Classifying the Circuit De-obfuscation Runtime based on Learning Methods and K-means Clustering

Department of Electrical Engineering

Faculty of Electrical and Computer Engineering

University of California, Davis

Principal Investigator:

Dr. Mahdi Orooji

Spring 2022

Contents

[1. Introduction 1](#_Toc107652579)

[2. Materials and Methods 4](#_Toc107652580)

[2.1. Dataset 4](#_Toc107652581)

[2.2. Proposed Framework 4](#_Toc107652582)

[2.2.1. IC labeling 5](#_Toc107652583)

[2.2.2. Feature extraction and selection 7](#_Toc107652584)

[2.2.3. Training, validating, and testing CNF-NET model 11](#_Toc107652585)

[2.3. Implementation details of CNF-NET model 14](#_Toc107652586)

[3. Results 15](#_Toc107652587)

[3.1. Evaluation and assessment 15](#_Toc107652588)

[3.2. Results of applying whole ICs to the proposed framework 16](#_Toc107652589)

[3.2.1. SFFS results 16](#_Toc107652590)

[3.2.2. 9-fold cross-validation results on the whole ICs 16](#_Toc107652591)

[3.3. Results of applying each IC to the proposed framework 17](#_Toc107652592)

[4. Conclusion and Discussion 20](#_Toc107652593)

[4.1. Discussion 20](#_Toc107652594)

[4.2. Conclusion 20](#_Toc107652595)

[Appendix 21](#_Toc107652596)

[References 86](#_Toc107652597)

# Introduction

Semiconductor fabrication is one of the most important steps in designing integrated circuits (ICs). However, the capital and operational costs of this step is considerably high. Due to it, many semiconductor companies outsource their designed ICs to off-shore foundries. Although this trend has different benefits such as low manufacturing costs and reduced design efforts, it could pose various risks and threads to the security and trustworthiness of ICs. The most considerable risks of this trend are IC counterfeiting and unauthorized overproduction, trojan insertion, IP stealing, and circuit piracy [1-4]. It was estimated that these threats could cause $169 billion financial loss per year. Unfortunately, these threats and risks could be posed during any stage of production and design of an IC. Also, it may be remain prevalent after the design release into the industry and market. The main risk and threat that arises from attackers is the invasive reverse engineering (RE) by fully recognizing its functionality stage by stage and extracting the unveiling gate-level netlist. To protect ICs and hardware systems from these threads, **IC obfuscation** techniques have been extensively proposed in recent years [5]. The general approach for IC obfuscation is to introduce the ambiguity in the functionality of the IC based on the obfuscation of some gates. In this scheme, the obfuscated gates cannot be determined by reverse engineering optically, while they preserve the same functionality as the original gates. In other words, such approaches select a part of the IC as the obfuscated part, whose functionality can be retrieved only if correct keys are provided at the additional input gates.

These approaches were effective against conventional attacks [6]. The conventional attacks are based on the fact that there are limited types of gates (e.g., AND, OR, XOR, NOT, MNOR) in an IC, so the attackers can just brute force all the possible combinations of types for all obfuscated gates to find out the one that functions identically to the targeted IC to be deobfuscated. However, advanced attack techniques have been proposed today that obfuscation methods are not very resistant and effective against them [6]. One branch of these techniques that have attracted enormous attention is the Boolean satisfiability problem (SAT)-based attacks [7]. SAT attacks could deobfuscate ICs in milliseconds to several days, years and months. In other words, the runtime of IC deobfuscation by SAT attacks could vary from milliseconds to days or years. This running time highly depends on different factors such as the location of the obfuscated gates, types of obfuscated gates, number of obfuscated gates, circuit topology, and key size. Therefore, it is necessary to obtain a successful obfuscation scheme to resist ICs against advanced attacks. In detail, a successful obfuscation scheme should increase the running time of reverse engineering by advanced attacks, so that this time reaches days, months and even years. However, the implementation of obfuscation schemes will have significant costs in finance, space, time, and power. As a result, such trade-off requires us to search for optimal positions instead of purely increasing their quantity.

In order to maximize the runtime for deobfuscating while incurring minimal overheads, the best set of gates should be selected for obfuscation scheme. Until present, this process is based on human heuristics or experience, which is seriously arbitrary and sub-optimal [8]. In other words, it is unable to “try and error” all the various approaches of obfuscation because it could be millions of combinations to try and the runtime of each attack could be extremely time-consuming. Moreover, the runtime of unknown attacks may be dramatically long and simulating such attacks could be impossible due to the limited resources.

To address the aforementioned issues, we have focused on implementing an approach for automatic recognizing vulnerability or resilience of the ICs against SAT attacks. This research topic is highly under-explored in the community due to its significant challenges listed here:

* **Difficulty in characterizing the algorithmic mechanism of attackers which are mostly sophisticated and hide.** Until now, several deobfuscation techniques have been introduced, which most of them include sophisticated theories, rules, and heuristics [8]. Consequently, conventional learning methods (e.g., support vector machine (SVM), linear regression (LR) [8]) could not characterize the behavior of such highly non-linear and strongly coupled systems.
* **Difficulty in extracting determinant features from discrete and dynamic graph-structured ICs.** The main input of the IC vulnerability or resilience recognizer is the IC topology with selected gates obfuscated. The structure of these data is highly variable. Hence, formulating and integrating this type of data as mathematical forms that could be input to conventional computational and machine learning models is highly challenging. As a result, traditional feature extraction approaches are not intuitive to be applied to such type of data without significant information loss.
* **Requirement for an automatic framework to recognize the resistance or vulnerability of ICs.** As mentioned, it is essential to obtain a successful obfuscation scheme to resist ICs against advanced attacks. To this end, we should select the best scheme based on the simulations of runtimes of the attackers. However, it could be extremely time-consuming and simulating of some attacks could not be possible until present.

In this work, we proposed an automatic framework based on deep learning methods to recognize the resistance or vulnerability of ICs. Recently, deep learning approaches have attained immense success in various classification and regression tasks such as image segmentation, object recognition, machine translation, and data generation, which motivates the generalization of them into graph-structured data [9, 10]. To this end, different architectures of graph neural networks (GNNs) have been proposed to solve classification and regression problems of the graph-structured data [10]. In the problem of the classification of the ICs based on their resistance to advanced attacks, we can formulate ICs as multi-attributed graphs and use GCN models as the predictive models. However, formulating ICs as multi-attributed graphs for the input of GCN models is a challenging task because of varying the ICs structures, which causes the inputs of GCN have different structures. To address the mentioned issues, we used the Conjunctive Normal Form (CNF) method to represent each IC into the graph model. After that, some scalar and vector features were extracted from the CNF representation of each circuit. Next, we used sequneial forward feature selection (SFFS) to select the best scalar features. Then, we used a dual-input GNN model to implement an automatic framework for recognizing the resistance or vulnerability of ICs. Also, we utilized k-means clustering and elbow technique to determine the class of each IC according to the runtime of the attacks. The major contributions of this study are listed as follows:

* Automatically feature extraction of the characteristics and the structure of the IC in the form of CNF representation.
* Selecting the best scalar extracted features from the CNF representation using SFFS algorithm.
* Formulate a dual-input GNN model that could process varying sizes of graph data by taking the best scalar features and vector features.
* Conducting comprehensive experimental assessments and analyses on real-world datasets.

# Materials and Methods

## Dataset

In this work, the datasets are obtained by running SAT algorithm on real-world ISCAS-85 benchmark (c1355, c2670, c3540, c5315, c6288, c7552). It is worth mentioning that each IC was stored by bench files as per the SAT-solver requirements. The obfuscated ICs, along with the original benchmarks, are given to the SAT-solver as input parameters, and SAT-solver tries to find the key. The time taken by the SAT-solver to find the correct key is the actual de-obfuscation time. The samples generated for implementing the proposed framework contains the obfuscation samples generated using varied obfuscation percentage and replacement policies. By this process, we collected the IC samples as bench files and the runtime of SAT-solver on each circuit.

## Proposed Framework

Figure 1 shows the main steps of the proposed framework for automatic recognition of IC resistance against SAT attacks. As shown in Figure 1, we first assigned labels to each IC based on the runtime of SAT attacks. To this end, elbow technique was employed to determine the best number of clusters of the runtimes of SAT attacks on the ICs. After that, k-means clustering with the best number of clusters applied to the runtime of SAT attacks on the ICs. Then, the labels were assigned to the ICs based on the clustering results and boundary between clusters. **In this study, the best number of clusters was equal 2 and the ICs were categorized into the SAT-resilient and SAT-vulnerable.** In the second step, the CNF representation was obtained from the bench file format of each IC. Next, the scalar and vector features were extracted from the CNF representation of each IC. Furthermore, the best scalar features were selected in this step by SFFS method. In the third step, a dual-input GNN model called CNF-NET was trained, validated, and tested, respectively by the scalar and vector features and the labels of the ICs. Finally, the trained and validated CNF-NET model was deployed to classify each IC into the SAT-resilient or SAT-vulnerable classes. Following, each step was described in detail.

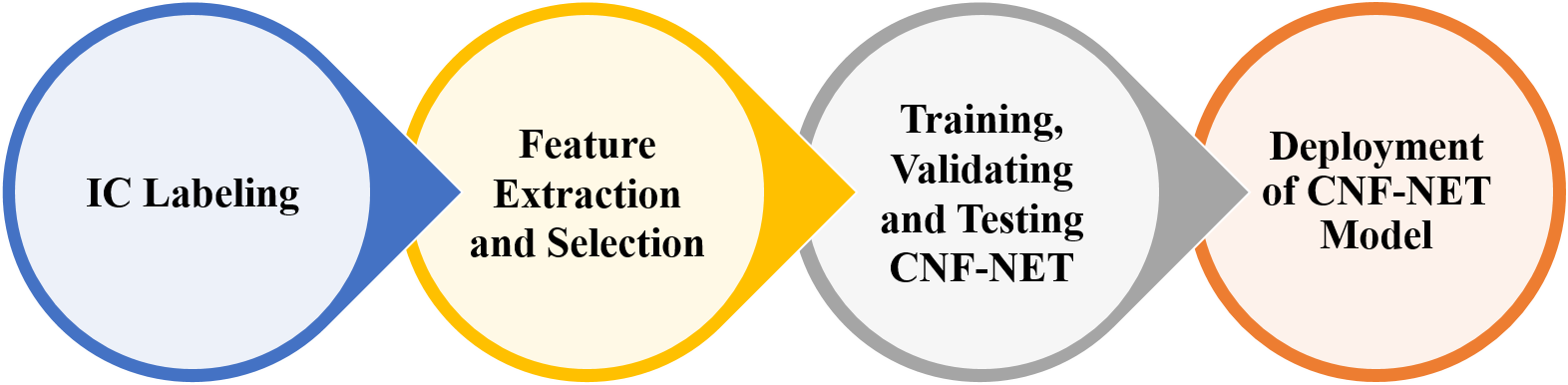


Figure 1: The main steps for implementing the presented framework for automatic recognition of IC resistance against SAT attacks

### IC labeling

For developing a supervised classification framework, some labels should be defined for each sample. In this study, the runtimes of SAT attacks on the ICs were available and we should perform a procedure to map these runtime into the class labels. To this end, an unsupervised process with the best number of clusters should be used to cluster each runtime into the separated clusters. In order to determine the best number of clusters, the elbow technique method was employed to determine the best number of clusters. This method selects the optimal number of clusters by fitting an unsupervised model with a range of values for number of clusters. In this work, the minimum and maximum number of clusters were chosen to 2 and 10, respectively. Also, the unsupervised model used in the elbow technique in this study was K-means clustering. It is worth mentioning that all SAT runtimes for all ICs were collected and applied to the elbow technique. Figure 2 shows the elbow technique results for determining the best number of clusters for the SAT runtimes based silhouette score. As illustrated in Figure 2, the highest silhouette score was obtained in the two number of clusters. Figure 3 shows the K-means clustering result by *K=2* on the SAT runtimes on

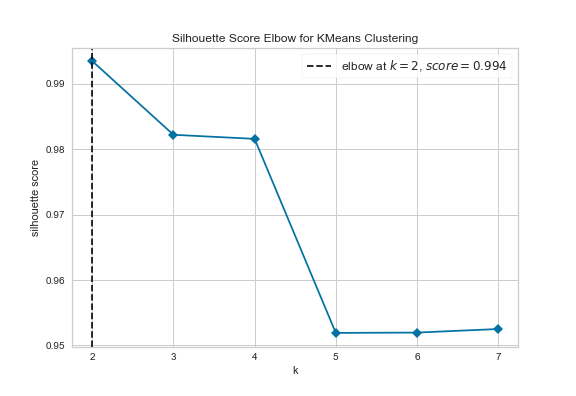


Figure 2: Elbow technique results for determining the best number of clusters for SAT runtimes on the ICs.

