Anomaly Detection System

Atakan Öztarak Barış Türnüklü

11 May 2024

Contents

1 Introduction **3**

2 Tools and Data Description **4**
2.1 Tools Utilized 4
2.2 Data Overview 4
2.3 Dataset Feature Description 4

3 Methodology **7**
3.1 Initial Attempts 7
3.2 Exploration with Autoencoders 7
3.3 Feature Selection and Refinement 7
3.4 Successful Methodology 7
3.4.1 Data Cleaning and Normalization 7
3.4.2 Conversion to Image Format 8
3.4.3 Visualization of Processed Data 8
3.5 Machine Learning Models and Techniques 10
3.5.1 Convolutional Neural Network (CNN) Architecture . . 10
3.5.2 Siamese Network Configuration 11
3.5.3 Training and Validation 12
3.5.4 Performance Evaluation 13
3.5.5 Results Visualization 13

4 Analysis and Discussion **14**
4.1 Initial Unsuccessful Results 14
4.2 Refinement and Feature Reduction 14
4.3 Successful Methodology Using Image-Converted Data 15

5 Conclusion **16**
5.1 Summary of Findings 16
5.2 Implications and Future Work 16
5.3 Concluding Remarks 17

1 Introduction

In the era of rapidly evolving cybersecurity threats, developing efficient and adaptable anomaly detection systems is crucial for securing digital infrastructures. Traditional anomaly detection techniques often require extensive datasets of labeled examples to train robust models. However, in many real-world scenarios, especially in cybersecurity, acquiring such comprehensive datasets can be impractical (Since human power is needed for labeling and collecting those datasets and humans cannot efficiently complete those tasks in a million lines of data.) or impossible due to the rarity of certain types of network attacks or anomalies. This challenge necessitates a shift toward more sophisticated learning strategies capable of making the most out of limited data.

One such advanced approach is few-shot learning, it is a technique that allows models to learn and make predictions from a very limited number of training examples (We're talking about really limited training examples such as one, two, three, and maybe at most 10-50). Few-shot learning is particularly promising for tasks where acquiring labeled data is costly or where new data classes frequently emerge, requiring rapid adaptation by the learning model.

The anomaly detection system developed in this project will utilize deep learning algorithms configured to perform under a few-shot learning paradigm, enabling the model to effectively generalize from minimal examples. The challenge is to design a model that can accurately identify deviations from normal behavior without having direct examples of anomalies during training. This requires transforming inherently one-dimensional network flow data—sourced directly from Kubernetes—into a two-dimensional format. Such transformation is inspired by methodologies commonly employed in computer vision and adapts them to the cybersecurity domain to leverage advanced neural network architectures.

The objective of this project is not merely to detect anomalies but to do so with high levels of accuracy, precision, and recall. The classification task is binary, distinguishing data as either 'anomalous-malicious' or 'not anomalous-benign'. Through many testing and optimization, the final deliverable will be a robust anomaly detection system that helps to provide secure infrastructures in a container, specifically tailored for the complex ecosystem of Kubernetes.

2 Tools and Data Description

2.1 Tools Utilized

For the successful implementation of this project, a range of computational tools, software, and platforms were used:

- **Google Colab:** Leveraged for its provision of powerful free GPU resources, essential for the computational demands of training deep learning models.
- **TensorFlow and Keras:** These frameworks were chosen for building and training neural network models due to their robustness, flexibility, and extensive support community.
- **Python Libraries:** Libraries such as Pandas for data manipulation, NumPy for numerical operations, and Matplotlib for data visualization were extensively used to support various stages of data preprocessing and analysis.

2.2 Data Overview

The dataset was provided by Yiğit Sever, and it is available in his repository, which can be accessed Yiğit Sever's Kubernetes Dataset. Data is captured using Wireshark, focusing on network flows within Kubernetes.

