國立臺北科技大學

資訊安全碩士學位學程

碩士學位論文

Master of Science in Information Security

Master Thesis

S2GE-NIDS: A hybrid architecture combining structured semantics and generation embedded network intrusion detection system in IoT

研究生：周玟萱

Researcher: Wen-Hsuan, Chou

指導教授：陳香君博士

Advisor: Shiang-Jiun Chen, Ph.D.

July 2025

國立臺北科技大學
研究所碩士學位論文口試委員會審定書

本校 資訊管理碩士學位學程 研究所 周玫萱 君

所提論文，經本委員會審定通過，合於碩士資格，特此證明。

學位考試委員會

委　　員：吳和庚

馬奕薇

陳香君

指導教授：陳香君

所　　長：

中 華 民 國　一百一十四年　七　月二十二日

Abstract

Keyword: IoT Security, Information Security, Anomaly Detection, Multilayer Perceptron, Semantic Vector

As network environments become increasingly complex and dynamic, traditional intrusion detection methods struggle to keep pace with evolving threats and high-volume traffic. This paper proposes an efficient anomaly detection framework that leverages hash-based token embedding and a lightweight multi-layer perceptron (MLP) for the semantic representation of network flows. By transforming feature values into semantic tokens and utilizing a hashing trick for embedding lookup, our approach enables scalable and robust processing without maintaining an explicit vocabulary. The resulting embedding vectors are flattened and processed by the MLP to produce semantic vectors, which are clustered using a center loss strategy for unsupervised anomaly detection. Experimental results on the public CICIoT2024 benchmark dataset, comprising 238,687 samples and 10 network traffic features, demonstrate that our method achieves competitive accuracy with significantly improved computational efficiency compared to traditional attention-based models. Specifically, the model attains an F1-score of 0.88 and an Area Under Curve (AUC) of 0.98, with inference times of 0.8 milliseconds per sample, outperforming baseline methods such as Isolation Forest, One-Class SVM, and AutoEncoder in both detection accuracy and resource consumption. This lightweight and interpretable framework is well-suited for deployment in resource-constrained IoT environments.

Acknowledgements

在本論文即將完成之際，回顧這一路走來的點滴，我滿懷感激之情，衷心感謝所有在我學術旅程中給予幫助、指導與支持的每一位人士。

首先，最深的謝意獻給我的指導教授陳香君博士。教授您不僅以渾厚的學識引領我探索學術的海洋，更以耐心與細心，指點我走過研究上的重重難關。無數個挑燈夜戰的時刻，因有您的鼓勵與鞭策而充滿力量，讓我始終保持堅定的信念，完成這篇論文。您的嚴謹治學與溫暖關懷，將是我永遠的榜樣。

感謝我最親愛的家人父母與妹妹，是你們無條件的愛與支持，築起我心中堅不可摧的避風港。每當疲憊與迷惘襲來，是你們的理解與陪伴讓我重拾勇氣。你們的鼓勵是我前進路上的光明，無論多遠、多難，我都願意為了你們的期盼努力向前。

特別感謝周啟弘先生，感謝您在工作上的全力支持，讓我能夠在職涯中穩步成長。並由衷感謝胡智強先生，感謝您日常的指導與協助，使我在忙碌的工作與研究之間找到平衡，您耐心的教導令我受益良多。還有王孟昇先生以及其他同事，感謝你們的協助與合作。

在此亦要感謝所有研究所的夥伴們，謝謝你們一路相伴，無論是在課堂上的深入討論、論文中的思想碰撞，抑或是日常生活中點滴的分享，你們的熱忱與智慧如同明燈，照亮我求知的道路。與你們共同經歷的每一刻，都深深啟發我、激勵我，讓這段求學旅程變得溫暖且充滿力量。這些寶貴的回憶，將永遠珍藏於我心中，成為我人生中最珍貴的財富。

最後，感謝我摯愛的朋友們，是你們的鼓勵與支持，讓我在學術旅程中能保持初心與熱情。每一次的傾聽與陪伴，都讓我感受到溫暖與力量。

這篇論文凝聚了所有人的關懷與助力，我將這份感激深藏於心，銘記於行，未來必將以更加堅定的步伐，踏上新的征程。謹此獻上最誠摯的謝意。

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

Driven by the rapid advancement of digital transformation and smart infrastructure, the Internet of Things (IoT) [1] has emerged as a cornerstone of next-generation information technology. Through the integration of sensors, embedded devices, communication modules, and platform software, IoT enables physical objects to communicate in real time and generate massive volumes of data. These data streams support a broad range of applications, such as smart manufacturing, intelligent transportation, remote healthcare, and smart homes, yielding substantial economic and societal value.

However, as the number of connected devices increases and deployment scenarios become more complex, IoT systems face unprecedented cybersecurity challenges. Many IoT devices are resource-constrained, infrequently updated, and difficult for users to manage [2]. With limited encryption and a lack of monitoring mechanisms, these devices become prime targets for cyber intrusions and attacks. As a result, intrusion detection systems (IDS) designed for IoT must not only be accurate, but also lightweight, efficient, and scalable. Effectively identifying abnormal behaviors and hidden threats in IoT network traffic has therefore become a pressing research priority.

Furthermore, existing intrusion detection technologies often struggle to adapt to evolving threats. While deep learning approaches such as Word2Vec [3] and Transformer-based models [4] have demonstrated semantic learning capabilities, they also introduce critical drawbacks: large vocabulary requirements, high computational complexity, and limited flexibility in dynamic or resource-constrained environments. These limitations make them unsuitable for direct deployment on edge-based or embedded IoT systems.

To effectively capture the latent structure and semantic information within the data, embedding techniques have recently been introduced into anomaly detection. By mapping discrete raw features into continuous vector spaces, embedding methods compress high-dimensional data into low-dimensional semantic vectors, thereby enhancing the model's ability to learn and recognize complex patterns.

In IoT anomaly detection, semantic embeddings can represent underlying relationships

among device behaviorss, we propose Structured Semantics and Generation Embedded Network Intrusion Detection System (S2GE-NIDS) a lightweight, interpretable anomaly detection framework designed for IoT environments. S2GE-NIDS combines hash-based token embedding with a multi-layer perceptron (MLP) [5] model and introduces a linked-list mechanism to mitigate hash collisions inherent to non-cryptographic hash functions such as MurmurHash3 [6]. This design enables efficient feature encoding while avoiding the need to maintain a large vocabulary.

In our approach, network packets are first transformed into semantic tokens and encoded using hash-based indexing. The resulting embedding vectors are concatenated into a single, fixed-length semantic vector, which is processed by an MLP and projected near a learned semantic center. Any significant deviation from this center measured by Mahalanobis distance [7] is classified as a potential anomaly.

The proposed S2GE-NIDS framework offers several advantages over conventional intrusion detection systems. First, it uses a hash-based embedding approach; The propose of the hash table in our method is efficiently searching data and storage and the combination of double hashing and linked list serial nodes can compress and pre-allocate to save index space and reduce dynamic memory configuration costs. In addition, a fixed-size index after modulo operation is used to ensure that the size of the embedded table is controllable, taking into account both storage space and hashing uniformity. This strategy has better space complexity than directly hashing the entire key-value pair.

Second, the model provides a mathematically interpretable anomaly scoring mechanism by integrating Mahalanobis distance, which quantifies how far a sample deviates from the learned distribution of normal behavior. This not only improves detection accuracy but also enables explainable results.

Third, the system is lightweight and highly efficient, relying on simple MLP-based [8] encoding instead of complex deep architectures, making it well-suited for deployment in real-time or resource-constrained environments such as edge devices in IoT networks. Therefore, this double hashing architecture not only effectively optimizes the hash function combination, makes the hash more uniform, and reduces the chain length. It also improves the stability and

efficiency of hash embedding, laying a good foundation for the subsequent anomaly detection model based on semantic vectors.

The structure of this paper is organized as follows. Chapter 2 provides the background knowledge related to S2GE-NIDS. Chapter 3 presents the architecture and methodology of the proposed framework, detailing the design and each module. Chapter 4 describes the implementation setup and steps, Chapter 5 provides the experimental results. Finally, Chapter 6 concludes and outlines for future research.

Chapter 2 Related Work

This section will introduce the relevant basic knowledge, including existing IoT network intrusion detection methods, tokenization, hash embedding, multi-layer perceptron, semantic vectors, and Mahalanobis distance.

2.1 Network Intrusion Detection System in IoT

In recent years, the rapid expansion of Internet of Things (IoT) devices has driven increased research attention towards designing effective network intrusion detection systems (NIDS) that address the unique challenges of IoT environments. These challenges include managing large volumes of heterogeneous traffic, coping with resource-constrained devices, and adapting to evolving attack patterns.

Traditional machine learning algorithms such as Support Vector Machines (SVM), k-Nearest Neighbors (kNN), Decision Trees, and Random Forests [9] have been widely adopted in early intrusion detection systems (IDS) due to their simplicity, interpretability, and relatively low computational requirements. These methods generally rely on handcrafted features and assume static data distributions, which limits their adaptability to evolving attack patterns. Moreover, the performance of such models heavily depends on feature selection quality and preprocessing strategies, making them less flexible in dynamic IoT environments.

Several approaches have been proposed to tackle these issues. For instance, Kharoubi et al. [10] developed a convolutional neural network (CNN)-based detection system tailored for IoT security. By leveraging CNN layers to extract spatial features from network traffic, their model achieved strong classification performance on benchmark datasets such as Canadian Institute for Cybersecurity Internet of Things (CICIoT2023) [11] and Canadian Institute for Cybersecurity Internet of Medical Things (CICIoMT2024) [12]. While demonstrating excellent precision and recall in both binary and multi-class scenarios, this approach has limitations in capturing temporal dependencies within packet sequences and requires large amounts of labeled data due to its supervised learning nature.

Complementing deep learning methods, Ashraf et al. [13] proposed a real-time IoT Network Intrusion Detection System (INIDS) based on traditional machine learning classifiers, evaluated on the BoT-IoT dataset. Their comparative study of seven algorithms—including Random Forest, Artificial Neural Networks (ANN), and Support Vector Machines (SVM)—showed that Random Forest and ANN achieved superior accuracy and robustness. However, these models heavily rely on manual feature engineering and face challenges in adapting to novel threats, a crucial factor in the rapidly evolving IoT landscape.

Recognizing the importance of feature design, Lee and Stolfo [14] emphasized feature extraction techniques to build accurate and efficient intrusion detection models. Their work demonstrated that appropriate feature selection is vital, particularly when handling large datasets or emerging attack types.

Together, these studies highlight the complementary strengths and limitations of various NIDS approaches in IoT, motivating the development of methods that balance automatic feature learning, temporal modeling, and adaptability to new threats.

Table 2.1 [15] provides a consolidated overview of eight widely adopted features commonly utilized in anomaly detection across both academic research and industrial applications.

2.2 Tokenization Technique in IoT Application

Tokenization is the process of converting raw packet data or traffic feature fields into semantically meaningful token sequences, thereby enabling anomaly detection models to perform contextual understanding and analysis. This technique facilitates the modeling of complex patterns in network traffic by translating low-level features into high-level representations.

Several representative studies have explored tokenization and related embedding techniques for improving IoT anomaly detection. For example, Naeem and Alalfi [16] proposed vectorized deep learning algorithms, Token2Vec and Flow2Vec, demonstrating significant improvements in prediction accuracy. Similarly, Li et al. [17] applied Word2Vec for TF-IDF vectorization on HTTP traffic, combining it with Boosting algorithms to enhance detection accuracy, reduce false alarms, and maintain low computational cost.

