Table of contents

Acknowledgement

I would like to express my deepest appreciation to Dr. Suchismita Roy, my project guide, for her exceptional guidance and support during the development of this project. Dr. Roy's expertise, insightful feedback, and encouragement have been instrumental in shaping the project's direction and outcomes.

I am grateful for the opportunity to work under Dr. Roy's mentorship, and I truly value the knowledge and skills I have gained throughout this journey. Her commitment to excellence and passion for the subject matter have inspired and enriched this project.

Problem Statement:

Title: Graph Neural Networks for Hardware Trojan Detection

Introduction: Hardware Trojan detection is a critical aspect of ensuring the security and integrity of integrated circuits. Traditional methods face challenges in detecting subtle and sophisticated Trojans. This project aims to employ Graph Neural Networks (GNNs) for hardware Trojan detection, utilizing graph representations of circuit designs to enhance accuracy and robustness.

Objectives:

1. **Graph Representation:** Develop effective graph representations of hardware circuit designs suitable for GNN-based analysis.
2. **GNN Architecture:** Design and implement specialized GNN architectures capable of identifying hardware Trojans.
3. **Pooling Strategies:** Investigate and apply graph pooling strategies to capture crucial features and patterns indicative of Trojan presence.
4. **Evaluation Metrics:** Define and employ appropriate evaluation metrics, including accuracy, F1 score, precision, recall, and confusion matrix, for assessing Trojan detection performance.

This project contributes to advancing hardware security by leveraging GNNs to improve Trojan detection capabilities. The developed models have the potential to enhance current detection methods, particularly in scenarios where traditional approaches may fall short.

SOLUTION:

The solution to this problem can be represented broadly, as the following steps:

1. **Input Hardware Code for Generation of Data Flow Graph:** In this step, the goal is to represent the hardware description in the form of a graph. The hardware code is parsed using LALR parser and processed to identify components, features, or entities within the design. Each of these components, represented as segments of Verilog code, becomes a node in the graph, and the relationships or connections between them are represented as edges. For example, if the hardware code describes interconnected modules or gates, nodes in the graph may correspond to these modules, and edges represent the connections between them.

2. **Data Flow Graph to Vector Generation:** Graph Convolutional Neural Network:

   Graph convolutional layers: These are applied to the graph representation. These layers learn node embeddings by aggregating information from neighbouring nodes. The result is a refined node representation that captures local graph structure.

   Pooling Layers: Pooling layers are used to down-sample the graph, reducing its size while retaining essential information. Pooling operations may involve selecting important nodes or consolidating information from groups of nodes.

   Readout Layers: Readout layers aggregate the node embeddings to generate a graph-level representation. This step is crucial as it captures the overall characteristics of the entire graph. Different readout methods (e.g., max pooling, mean pooling) may be used.

   Multi-Layer Perceptron (MLP): The final step involves passing the graph-level representation through an MLP. This layer performs non-linear transformations, enabling the model to capture complex relationships in the data. The output of the MLP is the fixed-size vector representation of the input graph.

3. **Training the Model:** The Graph2Vec model is trained on a dataset of labelled graphs, where each graph has a corresponding label indicating whether it is Trojan-infected or Trojan-free. The training process involves adjusting the model's parameters to minimize a certain loss function, involving a combination of the predicted vector and the ground truth label.

4. **Detecting the Presence of Hardware Trojans:** After training, the model can be used to predict whether a given hardware description contains a Trojan or not. This is done by inputting the graph representation of the hardware into the trained Graph2Vec model, obtaining the corresponding vector representation, loading the previous generated model weights and then using this trained model for classification.

The dataset used for this has been taken from TRUST-HUB.

Input Hardware Code to Graph Representation:

In this step, we convert a given input hardware code to a graphical representation. It is achieved in three stages:

1) Preprocessing
2) Graph Generation
3) Postprocessing

Preprocessing:

```
def preprocess(self, verilog_dir):
    flattened_hw_path = verilog_dir / "topModule.v"
    all_verilog_files = [Path(x).name for x in glob("%s/*.v"%str(verilog_dir))]

    if "topModule.v" not in all_verilog_files:
        self.flatten(verilog_dir, flattened_hw_path)
        self.remove_comments(flattened_hw_path)
        self.remove_underscores(flattened_hw_path)
        self.rename_topModule(flattened_hw_path)

    return flattened_hw_path
```

Preprocessing includes the following functions:

- Flattening: The primary purpose of this function is to take a directory containing Verilog files, flatten them into a single Verilog file, and storing the flattened content. If a file named "topModule.v" is already present in the directory, the function returns without further processing. This avoids unnecessary flattening, if it has already been done. The flatten method uses the 'glob' function from the 'glob' module in Python to find all files containing a specific input_path. For each file that is found, we append its contents to a string, called flatten_content, which is initially empty. Once we traverse through all the files, we write the flatten_content string to an output file, called outfile. Ultimately, this process ends up taking all the contents of files with a specific input_path and putting them into a single file.

```
def flatten(self, input_path, flattened_hw_path):
    flatten_content = ""
    all_containing_files = [Path(x).name for x in glob(fr'{input_path}/*.v', recursive=True)]
    if "topModule.v" in all_containing_files:
        return
    for verilog_file in glob(fr'{input_path}/*.v'):
        with open(verilog_file, "r") as infile:
            flatten_content += infile.read()
    with open(flattened_hw_path, "w") as outfile:
        outfile.write(flatten_content)
```