2.3 Dataset Feature Description

This section elaborates on each feature within the network flow dataset captured using Wireshark. Understanding these features is crucial for effectively processing the data and designing the anomaly detection model.

- **Flow ID**: A unique identifier for each network flow, typically composed of concatenated source IP, destination IP, source port, destination port, and protocol.
- **Src IP and Dst IP**: IP addresses for the source and the destination of the network packets.
- **Src Port and Dst Port**: Network ports used at the source and the destination.
- **Protocol**: The protocol used for the network communication, such as TCP, UDP, etc.

- **Timestamp**: The exact date and time at which the network flow was recorded.
- **Flow Duration**: The total duration of the flow from its start to its end, measured in microseconds or milliseconds.
- **Total Fwd Packet and Total Bwd Packets**: Counts of packets sent forward (from source to destination) and backward (from destination to source).
- **Total Length of Fwd Packet and Total Length of Bwd Packet**: Total lengths of all packets sent in the forward and backward directions.
- **Fwd Packet Length Max, Min, Mean, Std and Bwd Packet Length Max, Min, Mean, Std**: Statistical measures (maximum, minimum, mean, standard deviation) of lengths of packets sent in both forward and backward directions.
- **Flow Bytes/s and Flow Packets/s**: Rate of flow in bytes per second and packets per second.
- **Flow IAT Mean, Std, Max, Min**: Inter-Arrival Time (IAT) statistics (mean, standard deviation, maximum, minimum) for the entire flow.
- **Fwd IAT Total, Mean, Std, Max, Min and Bwd IAT Total, Mean, Std, Max, Min**: Specific IAT statistics for packets sent forward and backward.
- **Fwd PSH Flags and Bwd PSH Flags, Fwd URG Flags and Bwd URG Flags, etc.**: TCP flags like PSH (push) and URG (urgent), indicating special conditions of packet transmission.
- **Fwd RST Flags and Bwd RST Flags**: Flags indicating reset packets sent in forward and backward directions.
- **Fwd Header Length and Bwd Header Length**: Lengths of the headers in packets sent forward and backward.
- **Fwd Packets/s and Bwd Packets/s**: Rate of packets sent per second in both forward and backward directions.
- **Packet Length Min, Max, Mean, Std, Variance**: Overall packet length statistics for the entire flow.

- **FIN, SYN, RST, PSH, ACK, URG, CWR, ECE Flag Count**: Count of each type of TCP flag across all packets in the flow.
- **Down/Up Ratio**: Ratio of the downstream to the upstream traffic.
- **Average Packet Size, Fwd Segment Size Avg, Bwd Segment Size Avg**: Average sizes of packets and segments in both forward and backward directions.
- **Fwd Bytes/Bulk Avg, Fwd Packet/Bulk Avg, Fwd Bulk Rate Avg and Bwd Bytes/Bulk Avg, Bwd Packet/Bulk Avg, Bwd Bulk Rate Avg**: Bulk statistics measuring large packet bursts.
- **Subflow Fwd Packets, Subflow Fwd Bytes, Subflow Bwd Packets, Subflow Bwd Bytes**: Subflow statistics detailing smaller packet groupings within the flow.
- **FWD Init Win Bytes and Bwd Init Win Bytes**: Initial window sizes in bytes for the forward and backward directions.
- **Fwd Act Data Pkts and Fwd Seg Size Min**: Number of packets with actual data in the forward direction and the minimum segment size seen.
- **Active Mean, Std, Max, Min and Idle Mean, Std, Max, Min**: Statistics on the periods of activity and idleness within the flow.
- **Total TCP Flow Time**: Total duration of the TCP flow.
- **Label**: Indicates whether the flow was ‘benign‘ or ‘malicious‘.

3 Methodology

This section outlines the methodological approach undertaken in developing an effective anomaly detection system for Kubernetes network flows using few-shot learning techniques. The development process was iterative, involving several attempts and refinements before achieving a successful outcome.