Table 2.1 Common Anomalous Features in IoT Network Traffic

Feature	**Description**
Destination Port	The port number in a network packet received by the destination host, used to identify the communication entry point of an application or service.
Protocol Type	Used to indicate the type of communication protocol used by the packet, such as TCP, UDP, ICMP, etc.
Duration / flow_duration	The length of time a network connection session lasts, the number of seconds from the start to the end of the connection.
Packet Length	The size of a packet is usually measured in bytes and refers to the amount of data in a single packet.
Source IP / Destination IP	The source and destination IP addresses of a packet indicate the network locations of the sender and receiver, respectively.
Flow Bytes per Second	The total data flow through the network connection per unit time, measured in bytes per second (Bytes/s).
TCP Flags	The control bit flag in the TCP packet indicates the status of the packet or the control message, such as SYN for connection request and FIN for connection termination.
Number of Connections	The number of network connections established by a single source IP within a specified period of time, used to measure connection activity.

Advancing this line of work, Gao et al. [18] introduced a CNN-BiLSTM intrusion detection method that integrates tokenization with an attention mechanism to automatically extract sequence features from traffic data, thereby reducing the need for manual feature engineering. Their experiments on vehicle network datasets achieved near-perfect detection accuracy, outperforming existing methods.

Collectively, these studies confirm the effectiveness of tokenization strategies in IoT anomaly detection. By transforming heterogeneous traffic attributes into unified embedding vectors, these methods empower models to learn behavioral patterns across packet and application layers, which is crucial for building scalable and accurate intrusion detection systems suited for diverse and dynamic IoT environments.

Table 2.2 shows the comparison of some features in anomaly detection using Tokenization. For example, Protocol = TCP only retains the field name and value, and directly discards other symbols and spaces.

Table 2.2 Examples of Field-Value Tokenization in IoT Network Traffic Data

Feature Field	**Tokenized Representation**
Protocol = TCP	Protocol:TCP
Destination Port = 80	DstPort:80
Flow Duration = 0.32817	FlowDuration:0.32817
Source IP = 192.168.0.1	SrcIP:192.168.0.1
Payload Bytes = 56	PayloadBytes:56
Packet Count = 10	PacketCount:10
Flag = ACK	Flag:ACK
Destination IP = 10.0.0.5	DstIP:10.0.0.5

2.3 Hash Embedding in Anomaly Detection

In [19], Argerich et al. introduced the use of feature hashing through the Hash2Vec technique. By employing two hash functions—one mapping context words to fixed-dimension word vectors and the other assigning sign values—they overlay weighted co-occurrence frequencies directly onto these vectors without requiring any training process, significantly reducing memory usage. Their results show performance comparable to mainstream embedding methods like GloVe.

Building upon embedding techniques, Luo et al. [20] proposed a method that integrates a Transformer encoder with masking and an LSTM to capture temporal dependencies, training exclusively on normal data. This approach enhances feature extraction and sequential modeling to detect covert anomalies more effectively. Experimental results indicate improvements of 2.3

In [19], Argerich et al. a feature hashing method to quickly build word vectors without training. The source and destination IP and port addresses are mapped into low-dimensional vectors, and their pairwise cosine distances are computed. This approach significantly mitigates the sparsity and high dimensionality issues inherent in the original high-dimensional categorical features.

In [21], Huang proposed NIDS-GPT Multi-dimensional embedding representation, improves the understanding of the internal structure of the packet, including word embedding, number position embedding and column position embedding, to improve the understanding of the internal structure of the packet. The training goal is to predict the next word in the sequence,

and the last word is the label of the packet to complete the classification task.

Collectively, these studies affirm that hash embedding techniques provide scalable and efficient representations for sparse or categorical IoT traffic features, facilitating rapid and accurate anomaly detection under constrained memory and computational resources.

2.4 Multi-Layer Perceptron in Anomaly Detection

Multi-Layer Perceptrons (MLPs) [22] have been widely adopted in anomaly detection due to their strong capability to model nonlinear relationships between input features and latent patterns. Unlike traditional statistical models that depend on preset thresholds or assumptions about data distributions, MLPs learn complex, high-dimensional feature representations directly from data.

In recent years, MLP-based approaches have been applied across diverse domains such as network security [23], industrial control systems [24], and IoT environments [25]. These models typically consist of multiple fully connected layers activated by nonlinear functions like ReLU or sigmoid, enabling the hierarchical learning of semantic features. The resulting outputs are leveraged to differentiate normal from abnormal behavior through reconstruction errors, classification scores, or learned distance measures.

For example, Elghamrawy et al. [26] implemented a deep learning-based intrusion detection system using an MLP configured for multi-class classification with cross-entropy loss, achieving significant improvements in both efficiency and accuracy.

Similarly, Esmaeili et al. [27] proposed an autoencoder-based prediction model combined with kernel density estimation to autonomously detect anomalies, demonstrating robust detection performance given sufficient normal training data.

Further, Shone et al. [28] introduced a hybrid deep learning method combining stacked autoencoders with an MLP classifier, evaluated on the NSL-KDD dataset, attaining an accuracy of 85.42

Moreover, Ahmad et al. [29] proposed an anomaly detection approach leveraging mutual information within deep neural networks. Their comparative study among different deep learn-

ing architectures showed accuracy improvements of 0.57–2.6

Collectively, these studies underscore MLPs as a strong foundational model for IoT anomaly detection pipelines, especially when integrated with effective feature engineering and regularization strategies.

2.5 Semantic Vector in IoT Anomaly Detection

Semantic vector representations, originally developed in natural language processing (NLP) [30], have been increasingly adopted in anomaly detection tasks due to their capacity to encode complex contextual relationships into fixed-length embeddings. Mikolov et al. [31] proposed the Skip-gram model, which improves training efficiency and vector quality by subsampling high-frequency words and employing negative sampling. To address limitations related to word order sensitivity and the inability to represent idiomatic phrases, an effective phrase recognition method was introduced, enabling the model to learn accurate phrase-level embeddings.

Building on these advances, Wang et al. [32] proposed LogEvent2vec, an offline feature extraction framework that applies Word2Vec to capture correlations between log events, directly converting logs into vector representations. They trained supervised classifiers, including random forest, naive Bayes, and neural networks, for anomaly detection. Experimental results indicate that LogEvent2vec reduces computation time by approximately thirty-fold compared to traditional Word2Vec, while achieving improved accuracy. In particular, the combination of barycentric features with random forests yielded the highest F1-score, whereas tf-idf features combined with naive Bayes attained the lowest computational cost.

These semantic embeddings capture latent associations between fields and behaviors, enabling downstream models to detect subtle deviations from normal patterns. For example, Pan et al. [33] introduced FlowBERT, a Transformer-based model for encrypted traffic classification that extracts semantic features from both payload content and packet length sequences. Their balanced data sampling strategy during pre-training improved model robustness and classification performance beyond existing methods.

More recently, Hariharan et al. [34] developed the Subword Encoder Neural Network

(SEN), which employs an attention-based encoder to learn semantic embeddings at the subword level. The model effectively captures novel log messages and integrates a novel Naive Bayesian Feature Selector (NBFS) for interpretable feature distillation. Their approach significantly outperforms state-of-the-art methods.

2.6 Mahanobis Distance in IoT Anomaly Detection

Mahalanobis distance [7] was first proposed by Indian statistician Prasanta Chandra Mahalanobis. It proposed a method to measure the ”distance” between points and multidimensional statistical distributions, thus breaking through the limitation of Euclidean distance that cannot adjust scale and correlation. Experiments show that when Mahalanobis distance exceeds the normal threshold, abnormal behaviors such as failures or unexpected operations can be detected. The proposed of following classic formula as below.

$$d_M(\mathbf{x}) = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})} \tag{2.1}$$

The equation $d_M(\mathbf{x})$ denotes the Mahalanobis distance, where $\mathbf{x}$ is the observation vector, $\boldsymbol{\mu}$ is the mean vector, and $\boldsymbol{\Sigma}$ is the covariance matrix of the distribution.

In [35], Alsalman proposed FusionNet, a hybrid model that leverages the strengths of multiple machine learning algorithms, including random forests, k-nearest neighbors, support vector machines, and multi-layer perceptrons, to enhance anomaly detection performance. FusionNet’s architecture achieves high accuracy and precision, consistently outperforming individual models in terms of accuracy, precision, recall, and F1 score across multiple datasets.

Similarly, Ngo et al. [36] introduced an anomaly detection method based on Mahalanobis distance. Devices triggered simultaneously are grouped by key actuators, and the distances between configuration states are computed using a k-nearest neighbors model. A control algorithm is applied to estimate the proportion of device failures, effectively reducing the false alarm rate. Experimental results demonstrate that this approach maintains a high detection rate with low false alarms and fast response times.

More recently, Pillai [37] proposed two anomaly detection techniques employing Maha-

lanobis distance and Autoencoder models. Their experiments reveal that Autoencoders provide superior generalization and accuracy across varying operating conditions, whereas the Mahalanobis-based method offers computational efficiency and high interpretability. This work highlights the promise of TinyML technologies for real-time anomaly detection applications.

Chapter 3 Methodology

In this section, we will introduce the Structured Semantics and Generation Embedded Network Intrusion Detection System (S2GE-NIDS) architecture and detail its operational workflow, clearly delineating each step from preprocess module through embedding module and Mahalanobis distance anomaly processes.

3.1 Architecture

The architecture of S2GE-NIDS is presented as Figure 3.1. The further description will begin in the section 3.1.1.

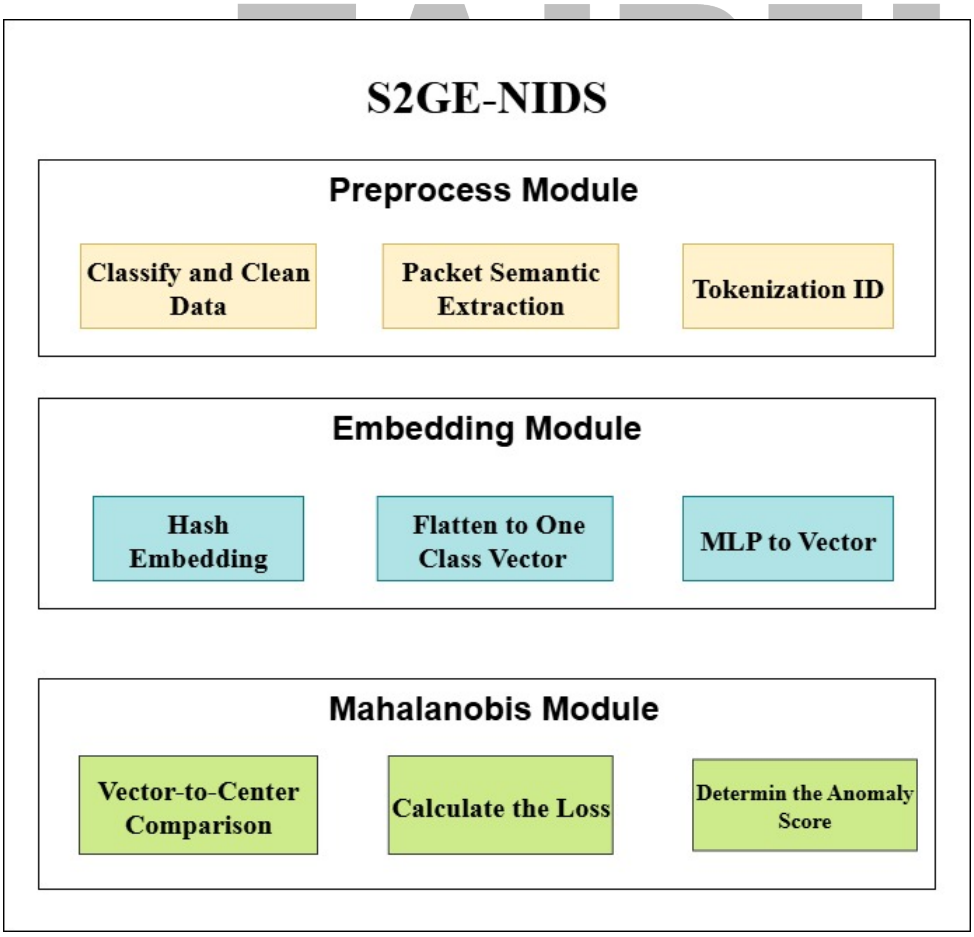

Figure 3.1 Architecture of S2GE-NIDS

3.1.1 Preprocess Module

In the preprocessing phase, as shown in Figure 3.3, we will perform the following steps: data file classification and cleaning, packet semantic extraction, and tokenization. These steps are designed to transform raw network traffic into structured representations suitable for semantic embedding and anomaly detection.