- Removing Comments and Underscores: to remove single-line comments (denoted by **//**) and remove underscores from a Verilog file. The functions reads the file, processes each

line, and writes the modified content back to the same file. It is a common preprocessing step in code analysis to simplify subsequent parsing or processing steps, to ensure that comments or underscores do not interfere with graph generation or other analyses.

```
def remove_comments(self, hw_path):
    with open(hw_path,'r') as file_in:
        lines = file_in.read().split("\n")

    with open(hw_path, "w") as file_out:
        for line in lines:
            idx = line.find('//')
            if idx == 0:
                continue
            elif idx == -1:
                file_out.write(line+'\n')
            else:
                file_out.write(line[:idx]+'\n')

def remove_underscores(self, hw_path):
    with open(hw_path, 'r') as file_in:
        lines = file_in.read().replace('_', '')

    with open(hw_path, "w") as file_out:
        file_out.write(lines)
```

- Rename the top module: The primary purpose of this function is to rename the top module in a Verilog file to "top." It achieves this by identifying the module declarations and determining the module with a single occurrence, assuming it is the top module. Itinitializes an empty dictionary, called modules_dic, which is meant to store module names as keys and the module name frequency as the values.

```
def rename_topModule(self, hw_path):

    with open(hw_path,'r') as file_in:
        lines = file_in.read().split("\n")

    modules_dic={}
    for line in lines:
        words = line.split()
        for word_idx, word in enumerate(words):
            if word == 'module':
                module_name = words[word_idx+1]
                if '(' in module_name:
                    idx = module_name.find('(')
                    module_name = module_name[:idx]
                    modules_dic[module_name]= 1

                else:
                    modules_dic[module_name]= 0

    for line in lines:
        words = line.split()
        for word in words:
            if word in modules_dic.keys():
                modules_dic[word] += 1

    for m in modules_dic:
        if modules_dic[m] == 1:
            top_module = m
            break

    with open(hw_path, "w") as file_out:
        for line in lines:
            file_out.write(line.replace(top_module, 'top')+'\n')
```

Graph Generation:

After preprocessing, the hardware code must be converted to graph representation, as a DFG(Data Flow Graph) or an AST(Abstract Syntax Tree).

Data Flow Graph represents the flow of data through a digital design. It is a graphical representation that illustrates how data moves between different components or elements in a digital circuit. Nodes represent operations or computations, Edges represent the flow of data between nodes and the direction of the edges indicates the direction in which data flows. In order to produce a DFG, DFGGenerator.process creates an instance of the VerilogDataflowAnalyzer class, called dataflow_analyzer. The VerilogDataflowAnalyzer.generate outputs a parse tree also with the help of YACC. An instance of the VerilogGraphGenerator is then created and assigned to the variable name dfg_graph_generator. The dfg_graph_generator has a binddict, which is a dictionary whose keys are nodes (the signals) and the values are each node's associated dataflow object. For each signal in the dfg_graph_generator binddict, we call the DFGGenerator.generate to create a signal DFG. Lastly, we merge all the signal DFGs together. The resulting graph is a DFG that is in JSON format.

```
class DFGGenerator:
    def __init__(self):
        pass

    def process(self, verilog_file):
        dataflow_analyzer = VerilogDataflowAnalyzer(verilog_file, "top")
        dataflow_analyzer.generate()
        binddict = dataflow_analyzer.getBinddict()
        terms = dataflow_analyzer.getTerms()

        dataflow_optimizer = VerilogDataflowOptimizer(terms, binddict)
        dataflow_optimizer.resolveConstant()
        resolved_terms = dataflow_optimizer.getResolvedTerms()
        resolved_binddict = dataflow_optimizer.getResolvedBinddict()
        constlist = dataflow_optimizer.getConstlist()
        dfg_graph_generator = VerilogGraphGenerator("top", terms, binddict, resolved_terms,
                                                    resolved_binddict, constlist, './seperate_modules.pdf')

        signals = [str(bind) for bind in dfg_graph_generator.binddict]

        for num, signal in enumerate(sorted(signals, key=str.casefold), start=1):
            dfg_graph_generator.generate(signal, walk=False)


    num_nodes = len(dfg_graph_generator.graph.nodes())
    for num, node in enumerate(dfg_graph_generator.graph.nodes(), start=1):
        label = node.attr['label'] if node.attr['label'] != '\\N' else str(node)
        if '_' in label and label.replace('_', '.') in label_to_node:
            parents = dfg_graph_generator.graph.predecessors(node)
            dfg_graph_generator.graph.delete_node(node)
            for parent in parents:
                dfg_graph_generator.graph.add_edge(parent, label_to_node[label.replace('_', '.')])
```

PostProcessing:

The graph obtained G, is in JSON format, which is converted it a NetworkX object, since, libraries like PyTorch Geometric take a NetworkX graph object as their primary data structure in their pipelines, so a conversion from JSON format to a NetworkX format is necessary.
For both the DFG and AST, we initialize the NetworkX graph object, which we name nx_graph, as a NetworkX DiGraph object, which holds directed edges.

```
    nx_graph = nx.DiGraph()

    for node in dfg_graph_generator.graph.nodes():
        node_name = node.name
        if '_graphrename' in node.name:
            node_name = node.name[:node.name.index('_graphrename')]
        if '.' in node_name:
            type_of_node = node_name.split('.')[-1]
        elif '_' in node_name:
            type_of_node = node_name.split('_')[-1]
        else:
            type_of_node = node_name.lower()
        nx_graph.add_node(node.name, label=type_of_node)
        for child in dfg_graph_generator.graph.successors(node):
            # edgeLabel = dfg_graph_generator.graph.get_edge(node, child).attr['label']
            nx_graph.add_edge(node.name, child.name)

    return nx_graph
```

