Dynamic Ensemble Selection for Network Intrusion Detection on FPGA Using Dynamic Partial Reconfiguration

Sean Berrios
Arizona State University
sfberrio@asu.edu

Ruthwik Reddy Sunketa
Arizona State University
rsunketa@asu.edu

Deepesh Sahoo
Arizona State University
dsahoo4@asu.edu

***Abstract*—This work explores an ensemble-based Dynamic Partial Reconfiguration (DPR) approach for Network Intrusion Detection Systems (NIDS) on FPGA platforms. Our design uses an ensemble of 3 classifiers selected from Gaussian Naive Bayes, Gradient Boost, Multi-Layer Perceptron (MLP), and Logistic Regression. Using DPR, we dynamically reconfigure the ensemble during run time to optimize power management and resource utilization and compare this to GPU implementations. By leveraging the inherent parallelism of FPGAs and DPR, we aim to demonstrate lower power consumption and efficient resource usage while maintaining performance.**

I. Introduction

Network Intrusion Detection Systems (NIDS) play a crucial role in maintaining cybersecurity by detecting malicious activities, such as Distributed Denial-of-Service (DDoS) attacks and unauthorized access. As threats become increasingly complex, traditional single-classifier methods often struggle to effectively address evolving attack patterns and varied datasets.

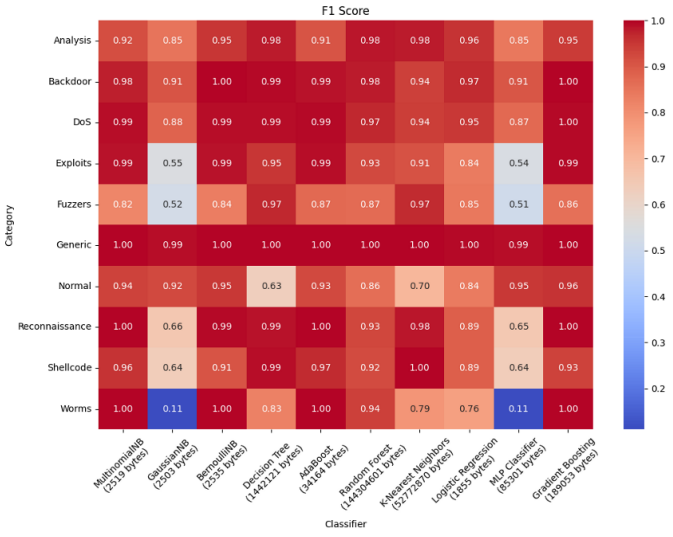

Fig. 1. Classifier performance (F1 Score) across different attack categories.

Given the diverse nature of network attacks, no single classifier consistently performs best across all attack categories. For example, In 1 within the 'Fuzzers' category, classifiers like Decision Trees and k-NN achieve relatively higher F1 scores, while these classifiers are less effective for categories such as 'Worms' or 'Normal.' In the 'Backdoor' category, Bernoulli Naive Bayes (BernoulliNB) and Gradient Boosting perform equally well in F1 scores. However, BernoulliNB has a smaller memory footprint and provides better latency, making it more resource-efficient. This variation in classifier performance across different attack categories highlights the necessity for dynamic, runtime switching of classifiers. A static ensemble approach may not adapt well to these diverse data characteristics. In contrast, dynamic ensemble selection ensures that the optimal classifier or combination of classifiers is employed based on the specific input data, maximizing accuracy while minimizing resource consumption.

In this project, we implement a dynamic ensemble selection system for NIDS on FPGA using Dynamic Partial Reconfiguration (DPR). The proposed approach allows for the dynamic reconfiguration of ensembles of classifiers, including Gaussian Naive Bayes, Gradient Boosting, Multi-Layer Perceptron (MLP), and Logistic Regression, during runtime. This capability enables the system to optimize resource utilization and power consumption without compromising detection performance. By taking advantage of the inherent parallelism of FPGAs and DPR, our methodology aims to deliver an efficient, scalable, high-performance solution for real-time network intrusion detection.

This study improves resource and power efficiency compared to GPU implementation through dynamic ensemble selection and majority voting for final classification. These findings emphasize the potential of adaptive ensemble techniques in Network Intrusion Detection systems, especially in edge applications.

II. Background

A. Network Intrusion Detection Systems

A Network Intrusion Detection System (NIDS) protects network infrastructure by identifying malicious activities such as Distributed Denial-of-Service (DDoS) attacks, reconnaissance, and worms. Recent advancements have demonstrated the efficacy of machine learning (ML) and deep learning (DL) techniques in enhancing NIDS performance, allowing these systems to detect attack patterns and adapt to evolving threats.

For example, models like Random Forest and Long Short-Term Memory (LSTM) networks have shown exceptional detection rates, particularly in categories such as DDoS attacks and exploits [1]. However, challenges persist in detecting low-frequency attack classes like Fuzzers and Worms, indicating the need for further research in hybrid and ensemble models. High-impact works, such as leveraging deep learning architectures for intrusion detection [2], and the exploration of adversarial examples to improve classifier robustness [3], have laid the foundation for modern NIDS frameworks. These systems address challenges like handling encrypted traffic, minimizing false positives, and enhancing detection across diverse attack categories, making them indispensable for robust cybersecurity infrastructure.

B. Multiple Classifier Systems (MCS)

Several studies have shown that using multiple classifiers instead of a single classifier provides superior accuracy for multiple applications [4], [5]. In [3], authors evaluated the effectiveness of multiple Dynamic Ensemble selection techniques compared to K-NN on 30 different datasets. They showed that the majority of DS techniques achieve significant performance improvement over K-NN for every classification problem evaluated. [6] have demonstrated that the Ensemble of classifiers outperforms any single ML technique for Network Intrusion Detection Systems. add some statistics here

C. Ensemble Selection Techniques

In this section, we review relevant dynamic ensemble selection selection techniques.

1) DES-clustering (DES-KMEANS): In DES-Clustering, algorithms like K-Means divide the entire train data into clusters, and an ensemble of classifiers is trained for each cluster. For each test data sample, a cluster is first selected based on a similarity score like Euclidean distance, and the ensemble associated with the selected cluster is chosen for inference. This approach is based on the assumption that classifier performance varies across different regions of feature space and that the effectiveness of the clustering process significantly impacts the performance of DES-Clustering

2) DES-KNN: In DES-KNN, a set of classifiers with the highest competence in a neighborhood are chosen for the ensemble. Competence scores of all the classifiers on train data can be precomputed, and for each test sample, K nearest neighbors in feature space will be identified. The classifiers with the highest competence score in this neighborhood will form an ensemble for inference. This approach is based on the assumption that the local neighborhood in a feature space gives meaningful information about the performance of classifiers. Choice of K value and the distance metric significantly impacts performance of DES-KNN.