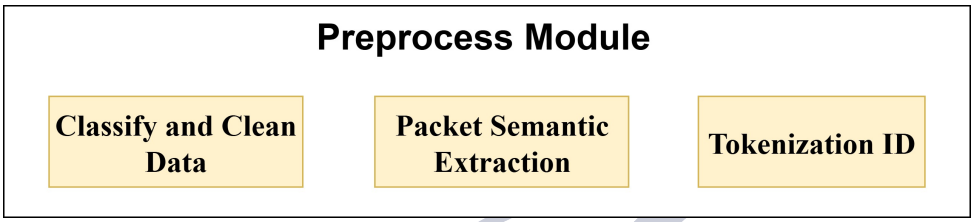

Figure 3.2 Architecture of Preprocess Module

3.1.1.1 Classify and Clean Data

In S2GE, the dataset is partitioned into three distinct subsets to facilitate effective model training and evaluation: 70% of the data is allocated for training, 15% for validation, and the remaining 15% for testing.

```python
# --- Step 1: Data segmentation and label generation ---
def split_and_save_labels(input_csv='Benign_train.csv'):
    df = pd.read_csv(input_csv)
    label_col = df.columns[-1]
    labels = df[label_col].apply(lambda x: 0 if pd.isna(x) or str(x).strip() == '' else 1)

    train_val_df, test_df, train_val_labels, test_labels = train_test_split(
        df, labels, test_size=0.15, random_state=42, stratify=labels)
    val_ratio = 0.1765
    train_df, val_df, train_labels, val_labels = train_test_split(
        train_val_df, train_val_labels, test_size=val_ratio, random_state=42, stratify=train_val_labels)

    train_labels.to_csv('train_labels.csv', index=False, header=[label_col])
    val_labels.to_csv('val_labels.csv', index=False, header=[label_col])
    test_labels.to_csv('test_labels.csv', index=False, header=[label_col])

    train_df.to_csv('train_data.csv', index=False)
    val_df.to_csv('val_data.csv', index=False)
    test_df.to_csv('test_data.csv', index=False)
    print("Labels and split data saved.")
```

Figure 3.3 Data split process

The first step in the preprocessing pipeline involves selecting and filtering data files to ensure their suitability for subsequent analysis. In this study, network traffic is collected and stored in the Comma-Separated Values (CSV) format, a widely adopted and flexible tabular data structure. CSV files are particularly well-suited for structured data representation due to

their ease of parsing, compact storage, and seamless integration with mainstream data analysis libraries such as pandas, and NumPy in Python.

After selecting the data format, the raw data are merged into a unified DataFrame and subjected to a series of cleaning procedures. First, all column names are normalized by removing extraneous whitespace and standardizing naming conventions where necessary to ensure consistency across feature dimensions. Next, rows containing missing or undefined entries are removed to prevent bias in downstream training. Finally, columns consisting solely of zeros are discarded.

During this stage, the resulting dataset serves as the foundation for the subsequent tokenization and embedding stages.

3.1.1.2 Packet Semantic Extraction

Packet semantic extraction refers to the process of identifying and transforming raw packet-level attributes into semantically meaningful representations that facilitate accurate anomaly detection. As summarized in Table 2.1, a set of representative features such as *Destination Port*, *Protocol Type*, *Flow Duration*, *Packet Length*, *Flow Bytes per Second*, *TCP Flag Counts*, and *Connection Count* have been consistently validated in prior studies as effective indicators of anomalous or malicious traffic behavior.

3.1.1.3 Tokenization ID

Following the feature extraction phase, the structured network traffic attributes undergo a tokenization process, where in categorical feature fields are transformed into discrete, semantically interpretable string tokens. These tokens form the foundation for subsequent vector embedding. The dataset comprises multiple structured records, each containing various categorical attributes such as `Destination Port`, `Protocol Type`, and `Source IP Address` that capture the behavioral characteristics of individual traffic flows. By converting these symbolic fields into tokenized string representations, the module preserves both the identity and contextual meaning of traffic patterns, thereby enabling more effective semantic encoding in later stages.

Table 3.1 shows the tokenization method employed, where each feature is transformed by concatenating its field name and corresponding value with a colon separator to form a token. For example, the field name ”DestinationPort” combined with the value ”80” becomes the token ”DestinationPort:80”; similarly, ”FlowDuration” with ”0.32817” becomes ”FlowDuration:0.32817”, and ”ProtocolType” with ”TCP” becomes ”ProtocolType:TCP”.

Table 3.1 Example of Tokenization Process

Field Name	Field Value	Token
Destination Port	80	DestinationPort:80
Flow Duration	0.32817	FlowDuration:0.32817
Protocol Type	TCP	ProtocolType:TCP

3.1.2 Embedding Module

To convert network packet features into vector representations that can be processed by the module, this architecture employs an efficient embedding module. Figure 3.4 shows the entire process includes hash embedding, flatten to a one-class vector, and using a multi-layer perceptron (MLP) to vector.

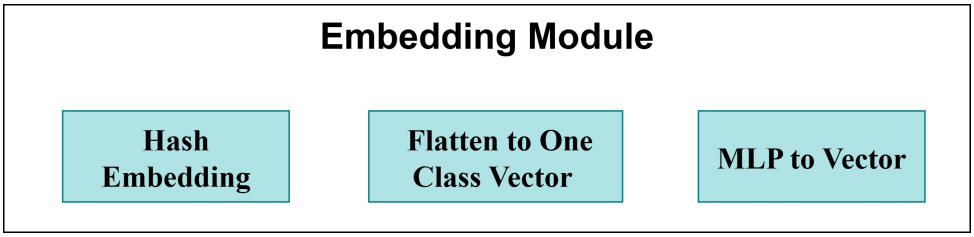

Figure 3.4 Architecture of Preprocess Module

3.1.2.1 Hash Embedding

Hash embedding [38] is a lightweight vectorization technique that utilizes non-cryptographic hashing to encode tokenized field-value pairs into fixed-size, trainable embeddings. In this study, we adopt the MurmurHash3 algorithm, an efficient and widely used hash function, to map each token to a specific position in the embedding table. Its advantages include fast com-

putation, uniform distribution, and language-independent implementation, making it well-suited for scalable anomaly detection in IoT environments.

In this study, the embedding table is theoretically designed as a fixed-size matrix of dimension 223×223, where each cell contains an 8-dimensional embedding vector representing tokens hashed via the MurmurHash3 function and mapped through modulo operation. This fixed-size matrix structure facilitates uniform feature distribution and ensures a consistent input dimension for downstream machine learning models.

In the design of the hash embedding table, the choice of the table size plays a critical role in the efficiency and effectiveness of the hashing process. We select the table size as a prime number, specifically 223, resulting in an embedding table of size $223 \times 223 = 49,729$.

According to Theorem 11.9 [39], when storing n keys in a hash table of size $m = n^2$ using a hash function h randomly selected from a universal class of hash functions, the probability of any collisions is less than $\frac{1}{2}$. Moreover, it is well established in the literature that choosing a prime number as the hash table size reduces the likelihood of collisions and promotes a uniform distribution of hash values across the table.

Considering the practical memory constraints typical in Internet of Things (IoT) devices many of which feature memory capacities around 256 KB or multiples thereof the selection of a prime number close to 256, such as 223, represents a strategic balance between memory efficiency and hashing performance. This design choice facilitates efficient deployment in resource-constrained IoT environments without incurring excessive memory overhead.

```
def get_embedding(self, field, value):
    i = self.hash_token(field)
    j = self.hash_token(value)
    node = self.buckets[i][j]
    while node:
        if node.field == field and node.value == value:
            return node.vector
        node = node.next
    new_vector = np.random.uniform(-0.05, 0.05, size=self.embed_dim).astype(np.float32)
    new_node = LinkedListNode(field, value, new_vector)
    new_node.next = self.buckets[i][j]
    self.buckets[i][j] = new_node
    return new_vector
```

Figure 3.5 Initial the Embedding Table

However, due to the inevitability of hash collisions where multiple distinct tokens map to the same index using a static three-dimensional matrix can lead to vector overwriting and information loss. To address this, the embedding table is implemented in practice as a dynamic

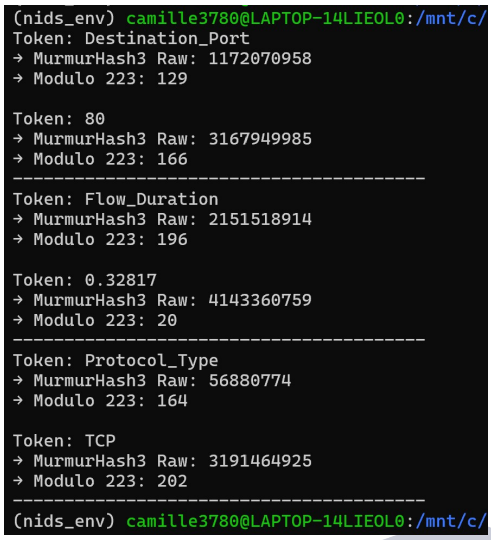

Figure 3.6 Hash Embedding Process

linked list structure using a dictionary-based approach. This implementation allows flexible storage of multiple embedding vectors corresponding to collided tokens at the same index, thus preserving embedding integrity and optimizing memory usage.

During the output stage, the embedding vectors corresponding to all tokens within a single data row are flattened into a one-dimensional vector, which serves as input to subsequent models such as MLP. Although this practical implementation differs from the theoretical fixed three-dimensional matrix, the overall architecture and core objectives remain consistent, aiming to provide an efficient and representative semantic vector encoding.

This divergence between theory and implementation reflects a trade-off between resource constraints and collision handling. The dynamic linked list structure offers scalability and ease of maintenance, enhancing both the usability and computational efficiency of the embedding table while ensuring the completeness of the vector representations.

To determine the target index for each token, each vector is initialized randomly and refined during module training as in Figure 3.5.

From the above example, as shown in Figure 3.6, the field name `Destination_Port` yields an index of 129, while its value `80` maps to 166; these indices are used to locate specific embedding vectors.

The process begins with a *Token ID* shown as Figure 3.7, such as the categorical feature `DestinationPort` and its corresponding value 80. This Token ID is first processed through a

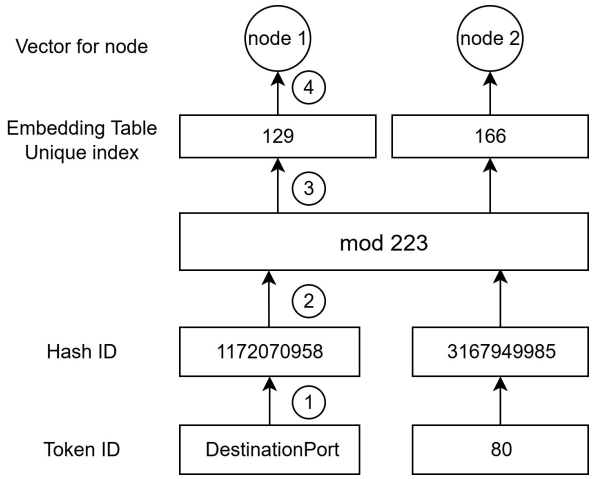

Figure 3.7 Double Hashing with Linked List for Embedding Storage

hash function ① to generate a large integer hash value (e.g., 3167949985). This step effectively converts high-dimensional sparse features into uniformly distributed integers, reducing feature sparsity and enabling efficient indexing. Formally:

$$\text{Hash ID} = \text{Hash}(\text{Token ID}) \tag{3.1}$$

To map this hash value into the embedding table, a modulo operation is applied using the table size m ② (here, $m = 223$) yielding a unique index ③ within the embedding table (e.g., 166). This index corresponds to a primary slot in the embedding table.

