one can use few shot learning using siamese network. One should also consider using CFG for the task.

References:

https://pypi.org/project/pyverilog/