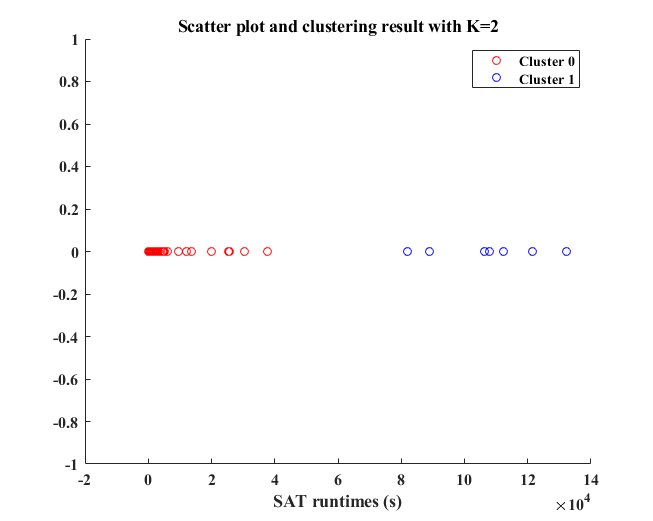


Figure 3: The K-means clustering result with K=2 on the SAT runtimes on the ICs.

the ICs. According to the K-means clustering results, the label of cluster 0 and cluster 1 were assigned to the SAT-vulnerable and SAT-resilient classes, respectively. Also, a mapping function was formulated based on the boundaries between cluster 0 and cluster 1, which is shown as follows:

|  |  |
| --- | --- |
|  | ‑ |

where *g(x)* is the mapping function, x is the SAT runtime on the IC in seconds, *y* is the label of the IC, and *m* is the threshold value between classes. In this function, ­the 0 and 1 values of *y* refer to the SAT-resilient and SAT-vulnerable cases, respectively. The value of *m* was obtained from the average of the highest value of cluster 0 and the lowest value of cluster 1. In our experiments, the value of ­*m* was obtained equal 59830 second.

### Feature extraction and selection

In this step, the CNF representation was obtained from bench file format of each IC. Next, some scalar and vector features were extracted from the CNF repression of each IC. After that, the best scalar features were selected using SFFS algorithm. In the following, the CNF representation and its derived features were explained in details. The blockdiagram of this step is shown in Figure 4.

#### CNF representation and its derived features

One of the main challenges in this study was the representing the obfuscated netlist in an intact and structured way for a machine leaning framework. Although CNF is typically as a sequence, it is mathematically not a sequence as the order among different clauses is meaningless. Moreover, one literal can appear in multiple clauses with or without their negation forms, which further complicates its representation. Also, there are very few existing machine learning methods that are designed for directly treating CNF representation as input without information loss. In order to solve this problem, the converted CNF of an obfuscated IC was modeled as an undirected and signed bipartite graph which uses one node type for clauses and the other for literals. Mathematically, the CNF bipartite graph could be defined as *G(E, Vliteral, Vclause)*, where *E, Vliteral,* and *Vclause* are the edges between nodes, the set of literal nodes, and the set of clause nodes, respectively. The value of *E* could be 0,+1, and -1. Consider a literal node as *l­­* and a clause node as ­*c*. If *l­* is in *c* and *l* is positive, the value of *E* between them is 1. If *l­* is in *c* and *l* is negative, the value of *E* between them is -1, and if there is no connection between *l* and *c*, the value of *E* is 0.

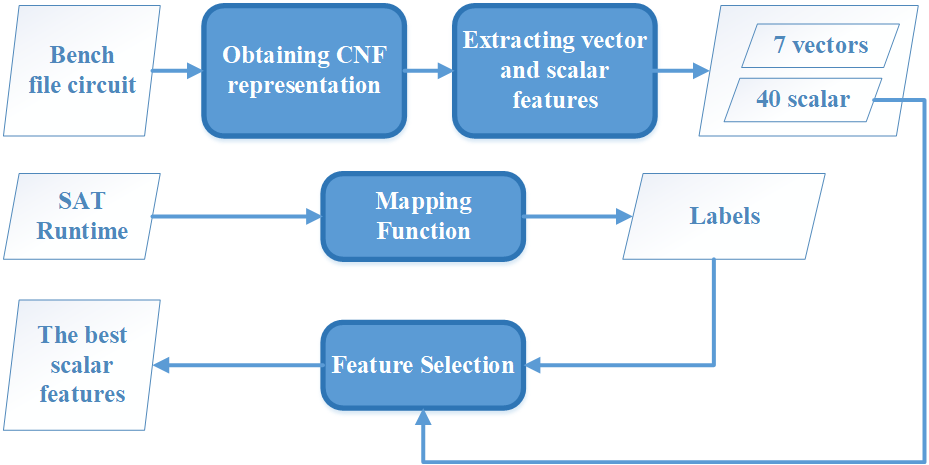


Figure 4: The blockdiagrm of feature extraction and selection of the proposed method

In other words, the adjacency matrix of CNF bipartite graph representation of an IC could be modeled as follows:

|  |  |
| --- | --- |
|  | ‑ |

where *IM* is the adjacency matrix, *M ­*is the number of clause nodes, and *N* is the number of literal nodes. Firstly, ­*IM* was obtained by the CNF transformation from each IC bench file. Next, all zero rows were eliminated from *IM* matrix. In the next step, literal and clause degrees were computed from ­­*IM* matrix using following equations:

|  |  |
| --- | --- |
|  | ‎2‑ |
|  | ‑ |

where |.|, *cd*, and *ld* are the absolute function, clause degree, and literal degree, respectively. Literal graph (LG) was also obtained from *IM* matrix, which could be modeled as *G1(E1,V1)*, where *V1*is the literal nodes and *E­1* is the edges of LG graph. In LG graph, *Ei,j* is defined as the count of clauses in which both *Vi ­*and *Vj* exist. After obtaining LG graph, the literal degree of it (*ldLG)* was also computed similar ‎2‑3 equation. Next, positive adjacency matrix (*pam*) and negative adjacency matrix (*nam*) were obtained based on the *IM* matrix. To obtain ­*pam*, all values of -1 of ­*IM* matrix was replaced with 0. For computing ­*nam*, all values of 1 and -1 of ­*IM* matrix was replaced with 0 and 1, respectively. Mathematically, *pam=[b­i,j]­M×N­* and *nam=[c­i,j]­M×N­* where each element of these matrixes are 0 or 1. Then, *pratio0*, *pratio1*, ­*nratio0*, and *nratio1* parameters were calculated as follows:

|  |  |
| --- | --- |
|  | ‑ |
|  | ‑ |
|  | ‑ |
|  | ‑ |

The vector features that are extracted from CNF representation and considered as the inputs of the machine learning model are *ld*, *cd, ldVG, pratio0*, *pratio1*, ­*nratio0*, and *nratio1*. After extracting vector features, the scalar features were obtained from the mentioned vector features and *IM* matrix. Three first scalar features are ­*M*, *N,* and *M/N*, which *M* and N denote the number of clause and literal nodes, respectively. Moreover, the statistical features of the mentioned vector features are extracted. These features are mean, variance, entropy, skewness, and kurtosis. Also, the ratios of binary and ternary clauses are calculated as another scalar features. Figure 4 illustrates the diagram of feature extraction of the CNF representation, which causes to obtain 40 scalar features and 7 vector features.

#### Feature Selection

The extracted features of a dataset in machine learning frameworks may contain redundant or irrelevant features. Moreover, redundant or irrelevant features could increase the computational load and lead to overfitting the model. The main purpose of feature selection is to find the best subset of features, which could reduce the dimension of feature space, and improve the performance of the machine learning framework. In this study, we applied the scalar features to SFFS algorithm to obtain the best subset of them. The steps of SFFS method in this work are summarized in Algorithm 1. The objective criterion (*J*) was defined as the mean of accuracies during 10-fold cross-validation. Based on the steps in Algorithm 1, the features are sequentially added to an empty candidate set until the addition of further features does not decrease ­*J* value. In this study, the classification model in the SFFS was k-nearest neighbor (KNN) model with K=2. In our experiments, SFFS returned 3 features form 40 features as the best subset of features. These features are ­*M*, mean of *cd*, and kurtosis of ­*pratio1*.

