network

MDPI

Article

Bijective Network-to-Image Encoding for Interpretable CNN-Based Intrusion Detection System

Omesh A. Fernando [1,*], Joseph Spring [1] and Hannan Xiao [2]

[1] Department of Computer Science, University of Hertfordshire, Hatfield AL10 9AB, UK; j.spring@herts.ac.uk
[2] Department of Informatics, King's College London, London WC2R 2LS, UK; hannan.xiao@kcl.ac.uk
* Correspondence: w.k.fernando@herts.ac.uk

Abstract

As 5G and beyond networks grow in heterogeneity, complexity, and scale, traditional Intrusion Detection Systems (IDS) struggle to maintain accurate and precise detection mechanisms. A promising alternative approach to this problem has involved the use of Deep Learning (DL) techniques; however, DL-based IDS suffer from issues relating to interpretation, performance variability, and high computational overheads. These issues limit their practical deployment in real-world applications. In this study, CiNeT is introduced as a novel DL-based IDS employing Convolutional Neural Networks (CNN) within a bijective encoding–decoding framework between network traffic features (such as IPv6, IPv4, Timestamp, MAC addresses, and network data) and their RGB representations. This transformation facilitates our DL IDS in detecting spatial patterns without sacrificing fidelity. The bijective pipeline enables complete traceability from detection decisions to their corresponding network traffic features, enabling a significant initiative towards solving the 'black-box' problem inherent in Deep Learning models, thus facilitating digital forensics. Finally, the DL IDS has been evaluated on three datasets, UNSW NB-15, InSDN, and ToN_IoT, with analysis conducted on accuracy, GPU usage, memory utilisation, training, testing, and validation time. To summarise, this study presents a new CNN-based IDS with an end-to-end pipeline between network traffic data and their RGB representation, which offers high performance and enhanced interpretability through revisable transformation.

Keywords: convolutional neural network (CNN); lossless encoding and decoding; network traffic to images; images to network traffic; intrusion detection system (IDS); computational complexity; TensorFlow; PyTorch; 5G; B5G

Academic Editors: Muddasar Naeem and Antonio Coronato

Received: 11 August 2025
Revised: 5 September 2025
Accepted: 15 September 2025
Published: 25 September 2025

Citation: Fernando, O.A.; Spring, J.; Xiao, H. Bijective Network-to-Image Encoding for Interpretable CNN-Based Intrusion Detection System. *Network* **2025**, *5*, 42. https://doi.org/10.3390/network5040042

1. Introduction

The rollout of 5th Generation (5G) through 3rd generation partnership project (3GPP) releases 15th to 19th [1] has created a standardised foundation for scalable applications with guaranteed performance. Rapid advances towards 5G-Advanced [2,3] further enhance this infrastructure across diverse environments. Examples include Connected and Automated Vehicles (CAVs), the Internet of Things (IoT), and Multi-Access Edge Computing (MEC) [4]. These use cases, which complement primary use cases such as enhanced mobile broadband (eMBB), ultra-reliable low-latency communications (URLLC), and massive machine type communications (mMTC), have placed a heightened demand for secure networks requiring privacy [5], reliability [6], and availability [7].

The exponential growth of connected devices and end users in mobile telecommunications with smart devices is expected to reach 7.95 billion by 2028, from 7.21 billion at the

end of 2024 [8]. Mobile hand-held smart devices are already reported to have generated 1.3 zettabytes of data in 2024 [9]. This exponential growth of users and traffic volume raises an important question: *How do we protect the 5G and beyond from adversarial threats?* The application of traditional Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS), such as Firewalls, has proven to be ineffective in the face of diverse traffic patterns with exponential volumes [10]. The complexity of attacks, sophistication of Advanced Persistent Threats (APTs), and large-scale Distributed Denial of Service (DDoS) attacks, as well as the widespread availability of tools to conduct a sophisticated attack, continue to challenge existing security capabilities [11–13]. As per the research by [13], Machine Learning (ML) methods have proven to be an effective, faster, and more accurate threat detection and prevention mechanism compared to the traditional security solutions in the big-data era [14]. Despite its many advantages in cybersecurity, ML methods alone do possess drawbacks. Adversarial attacks and evasion [15], adaptability limitations, the black box problem, and lack of interpretability [12,16], for example, can be construed as drawbacks. Deep Learning (DL), a subset of ML, has attracted the attention of both academia and industry as a viable method to overcome drawbacks such as the above, with potential to autonomously grasp intricate patterns and connections within data [17]. Similarly, the authors of [18] postulate that DL methods are becoming increasingly essential in cybersecurity solutions and deployments in an attempt to address the fundamental limitations of ML-based solutions.

Recent studies have explored the use of images based on network traffic to leverage the strong feature extraction and extrapolation capabilities offered through the use of CNN, and hence it has become a popular DL model for the intrusion detection problem. Popular methods include converting network traffic to grayscale [19], RGB images [20], and serialised images [21]. Higher accuracy generated by the use of RGB images as opposed to grayscale methods motivated us to develop a new bijective method for image encoding and decoding to ensure that CNN approaches to the IDS problem remain interpretable by security experts post detection, unlike in serialised approaches where reconstruction of the original network data remains inaccessible.

Despite the popularity and the advances in the field of CNN-based IDSs, comparative studies involving computational efficiency and overhead remain unavailable, particularly with respect to the use of TensorFlow and PyTorch frameworks. Differences in constructing computational graphs, hardware utilisation, optimisation strategies, and memory allocation and de-allocation, can significantly impact model performance, training time, testing time, and scalability [22]. Hence, to test the effectiveness of this approach, three popular datasets used in the study of ML/DL for the intrusion detection problem were employed. They are UNSW NB-15 [23], InSDN [24], and ToN_IoT [25].

This paper introduces CiNeT (Classify in Network Transformation), a novel CNN-based IDS for 5G and beyond networks. While CNNs are well-established for intrusion detection, existing approaches face critical challenges, including the 'black-box' nature of deep learning models, limited traceability for forensics analysis, and performance variability stemming from different DL framework choices. The CiNeT algorithm has been developed to address these challenges through integration with the NeT2I (NEtwork Traffic to Images) and I2NeT (Images to NEtwork Traffic) pipeline [26], leveraging the PyTorch and TensorFlow frameworks. This process alleviates the 'black-box' problem when ML/DL models have been applied and facilitates interpretability in detection and results [27]. Additionally, a rigorous comparative study of CiNeT's implementation across PyTorch and TensorFlow provides insights into framework-specific performance variations.

Two variants of the CiNeT algorithm are implemented and compared: CiNeT-TF (developed using TensorFlow) and CiNeT-PT (developed using PyTorch). Our analysis

and evaluation in terms of accuracy, loss, memory utilisation, GPU usage, training, and validation time, and architectural layers are presented in Section 5. The experiments and results collected demonstrate that CiNeT-PT outperformed CiNeT-TF in all use cases for the above metrics. Furthermore, the robustness of NeT2I and I2NeT across theoretical and empirical complexities was analysed to test its application to systems requiring real-time detection.

1.1. Contribution

This paper makes the following contributions to the field of 5G and beyond security:

- An improved NeT2I and I2NeT implementation that encodes network traffic into RGB images and decodes the images back to network traffic in a bijective manner.
- Developed and evaluated a novel Plug-and-Play CiNeT detection algorithm, a CNN-based IDS tailored for resource constrained edge devices for 5G and beyond.
- A comparative study of CiNeT between TensorFlow and PyTorch across varying architectural depths in terms of performance trade-offs, resource usage, scalability, and speed of training and testing.
- Validation of results across UNSW NB-15, InSDN, and TON_IoT datasets.

1.2. Structure of the Paper

The remainder of this paper is organised as follows. Related work is discussed in Section 2. Proposed algorithms in this research are discussed in Section 3, outlining the evaluation metrics in Section 4 and programs used. Results and analysis for the proposed algorithms across datasets are presented in Section 5. The overall results obtained from our research are discussed in Section 6. Concluding remarks, observations, and planned future work are presented in Section 7.

2. Related Work

This section presents a review of the state-of-the-art literature in the field of Network Intrusion Detection Systems (NIDS), datasets, deep learning frameworks, deep learning models, various model architectures, and finally, trends and advancements of image-based representations for network traffic, which lays the foundation of this work.

The development, evaluation, and deployment of IDS based on ML models and the acquired accuracy rely heavily on the dataset. The quality and the realistic nature of network traffic pertaining to the dataset contribute towards the applicability and performance of such NIDS based on ML models. The UNSW NB-15 dataset [23], was introduced and designed to address the limitations and drawbacks found in the KDD Cup 99 dataset, which can be construed as outdated. UNSW NB-15 contained modern network attacks, attack vectors, and a balanced distribution between normal and malicious network traffic. Another dataset used in the context of research into ML models and the NIDS problem is the InSDN dataset, based within a Software Defined Network environment [24]. InSDN provides a large-scale collection of network traffic distributed amongst a multi-class classification of malicious and normal traffic. Finally, the recently collected ToN_IoT [25] dataset has emerged as a comprehensive collection of network traffic in a multi-class classification, a benchmark dataset for training, testing, and evaluating ML models for the NIDS problem. When conducted, a cross-evaluation amongst the three datasets reveals that the ToN_IoT dataset offers a significant performance gain in comparison to the UNSW NB-15 [28] and InSDN [29] datasets, highlighting its applicability and the quality of the data found within the collection. Hence, the ToN _IoT dataset may be considered a benchmark dataset for applying ML models to advance research into NIDSs.

Deep Learning models have demonstrated superiority over ML models for the IDS problem through their ability to reveal complex, hierarchical features and their representations from raw, unprocessed data. Unlike ML models, whose accuracy relies on accurate feature selection and engineering, DL models such as Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN), Artificial Neural Networks (ANNs), and Deep Neural Networks (DNN), for example, facilitate the extraction and identification of patterns in network traffic, increasing their robustness to ever-evolving network attacks. Researchers at [14] demonstrated that a DL-based NIDS outperformed classical ML-based NIDS, achieving higher accuracy and lower false alarms, in sophisticated multi-class attacks [30]. The research in [31] further emphasised that the automatic feature extraction capability facilitated by DL models was a key contributing factor for their superior performance. These studies collectively confirm the validity of applying DL models to the NIDS problem, as opposed to using traditional ML models.

The choice of DL framework can significantly impact the development, training, evaluation, testing, and deployment of an IDS based on a DL model. TensorFlow and PyTorch are the most commonly used frameworks for deploying a model for DL tasks [32]. The extensive community support, ecosystem, and integration into cloud platforms such as Google Colab, Lambda Labs, and Jupyter Notebooks, for example, have contributed to these two frameworks becoming the de facto standards in both academia and industry. TensorFlow, due to its static computational graph, offers high performance and a seamless capacity for integration and deployment. Conversely, PyTorch offers a dynamic computation graph that offers greater flexibility, ease of debugging, and faster prototyping, which have led to this framework's increasing popularity [2,32]. A recent survey [5] highlights the interchange between the two frameworks, PyTorch and TensorFlow, with the choice of framework having the potential to impact their application, particularly for real-time interference, model interpretability, and ease of development and deployment.

Long Short-Term Memory (LSTM) networks have been widely adopted and applied in NIDSs based on DL models. Their ability to model temporal dependencies, the capability to handle the multivariate, and the sequential nature of network traffic have contributed to its wider adoption in intrusion detection problems. Respective research by [33–36] developed IDSs employing LSTM models and tested their accuracy for a variety of datasets. The LSTM models, however, require high resource availability and computational complexity, which can constitute a bottleneck when applied in a real-time system. Also, the sequential nature of LSTM hinders parallelisation, leading to longer training, validation, and testing periods. Furthermore, the research in [33] highlights that LSTMs' struggle with extensive dependencies in network traffic, as well as their reliance on hyper-parameter tuning, can contribute to overfitting and reduced model generalisation.

The adaptation of Large Language Models (LLMs) into various applications across industry and academia has opened an avenue for them to be considered in the intrusion detection problem. Research by [37,38] employed LLMs for exploring and analysing patterns in network traffic generating responses pertaining to network activity (malicious or non_malicious). Despite the adoption of LLMs into the intrusion detection problem, similar to the LSTM models, resource requirements and computation remain extremely high [39]. Inference latency and the 'black-box' nature of LLMs create a bottleneck in their application to IDSs, requiring real-time detection and the rationale for decisions.

Deep Neural Networks and Artificial Neural Networks have been foundational DL models for the intrusion detection problem. Research by [40–43] presented DL models tested across various benchmark datasets to support the usability of DL in NIDSs. Despite the popularity and the foundational work, these models face a tendency to overfit, which can lead to higher rates of false positive alarms. The extensive requirement for hyper-

parameter tuning, the higher computational complexity, and the 'black-box' problem create barriers to their deployment in real-time systems and resource-constrained environments.