3) META-DES: In Meta-DES, Instead of using static rules like accuracy for competence measurement, a meta-learning process is used to train a meta-classifier that determines the best set of classifiers for a given test sample. This meta-classifier is trained on features that describe the competence of classifiers, like diversity, confidence, or agreement among classifiers. This approach is highly flexible and adaptable for multiple datasets or applications, but its performance depends on the quality of meta-feature representation and the meta-classifier used.

D. Output Aggregation Techniques

In this section, we review Output Aggregation techniques to combine the outputs from all the classifiers in an ensemble to determine the final output based on a combination rule. These techniques can be trainable or non-trainable.

1) Non-Trainable: Static combination rules like Sum, Product, Maximum, Minimum, Median, and Majority voting schemes can be used as output aggregation logic. As these techniques are not adapted based on the training data, certain assumptions about classifier performance are required for implementing these rules.

2) Trainable: Instead of using a static combination rule, the output aggregation strategy can be customized to a particular application by training another classifier on the predictions of the base classifier. A popular strategy used for output aggregation is Mixture of Experts (ME), which involves training both base classifiers and aggregation classifiers together.

E. Dynamic Partial Reconfiguration

Partial Reconfiguration (PR) is a crucial feature of modern FPGAs, enabling specific regions of the FPGA fabric to be reconfigured at runtime without interrupting the operation of the remaining design. This allows dynamic allocation of hardware resources to changing application requirements, optimizing resource utilization, power efficiency, and system flexibility. Xilinx's Dynamic Function Exchange (DFX) [7] offers a robust framework for implementing PR by partitioning the FPGA design into static and reconfigurable regions, where the static region ensures continuous operation while reconfigurable regions are updated with new functionality via partial bitstreams. Integrated within the Vivado Design Suite, DFX supports dynamic module swapping with reductions in resource usage by up to 50% and power consumption by up to 30% in certain applications. [8]

III. Literature Survey

Ensemble methods have become increasingly popular for applications like network intrusion detection due to the enhanced accuracy offered by combining various classifiers. Dynamic Ensemble Selection (DES) techniques, such as META-DES, adaptively select classifiers based on the input data and improve accuracy for complex and dynamic attack patterns.

Alserhani et al. [6] propose an ensemble-based NIDS that integrates multiple machine learning classifiers to enhance detection accuracy and reduce false positives. They achieve over 99% accuracy and reduce false positives to less than 1% on the UNSW-NB15 dataset by using ensembles.

FPGA is an ideal platform for machine learning applications due to its inherent parallelism, energy efficiency, and real-time performance. Dynamic Partial Reconfiguration (DPR)

enables FPGAs to adapt to changing workloads by reconfiguring regions of the hardware without interrupting the entire system. Several studies have explored the potential of DPR for classifier switching:

Hussain et al. [9] proposed an adaptive FPGA implementation of a multi-core k-nearest Neighbor (KNN) ensemble classifier using DPR. While this approach effectively utilizes DPR to adapt KNN models with varying values of K, the lack of diversity in classifiers limits its applicability to datasets that require heterogeneous models. Our work builds upon this by incorporating a diverse ensemble of classifiers, including Naive Bayes, Decision Trees, and Gradient Boosting.

El Bouazzaoui et al. [10] implemented a real-time adaptive neural network on FPGA using dynamic classifier selection, achieving adaptability with a single dynamic reconfiguration region and one classifier at a time. However, this method is restricted to using one classifier during runtime, which limits its flexibility. Instead of using a single classifier for inference, our work uses a multiple-classifier system with dynamic ensemble selection for NIDS.

IV. Methodology

We leverage Dynamic Partial Reconfiguration (DPR) on an FPGA to implement a dynamic ensemble selection system for real-time classification. The following key components define our methodology:

A. Dynamic Classifier Selection Component

This component currently manages the selection of an ensemble of three classifiers in real-time from a larger pool of pre-trained classifiers using a random selection strategy. This approach will be enhanced in future iterations to dynamically select classifiers based on network traffic characteristics and throughput requirements. This planned adaptation will allow the system to optimize performance by tailoring classifier selection to the specific demands of diverse types of network traffic data.

B. Dynamic Partial Reconfiguration (DPR)

Using DPR, the selected classifiers are dynamically loaded as partial bitstreams into pre-defined partially reconfigurable regions on the FPGA. This approach ensures efficient use of resources by enabling runtime switching of classifiers based on current system demands.

C. Ensemble Classification with Majority Voting

Once the classifiers are reconfigured, they operate in parallel to process input data. The results from each classifier are aggregated using a majority voting mechanism to determine the final output. This ensures robustness and accuracy across diverse input scenarios.

D. Comparison of Hardware Metrics

We plan to evaluate the performance of our FPGA-based dynamic ensemble system against a GPU implementation. While our current focus is on implementing and testing the FPGA-based system, intermediate results will guide the evaluation metrics for GPU-based comparisons. The key metrics for evaluation include:

- **Latency:** Time taken for classifier selection, reconfiguration, and classification.
- **Throughput:** Number of classifications performed per second.
- **Reconfiguration Time:** Overhead associated with loading partial bitstreams.
- **Power Efficiency:** Trade-offs in power consumption between FPGA and GPU implementations.

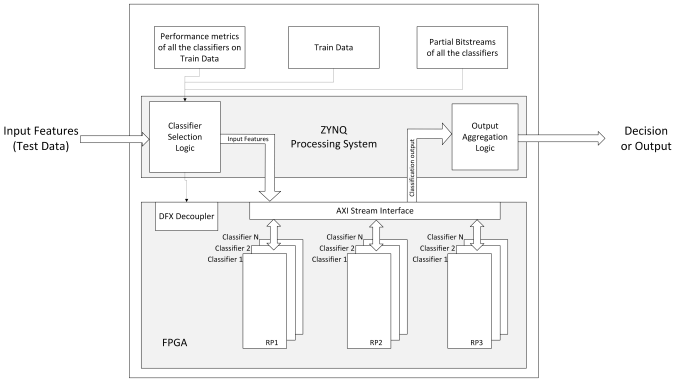

Fig. 2. Proposed system architecture for dynamic ensemble selection.

The GPU-based implementation and comparisons will be conducted in subsequent phases to provide a comprehensive analysis of the trade-offs and performance differences.

E. Data Pre-Processing

We used UNSW Edge IIoT dataset to train and evaluate our classifier models. Several data preprocessing steps were used to prepare the dataset for efficient training. To address missing values, rows or columns with missing data were imputed using mean, median, or mode values. Categorical variables were encoded using One-Hot Encoding technique. Feature scaling was performed using MinMaxScaler to normalize the data. To handle class imbalance, the Synthetic Minority Oversampling Technique (SMOTE) from the imblearn library was used. Finally, the dataset was partitioned into training and testing sets using the 'train_test_split' function from sklearn. The primary libraries used in this process are pandas, numpy, and sklearn.