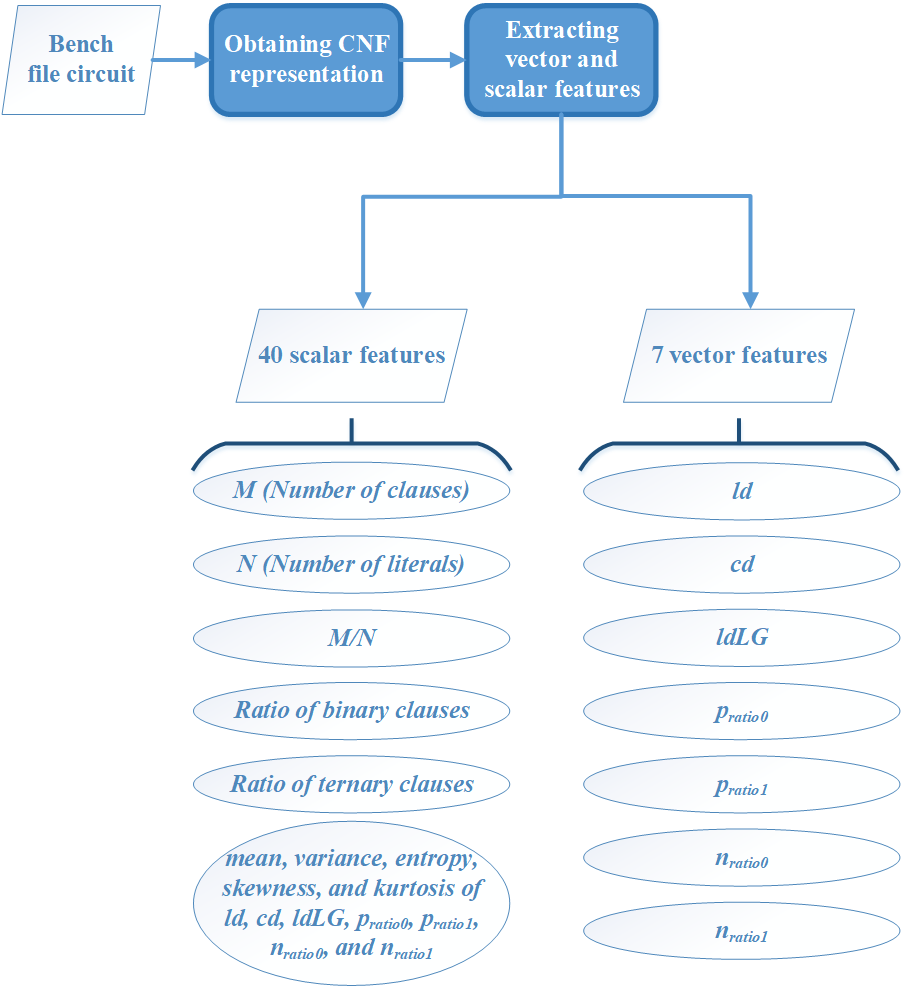


Figure 5: The diagram of feature extraction of the proposed framework

|  |
| --- |
| **Algorithm 1: The SFFS algorithm** |
| Input: The entire feature set, *Y = {y1, y2, ..., yd}*  Output: The selected feature subset, *S*k *= {Sj|j = 1, 2, ..., k; SjϵY }* where *k = 1, 2, ..., d*   1. Start with the empty set, *S0 = {}*, *k=0*. 2. *s∗ = argmax (J (Sk + s))*, where *s ϵ Y − Sk*. 3. Update *Sk+1 = Sk + s∗*; *k = k + 1*. 4. If *J(Sk) > J(Sk+1)*, go to step 6. 5. Else go to the step 2. 6. Stop |

### Training, validating, and testing CNF-NET model

After extracting scalar and vector features from CNF representation of each IC, the CNF-NET model was trained and validated to classify each IC into the SAT-resilient or SAT-vulnerable classes. The CNF-NET is a dual-input GNN model, which receives the vector, and scalar features as individual inputs. Figure 6 depicts the details of the structures of the CNF-NET model. As depicted in Figure 6, the main parts of the CNF-NET are incidence convolution, categorizing layer, and fully-connected (FC) layers. For each case, the vector features are applied to the incidence convolution part, and then the selected scalar features are categorized with incidence convolution’s output. Next, FC layers receives the categorization results and generates two probabilities for each case. The value of these probabilities indicates the probability that the input case of CNF-NET belongs to each of the SAT-vulnerable and SAT-resilient classes. Based on these probabilities, the input case is classified into the mentioned classes. In the following, each part of CNF-NET is explained in details.

#### Incidence convolution

The first part of CNF-NET, which receives the vector features as inputs, is incidence convolution. This part generates a vector with 7 elements as the output. In other words, it generates a scalar value for each input vector. In this part, consider *f­φ()* as a sequence of FC layers, which its input and output are scalar values. The details of the structure of *f­φ()* is given in Table 1. Also, suppose the core function of incidence convolution as *gα()* and each input vector as *Vi* which is applied to *gα()*. The main steps of *gα()* function is summarized in Algorithm 2. *gα()* generates a scaler value for each *Vi* *i=1, 2, …, 7­* and stores them in a vector. In the first step of *gα()*, ­*Vi* is normalized which results in *Vinorm*. Then, each element of ­*Vi* is applied to *f­φ()* and the outputs of it is stored in a vector

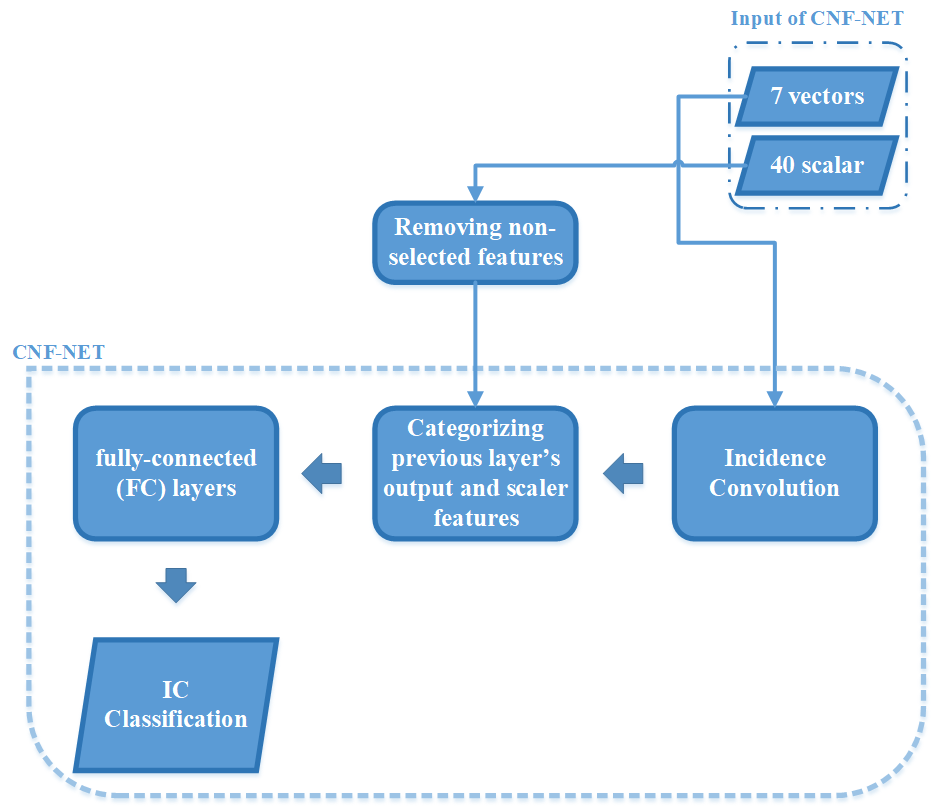


Figure 6: The diagram of the structures of CNF-NET model.

Table 1: The architecture and layers of fα() function

|  |  |  |  |
| --- | --- | --- | --- |
| **Layer Number** | **Layer** | **Input** | **Output** |
| 1 | FC | 1 | 32 |
| 2 | FC | 32 | 32 |
| 3 | FC | 32 | 32 |
| 4 | FC | 32 | 1 |

called *Wi­*. After that, *Wi* is multiplied to *Vinorm* and then the summation of the output of this multiplication is computed. Finally, the summation result is stored in the output vector of incidence convolution.

|  |
| --- |
| **Algorithm 2: *gα() function*** |
| Input: Vector features, *V = {V1, V2, ..., V7}*  Output: A vector with 7 elements, *Vo={Vok}* where *k=1, 2, …, 7*   1. Initialization: *i=1, Wi={},* ­and *Vo={}* 2. Normalizing *Vi* using 3. Apply each element of *Vinorm* to *f­φ()* function and store the outputs in *Wi* 4. Computing and appending it to *Vo* 5. *i = i +1* 6. If *i≤7*, go to step 2 7. Else go to step 8 8. Stop |

#### Categorizing and FC parts

The next parts of CNF-NET model after incidence convolution are categorizing and FC layers parts, respectively. The categorizing layer receives the selected scalar features and the outputs of incidence convolution and categorizes them into a vector. Next, the categorization result is applied to FC part, which consists of multiple FC layers. Table 2 shows the architecture of the layers of FC part. The last layer of the FC part is a softmax layer, which generates the probabilities that indicate the input case of CNF-NET belongs to each of the SAT-vulnerable and SAT-resilient classes. Based on the values of these probabilities, the input case is classified into the mentioned classes.

Table 2: The architecture and layers of FC part

|  |  |  |  |
| --- | --- | --- | --- |
| **Layer Number** | **Layer** | **Input** | **Output** |
| 1 | FC | 10 | 32 |
| 2 | FC | 32 | 32 |
| 3 | FC | 32 | 32 |
| 4 | FC | 32 | 2 |
| 5 | Softmax | 2 | 2 |

## Implementation details of CNF-NET model

The proposed framework for recognizing IC resistance against SAT attacks was implemented under Python 3.7 coding environment. The CNF representation of each IC was obtained using bench2cnf module. To extract features from CNF representation, we used numpy and scipy libraries. For implementing SFFS algorithm, we used mlxtend library in our study. For implementing k-means clustering and 10-fold cross-validation, scikit-learn framework was used. Also, elbow technique was employed in this work using Yellowbrick library. In order to implement the CNF-NET as the core of the proposed method, Pytorch framework was used. For training the model, the epochs, learning rate, batch size was set 50, 0.01, and 32 values, respectively. It is worth mentioning that that all simulations and implementations were conducted on a system with Intel® Xeon® Processor E5-2697 v2 CPU at 2 GHz and 8 GB memory.

# Results

In this section, the performance of the proposed framework for classifying each IC into the SAT-vulnerable and SAT-resilient classes is investigated. First, the evaluation of the proposed method and performance metrics are explained in details. After that, the obtained results of the assessment of the proposed framework are reported.

## Evaluation and assessment

To evaluate the proposed framework, 10-fold cross-validation technique was employed in this study. In 10-fold cross-validation technique, the dataset is initially divided into 10 folds, of which 9 folds are randomly used as a training set, and the remaining fold is employed as a testing set. This process is repeated 10 times until each fold is utilized as a testing set. During each iteration, the testing set is applied to the trained model, which results in 10 different evaluation metrics values. In this work, the 10% of training set is randomly categorized into the validation set in each iteration of 10-fold cross-validation. The main reason for using validation set during training phase is to prevent overfitting by early stopping the training phase. The performance metrics used in this study to investigate the classification performance of the proposed framework are accuracy (*AC*), sensitivity (*SE*), specificity (*SP*), and F1-score (*F1*), which are defined as follows:

|  |  |
| --- | --- |
|  | ‑ |
|  | ‑ |
|  | ‑ |
|  | ‑ |

where *TP* is the number of SAT-vulnerable cases that are correctly classified, *FN* is the number of SAT-vulnerable samples that are incorrectly classified as SAT-resilient samples, *FP* is the number of SAT-resilient samples that are incorrectly classified as SAT-vulnerable cases, and *TN* is the number of SAT-resilient cases that are correctly classified.

## Results of applying whole ICs to the proposed framework

### SFFS results

Figure 7 shows the SFFS result for selecting best subset of features where y-axis and x-axis are returned accuracy and number of features. In the experiments, SFFS returned a subset of features with 3 attributes as the selected features. By these three features, the returned accuracy by SFFS was 0.95, which it is the maximum accuracy in Figure 7. The selected features are ­*M*, mean of *cd*, and kurtosis of ­*pratio1*.

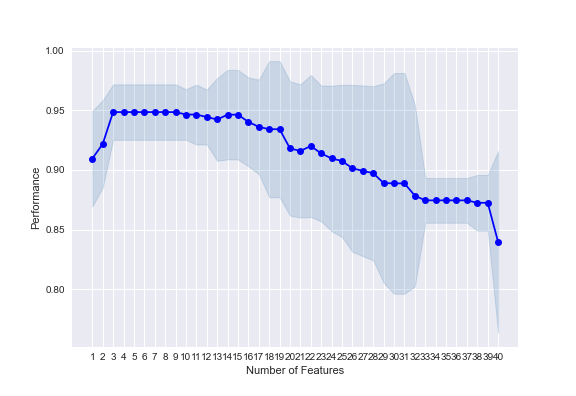


Figure 7: The returned accuracies by SFFS for each number of candidate features

### 9-fold cross-validation results on the whole ICs

Table 3 reports the obtained *AC*, *SE*, *SP­­*, and *F1* values by the proposed framework in each repletion of 10-fold cross-validation. The average of AC, SE, and F1 metrics are 88.54%, 88.32%, and 93.17%, respectively. As shown in Table 7, the distribution of data between positive and negative classes is so imbalance. This is due to lower number of negative cases against positive samples.