Convolutional Neural Networks (CNN) have become a popular choice for IDSs due to high accuracy, precision, and excellent generalisation properties [44]. The work presented in [45] highlighted that the CNN model outperformed ML models such as Random Forest and Support Vector Machine. Furthermore, their work highlighted that employing a 2D CNN model produced superior performance in comparison to 1D CNNs. The same observation has been recorded in the works of [46] in their intrusion detection system, where a 2D CNN model produced superior accuracy. This prompted us to explore a 2D CNN as a viable option for the detection of malicious traffic in the 5G and beyond network infrastructure. Surveys conducted by [47,48] provide a comprehensive overview of the CNN-based IDSs with various architectures, performance, and their applicability for intrusion detection problems. Respective research by [26,49–55] introduces various applications of CNNs to further solidify the usability of CNNs to the intrusion detection problem, emphasising their effectiveness in handling high-dimensional network data. Compared to LSTM and LLM models, CNNs are computationally viable, with a lower cost and smaller memory footprint. CNNs also allow parallel processing, leading to faster training and are suitable for inference times, unlike LSTM models. The ability to handle high-dimensional data, shared weights, and parameter reduction through pooling avoids the overfitting problem, unlike in ANN and DNN models. Finally, CNNs offer greater interpretability avoiding the 'black-box' problem found in other models.

One of the greatest challenges in the application of CNNs to network security involves the process of encoding data into a form recognisable by the CNN. As CNNs accept images for training and testing, data collected from a dataset has to be converted into images. Existing research uses various methods to encode network traffic data into images that can be used to train a CNN algorithm. Research published in [19,56–61] employed grayscale images to represent and encode the desired features of network traffic for training and testing a CNN algorithm. However, representing network traffic by grayscale images can lead to a loss of information since modern network traffic, due to heterogeneity and complexity, can contain data that exceeds a pixel value in the grayscale (0–255).

Due to this limitation on grayscale images, the authors of [20,21,62,63] employed mechanisms to encode network traffic as RGB images. Improvements in accuracy were observed [20,63] when employing RGB images in preference to grayscale images. The authors of [20,62] employed a tiled image approach along the x and y axes, for a given pixel length and width, whereas [21,63] generated serialised RGB images. Although these images are capable of producing a higher accuracy than with grayscale images, RGB can represent a pixel value between 0 and 16,777,215 in a tiled image, which places a high demand on CPU, memory, and time of execution. In comparison, [26] showed that the proposed encoding algorithm, coupled with the CNN, achieved higher detection and better computational complexity than a significant baseline model. Serialised images can be construed as computationally viable, but are not bijective with information being lost during conversion, between network traffic and images. This information loss could potentially classify a CNN-based IDS as a 'black-box' solution. In this work, an improved NeT2I and I2NeT bijective algorithm is proposed for encoding and decoding of network traffic, including IPv6 addresses, MAC addresses, IPv4 addresses, timestamps, and other network features. A novel detection algorithm, CiNeT, is also presented, which is a dynamic plug-and-play algorithm capable of detecting classes and determining related parameters.

3. Proposed Algorithm

In this section, a novel encoding algorithm (NeT2I) is presented that is used to represent network traffic as RGB images, along with a corresponding decoding algorithm (I2NeT) to translate the RGB images back into network traffic. A dynamic detection algorithm (CiNeT) developed using both TensorFlow and PyTorch is also included in this section, which is used to identify observed traffic.

3.1. Encoding and Decoding Network Traffic (NeT2I-I2NeT Pipeline)

The secure, structured, and deterministic transformation of network data traffic into images is achieved through the integration of a NeT2I-I2NeT pipeline, initially introduced in [26]. The pipeline has been significantly developed since its first inception, to now include a comprehensive set of network features that accommodate, for example, IPv4 and IPv6 addresses, timestamps, MAC addresses, floating-point numbers (e.g., flow duration, jitter), and integer numbers (e.g., protocol identifiers, packet count, port numbers, and packet load). These features form a multi-dimensional representation of network traffic and behaviour, for the detection of malicious attacks with advanced complexity.

The pseudo code presented in Algorithm 1 (Encoding Algorithm: NeT2I) describes our method for representing network traffic as PNG images. A generated image from the NeT2I can be seen in Figure 1. The ability to handle IPv6 addresses is a significant new feature of this pipeline, driven by the urgent need for network security solutions to support the global transition to IPv6 [64]. The US Executive Order on furthering the Nation's Cybersecurity and Cyber readiness and the European Union's IPv6 action plan require all public internet assets to be IPv6 ready [65] by the end of 2025. In response to this, our NeT2I-I2NeT pipeline provides a meticulous, lossless, and deterministic method for encoding and decoding IPv6 addresses. To achieve this each 128-bit value is converted to its packed binary form. The 16-byte sequence is padded with two additional zero bytes to form an 18-byte array, compatible with a 3-byte RGB grouping, to form six consecutive RGB pixel tuples, with values ranging from 0 to 255. Each of these now corresponds to one horizontal stripe in the generated image. This approach ensures that each IPv6 address that is to be mapped and encoded to an RGB image does so in a lossless manner, maintaining full address fidelity. The NeT2I-I2NeT pipeline handles date-time fields with a deterministic approach, ensuring lossless integrity. Each timestamp (date-time) is decomposed into six components. They are year (Y), month (M), day (D), hour (H), minute (M), and seconds (S). The values are converted to float and form twelve RGB pixels per timestamp. This ensures full temporal fidelity and enables accurate reconstruction during the decoding stage.

During the encoding phase, NeT2I automatically generates a JSON companion file containing metadata to ensure accurate and unambiguous data reconstruction. This file meticulously documents the structure of the network traffic in the input CSV file. The I2NeT decoder, as described in Algorithm 2 (Decoding Algorithm: I2NeT), reverses this process by reading the JSON metadata file by identifying six consecutive RGB pixels from the PNG image, with each pixel representing three of the 18-byte padded binary representation. The first 16 bytes are interpreted as the packed binary of the IPv6 address, allowing an accurate reconstruction of the original IPv6 address in a standardised colon-separated hexadecimal format. In order to ensure robustness at the decoding stage, images generated from IPv6 addresses are tagged with an additional prefix (e.g., *ipv6_1022.png*). This ensures that in a scenario where the JSON file is not available, I2NeT can default to the appropriate path for decoding. The structured, type-aware, and error-resilient approach ensures that I2NeT achieves high fidelity for the recovery of IPv6 addresses.

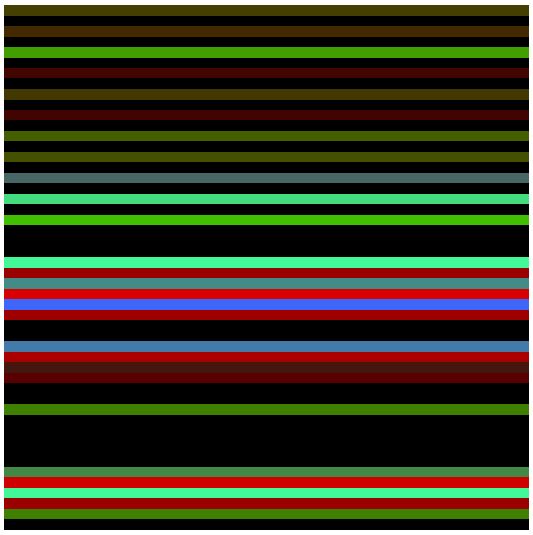


Figure 1. A Visual representation of an image generated via NeT2i.

In the NeT2I algorithm, floating-point numbers are mapped to image data using an invertible binary serialisation approach that ensures both accuracy and integrity. Initially, a floating-point number is converted to its 32-bit IEEE 754 [66] binary representation, ensuring network byte-order consistency. This 4-byte floating-point number is then represented in two RGB pixels. The first RGB pixel carries the first three bytes (RGB), and the second pixel encodes the fourth byte in its Red channel, setting the Green and Blue channels to zero. This deterministic mapping sequence allows the decoder in the I2NeT algorithm to accurately reconstruct the original floating-point number from the RGB representation, correctly reassembling the byte sequence in a lossless manner. Prior to the NeT2I encoding phase, unstructured string data, such as free-form text that may appear in the CSV, are removed as they can not be encoded in a bijective manner. To ensure robustness and fault tolerance, unstructured string data are handled using a dedicated mechanism that activates `if` the residual string-like data may cause pipeline failures, thus prioritising system stability and continuity. The metadata is stored in the JSON file to aid at the decoding stage.

The I2NeT decoder captures and reconstructs floating-point numbers from images through a deterministic process that ensures accuracy, integrity, and fidelity. During the decoding phase, I2NeT extracts the RGB stripe from the horizontal bands in the image and processes each pair of pixels to recover and reconstruct the original 4-byte float structure. The original floating-point number is reconstructed by concatenating these four bytes accurately, following the byte order. The I2NeT decoder follows type-aware reconstruction employing the metadata from the JSON file to determine fields which are floating-point numbers. In a scenario where the JSON file is not available, I2NeT uses a default generic decoding mechanism, ensuring a robust recovery. Similarly, I2NeT also follows bound checking, exception handling, and error resilience features to manage potential mismatches or corrupted data.

This end-to-end pipeline of NeT2I for encoding and I2NeT for decoding creates a bijective transformation between structured data and the corresponding image representation. The latest version of NeT2I can be found in the GitHub repository [67] and in the Python Package Index (PyPI) [68]. The latest version of I2NeT can be found in the GitHub repository [69] and in the Python Package Index (PyPI) [70]. By integrating support for modern network features such as IPv6 and floating-point numbers, the NeT2I-I2NeT pipeline provides a robust, scalable, future-proof, errorless, and deterministic solution for securing network data transmission.

Algorithm 1: Encoding Algorithm: NeT2I

```
Input: input.csv
Output: image.png
Function encodeCSVToRGB(csv_path):
    Data: Loaded CSV data
    Data: Detected data types per column (Float, IPv4, IPv6, MAC, DateTime, String)
    Data: Processed RGB pixel array
    /* Step 1: Clean output directory and temporary files */
    cleanOutputDirectory()
    /* Step 2: Split CSV into IPv4 and IPv6 datasets */
    ipv4_data, ipv6_data ← splitCSVByIPType(csv_path)
    /* Step 3: Process each dataset separately */
    processDataset(ipv4_data, is_ipv6=False)
    processDataset(ipv6_data, is_ipv6=True)
    /* Step 4: Save type information for decoding */
    saveTypeInformation()
    /* Step 5: Generate images from RGB pixel arrays */
    generateImages()
Function convertDataToRGB(row, types):
    Data: List of RGB pixel tuples per row
    rgb_row ← [ ]
    foreach value, dtype in zip(row, types) do
        switch dtype do
            case "DateTime" do
                comps ← [Y, M, D, H, M, S] from parse(value)
                rgb1, rgb2 ← floatToTwoRGB(float(comp))
            case "IPv6 Address" do
                rgb_pixels ← convertIPv6ToRGB(value)
            case "IPv4 Address" do
                float_val ← convertIPv4ToFloat(value)
            case "MAC Address" do
                float_val ← convertMACToFloat(value)
            case "Float" do
                rgb1, rgb2 ← floatToTwoRGB(value)
            case "String" do
                /* Ensure System Continuity */
                hash_val ← hashStringToFloat(value)
            otherwise do
                Error: Unknown data type
            end
        end
    end
    return rgb_row
Function createImageFromRGB(rgb_row, image_id, prefix):
    initializeImageArray()
    /* Fill image with RGB pixel stripes */
    rows_per_color ← calculateStripeHeight(rgb_row)
    current_row ← 0
    foreach rgb in rgb_row do
        (r, g, b) ← normalizeRGB(rgb)
        for i = 0 to rows_per_color do
            if current_row < image_size then
                fillRow(current_row, r, g, b)
                current_row ← current_row + 1
            end
        end
    end
    /* Fill remaining rows with last colour if needed */
    while current_row < image_size do
        fillRow(current_row, r, g, b)
        current_row ← current_row + 1
    end
    return saveImage(image_id, prefix)
```

Algorithm 2: Decoding Algorithm: I2NeT

```
Input: input.png (directory or single image)
Output: output.csv
Function decodeImagesToCSV(image_path):
    /* Step 1:  Auto-detect IP version and load type info                    */
    autoDetectIPVersion(image_path)
    /* Step 2:  Extract RGB pixel values                                     */
    rgb_values ← extractRGBFromImage(image_path)
    /* Step 3:  Detect expected data structure                               */
    adaptive_type_info ← detectDataStructure(len(rgb_values), type_info)
    /* Step 4:  Reconstruct structured values                                */
    decoded_values ← reconstructValues(rgb_values, adaptive_type_info)
    /* Step 5:  Write to CSV                                                 */
    writeToCSV(decoded_values)
      return output.csv

Function extractRGBFromImage(mage_path):
    /* Extract RGB pixel values using stripe detection                      */
    img = openImage(image_path)
      array = convertToNumpyArray(img)
      stripe_colors = detectColorStripes(array)
      return stripe_colors

Function detectDataStructure(rgb_count, type_info):
    if type_info is available then
        calculateExpectedPixelCount(type_info)
        if rgb_count < expected_count then
            return truncated_types
        end
        else
            return type_info
        end
    end
    else
        /* Fallback:  assume 2 pixels per float                             */
    end

Function reconstructValues(rgb_values, adaptive_type_info):
    for orig_type in original_types do
        if orig_type == "Date Time" then
            comps ← parseDateTimeToComponents(value)
              datetime ← float(comps)
              reconstructed.append(datetime)
        end
        if orig_type == "IPv6 Address" then
            ipv6 ← decodeIPv6From6Pixels(rgb_values)
            reconstructed.append(ipv6)
        end
        else if orig_type == "IPv4 Address" then
            ipv4 ← decodeIPv4From6Pixels(rgb_values)
            reconstructed.append(ipv4)
        end
        else if orig_type == "MAC Address" then
            mac ← formatMACFromChunks(chunk1, chunk2)
            reconstructed.append(mac)
        end
        else if orig_type == "Float" then
            float_val ← decodeFloatFrom2Pixels(rgb_values)
            reconstructed.append(float_val)
        end
        else if orig_type == "String" then
            hash_val ← decodeHashFromPixel(rgb_values)
            reconstructed.append(str(hash_val))
        end
        else
            /* Error                                                        */
        end
    end
    return reconstructed
Function mergeComponents(values):
    /* Merge [Y,M,D,H,M,S] → "YYYY-MM-DD HH:MM:SS"                          */
    /* Reconstruct IPv4/IPv6 and MAC from components                        */
    /* Handle partial or invalid sequences gracefully                       */

Function writeToCSV(decoded_values):
    /* Write decoded values to CSV file                                     */
    createNewCSVFile()
    for row in decoded_values do
        writeRowToCSV(row)
    end
    return output.csv
```