F. Classifier Implementation

Each classifier implementation reads input features from an AXI stream input and processes them using predefined model parameters. For each classifier, these model parameters or weights are calculated using sklearn library in python and are exported to be used in HLS implementations. These classifiers are optimized for FPGA using HLS pragmas for efficient parallel processing and pipelining. The predicted class is determined based on the specific algorithm used by each classifier.

1) Ada Boost: The AdaBoost classifier contains multiple stumps, which are hardcoded in our implementation. The decision value is computed by aggregating the weighted decisions of these stumps. We used HLS pragmas to partition arrays (X, stumps) for parallel processing and to unroll loops for efficient computation. Pipelining is used to improve throughput.

2) Bernoulli Naive Bayes: This classifier calculates the log probabilities of each class using binary feature inputs and selects the class with the highest score. We used a log_lut lookup table to improve the latency of logarithm calculation, all the class_priors are precomputed and are stored in partitioned arrays for efficient parallel processing.

3) Decision Tree: This classifier navigates a tree structure based on the input value and the values of the nodes. Loop unrolling significantly improved throughput during node evaluations and led to faster tree traversal.

4) Multi Layer Perceptron: MLP classifier uses a simple neural network with one hidden layer to compute predictions. The weights of hidden and output layers are partitioned to improve parallelism. We used an approximation of Sigmoid activation function to improve latency while not compromising on accuracy of computation.

5) Support Vector Machines: SVM computes predictions based on weighted support vectors and their coefficients. We used double buffering along with array partitioning for efficient data handling during operations to improve latency.

6) Logistic Regression: This classifier uses a weighted sum of inputs, applies the sigmoid function, and classifies based on a threshold. We simplified the sigmoid function using linear approximation to improve both latency and resource utilization. We also used dataflow optimization combined with array partitioning to allow different computation stages to run parallelly.

G. PYNQ Implementation

PYNQ (Python Productivity for Zynq) is an open-source framework developed by Xilinx to program Xilinx Zynq-7000 and Zynq UltraScale+ MPSoC devices using high-level programming languages like Python. PYNQ abstracts much of this complexity away, allowing developers to interact with FPGA-accelerated functions through simple Python libraries and Jupyter notebooks. We use Overlays to program the board and Direct Memory Access to send and receive data from the Programmable logic.

- Overlay: An overlay is the FPGA equivalent of a software program or library that you "load" into the programmable logic (PL) fabric. Instead of hard-coding functionality into the FPGA at manufacturing time, you can dynamically configure the FPGA at runtime by loading a bitstream and associated Python APIs.
- DMA: PYNQ overlays rely on an AXI DMA controller IP to facilitate these data transfers. AXI is a standard interconnect protocol used in Xilinx designs. The DMA controller sits between the Processing System (PS) memory and the FPGA-based accelerator logic.
- Memory Buffers: The allocate() function is used to create contiguous memory buffers that can be directly accessed by both the Processing System (PS) and the Programmable Logic (PL) for high-speed data transfer. These buffers are essential when working with DMA or other hardware accelerators because they ensure that data is laid out in a predictable, physically continuous way that the hardware IPs can directly read from or write to, without the need for software copies.

We begin by loading a specific FPGA overlay and establishing two DMA channels—one dedicated to transmitting input data to the FPGA and another dedicated to receiving the processed output. Next, we allocate contiguous memory buffers to ensure efficient data transfers between the processor and the programmable logic. After populating the input buffer with our chosen data set, we flush it, allowing the programmable logic to access it. Subsequently, we initiate the DMA transfers, allowing the FPGA to process the input data and place the results into the output buffer. Once these operations are complete, we wait for the DMA channels to signal their completion, confirming that the data transfer and processing have concluded successfully. The classifier selection function is implemented using the numpy random function.

H. Input Muxing and Majority Voting Mechanism

In our hardware implementation, we designed a datapath that efficiently feeds the same input data to all three classifiers using a multiplexing (MUX) mechanism. This ensures that each classifier receives identical inputs simultaneously, which is crucial for consistent and synchronized classification results.

Once the classifiers process the input data, their outputs are forwarded to the majority voting logic. The majority vote module waits for the outputs from all three classifiers before making a decision. It aggregates the classification results and determines the final output based on the majority of the classifier outputs. This approach enhances the robustness and accuracy of the system by mitigating the impact of any single classifier's misprediction.

To verify the functionality of the implemented design, we conducted simulations using the Register Transfer Level (RTL) models of the classifiers, along with our custom MUX and majority vote modules. We verified each ensemble configuration to ensure correct operation, with simulations covering Ensembles 1 through 10.

Figures 3 to 12 show the simulation waveforms for all ensembles. In each waveform, the ensemble output signal, named `ensemble_to_vote`, is color-coded to signify the outputs from each classifier to the voting logic. The simulations demonstrate that once the last classifier provides its output, the majority vote module processes these outputs and generates the final decision at the top level. These results confirm the correct integration and functionality of the MUX, classifiers, and majority vote logic in our design across all ensembles.

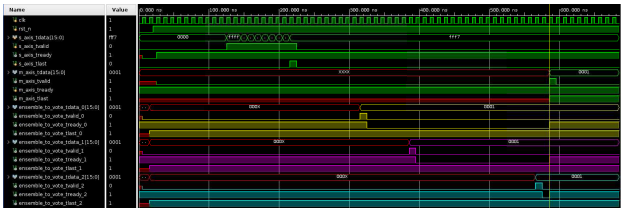

Fig. 3. Simulation waveform for Ensemble 1, highlighting the ensemble outputs and the majority vote decision.

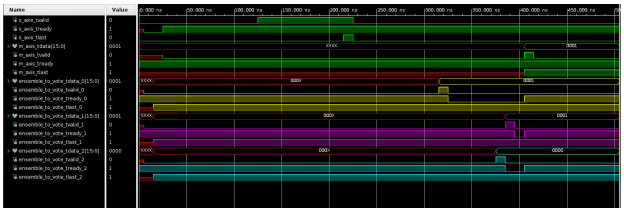

Fig. 4. Simulation waveform for Ensemble 2, highlighting the ensemble outputs and the majority vote decision.

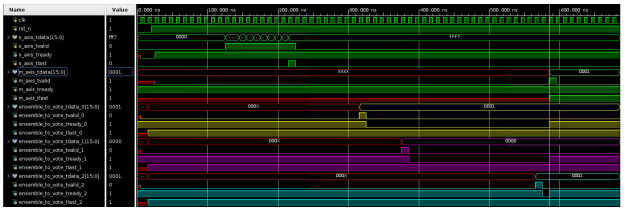

Fig. 5. Simulation waveform for Ensemble 3, highlighting the ensemble outputs and the majority vote decision.