Table 3: The obtained values of the percentage (%) of AC, SE, SP, and F1 metrics of the proposed method in each iteration of 9-fold cross-validation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Iteration Number** | **Number of positive cases** | **Number of negative cases** | **AC (%)** | **SE (%)** | **SP (%)** | **F1 (%)** |
| 1 | 32 | 0 | 100 | 100 | nan | 100 |
| 2 | 32 | 0 | 100 | 100 | nan | 100 |
| 3 | 30 | 2 | 93.75 | 93.75 | nan | 96.77 |
| 4 | 23 | 9 | 81.25 | 79.31 | 100 | 88.46 |
| 5 | 30 | 2 | 93.75 | 93.75 | nan | 96.77 |
| 6 | 19 | 13 | 59.37 | 59.37 | nan | 74.50 |
| 7 | 31 | 1 | 96.87 | 96.87 | nan | 98.41 |
| 8 | 32 | 0 | 100 | 100 | nan | 100 |
| 9 | 23 | 9 | 71.87 | 71.87 | nan | 83.63 |

## Results of applying each IC to the proposed framework

In this step, the proposed framework was evaluated by each IC category individually using 10-fold cross-validation technique. In detail, it was assessed by C2670, C3540, and C6288 ICs, which were obfuscated using different approaches and keys. Tables 4, 5, and 6 report the obtained results of the proposed framework in 10-fold cross-validation for C2670, C3540, and C6288 ICs, respectively. For C2670 IC set, the average of AC, SE, and F1 metrics obtained by the proposed method were 96.88%, 98%, and 95.55%, respectively. The average of AC, SE, and F1 metrics obtained by the proposed method for C3540 IC set were 87.98%, 92%, and 86.72%, respectively. For C6288 IC set, the proposed method provided an average AC of 64.66%, SE of 72.44%, and F1 of 71.20%.

Table 4: The obtained values of the percentage (%) of AC, SE, SP, and F1 metrics of the proposed method in each iteration of 10-fold cross-validation for C2670 IC set

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Iteration Number** | **Number of positive cases** | **Number of negative cases** | **AC (%)** | **SE (%)** | **SP (%)** | **F1 (%)** |
| 1 | 8 | 2 | 80 | 80 | nan | 88.88 |
| 2 | 10 | 0 | 100 | 100 | nan | 100 |
| 3 | 10 | 0 | 100 | 100 | nan | 100 |
| 4 | 9 | 0 | 100 | 100 | nan | 100 |
| 5 | 1 | 8 | 100 | 100 | 100 | 100 |
| 6 | 2 | 7 | 100 | 100 | 100 | 100 |
| 7 | 9 | 0 | 100 | 100 | nan | 100 |
| 8 | 9 | 0 | 100 | 100 | nan | 100 |
| 9 | 2 | 7 | 88.88 | 100 | 87.5 | 66.66 |
| 10 | 8 | 1 | 100 | 100 | 100 | 100 |

Table 5: The obtained values of the percentage (%) of AC, SE, SP, and F1 metrics of the proposed method in each iteration of 10-fold cross-validation for 3540 IC set

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Iteration Number** | **Number of positive cases** | **Number of negative cases** | **AC (%)** | **SE (%)** | **SP (%)** | **F1 (%)** |
| 1 | 8 | 2 | 80 | 80 | nan | 88.88 |
| 2 | 10 | 0 | 100 | 100 | nan | 100 |
| 3 | 10 | 0 | 100 | 100 | nan | 100 |
| 4 | 9 | 0 | 100 | 100 | nan | 100 |
| 5 | 1 | 8 | 100 | 100 | 100 | 100 |
| 6 | 2 | 7 | 66.66 | 40 | 100 | 57.14 |
| 7 | 9 | 0 | 100 | 100 | nan | 100 |
| 8 | 9 | 0 | 100 | 100 | nan | 100 |
| 9 | 2 | 7 | 88.88 | 100 | 87.5 | 66.66 |
| 10 | 8 | 1 | 44.44 | 100 | 16.66 | 54.54 |

Table 6: The obtained values of the percentage (%) of AC, SE, SP, and F1 metrics of the proposed method in each iteration of 10-fold cross-validation for 6288 IC set

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Iteration Number** | **Number of positive cases** | **Number of negative cases** | **AC (%)** | **SE (%)** | **SP (%)** | **F1 (%)** |
| 1 | 8 | 2 | 80 | 80 | nan | 88.88 |
| 2 | 10 | 0 | 100 | 100 | nan | 100 |
| 3 | 10 | 0 | 100 | 100 | nan | 100 |
| 4 | 9 | 0 | 100 | 100 | nan | 100 |
| 5 | 1 | 8 | 11.11 | 11.11 | nan | 20 |
| 6 | 2 | 7 | 22.22 | 22.22 | nan | 36.36 |
| 7 | 9 | 0 | 22.22 | 100 | 0 | 36.36 |
| 8 | 9 | 0 | 100 | 100 | nan | 100 |
| 9 | 2 | 7 | 22.22 | 22.22 | nan | 36.36 |
| 10 | 8 | 1 | 88.88 | 88.88 | nan | 94.11 |

# Conclusion and Discussion

## Discussion

As mentioned, the main purpose of this study is to implement an automatic machine-learning framework for classifying ICs to SAT-vulnerable and SAT-resilient classes. In other words, the proposed framework should recognize the IC resistance against SAT attacks. In terms of classification performance, the proposed method performed well in classifying SAT-vulnerable cases. However, it did not provide an acceptable performance in classifying SAT-resilient cases. This could be due to the imbalance distribution between SAT-vulnerable and SAT-resilient cases. In terms of time performance, the proposed framework could classify each IC in 236 seconds. This time is much lower than applying directly SAT attacks to ICs. Therefore, it could be used for automatic recognition of resistance of ICs against SAT attacks. As future works, such framework could be implemented to find the appropriate obfuscation method and key.

## Conclusion

In this work, an automatic framework for recognizing the IC resistance against SAT attacks was introduced. In the first step of this framework, the SAT runtimes of each IC was mapped to SAT-vulnerable and SAT-resilient classes to prepare the labels for training the CNF-NET model. It was performed by k-means clustering and elbow technique. Next, the CNF representation was extracted from IC bench file. After that, 7 vector features and 40 scalar features were computed from the CNF representation and then, the scalar features were applied to SFFS algorithm to select the best scalar features. Next, the CNF-NET model was trained, validated, and tested, respectively by the selected scalar and vector features and the labels of the ICs to classify each IC to the SAT-vulnerable and SAT-resilient classes. The model was validated using 9-fold cross-validation technique, which provided an average AC of 88.54%, SE of 88.32%, and F1 of 93.17%. Based on the obtained results, the proposed method had an acceptable performance in in recognizing SAT-vulnerable cases. However, it did not perform well in classifying SAT-resilient cases.

# Appendix

1. **Main codes**

In the first step, the Data\_Generator.py should be run to extract the features and SAT runtime for each circuit. This code generates two pickle files with \_X and \_Y suffixes that instead of each \_, there is circuit name.

* **Data\_Generator.py:**

import numpy as np

import os

import itertools

import copy

from collections import Counter

from bench2cnf import simple\_read\_bench, tseytin\_t, cnf\_to\_matrix

import pickle as pk

import scipy.stats as ss

import pandas as pd

#from scipy.sparse import csgraph

#from grakel.graph import floyd\_warshall

from scipy.stats import skew

from scipy.stats import kurtosis

def Time\_calculation(Time\_List):

Time\_Target = float(Time\_list[0])\*60 + float(Time\_list[1].split('s')[0])

return Time\_Target

def ent(data):

""" input probabilities to get the entropy """

entropy = ss.entropy(data)

return entropy

data\_dir = os.path.join(os.getcwd(),'Dataset 2\EncryptedBenches')

df = pd.read\_csv(os.path.join(os.getcwd(),'Dataset 2\decryption\_time.csv'))

df1 = df[["CircuitName"]]

CircuitNames = df1.values.tolist()

CircuitNames = [x[0] for x in CircuitNames]

CircuitNames = set(CircuitNames)

for each\_benchmark in CircuitNames:

each\_df = df.loc[df['CircuitName']==each\_benchmark]

All\_data\_features = []

Time\_Targets = []

#print(each\_benchmark)

for ind in each\_df.index:

Time = each\_df['Time'][ind]

if Time =='-1':

#continue

Time\_Target = float(Time)

else:

Time\_list = Time.split('m') # The first and second elements are minute and second, respectively.

Time\_Target = Time\_calculation(Time\_list)

CircuitName = each\_df['CircuitName'][ind]

EncryptionScheme = each\_df['EncryptionScheme'][ind]

KeyLength = each\_df['KeyLength'][ind]

Fraction = each\_df['Fraction'][ind]

bench\_file\_suffix = "\_keys" if Fraction== 0 else "\_per" # It determines the name of bench file which could be formed by \_per or \_keys

percent\_obfuscated = Fraction if Fraction != 0 else KeyLength

Primary\_Link\_0 = os.path.join(data\_dir,CircuitName,EncryptionScheme,'1')

Primary\_Link\_1 = os.path.join(Primary\_Link\_0,str(percent\_obfuscated) + bench\_file\_suffix + ".bench")

wires = simple\_read\_bench(Primary\_Link\_1)

cnf\_clause\_count, cnf\_content = tseytin\_t(wires)

incidence\_mat = cnf\_to\_matrix(cnf\_content)

if incidence\_mat.size == 0:

continue

feat = []

# check if there is zero rows

zero\_row\_id = (incidence\_mat == 0).all(1)

incidence\_mat = incidence\_mat[~zero\_row\_id] # remove all zero rows

cnf\_clause\_count, cnf\_variable\_counts = incidence\_mat.shape # refresh var and clause num

# derive literal degree and clause degree

var\_degree = np.sum(np.abs(incidence\_mat), axis=0)

clause\_degree = np.sum(np.abs(incidence\_mat), axis=1)

# build literal graph and extract literal degree and features

var\_graph = np.zeros((cnf\_variable\_counts, cnf\_variable\_counts))

for i in range(cnf\_clause\_count):

non\_zero\_list = np.where(incidence\_mat[i] != 0)

pair\_combo = [pair for pair in itertools.product(non\_zero\_list[0], repeat=2)]

for p in pair\_combo:

var\_graph[p] = 1

var\_graph\_degree = np.sum(var\_graph, axis=0)

#var\_graph\_floyd = floyd\_warshall(var\_graph)

#var\_graph\_floyd\_degree = np.sum(var\_graph\_floyd, axis=0)