3.2. Detection Algorithm (CiNeT)

The CiNeT algorithm conducts a multi-class classification of network traffic by leveraging DL techniques. This algorithm has been developed and designed to integrate seamlessly with the NeT2I and I2NeT frameworks. CiNeT is implemented using both TensorFlow and PyTorch to enable cross-framework evaluation. This method allows for comparative analysis on model performance, training efficiency, accuracy, loss convergence, and computational complexity. The TensorFlow variant of CiNeT employs a CNN architecture with three convolutional blocks, whereas the PyTorch variant employs four convolutional blocks, with batch normalisation and dropout for robustness. The model is trained on images generated via NeT2i, and classification accuracy is validated by reconstructing predictions via I2NeT, ensuring a seamless encoding–classification–decoding pipeline of a full-fledged DL-based IDS.

The CiNeT algorithm features an autonomous and adaptive class detection mechanism that enables automation without requiring manual intervention or the specification of output classes. During the initialisation phase CiNeT scans the input directory and subdirectories within and passes the sub-directory names to dynamically construct a list of class labels and assign a unique index to each. The detected class count is used to configure the CNN, setting the output units to K, where K denotes the number of classes. This is used to select the appropriate loss function. For $K = 2$, binary cross-entropy is used, and for $K > 2$, categorical cross-entropy is used.

In addition to autonomous class detection, CiNeT employs a data-driven mechanism to select the optimal batch size. This decision is based on the total number of training samples. The algorithm determines the best candidate batch size from a predefined set of rules to reduce underutilisation and thereby improve training efficiency. Along with adaptive class detection and automated batch size selection, CiNeT is a highly scalable and efficient plug-and-play classification system that integrates seamlessly with NeT2I and I2NeT for encoding and decoding. The latest version of our CiNeT variants can be found in the GitHub repository [71].

3.2.1. TensorFlow

The CiNeT algorithm implemented within this ecosystem, which is implemented in TensorFlow, performs multi-class classification using a CNN architecture, employing three sequentially stacked convolutional blocks. The pseudo code can be found at Algorithm 3 (CiNeT-TF Algorithm). Each of these comprises a Conv2D layer with ReLU activation, followed by max-pooling, and dropout for spatial downsampling and regularisation. The input layer accepts images in RGB format specifying the three colour channels and the size of 150 pixels ($150 \times 150 \times 3$). The size corresponds to the output image generated via NeT2I. The TensorFlow variant of CiNeT progressively extracts features through increasing filter depth of 32, 64, and 128, respectively, per layer. Regularisation is achieved by a 25% dropout upon each max-pooling operation, while the fully connected layer includes a 512-unit dense layer with a 50% dropout to ensure overfitting is mitigated.

CiNeT leverages the Keras API for model creation, complication, training, and evaluation. The RMSProp optimiser was utilised with a learning rate set for 1×10^{-4} with an adaptive loss function based on the class count (binary cross-entropy for two classes or categorical cross-entropy for multi-classes). Data augmentation was applied during training to improve generalisation and avoid overfitting with random rotation, shifting, shearing, zooming, and flipping across the horizontal plane. Performance validation was conducted using a test set (15%), with training and validation at 70% and 15%, respectively. The confusion matrix consisted of accuracy, F1-score, recall, and precision, these being generated using the results obtained from the test set.

Algorithm 3: CiNeT-TF Algorithm (3-Layer)

```
Input: Directory with class folders (e.g., data_A, data_B)
Output: Trained model (.h5), accuracy, confusion matrix
Function main():
    class_names ← discoverClasses("/home/ubuntu/Images/")
    num_classes ← len(class_names)
    createDirs("{training,validation,testing}")
    for cls in class_names do
        splitData("data_" + cls, f"training/{cls}", f"validation/{cls}", f"testing/{cls}")
    end
    model ← buildModel(num_classes)
      trainGen, valGen, testGen ← createLoaders(selectBatchSize(), num_classes)
    history ← train(model, trainGen, valGen, epochs=100)
      evaluation ← evaluateModel(model, testGen, num_classes)
      saveModel(model, "cinet_tf_model")
      return {success=True, test_accuracy=evaluation.accuracy}
Function discoverClasses(dir):
    for item in listDir(dir) do
        if isDir(item) and startsWith(item,"data_") then
            classes.append(removePrefix(item,"data_"))
        end
    end
    return sorted(classes)
Function splitData(src, train, val, test):
     files ← shuffle(nonEmptyFiles(src))
       splitAndCopy(files, [0.7, 0.15, 0.15], [train, val, test])
Function buildModel(n):
    model ← tf.keras.Sequential([
       Conv2D(32,(3,3),padding='same',input_shape=(150,150,3)), BatchNormalization(),
       Activation('relu'), MaxPooling2D(2), Dropout(0.25),
       Conv2D(64,(3,3),padding='same'), BatchNormalization(), Activation('relu'),
       MaxPooling2D(2), Dropout(0.25),
       Conv2D(128,(3,3),padding='same'), BatchNormalization(), Activation('relu'),
       MaxPooling2D(2), Dropout(0.25),
       Flatten(), Dense(512), BatchNormalization(), Activation('relu'), Dropout(0.5)
       ])
    if n == 2 then
        model.add(Dense(1, activation='sigmoid'))
        loss_fn ← 'binary_crossentropy'
    else
        model.add(Dense(n, activation='softmax'))
        loss_fn ← 'categorical_crossentropy'
    end
    model.compile(optimizer=RMSprop(1e-4), loss=loss_fn, metrics=['accuracy'])
      return model
Function createLoaders(bs, num_classes):
    class_mode ← 'binary' if num_classes==2 else 'categorical'
    trainGen ← ImageDataGenerator(rescale=1./255, rotation_range=20,
       width_shift_range=0.2, height_shift_range=0.2, horizontal_flip=True,
       brightness_range=[0.8,1.2], zoom_range=0.2, fill_mode='nearest')
    valTestGen ← ImageDataGenerator(rescale=1./255)
    trainFlow ← trainGen.flow_from_directory("training/", (150,150), bs, class_mode, True)
      valFlow ← valTestGen.flow_from_directory("validation/", (150,150), bs, class_mode, False)
      testFlow ← valTestGen.flow_from_directory("testing/", (150,150), bs, class_mode, False)
    return trainFlow, valFlow, testFlow
Function train(model, trainGen, valGen, epochs):
    callbacks ← [
       EarlyStopping(monitor='val_loss', patience=15, restore_best_weights=True),
       ModelCheckpoint('best_model.h5', monitor='val_accuracy', save_best_only=True),
       ReduceLROnPlateau(monitor='val_loss', factor=0.5, patience=7, min_lr=1e-7)
       ]
    history ← model.fit(trainGen, epochs=epochs, validation_data=valGen,
       callbacks=callbacks, verbose=1)
       return history
Function evaluateModel(model, testGen, num_classes):
    testGen.reset()
      test_loss, test_acc ← model.evaluate(testGen, verbose=0)
      predictions ← model.predict(testGen)
    true_labels ← testGen.classes
    if num_classes == 2 then
        pred_labels ← (predictions > 0.5).astype(int).flatten()
    else
        pred_labels ← np.argmax(predictions, axis=1)
    end
    return {accuracy=test_acc, predictions=pred_labels, targets=true_labels}
Function saveModel(model, name):
    model.save(f"{name}.h5")
      saveMetadata({'architecture': 'CiNeT-TF-3Layer', 'classes': class_names}, f"{name}_meta.json")
```

3.2.2. PyTorch

The second variant of the CiNeT algorithm was implemented using PyTorch, performing multi-class classification leveraging CNN. The pseudo code can be found at Algorithm 4 (CiNeT-PT Algorithm). The optimal model architecture is composed of 4 sequentially stacked convolutional blocks, each consisting of a Conv2D layer, batch normalisation, ReLU activation, max-pooling, and dropout for regularisation. Similar to the TensorFlow variant, the input layer accepts RGB images of three channels ($150 \times 150 \times 3$). The PyTorch variant of CiNeT progressively extracts features through increasing filter depths of $32, 64, 128$, and 256, respectively, per layer. Regularisation is achieved by a 25% dropout upon each max-pooling operation, while the fully connected layer includes 512 neurons and a 50% dropout to ensure overfitting is mitigated, before the final classification layer.

RMSProp optimisation is employed with a learning rate of 1×10^{-4}, where the loss function is based on the number of classes. BCEWithLogitsLoss for binary classes and CrossEntropyLoss for multi-classes were chosen. Random rotation, horizontal flipping, shifting, shearing, and zooming under augmentation were used to increase input diversity and to reduce overfitting. Performance validation was conducted using a test set (15%), with training and validation accruing for 70% and 15%, respectively. The confusion matrix, comprising accuracy, F1-score, recall, and precision, was generated using the results collated from the test set.

Algorithm 4: CiNeT-PT Algorithm (4-Layer)

```
Input: Directory with class folders (e.g., data_A, data_B)
Output: Trained model (.pth), accuracy, confusion matrix
Function main():
    class_names ← discoverClasses("/home/ubuntu/Images/")
      num_classes ← len(class_names)
      createDirs("{training,validation,testing}")
    for cls in class_names do
        splitData("data_" + cls, f"training/{cls}", f"validation/{cls}", f"testing/{cls}")
    end
    model, criterion, optimizer ← buildModel(num_classes)
      trainLoader, valLoader, testLoader ← createLoaders(selectBatchSize())
    best_val_acc ← 0.0
    for epoch ← 1 to 100 do
        train_loss, train_acc ← trainEpoch(model, trainLoader, optimizer, criterion)
          val_loss, val_acc ← valEpoch(model, valLoader, criterion)
          if val_acc > best_val_acc then
              best_val_acc ← val_acc
              saveModel(model, "best_model.pth")
        end
    end
    test_acc, preds, targets ← evalTest(model, testLoader)
Function discoverClasses(dir):
    for item in listDir(dir) do
        if isDir(item) and startsWith(item,"data_") then
            classes.append(removePrefix(item,"data_"))
        end
    end
    return sorted(classes)
Function splitData(src, train, val, test):
    files ← shuffle(nonEmptyFiles(src))
      splitAndCopy(files, [0.7, 0.15, 0.15], [train, val, test])
Function buildModel(n):
    model ← nn.Sequential(
      nn.Conv2d(3,32,3,padding=1), nn.BatchNorm2d(32), nn.ReLU(),
      nn.MaxPool2d(2), nn.Dropout(0.25),
      nn.Conv2d(32,64,3,padding=1), nn.BatchNorm2d(64), nn.ReLU(),
      nn.MaxPool2d(2), nn.Dropout(0.25),
      nn.Conv2d(64,128,3,padding=1), nn.BatchNorm2d(128), nn.ReLU(),
      nn.MaxPool2d(2), nn.Dropout(0.25),
      nn.Conv2d(128,256,3,padding=1), nn.BatchNorm2d(256), nn.ReLU(),
      nn.MaxPool2d(2), nn.Dropout(0.25),
      nn.Flatten(), nn.Linear(256×9×9, 1024), nn.BatchNorm1d(1024),
      nn.ReLU(), nn.Dropout(0.5), nn.Linear(1024, n))

    criterion ← nn.BCEWithLogitsLoss() if n==2 else nn.CrossEntropyLoss()
    optimizer ← optim.RMSprop(model.parameters(), lr=1e-4)
    model.to(device)
    return model, criterion, optimizer
```