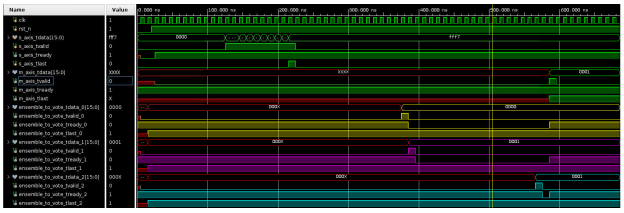

Fig. 6. Simulation waveform for Ensemble 4, highlighting the ensemble outputs and the majority vote decision.

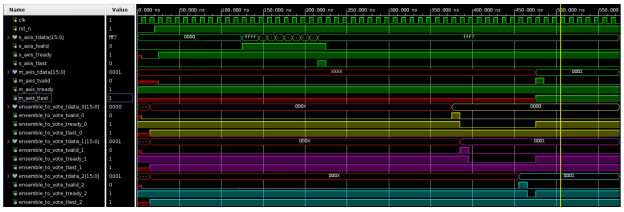

Fig. 7. Simulation waveform for Ensemble 5, highlighting the ensemble outputs and the majority vote decision.

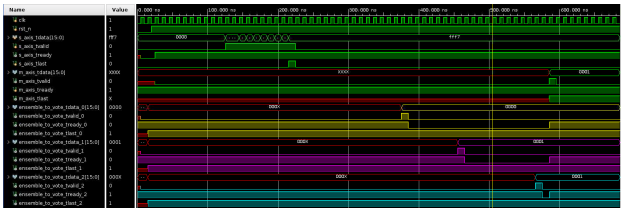

Fig. 8. Simulation waveform for Ensemble 6, highlighting the ensemble outputs and the majority vote decision.

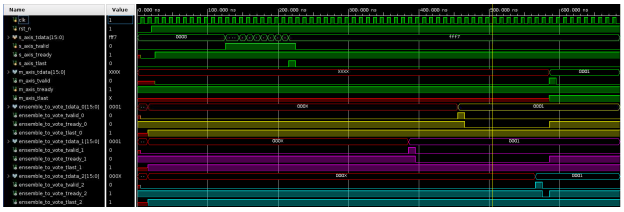

Fig. 9. Simulation waveform for Ensemble 7, highlighting the ensemble outputs and the majority vote decision.

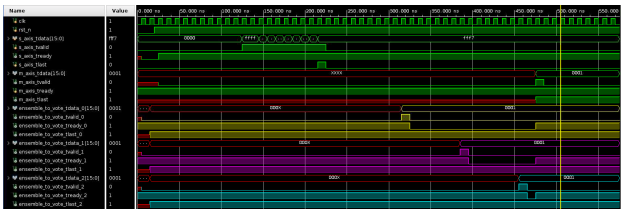

Fig. 10. Simulation waveform for Ensemble 8, highlighting the ensemble outputs and the majority vote decision.

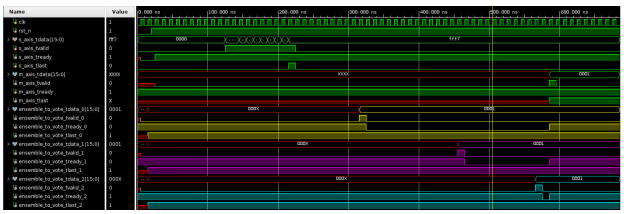

Fig. 11. Simulation waveform for Ensemble 9, highlighting the ensemble outputs and the majority vote decision.

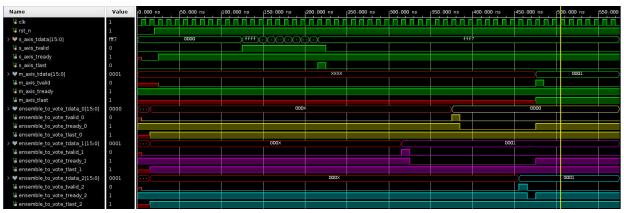

Fig. 12. Simulation waveform for Ensemble 10, highlighting the ensemble outputs and the majority vote decision.

V. Results

Implemented Classifiers

The classifiers implemented in HLS are trained using a quantized dataset of UNSW-NB15, with the classifier weights loaded into each respective HLS implementation. The C-RTL co-simulation results show the accuracy of each classifier on the UNSW-NB15 dataset.

The accuracy achieved with the HLS tools is observed to be lower than the sklearn baseline, which is expected due to the precision limitations inherent to HLS implementations.

Classifier Name	**CPU Latency** (ns)	**FPGA Latency** (ns)	**BRAM**	**DSP**	**FF**	**LUT**	**Accuracy** sklearn	**Accuracy** C-RTL Simulation
Total Available	-	-	144	1248	234,240	117,120	-	-
Gaussian NB	149	96.9	8	263	1,558	3,357	99.62	89.6
Logistic Regression	11	64.6	0	12	512	992	99.7	89.3
MLP	100,000	85.3	0	201	2,916	6,290	99.9	75.57
Gradient Boost	150	145.9	0	0	636	2,730	99.8	89.5
Support Vector Machine	-	213.6	0	11	2550	2663	99.9	75.57

TABLE I
COMPARISON OF CLASSIFIERS IMPLEMENTED IN HLS WITH C-RTL CO-SIMULATION AND SKLEARN ACCURACY.

However, we are optimistic that accuracy and latency can be improved with further optimizations to our HLS code. Table I shows the details of the classifiers, including their accuracy for both sklearn and C-RTL co-simulation.

Implemented Ensembles

We have implemented ten total ensembles, each consisting of three classifiers. These ensembles are designed to demonstrate our system's dynamic selection capabilities.

Ensemble #	**classifier 1**	**classifier 2**	**classifier 3**
Ensemble 1	Gaussian Naive Bayes	Logistic Regression	MLP
Ensemble 2	Gaussian Naive Bayes	Logistic Regression	Gradient Boost
Ensemble 3	Gaussian Naive Bayes	Gradient Boost	MLP
Ensemble 4	Gradient Boost	Logistic Regression	MLP
Ensemble 5	Gradient Boost	Logistic Regression	SVM
Ensemble 6	Gradient Boost	MLP	SVM
Ensemble 7	SVM	MLP	Logistic Regression
Ensemble 8	SVM	Logistic Regression	Gaussian Naive Bayes
Ensemble 9	SVM	Gaussian Naive Bayes	MLP
Ensemble 10	SVM	Gaussian Naive Bayes	Gradient Boost

TABLE II
IMPLEMENTED ENSEMBLES AND THEIR CONSTITUENT CLASSIFIERS.