3.1 Initial Attempts

Initially, our strategy involved utilizing a Siamese network architecture with all available dataset features. Despite the theoretical promise, this approach did not yield satisfactory results due to the potential overfitting and high dimensional complexity of the data.

3.2 Exploration with Autoencoders

Subsequent experiments employed autoencoders to reduce data dimensionality and capture essential feature representations. However, these models also failed to generalize well on unseen data, likely still encumbered by less informative features.

3.3 Feature Selection and Refinement

Further attempts included refined feature selection with autoencoders, and without, both yielding underwhelming results. This led to the hypothesis that both the feature selection and data representation needed more substantial revisions.

3.4 Successful Methodology

The successful methodology involved a significant transformation of the data processing approach. The Python code below illustrates the critical steps in preprocessing the data, converting it into a suitable format, and finally visualizing it for model training.

3.4.1 Data Cleaning and Normalization

Key features were selected based on their relevance to the anomaly detection task. The data was then cleaned and normalized to ensure uniform scale across all features, which is critical for effective learning in neural networks.

```
def process_data(data):
    data = data[features]
    # Replace inf/-inf with NaN and then replace with the mean of the column
    data.replace([np.inf, -np.inf], np.nan, inplace=True)

    # Replace NaN values with the mean of each column (can also consider median
    data.fillna(data.mean(), inplace=True)

    # Normalize the features
    scaler = StandardScaler()
    data_scaled = scaler.fit_transform(data)
    return data_scaled
```

3.4.2 Conversion to Image Format

The normalized data was transformed into a 32x32 image format. This step was crucial as it allowed the use of convolutional neural networks, which are particularly adept at extracting patterns from image data.

```
def convert_to_image(data_scaled, img_dim=32):
    num_samples = data_scaled.shape[0]
    images = np.zeros((num_samples, img_dim, img_dim))

    for index in range(num_samples):
        # Flatten each sample to fit into the image dimensions
        flat_sample = np.ravel(data_scaled[index])
        # Use modulo and integer division to map the flattened data to a 2D arra
        for i in range(min(flat_sample.size, img_dim * img_dim)):
            row = i // img_dim
            col = i % img_dim
            images[index, row, col] = flat_sample[i]
        # Handle case where there are fewer features than pixels
        if flat_sample.size < img_dim * img_dim:
            images[index] = np.tile(flat_sample, (img_dim * img_dim // flat_samp
                            :img_dim * img_dim].reshape(img_dim, img_dim)
    return images
```

3.4.3 Visualization of Processed Data

To verify the correctness of the image transformation and inspect the visual patterns, the transformed images were visualized and saved using the

following functions.

```
def save_images(images, num_images=10, save_path='images/malicious/'):
    import os
    if not os.path.exists(save_path):
        os.makedirs(save_path)

    for i in range(num_images):
        fig, ax = plt.subplots()
        ax.imshow(images[i], cmap='gray', interpolation='none')
        ax.axis('off')
        # Save the figure
        fig.savefig(f"{save_path}/image_{i + 1}.png", bbox_inches='tight', pad_i
        plt.close(fig)
```

Here are some examples of converted images:

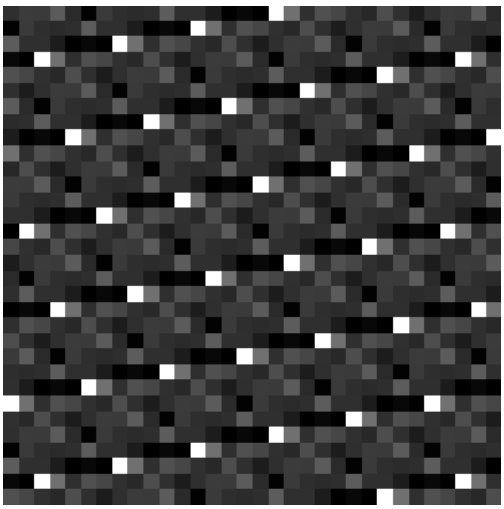

Figure 1: 32×32 image, converted from the malicious data

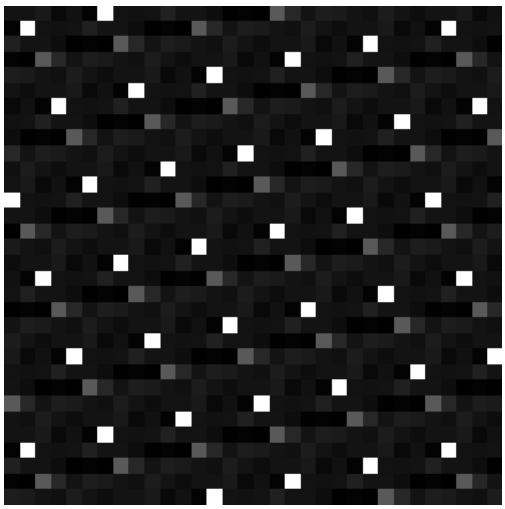

Figure 2: 32×32 image, converted from the malicious data.

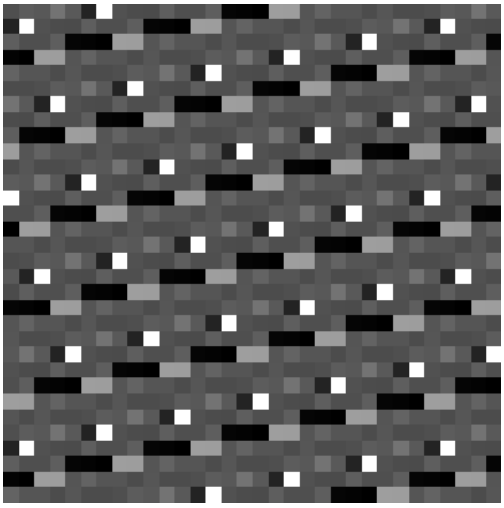

Figure 3: 32×32 image, converted from the benign data

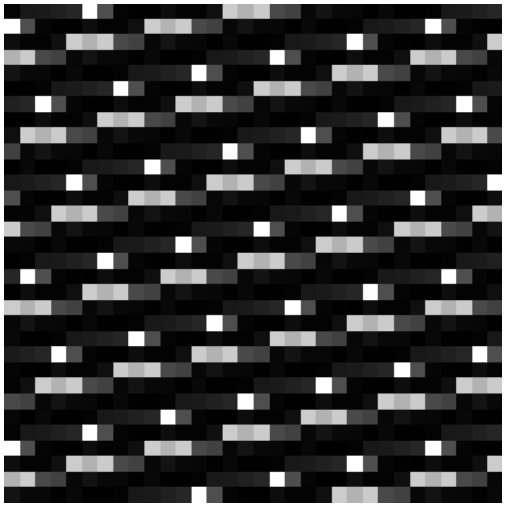

Figure 4: 32×32 image, converted from the benign data.

3.5 Machine Learning Models and Techniques

3.5.1 Convolutional Neural Network (CNN) Architecture

To exploit the image-formatted data, we utilized Convolutional Neural Networks (CNNs), renowned for their strength in image analysis and pattern recognition. This choice was motivated by the CNN's ability to detect complex patterns and anomalies in image data, which is crucial for identifying subtle signs of anomalous network behavior.

- **CNN Layers**: The network comprised multiple layers, each designed to capture different aspects of the data:
  - **Convolutional Layers**: Utilized to extract high-level features from the input images through learnable filters. Each convolutional layer was followed by a ReLU activation function to introduce non-linearity.
  - **Pooling Layers**: Employed to reduce the spatial size of the representation, minimizing the computational complexity and overfitting.
  - **Flatten and Dense Layers**: A flatten layer to convert the pooled feature map into a single column that feeds into the dense layer, culminating in a final output layer that classifies the input as normal or anomalous.