Algorithm 4: *Cont.*

```
Function createLoaders(bs):
    trainTransforms ← transforms.Compose([Resize(150,150), RandomRotation(20),
      RandomHorizontalFlip(), ColorJitter(0.2,0.2), ToTensor(), Normalize()]);
      valTransforms ← transforms.Compose([Resize(150,150), ToTensor(), Normalize()]);
    trainLoader ← DataLoader(ImageFolder("training/", trainTransforms), bs, True);
      valLoader ← DataLoader(ImageFolder("validation/", valTransforms), bs, False);
      testLoader ← DataLoader(ImageFolder("testing/", valTransforms), bs, False);
      return trainLoader, valLoader, testLoader;
Function trainEpoch(model, dataloader, optimizer, criterion):
    model.train();
    running_loss ← 0, correct ← 0, total ← 0;
    foreach (data, target) in dataloader do
        data, target ← data.to(device), target.to(device);
          optimizer.zero_grad(), output ← model(data);
        loss ← criterion(output, target);
          loss.backward();
        optimizer.step();
          running_loss += loss.item() ;
        , predicted ← torch.max(output, 1);
          total += target.size(0);
        correct += (predicted == target).sum().item();
    end
    return running_loss/len(dataloader), 100*correct/total;
Function valEpoch(model, dataloader, criterion):
    model.eval();
    running_loss ← 0, correct ← 0, total ← 0;
    With torch.no_grad() foreach (data, target) in dataloader do
        data, target ← data.to(device), target.to(device);
          output ← model(data), loss ← criterion(output, target);
          running_loss += loss.item(), predicted ← torch.max(output, 1);
          total += target.size(0);
        correct += (predicted == target).sum().item();
    end
    return running_loss/len(dataloader), 100*correct/total;
Function evalTest(model, dataloader):
    model.eval(), predictions ← [], targets ← [], correct ← 0, total ← 0;
    With torch.no_grad() foreach (data, target) in dataloader do
        output ← model(data.to(device)), predicted ← torch.max(output, 1);
          predictions.extend(predicted.cpu().numpy()), targets.extend(target.numpy());
          total += target.size(0), correct += (predicted.cpu() == target).sum().item();
    end
    return 100*correct/total, predictions, targets;
Function saveModel(model, path):
    torch.save({'model_state_dict': model.state_dict(), 'architecture': 'CiNeT-PT-4Layer'}, path);
```

4. Evaluation Metrics for the Algorithms

4.1. Workflow and Datasets

Among publicly available datasets, InSDN contains the most recently collected data for malicious and non-malicious traffic. The dataset contains multi-classes and has been collated on a Software Defined Networking environment with 80 features; 18% of the dataset consists of malicious network data. The ToN_IoT dataset is another comprehensive dataset consisting of multi-class traffic along with malicious and non-malicious traffic. The dataset contains 44 features with approximately 15% of the dataset classified as malicious. Finally, the UNSW-NB 15 dataset contains 49 features, with 12% of the available traffic in the dataset corresponding to malicious traffic. The workflow of the proposed algorithms and their applications on the datasets is shown in Figure 2. Due to class imbalance in the datasets, to ensure fairness in representation, the training, validation, and testing datasets were generated using stratified sampling to ensure class distribution. Furthermore, class-weighted loss functions were also applied during the training stage. Thus, improving model fairness and performance on minority classes. CSV files were extrapolated from the datasets and used as input to the NeT2I algorithm, generating PNG images based on the network traffic. The generated PNG images are then input into the CiNeT algorithm. Following detection by the CiNeT algorithm, the I2NeT algorithm is used to decode the images back to a CSV file in order to evaluate the accuracy of our detection algorithm. The following Table 1 shows the selected features for each dataset, through Recursive Feature Elimination with Random Forest due to its ability to handle data with high dimensions and its ability to handle non-linear complex interactions [72].

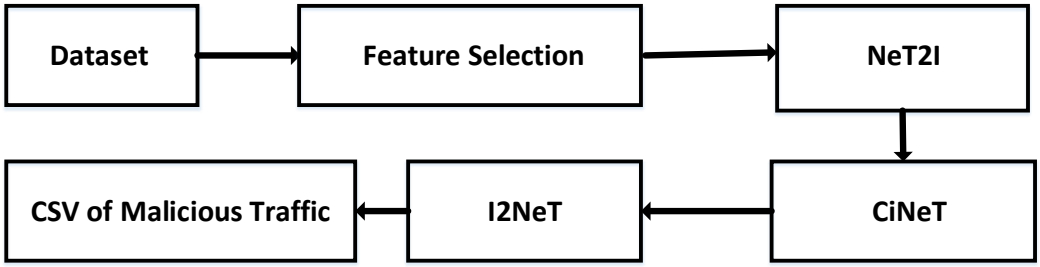


Figure 2. Workflow of the proposed algorithms.

Table 1. Features selected from each dataset.

Dataset	Selected Features
In-SDN	f2, f3, f4, f5, f6, f8, f9, f10, f11, f12, f13, f15, f19, f21, f22, f23, f24, f27, f31, f68, f78
UNSW NB-15	f1, f2, f4, f5, f6, f7, f10, f11, f16, f17, f18, f19, f21, f24, f25, f26, f27, f28, f38, f40, f45
ToN IoT	f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f16, f17, f18, f19, f20, f21

4.2. Theoretical and Empirical Performance Analysis

The NeT2I and I2NeT algorithms were evaluated with respect to execution time, CPU usage, memory utilisation, and time complexity in terms of the Big-O notation, supporting both theoretical and empirical performance analysis.

4.3. Theoretical Analysis

For our theoretical evaluation of the computational complexity, the Big-O notation framework is applied for NeT2I, I2NeT, and CiNeT algorithms. As per the study of [73], the application of a theoretical evaluation using this framework provides a characterisation of the algorithm as a function of the input size. The encoding to detection to decoding processes within the proposed pipeline involve a deterministic transformation that encompasses problems that can be solved in polynomial time. This efficiency is critical for both desk approaches and real-time applications; requiring predictable and scalable performance is essential [73].

4.4. Reproducibility and Implementation Details

Table 2 provides the information necessary to reproduce our experiments to ensure credibility and transparency. All experiments were conducted with a random seed of 42 to ensure deterministic outcomes by eliminating stochastic variability. The models were trained to 100 epochs employing the RMSProp optimiser using a learning rate of 1×10^{-4}. To avoid overfitting and increase generalisation of the training dataset, a series of data augmentation techniques was applied. NeT2I generated images that are structured and deterministic; the application of a standard vision-based augmentation process ensured a prevention of overfitting to spatial patterns or orientations. The following were included in the augmentation pipeline for the training dataset.

- RandomRotation ($\pm 40^{\circ}$)
- RandomHorizontalFlip (0.5)
- RandomAffine (translate = (0.2, 0.2), scale = (0.8, 1.2), shear = 0.2)
- ColorJitter (brightness = 0.2, contrast = 0.2, saturation = 0.2, hue = 0.1)

Table 2. Specifications of hyperparameters, software versions, and data augmentation settings for reproducibility and implementation.

Aspect	Details
Deep Learning Framework	PyTorch 2.1.0, TensorFlow 2.15.0
CUDA/cuDNN	CUDA 12.2, cuDNN 8.9
Python Version	3.10.12
Random Seeds	42 for all experiments
Learning Rate	1×10^{-4} (RMSProp)
Batch size	Selected dynamically based on dataset size
Epochs	100
Data Augmentation	RandomRotation RandomHorizontalFlip RandomAffine ColorJitter
Training/Validation/Test	70%, 15%, 15%
Kernel Size	3×3
Pooling Kernel	2×2
Loss Function (TF)	Binary_crossentropy/Categorical Cross _Entropy
Loss Function (PT)	BCEWithLogitsLoss/CrossEntropyLoss
Class Imbalance Handling	Class-weighted loss (inverse frequency weighting)

The Augmentation process was not applied to the validation and testing datasets to ensure an unbiased evaluation. The batch size was selected dynamically based on the dataset size, adhering to the following logic to balance GPU utilisation and memory efficiency.

- *If* `total_images` $<$ 100, `candidates`: {4, 8, 16}
- *If* 100 $\leq$ `total_images` $<$ 500, `candidates`: {8, 16, 32}
- *If* 500 $\leq$ `total_images` $<$ 2000, `candidates`: {16, 32, 64}
- *If* 2000 $\leq$ `total_images` $<$ 5000, `candidates`: {32, 64, 128}
- *Otherwise*, `candidates`: {64, 128, 256}

Finally, models used 3×3 convolutional kernel 2×2 max-pooling layers with a loss function chosen based on the classification problem

- Binary: `binary cross-entropy` (TensorFlow)/`BCEWithLogitsLoss` (PyTorch)
- Multi-class: `categorical cross-entropy` (TensorFlow)/`CrossEntropyLoss` (PyTorch)

4.5. Empirical Computational Complexity

As discussed in [73], identifying accurate empirical measurements of an algorithm's execution time and memory utilisation is both important and essential, as theoretical complexity alone may not reflect accurate performance due to memory allocation, de-allocation, system-level overheads, and I/O operations. To accommodate this, execution time, CPU usage, memory utilisation, and GPU utilisation are collected and collated.

4.5.1. Execution Time

For measuring the time of execution, the Python `time` library was employed. Variation in execution time can affect performance and efficiency; this is particularly important when applying the algorithm to an intrusion detection problem based in an environment with low resources. To ensure reliability average execution time is presented in Section 5 with each experiment being repeated over multiple runs ($n = 10$), to mitigate caching and Just-In-Time (JIT) compilation effects [74].

4.5.2. CPU Usage

To monitor CPU usage the Python `psutil` library was employed. The library was utilised to monitor and sample CPU usage at one-second intervals during the execution of NeT2I and I2NeT. To ensure statistical reliability, the average of multiple runs ($n = 10$) was recorded. The experiments were conducted in an environment with the Global Interpreter Lock, which restricts code to be executed as a single thread. This restriction is important to ensure that the proposed algorithms can be executed in environments with low-processing power [75] such as Multi-Access Edge Computing (MEC) nodes.

4.5.3. Memory Utilisation

As discussed in the work presented by [76], memory allocation in a modern operating system is inherently imprecise compared to CPU usage or execution time. This is due to over-allocation, caching, and ineffective garbage collection. To monitor the memory consumption across NeT2I, I2NeT, and CiNeT, the Python `memory-profiler` library was employed at 1-s intervals. Computer systems often over-allocate memory to minimise the allocation and deallocation frequency, and garbage collection does not occur instantaneously. Due to this non-deterministic behaviour, averages of multiple independent runs ($n = 10$) were recorded, to ensure a more reliable and comparable indicator.

4.5.4. GPU Utilisation

The CiNeT algorithm was designed to leverage GPU acceleration, while NeT2I and I2NeT were designed to be CPU bound. To determine the GPU compute usage, and memory allocation, the `nvidia-smi` and CUDA interface were employed. GPU utilisation was measured using `pynvml` at predefined stages (e.g., epoch start, data loading) for GPU monitoring. The experiments were conducted in a controlled environment with access to a single GPU to emulate an edge-launched intelligent IDS. As highlighted in [39], DL executions exhibit suboptimal GPU utilisation despite resource allocation and availability. This leads to inefficient compute usage and prolonged execution times. By measuring utilisation, we ensure that CiNeT achieves high computational throughput and that it effectively leverages the parallel processing capabilities of the GPU. GPU utilisation is a key indicator for algorithmic efficiency. High and sustained utilisation reflects minimal idle time and effective kernel execution. Low and erratic usage may indicate underutilisation, poor resource management and orchestration, despite high accuracy. Therefore, GPU monitoring provides a valuable and necessary statistic collected over multiple independent runs ($n = 10$), to ensure performance gains are not achieved at the cost of excessive hardware dependency or energy consumption.

5. Results and Analysis

Data was collected on a UVT_Cloud deployment running Ubuntu 22.04 LTS, with a single CPU, 8 GB of RAM and 20 GB of HDD space for the NeT2I and I2NeT algorithms. The CiNeT algorithm was trained, validated, and tested using an NVIDIA H100 GPU (96 GB VRAM) within a system based on the ARM64 architecture. Two hundred twenty-five thousand images representing network traffic that belong to various malicious and non-malicious classes were selected, with images being subsequently grouped following the ratio of 70:15:15, for training, validation, and testing, respectively.

5.1. Encoded Images

Figure 3 represents images generated from distinct lines of network traffic in the input CSV. The image consists of one-dimensional horizontal lines with a variable x value and a fixed y value. Each line in the generated PNG encompasses a network feature such as

source IP, destination IP, MAC address, timestamp, protocol, source port, destination port, packet length, jitter, duration and load. Each file name consists of a prefix (ipv4, ipv6) to aid the I2NeT algorithm in distinguishing and applying the correct decoding strategy.