The implemented ensembles are shown in Table II, with each ensemble's composition defined by the classifier names.

Synthesized and Implementation Results in Vivado

The synthesized and implemented results of each ensemble in Vivado include resource utilization and power consumption metrics. These metrics are essential for evaluating the efficiency of our design. The data includes details on Look-Up Tables (LUTs), Flip-Flops (FFs), Block RAM (BRAM), Digital Signal Processors (DSPs), and power consumption parameters such as total power, junction temperature, and thermal margin.

Resource Utilization: Table III shows the resource utilization for each ensemble, including the number of LUTs, LUTRAM, FFs, BRAMs, DSPs, and BUFGs. These results indicate the hardware resources utilization of each ensemble.

Name	**LUT**	**LUTRAM**	**FF**	**BRAM**	**DSP**	**BUFG**
Ensemble black box	8767	502	11052	2	0	2
Ensemble 1	18511	12226	15756	6	476	2
Ensemble 2	15668	518	13475	6	275	2
Ensemble 3	20141	1226	15853	6	464	2
Ensemble 4	17831	1210	14833	2	213	2
Ensemble 5	12824	520	12342	2	23	2
Ensemble 6	17278	1228	14720	2	212	2
Ensemble 7	15672	1228	14623	2	224	2
Ensemble 8	13502	536	13265	6	286	2
Ensemble 9	17969	1244	15643	6	475	2
Ensemble 10	15127	536	13362	6	274	2

TABLE III
RESOURCE UTILIZATION FOR EACH ENSEMBLE IN VIVADO.

Power Consumption: The power consumption of each ensemble is summarized in Table IV. Key metrics include the total power (in watts), junction temperature (in Celsius), and thermal margin (in watts). These metrics provide insights into each ensemble design's thermal and power efficiency.

Name	**Total Power Watts**	**Junct-Temp Cels.**	**Them-Marg Watts**
Ensemble black box	3.198	32.4	28.4
Ensemble 1	3.556	33.2	28.1
Ensemble 2	3.447	33	28.2
Ensemble 3	3.511	33.1	28.1
Ensemble 4	3.294	32.6	28.3
Ensemble 5	3.223	32.5	28.4
Ensemble 6	3.301	32.7	28.3
Ensemble 7	3.308	32.7	28.3
Ensemble 8	3.443	33	28.2
Ensemble 9	3.524	33.2	28.1
Ensemble 10	3.342	32.8	28.3

TABLE IV
POWER CONSUMPTION METRICS FOR EACH ENSEMBLE IN VIVADO.

These results highlight the trade-offs between resource utilization and power consumption for each ensemble, helping us evaluate the performance and efficiency of the dynamic ensemble system implemented with DPR.

Partial Region for Each Classifier Implementation

The final floorplan for each partial configuration of the ensembles is shown in the following diagrams. The static region is highlighted in orange, representing the fixed logic

that remains unchanged across configurations. The dynamic region, shown in the pink subregion, contains the dynamically reconfigured classifiers for each ensemble. The blue routed cells within the pink region represent the dynamically reconfigured logic specific to each classifier.

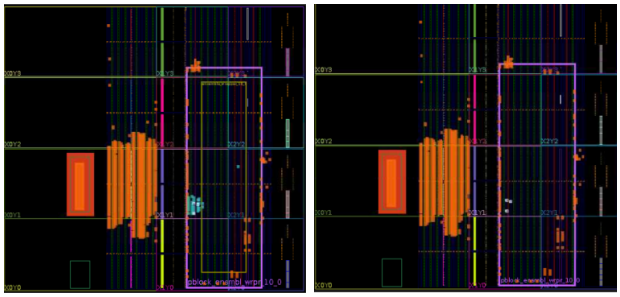

Fig. 13. Floorplan for Static Region (left) and Ensemble Blackbox (right).

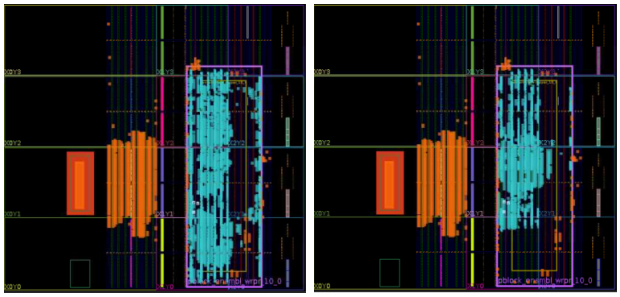

Fig. 14. Floorplan for Ensemble 1 (left) and Ensemble 2 (right).

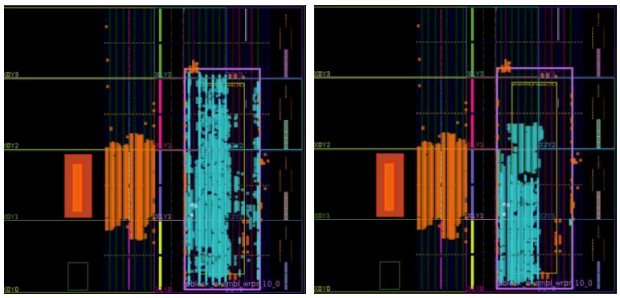

Fig. 15. Floorplan for Ensemble 3 (left) and Ensemble 4 (right).

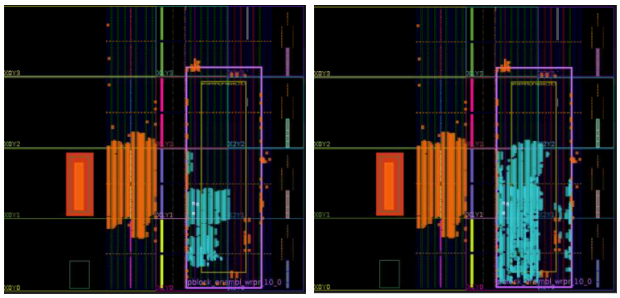

Fig. 16. Floorplan for Ensemble 5 (left) and Ensemble 6 (right).

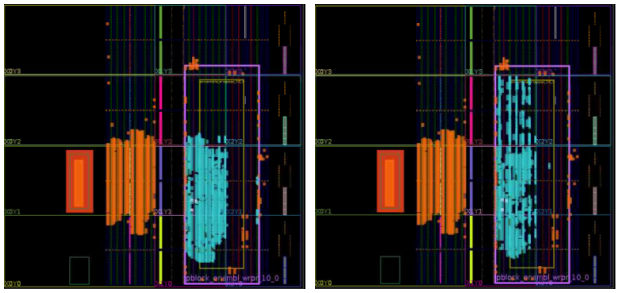

Fig. 17. Floorplan for Ensemble 7 (left) and Ensemble 8 (right).