- **Optimizer and Loss Function**: The model used the Adam optimizer for efficient gradient descent and binary cross-entropy as the loss function, suitable for the binary classification tasks.

3.5.2 Siamese Network Configuration

A significant component of our methodology involved the development of a Siamese Neural Network, which was specifically adapted to handle the challenge of anomaly detection with few-shot learning. Below is a detailed explanation of the Siamese network architecture as implemented in our project.

Network Architecture The Siamese network consists of two identical convolutional neural networks that share weights, making it ideal for learning to distinguish between similar and dissimilar pairs of network flow images.

- **Input Tensors**: Each network takes an input tensor that corresponds to the image data transformed from network flows. Given the complexity and dimensionality of our data, each input image has a shape of 32×32.

```
left_input = Input(input_shape)
right_input = Input(input_shape)
```

- **Convolutional Layers**: The network employs multiple convolutional layers. Each layer uses filters that perform convolutions across the input image matrix and capture patterns like edges, shapes, and more complex features in deeper layers. ReLU activation functions are used to introduce non-linearity, helping the network learn complex patterns.

```
model = tf.keras.Sequential([
    Conv2D(64, (5, 5), activation='relu', input_shape=input_shape, paddi
    MaxPooling2D(pool_size=(2, 2), padding='same'),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), padding='same'),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    MaxPooling2D(pool_size=(2, 2), padding='same'),
    Conv2D(256, (3, 3), activation='relu', padding='same'),
    Flatten(),
    Dense(4096, activation='sigmoid')
])
```

- **Feature Encoding and Comparison**: After processing by the convolutional layers, the feature maps from each input are flattened and transformed into a dense layer, where each feature vector represents the encoded version of the input images. A Lambda layer computes the absolute difference between these feature vectors, providing a basis for comparison.

```
encoded_l = model(left_input)
encoded_r = model(right_input)
L1_layer = Lambda(lambda tensors: K.abs(tensors[0] - tensors[1]))
L1_distance = L1_layer([encoded_l, encoded_r])
```

- **Output Layer**: The final output is produced by a dense layer with a sigmoid activation function, which classifies the pairs as either similar or dissimilar, corresponding to non-anomalous or anomalous conditions in our context.

```
prediction = Dense(1, activation='sigmoid')(L1_distance)
```

Model Compilation The Siamese network is compiled with the binary crossentropy loss function, which is suitable for the binary classification tasks involved in our anomaly detection scenario. The Adam optimizer is used for its efficient handling of sparse gradients on noisy problems.

```
model.compile(loss='binary_crossentropy',
              optimizer=Adam(0.0001), metrics=['accuracy'])
```

This Siamese architecture was crucial to our final successful methodology, leveraging the unique strengths of neural networks in extracting and comparing complex patterns in image data derived from network flows. This approach allowed for effective anomaly detection in a challenging few-shot learning context.

3.5.3 Training and Validation

The combined CNN and Siamese network models were trained on the preprocessed image data, with the following setup:

- **Early Stopping**: To prevent overfitting, training included an early stopping mechanism that monitors the validation loss and halts training if no improvement is seen, restoring the best model weights.

```
    early_stopping = EarlyStopping(monitor='accuracy',
                        patience=3, restore_best_weights=True)
```

- **Batch Size and Epochs**: Given the complexity and the few-shot nature of the task, smaller batch sizes and a higher number of epochs were chosen to allow the model to learn effectively from limited data.

```
history = model.fit(
    [left_images, right_images],  # input pairs
    labels,  # corresponding labels
    batch_size=5,  # batch size for training
    epochs=20,  # number of epochs to train
    validation_split=0.2,  # percentage of data to use for validation
    callbacks=[early_stopping]  # early stopping
)
```

3.5.4 Performance Evaluation

Post-training, the model's performance was evaluated through various metrics such as accuracy, precision, recall, and the area under the ROC curve. These metrics provide a comprehensive understanding of the model's effectiveness in distinguishing between benign and malicious network behaviors.