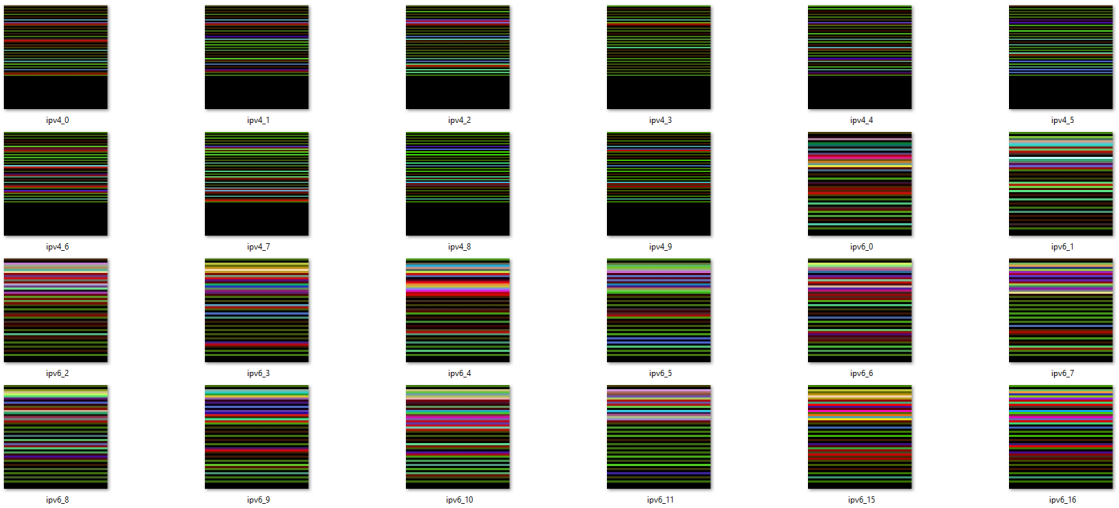


Figure 3. Images generated by NeT2I.

As stated in Section 3.1, NeT2I converts floating-point numbers to RGB pixel values without loss, using the IEEE 754 standardisation. For example, for a floating-point value 3.14159, the Python function `struct.pack('!f',3.14159)` would be used to generate the IEEE 754 binary representation $[64, 9, 33, 25]$. The respective binary representation would then be split into 2 RGB pixel values $((64, 9, 33), (25, 0, 0))$, which would then be employed at the image generation phase in NeT2I.

When encoding an IPv4 address, NeT2I splits the IP address into 4 octets. Each integer octet is treated as a float number and mapped to the respective RGB channel. For instance, for the IP address 192.168.1.100,

- `struct.pack('!f',192.0)` $\rightarrow$IEEE 754 $[66, 192, 0, 0] \rightarrow \text{RGB}((66, 192, 0), (0, 0, 0))$
- `struct.pack('!f',168.0)` $\rightarrow$IEEE 754 $[66, 160, 0, 0] \rightarrow \text{RGB}((66, 160, 0), (0, 0, 0))$
- `struct.pack('!f',1.0)` $\rightarrow$IEEE 754 $[63, 128, 0, 0] \rightarrow \text{RGB}((63, 128, 0), (0, 0, 0))$
- `struct.pack('!f',100.0)` $\rightarrow$IEEE 754 $[66, 100, 0, 0] \rightarrow \text{RGB}((66, 100, 0), (0, 0, 0))$

The generated RGB pixel values are employed in the image generation.

MAC addresses are handled by converting the hexadecimal string into a 48-bit long integer. This integer value is split into two 24-bit chunks, which are converted to a float and mapped using the two RGB pixel method. For example, for a MAC address 00:1A:2B:3C:4D:5E, the following procedure is followed:

- 00:1A:2B:3C:4D:5$E$$\rightarrow$001$A2B$ and 3C4D5E
- 001A2$B$$\rightarrow$67019
- 3C4D5$E$$\rightarrow$3951966
- `struct.pack('!f', 67019.0)` $\rightarrow$IEEE 754 $[72, 131, 128, 0] \rightarrow \text{RGB}((72, 131, 128), (0, 0, 0))$
- `struct.pack('!f', 3951966.0)` $\rightarrow$IEEE 754 $[87, 102, 128, 0] \rightarrow \text{RGB}((87, 102, 128), (0, 0, 0))$

The generated RGB pixel values from the MAC address are employed in image generation.

Finally, IPv6 addresses are mapped to their 128-bit long representation, which is converted to 16 bytes with two additional zeros appended to the end, thus forming an 18-byte long array. For example, for the IP address 2001:0db8:85a3:0000:0000:8a2e:0370:7334:

- `IPv6Adress.packed(2001:0db8:85a3:0000:0000:8a2e:0370:7334)` $\rightarrow$
  $[32, 1, 13, 184, 133, 163, 0, 0, 0, 0, 0, 0, 138, 46, 3, 112]$

- Appending $[0, 0] \rightarrow [32, 1, 13, 184, 133, 163, 0, 0, 0, 0, 0, 0, 138, 46, 3, 112, 0, 0] \rightarrow$ RGB $((32, 1, 13), (184, 133, 163), (0, 0, 0), (0, 0, 0), (138, 46, 3), (112, 0, 0))$

Following the above process, network features are converted to RGB pixel values and these are then used for image generation without loss.

5.2. Computational Complexity

In Table 3, the following averaged times are presented: execution, CPU usage, and memory utilisation for the NeT2I and I2NeT algorithms for each of the three datasets. In Table 4 the computational complexity for the two variants of the CiNeT algorithm is found, which is averaged across the three datasets. Evaluation for the CiNeT algorithm is carried out in terms of time complexity, GPU usage, and memory utilisation.

Table 3. Averaged computational complexity over 10 independent runs for the NeT2I and I2NeT algorithms for each of the three benchmarked datasets.

Algorithm	Dataset	Number of Images	Total Execution Time	Execution Time per Image	CPU Usage	Memory Utilisation
NeT2I	InSDN	215,000	99 s	0.00046 s	100%	18%
	UNSW NB 15	215,000	100 s	0.000465 s	100%	19%
	TON-IoT	215,000	100 s	0.000465 s	100%	19%
I2NeT	InSDN	215,000	103 s	0.00047 s	100%	20%
	UNSW NB 15	215,000	102 s	0.000474 s	100%	20%
	TON-IoT	215,000	101 s	0.000469 s	100%	20%

Table 4. Averaged computational complexity over 10 independent runs for the CiNeT algorithm across varying architectural depths for the three benchmarked datasets.

Algorithm	Layers	Training Time	Validation Time	Testing Time	GPU Usage	Memory Utilisation
CiNeT-TF	1 Layer	12.35 h	1.49 h	25 s	98.2%	26.9%
	2 Layers	12.11 h	2.01 h	43 s	99.9%	27.5%
	3 Layers	13.25 h	2.11 h	1.24 min	99.9%	27.7%
	4 Layers	15.1 h	2.35 h	2.15 min	99.9%	29.5%
	5 Layers	17.45 h	3.05 h	3.30 min	99.9%	30%
CiNeT-PT	1 Layer	5.1 h	3.19 h	2.01 min	5.1%	8.4%
	2 Layers	5.24 h	3.29 h	2.09 min	8.9%	9.4%
	3 Layers	5.38 h	3.31 h	2.11 min	13.2%	10.6%
	4 Layers	6.01 h	3.42 h	2.13 min	14.8%	11%
	5 Layers	6.22 h	3.45 h	2.20 min	15.8%	12.1%

5.2.1. Execution Time

Execution time per image was formulated based on the total execution time, since the task of image creation from network traffic is an Aperiodic Task [77]. The enhanced NeT2I and I2NeT algorithms demonstrate significant empirical improvements over our prior work [26], with a 4.1% reduction in execution time for NeT2I and a 16% reduction for I2NeT. These performance gains are attributed to comprehensive codebase improvements that incorporated Object Oriented Programming (OOP), resulting in a more modular, efficient, and maintainable implementation.

5.2.2. CPU Usage

Given the global interpreter lock and single-threaded execution of Python code, it was observed that the algorithms NeT2I and I2NeT used 100% of the CPU resources. However, due to their light weight and deterministic logic, NeT2I and I2NeT algorithms utilised the CPU for a smaller window, as seen in the total execution time in Table 3, compared to [62], when it was evaluated against NeT2I in [26], releasing the CPU for other tasks. This is an efficient use of resources and allows for rapid completion, thus freeing the CPU for other processes in an MEC environment. High CPU usage during active execution is a hallmark

of an efficient algorithm design, as it indicates reduced and minimal idle time, illustrating an efficient use of available computational resources [76].

5.2.3. Memory Utilisation

NeT2I utilised 19% of memory on average across the three datasets, consistent with the findings at [26]. This low memory footprint is attributed to the algorithm's deterministic approach and the absence of large intermediate data structures. The light-weight encoding operations, such as binary serialisation, struct packing, pixel generation, and mapping, make NeT2I ideal to launch in a resource-constrained environment. The higher utilisation of 20% by I2NeT can be attributed towards the overhead generated by image loading and the initial processing of NumPy arrays for RGB extraction. Although a marginal increase in memory utilisation was recorded, memory leaks or unbounded copies were not recorded, making the decoding I2NeT suitable for resource-constrained edge devices.

5.2.4. GPU Utilisation

During the training, validation, and testing phases, data relating to GPU usage was collected. The data can be seen in Table 4. The observed results and usage patterns revealed a dramatic divergence. The variant developed using the TensorFlow framework, namely CiNeT-TF, exhibited a high GPU usage consistently with an average of 99.9% across its application to the three datasets (for layers 1 to 5). This demonstrates a highly efficient execution in which the GPU is nearly saturated during the training and validation process, minimising idle time and maximum throughput. However, this high utilisation is attributed to a cost, where frequent out-of-memory (OOM) errors occur when employing larger batch sizes within deeper architectures (4 to 5 layers), indicating a resource boundary. The same has been observed in the works of Gao et al. [39], where utilisation constituted a constraint of computation, forcing reduced batch sizes or model complexities.

In contrast, the CiNeT developed using the PyTorch framework (CiNeT-PT) exhibited a low GPU utilisation across its application to the three datasets and all applications (layers 1 to 5). While this may constitute a performance deficiency, it reflects the fundamental difference between frameworks, where reproducibility and stability are crucial over optimal hardware usage [22]. Unlike the TensorFlow variant of CiNeT operating at the edge of GPU memory, which resulted in OOM errors, the PyTorch variant prevents resource contention and system instability. As stated in the work of Sencan et al. [78], such inefficiencies are common in real-world workloads where GPU usage remains low despite high resource availability. Thus suggesting that underutilisation of resources, computation, or memory can be construed as a deliberate design choice to enhance fault tolerance in production environments where crash avoidance is crucial and predictable behaviour sought, as opposed to unpredictability and OOM errors due to saturation [79,80]. Table 4 also shows the memory utilisation for the CiNeT algorithm across the two variants with various architectural depths. Initial observations reveal that the iterations of CiNeT-TF have consumed more memory, ranging from 26.9% to 30% of the system's total memory allocation, compared to the CiNeT-PT variants, utilising only 8.4% to 12.1%. The disparity in memory utilisation is attributed to the memory management strategy of the underlying framework, where TensorFlow tends to reserve larger memory blocks at the start of execution for graph construction and data handling, contributing to a higher memory utilisation. Conversely, PyTorch employs a more dynamic on-demand memory allocation process, where memory is allocated as tensors during the training phase, and when these tensors are out of scope, the allotted memory is released back to the system through efficient and prompt reference counting and garbage collection mechanisms. This process employed

by PyTorch has led to a more conservative memory footprint in contrast to the memory footprint created by TensorFlow [32].

The CiNeT-PT variant can therefore be viewed as a failure-averse deployment for the intrusion detection problem, with a priority for stability and energy efficiency. As stated by [81], peak utilisation of resources, memory, and computation is largely proportional to higher energy consumption, and by operating at a lower GPU usage, CiNeT-PT avoids resource saturation and power demands, whilst maintaining an enhanced fault tolerance, resulting in a reliable and sustainable choice for real-time security applications.

5.3. Theoretical Analysis (Big-O Notation)

The theoretical analysis of NeT2I, I2NeT, and CiNeT is conducted by employing the Big-O notation. Big-O notation is used to characterise an algorithm as a function of time and space with respect to the input [82].

5.3.1. Theoretical Complexity for NeT2I

In evaluating NeT2I, let n denote the number of rows in the input csv file, d the number of features, and p the dimension of the output image.

- Reading the CSV file is $O(n \cdot d)$, as each row must be scanned and passed across columns.
- The nested loop, which iterates over each row and feature, results in $O(n \cdot d)$ iterations.
  - Within the loop, each data entry is encoded using $O(1)$
  - Therefore, processing one row is $O(d)$ and for n rows, $O(n \cdot d)$
- Image generation of p by p pixels, where p is the number of RGB stripes, will result in p^2
- As the pixels are of a fixed size and do not scale with the number of rows or features, $O(p^2) = O(1)$

Therefore, the total time complexity can be stated as ($T(n, d)$)

$$T(n, d) = O(n \cdot d) + O(1) = O(n \cdot d) \tag{1}$$

For the space complexity $S(n, d)$, it can be seen that

- The input as per the above $O(n \cdot d)$
- The output image, as per the above $O(p^2) = O(1)$

Hence,

$$S(n, d) = O(n \cdot d) \tag{2}$$

It therefore follows that the NeT2I algorithm belongs to the polynomial-time class bounded by $O(n \cdot d)$, making it suitable for real-time applications.

5.3.2. Theoretical Complexity for I2NeT

In applying the Big-O notation to I2NeT, let m denote the number of images, p the number of RGB pixel stripes, and d the number of features.