# extract positive and negative features from literal-clause graph

positive\_mat = copy.deepcopy(incidence\_mat)

positive\_mat[positive\_mat == -1] = 0

negative\_mat = copy.deepcopy(incidence\_mat)

negative\_mat[negative\_mat == 1] = 0

negative\_mat[negative\_mat == -1] = 1

positive\_mat\_0 = np.sum(positive\_mat, axis=0)

negative\_mat\_0 = np.sum(negative\_mat, axis=0)

positive\_mat\_1 = np.sum(positive\_mat, axis=1)

negative\_mat\_1 = np.sum(negative\_mat, axis=1)

p\_ratio\_0 = positive\_mat\_0 / (positive\_mat\_0 + negative\_mat\_0)

n\_ratio\_0 = negative\_mat\_0 / (positive\_mat\_0 + negative\_mat\_0)

p\_ratio\_1 = positive\_mat\_1 / (negative\_mat\_1 + positive\_mat\_1)

n\_ratio\_1 = negative\_mat\_1 / (negative\_mat\_1 + positive\_mat\_1)

# count ratio of binary and ternary clause

bin\_tern\_cnt = Counter(np.sum(np.abs(incidence\_mat), axis=1))

bin\_ratio = bin\_tern\_cnt[2] / sum(bin\_tern\_cnt.values())

tern\_ratio = bin\_tern\_cnt[3] / sum(bin\_tern\_cnt.values())

feat.append(tern\_ratio)

feat.append(bin\_ratio)

feat.append(cnf\_variable\_counts) # 0

feat.append(cnf\_clause\_count) # 1

feat.append(float(cnf\_clause\_count / cnf\_variable\_counts)) # 2

# extract entropy for each feature

feat.append(ent(var\_degree))

feat.append(ent(clause\_degree))

feat.append(ent(var\_graph\_degree))

feat.append(ent(p\_ratio\_0))

feat.append(ent(p\_ratio\_1))

feat.append(ent(n\_ratio\_0))

feat.append(ent(n\_ratio\_1))

#feat.append(ent(var\_graph\_floyd\_degree))

feat.append(np.mean(var\_degree))

feat.append(np.mean(clause\_degree))

feat.append(np.mean(var\_graph\_degree))

feat.append(np.mean(p\_ratio\_0))

feat.append(np.mean(p\_ratio\_1))

feat.append(np.mean(n\_ratio\_0))

feat.append(np.mean(n\_ratio\_1))

#feat.append(np.mean(var\_graph\_floyd\_degree))

feat.append(np.var(var\_degree))

feat.append(np.var(clause\_degree))

feat.append(np.var(var\_graph\_degree))

feat.append(np.var(p\_ratio\_0))

feat.append(np.var(p\_ratio\_1))

feat.append(np.var(n\_ratio\_0))

feat.append(np.var(n\_ratio\_1))

#feat.append(np.var(var\_graph\_floyd\_degree))

feat.append(skew(var\_degree))

feat.append(skew(clause\_degree))

feat.append(skew(var\_graph\_degree))

feat.append(skew(p\_ratio\_0))

feat.append(skew(p\_ratio\_1))

feat.append(skew(n\_ratio\_0))

feat.append(skew(n\_ratio\_1))

#feat.append(skew(var\_graph\_floyd\_degree))

feat.append(kurtosis(var\_degree))

feat.append(kurtosis(clause\_degree))

feat.append(kurtosis(var\_graph\_degree))

feat.append(kurtosis(p\_ratio\_0))

feat.append(kurtosis(p\_ratio\_1))

feat.append(kurtosis(n\_ratio\_0))

feat.append(kurtosis(n\_ratio\_1))

#feat.append(kurtosis(var\_graph\_floyd\_degree))

feat.append([var\_degree])

feat.append([clause\_degree])

feat.append([var\_graph\_degree])

feat.append([p\_ratio\_0])

feat.append([p\_ratio\_1])

feat.append([n\_ratio\_0])

feat.append([n\_ratio\_1])

#feat.append(var\_graph\_floyd\_degree)

All\_data\_features.append(feat)

Time\_Targets.append(Time\_Target)

pk.dump(All\_data\_features, open(os.path.join(os.getcwd(),'{}\_X.pk'.format(each\_benchmark)), 'wb'))

pk.dump(Time\_Targets, open(os.path.join(os.getcwd(),'{}\_Y.pk'.format(each\_benchmark)), 'wb'))

print("Data Generating for " + each\_benchmark + " is completed")

* **Kmeans\_elbow\_code.py:**

In order to determine the best number of clusters for IC labeling, the Kmeans\_elbow\_code.py should be run after running Data\_Generator.py.

import numpy as np

import \_pickle as pk

import os

from sklearn.cluster import KMeans

from yellowbrick.cluster import KElbowVisualizer

CircuitNames = ['74181','74182','74283','74L85','c1196','c1238','c1355','c17',

'c2670','c3540','c432','c499','c5315','c6288','c7552','c880','s1196','s1196a',

's1238','s1238a']

All\_times = []

for each\_benchmark in CircuitNames:

Times = pk.load(open(os.path.join(os.getcwd(),each\_benchmark + "\_Y.pk"), 'rb'))

All\_times.extend(Times)

Times\_array = np.asarray(All\_times)

Times\_array = np.expand\_dims(Times\_array,axis=1)

###Removing -1 values from the Times\_array

r,c = np.where(Times\_array == -1)

Times\_array = np.delete(Times\_array, r)

Times\_array = np.expand\_dims(Times\_array,axis=1)

kmeans = KMeans()

Max\_Number\_clusters = 7

elbow\_obj = KElbowVisualizer(kmeans, k=Max\_Number\_clusters,

metric='silhouette',

timings=False)

# Fit the data to the elbow technique for determining the optimum number of clusters

elbow\_obj.fit(Times\_array)

elbow\_obj.show(outpath="elbow\_result.png")

* **Kmeans\_code.py**

The third main code of this project is Kmeans\_code.py. By running this code, k-means clustering method is applied to the SAT runtimes with different number of clusters. This code is used for investing the threshold values between clusters and formulating the mapping labeling function.

import numpy as np

import \_pickle as pk

import os

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

#from scipy.io import savemat

def Clusters\_division(Times\_array,Clustering\_result,n\_clusters):

Clutsers = dict()

for x in range(0,n\_clusters):

Clutsers[x] = Times\_array[Clustering\_result==x]

return Clutsers

def plot\_clusters(Clusters,n\_clusters, val):

plt.figure(figsize=(10,5), dpi=150)

Legends = ["Cluster " + str(x) for x in range(0,n\_clusters)]

for x,y in Clusters.items():

plt.plot(y, np.zeros\_like(y) + val, 'x')

plt.ylim(-0.01, 0.01)

plt.legend(Legends)

plt.title("Scatter plot and clustering result with K="+str(n\_clusters))

plt.xlabel("SAT Runtime")

plt.savefig("K-means\_result\_n="+ str(n\_clusters) + ".jpg")

plt.show()

plt.close()

CircuitNames = ['74181','74182','74283','74L85','c1196','c1238','c1355','c17',

'c2670','c3540','c432','c499','c5315','c6288','c7552','c880','s1196','s1196a',

's1238','s1238a']

All\_times = []

for each\_benchmark in CircuitNames:

Times = pk.load(open(os.path.join(os.getcwd(),each\_benchmark + "\_Y.pk"), 'rb'))

All\_times.extend(Times)

Times\_array = np.asarray(All\_times)

Times\_array = np.expand\_dims(Times\_array,axis=1)

### Removing -1 values from the Times\_array

r,c = np.where(Times\_array == -1)

Times\_array = np.delete(Times\_array, r)

Times\_array = np.expand\_dims(Times\_array,axis=1)

Number\_of\_clusters = 5

Each\_clutsering\_result = dict()

Criterions = dict()

for n\_clusters in range(2,Number\_of\_clusters+1):

kmeans = KMeans(n\_clusters=n\_clusters, random\_state=0)

kmeans.fit(Times\_array)

Clustering\_result = kmeans.labels\_

Criterions[n\_clusters] = kmeans.inertia\_

Clusters = Clusters\_division(Times\_array,Clustering\_result,n\_clusters)

Each\_clutsering\_result[n\_clusters] = Clusters

plot\_clusters(Clusters,n\_clusters, 0)

* **SFS\_code.py**

After determine the threshold value and data generating, SFS\_code.py should be run to select the best scalar features. In this code, m2 is a variable, which determines the threshold value in seconds between SAT-resilient and SAT-vulnerable classes. Also, CircuitNames should be assigned to a circuit name that the proposed framework will be applied to it.

import numpy as np

import sklearn

from mlxtend.feature\_selection import SequentialFeatureSelector as SFS

from mlxtend.plotting import plot\_sequential\_feature\_selection as plot\_sfs

import matplotlib.pyplot as plt

from scipy.io import savemat

from util import read\_ic

def Best\_Comb(Results):

Avg\_metrics = []

for index, each\_result in Results.items():

Avg\_metrics.append(each\_result["avg\_score"])

Best\_com\_index = Avg\_metrics.index(max(Avg\_metrics))

return Results[Best\_com\_index+1]['feature\_idx'], np.asarray(Avg\_metrics)

CircuitNames = ['c1355','c2670','c3540','c5315','c6288','c7552']

CircuitNames = [CircuitNames[1]]

Number\_of\_scaler\_features = 40

m2 = 59830

\_, X, Y, \_, \_, \_ = read\_ic(CircuitNames,Number\_of\_scaler\_features,m2,False)

X = np.asarray(X)

Y = np.asarray(Y)

### Feature Selection

from sklearn.neighbors import KNeighborsClassifier

Model = KNeighborsClassifier()

sfs1 = SFS(Model,

k\_features = 40,

forward = True,

cv = 10)

sfs1.fit(X,Y)

Results = sfs1.subsets\_

SF, Metrics = Best\_Comb(Results)

fig = plot\_sfs(sfs1.get\_metric\_dict())

plt.savefig('my\_plot.png')

plt.close

Mydic = {"Selected\_Features":SF, "Metrics":Metrics}

savemat('SF.mat',Mydic)

* **Run\_iterate\_10fold.py**

In the next step, Run\_iterate\_10fold.py should be run to train and validate the CNF-NET model for classifying each IC to SAT-resilient and SAT-vulnerable classes. In this code, m2 is a variable, which determines the threshold value in seconds between SAT-resilient and SAT-vulnerable classes. It is worth mention that this code trains and validates the CNF-NET model using 10-fold cross-validation technique.

import time

import argparse

import numpy as np

import os

import torch

import torch.nn as nn

import torch.optim as optim

from operator import itemgetter

from torch.utils.data import DataLoader

import pickle as pk

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import KFold

from model import CNFNet

from util import GraphDataset, chunks, print\_network, plot\_metric, read\_ic, read\_ic\_10fold

from scipy.io import savemat, loadmat

def Evaluate\_CM(L\_test,Predict):

NTP = 0

NTN = 0

NFP = 0

NFN = 0

for i in range(len(Predict)):

if (Predict[i]==1)and(L\_test[i]==1):

NTP = NTP + 1

elif (Predict[i]==0)and(L\_test[i]==0):

NTN = NTN + 1

elif (Predict[i]==1)and(L\_test[i]==0):

NFP = NFP + 1

else:

NFN = NFN + 1

Accuracy = (NTP+NTN)/(NTP+NTN+NFP+NFN)

Sensitivity = NTP/(NTP+NFN)

try:

Specificity = NTN/(NTN+NFP)

except:

Specificity = np.nan

F1 = (2\*NTP)/((2\*NTP)+NFP+NFN)

Loss = 1 - Accuracy

return Accuracy, Sensitivity, Specificity, F1, Loss

def Prediction(output,model):