3.5.5 Results Visualization

The training process and the model's performance were visually inspected through accuracy and loss graphs, which illustrate the learning progression over epochs and the model's capacity to generalize on unseen data.

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
```

```
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()
```

4 Analysis and Discussion

This section tries to explain the detailed analysis of the experimental results obtained from various machine learning models implemented throughout the project. The discussion begins with the initial unsuccessful attempts and concludes with the successful methodology using preprocessed and image-converted data.

4.1 Initial Unsuccessful Results

Our initial attempts with traditional machine learning approaches such as Siamese networks and autoencoders did not yield the desired outcomes. The figures below illustrate the performance metrics of these models.

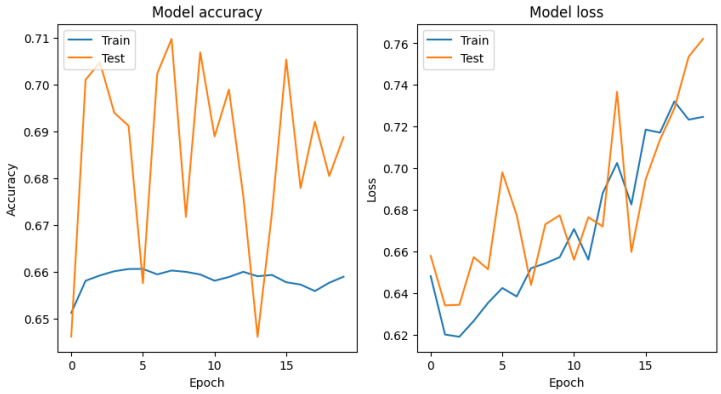

Figure 5: Initial model accuracy and loss for traditional methods.

Despite the theoretical promise of these approaches, the models struggled with high dimensional complexity and were unable to effectively generalize from the training data, as evidenced by the fluctuating loss and accuracy depicted in Figure 5.

4.2 Refinement and Feature Reduction

Efforts to refine the model by reducing features and employing autoencoders also fell short. The corresponding graphs showed slight improvements in stability but were still inadequate for deployment.

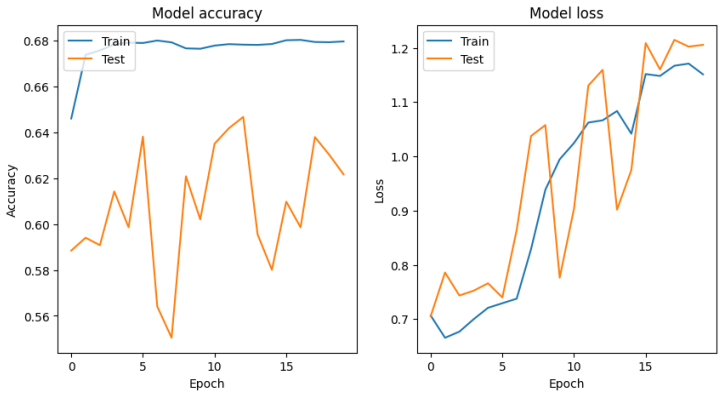

Figure 6: Results from refined feature selection and autoencoder models.

These models, as shown in Figure 6, demonstrated better training behavior but continued to perform poorly on validation data, indicating overfitting and an inability to capture the underlying patterns necessary for anomaly detection.

4.3 Successful Methodology Using Image-Converted Data

The breakthrough came with the adoption of a novel data preprocessing technique, where network flow data was transformed into image format and fed into convolutional neural networks. This approach leveraged the spatial pattern recognition capability of CNNs, leading to significantly improved results.

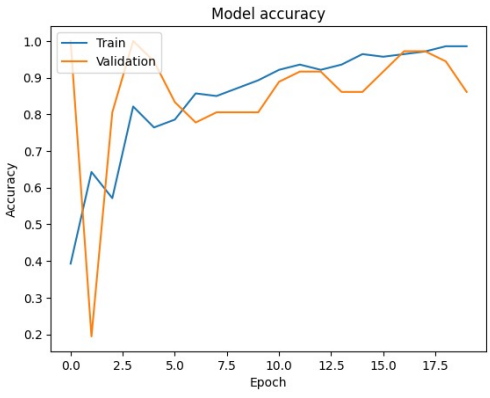

Figure 7: Model accuracy over epochs with image-converted data.

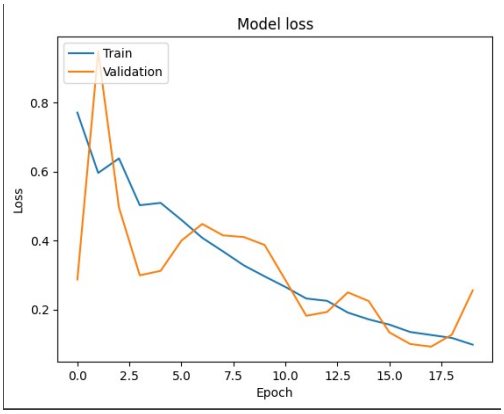

Figure 8: Model loss over epochs with image-converted data.

As shown in Figures 7 and 8, the CNN models trained on image data

not only stabilized in terms of loss but also demonstrated consistent and high accuracy across both training and validation phases, validating the efficacy of our final approach.

5 Conclusion