Graph to Vector Representation:

```
def process(self, nx_graph):
    self.normalize(nx_graph)
    data = from_networkx(nx_graph)
    data.hw_name = nx_graph.name
    data.hw_type = nx_graph.type
    self.graph_data.append(data)
```

The next step is to normalize the NetworkX graph object by iterating through the nodes of the nx_graph and giving each node a label that indicates its type.
For the DFG, the type of the node can be numeric, output, input, or signal. The AST node can have a type of names or pure numeric, otherwise the type remains unchanged. The label type is then used to convert each one of the nodes into a vectorized representation. This is done by the normalize function in the DataProcessor class.

```
def normalize(self, nx_graph):
    if self.cfg.graph_type == 'DFG': # normalize for DFG
        in_degrees = [val for (node, val) in nx_graph.in_degree()]
        out_degrees = [val for (node, val) in nx_graph.out_degree()]
        for idx, node in enumerate(nx_graph.nodes(data=True)):
            node_name = node[0]
            if '_graphrename' in node_name:
                node_name = node_name[:node_name.index('_graphrename')]
            if '\'d' in node_name or '\'b' in node_name or '\'o' in node_name or '\'h' in node_name:
                type_of_node = "numeric"
            elif in_degrees[idx] == 0:
                type_of_node = "output"
            elif out_degrees[idx] == 0:
                type_of_node = "input"
            elif '.' in node_name or '_' in node_name:
                type_of_node = "signal"
            else:
                type_of_node = node_name.lower()

            if type_of_node not in self.global_type2idx_DFG:
                print("----------"+type_of_node+"-------------")
                raise Exception("The operation is not in the global_type2idx_DFG table")

            node[1]['x'] = self.global_type2idx_DFG[type_of_node]
        self.num_node_labels = len(self.global_type2idx_DFG)
        self.cfg.num_feature_dim = self.num_node_labels

    elif self.cfg.graph_type == "AST": # normalize for AST
        out_degrees = [val for (node, val) in nx_graph.out_degree()]
        for idx, node in enumerate(nx_graph.nodes(data=True)):
            label = node[1]['label']
            if out_degrees[idx] == 0 and not isInt(label):
                type_of_node = "names"
            elif isInt(label):
                type_of_node = "pure numeric"
            else:
                type_of_node = label.lower()

            if type_of_node not in self.global_type2idx_AST:
                print("----------"+type_of_node+"-------------")
                raise Exception("The operation is not in the global_type2idx_AST table")

            node[1]['x'] = self.global_type2idx_AST[type_of_node]
        self.num_node_labels = len(self.global_type2idx_AST)
        self.cfg.num_feature_dim = self.num_node_labels
```

It converts the graph g into the matrices X, which represents the node embeddings, and A, which represents the adjacency information of the graph. The from_networkx has an empty dictionary, data, which is filled with the graph's nodes and edges. By calling torch.tensor on

each item in data, we create feature vectors for each key in the dictionary. These feature vectors represent the node embeddings, X. from_networkx also creates an adjacency matrix (this represents A) of the NetworkX graph object's edges by calling torch.LongTensor on the list of the graph's edges. This adjacency matrix, called edge_index, becomes an attribute of a dictionary, called data. Finally, data is converted into a PyTorch geometric Data instance by calling torch_geometric.data.Data.from_dict on the dictionary and this object is returned by the function.

The Graph Convolution:

1) Graph Convolution Layers
2) Pooling Layers
3) Readout Layers

Graph Convolution Layer

The core idea of GCNs is based on message passing between nodes. Each node aggregates information from its neighbours, transforming its own feature representation. The aggregation involves weighted combinations of features from neighbouring nodes, where weights are determined by the edges in the graph. Thus, message passing involves two sub-phases: Aggregate and Combine. The AGGREGATE function updates the node embeddings after each k-th iteration to produce $X^{(k)}$ using each node representation $h_v^{(k-1)}$ in $X^{(k-1)}$. The function essentially accumulates the features of the neighboring nodes and produces an aggregated feature vector $a_v^{(k)}$ for each layer k. The COMBINE function combines the previous node feature $h_v^{(k-1)}$ with $a_v^{(k)}$ to output the next feature vector $hv^{(k)}$. The final node embedding after the message propagation is denoted as X^{prop}.

```
class GRAPH_CONV(nn.Module):
    def __init__(self, type, in_channels, out_channels):
        super(GRAPH_CONV, self).__init__()
        self.type = type
        self.in_channels = in_channels
        self.out_channels = out_channels
        if type == "gcn":
            self.graph_conv = GCNConv(in_channels, out_channels)

    def forward(self, x, edge_index):
        return self.graph_conv(x, edge_index)
```

The **__init__** method initializes the layer with the specified type, input channels, and output channels. The **forward** method takes node features (**x**) and edge indices (**edge_index**) as input and applies the GCN operation.