\_, predicted = torch.max(output.data, 1)

return predicted

def Obtain\_Accuracy(output,labels,model):

correct = 0

total = 0

\_, predicted = torch.max(output.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

Accuracy = correct/total

return Accuracy

parser = argparse.ArgumentParser()

parser.add\_argument('--no-cuda', action='store\_true', default=False, help='Disables CUDA training.')

parser.add\_argument('--fastmode', action='store\_true', default=False, help='Validate during training pass.')

parser.add\_argument('--seed', type=int, default=8, help='Random seed.')

parser.add\_argument('--epochs', type=int, default=1, help='Number of epochs to train.')

parser.add\_argument('--lr', type=float, default=0.01, help='Initial learning rate.')

parser.add\_argument('--weight\_decay', type=float, default=5e-4, help='Weight decay (L2 loss on parameters).')

parser.add\_argument('--hidden', type=int, default=32, help='Number of hidden units.')

parser.add\_argument('--dropout', type=float, default=0.5, help='Dropout rate (1 - keep probability).')

parser.add\_argument('--data', type=str, default='c499', help='Dataset name')

#CircuitNames = ['74181','74182','74283','74L85','c1196','c1238','c1355','c17',

# 'c2670','c3540','c432','c499','c5315','c6288','c7552','c880','s1196','s1196a',

# 's1238','s1238a']

CircuitNames = ['c1355','c2670','c3540','c5315','c6288','c7552']

CircuitNames = [CircuitNames[1]]

args = parser.parse\_args()

args.cuda = True

# Loading Selected Features indexes

Dict\_SF = loadmat("SF.mat")

SF = Dict\_SF["Selected\_Features"]

# Training setting

args.epochs = 25

args.num\_feat = SF.size

args.batch\_size = 32

# FC setting

args.energy\_input\_dim = SF.size + 7

Number\_of\_scaler\_features = 40

#m1 = 16780

m2 = 59830

Num\_iter = 1

All\_test\_acc = []

All\_test\_loss = []

All\_test\_predic = []

All\_test\_label = []

inc\_feat, feats, times = read\_ic\_10fold(CircuitNames,Number\_of\_scaler\_features,

m2,True,SF)

kf = KFold(n\_splits=2)

kf.get\_n\_splits(feats,times)

ACs = []

SEs = []

SPs = []

F1s = []

NPs = []

NNs = []

i=0

for train\_index, test\_index in kf.split(feats,times):

i = i + 1

inc\_feat\_tr, inc\_feat\_te, feat\_tr, feat\_te, labels\_tr, labels\_te = train\_test\_split([inc\_feat[p] for p in train\_index], feats[train\_index,:],

times[train\_index], test\_size=0.1)

model = CNFNet(args)

optimizer = optim.Adam(model.parameters(), lr=args.lr, weight\_decay=args.weight\_decay)

print\_network(model)

#cri = nn.CrossEntropyLoss(weight=torch.Tensor([float((times==1).sum()/(times==0).sum()) ,1]))

cri = nn.CrossEntropyLoss()

Tr\_incides = np.arange(len(inc\_feat\_tr))

Tr\_incides = torch.LongTensor(Tr\_incides)

test\_index = torch.LongTensor(test\_index)

graph\_loader = DataLoader(GraphDataset(Tr\_incides[:int(len(Tr\_incides) / args.batch\_size) \* args.batch\_size]),

batch\_size=args.batch\_size, shuffle=True)

train\_loss = []

eval\_loss = []

train\_acc = []

val\_acc = []

os.environ['CUDA\_VISIBLE\_DEVICES'] = '0'

torch.backends.cudnn.benchmark = True

# Training Model

for epoch in range(args.epochs):

for step, ids in enumerate(graph\_loader):

t = time.time()

model.train()

output = model(itemgetter(\*ids)(inc\_feat\_tr), feat\_tr[ids])

loss\_train = cri(output, labels\_tr[ids])

optimizer.zero\_grad()

loss\_train.backward()

optimizer.step()

print('Epoch: {:02d}/{:04d}'.format(epoch, step + 1),

'loss\_train: {:.4f}'.format(loss\_train.item()),

'time: {:.4f}s'.format(time.time() - t), end='\n')

if step % 10 == 0:

model.eval()

#val\_ids = list(chunks(val\_num, args.batch\_size))[:-1]

output\_eval = [model(inc\_feat\_te, feat\_te)]

#output\_eval = [model(itemgetter(\*\_)(inc\_feat), feats[\_]) for \_ in val\_ids]

loss\_val = np.mean([cri(output\_eval[0][\_], labels\_te[\_]).item() for \_ in range(len(labels\_te))])

#loss\_val = np.mean([cri(output\_eval[\_], times[val\_ids[\_]]).item() for \_ in range(len(val\_ids))])

print("Eval loss: {}".format(loss\_val), end='\n')

train\_acc.append(Obtain\_Accuracy(output,labels\_tr[ids],model))

val\_acc.append(Obtain\_Accuracy(output\_eval[0],labels\_te,model))

train\_loss.append(loss\_train.item())

eval\_loss.append(loss\_val.item())

# print training info

plot\_metric(range(len(train\_loss)), train\_loss, eval\_loss, '{}\_train\_{}'.format('loss',i),

'{}\_eval\_{}'.format('loss',i))

plot\_metric(range(len(train\_acc)), train\_acc, val\_acc, '{}\_train\_{}'.format('Acc',i),

'{}\_eval\_{}'.format('Acc',i))

print("Optimization Finished!")

model.eval()

test\_ids = list(chunks(test\_index, args.batch\_size))

output\_test = []

output\_test = [model(itemgetter(\*\_)(inc\_feat), feats[\_]) for \_ in test\_ids]

loss\_val = np.mean([cri(output\_test[\_], times[test\_ids[\_]]).item() for \_ in range(len(test\_ids))])

print("Test loss: {}".format(loss\_val))

test\_acc = Obtain\_Accuracy(output\_test[0],times[test\_ids[0]],model)

test\_pred = Prediction(output\_test[0],model)

NPs.append(np.sum(np.asarray(times[test\_ids[0]].tolist())==1))

NNs.append(np.sum(np.asarray(times[test\_ids[0]].tolist())==0))

AC, SE, SP, F1, \_ = Evaluate\_CM(test\_pred.tolist(),times[test\_ids[0]].tolist())

ACs.append(AC)

SEs.append(SE)

SPs.append(SP)

F1s.append(F1)

print(test\_acc)

All\_test\_acc.append(test\_acc)

All\_test\_loss.append(loss\_val)

All\_test\_acc = np.asarray(All\_test\_acc)

Mean\_All\_test\_acc = np.mean(All\_test\_acc)

All\_test\_loss = np.asarray(All\_test\_loss)

Mean\_All\_test\_loss = np.mean(All\_test\_loss)

Mydic = {"All\_test\_acc" : All\_test\_acc, "Mean\_All\_test\_acc" : Mean\_All\_test\_acc,

"Mean\_All\_test\_loss":Mean\_All\_test\_loss, "All\_test\_loss" : All\_test\_loss,'ACs':np.asarray(ACs),

'SEs':np.asarray(SEs), 'SPs':np.asarray(SPs), 'F1s':np.asarray(F1s)}

savemat('results.mat',Mydic)

1. **Modules and libraries**

* **bench2cnf.py**

This module is used for obtaining CNF representation for the bench file of an IC.

import argparse

import itertools

import numpy as np

import copy

class wire:

def \_\_init\_\_(self, name, type, operands, logic\_value, logic\_level, prob0, prob1, absprob,

fanout, mainout, tag, index):

self.name = name

self.type = type

self.operands = operands

self.logic\_value = logic\_value

self.logic\_level = logic\_level

self.prob0 = prob0

self.prob1 = prob1

self.absprob = absprob

self.fanout = fanout

self.mainout = mainout

self.tag = tag

self.index = index

def findsubsets(S, m):

return set(itertools.combinations(S, m))

def sub\_lists(list1):

# store all the sublists

sublist = [[]]

# first loop

for i in range(len(list1) + 1):

# second loop

for j in range(i + 1, len(list1) + 1):

# slice the subarray

sub = list1[i:j]

sublist.append(sub)

return sublist

def wire\_print(wire\_in):

print("w\_name: ", wire\_in.name)

print("w\_type: ", wire\_in.type)

for i in range(0, len(wire\_in.operands)):

print("w\_opr", i, ": ", wire\_in.operands[i].name)

print("w\_valu: ", wire\_in.logic\_value)

print("w\_glev: ", wire\_in.logic\_level)

print("w\_prob0:", wire\_in.prob0)

print("w\_prob1:", wire\_in.prob1)

print("w\_abs\_prob:", wire\_in.absprob)

print("w\_fanout:", wire\_in.fanout)

print("w\_mainout:", wire\_in.mainout)

print("index:", wire\_in.index, "\n")

def wire\_fanin\_cone(wire\_in, cone\_size):

index\_traverse = 0

if wire\_in.type == "inp":

return []

else:

fanin\_cone = [wire\_in]

cone\_size = cone\_size - 1

temp = copy.deepcopy(fanin\_cone[index\_traverse].operands)

while cone\_size != 0:

added\_bef = 0

if len(temp) == 0:

index\_traverse = index\_traverse + 1

if index\_traverse >= len(fanin\_cone):

cone\_size = 0

else:

temp = copy.deepcopy(fanin\_cone[index\_traverse].operands)

elif temp[0].type == "inp":

temp.remove(temp[0])

else:

for i in range(0, len(fanin\_cone)):

if fanin\_cone[i].name == temp[0].name:

added\_bef = 1

if added\_bef == 0:

fanin\_cone.append(temp[0])

cone\_size = cone\_size - 1

temp.remove(temp[0])

return fanin\_cone

def get\_unique\_fanin\_cone(wire\_in):

return uniquify\_wire\_list(get\_fanin\_cone(wire\_in))

def get\_fanin\_cone(wire\_in):

if wire\_in.type == "inp":

return []

else:

fanin\_cone = []

fanin\_cone.append(wire\_in)

for i in range(len(wire\_in.operands)):

fanin\_cone += get\_fanin\_cone(wire\_in.operands[i])

return fanin\_cone

def get\_fanin\_cone2(wire\_in, fanin\_cone):

if wire\_in.type != "inp":

fanin\_cone.add(wire\_in)

for i in range(len(wire\_in.operands)):

get\_fanin\_cone2(wire\_in.operands[i], fanin\_cone)

def uniquify\_wire\_list(wire\_list):

seen = set()

unique = []

for obj in wire\_list:

if obj.name not in seen:

unique.append(obj)

seen.add(obj.name)

return unique

def wire\_dep(benchmark\_address, args):

inputs = []

outputs = []

wires = []

input\_array = []

temp = []

bench\_file = open(benchmark\_address)

index = 1

for line in bench\_file:

if "INPUT" in line:

inputs.append(line[line.find("(") + 1:line.find(")")])

input\_array.append(line[line.find("(") + 1:line.find(")")])

wires.append(wire(line[line.find("(") + 1:line.find(")")], "inp", [], "1", 0, 0.5, 0.5, 0, 0, 0, 0, index))

input\_array = []

index += 1

elif "OUTPUT" in line:

out\_name = line[line.find("(") + 1:line.find(")")]

outputs.append(out\_name)

elif " = " in line:

gate\_out = line[0: line.find(" =")]

gate\_type = line[line.find("= ") + 2: line.find("(")]