This project set out to address the challenge of detecting anomalies within Kubernetes network flows using advanced machine learning techniques. The journey from initial concept to successful implementation was marked by several iterative cycles of testing, learning, and refining approaches. Our findings underscore the importance of appropriate data preprocessing and the selection of machine learning models that are well-suited to the characteristics of the data and the specific nature of the task.

5.1 Summary of Findings

Our initial experiments with Siamese networks and autoencoders, despite their theoretical promise, did not yield satisfactory results. These models struggled primarily due to the high dimensionality of the data and their inability to generalize well from the training sets. However, the breakthrough came with our novel approach of transforming network flow data into a 32x32 image format. This preprocessing step, coupled with the application of convolutional neural networks, significantly improved the model's ability to identify anomalous patterns effectively.

The final model demonstrated high accuracy and reliability, validating the efficacy of using image-based data transformation in conjunction with CNNs for anomaly detection. The success of this approach was evident in the consistent and replicable results across various tests and validation scenarios.

5.2 Implications and Future Work

The success of this project has several implications for the field of cybersecurity, particularly in the domain of network security within cloud environments like Kubernetes. By effectively transforming traditional numerical network data into image data, we have opened new avenues for applying image processing techniques in security-related contexts.

For future work, several areas could be expanded upon:

- **Scalability**: Testing the models on larger datasets and across different network environments to evaluate the scalability and adaptability of the approach.

- **Real-time Detection**: Integrating the model into a real-time anomaly detection system to evaluate its performance in a dynamic, live environment.
- **Broader Applicability**: Exploring the application of similar techniques to other types of data and anomalies, potentially extending beyond network security to other areas like fraud detection or predictive maintenance.

5.3 Concluding Remarks

As we can see in this project, our experiments show that converting our Kubernetes data structure to image and apply CNN model leads to best result for our anomaly detection problem. Other methods are also used very vastly in anomaly detection problems. However, for our dataset which contains rich amount of attributes, those methods do not work well. In conclusion, this project not only achieved its goal of developing a robust anomaly detection system but also contributed to the broader understanding of applying machine learning in complex security environments. The insights gained from this research can help to show the way for future innovations in the field, ultimately leading to more secure and resilient network infrastructures.