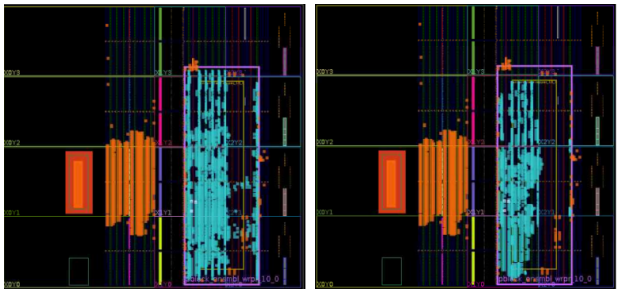

Fig. 18. Floorplan for Ensemble 9 (left) and Ensemble 10 (right).

These floorplans illustrate the effectiveness of our dynamic partial reconfiguration approach, ensuring that each ensemble's classifiers are efficiently loaded into the dynamic region while preserving the integrity of the static region.

A. *Reconfiguration Time*

We generated complete and partial bitstreams for all the ensembles implemented and configured the Xilinx PYNQ-ZU board using the PYNQ Python interface.

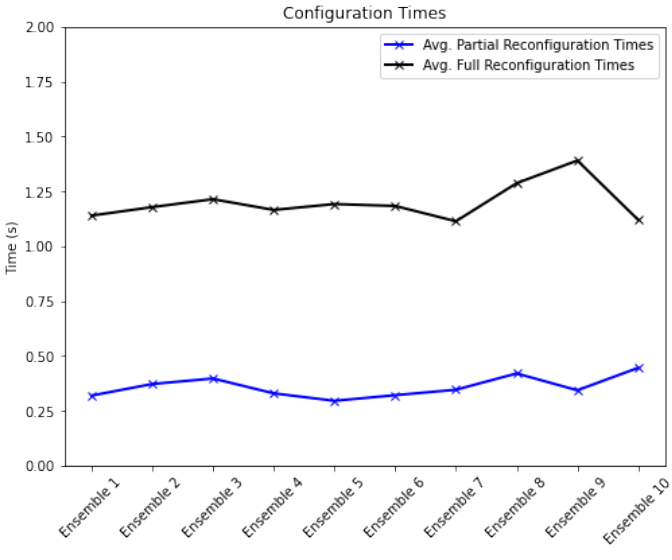

Fig. 19. Reconfiguration Times for each ensemble.

[h!]

This shows that partial reconfiguration times are considerably lower than full reconfiguration times for all the ensembles, which justifies the use of DPR instead of complete reconfiguration. The current implementation uses the Xilinx PYNQ framework, which relies on JTAG offering a transfer speed of 66 Mbps. While we planned to implement HWICAP for DPR, which provides transfer speeds of up to 3.1 Gbps and would significantly improve reconfiguration times and reduce reconfiguration overhead in the dynamic ensemble selection system, this was not feasible in our project due to the lack of HWICAP driver support in the PYNQ framework. However, we argue that a design leveraging a traditional processor or a soft processor with HWICAP support could significantly enhance partial reconfiguration times, overcoming the limitations imposed by the current PYNQ-based implementation

VI. Evaluation Platform

Our design is evaluated on a PYNQ-ZU board, utilizing its advanced features for dynamic partial reconfiguration. Table V provides the specifications of the PYNQ-ZU board used in our experiments.

Feature	**Specification**
FPGA Model	Zynq UltraScale+ XCZU5EG-1SFVC784 MPSoC
Programmable Logic (PL)	117k LUTs
Block RAM	5.1 Mbits
DSP Slices	1248
On-Chip Memory (OCM)	256 KB

TABLE V
Relevant hardware specifications of the PYNQ-ZU board.

Development and Evaluation Tools

The design was synthesized and implemented using Xilinx Vivado and Vitis HLS 2022.1, running on the Arizona State University (ASU) Sol Server. The Sol Server provided the computational resources required for the design's synthesis, implementation, and simulation.

Deployment and Execution Framework

The final bitstreams, including the partial bitstreams, will be deployed and executed on the PYNQ-ZU board using the PYNQ framework. Jupyter Notebooks will be used to load and manage the bitstreams dynamically, enabling real-time testing and evaluation of the classifiers. This framework allows us to capture and analyze results efficiently, which will be included in our final evaluation.

Classifier Implementation

Baseline models for the classifiers were trained using scikit-learn (sklearn) on UNSW-NB15 dataset for binary classification of Network Intrusion Detection Problem. These models served as the foundation for our HLS implementations. Trained weights and parameters are exported and used in Vitis HLS to generate hardware-efficient classifiers for our FPGA design.

VII. Next Steps

The following steps outline the plan to complete and evaluate our project:

Improving Accuracy and Latency of HLS Models

We aim to optimize our High-Level Synthesis (HLS) models to improve both accuracy and latency. This includes refining the quantization process, optimizing the HLS implementation for better precision, and minimizing computational delays.

Adding an Additional Classifier

We plan to implement an additional classifier to enable a more comprehensive comparison in the final results. This will increase the total number of ensembles from 4 to 10, allowing for a broader performance evaluation across diverse ensemble configurations.

Running Partial Bitstreams on the PYNQ-ZU Platform

The next step is to deploy the generated partial bitstreams on the PYNQ-ZU platform using Dynamic Partial Reconfiguration (DPR). This will allow us to load and test each ensemble configuration dynamically in real-time, capturing performance metrics and validating the design.

Implementing GPU and CPU Comparisons

In addition to the FPGA-based implementation, we will develop a GPU-based implementation of the ensemble classifiers. This implementation will serve as a performance baseline for comparison with our FPGA design.

We also plan to use the PYNQ-ZU board's built-in hard processor (CPU) to run the ensembles. By implementing both GPU and CPU versions, we will be able to comprehensively evaluate and compare the performance of the FPGA, GPU, and CPU implementations.

Comparing Resource Utilization and Power Efficiency

The final step involves comparing the resource utilization and power efficiency of the three platforms:

- **FPGA:** Measure resource utilization (LUTs, FFs, BRAMs, and DSPs) and power consumption during DPR and runtime operation.
- **GPU:** Evaluate latency, throughput, and power efficiency during ensemble inference.
- **CPU:** Use the PYNQ-ZU's hard processor to assess resource usage and power efficiency during CPU-based ensemble inference.

References

[1] Shone, N., Ngoc, T.N., Phai, V.D., Shi, Q. "A Deep Learning Approach to Network Intrusion Detection." IEEE Transactions on Emerging Topics in Computational Intelligence, 2018.

[2] Alom, M.Z., Taha, T.M., Yakopcic, C., et al. "Intrusion Detection Using Deep Learning-Based Models." IEEE Access, 2019.