gate\_list\_inputs = line[line.find("(") + 1:line.find(")")]

gate\_oprs = gate\_list\_inputs.split(", ")

for i in range(0, len(gate\_oprs)):

found = False

for j in range(0, len(wires)):

if wires[j].name == gate\_oprs[i]:

found = True

temp.append(wires[j])

break

if not found:

print("ERROR")

print(gate\_out, gate\_oprs[i])

exit()

max\_level = 0

for i in range(0, len(temp)):

if temp[i].logic\_level > max\_level:

max\_level = temp[i].logic\_level

temp\_prob0 = 0.25

temp\_prob1 = 0.25

for i in range(0, len(temp)):

if len(temp) == 1:

if gate\_type == "NOT" or gate\_type == "not":

temp\_prob = temp[0].prob0

temp\_prob0 = temp[0].prob1

temp\_prob1 = temp\_prob

elif gate\_type == "BUFF" or gate\_type == "buff":

temp\_prob0 = temp[0].prob0

temp\_prob1 = temp[0].prob1

else:

if gate\_type == "NAND" or gate\_type == "nand":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0 + temp\_prob1 \* temp[i].prob0 + temp\_prob0 \* temp[

i].prob1

temp\_prob1 = temp\_prob1 \* temp[i].prob1

elif gate\_type == "AND" or gate\_type == "and":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0 + temp\_prob1 \* temp[i].prob0 + temp\_prob0 \* temp[

i].prob1

temp\_prob1 = temp\_prob1 \* temp[i].prob1

elif gate\_type == "NOR" or gate\_type == "nor":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0

temp\_prob1 = 1 - temp\_prob0

elif gate\_type == "OR" or gate\_type == "or":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0

temp\_prob1 = 1 - temp\_prob0

elif gate\_type == "XNOR" or gate\_type == "xnor":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0 + temp\_prob1 \* temp[i].prob1

temp\_prob1 = 1 - temp\_prob0

elif gate\_type == "XOR" or gate\_type == "xor":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob1 + temp\_prob1 \* temp[i].prob0

temp\_prob1 = 1 - temp\_prob0

if gate\_type == "NAND" or gate\_type == "nand":

temp\_prob = temp\_prob0

temp\_prob0 = temp\_prob1

temp\_prob1 = temp\_prob

if gate\_type == "NOR" or gate\_type == "nor":

temp\_prob = temp\_prob0

temp\_prob0 = temp\_prob1

temp\_prob1 = temp\_prob

wires.append(wire(gate\_out, gate\_type, temp, "1", max\_level + 1, temp\_prob0, temp\_prob1,

abs(temp\_prob0 - temp\_prob1), 0, 0, 0, index))

temp = []

index += 1

bench\_file.close()

# just check for correctness

for i in range(0, len(wires)):

for j in range(i + 1, len(wires)):

if wires[i].name == wires[j].name:

print(wires[i].name, wires[j].name)

print("ERROR in reading wires")

exit()

for i in range(0, len(wires)):

fanout\_temp = 0

for j in range(0, len(wires)):

if i != j:

for k in range(0, len(wires[j].operands)):

if wires[i].name == wires[j].operands[k].name:

fanout\_temp = fanout\_temp + 1

wires[i].fanout = fanout\_temp

return wires

def simple\_read\_bench(benchmark\_address):

wires = []

temp = []

bench\_file = open(benchmark\_address)

index = 1

for line in bench\_file:

if "#" in line:

continue

elif "INPUT" in line:

wires.append(wire(line[line.find("(") + 1:line.find(")")], "inp", [], "1", 0, 0, 0, 0, 0, 0, 0, index))

elif "OUTPUT" in line:

wires.append(wire(line[line.find("(") + 1:line.find(")")], "out", [], "1", 0, 0, 0, 0, 0, 0, 0, index))

elif " = " in line:

gate\_out = line[0: line.find(" =")]

gate\_type = line[line.find("= ") + 2: line.find("(")].lower()

gate\_list\_inputs = line[line.find("(") + 1:line.find(")")]

gate\_oprs = gate\_list\_inputs.split(",")

gate\_oprs = [x.strip(' ') for x in gate\_oprs]

for i in range(0, len(gate\_oprs)):

found = False

for j in range(0, len(wires)):

if wires[j].name == gate\_oprs[i]:

found = True

temp.append(wires[j])

break

if not found:

# print gate\_out, gate\_oprs[i]

temp.append(wire(gate\_oprs[i], "dummy", [], "1", 0, 0, 0, 0, 0, 0, 0, 0))

wires.append(wire(gate\_out, gate\_type, temp, "1", 0, 0, 0, 0, 0, 0, 0, index))

else:

continue

temp = []

index += 1

bench\_file.close()

for i in range(0, len(wires)):

for j in range(0, len(wires[i].operands)):

if wires[i].operands[j].type == "dummy":

found = False

for k in range(0, len(wires)):

if wires[k].name == wires[i].operands[j].name:

found = True

wires[i].operands[j] = wires[k]

break

if not found:

print(wires[i].operands[j].name)

print("ERROR1 in read\_circuit()")

exit()

# just to be sure!

for i in range(len(wires)):

if wires[i].name != wires[wires[i].index - 1].name:

print(wires[i].name)

print("ERROR2 in read\_circuit()")

exit()

return wires

def tseytin\_t(wires\_in):

cnf\_clause\_count = 0

cnf\_clause\_list = ""

for i in range(len(wires\_in)):

if wires\_in[i].type == "NOT" or wires\_in[i].type == "not":

cnf\_clause\_list += str(wires\_in[i].index) + " " + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_count += 2

elif wires\_in[i].type == "BUFF" or wires\_in[i].type == "buff":

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_count += 2

elif wires\_in[i].type == "AND" or wires\_in[i].type == "and":

cnf\_clause\_list += str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " -" + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " " + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "NAND" or wires\_in[i].type == "nand":

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " -" + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += str(wires\_in[i].index) + " " + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "OR" or wires\_in[i].type == "or":

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " " + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "NOR" or wires\_in[i].type == "nor":

cnf\_clause\_list += str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " " + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "XOR" or wires\_in[i].type == "xor":

operand\_subsets = sub\_lists(wires\_in[i].operands)

for j in range(0, len(operand\_subsets)):

if len(operand\_subsets[j]) % 2 == 0:

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif len(operand\_subsets[j]) % 2 == 1 and len(operand\_subsets[j]) != 0:

cnf\_clause\_list += str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif wires\_in[i].type == "XNOR" or wires\_in[i].type == "xnor":

operand\_subsets = sub\_lists(wires\_in[i].operands)

for j in range(0, len(operand\_subsets)):

if len(operand\_subsets[j]) % 2 == 1 and len(operand\_subsets[j]) != 0:

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif len(operand\_subsets[j]) % 2 == 0 and len(operand\_subsets[j]) != 0:

cnf\_clause\_list += str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

cnf\_clause\_list += str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " " + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif wires\_in[i].type == "MUX" or wires\_in[i].type == "mux":

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[1].index) + " " + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[2].index) + " -" + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " " + str(wires\_in[i].operands[1].index) + " " + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " " + str(wires\_in[i].operands[2].index) + " -" + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_count += 4

return cnf\_clause\_count, cnf\_clause\_list

def cnf\_to\_matrix(cnf\_str):

"""Convert cnf file to bipartite graph matrix (num\_clauses,num\_clauses)

Args:

fpath: The path to the cnf file.

The first two lines of the file are notes.

The clauses start from the 3rd line.

Returns:

A numpy 2-dimensional array.

The rows are for clauses.

The columns are for literals.

example:

array([[ 0, 0, 0, 0, -1, 1],

[ 0, 0, 0, 1, 0, -1],

[ 0, 0, 0, 0, 1, -1]])

The example denotes a cnf file with 3 clauses and 6 parameters.

0 denotes that the literal in this column didn't show up in the

cluase of this row.

1 denotes the positive literal exists in this clause.

-1 denotes the negative literal exists in this clause.

"""

# lines = open(fpath).readlines()[2:]

lines = cnf\_str.split("\n")

literals = {}

clauses = []

for l in lines:

for ele in l.strip().split(" ")[:-1]:

literals[abs(int(ele))] = 0

num\_literals = len(literals)

num\_clauses = len(lines)

literals\_list = list(literals)

l\_to\_pos = {}

pos\_init = 0

for l in literals\_list:

l\_to\_pos[l] = pos\_init

pos\_init += 1

bi\_g = np.zeros((num\_clauses, num\_literals), dtype=np.int)

for i in range(num\_clauses):

for ele in lines[i].strip().split(" ")[:-1]:

literal = int(ele)

pos = l\_to\_pos[abs(literal)]

if literal > 0:

bi\_g[i, pos] = 1

else:

bi\_g[i, pos] = -1

return bi\_g

if \_\_name\_\_ == "\_\_main\_\_":

parser = argparse.ArgumentParser(description='ISCAS-85 bench to CNF Converter v1.0 (c) Hadi Mardani Kamali.')

parser.add\_argument("-b", action="store", required=True, type=str, help="benchmark path")

parser.add\_argument("-c", action="store", required=True, type=str, help="cnf path")

args = parser.parse\_args()

bench\_address = args.b

cnf\_address = args.c

wires = simple\_read\_bench(bench\_address)

cnf\_variable\_counts = len(wires)

cnf\_clause\_count, cnf\_content = tseytin\_t(wires)

cnf\_file\_content = "Circuit-CNF generated by BENCH2CNFv1.0 (c) Hadi Mardani Kamali\n"

cnf\_file\_content += "p cnf " + str(cnf\_variable\_counts) + " " + str(cnf\_clause\_count) + "\n" + cnf\_content

cnf\_file = open(cnf\_address, 'w')

cnf\_file.write(cnf\_file\_content)

cnf\_file.close()

print("cnf has been successfully written in", cnf\_address)

* **util.py**

This module has different function, which the most important of them are cnf\_to\_matrix, read\_ic, and read\_ic\_10fold. The cnf\_to\_matrix computes the matrix of CNF representation. The read\_ic is used to load the features and SAT-runtimes of each IC from the saved pickle files. The read\_ic\_10fold perform the same work as the read\_ic function based on 10-fold cross-validation technique.

import \_pickle as pk

import torch

from pylab import \*

from torch.utils.data import Dataset

def cnf\_to\_matrix(cnf\_str):

"""Convert cnf file to bipartite graph matrix (num\_clauses,num\_clauses)

Args:

fpath: The path to the cnf file.

The first two lines of the file are notes.

The clauses start from the 3rd line.

Returns:

A numpy 2-dimensional array.

The rows are for clauses.

The columns are for literals.

example:

array([[ 0, 0, 0, 0, -1, 1],

[ 0, 0, 0, 1, 0, -1],

[ 0, 0, 0, 0, 1, -1]])

The example denotes a cnf file with 3 clauses and 6 parameters.

0 denotes that the literal in this column didn't show up in the

cluase of this row.

1 denotes the positive literal exists in this clause.

-1 denotes the negative literal exists in this clause.