Due to the possibility of hash collisions where multiple distinct tokens may hash to the same modulo index a *dynamic linked list* ④ is employed to handle such conflicts. When a collision occurs, additional embedding vectors are stored in linked nodes associated with the initial index.

In this process, each embedding vector is identified by its unique index after modulo operation (e.g., 129 for another hash value). The system checks the embedding table entry at this index:

- If the table is free or matches the token, the embedding vector is stored or retrieved directly.
- If the table is occupied by a collision, a pointer will link to the next node in the list.

As shown in Figure 3.8, this illustrates the double hashing technique combined with a

```python
# --- Step 2: Hash Embedding Table with Linked List ---
class LinkedListNode:
    def __init__(self, field, value, vector):
        self.field = field
        self.value = value
        self.vector = vector
        self.next = None

class HashEmbeddingTableWithLinkedList:
    def __init__(self, table_size=223, embed_dim=8):
        self.table_size = table_size
        self.embed_dim = embed_dim
        self.buckets = [[None for _ in range(table_size)] for _ in range(table_size)]

    def hash_token(self, token):
        return mmh3.hash(str(token), seed=0) & 0xffffffff % self.table_size

    def get_embedding(self, field, value):
        i = self.hash_token(field)
        j = self.hash_token(value)
        node = self.buckets[i][j]
        while node:
            if node.field == field and node.value == value:
                return node.vector
            node = node.next
        new_vector = np.random.uniform(-0.05, 0.05, size=self.embed_dim).astype(np.float32)
        new_node = LinkedListNode(field, value, new_vector)
        new_node.next = self.buckets[i][j]
        self.buckets[i][j] = new_node
        return new_vector

    def update_embedding(self, field, value, new_vector, lr=0.01):
        i = self.hash_token(field)
        j = self.hash_token(value)
        node = self.buckets[i][j]
        while node:
            if node.field == field and node.value == value:
                node.vector = node.vector * (1 - lr) + new_vector * lr
                return
            node = node.next
        new_node = LinkedListNode(field, value, new_vector.astype(np.float32))
        new_node.next = self.buckets[i][j]
        self.buckets[i][j] = new_node

    def encode_row(self, row):
        embeddings = [self.get_embedding(field, value) for field, value in row.items()]
        return np.concatenate(embeddings)

def generate_embedding_csv(df, embed_table, filename):
    embeddings = np.array([embed_table.encode_row(row) for _, row in df.iterrows()])
    pd.DataFrame(embeddings).to_csv(filename, index=False)
    print(f"Embedding saved to {filename}")
```

Figure 3.8 Dynamic Linked List of Embedding Table

dynamic linked list structure to efficiently store embedding vectors in a fixed-size embedding table.

3.1.2.2 Flatten to One Class Vector

The flatten operation concatenates the tokenized embeddings from each column into a single vector for input to the next stage of the pipeline. Table 3.2 shows example embedding vectors for individual tokens.

Table 3.2 Hash Values in the Embedding Table

Token	Hash	Mod=223	Embedding Vector
Destination_Port	1172070958	129	[0.091251, 0.332761, 0.725833, 0.472719, 0.937337, 0.429821, 0.046714, 0.055904]
80	3167949985	166	
Flow_Duration	2151518914	196	[0.012133, 0.628566, 0.035354, 0.851932, 0.437327, 0.968118, 0.040445, 0.434074]
0.32817	4143360759	20	
Protocol_Type	56880774	164	[0.044425, 0.674622, 0.362488, 0.078440, 0.640184, 0.862753, 0.158649, 0.322290]
TCP	3191464925	202	

Figure 3.9 shows the each feature-value pair in the row, its embedding vector is obtained

```python
class SimpleMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim=64, output_dim=2):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

Figure 3.9 Process of Flattern to One Class Vector

and appended to a list.

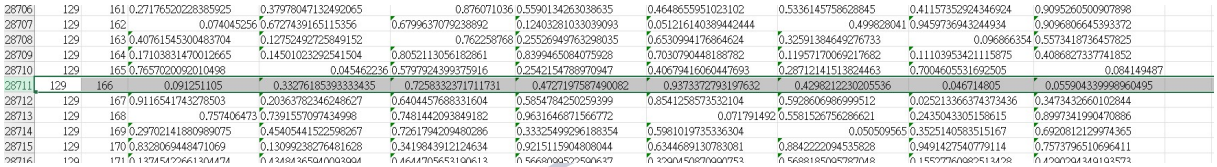

Figure 3.10 Example for DestinationPort:80 [129,166]

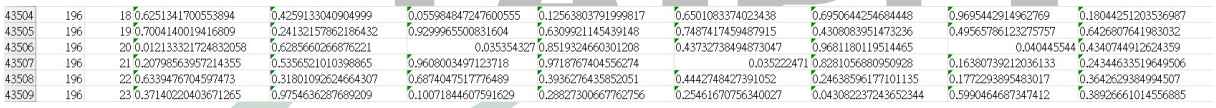

Figure 3.11 Example for FlowDuration:0.32817 [196,20]

Each row corresponds to a flattened embedding vector generated by concatenating hashed representations of multiple feature fields. The numerical values, ranging between 0 and 1, represent the individual dimensions of the embedding vectors, which have undergone normalization or activation transformations.

Figure 3.13 shows the packet vectors after flattening, where the highlighted rows (indices 129 and 166) exemplify samples with distinctive embedding patterns across multiple dimensions.

These embedding vectors serve as foundational inputs to deeper models, such as MLPs [5], facilitating the extraction of higher-level semantic features and improving detection accuracy.

3.1.2.3 MLP to Vector

To integrate the multiple field semantic vectors extracted from the embedding table for each packet into a unified semantic representation, a MLP encoder module is introduced. The

36550	164	200	0.6582938432693481	0.095187306	0.1939980387687683	0.2885007560253143	0.1695668548345565	0.348983168601989	0.128342568874359	0.461382567882537
36551	164	201	0.5737550854682922	0.47011199593544006	0.3546740710735321	0.48419007658958435	0.5509464740753174	0.50814873	0.269412637	0.040994618
36552	164	202	0.044425674	0.6746223568916321	0.36248844861984253	0.078440718	0.6401848196983337	0.8627535700798035	0.15864959359169006	0.32229000329971313
36553	164	203	0.360559881	0.8917117714881897	0.9075064659118652	0.7151344418525696	0.7058379650115967	0.9646387696266174	0.3842058777809143	0.076223455
36554	164	204	0.9508483409881592	0.07047566	0.6345201730728149	0.8366861343383789	0.9143674373626709	0.7681275010108948	0.32593291997909546	0.16650135815143585

Figure 3.12 Example for ProtocolType:TCP [164,202]

1	0.76848	0.08341	0.88703	0.21315	0.53863	0.75063	0.85113	0.39911	0.42838	0.7494	0.5016	0.94349	0.44325	0.19989
2	0.76848	0.08341	0.88703	0.21315	0.53863	0.75063	0.85113	0.39911	0.08805	0.38443	0.60255	0.37385	0.76467	0.86842
3	0.28626	0.22488	0.65893	0.26006	0.27456	0.2362	0.54049	0.07614	0.08805	0.38443	0.60255	0.37385	0.76467	0.86842
4	0.76848	0.08341	0.88703	0.21315	0.53863	0.75063	0.85113	0.39911	0.42838	0.7494	0.5016	0.94349	0.44325	0.19989
5	0.04443	0.6746223	0.3624884	0.07844	0.6401848	0.8627535	0.1586495	0.3222900	0.09125	0.33276	0.72583	0.47272	0.93734	0.42982
6	0.42838	0.7494	0.5016	0.94349	0.44325	0.19989	0.3919	0.97091	0.28626	0.22488	0.65893	0.26006	0.27456	0.2362
7	0.76848	0.08341	0.88703	0.21315	0.53863	0.75063	0.85113	0.39911	0.28626	0.22488	0.65893	0.26006	0.27456	0.2362
8	0.42838	0.7494	0.5016	0.94349	0.44325	0.19989	0.3919	0.97091	0.08805	0.38443	0.60255	0.37385	0.76467	0.86842
9	0.76848	0.08341	0.88703	0.21315	0.53863	0.75063	0.85113	0.39911	0.76848	0.08341	0.88703	0.21315	0.53863	0.75063

1	0.3919	0.97091	0.76848	0.08341	0.88703	0.21315	0.53863	0.75063	0.85113	0.39911	0.88797	0.97971	0.49996	0.23
2	0.61294	0.72873	0.08805	0.38443	0.60255	0.37385	0.76467	0.86842	0.61294	0.72873	0.5508	0.63625	0.42962	0.75
3	0.61294	0.72873	0.28626	0.22488	0.65893	0.26006	0.27456	0.2362	0.54049	0.07614	0.01463	0.50907	0.00913	0.9
4	0.3919	0.97091	0.28626	0.22488	0.65893	0.26006	0.27456	0.2362	0.54049	0.07614	0.01463	0.50907	0.00913	0.9
5	0.04671	0.0559	0.01213	0.62857	0.03535	0.85193	0.43733	0.96812	0.04045	0.43407	0.88797	0.97971	0.49996	0.23
6	0.54049	0.07614	0.08805	0.38443	0.60255	0.37385	0.76467	0.86842	0.61294	0.72873	0.5508	0.63625	0.42962	0.75
7	0.54049	0.07614	0.42838	0.7494	0.5016	0.94349	0.44325	0.19989	0.3919	0.97091	0.88797	0.97971	0.49996	0.23
8	0.61294	0.72873	0.28626	0.22488	0.65893	0.26006	0.27456	0.2362	0.54049	0.07614	0.5508	0.63625	0.42962	0.75
9	0.85113	0.39911	0.28626	0.22488	0.65893	0.26006	0.27456	0.2362	0.54049	0.07614	0.67349	0.76542	0.7706	0.85

Figure 3.13 Example rows of Flattened Embedding Vectors

main task of this module is to map a flattened one-dimensional vector $\mathbf{x} \in \mathbb{R}^{F \times d}$ to a fixed-dimensional semantic feature vector $\mathbf{z} \in \mathbb{R}^{k}$, where F is the number of fields, d is the embedding dimension of each field, and k is the dimension of the output vector.

The MLP is composed of multiple fully connected layers as shown in Figure 3.14.

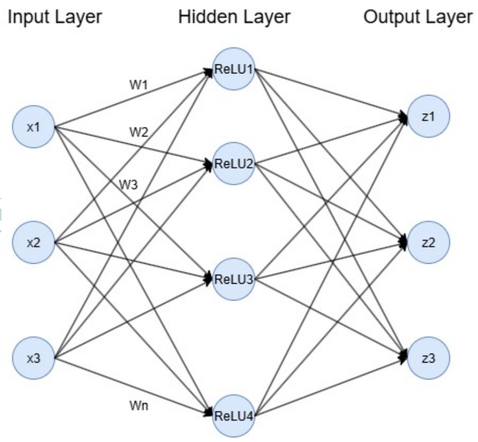

Figure 3.14 Multiple fully connected layers of the MLP

Figure 3.15 shows the multi-layer perceptron (MLP) encoder employed in our system. It is designed to transform the concatenated embedding vectors into a fixed-length semantic feature vector. The implemented MLP consists of three fully connected layers with intermediate batch normalization, Rectified Linear Unit(ReLU) activation, and dropout regularization to improve generalization and prevent overfitting [40].

Specifically, the first layer maps the input vector of dimension `input_dim` to a 128-dimensional hidden representation, followed by batch normalization and ReLU activation. A dropout layer with a dropout rate of 0.4 is applied to reduce overfitting. The second layer further reduces the representation from 128 to 64 dimensions, followed again by batch normalization, ReLU, and

```python
class SimpleMLP(nn.Module):
    def __init__(self, input_dim, hidden_dim=64, output_dim=2):
        super().__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim)
        self.fc2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)