Pooling Layer

The GRAPH_POOL class is designed for graph pooling in a Graph Neural Network (GNN). Graph pooling is a technique used to down-sample or reduce the size of a graph while retaining

important structural information. In this case, two types of pooling methods are implemented: SAGPooling (Self-Attention Graph Pooling) and TopKPooling.

```
class GRAPH_POOL(nn.Module):
    def __init__(self, type, in_channels, poolratio):
        super(GRAPH_POOL, self).__init__()
        self.type = type
        self.in_channels = in_channels
        self.poolratio = poolratio
        if self.type == "sagpool":
            self.graph_pool = SAGPooling(in_channels, ratio=poolratio)
        elif self.type == "topkpool":
            self.graph_pool = TopKPooling(in_channels, ratio=poolratio)

    def forward(self, x, edge_index, batch):
        return self.graph_pool(x, edge_index, batch=batch)
```

The parameters it requires are: type- Specifies the pooling method, either "sagpool" or "topkpool", in_channels- Number of input channels, typically representing the node features and poolratio- A parameter specific to the pooling method, e.g., the ratio of nodes to be retained after pooling, to initialize and then create an instance of either SAGPooling or TopKPooling based on the specified type. In case of the forward function, takes in parameters: x- Input node features, edge_index- Graph connectivity information and batch- A vector indicating which nodes in the batch are part of the same graph. This method passes the input data through the selected pooling layer (SAGPooling or TopKPooling). The pooling operation is applied to the input data, and the down-sampled output is returned. The batch parameter is used to identify the nodes belonging to the same graph during the pooling operation.

Readout Layer

The GRAPH_READOUT class is responsible for performing graph readout in the context of a graph neural network (GNN). Graph readout involves aggregating information from individual nodes in a graph to produce a graph-level representation. The specific type of aggregation is determined by the type parameter, which can be one of three options: "max," "mean," or "add."

```
class GRAPH_READOUT(nn.Module):
    def __init__(self, type):
        super(GRAPH_READOUT, self).__init__()
        self.type = type

    def forward(self, x, batch):
        if self.type == "max":
            return global_max_pool(x, batch)
        elif self.type == "mean":
            return global_mean_pool(x, batch)
        elif self.type == "add":
            return global_add_pool(x, batch)
```

The constructor initializes the GRAPH_READOUT object with a specified type. The type parameter determines the type of aggregation operation to be used during graph readout. The forward method defines the forward pass of the module, which describes how input data is transformed to produce the output. In this case, the input x represents node features, and batch is a vector that indicates which nodes in the batch belong to the same graph. For each type of readout operation, a corresponding global pooling function is called on the node

features x. The choice of readout type determines how information is aggregated across nodes. If type is "max," global_max_pool is applied, which computes the maximum value along each feature dimension across nodes. If type is "mean," global_mean_pool is applied, which computes the mean value along each feature dimension across nodes. If type is "add," global_add_pool is applied, which sums the node features along each feature dimension across nodes. The resulting output represents a graph-level representation that captures the essential information from the individual nodes.

Training the Model and Evaluation:

In this section we aim to train the dataset, validate the dataset and a set of hyperparameter configurations to train the GNN model. This is achieved with the help of following classes:

BaseTrainer:

The BaseTrainer class serves as a generic foundation for training GNN models, providing common functionalities such as model building, visualization of embeddings, metric calculation, and printing. Subclasses, like GraphTrainer and Evaluator, build upon this base for specific tasks. BaseTrainer creates an instance of an Adam optimizer, which implements the Adam algorithm.

```
class BaseTrainer:
    def __init__(self, cfg):
        self.config = cfg
        self.min_test_loss = np.Inf
        self.task = None
        self.metrics = {}
        self.model = None
        np.random.seed(self.config.seed)
        torch.manual_seed(self.config.seed)

    def build(self, model, path=None):
        self.model = model
        self.optimizer = optim.Adam(model.parameters(), lr=self.config.learning_rate, weight_decay=5e-4)

    def visualize_embeddings(self, data_loader, path=None):
        save_path = "./visualize_embeddings/" if path is None else Path(path)
        save_path.mkdir(parents=True, exist_ok=True)

        embeddings, hw_names = self.get_embeddings(data_loader)

        with open(str(save_path / "vectors.tsv"), "w") as vectors_file, \
             open(str(save_path / "metadata.tsv"), "w") as metadata_file:

            for embed, name in zip(embeddings, hw_names):
                vectors_file.write("\t".join([str(x) for x in embed.detach().cpu().numpy()[0]]) + "\n")
                metadata_file.write(name+"\n")


    def get_embeddings(self, data_loader):
        embeds = []
        hw_names = []

        with torch.no_grad():
            self.model.eval()

            for data in data_loader:
                data.to(self.config.device)
                embed_x, _ = self.model.embed_graph(data.x, data.edge_index, data.batch)
                embeds.append(embed_x)
                hw_names += data.hw_name

        return embeds, hw_names

    def metric_calc(self, loss, labels, preds, header):
        acc = accuracy_score(labels, preds)
        f1 = f1_score(labels, preds, average="binary")
        conf_mtx = str(confusion_matrix(labels, preds)).replace('\n', ',')
        precision = precision_score(labels, preds, average="binary")
        recall = recall_score(labels, preds, average="binary")

        self.metric_print(loss, acc, f1, conf_mtx, precision, recall, header)

        if header == "Test" and (self.min_test_loss >= loss):
            self.min_test_loss = loss
            self.metrics["acc"] = acc
            self.metrics["f1"] = f1
            self.metrics["conf_mtx"] = conf_mtx
            self.metrics["precision"] = precision
            self.metrics["recall"] = recall
```

```
def metric_print(self, loss, acc, f1, conf_mtx, precision, recall, header):
    print("%s Loss: %4f" % (header, loss) +
        ", %s Accuracy: %.4f" % (header, acc) +
        ", %s F1 score: %.4f" % (header, f1) +
        ", %s Confusion_matrix: %s" % (header, conf_mtx) +
        ", %s Precision: %.4f" % (header, precision) +
        ", %s Recall: %.4f" % (header, recall))
```