- Discovery of images with $O(m)$ and sorting m files $O(mlogm)$
- Loading the JSON file as $O(1)$
- Image decoding and RGB extraction, involves iterating over p rows in the image, resulting in $O(p)$
- Employing the JSON file, calculation of the pixel count, with d as the number of features, which is constant, resulting in $O(1)$
- Value reconstruction similar to value encoding is $O(1)$

- For one image $T(1, p) = O(p)$ with m images, $T(m, p) = O(m \cdot p)$. With $O(mlogm)$ cost being smaller, the total time complexity can be stated using

$$T(m, p) = O(m \cdot p) \tag{3}$$

With regards to the space complexity of I2NeT,

- The list of images $O(m)$
- Extracting RGB pixel data and the reconstruction for a row $O(p)$
- Hence, the final CSV $O(m \cdot p)$

It follows that the space complexity ($S(m, p)$) is given by

$$S(m, p) = O(m \cdot p) \tag{4}$$

Similar to NeT2I, the decoding algorithm I2NeT also belongs to the polynomial-time class bounded by $O(m \cdot p)$. As stated by [82], polynomial-time algorithms are regarded as tractable or efficiently solvable.

5.3.3. Theoretical Complexity for CiNeT

When evaluating the theoretical complexity for CiNeT, the training, validation, and testing phases of the algorithm must be taken into consideration. Let E denote the epochs, N_{train} the number of training samples, N_{val} the validation samples, and F the number of floating-point operations for a single forward pass through the network. This theoretical application is embedded into the standard methodology for training, validating, and testing a DL model [39,83].

- For each sample, the model conducts a forward pass through convolutional layers and fully connected layers. As the number of operations is determined by the model architecture and the operations per sample are constant, the forward pass is $O(F)$
- Similarly, the back propagation can be construed as being approximately proportional to the above, hence it is also $O(F)$
- The optimizer updates the weights and this can be $O(P)$, where P is the number of parameters.
- The above steps are repeated for each sample, resulting in N_{train} times per epoch, with the loop being repeated E times.

The total training time (T_{train}) is therefore given to be

$$T_{train} = O(E \cdot N_{train} \cdot (F + F + P)) = O(E \cdot N_{train} \cdot F) \tag{5}$$

During the validation and the testing phase, the model requires a forward pass $O(F)$ for each N, with N_{val} for validation and N_{test} for testing, resulting in $T_{inference}$

$$T_{inference} = O(N \cdot F) \tag{6}$$

For the space complexity S,

$$S = O(P + B \cdot F_{activation}) \tag{7}$$

where P is the total number of parameters, batch size B, and the number of floating-point values in the activation map $F_{activation}$ created during the forward pass.

5.4. Training and Validation

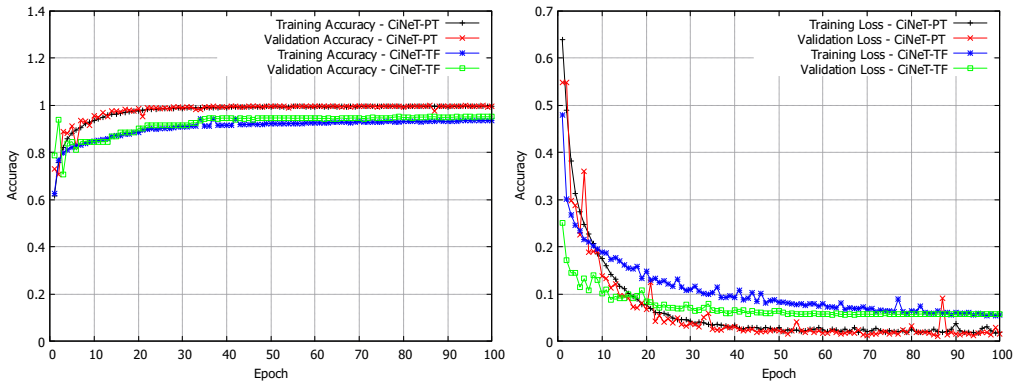
Table 5 presents the validation accuracy for CiNeT models with 1 to 5 layers, instantiated using both TensorFlow and PyTorch. While both variants show improved accuracy

with layer depth up to 3, the CiNeT-TF suffered performance degradation at layers 4 and 5. In contrast CiNeT-PT maintained its performance up to 4 layers prior to plateauing at 5 layers. The collated results demonstrate that CiNeT-PT achieves a peak validation accuracy of 99.2% at 4 layers. Conversely, CiNeT-TF achieved a peak of 97.1% at 3 layers. This finding corroborates the work of [20], where the accuracy of their model developed in TensorFlow degraded accuracy and precision when the number of layers exceeded three, suggesting architectural depth and sensitivity to accuracy.

Table 5. Averaged validation accuracy across 10 independent runs of CiNeT variants with 1 to 5 layers for each of the three benchmarked datasets.

Dataset	CiNeT-TF (%)					CiNeT-PT (%)				
	1L	2L	3L	4L	5L	1L	2L	3L	4L	5L
InSDN	94.3	95.8	97.1	96.5	95.9	96.7	97.8	98.4	99.1	98.6
UNSW-NB15	93.6	95.0	96.8	96.0	95.3	95.9	97.1	98.0	98.9	98.3
ToN-IoT	94.8	96.0	97.2	96.7	96.1	97.0	98.0	98.7	99.2	98.8

As evident in Figure 4a,b, the CiNeT-PT (4 Layer) model acting on the ToN_IoT dataset exhibits a steady increase in training and validation accuracy, suggesting effective learning with minimal overfitting. Conversely, the CiNeT-TF (3 Layer) model acting on ToN_IoT shows fluctuations in both accuracy and loss, with the validation curve displaying a slower convergence, suggesting potential numerical issues during training.



(**a**) Training and Validation Accuracy (**b**) Training and Validation Loss

Figure 4. Training and validation data from the TON_IoT dataset.

5.5. Evaluation of Detection

Albeit marginal, this performance gap suggests that CiNeT-PT implementation benefitted from the efficient training process, better gradient handling, consistent weight initialisation, efficient memory usage (as shown in Table 4), and numerical instabilities, as suggested by the authors of [84,85]. These studies further suggest that TensorFlow models are prone to silent bugs and incorrect gradient computations that can degrade model accuracy. In contrast, PyTorch's dynamic computation graph and programming model offer greater transparency and control during the training and validation stages, enabling a reliable and more accurate model with a deeper architecture, making it more suitable for a complex multi-class intrusion detection problem.

The performance of the two CiNeT algorithms was evaluated using the results shown in Tables 6 and 7. Table 6 presents the accuracy acquired during class-wise tests for both CiNeT-TF(3 Layer) and CiNeT-PT (4 Layer) for each of the three datasets, with Table 7 providing a more granular analysis of the best model (CiNeT-PT(4 Layer)), presenting the accuracy (Acc), precision (Prec), recall (Rec), and F1-score (F1) for each traffic class.

Table 6. Averaged test accuracy across 10 independent runs of CiNeT variants with 1 to 5 layers for the three benchmarked datasets and traffic classes.

Traffic Class	InSDN		UNSW-NB15		ToN-IoT	
	CiNeT-TF (3L)	CiNeT-PT (4L)	CiNeT-TF (3L)	CiNeT-PT (4L)	CiNeT-TF (3L)	CiNeT-PT (4L)
Normal	98.5	99.0	96.9	97.5	99.1	99.4
DDoS	98.7	99.3	97.2	98.1	99.0	99.5
DoS	97.5	98.4	95.8	96.9	98.1	98.8
Reconnaissance	96.3	97.5	94.6	95.7	97.0	97.8
Exploits	95.1	96.4	93.2	94.5	95.9	96.7
Backdoor	94.0	95.2	91.8	93.0	94.8	95.6

Table 7. Averaged confusion matrix of traffic classes across 10 independent runs of CiNeT-PT (4 Layer) for the three benchmarked datasets.

Traffic Class	InSDN				UNSW-NB15				ToN-IoT			
	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1	Acc	Prec	Rec	F1
Normal	99.0	98.7	99.2	98.9	97.5	97.1	97.8	97.4	99.4	99.2	99.5	99.3
DDoS	99.3	99.1	99.4	99.2	98.1	97.8	98.3	98.0	99.5	99.3	99.6	99.4
DoS	98.4	98.0	98.7	98.3	96.9	96.5	97.2	96.8	98.8	98.5	99.0	98.7
Reconnaissance	97.5	97.0	97.9	97.4	95.7	95.2	96.0	95.6	97.8	97.4	98.1	97.7
Exploits	96.4	95.9	96.8	96.3	94.5	94.0	94.9	94.4	96.7	96.3	97.0	96.6
Backdoor	95.2	94.7	95.6	95.1	93.0	92.5	93.4	92.9	95.6	95.2	95.9	95.5

Further to the results found in Table 5, Table 6 confirms superior detection for the CiNeT-PT (4 Layer) algorithm. For the ToN-IoT dataset (a highly sought-after dataset with diverse and realistic data), CiNeT-PT achieved an exceptional accuracy of 99.4% for normal traffic and 99.5% for DDoS traffic, suggesting an improvement from 99.1% and 99% for the CiNeT-TF algorithm. The performance variation is observable for each of the cases investigated.

As shown in Table 7, the CiNeT-PT variant achieves an F1-score of 99.4%, 99.3% for precision, and 99.6% for recall, over DDoS traffic, suggesting that the model performed with high precision and accuracy and a lower false alarm rate. Similarly, for other attack types such as backdoor, exploits, DoS, and reconnaissance, the CiNeT-PT variant maintained high accuracy and precision, demonstrating the robust application of CiNeT-PT for detecting a wide range of attacks. Achieving an accuracy of 100% would signify an overfitting of data in neural networks; the achieved accuracy can be considered as having achieved cross-validation in our methodology due to augmentation, regularisation, and increase of varied training data [86].

6. Discussion

In the context of this research, a comprehensive performance evaluation of the CiNeT algorithm is presented. CiNeT is a novel DL algorithm capable of automated attack class detection and represents a new IDS that operates on images encoded and decoded via a NeT2I-I2NeT pipeline. The work has improved and extended the pipeline through which a broader range of network features, including IPv6 and floating-point numbers without loss, can be encoded as RGB images, allowing for a rigorous comparative analysis of the CiNeT algorithm deployed using TensorFlow and PyTorch. The evaluation focuses not only on critical performance metrics such as accuracy, prediction, recall, and F1-score, but also on theoretical and empirical evaluations of its computational complexity and resource utilisation.

The computational complexity of the CiNeT algorithms reveals a fundamental trade-off between resource usage and model robustness. The TensorFlow variant achieving peak GPU usage demonstrated hardware efficiency, but fragility, and error-prone behaviour

during training and validation. The PyTorch variant exhibited low GPU usage and demonstrated exceptional stability, robustness, and consistent performance. This observation highlighted an important insight: high GPU usage may constrain performance, efficiency, and sustainability. The CiNeT-PT 4 Layer model, therefore, produces a superior performance that prioritises stability, reproducibility, and sustainability, without compromising accuracy of detection.

To further validate the observed performance increase of CiNeT-PT compared to CiNeT-TF, the two best-performing models (CiNeT-PT (4 Layer) and CiNeT-TF (3 Layer)) were subjected to a two-sample *t*-test on the collected results from 10 independent runs, employing the below hypothesis.

Null Hypothesis (H_0). *There is no difference in the mean performance between CiNeT-TF and CiNeT-PT ($\mu_1 = \mu_2$).*

Alternative Hypothesis (H_1). *There is a difference in the mean performance between CiNeT-TF and CiNeT-PT ($\mu_1 \neq \mu_2$).*

Table 8 presents the standard deviation across independent runs. For Training time, the mean of CiNeT-TF (13.25 ± 0.32) showed a significant increase over CiNeT-PT (6.01 ± 0.15), with a `t-statistic` of 64.76 and 13 degrees of freedom, resulting a p`-value` < 0.001. For accuracy, a `t-statistic` of 15.34 and 10 degrees of freedom, resulting a p`-value` < 0.001. For GPU usage `t-statistic` of 224.48 and 9 degrees of freedom, resulting a p`-value` < 0.001. Finally, for memory usage, `t-statistic` of 20.34 and 10 degrees of freedom, resulting a p`-value` < 0.001. Since all the p`-value` were less than 0.001, H_0 can be rejected, accepting the H_1.

Table 8. Statistical comparison of CiNeT-TF (3L) and CiNeT-PT (4L). The mean and standard deviation ($\pm$ STD) of key performance metrics reported over 10 independent runs on the ToN-IoT dataset.

Metric	CiNeT-TF (3 Layer) Mean $\pm$ STD	CiNeT-PT (4 Layer) Mean $\pm$ STD
Training Time (h)	13.25 ± 0.32	6.01 ± 0.15
Accuracy (%)	97.2 ± 0.4	99.2 ± 0.1
GPU Usage (%)	99.9 ± 0.1	14.8 ± 1.2
Memory Utilisation (%)	27.7 ± 2.5	11 ± 0.7

It is evident that the CiNeT-PT variant has outperformed the CiNeT-TF variant. As discussed previously, this performance gain can be attributed to the stable and efficient training process, gradient handling, weight initialisation, and resource usage. As the CiNeT-PT (4 Layer) model was able to detect with high precision and minimal false positives, it can be stated that the model is exceptionally well-suited for application as an IDS.

Finally, across the three datasets, the ToN-IoT dataset outperformed both the UNSW-NB15 and InSDN datasets. This can be attributed to the dataset quality and diversity of traffic when evaluating intrusion detection problems. This aligns with recent studies that have suggested ToN-IoT is better suited for IDSs equipped with DL models [28,29]. Following our research, it can be stated that the CiNeT-PT (4-layer) variant represents a significant advancement in the field of network security.