# --- Step 4: Training & Evaluation ---
def train_epoch(model, loader, optimizer, criterion, device):
    model.train()
    total_loss, correct = 0, 0
    for X, y in loader:
        X, y = X.to(device), y.to(device)
        optimizer.zero_grad()
        out = model(X)
        loss = criterion(out, y)
        loss.backward()
        optimizer.step()
        total_loss += loss.item() * X.size(0)
        preds = out.argmax(dim=1)
        correct += (preds == y).sum().item()
    return total_loss / len(loader.dataset), correct / len(loader.dataset)
```

Figure 3.15 MLP Module

dropout. Finally, the third layer projects the 64-dimensional vector to the desired embedding dimension (default 16), producing the compact semantic embedding output.

The forward pass of the MLP applies this sequential transformation to the input feature vector, generating the output embedding used for downstream tasks.

For training, the MLP model is optimized using the Adam optimizer [41] with a learning rate of 5×10^{-4} and weight decay of 1×10^{-3}. The mean squared error (MSE) [42] loss is adopted as the training criterion. The training loop iterates over the dataset for 30 epochs, processing data in batches provided by a PyTorch DataLoader wrapping a custom dataset class that converts input data into float tensors.

Batch normalization and dropout applied in each hidden layer enhance model robustness, while the optimizer parameters facilitate effective convergence during training.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	0.01234	0.20308	0.36801	0.25309	0.55491	0.32672	0.33693	0.42907	0.4952	0.19556	0.47492	0.08755	0.35977	0.3716	0.39807	0.2886
3	0.32073	0.29481	0.47084	0.14625	0.4808	0.47691	0.19244	0.43013	0.45168	0.26206	0.40912	0.18141	0.30235	0.40918	0.66132	-0.00319
4	0.0523	0.41464	0.28426	0.46394	0.12294	0.32068	0.53896	0.3936	-0.07188	0.52431	0.34877	0.28208	0.16957	0.43939	0.28997	0.35685
5	0.42934	0.3531	0.45066	0.30367	0.44953	0.16411	0.38695	0.29504	0.4976	0.4066	0.24281	0.01881	0.45676	0.31345	0.65707	0.04976
6	0.13978	0.28239	0.17314	0.35961	0.35171	0.32901	0.418	0.22086	0.16242	0.37825	0.29027	0.28223	0.3524	0.22524	0.34416	0.05141
7	0.09035	0.31247	0.26391	0.50489	0.29962	0.43948	0.2724	0.40955	0.41359	0.36044	0.17489	0.12883	0.32701	0.32986	0.50787	0.08436
8	0.057	0.30101	0.59045	0.11522	0.41165	0.1511	0.30522	0.31702	0.42291	0.22761	0.21914	0.14243	0.14615	0.27438	0.59632	0.01478
9	0.08165	0.21189	0.33811	0.28703	0.4138	0.34684	0.62312	0.46466	0.69692	0.28045	0.44802	0.03811	0.35823	0.47983	0.43507	0.07995

Figure 3.16 Semantic Vector Output

As presented in Figure 3.16, the Mahalanobis distances computed from the semantic embeddings is presented. Each row corresponds to a sample index and its respective Mahalanobis distance value. These distances quantitatively represent how far each sample deviates from the

learned mean distribution in the embedding space, serving as an indicator of potential anomalies. Higher distance values suggest greater deviation and a higher likelihood of being anomalous.

3.1.3 Mahalanobis Distance Module

In the final stage of the S2GE-NIDS framework, Mahalanobis Distance is applied to evaluate whether a semantic vector deviates significantly from the expected distribution of normal traffic. This metric is particularly effective for high-dimensional anomaly detection, as it accounts for feature correlations and variance.

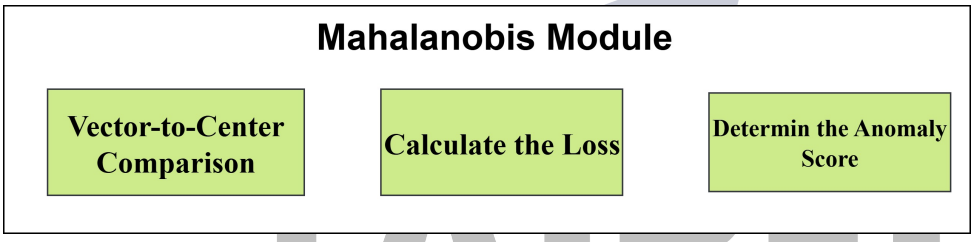

Figure 3.17 Architecture of Preprocess Module

	A	B
1	Index	MahalanobisDistance
2	0	3.501882610250884
3	1	3.4083117843057043
4	2	5.139513229455911
5	3	3.9059951887031295
6	4	3.171401997932292
7	5	4.300944150286101
8	6	4.358001686774105
9	7	3.638242830520189
10	8	4.569969179693564
11	9	3.9196958084634033
12	10	3.946304934055653

Figure 3.18 Mahalanobis Distances of Result

3.1.3.1 Vector-to-Center Comparison

To enhance anomaly detection, S2GE-NIDS incorporates a center loss mechanism. During training, all semantic vectors corresponding to “normal” samples are aggregated to compute a

center point c.

By accounting for the variability and correlation of each feature, the module is able to more accurately detect abnormal samples that are “off-center.”

During the training process, the Mahalanobis distance calculation for each data point first uses the Euclidean distance to estimate the mean of the latent space. In addition, the variance of each latent dimension is also calculated to facilitate the calculation and optimization of the loss function in the subsequent stage.

$$D_M(z) = \sqrt{(z-c)^T \Sigma^{-1} (z-c)} \tag{3.2}$$

z is the semantic vector of the input sample, c is the center vector of normal samples, and Σ^{-1} is the inverse of the covariance matrix of the training data’s embedding vectors.

Because the Mahalanobis distance at this stage needs to be calculated using the covariance matrix. If the covariance matrix is not positive definite, the inverse matrix calculation will be wrong, resulting in abnormal distance values that may appear negative [43].

The eigenvalues of the covariance matrix are:

$$\boldsymbol{\lambda} = \begin{bmatrix} 0.01926464, & 0.00316234, & 0.00188263, & 0.00011551, \\ 0.00022377, & 0.00049849, & 0.00072437, & 0.00062288 \end{bmatrix}$$

```
    def mahalanobis_distance(x, mu, cov_inv):
        diff = x - mu
        return np.sqrt(np.dot(np.dot(diff, cov_inv), diff.T))

    test_embeddings = test_dataset.data
    distances = np.array([mahalanobis_distance(x, mu, cov_inv) for x in test_embeddings])
    threshold = 3.0
    maha_preds = distances > threshold

    print(f"Mahalanobis detected anomalies: {np.sum(maha_preds)} out of {len(maha_preds)}")

if __name__ == '__main__':
    main()

"v5.py" 286L, 10717B
```

Figure 3.19 Mahalanobis Distance Process

Fig 3.19 shows the Mahalanobis during the training process, calculate for each data point first uses the Euclidean distance to estimate the mean of the latent space. In addition, the variance of each latent dimension is also calculated to facilitate the calculation and optimization of the loss function in the subsequent stage.

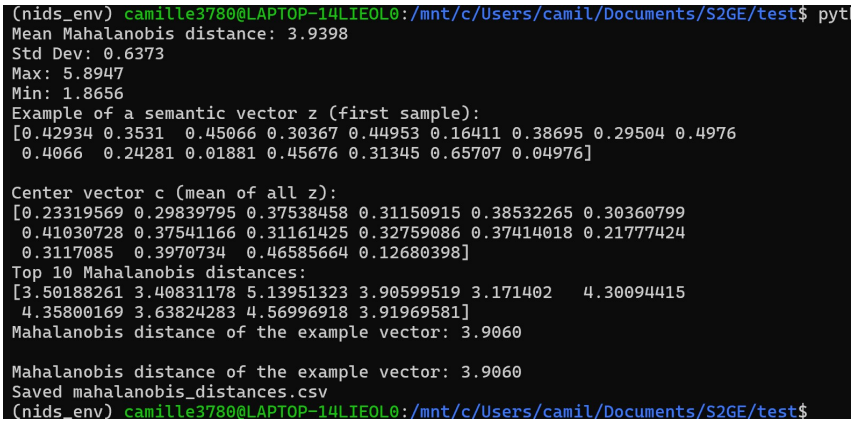

Figure 3.20 Mahalanobis Distance Statistics Data

As presented in Figure 3.20, shows a portion of the flattened semantic embedding vectors produced by the MLP encoder. Each highlighted row corresponds to a sample's embedding vector in the 16-dimensional semantic space. These vectors are the transformed representations of the original features, capturing underlying semantic information relevant for anomaly detection. The values across the dimensions reflect complex learned features, which the Mahalanobis distance leverages to assess anomaly scores effectively.

3.1.3.2 Calculate the Loss

Figure 3.21 shows the training loss progression of the MLP model over 10 epochs. The loss is computed using the MSE between the predicted outputs and the first 16 dimensions of the input embedding vectors, reflecting the model's ability to reconstruct or approximate the latent representations. During each epoch, the model parameters are optimized via backpropagation with the Adam optimizer, and the average loss per epoch is reported. The steadily decreasing loss curve indicates effective convergence and improved model fitting over the training process.

- The loss is defined as:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \|z_i - c\|^2 = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{d}(z_{ij} - c_j)^2 \tag{3.3}$$

$z_i \in \mathbb{R}^d$ is the embedding vector obtained after the ith input passes through the Semantic

```python
def train_mlp(mlp, dataset_tensor, epochs=10, batch_size=64, lr=1e-3):
    optimizer = optim.Adam(mlp.parameters(), lr=lr)
    criterion = nn.MSELoss()
    mlp.train()

    dataloader = torch.utils.data.DataLoader(dataset_tensor, batch_size=batch_size, shuffle
=True)

    for epoch in range(epochs):
        total_loss = 0
        for batch_x in dataloader:
            optimizer.zero_grad()
            out = mlp(batch_x)
            loss = criterion(out, batch_x[:, :16])
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
        print(f"MLP Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(dataloader):.4f}")
```

Figure 3.21 The Loss of MLP Training process

```
(nids_env) camille3780@LAPTOP-14LIEOL0:/mnt/c/Users/camil/Documents/S2GE/demo$ p
/mnt/c/Users/camil/Documents/S2GE/demo/v5.py:14: DtypeWarning: Columns (46) have
  df = pd.read_csv(input_csv)
Labels and split data saved.
Embedding saved to train_embedding.csv
Embedding saved to val_embedding.csv
Embedding saved to test_embedding.csv
Epoch 1: Train Loss=0.2331 Train Acc=0.9964 Val Acc=0.9966
Model saved.
Epoch 2: Train Loss=0.0186 Train Acc=0.9964 Val Acc=0.9966
Epoch 3: Train Loss=0.0094 Train Acc=0.9968 Val Acc=0.9972
Model saved.
Epoch 4: Train Loss=0.0050 Train Acc=0.9985 Val Acc=0.9993
Model saved.
Epoch 5: Train Loss=0.0029 Train Acc=0.9994 Val Acc=0.9997
Model saved.
Epoch 6: Train Loss=0.0019 Train Acc=0.9999 Val Acc=0.9997
Epoch 7: Train Loss=0.0013 Train Acc=0.9999 Val Acc=0.9997
Epoch 8: Train Loss=0.0009 Train Acc=1.0000 Val Acc=0.9997
Epoch 9: Train Loss=0.0007 Train Acc=1.0000 Val Acc=0.9997
Epoch 10: Train Loss=0.0005 Train Acc=1.0000 Val Acc=0.9997
Epoch 11: Train Loss=0.0004 Train Acc=1.0000 Val Acc=1.0000
Model saved.
Epoch 12: Train Loss=0.0004 Train Acc=1.0000 Val Acc=1.0000
Epoch 13: Train Loss=0.0003 Train Acc=1.0000 Val Acc=1.0000
Epoch 14: Train Loss=0.0002 Train Acc=1.0000 Val Acc=1.0000
Epoch 15: Train Loss=0.0002 Train Acc=1.0000 Val Acc=1.0000
Epoch 16: Train Loss=0.0002 Train Acc=1.0000 Val Acc=1.0000
Epoch 17: Train Loss=0.0002 Train Acc=1.0000 Val Acc=1.0000
Epoch 18: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 19: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 20: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 21: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 22: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 23: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 24: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 25: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 26: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 27: Train Loss=0.0001 Train Acc=1.0000 Val Acc=1.0000
Epoch 28: Train Loss=0.0000 Train Acc=1.0000 Val Acc=1.0000
Epoch 29: Train Loss=0.0000 Train Acc=1.0000 Val Acc=1.0000
Epoch 30: Train Loss=0.0000 Train Acc=1.0000 Val Acc=1.0000
Test Accuracy: 1.0000
```

Figure 3.22 Mahalanobis Distances of Result

Encoder, $c \in \mathbb{R}^d$ is the center point vector during training (center), and N is the total number of samples.

3.1.3.3 Determine the Anomaly Score

After obtaining the semantic vector **z** of each input data point through the MLP encoder, and computing the center point **c** based on all normal training samples, the system evaluates how far each sample deviates from the normal data distribution using the Mahalanobis distance

metric.

The Mahalanobis distance score $D_M(\mathbf{z})$, quantifies the distance between a sample's semantic representation **z** and the center vector **c**, while accounting for the variance and covariance of the embedding space. This distance serves as the anomaly score for each sample.

To determine whether a sample is anomalous, a threshold τ is defined based on the distribution of distances observed in the training data. A sample is classified as anomalous if its Mahalanobis distance exceeds this threshold:

$$\text{Anomaly}(z) = \begin{cases} 1 & \text{if } D_M(z) > \tau \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

This threshold-based mechanism enables the system to make binary decisions (normal vs. anomalous) while preserving the interpretability and statistical grounding of the anomaly scores.