[3] Goodfellow, I., Pouget-Abadie, J., Mirza, M., et al. "Explaining and Harnessing Adversarial Examples." IEEE Security and Privacy, 2015.

[4] Rafael M.O. Cruz, Robert Sabourin, George D.C. Cavalcanti, Dynamic classifier selection: Recent advances and perspectives, Information Fusion, Volume 41, 2018, Pages 195-216, ISSN 1566-2535, https://doi.org/10.1016/j.inffus.2017.09.010

[5] I. D. Mienye and Y. Sun, "A Survey of Ensemble Learning: Concepts, Algorithms, Applications, and Prospects," in IEEE Access, vol. 10, pp. 99129-99149, 2022, doi: 10.1109/ACCESS.2022.3207287.

[6] F. Alserhani and A. Aljared, "Evaluating Ensemble Learning Mechanisms for Predicting Advanced Cyber Attacks," Applied Sciences, vol. 13, no. 24, Art. no. 24, Jan. 2023, doi: 10.3390/app132413310.

[7] Xilinx, "Vivado Design Suite User Guide: Dynamic Function Exchange," Xilinx Documentation, 2020.

[8] Kizheppatt Vipin and Suhaib A. Fahmy. 2018. FPGA Dynamic and Partial Reconfiguration: A Survey of Architectures, Methods, and Applications. ACM Comput. Surv. 51, 4, Article 72 (July 2019), 39 pages. https://doi.org/10.1145/3193827

[9] H. Hussain, K. Benkrid, C. Hong and H. Seker, "An adaptive FPGA implementation of multi-core K-nearest neighbour ensemble classifier using dynamic partial reconfiguration," 22nd International Conference on Field Programmable Logic and Applications (FPL), Oslo, Norway, 2012, pp. 627-630, doi: 10.1109/FPL.2012.6339251

[10] Achraf El Bouazzaoui, Abdelkader Hadjoudja, Omar Mouhib, "Real-Time Adaptive Neural Network on FPGA: Enhancing Adaptability through Dynamic Classifier Selection," arXiv, 2023. [Online]. Available: https://arxiv.org/abs/2311.09516v2.

[11] Luca Pezzarossa, Andreas Toftegaard Kristensen, Martin Schoeberl, Jens Sparsø, Using dynamic partial reconfiguration of FPGAs in real-Time systems, Microprocessors and Microsystems, Volume 61, 2018, Pages 198-206, ISSN 0141-9331, https://doi.org/10.1016/j.micpro.2018.05.017

[12] B. Seyoum, M. Pagani, A. Biondi, S. Balleri and G. Buttazzo, "Spatio-Temporal Optimization of Deep Neural Networks for Reconfigurable FPGA SoCs," in IEEE Transactions on Computers, vol. 70, no. 11, pp. 1988-2000, 1 Nov. 2021, doi: 10.1109/TC.2020.3033730

[13] O. Almomani, M. A. Almaiah, A. Alsaaidah, S. Smadi, A. H. Mohammad and A. Althunibat, "Machine Learning Classifiers for Network Intrusion Detection System: Comparative Study," 2021 International Conference on Information Technology (ICIT), Amman, Jordan, 2021, pp. 440-445, doi: 10.1109/ICIT52682.2021.9491770

[14] Sarhan, M., Layeghy, S. & Portmann, M. Towards a Standard Feature Set for Network Intrusion Detection System Datasets. Mobile Netw Appl 27, 357–370 (2022). https://doi.org/10.1007/s11036-021-01843-0

[15] Khraisat, A., Gondal, I., Vamplew, P. et al. Survey of intrusion detection systems: techniques, datasets and challenges. Cybersecur 2, 20 (2019). https://doi.org/10.1186/s42400-019-0038-7

[16] R. M. O. Cruz, H. H. Zakane, R. Sabourin and G. D. C. Cavalcanti, "Dynamic ensemble selection VS K-NN: Why and when dynamic selection obtains higher classification performance?," 2017 Seventh International Conference on Image Processing Theory, Tools and Applications (IPTA), Montreal, QC, Canada, 2017, pp. 1-6, doi: 10.1109/IPTA.2017.8310100.

APPENDIX A
ARTIFACT APPENDIX

A. Abstract

This section offers an overview of the artifact evaluation process. We begin by presenting a detailed checklist describing the artifact's components. We outline the directory structure and associated documentation. Finally, we demonstrate the installation procedure, experimental workflow, and evaluation steps, illustrating how to use the artifact to both reproduce and extend our results.

B. Artifact check-list (meta-information)

- **Program:** C++ and Verilog
- **Model:** Gaussian Naive Bayes, Logistic Regression, Gradient Boost, MLP and Support Vector Machine
- **Datasets:** UNSW-Edge IIoT
- **Infrastructure:** ASU Sol Supercomputer with 8 cores, Pynq ZU FPGA development board.
- **Software Tools:** Vitis HLS, Vivado, Jupyter Notebook(Pynq)
- **Publicly Available:** Yes
- **Code Repository:** https://github.com/berrios96sean/eee-598-project

C. Description

This section describes the dependencies, models and datasets used.

1) Software Access: The code for this paper can be accessed by cloning the public GitHub repository: https://github.com/berrios96sean/eee-598-project. The directory structure is explained below:

- **build:** The build directory contains scripts to create and build the Vivado project.
- **IP:** The IP directory consists of all the Vivado IP implemented using VITIS HLS.
- **pr:** The pr directory contains the bitstreams, static and dynamic of all the implemented ensembles.
  This directory also has design checkpoint (dcp) files for all the ensembles. There is also a hwh file that contains the information of all the ips used.
- **sim:** The sim directory contains the simulation files and test benches used to verify ensemble functionality
- **src:** The src directory contains all the source files used for the project.
  - axis: Contains the code for the majority vote in Verilog.
  - block_design: Contains the Vivado block designs for all the IPs.
  - hls: Contains the top-level C++ implementations for all the classifiers.
  - top: Contains the top-level Verilog wrapper for the project
  - wrapper: Contains all the ensemble and classifier wrappers used for partial reconfiguration.
- **train**: Contains all the Jupyter notebooks used to train the classifiers.
- **PYNQ**: Contains the Jupyter notebooks used for PYNQ workflow and also the partial bitstreams along with corresponding dwh files required for DPR.

2) Software Dependencies: The classifiers are written in C++ and run using the Vitis HLS software. Implementation and IP generation are performed using Vitis and imported into Vivado. The majority voting scheme and the data flow are designed on Vivado using Verilog. The FPGA board requires a browser and uses jupyter notebook and a python3 kernel.

3) Hardware and Infrastructure Dependencies: The classifier implementation and training were performed on ASU Sol supercomputer. Based on the LUT requirements, the PYNQ ZU board was selected with the Zynq Ultrascale+ XCZU5EG FPGA. An FPGA with a processing system is also recommended.