"""

# lines = open(fpath).readlines()[2:]

lines = cnf\_str.split("\n")

literals = {}

clauses = []

for l in lines:

for ele in l.strip().split(" ")[:-1]:

literals[abs(int(ele))] = 0

num\_literals = len(literals)

num\_clauses = len(lines)

literals\_list = list(literals)

l\_to\_pos = {}

pos\_init = 0

for l in literals\_list:

l\_to\_pos[l] = pos\_init

pos\_init += 1

bi\_g = np.zeros((num\_clauses, num\_literals), dtype=np.int)

for i in range(num\_clauses):

for ele in lines[i].strip().split(" ")[:-1]:

literal = int(ele)

pos = l\_to\_pos[abs(literal)]

if literal > 0:

bi\_g[i, pos] = 1

else:

bi\_g[i, pos] = -1

return bi\_g

def read\_ic(CircuitNames,q,m2,

Torch\_transform=True,

SF=[]):

"""read preprocessed data files"""

""" q is the start index for vector features """

""" m1 and m2 are the thresholds """

def Map\_labels(t):

if 0<=t<=m2:

return 1 #Label 1 refers to SAT-vulnerable samples

else:

return 0 #Label 0 refers to SAT-resilient samples

root\_dir = '{}\_{}.pk'

#times = [pk.load(open(root\_dir.format(c, 'Y'), 'rb')) for c in CircuitNames]

times = []

input = []

for c in CircuitNames:

T = pk.load(open(root\_dir.format(c, 'Y'), 'rb'))

times.extend(T)

X = pk.load(open(root\_dir.format(c, 'X'), 'rb'))

input.extend(X)

times = list(map(Map\_labels,times))

times = np.array(times)

input = np.array(input)

inc\_feat = [\_[q:] for \_ in input]

feat\_list = [\_[:q] for \_ in input]

#feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

if SF==[]:

feat = feat\_list

else:

feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

# feat = np.asarray(feat)

# feat = feat[:,SF]

# feat = feat.squeeze(axis=1)

print('Data size: {}'.format(times.size))

train\_rate = 0.8 # 80% of data is divided into the training set

val\_rate = 0.9 # 10% of data is divided into the validation set

test\_rate = 1 # 10% of data is divided into the testing set

DATA\_NUM = len(times)

data\_ind = np.arange(DATA\_NUM)

np.random.seed(8)

np.random.shuffle(data\_ind)

train\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate))])

val\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate), int(DATA\_NUM \* val\_rate))])

test\_num = sorted(data\_ind[range(int(DATA\_NUM \* val\_rate), int(DATA\_NUM \* test\_rate))])

if Torch\_transform == True:

train\_num = torch.LongTensor(train\_num)

val\_num = torch.LongTensor(val\_num)

test\_num = torch.LongTensor(test\_num)

#times = torch.FloatTensor(np.log1p(times))

times = torch.LongTensor(times)

feat = torch.FloatTensor(feat)

else:

pass

return inc\_feat, feat, times, train\_num, val\_num, test\_num

def read\_ic\_10fold(CircuitNames,q,m2,

Torch\_transform=True,

SF=[]):

"""read preprocessed data files"""

""" q is the start index for vector features """

""" m1 and m2 are the thresholds """

def Map\_labels(t):

if 0<=t<=m2:

return 1 #Label 1 refers to SAT-vulnerable samples

else:

return 0 #Label 0 refers to SAT-resilient samples

root\_dir = '{}\_{}.pk'

#times = [pk.load(open(root\_dir.format(c, 'Y'), 'rb')) for c in CircuitNames]

times = []

input = []

for c in CircuitNames:

T = pk.load(open(root\_dir.format(c, 'Y'), 'rb'))

times.extend(T)

X = pk.load(open(root\_dir.format(c, 'X'), 'rb'))

input.extend(X)

times = list(map(Map\_labels,times))

times = np.array(times)

input = np.array(input)

inc\_feat = [\_[q:] for \_ in input]

feat\_list = [\_[:q] for \_ in input]

#feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

if SF==[]:

feat = feat\_list

else:

feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

# feat = np.asarray(feat)

# feat = feat[:,SF]

# feat = feat.squeeze(axis=1)

# print('Data size: {}'.format(times.size))

# train\_rate = 0.8 # 80% of data is divided into the training set

# val\_rate = 0.9 # 10% of data is divided into the validation set

# test\_rate = 1 # 10% of data is divided into the testing set

# DATA\_NUM = len(times)

# data\_ind = np.arange(DATA\_NUM)

# np.random.seed(8)

# np.random.shuffle(data\_ind)

# train\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate))])

# val\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate), int(DATA\_NUM \* val\_rate))])

# test\_num = sorted(data\_ind[range(int(DATA\_NUM \* val\_rate), int(DATA\_NUM \* test\_rate))])

if Torch\_transform == True:

#train\_num = torch.LongTensor(train\_num)

#val\_num = torch.LongTensor(val\_num)

#test\_num = torch.LongTensor(test\_num)

#times = torch.FloatTensor(np.log1p(times))

times = torch.LongTensor(times)

feat = torch.FloatTensor(feat)

else:

pass

return inc\_feat, feat, times,

def print\_network(net):

"""print brief structure of neural networks"""

num\_params = 0

for param in net.parameters():

num\_params += param.numel()

print(net)

print('Total number of parameters: %d' % num\_params)

def plot\_metric(x, y, z, yc='train', zc='eval'):

"""plot result image and save in local files"""

matplotlib.style.use('seaborn')

plt.figure(figsize=(12, 6))

plt.xticks(fontsize=20)

plt.yticks(fontsize=20)

# plt.scatter(x, y)

# plot(x, y, markerfacecolor='salmon', markeredgewidth=1, markevery=slice(40, len(y), 70),

# linestyle=':', marker='o', color='crimson', linewidth=3, label='fit') # fit result

plot(x, y, color='crimson', linewidth=5, label=yc) # fit result

# plt.scatter(x, z)

# plot(x, y, markerfacecolor='salmon', markeredgewidth=1, markevery=slice(40, len(y), 70),

# linestyle=':', marker='o', color='crimson', linewidth=3, label='fit') # fit result

plot(x, z, color='blue', linewidth=5, label=zc) # fit result

plt.legend(loc="best", prop={'size': 20})

# plt.savefig('loss.png')

plt.savefig('{}-{}.png'.format(yc, zc))

plt.close()

class GraphDataset(Dataset):

"""wrap function for sampling training instances"""

def \_\_init\_\_(self, ids):

self.ids = ids

def \_\_getitem\_\_(self, index):

return self.ids[index]

def \_\_len\_\_(self):

return len(self.ids)

def chunks(l, n):

# For item i in a range that is a length of l,

for i in range(0, len(l), n):

# Create an index range for l of n items:

yield l[i:i + n]

def store\_model(model, name='test'):

torch.save(model, '{}'.format(name))

def restore\_model(path):

model = torch.load(path)

model.eval()

return model

* **model.py**

This module defines the main structure of CNF-NET model.

import torch

import torch.nn as nn

from layer import FC, IncidenceConvolution

class CNFNet(nn.Module):

def \_\_init\_\_(self, opt):

super(CNFNet, self).\_\_init\_\_()

self.opt = opt

# initialize CNFNet

self.energy\_kernel = IncidenceConvolution(opt)

self.fc = FC(opt)

def forward(self, inc, f):

# calculate energy kernel

y = self.energy\_kernel(inc)

# concatenate with CNF properties

z = torch.cat((f, y), 1)

# connect to fully-connected layers

z = self.fc(z)

return z

* **layer.py**

The layers and operations of the CNF-NET parts is defined in layer.py

import torch

import torch.nn as nn

import torch.sparse

from torch.nn.modules.module import Module

class IncidenceConvolution(Module):

"""Core operation of CNFNet"""

def \_\_init\_\_(self, opt):

super(IncidenceConvolution, self).\_\_init\_\_()

self.opt = opt

self.in\_features = 1

self.out\_features = opt.num\_feat

self.fc\_dim = opt.hidden

self.func1 = nn.Linear(1, self.fc\_dim)

self.func2 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func3 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func4 = nn.Linear(self.fc\_dim, 1)

def forward(self, inc\_m):

"""iterate instances and call energy function"""

out = []

for instance in inc\_m: # itearate on each matrix data

feat = []

for arr in instance:

feat.append(self.\_to\_kernel(arr))

out.append(torch.FloatTensor(feat))

return torch.stack(out)

def \_fc\_kernal(self, sig):

"""calculate the prediction given energy"""

return self.func4(self.func3(self.func2(self.func1(sig))))

def \_to\_kernel(self, arr):

"""calculate the energy representation of distribution"""

# arr[0] is a vector

l = torch.FloatTensor(arr[0])

# normalization

norm = l / torch.sum(l)

# FC and product

weight = torch.tensor([self.\_fc\_kernal(\_.unsqueeze(0)) for \_ in norm])

prod = torch.mul(weight, norm)

# sum

return torch.sum(prod)

def \_\_repr\_\_(self):

return self.\_\_class\_\_.\_\_name\_\_ + ' (' \

+ str(self.in\_features) + ' -> ' \

+ str(self.out\_features) + ')'

class FC(Module):

"""Fully connected layer for predicting runtime"""

def \_\_init\_\_(self, opt):

super(FC, self).\_\_init\_\_()

self.opt = opt

self.in\_features = opt.energy\_input\_dim

self.out\_features = 2

self.fc\_dim = opt.hidden

self.func1 = nn.Linear(self.in\_features, self.fc\_dim)

self.func2 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func3 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func4 = nn.Linear(self.fc\_dim, self.out\_features)

def forward(self, val):

return self.func4(self.func3(self.func2(self.func1(val))))

def \_\_repr\_\_(self):

return self.\_\_class\_\_.\_\_name\_\_ + ' (' \

+ str(self.in\_features) + ' -> ' \

+ str(self.out\_features) + ')'

# References

[1] P. Subramanyan, S. Ray, and S. Malik, "Evaluating the security of logic encryption algorithms," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2015, pp. 137-143: IEEE.

[2] K. Rosenfeld, "Trustworthy Hardware: Identifying and Classifying Hardware Trojans," 2010.

[3] U. Guin, D. Forte, and M. Tehranipoor, "Anti-counterfeit techniques: From design to resign," in *2013 14th International workshop on microprocessor test and verification*, 2013, pp. 89-94: IEEE.

[4] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE,* vol. 102, no. 8, pp. 1283-1295, 2014.

[5] M. Yasin, A. Sengupta, M. T. Nabeel, M. Ashraf, J. Rajendran, and O. Sinanoglu, "Provably-secure logic locking: From theory to practice," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1601-1618.

[6] M. El Massad, S. Garg, and M. V. Tripunitara, "Integrated Circuit (IC) Decamouflaging: Reverse Engineering Camouflaged ICs within Minutes," in *NDSS*, 2015, pp. 1-14.

[7] D. Liu, C. Yu, X. Zhang, and D. Holcomb, "Oracle-guided incremental SAT solving to reverse engineer camouflaged logic circuits," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 433-438: IEEE.

[8] S. Khaleghi and W. Rao, "Hardware obfuscation using strong pufs," in *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2018, pp. 321-326: IEEE.

[9] S. Wang, G. Ananthanarayanan, Y. Zeng, N. Goel, A. Pathania, and T. Mitra, "High-throughput CNN inference on embedded ARM Big. LITTLE multicore processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 39, no. 10, pp. 2254-2267, 2019.

[10] M. Lechner, A. Jantsch, and S. M. P. Dinakarrao, "ResCoNN: Resource-efficient FPGA-accelerated CNN for traffic sign classification," in *2019 Tenth International Green and Sustainable Computing Conference (IGSC)*, 2019, pp. 1-6: IEEE.