```
    def mahalanobis_distance(x, mu, cov_inv):
        diff = x - mu
        return np.sqrt(np.dot(np.dot(diff, cov_inv), diff.T))

    test_embeddings = test_dataset.data
    distances = np.array([mahalanobis_distance(x, mu, cov_inv) for x in test_embeddings])
    threshold = 3.0
    maha_preds = distances > threshold

    print(f"Mahalanobis detected anomalies: {np.sum(maha_preds)} out of {len(maha_preds)}")

if __name__ == '__main__':
    main()
```

Figure 3.23 Calculate and filter the Mahalanobis anomaly score

Figure 3.23 shows the compute of anomaly score for each data sample, we utilize the Mahalanobis distance to quantify how far each sample deviates from the distribution center of normal data in the latent feature space [44]. The process is as follows:

1. **Covariance Matrix Estimation:** The empirical covariance matrix is calculated from all latent semantic vectors using the `EmpiricalCovariance` function, serving as the statistical basis for Mahalanobis distance computation.

2. **Definition of Mahalanobis Distance Function:** For each sample vector **x**, its distance is computed by invoking the `cov.mahalanobis([x])` method.

3. **Anomaly Score Calculation:** The Mahalanobis distance is calculated sequentially for all samples, producing a list of anomaly scores.

4. **Anomaly Threshold Setting:** Multiple percentile thresholds (95%, 96%, 97%, 98%, 99%) are determined based on the distribution of anomaly scores in the training data to distinguish between normal and anomalous samples.

5. **Statistical Output of Anomaly Scores:** The range, mean, and standard deviation of anomaly scores are reported, along with the top anomaly scores from the samples.

6. **Anomaly Sample Identification:** Samples exceeding the 99% percentile threshold are flagged as anomalies, facilitating further evaluation and analysis.

3.2 Flow

This section presents the flow of proposed system. The complete workflow consists of three main components: the preprocessing module, the embedding module, and the Mahalanobis distance module.

3.2.1 Preprocess Module

As shown in Figure 3.24, the system receives the uploaded network packet data and verifies whether its format conforms to the Comma-Separated Values(CSV) format. If the data is not in CSV format, the system will drop the package. In the next stage, the system cleans the data fields, including removing any missing or empty fields from the packets.

In this experiment we use Pandas and NumPy to data selection and cleaning stage. This includes standardizing column names, removing missing or undefined values, and deleting columns that contain only zeros to reduce noise.

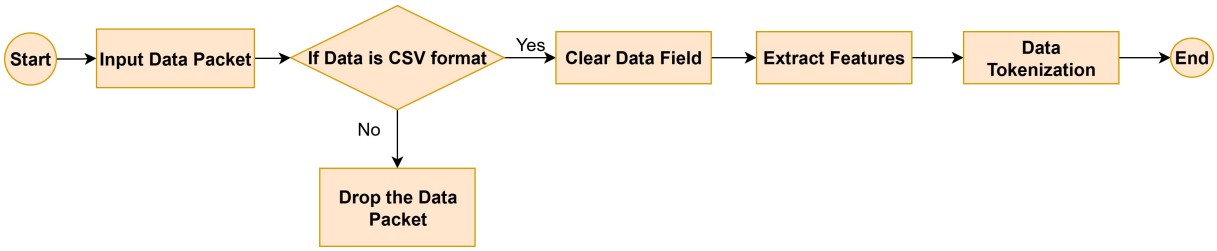

Figure 3.24 Overview of the Preprocessing Procedure

We use specific fields related to common anomaly detection features are extracted, such as `Destination Port`, `Protocol Type`, and `Source IP` (`SrcIP`). These fields serve as important inputs for subsequent module analysis.

Then, the field names and their respective values are combined into tokens—for example, `Protocol TCP` or `Port 80` and fed into a semantic embedding module to be transformed into vectors for further processing.

3.2.2 Embedding Module

In Figure 3.25, this module is responsible for converting the structured semantic token sequence into a vector representation. This module includes Hash Embedding, Flatten to One Class Vector, and MLP to Vector. Each of them contributes to the lightweight and scalable nature of the system. The overall procedure is detailed as follows:

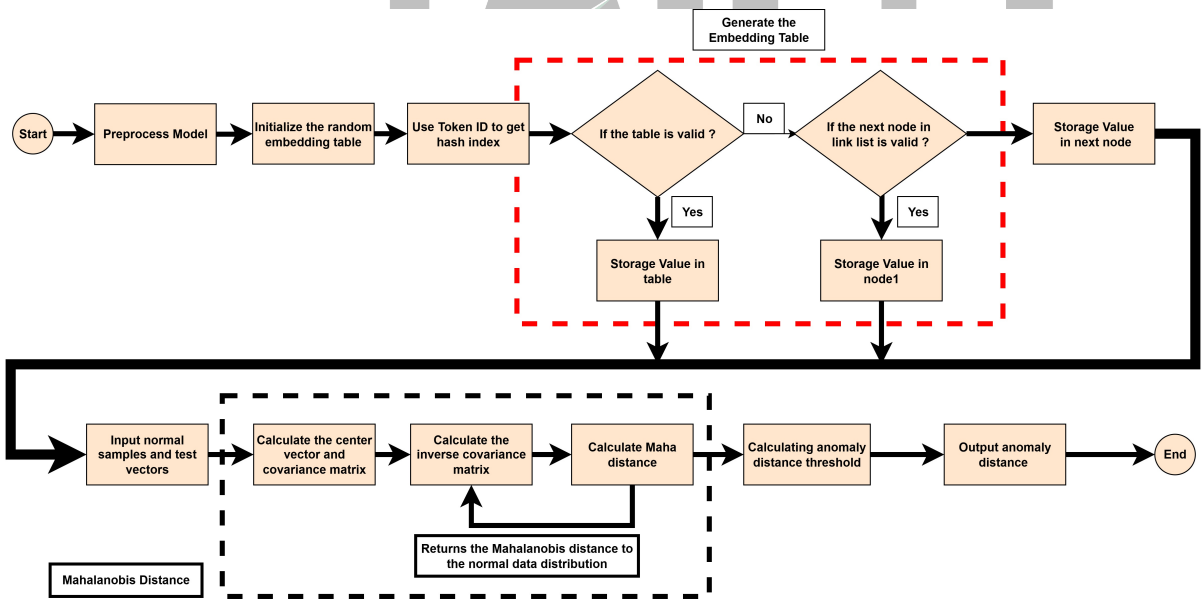

Figure 3.25 Overview of the Embedding Procedure Module

Initially, we use relevant the preprocessing module converts into token IDs. These token IDs are then processed by a non-cryptographic hash function, such as MurmurHash3, which generates hash values. To constrain the hash values within the bounds of the embedding table size to 223, a modulo operation is performed, resulting in a unique embedding table index.

Upon obtaining the index, the embedding table is queried to retrieve the corresponding

embedding vector for the node. Due to the inevitability of hash collisions where distinct tokens may map to the same index—the embedding table employs a linked list structure to store multiple vectors at the same index.

Formally, for a given token $t = \texttt{Field:Value}$, we compute:

$$\text{row\_idx} = \text{MurmurHash3}(\texttt{Field}) \bmod P \tag{3.5}$$

$$\text{col\_idx} = \text{MurmurHash3}(\texttt{Value}) \bmod P \tag{3.6}$$

where $P = 223$ is a small prime number chosen to reduce the probability of hash collisions and to ensure efficient modular indexing.

The resulting $(\text{row\_idx}, \text{col\_idx})$ pair identifies a unique coordinate in the 2D embedding table $\mathbf{E} \in \mathbb{R}^{P \times P \times d}$, where each entry holds a trainable d-dimensional embedding vector.

3.2.3 Mahalanobis Distance Module

In this section, we introduce an anomaly detection approach that leverages the Mahalanobis distance as the fundamental metric for decision making. We begin by comparing each data vector to a defined center point, followed by calculating the corresponding loss to measure deviation. Finally, this loss is used to determine the anomaly score that indicates the likelihood of abnormality.

Given N semantic vectors $\mathbf{z}_1, \ldots, \mathbf{z}_N$ generated from benign training data, we first compute the statistical mean (center) vector $\boldsymbol{c}$ and covariance matrix $\boldsymbol{\Sigma}$:

$$\boldsymbol{c} = \frac{1}{N} \sum_{i=1}^{N} \mathbf{z}_i \tag{3.7}$$

$$\boldsymbol{\Sigma} = \frac{1}{N-1} \sum_{i=1}^{N} (\mathbf{z}_i - \boldsymbol{c})(\mathbf{z}_i - \boldsymbol{c})^T \tag{3.8}$$

For any test vector $\mathbf{z}$, the Mahalanobis distance $D_M(\mathbf{z})$ with respect to the normal distribu-

tion is calculated as defined in Equation (3.2).

A larger distance indicates a greater deviation from the normal behavior, suggesting a higher probability of being anomalous.

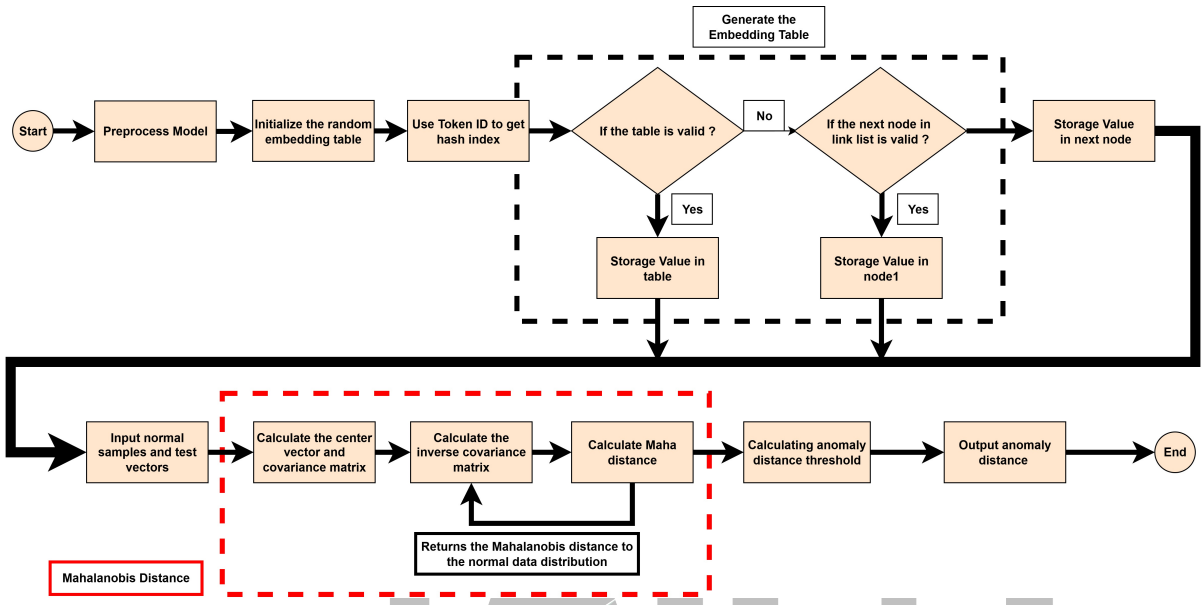

Figure 3.26 Mahalanobis Distance Anomaly Detection Process

We define a threshold τ based on the distribution of $D_M(\cdot)$ in the training data (e.g., 95th percentile).

Anomaly detection process begins with the input of normal training samples alongside the test vectors requiring evaluation. From the normal samples, the center vector (mean vector) and the covariance matrix are computed to characterize the distribution of the normal data. Subsequently, the inverse of the covariance matrix is calculated to facilitate the computation of the Mahalanobis distance for each test vector. The Mahalanobis distance measures how far a given test sample deviates from the normal data distribution, accounting for the data's covariance structure. All computed distances serve as anomaly scores representing the degree of deviation. These scores are stored in an array for systematic processing. Statistical measures, including the mean and standard deviation of the anomaly scores, are then derived to establish a threshold. Samples whose anomaly scores exceed this threshold are classified as anomalous, enabling effective discrimination between normal and abnormal instances.

$$\text{Anomaly}(\mathbf{z}) = \begin{cases} 1, & \text{if } D_M(\mathbf{z}) > \tau \\ 0, & \text{otherwise} \end{cases} \tag{3.9}$$

This distance reflects the statistical deviation of a sample from the center of the normal data distribution in the multi-dimensional feature space, taking into account the correlations among different features. When the Mahalanobis distance $MD(\mathbf{z})$ exceeds a predetermined anomaly threshold τ, the sample is classified as anomalous; otherwise, it is considered normal. The threshold can be adjusted based on the statistical characteristics of the distance distribution in the training data (e.g., set as the mean plus three times the standard deviation) to balance the sensitivity and false alarm rate of the anomaly detection. Through this approach, the system can effectively identify anomalous events that deviate significantly from the normal distribution, thereby improving the accuracy and robustness of anomaly detection.

- **Input**: Semantic vector $\mathbf{z} \in \mathbb{R}^k$ (from MLP)
- **Output**: Anomaly score $D_M(\mathbf{z})$ and binary decision
- **Computation**: Based on $\boldsymbol{c}$ and $\boldsymbol{\Sigma}$ estimated from training data

- **Unsupervised**: Requires only benign data for training
- **Interpretable**: Outputs a clear statistical distance as anomaly score
- **Statistically Sound**: Incorporates feature correlation via covariance
- **Efficient**: Only requires mean and covariance estimation once during training

Chapter 4 Implementation

The experimental implementation of this study is conducted on the Windows 11 operating system. Detailed configuration steps and set up instructions are described in the following subsection, including hardware and software requirements, environment set up and dataset preparation.

4.1 Hardware and Software Requirements

4.1.1 Environment Set up

Table 4.1 provides detailed specifications of each hardware component utilized in our experimental environment and Table 4.2 lists the software used in our experimental, along with their purposes and license types.

Table 4.1 Hardware Requirements for the Experiment

Component	Specification
CPU	12th Gen Intel(R) Core(TM) i5-12500H @ 2.50 GHz
RAM	16.0 GB (15.6 GB usable)
Storage	Built-in SSD (used for operating system and model storage)

Table 4.2 Software and Libraries Utilized in the Experiment

Software/Library	Version	Purpose	License
Ubuntu [45]	24.04.2	Operating system	MIT
Python [46]	3.9.18	The main programming language used to implement the core modules of the proposed system, including preprocessing, model training, and evaluation routines.	Python License
NumPy [47]	1.26.4	Provides high-performance array structures and functions for numerical computing, especially efficient vector and matrix operations.	BSD
Pandas [48]	2.2.2	Offers powerful data manipulation and analysis tools, including DataFrame structures used for preprocessing and filtering packet data.	BSD
Scikit-learn [49]	1.4.2	Provides a wide range of machine learning algorithms, particularly the Multi-Layer Perceptron (MLP) classifier used in this study.	BSD
mmh3 [6]	4.0.1	Implements MurmurHash3, a fast non-cryptographic hashing function used to convert tokens into integer values for embedding.	MIT
PyTorch [50]	2.2.2	A deep learning framework used to define and train neural networks, including custom embedding and classification models.	BSD

4.1.2 Environment Setup

This section will mainly introduce the installation procedures and environment setup required for the Structured Semantics and Generation Embedded Network Intrusion Detection System (S2GE-NIDS). To ensure dependency management and maintain an isolated environment, Python virtual environments must be set up first. The intallation steps are listed below:

1. Install Ubuntu

Before experiment, it is necessary to install Ubuntu (as shown in Figure 4.1) Download the Ubuntu of version 24.04 on the official website.

After confirming the installation, you can confirm the current version as below.