Consider each of the following methods:

Initialization:

- Attributes:
    - self.config: Stores the configuration object, which holds various hyperparameters and settings for the training process.
    - self.min_test_loss: Keeps track of the minimum test loss during training.
    - self.task: Represents the task being performed (initialized as None).
    - self.metrics: A dictionary to store various metrics during training.
    - self.model: Holds the GNN model being trained.
- Random Seed:
    - Sets the random seed for NumPy and PyTorch to ensure reproducibility.

Methods:

1. build(model, path=None):
    - Inputs:
        - model: The GNN model to be trained.
        - path (optional): Path to load a pre-existing model.
    - Functionality:
        - Initializes the model and an Adam optimizer.
        - Optionally loads a pre-existing model from the specified path.
2. visualize_embeddings(data_loader, path=None):
    - Inputs:
        - data_loader: DataLoader containing graph data.
        - path (optional): Path to save the visualization files.
    - Functionality:
        - Obtains graph embeddings and hardware names using get_embeddings.
        - Saves embeddings in TSV format for visualization.
3. get_embeddings(data_loader):
    - Input:
        - data_loader: DataLoader containing graph data.
    - Functionality:
        - Retrieves graph embeddings and hardware names from the model.
    - Returns:
        - Tuple containing embeddings and hardware names.
4. metric_calc(loss, labels, preds, header):
    - Inputs:
        - loss: Current loss value.
        - labels: True labels.

        - preds: Predicted labels.
        - header: Indicates whether it's a training or testing metric.
    - Functionality:
        - Calculates and prints various classification metrics.
        - Updates self.metrics and self.min_test_loss if applicable.
5. metric_print(loss, acc, f1, conf_mtx, precision, recall, header):
    - Inputs:
        - Metrics to print.
    - Functionality:
        - Prints the calculated metrics.

GraphTrainer:

The GraphTrainer class specializes in training GNNs for graph classification tasks, specifically designed for the task labeled "TJ". It leverages the BaseTrainer functionalities and extends them for the specific requirements of the graph classification task. The training process involves forward passes, loss computation, and periodic evaluation of the model's performance.

It inherits from BaseTrainer, meaning it inherits all the attributes and methods from the BaseTrainer class.

```
class GraphTrainer(BaseTrainer):
    ''' trainer for graph classification '''
    def __init__(self, cfg, class_weights=None):
        super().__init__(cfg)
        self.task = "TJ"
        if class_weights.shape[0] < 2:
            self.loss_func = nn.CrossEntropyLoss()
        else:
            self.loss_func = nn.CrossEntropyLoss(weight=class_weights.float().to(cfg.device))

    def train(self, data_loader, valid_data_loader):
        tqdm_bar = tqdm(range(self.config.epochs))

        for epoch_idx in tqdm_bar:
            self.model.train()
            acc_loss_train = 0

            for data in data_loader:
                self.optimizer.zero_grad()
                data.to(self.config.device)

                loss_train = self.train_epoch_tj(data)
                loss_train.backward()
                self.optimizer.step()
                acc_loss_train += loss_train.detach().cpu().numpy()

            tqdm_bar.set_description('Epoch: {:04d}, loss_train: {:.4f}'.format(epoch_idx, acc_loss_train))

            if epoch_idx % self.config.test_step == 0:
                self.evaluate(epoch_idx, data_loader, valid_data_loader)
```

```
def train_epoch_tj(self, data):
    output, _ = self.model.embed_graph(data.x, data.edge_index, data.batch)
    output = self.model.mlp(output)
    output = F.log_softmax(output, dim=1)

    loss_train = self.loss_func(output, data.label)
    return loss_train

def inference_epoch_tj(self, data):
    output, attn = self.model.embed_graph(data.x, data.edge_index, data.batch)
    output = self.model.mlp(output)
    output = F.log_softmax(output, dim=1)

    loss = self.loss_func(output, data.label)
    return loss, output, attn
```

Initialization:

- Attributes:
  - self.task: Initialized as "TJ," representing the task specific to this trainer.
  - self.loss_func: Defines the loss function for the task (cross-entropy loss).
- Initialization based on Class Weights:
  - Checks if the number of classes is less than 2 to determine whether to use class weights in the loss function.