7. Conclusions and Future Work

In this paper, we presented CiNeT, a novel CNN-based IDS capable of detecting multiple classes of malicious traffic, leveraging both PyTorch and TensorFlow. Alongside CiNeT, an advanced pipeline of NeT2I and I2NeT was also introduced, enabling a bijective

encoding–decoding process to be established, which allowed for the application of the CiNeT detection algorithm. This approach allowed for the utilisation of CNN for spatial pattern recognition in network flows, which maintained full traceability from detection to packet-level information, providing a step towards intrusion prevention. This method enhances the interpretability of the model by enabling complete traceability from the detection decision to the packet-level information through the bijective NeT2I-I2NeT pipeline. While not a complete solution to the 'black-box' problem associated with Deep Learning models [18], this reversibility allows for the reconstruction and inspection of the network flows that trigger an alert and paves the way to a explainable intrusion detection system utilising a Deep Learning model. In the current NeT2I-I2NeT pipeline, unstructured string data is removed during preprocessing. To ensure system continuity, any residual temporal string data is handled and managed gracefully using the hash function to preserve system continuity.

Two variants of the CiNeT algorithms were evaluated across three datasets, UNSW NB-15, InSDN, and TON_IoT, with a focus on multi-class classification in intrusion detection. Our results demonstrated that CiNeT-PT (4 Layer) achieved a superior accuracy of 99.5%, outperforming the CiNeT-TF architectures. CiNeT-PT outperformed CiNeT-TF across computational metrics, calculating a 60% reduction in training time, a 63% reduction in memory utilisation, and an 88% reduction in GPU usage, making CiNeT-PT a strong candidate for deployment in resource-constrained environments.

Currently, research is being conducted towards the deployment of CiNeT-PT (4 Layer) in a 5G testbed, extending the research in [87], to include the integration of a Next Unit of Computing (NUC) device using an Intel x86 CPU, which is capable of conducting edge detection. The environment will also integrate control and data plane programmability using technologies such as Software Defined Networking and Programming Protocol independent Packet Processing [88] to realise a 5G and B5G testbed that employs a real-time CNN-based NIDS. This work draws upon the foundational framework of Real-Time Deep Learning-based NIDS (RTDL-NIDS) [89], where the NeT2I-CNN-I2NeT pipeline was successfully implemented and evaluated within a 5G-Multi-Access Edge Computing (MEC) mobile telecommunication testbed. This work demonstrates that the aforementioned pipeline enables intrusion detection in real-time as opposed to a 'desk-approach'. The ongoing research is aimed at implementing the CiNeT-PT (4 Layer) within a 5G testbed, paving the way for an intelligent and automated security implementation.

The success of this application not only relies on a higher accuracy of detection, but also on its resilience to adversarial attacks. The current application of this pipeline possesses a limitation in that its robustness to adversarial perturbations has not been assessed. To mitigate misclassification of crafted messages by an adversary, the robustness of CiNeT will be evaluated using the Fast Gradient Sign Method and Projected Gradient Descent methods. Applying Kerckhoffs's principle, input sanitation against adversarial manipulation for the NeT2I-CiNeT-I2NeT pipeline will also be conducted. This analysis will be crucial for understanding end-to-end security, together with associated protocol aspects relating to confidentiality, integrity, availability, and non-repudiation.

Author Contributions: Conceptualisation, O.A.F., J.S. and H.X.; methodology, O.A.F., J.S. and H.X.; software, O.A.F.; validation, O.A.F., J.S. and H.X.; formal analysis, O.A.F., J.S. and H.X.; investigation, O.A.F., J.S. and H.X.; resources, O.A.F.; data curation, O.A.F., J.S. and H.X.; writing—original draft preparation, O.A.F., J.S. and H.X.; writing—review and editing, O.A.F., J.S. and H.X.; visualisation, O.A.F.; supervision, J.S. and H.X. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The original contributions presented in this study are included in the article.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Parkvall, S.; Dahlman, E.; Furuskar, A.; Frenne, M. NR: The new 5G radio access technology. *IEEE Commun. Stand. Mag.* **2018**, *1*, 24–30. [CrossRef]
2. Lin, X. The bridge toward 6G: 5G-Advanced evolution in 3GPP Release I9. *IEEE Commun. Stand. Mag.* **2025**, *9*, 28–35. [CrossRef]
3. Chen, W.; Lin, X.; Lee, J.; Toskala, A.; Sun, S.; Chiasserini, C.F.; Liu, L. 5G-advanced toward 6G: Past, present, and future. *IEEE J. Sel. Areas Commun.* **2023**, *41*, 1592–1619. [CrossRef]
4. Lin, X. An overview of 5G advanced evolution in 3GPP release 18. *IEEE Commun. Stand. Mag.* **2022**, *6*, 77–83. [CrossRef]
5. Eleftherakis, S.; Giustiniano, D.; Kourtellis, N. SoK: Evaluating 5G-Advanced Protocols Against Legacy and Emerging Privacy and Security Attacks. In Proceedings of the 18th ACM Conference on Security and Privacy in Wireless and Mobile Networks, Arlington, VA, USA, 30 June–3 July 2025; pp. 196–210.
6. Michaelides, S.; Lenz, S.; Vogt, T.; Henze, M. Secure integration of 5G in industrial networks: State of the art, challenges and opportunities. *Future Gener. Comput. Syst.* **2025**, *166*, 107645. [CrossRef]
7. Bodenhausen, J.; Sorgatz, C.; Vogt, T.; Grafflage, K.; Rötzel, S.; Rademacher, M.; Henze, M. Securing wireless communication in critical infrastructure: Challenges and opportunities. In Proceedings of the International Conference on Mobile and Ubiquitous Systems: Computing, Networking, and Services, Melbourne, Australia, 14–17 November 2023; Springer: Berlin/Heidelberg, Germany, 2023; pp. 333–352.
8. Taylor, P. Mobile Phone Subscriptions Worldwide 2024. 2025. Available online: https://www.statista.com/statistics/262950/global-mobile-subscriptions-since-1993/ (accessed on 17 September 2025).
9. Ericsson. Mobile Data Traffic Forecast—Ericsson Mobility Report. 2024. Available online: https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/mobile-traffic-forecast (accessed on 17 September 2025).
10. Wang, Y.; Wang, J. Research on Computer Network Big Data Security Defense System Based on Support Vector Machine and Deep Learning. In Proceedings of the 2025 IEEE International Conference on Electronics, Energy Systems and Power Engineering (EESPE), Shenyang, China, 17–19 March 2025; IEEE: Piscataway, NJ, USA, 2025; pp. 386–392.
11. Ma, J.; Li, S. The construction method of computer network security defense system based on multisource big data. *Sci. Program.* **2022**, *2022*, 7300977. [CrossRef]
12. Apruzzese, G.; Laskov, P.; Montes de Oca, E.; Mallouli, W.; Brdalo Rapa, L.; Grammatopoulos, A.V.; Di Franco, F. The role of machine learning in cybersecurity. *Digit. Threat. Res. Pract.* **2023**, *4*, 8. [CrossRef]
13. Ozkan-Okay, M.; Akin, E.; Aslan, Ö.; Kosunalp, S.; Iliev, T.; Stoyanov, I.; Beloev, I. A comprehensive survey: Evaluating the efficiency of artificial intelligence and machine learning techniques on cyber security solutions. *IEEE Access* **2024**, *12*, 12229–12256. [CrossRef]
14. Ali, M.L.; Thakur, K.; Schmeelk, S.; Debello, J.; Dragos, D. Deep Learning vs. Machine Learning for Intrusion Detection in Computer Networks: A Comparative Study. *Appl. Sci.* **2025**, *15*, 1903. [CrossRef]
15. Rosenberg, I.; Shabtai, A.; Elovici, Y.; Rokach, L. Adversarial machine learning attacks and defense methods in the cyber security domain. *ACM Comput. Surv. (CSUR)* **2021**, *54*, 108.
16. Kaloudi, N.; Li, J. The ai-based cyber threat landscape: A survey. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 20. [CrossRef]
17. Chauhan, N.K.; Singh, K. A review on conventional machine learning vs deep learning. In Proceedings of the 2018 International Conference on Computing, Power and Communication Technologies (GUCON), Greater Noida, India, 28–29 September 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 347–352.
18. Charmet, F.; Tanuwidjaja, H.C.; Ayoubi, S.; Gimenez, P.F.; Han, Y.; Jmila, H.; Blanc, G.; Takahashi, T.; Zhang, Z. Explainable artificial intelligence for cybersecurity: A literature survey. *Ann. Telecommun.* **2022**, *77*, 789–812. [CrossRef]
19. Al-Turaiki, I.; Altwaijry, N. A convolutional neural network for improved anomaly-based network intrusion detection. *Big Data* **2021**, *9*, 233–252. [CrossRef]
20. Kim, J.; Kim, J.; Kim, H.; Shim, M.; Choi, E. CNN-based network intrusion detection against denial-of-service attacks. *Electronics* **2020**, *9*, 916. [CrossRef]
21. Nazarri, M.N.A.A.; Yusof, M.H.M.; Almohammedi, A.A. Generating network intrusion image through IGTD algorithm for CNN classification. In Proceedings of the 2023 3rd International Conference on Computing and Information Technology (ICCIT), Tabuk, Saudi Arabia, 10–11 May 2023; IEEE: Piscataway, NJ, USA, 2023; pp. 172–177.
22. Shahriari, M.; Ramler, R.; Fischer, L. How do deep-learning framework versions affect the reproducibility of neural network models? *Mach. Learn. Knowl. Extr.* **2022**, *4*, 888–911. [CrossRef]

23. Moustafa, N.; Slay, J. UNSW-NB15: A comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set). In Proceedings of the 2015 Military Communications and Information Systems Conference (MilCIS), Canberra, Australia, 10–12 November 2015; pp. 1–6.
24. Elsayed, M.S.; Le-Khac, N.A.; Jurcut, A.D. InSDN: A novel SDN intrusion dataset. *IEEE Access* **2020**, *8*, 165263–165284. [CrossRef]
25. Moustafa, N. The ToN_IoT Datasets. Available online: https://research.unsw.edu.au/projects/toniot-datasets (accessed on 17 September 2025).
26. Fernando, O.A.; Xiao, H.; Spring, J. New Algorithms for the Detection of Malicious Traffic in 5G-MEC. In Proceedings of the 2023 IEEE Wireless Communications and Networking Conference (WCNC), Glasgow, UK, 26–29 March 2023; IEEE: Piscataway, NJ, USA, 2023.
27. Adadi, A.; Berrada, M. Peeking inside the black-box: A survey on explainable artificial intelligence (XAI). *IEEE Access* **2018**, *6*, 52138–52160. [CrossRef]
28. Tareq, I.; Elbagoury, B.M.; El-Regaily, S.; El-Horbaty, E.S.M. Analysis of ton-iot, unw-nb15, and edge-iiot datasets using dl in cybersecurity for iot. *Appl. Sci.* **2022**, *12*, 9572. [CrossRef]
29. Kolhar, M.; Aldossary, S.M. A deep learning approach for securing IoT infrastructure with emphasis on smart vertical networks. *Designs* **2023**, *7*, 139. [CrossRef]
30. Yin, C.; Zhu, Y.; Fei, J.; He, X. A deep learning approach for intrusion detection using recurrent neural networks. *IEEE Access* **2017**, *5*, 21954–21961. [CrossRef]
31. Shaheen, F.; Verma, B.; Asafuddoula, M. Impact of automatic feature extraction in deep learning architecture. In Proceedings of the 2016 International Conference on Digital Image Computing: Techniques and Applications (DICTA), Gold Coast, Australia, 30 November–2 December 2016; IEEE: Piscataway, NJ, USA, 2016; pp. 1–8.
32. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. Pytorch: An imperative style, high-performance deep learning library. *Adv. Neural Inf. Process. Syst.* **2019**, *32*.
33. Dash, N.; Chakravarty, S.; Rath, A.K.; Giri, N.C.; AboRas, K.M.; Gowtham, N. An optimized LSTM-based deep learning model for anomaly network intrusion detection. *Sci. Rep.* **2025**, *15*, 1554. [CrossRef] [PubMed]
34. Thakkar, A.; Kikani, N.; Geddam, R. Fusion of linear and non-linear dimensionality reduction techniques for feature reduction in LSTM-based Intrusion Detection System. *Appl. Soft Comput.* **2024**, *154*, 111378. [CrossRef]
35. Bukhari, S.M.S.; Zafar, M.H.; Abou Houran, M.; Moosavi, S.K.R.; Mansoor, M.; Muaaz, M.; Sanfilippo, F. Secure and privacy-preserving intrusion detection in wireless sensor networks: Federated learning with SCNN-Bi-LSTM for enhanced reliability. *Ad Hoc Netw.* **2024**, *155*, 103407. [CrossRef]
36. Hossain, M.D.; Inoue, H.; Ochiai, H.; Fall, D.; Kadobayashi, Y. LSTM-based intrusion detection system for in-vehicle can bus communications. *IEEE Access* **2020**, *8*, 185489–185502. [CrossRef]
37. Lira, O.G.; Marroquin, A.; To, M.A. Harnessing the advanced capabilities of llm for adaptive intrusion detection systems. In Proceedings of the International Conference on Advanced Information Networking and Applications, Kitakyushu, Japan, 17–19 April 2024; Springer: Berlin/Heidelberg, Germany, 2024; pp. 453–464.
38. Adjewa, F.; Esseghir, M.; Merghem-Boulahia, L.; Kacfah, C. Llm-based continuous intrusion detection framework for next-gen networks. In Proceedings of the 2025 International Wireless Communications and Mobile Computing (IWCMC), Abu Dhabi, United Arab Emirates, 12–16 May 2025; IEEE: Piscataway, NJ, USA, 2025; pp. 1198–1203.
39. Gao, Y.; He, Y.; Li, X.; Zhao, B.; Lin, H.; Liang, Y.; Zhong, J.; Zhang, H.; Wang, J.; Zeng, Y.; et al. An empirical study on low gpu utilization of deep learning jobs. In Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024; pp. 1–13.
40. Amini, M.; Asemian, G.; Kantarci, B.; Ellement, C.; Erol-Kantarci, M. Deep Fusion Intelligence: Enhancing 5G Security Against Over-the-Air Attacks. *IEEE Trans. Mach. Learn. Commun. Netw.* **2025**, *3*, 263–279. [CrossRef]
41. Rajabi, S.; Asgari, S.; Jamali, S.; Fotohi, R. An intrusion detection system using the artificial neural network-based approach and firefly algorithm. *Wirel. Pers. Commun.* **2024**, *137*, 2409–2440. [CrossRef]
42. Alzubi, O.A.; Alzubi, J.A.; Qiqieh, I.; Al-Zoubi, A. An IoT intrusion detection approach based on salp swarm and artificial neural network. *Int. J. Netw. Manag.* **2025**, *35*, e2296. [CrossRef]
43. Azzaoui, H.; Boukhamla, A.Z.E.; Perazzo, P.; Alazab, M.; Ravi, V. A lightweight cooperative intrusion detection system for rpl-based iot. *Wirel. Pers. Commun.* **2024**, *134*, 2235–2258. [CrossRef]
44. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef]
45. Azizjon, M.; Jumabek, A.; Kim, W. 1D CNN based network intrusion detection with normalization on imbalanced data. In Proceedings of the 2020 International Conference on Artificial Intelligence in Information and Communication (ICAIIC), Fukuoka, Japan, 19–21 February 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 218–224.
46. Abdulraheem, M.H.; Ibraheem, N.B. Anomaly-Based Intrusion Detection System Using One Dimensional and Two Dimensional Convolutions. In Proceedings of the International Conference on Applied Computing to Support Industry: Innovation and Technology, Ramadi, Iraq, 15–16 September 2019; Springer: Berlin/Heidelberg, Germany, 2019; pp. 409–423.