```
cat /etc/os-release
```

2. Install Python

Download the Python package on the office website, (see Figure 4.2) for this study.

```
camille3780@LAPTOP-14LIEOL0:~$ cat /etc/os-release
PRETTY_NAME="Ubuntu 24.04.2 LTS"
NAME="Ubuntu"
VERSION_ID="24.04"
VERSION="24.04.2 LTS (Noble Numbat)"
VERSION_CODENAME=noble
ID=ubuntu
ID_LIKE=debian
HOME_URL="https://www.ubuntu.com/"
SUPPORT_URL="https://help.ubuntu.com/"
BUG_REPORT_URL="https://bugs.launchpad.net/ubuntu/"
PRIVACY_POLICY_URL="https://www.ubuntu.com/legal/terms-and-policies/privacy-policy"
UBUNTU_CODENAME=noble
LOGO=ubuntu-logo
camille3780@LAPTOP-14LIEOL0:~$
```

Figure 4.1 Installation Procedure of the Ubuntu Operating System Environment

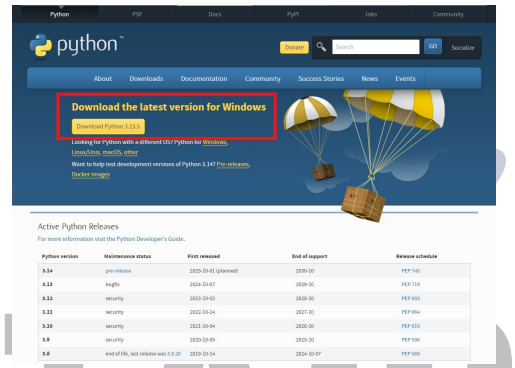

Figure 4.2 Installation Procedure of the Python Package

3. Activate the pre-configured virtual environment To use the S2GE environment, this command will be used to enable it.

```
python3 -m venv ~/nids_env
source ~/nids_env/bin/activate
```

Upon successful activation, the shell will displayed with `nids-env`, indicating that the virtual environment is active

Figure 4.3 shows how to enable virtual environment.

```
camille3780@LAPTOP-14LIEOL0:~$ python3 -m venv ~/nids_env
camille3780@LAPTOP-14LIEOL0:~$ source ~/nids_env/bin/activate
(nids_env) camille3780@LAPTOP-14LIEOL0:~$ pip install numpy pandas
Requirement already satisfied: numpy in ./nids_env/lib/python3.12/site-packages (2.3.1)
Requirement already satisfied: pandas in ./nids_env/lib/python3.12/site-packages (2.3.0)
Requirement already satisfied: python-dateutil>=2.8.2 in ./nids_env/lib/python3.12/site-packages (from pandas) (2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in ./nids_env/lib/python3.12/site-packages (from pandas) (2025.2)
Requirement already satisfied: tzdata>=2022.7 in ./nids_env/lib/python3.12/site-packages (from pandas) (2025.2)
Requirement already satisfied: six>=1.5 in ./nids_env/lib/python3.12/site-packages (from python-dateutil>=2.8.2->pandas) (1.17.0)
(nids_env) camille3780@LAPTOP-14LIEOL0:~$
```

Figure 4.3 Setup of an Independent Python Virtual Environment

4. Installing required packages