Methods:

1. train(data_loader, valid_data_loader):
   - Inputs:
     - data_loader: DataLoader for training data.
     - valid_data_loader: DataLoader for validation data.
   - Functionality:
     - Trains the GNN model for a specified number of epochs.
     - Uses cross-entropy loss and Adam optimizer.
     - Periodically evaluates the model using evaluate during training.
2. train_epoch_tj(data):
   - Input:
     - data: Graph data for a single training batch.
   - Functionality:
     - Performs a forward pass to get graph embeddings.
     - Applies an MLP layer and log-softmax activation.
     - Computes and returns the cross-entropy loss.
3. inference_epoch_tj(data):
   - Input:
     - data: Graph data for a single batch during inference.
   - Functionality:
     - Similar to train_epoch_tj but used during inference.
     - Returns the loss, model outputs, and attention scores.

```
def inference(self, data_loader):
    labels = []
    outputs = []
    node_attns = []
    total_loss = 0
    folder_names = []

    with torch.no_grad():
        self.model.eval()
        for i, data in enumerate(data_loader):
            data.to(self.config.device)

            loss, output, attn = self.inference_epoch_tj(data)
            total_loss += loss.detach().cpu().numpy()

            outputs.append(output.cpu())

            if 'pool_score' in attn:
                node_attn = {}
                node_attn["original_batch"] = data.batch.detach().cpu().numpy().tolist()
                node_attn["pool_perm"] = attn['pool_perm'].detach().cpu().numpy().tolist()
                node_attn["pool_batch"] = attn['batch'].detach().cpu().numpy().tolist()
                node_attn["pool_score"] = attn['pool_score'].detach().cpu().numpy().tolist()
                node_attns.append(node_attn)

            labels += np.split(data.label.cpu().numpy(), len(data.label.cpu().numpy()))

        outputs = torch.cat(outputs).reshape(-1,2).detach()
        avg_loss = total_loss / (len(data_loader))

        labels_tensor = torch.LongTensor(labels).detach()
        outputs_tensor = torch.FloatTensor(outputs).detach()
        preds = outputs_tensor.max(1)[1].type_as(labels_tensor).detach()

    return avg_loss, labels_tensor, outputs_tensor, preds, node_attns
```

4. inference(data_loader):
    - Input:
        - data_loader: DataLoader for inference.
    - Functionality:
        - Evaluates the model on the provided data loader.
        - Collects and returns average loss, true labels, model outputs, predictions, and node attention scores.

```
def evaluate(self, epoch_idx, data_loader, valid_data_loader):
    train_loss, train_labels, _, train_preds, train_node_attns = self.inference(data_loader)
    test_loss, test_labels, _, test_preds, test_node_attns = self.inference(valid_data_loader)

    print("")
    print("Mini Test for Epochs %d:"%epoch_idx)

    self.metric_calc(train_loss, train_labels, train_preds, header="Train")
    self.metric_calc(test_loss,  test_labels,  test_preds,  header="Test")

    if self.min_test_loss >= test_loss:
        self.model.save_model(str(self.config.model_path_obj/"model.cfg"), str(self.config.model_path_obj/"model.pth"))

    # on final evaluate call
    if(epoch_idx==self.config.epochs):
        self.metric_print(self.min_test_loss, **self.metrics, header="Best")
```

5. evaluate(epoch_idx, data_loader, valid_data_loader):
    - Inputs:
        - epoch_idx: Current epoch index.
        - data_loader: DataLoader for training data.
        - valid_data_loader: DataLoader for validation data.

- Functionality:
    - Calls inference on training and validation data.
    - Prints and stores metrics, and saves the model if the current test loss is the minimum.

Detecting Hardware Trojans

```
import os, sys
sys.path.append(os.path.dirname(sys.path[0]))

import torch
from pathlib import Path
import posixpath
from hw2vec.config import *
from hw2vec.graph2vec.models import *
from hw2vec.hw2graph import *

cfg = Config(sys.argv[1:])

nx_graphs = []
hw2graph = HW2GRAPH(cfg)
hw_project_path = Path(posixpath.normpath("/home/sharmisthak610/Personal/finalproj/TJ-GNN/driver"))
hw_graph = hw2graph.code2graph(hw_project_path)
nx_graphs.append(hw_graph)
```

We import the required modules and then append the directory of the parent of the script's directory to the sys.path. This is to ensure that the script can import modules from its parent directory. An instance of Config is created using command-line arguments (sys.argv[1:]). An empty list nx_graphs is initialized to store NetworkX graphs. An instance of HW2GRAPH is created with the provided configuration. The script then processes hardware code from the specified project path (/home/sharmisthak610/Personal/finalproj/TJ-GNN/driver) into a graph representation using hw2graph.code2graph. The resulting graph is appended to the nx_graphs list.

```
data_proc = DataProcessor(cfg)
for hw_graph in nx_graphs:
    data_proc.process(hw_graph)

input_data = data_proc.get_graphs()[0]


model = GRAPH2VEC(cfg)
model_path = Path(cfg.model_path)
model.load_model(str(model_path/"model.cfg"), str(model_path/"model.pth"))
model.eval()
```

An instance of DataProcessor is created using the configuration (cfg). The script then iterates over the list of hardware graphs (nx_graphs) and processes each graph using data_proc.process(hw_graph). After processing the hardware graphs, the script retrieves the processed graphs using data_proc.get_graphs(). It takes the only processed graph ([0]) and assigns it to the variable input_data. An instance of the GRAPH2VEC model is created using the configuration (cfg). The script then specifies a path for the model (model_path), loads the model configuration (model.cfg), and loads the model weights (model.pth) using model.load_model(). The model is set to evaluation mode using model.eval(). This is important when using the model for inference rather than training, as it disables certain operations like dropout.

```
x = input_data['x']
edge_index = input_data['edge_index']

output = model(x, edge_index, batch=input_data.batch)
predictions = torch.sigmoid(output[0])

threshold = 0.5
res = (predictions > threshold).float().mean().item()

if(res == float(1)):
    print("\nTrojan detected.")
else:
    print("\nTrojan not detected.")
```

This part of the code is performing inference using the previously loaded GRAPH2VEC model. x is extracted from the processed input data, this represents the node features of the graph. edge_index is extracted from the processed input data, representing the edge connections in the graph. The model (GRAPH2VEC) is used for inference by passing the node features (x), edge connections (edge_index), and batch information (input_data.batch) to the model. The model output is stored in the variable output. Then, the model output is passed through a sigmoid function using torch.sigmoid(output[0]) to obtain probability scores. A threshold of 0.5 is used to convert these probabilities into binary predictions (0 or 1) by thresholding with predictions > threshold.