47. Mohammadpour, L.; Ling, T.C.; Liew, C.S.; Aryanfar, A. A survey of CNN-based network intrusion detection. *Appl. Sci.* **2022**, *12*, 8162. [CrossRef]
48. Elouardi, S.; Motii, A.; Jouhari, M.; Amadou, A.N.H.; Hedabou, M. A survey on Hybrid-CNN and LLMs for intrusion detection systems: Recent IoT datasets. *IEEE Access* **2024**, *48*, 180009–180033. [CrossRef]
49. Wang, L.H.; Dai, Q.; Du, T.; Chen, L.f. Lightweight intrusion detection model based on CNN and knowledge distillation. *Appl. Soft Comput.* **2024**, *165*, 112118. [CrossRef]
50. Abed, R.A.; Hamza, E.K.; Humaidi, A.J. A modified CNN-IDS model for enhancing the efficacy of intrusion detection system. *Meas. Sens.* **2024**, *35*, 101299. [CrossRef]
51. El-Ghamry, A.; Darwish, A.; Hassanien, A.E. An optimized CNN-based intrusion detection system for reducing risks in smart farming. *Internet Things* **2023**, *22*, 100709. [CrossRef]
52. Yang, L.; Shami, A. A transfer learning and optimized CNN based intrusion detection system for Internet of Vehicles. In Proceedings of the ICC 2022—IEEE International Conference on Communications, Seoul, Republic of Korea, 16–20 May 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 2774–2779.
53. Lu, K.D.; Huang, J.C.; Zeng, G.Q.; Chen, M.R.; Geng, G.G.; Weng, J. Multi-objective discrete extremal optimization of variable-length blocks-based CNN by joint NAS and HPO for intrusion detection in IIoT. *IEEE Trans. Dependable Secur. Comput.* **2025**, *22*, 4266–4283. [CrossRef]
54. Kharoubi, K.; Cherbal, S.; Mechta, D.; Gawanmeh, A. Network intrusion detection system using convolutional neural networks: Nids-dl-cnn for iot security. *Clust. Comput.* **2025**, *28*, 219. [CrossRef]
55. Torre, D.; Chennamaneni, A.; Jo, J.; Vyas, G.; Sabrsula, B. Toward enhancing privacy preservation of a federated learning cnn intrusion detection system in iot: Method and empirical study. *ACM Trans. Softw. Eng. Methodol.* **2025**, *34*, 53. [CrossRef]
56. Kim, I.; Chung, T.M. Malicious-Traffic Classification Using Deep Learning with Packet Bytes and Arrival Time. In Proceedings of the International Conference on Future Data and Security Engineering, Quy Nhon, Vietnam, 25–27 November 2020; Springer: Berlin/Heidelberg, Germany, 2020; pp. 345–356.
57. Wang, Z.; Ghaleb, F.A.; Zainal, A.; Siraj, M.M.; Lu, X. An efficient intrusion detection model based on convolutional spiking neural network. *Sci. Rep.* **2024**, *14*, 7054. [CrossRef]
58. Yue, C.; Wang, L.; Wang, D.; Duo, R.; Nie, X. An ensemble intrusion detection method for train Ethernet consist network based on CNN and RNN. *IEEE Access* **2021**, *9*, 59527–59539. [CrossRef]
59. Liu, Y.; Kang, J.; Li, Y.; Ji, B. A Network Intrusion Detection Method Based on CNN and CBAM. In Proceedings of the IEEE INFOCOM 2021—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Vancouver, BC, Canada, 9–13 May 2021; IEEE: Piscataway, NJ, USA, 2021; pp. 1–6.
60. Ding, Y.; Zhai, Y. Intrusion detection system for NSL-KDD dataset using convolutional neural networks. In Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence, Las Vegas, NV, USA, 12–14 December 2018; pp. 81–85.
61. Shapira, T.; Shavitt, Y. Flowpic: Encrypted internet traffic classification is as easy as image recognition. In Proceedings of the IEEE INFOCOM 2019—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Paris, France, 29 April–2 May 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 680–687.
62. Janabi, A.H.; Kanakis, T.; Johnson, M. Convolutional Neural Network Based Algorithm for Early Warning Proactive System Security in Software Defined Networks. *IEEE Access* **2022**, *10*, 14301–14310. [CrossRef]
63. Farrukh, Y.A.; Wali, S.; Khan, I.; Bastian, N.D. Senet-i: An approach for detecting network intrusions through serialized network traffic images. *Eng. Appl. Artif. Intell.* **2023**, *126*, 107169. [CrossRef]
64. Zilberman, A.; Dvir, A.; Stulman, A. IPv6 Routing Protocol for Low-Power and Lossy Networks Security Vulnerabilities and Mitigation Techniques: A Survey. *ACM Comput. Surv.* **2025**, *57*, 280. [CrossRef]
65. IR Team. *IPv4 to IPv6 Migration: Complete Enterprise Guide for 2025*; IR Team: Sydney, Australia, 2025.
66. Riedy, J.; Demmel, J. Augmented arithmetic operations proposed for IEEE-754 2018. In Proceedings of the 2018 IEEE 25th Symposium on Computer Arithmetic (ARITH), Amherst, MA, USA, 25–27 June 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 45–52.
67. Fernando, O. OMESHF/NeT2I: NeT2I (Network to Image) Is a Python Package for Converting Network Traffic Data into Image Representations. Available online: https://github.com/omeshF/NeT2I (accessed on 17 September 2025).
68. Fernando, O. NeT2I. Available online: https://pypi.org/project/net2i/ (accessed on 17 September 2025).
69. Fernando, O. OMESHF/I2NeT: I2NeT (Image to Network) Is the Reverse Companion to NeT2I. It Decodes RGB Images Created from Network Traffic Data Back into Structured Tabular Form. Available online: https://github.com/omeshF/I2NeT (accessed on 17 September 2025).
70. Fernando, O. I2NeT. Available online: https://pypi.org/project/i2net/ (accessed on 17 September 2025).

71. Fernando, O. OMESHF/CiNeT: CiNeT (Classify in Network Transformation) Is a Project That Provides Dynamic CNN Classifiers with GPU/RAM Monitoring in Both TensorFlow/Keras and PyTorch. Available online: https://github.com/omeshF/CiNeT (accessed on 17 September 2025).
72. Darst, B.F.; Malecki, K.C.; Engelman, C.D. Using recursive feature elimination in random forest to account for correlated variables in high dimensional data. *BMC Genet.* **2018**, *19*, 65. [CrossRef] [PubMed]
73. Sedgewick, R.; Wayne, K. *Algorithms: Part I*; Addison-Wesley Professional: Boston, MA, USA, 2014.
74. Blackburn, S.M.; Garner, R.; Hoffmann, C.; Khang, A.M.; McKinley, K.S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S.Z.; et al. The DaCapo benchmarks: Java benchmarking development and analysis. In Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications, Portland, OR, USA, 22–26 October 2006; pp. 169–190.
75. Barbarossa, S.; Sardellitti, S.; Ceci, E. Joint Communications and Computation: A Survey on Mobile Edge Computing. *IEEE Signal Process. Mag.* **2018**, *35*, 36–58.
76. Gorelick, M.; Ozsvald, I. *High Performance Python: Practical Performant Programming for Humans*; O'Reilly Media: Sebastopol, CA, USA, 2020.
77. Buttazzo, G.C. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2011; Volume 24.
78. Sencan, E.; Kulkarni, D.; Coskun, A.; Konate, K. Analyzing GPU Utilization in HPC Workloads: Insights from Large-Scale Systems. In Proceedings of the Practice and Experience in Advanced Research Computing (PEARC '25), New York, NY, USA, 20–24 July 2025; pp. 1–10. [CrossRef]
79. Ganguly, D.; Mofrad, M.H.; Znati, T.; Melhem, R.; Lange, J.R. Harvesting Underutilized Resources to Improve Responsiveness and Tolerance to Crash and Silent Faults for Data-Intensive Applications. In Proceedings of the 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, 25–30 June 2017; pp. 536–543. [CrossRef]
80. Ananthanarayanan, G.; Ghodsi, A.; Shenker, S.; Stoica, I. Why let resources idle? Aggressive cloning of jobs with Dolly. In Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 12), Boston, MA, USA, 12–13 June 2012.
81. Govindan, S.; Sivasubramaniam, A.; Urgaonkar, B. Benefits and limitations of tapping into stored energy for datacenters. In Proceedings of the 38th Annual International Symposium on Computer Architecture, San Jose, CA, USA, 4–8 June 2011; pp. 341–352.
82. Papadimitriou, C.H. Computational complexity. In *Encyclopedia of Computer Science*; Wiley: Hoboken, NJ, USA, 2003; pp. 260–265.
83. Goodfellow, I.; Bengio, Y.; Courville, A. *Deep Learning*; MIT Press: Cambridge, MA, USA, 2016.
84. Tambon, F.; Nikanjam, A.; An, L.; Khomh, F.; Antoniol, G. Silent Bugs in Deep Learning Frameworks: An Empirical Study of Keras and TensorFlow. *arXiv* **2021**, arXiv:2112.13314. [CrossRef]
85. Zhang, Y.; Chen, Y.; Cheung, S.C.; Xiong, Y.; Zhang, L. An Empirical Study on TensorFlow Program Bugs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018; ACM: New York, NY, USA, 2018; pp. 129–140.
86. Zhang, H.; Zhang, L.; Jiang, Y. Overfitting and underfitting analysis for deep learning based end-to-end communication systems. In Proceedings of the 2019 11th International Conference on Wireless Communications and Signal Processing (WCSP), Xi'an, China, 23–25 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 1–6.
87. Fernando, O.A.; Xiao, H.; Spring, J. Developing a Testbed with P4 to Generate Datasets for the Analysis of 5G-MEC Security. In Proceedings of the 2022 IEEE Wireless Communications and Networking Conference (WCNC), Austin, TX, USA, 10–13 April 2022; IEEE: Piscataway, NJ, USA, 2022; pp. 2256–2261.
88. Fernando, O.A.; Xiao, H.; Spring, J.; Che, X. A Performance Evaluation for Software Defined Networks with P4. *Network* **2025**, *5*, 21. [CrossRef]
89. Fernando, O.A. Real-Time Application of Deep Learning to Intrusion Detection in 5G-Multi-Access Edge Computing. Ph.D. Thesis, University of Hertfordshire, Hatfield, UK, 2024.