4) Datasets: UNSW-Edge IIoT

The dataset files we uploaded on canvas are quantised and scaled versions of Edge IIoT dataset. We also have a similar version of UNSW NB15 dataset in the same repo. Additionally we also split these files into train and test categories. Test data set which is further split into features and lables is converted to a text file for easier handling with C-RTL co simulation and PYNQ.

Scaled version of Edge IIoT dataset has a total of 815288 datapoints out of which, 436535 are normal and 378753 are various attack categories.

5) Models: The pre-trained model code and their training .ipynb files are provided in the src and train section in the GitHub repository. The following models were designed in C++ and AXI stream interfaces were added to allow compatibility with the Vivado AXI Stream workflow.

- Gaussian NB
- Logistic Regression
- MLP
- Gradient Boost
- Support Vector Machine

D. Workflow

For ease of evaluation, sample scripts are available to run our work. The following steps can get you started with the project.

- Download Code: You can find the project in our Github repository
- Tool Initialization: Export the paths to Vivado and Vitis. For the sol supercomputer, here's the command:
  export PATH=$PATH:/packages/apps/fpga/Vivado/2022.1/bin/
  export PATH=$PATH:/packages/apps/fpga/Vitis_HLS/2022.1/bin/
- Project setup: Once the paths have been set up, use the following commands from the root directory to set up the project.
  - make clean: Clean the work area.

  - make build_gui: Builds the Vivado project, sources the IP, and creates the connections.
  - make sim_bb: Performs Simulations on a black-boxed design
  - make sim_ensemble: Performs Simulations on the ensemble design.
- Dynamic Partial Reconfiguration: An Open GUI session is required to perform partial reconfiguration. Perform the following steps:
  - Modify the PR cell to be a reconfigurable cell. This can be done by modifying the following properties.
    * HD.RECONFIGURABLE: True
    * HD.RECONFIG_PLATFORM: True
    * DONT_TOUCH: True
  - Assign a PR region. This can be performed by right-clicking and defining a new PR boundary. Ensure that the PR Boundary has sufficient resources for the worst-case ensemble.
  - Compile the following design and create a checkpoint. This can be done with the following commands.
    * Compile: opt_design, place_design, route_design
    * Checkpoint: write_checkpoint

    This step is performed for all ensembles and the bitstreams are available in the repository.
- FPGA Implementation: It is possible to work on the FPGA without creating the bitstreams. The bitstreams are present in the GitHub repository. Use the overlay command to read in the base bitstream. The partial bitstreams can be read using the bitstream command. Use the DMA logic to send and receive data for the ensemble. A sample script is present in our GitHub repository.

E. Experiment Customization

The codebase is organized to facilitate customization and extension. The build file allows for easily specifying experiment configurations, including simulations, model configurations, and the number of ensembles to pick for the final design.

Usage of TCL scripts to automate Vivado flow makes this customization ever more simpler. To change the ensembles or classifiers used in ensembles, modifying or adding the new files in **src** folder and making minimal changes to ensemble wrappers would be sufficient.

APPENDIX B
TEAM MEMBER CONTRIBUTION

This section details the individual contributions of the team members to the project.

A. Sean

Contributed significantly to both the software and hardware aspects of the project. Initially, developed High-Level Synthesis (HLS) models for the Logistic Regression and Gaussian Naive Bayes classifiers, laying the foundation for the classification system. These models were later optimized by Ruthwik for use with the UNSW-NB15 dataset to improve their performance.

On the hardware side, Sean designed the block diagram and Register Transfer Level (RTL) logic for the datapath, which included muxing inputs to all three classifiers in the hardware design. Additionally, implemented the majority vote model to aggregate the outputs from the three classifiers, ensuring accurate and robust decision-making in the system.

Sean also created the full and partial bitstreams for each ensemble configuration, enabling the Dynamic Partial Reconfiguration (DPR) process. As part of this task, Sean reported the resource usage for each ensemble and finalized the floorplanning details, ensuring optimal resource allocation and efficient hardware implementation within the FPGA. Furthermore, Sean conducted the functional verification of each ensemble model using simulation to validate their correctness and ensure the successful integration of all components in the system.

These contributions were critical in bridging the software and hardware domains, ultimately ensuring the success of the dynamic ensemble selection system.

B. Ruthwik

Ruthwik made significant contributions to the project, focusing on both data preparation and advanced hardware-software co-simulation techniques. He preprocessed the UNSW NB15 dataset, Edge IIoT dataset and Botnet Dataset, performed feature space exploration, and identified the most suitable classifiers for the Edge IIoT dataset. To establish a performance baseline, he implemented classifiers on a CPU and benchmarked their accuracy.

By simulating the interaction between the HLS-generated RTL and the input data through AXI stream interface, Ruthwik ensured that the classifiers met the required accuracy in the FPGA environment. His focus on hardware-software consistency, particularly in terms of data representation and the precision of datatypes, was critical in aligning the project's objectives with the practical constraints of FPGA-based systems.

Ruthwik also developed efficient HLS implementations for various classifiers, including Gaussian Naive Bayes (gaussian_nb), Gradient Boosting (gradient_boost), MultiLayer Perceptron (mlp), Bernoulli Naive Bayes (bernoulli_nb), Decision Tree (decision_tree), Logistic Regression (lr), and AdaBoost (ada_boost) and Convolutional Neural Networks (cnn) ensuring they were suited for FPGA deployment. But eventually we decided to just use 5 classifiers which provided highest accuracy. His work laid the foundation for a reliable and high-performing classifier ensemble

C. Deepesh

Deepesh was integral to the development and integration of the system on the PYNQ platform, a crucial step in enabling dynamic adaptability. He implemented a system that dynamically performed Partial Reconfiguration (PR) based on the characteristics of incoming test inputs. This allowed the FPGA to reconfigure itself in real time, adapting to different

ensemble configurations as required. Deepesh's work ensured that test inputs were efficiently passed to the FPGA from the PYNQ system and that inference outputs were retrieved seamlessly using DMA, creating a robust end-to-end data processing pipeline.

In addition to the integration work, Deepesh contributed to the HLS implementation of the K-Nearest Neighbors (KNN) classifier. He optimized the KNN algorithm for FPGA deployment by introducing pre-calculation and approximation techniques for certain distance computations, significantly reducing computational overhead. These enhancements ensured that the KNN classifier met the performance and resource constraints of the FPGA.

Deepesh also generated partial bitstreams for ensemble configurations and tested Dynamic Partial Reconfiguration (DPR), a process later automated by Sean using TCL scripts. Deepesh's work on PYNQ system integration and DPR testing was pivotal in achieving a real-time dynamic ensemble selection system.