The resulting binary predictions are then converted to a float tensor using .float() and the mean of the tensor is calculated using .mean(). This mean represents the average predicted probability. If the average predicted probability is exactly 1 (with a tolerance for floating-point imprecision), it prints "Trojan detected." Otherwise, it prints "Trojan not detected."

Putting Everything Together:

In order to detect hardware trojan, the following steps are followed:

```
import os, sys
sys.path.append(os.path.dirname(sys.path[0]))
from hw2vec.config import Config
from hw2vec.hw2graph import *
from hw2vec.graph2vec.models import *

cfg = Config(sys.argv[1:])
```

This imports all the necessary files contents and modules. Cfg Creates an instance of the Config class, passing command-line arguments (sys.argv[1:]) to initialize the configuration settings. The configuration settings are likely used to control the behaviour of the script, such as specifying file paths, parameters, or other options.

```
''' prepare graph data '''
if not cfg.data_pkl_path.exists():
    ''' converting graph using hw2graph '''
    nx_graphs = []
    hw2graph = HW2GRAPH(cfg)
    for hw_project_path in hw2graph.find_hw_project_folders():
        hw_graph = hw2graph.code2graph(hw_project_path)
        nx_graphs.append(hw_graph)

    data_proc = DataProcessor(cfg)
    for hw_graph in nx_graphs:
        data_proc.process(hw_graph)
    data_proc.cache_graph_data(cfg.data_pkl_path)

else:
    ''' reading graph data from cache '''
    data_proc = DataProcessor(cfg)
    data_proc.read_graph_data_from_cache(cfg.data_pkl_path)
```

The above code snippet checks whether the file cfg.data_pkl_path exists. cfg.data_pkl_path keeps tracks of the pickled elements of the Verilog code, providing a way to serialize and deserialize Python objects, allowing them to be saved to a file and later reconstructed. If the file does not exist, the script proceeds to convert graphs using a tool called hw2graph, and then processes and caches the graph data using a DataProcessor class. If the file already exists, the script reads the graph data from the cache. In the above code snippet, we first convert the input hardware code to graph representation, where each hardware design is transformed into a NetworkX graph representation and then normalized and transformed into Data instances. If processed Data is already available, then it can be loaded immediately.

```
''' prepare dataset '''
TROJAN = 1
NON_TROJAN = 0

all_graphs = data_proc.get_graphs()
for data in all_graphs:
    if "TjFree" == data.hw_type:
        data.label = NON_TROJAN
    else:
        data.label = TROJAN

train_graphs, test_graphs = data_proc.split_dataset(ratio=cfg.ratio, seed=cfg.seed, dataset=all_graphs)
train_loader = DataLoader(train_graphs, shuffle=True, batch_size=cfg.batch_size)
valid_loader = DataLoader(test_graphs, shuffle=True, batch_size=1)
```

Then, in the dataset preparation shown above, we associate each Data instance with a label corresponding to whether a Trojan exists in the data. Then, we split the entire dataset into two subsets for training and testing depending on user-defined parameters such as ratio and seed. These splits are transformed into DataLoader instances so that PyTorch Geometric utilities can be leveraged.

```
''' model configuration '''
model = GRAPH2VEC(cfg)
if cfg.model_path != "":
    model_path = Path(cfg.model_path)
    if model_path.exists():
        model.load_model(str(model_path/"model.cfg"), str(model_path/"model.pth"))
else:
    convs = [
        GRAPH_CONV("gcn", data_proc.num_node_labels, cfg.hidden),
        GRAPH_CONV("gcn", cfg.hidden, cfg.hidden)
    ]
    model.set_graph_conv(convs)

    pool = GRAPH_POOL("sagpool", cfg.hidden, cfg.poolratio)
    model.set_graph_pool(pool)

    readout = GRAPH_READOUT("max")
    model.set_graph_readout(readout)

    output = nn.Linear(cfg.hidden, cfg.embed_dim)
    model.set_output_layer(output)
```

This code snippet involves the instantiation and configuration of a GRAPH2VEC model based on the provided configuration (cfg). It creates an instance of the GRAPH2VEC class, presumably representing a graph-to-vector model, with the configuration cfg. Then it checks if a pre-trained model path is specified in the configuration (cfg.model_path). If a model path is provided and the corresponding files ("model.cfg" and "model.pth") exist, it loads the pre-trained model using model.load_model().

If no pre-trained model path is specified, it configures the model architecture. It then defines a list of graph convolution layers (convs) with specified parameters (e.g., "gcn" for Graph Convolutional Network, data_proc.num_node_labels, cfg.hidden). The GNN sets the graph convolution layers for the model using model.set_graph_conv(convs) and defines a graph pooling layer (pool) with parameters like "sagpool" for SAGPool and cfg.hidden.

Following this, the GNN sets the graph pooling layer for the model using model.set_graph_pool(pool) and defines a graph readout layer (readout) with the "max" readout method. It sets the graph readout layer for the model using model.set_graph_readout(readout). It defines an output layer (output) as a linear layer with

input size cfg.hidden and output size cfg.embed_dim and sets the output layer for the model using model.set_output_layer(output).

```
''' training '''
model.to(cfg.device)
trainer = GraphTrainer(cfg, class_weights=data_proc.get_class_weights(train_graphs))
trainer.build(model)
trainer.train(train_loader, valid_loader)

''' evaluating and inspecting '''
trainer.evaluate(cfg.epochs, train_loader, valid_loader)
vis_loader = DataLoader(all_graphs, shuffle=False, batch_size=1)
trainer.visualize_embeddings(vis_loader, "./")
```

Now we, move the model to the device specified in the configuration (cfg.device). This is typically done to utilize GPU acceleration if available. We then, create an instance of the GraphTrainer class, which is responsible for training and evaluating the graph-to-vector model and passing the configuration (cfg) and class weights obtained from the training graphs to the trainer.

trainer.build(model) builds the trainer, involving setting up loss functions, optimizers, and other training-related configurations. It takes the model (an instance of GRAPH2VEC) as an argument. We then, initiate the training process using the train_loader for training data and the valid_loader for validation data and evaluate the trained model based on the specified number of epochs (cfg.epochs) using both the training and validation data.

We then, create a data loader (vis_loader) for all graphs (for visualization purposes), which invokes the visualize_embeddings method of the trainer, potentially to generate embeddings and visualize them. The results are saved in the current directory ("./").

Output:

We use the following command line input to run the program:

python3.8 detect.py --yaml_path ./config.yaml --raw_dataset_path ../dataset/TJ-RTL-toy --data_pkl_path dfg_tj_rtl.pkl --graph_type DFG

By this, the command is invoking a Python script called **detect.py** and providing it with several command line arguments. These arguments include paths to a YAML configuration file, a raw dataset, and a data pickle file. Additionally, specification for the type of graph, is set to "DFG."

The top_module.v code obtained for each of the Verilog file is the input for our program. So, either the contents could be copied or the path can be changed accordingly.

Now, the results obtained for the following inputs:

1. For RC6 which is Trojan Free

```
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$ python3.8 detect.py --yaml_path ./config.
yaml --raw_dataset_path ../dataset/TJ-RTL-toy --data_pkl_path dfg_tj_rtl.pkl --graph_type DFG
/home/sharmisthak610/Personal/Trojan/finalproj/TJ-GNN/driver/topModule.v , 2411 , 2716 , 5.207359790802002

Trojan not detected.
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$
```

2. PIC16F84-T300 which is Trojan Infected

```
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$ python3.8 detect.py --yaml_path ./config.
yaml --raw_dataset_path ../dataset/TJ-RTL-toy --data_pkl_path dfg_tj_rtl.pkl --graph_type DFG
/home/sharmisthak610/Personal/Trojan/finalproj/TJ-GNN/driver/topModule.v , 2636 , 3694 , 3.5231425762176514

Trojan detected.
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$
```

3. RSS232-T400 which is Trojan Infected

```
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$ python3.8 detect.py --yaml_path ./config.
yaml --raw_dataset_path ../dataset/TJ-RTL-toy --data_pkl_path dfg_tj_rtl.pkl --graph_type DFG
/home/sharmisthak610/Personal/Trojan/finalproj/TJ-GNN/driver/topModule.v , 747 , 875 , 1.4147279262542725

Trojan detected.
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$
```

4. VGA which is Trojan Free

```
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$ python3.8 detect.py --yaml_path ./config.
yaml --raw_dataset_path ../dataset/TJ-RTL-toy --data_pkl_path dfg_tj_rtl.pkl --graph_type DFG
/home/sharmisthak610/Personal/Trojan/finalproj/TJ-GNN/driver/topModule.v , 2180 , 2463 , 6.658108949661255

Trojan not detected.
(env) sharmisthak610@shark2001:~/Personal/Trojan/finalproj/TJ-GNN/driver$
```

Future scope and improvements:

- Reduce Overfitting- Overfitting occurs in machine learning when a model learns the training data too well, capturing noise or random fluctuations in the data rather than learning the underlying patterns. As a result, an overfit model perform s well on the training data but fails to generalize to new, unseen data. In this case, overfitting can be overcome by increasing the training data.
- Optimization Techniques- The goal is to find the optimal set of parameters that result in the best performance of the model on the given task. This aspect can be improvised on.
- Inference Detection- Currently, we are using logistic regression for inference detection, we can come up with a better and efficient algorithm.
- The variance can be reduced.

Bibliography and References

1. Tao Han, Yuze Wang, and Peng Liu, “Hardware Trojans Detection at Register Transfer Level Based on Machine Learning”
2. Jizhong Yang, Ying Zhang, Member, IEEE, Yifeng Hua, Jiaqi Yao, Zhiming Mao and Xin Chen, “Hardware Trojans Detection through RTL Features Extraction and Machine Learning”
3. Rozhin Yasaei*, Shih-Yuan Yu*, Mohammad Abdullah Al Faruque, “GNN4TJ: Graph Neural Networks for Hardware Trojan Detection at Register Transfer Level”