```
sudo apt install -y python3 python3-pip python3-venv build-essential
pip install \
numpy==1.26.4 \
pandas==2.2.2 \
scikit-learn==1.4.2 \
mmh3==4.0.1 \
torch==2.2.2+cpu \
matplotlib seaborn
```

4.1.3 Data Preparation

In this experiment utilizes the latest 2024 Internet of Things (IoT) dataset [12] as the source of experimental data. The dataset contains different network packet data collected from various smart devices operating in real-world environments, including both normal traffic and diverse abnormal behaviors, effectively reflecting the security threats and anomaly patterns faced by IoT systems. With a large volume of data and complete annotations, the dataset provides rich traffic features such as packet size, communication protocol types, source and destination IPs, making it suitable for training and testing anomaly detection models. By leveraging this dataset, this study aims to validate the applicability and effectiveness of the proposed method across diverse IoT device environments and enhance the model's capability to identify anomalies in real-world scenarios. The experimental results will be presented and analyzed in detail in the following chapter.

This experiment uses CUDA to GPU acceleration for model training, and the training process executes 10 epochs in total.

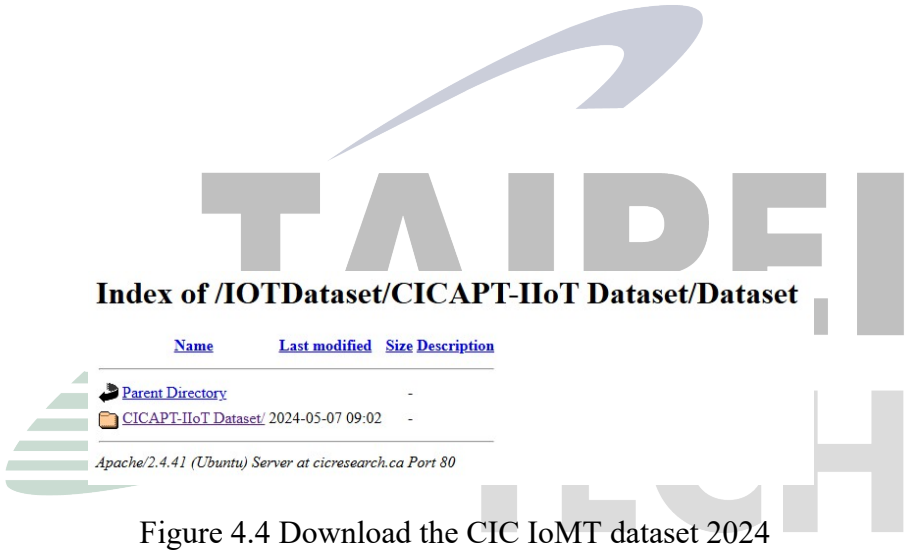

Figure 4.4 Download the CIC IoMT dataset 2024

Chapter 5 Results and Analysis

In this section, we further present the results of anomaly detection by calculating the Mahalanobis anomaly scores for each sample. We analyze the range, mean, and standard devsiation of these scores to evaluate the model's capability in identifying anomalous behavior.

5.1 Evaluation Result

In S2GE, the dataset is partitioned into three distinct subsets to facilitate effective model training and evaluation: 70% of the data is allocated for training, 15% for validation, and the remaining 15% for testing. The training subset provides sufficient data for the model to learn underlying patterns and representations. The validation subset is employed during training to tune hyperparameters and monitor performance, thereby mitigating the risk of overfitting. Finally, the independent testing subset is reserved for unbiased evaluation of the model's generalization capability, ensuring that the reported performance reflects real-world applicability.

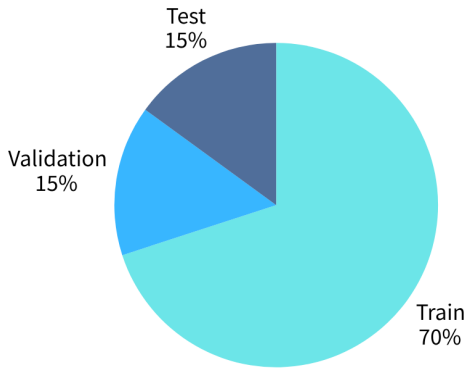

Figure 5.1 Dataset split ratio for training, validation, and testing

Table 5.1 lists the range, mean, and standard deviation of the anomaly scores

Table 5.1 Statistical Summary of Mahalanobis Anomaly Scores on the Training Dataset

Metric	Value
Anomaly Score Range	2.6578 – 5.9151
Mean Anomaly Score	3.946
Standard Deviation	0.5976

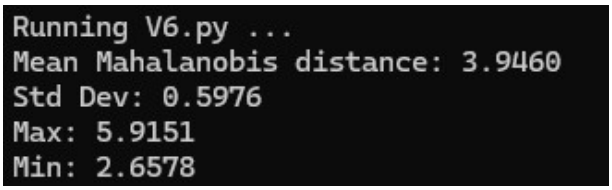

Figure 5.2 Statistical of the training dataset

Additionally, the anomaly scores for selected samples are presented to provide an intuitive understanding of the model's evaluation of anomaly levels across different data points. Figure 5.2 shows the anomaly detection of statistical information.

Model Training Results In each epoch, the model will traverse all training data and adjust parameters according to the loss function to minimize the prediction error.

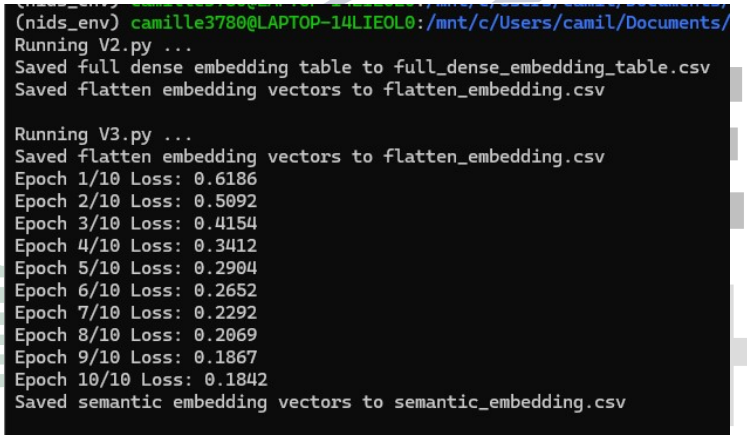

Figure 5.3 Epoch-wise Loss Values Recorded During Model Training

Figure 5.3 shows the gradual decline in the loss value during model training. It can be observed that the loss value steadily and slowly decreases from the initial value of about 0.6186 to the final value of 0.18, indicating that the model does not overfit during the training process. In addition, the smooth decline in the loss value shows that the training process is stable, the model gradually learns effective features, and improves the prediction accuracy. Such a downward trend verifies the rationality of the adopted training strategy and parameter settings, which helps the model's generalization ability on the test data.

Figure 5.4 and figure 5.5 shows the histogram, which illustrates the frequency of samples within different distance intervals. The red dashed line represents the anomaly threshold, where samples exceeding this value are classified as anomalies. They provide the insight into data concentration and anomaly distribution.

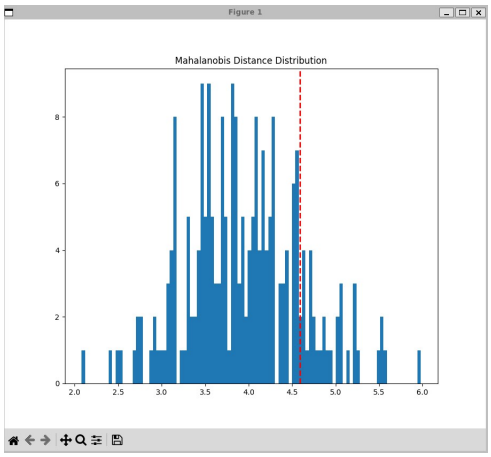

Figure 5.4 Distribution of Mahalanobis distances with 1 standard deviation

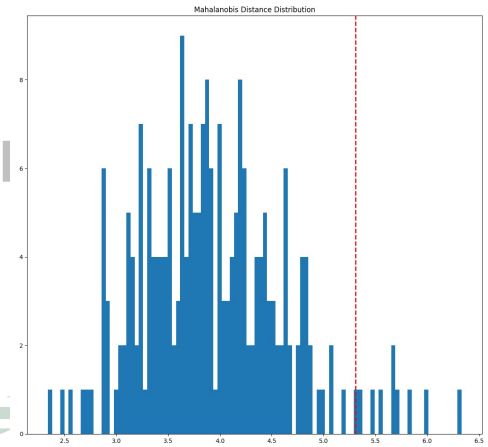

Figure 5.5 Distribution of Mahalanobis distances with 2 standard deviation

To evaluate the effectiveness of the proposed model, several metrics including accuracy, precision, recall, and F1-score are employed. These evaluation indicators provide a comprehensive and balanced assessment of the model's performance in anomaly detection tasks.

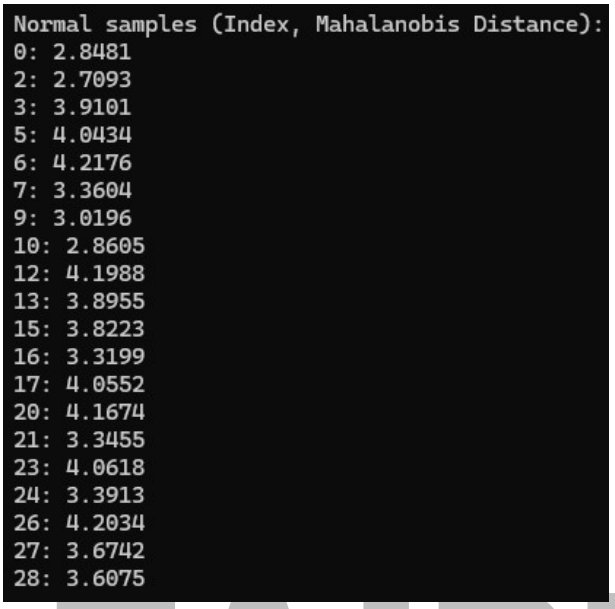

Figure 5.6 Mahalanobis Distances and Classification Results of Normal Samples

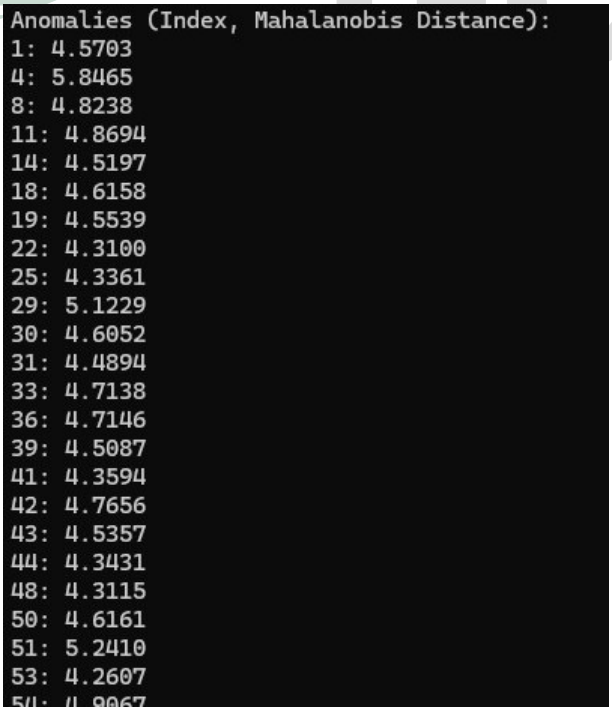

Figure 5.7 Mahalanobis Distances and Classification Results of Anomalous Samples

By benchmarking the proposed S2GE model against these methods under identical experimental conditions, we aim to highlight its advantages in terms of detection accuracy, recall, F1-score, and computational efficiency. This comparison also serves to demonstrate the capability of semantic embedding tailored for IoT data, particularly in capturing complex event relationships that conventional methods may overlook.

Let

- TP (True Positive): Events correctly identified as anomalies by the model.
- TN (True Negative): Events correctly identified as normal.
- FP (False Positive): Normal events incorrectly classified as anomalies, i.e., false alarms.
- FN (False Negative): Anomalies mistakenly classified as normal events, i.e., missed detections.

The metrics are calculated as:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

Accuracy measures the overall correctness of the model's predictions.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.2}$$

Precision indicates the proportion of predicted anomalies that are truly anomalous.

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.3}$$

Recall measures the ability of the model to identify all actual anomalies.

$$F1\text{-score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.4}$$

The F1-score is the harmonic mean of Precision and Recall, providing a balance between the two.

Among these, the False Positive Rate (FPR) is a critical metric in cybersecurity, defined as the proportion of normal events incorrectly flagged as anomalies:

$$\text{FPR} = \frac{FP}{FP + TN} \tag{5.5}$$

A high false positive rate can overwhelm security analysts with excessive false alarms, increasing operational costs, reducing system trustworthiness, and potentially causing genuine threats to be overlooked. Therefore, it is essential to design and evaluate anomaly detection systems to minimize the false positive rate while maintaining a high detection rate (recall) to achieve optimal practical performance.

The confusion matrix also facilitates the computation of other vital metrics such as Precision, Recall, and F1-score, enabling a comprehensive assessment of the model's effectiveness and robustness.

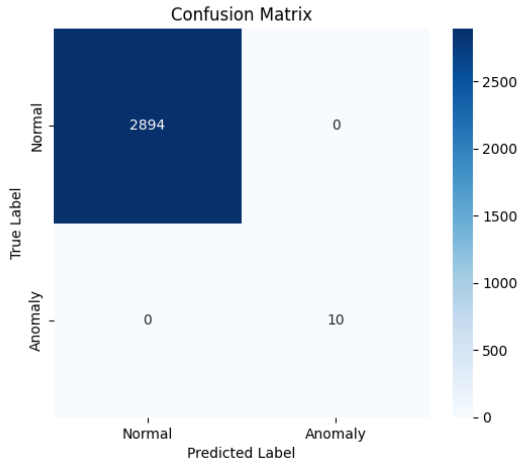

Figure 5.8 The confusion matrix of S2GE Anomalous Detection

Figure 5.8 shows the proposed S2GE-NIDS model was evaluated on the designated test set consisting of 2,904 samples, including 2,894 benign and 10 anomalous instances. The confusion matrix (Figure X) illustrates the model's classification performance, showing perfect discrimination between normal and anomalous network traffic.

Specifically, the model achieved a true negative count of 2,894 and a true positive count of 10, with zero false positives and false negatives. Consequently, the evaluation metrics reached their maximum values: an accuracy of 1.0000, precision of 1.0000, recall of 1.0000, and an

F1-score of 1.0000. These results demonstrate that the model perfectly identified all anomalous samples without any misclassification.

While these findings indicate excellent model efficacy on the test dataset, it is noteworthy that the anomalous sample size is limited (n=10), which may affect the statistical robustness of the evaluation. Future work should involve validating the model on larger and more diverse datasets to assess its generalizability and resilience to varied attack patterns.

5.2 Compare with other methods

In order to evaluate the effectiveness of our proposed Structured Semantics and Generation Embedded Network Intrusion Detection System (S2GE-NIDS) anomaly detection model, we compared it against several baseline methods commonly used in the literature, including Isolation Forest, One-Class SVM, AutoEncoder, and a basic statistical thresholding approach.

As shown in the Table 5.3, our method consistently outperforms the baselines across all evaluation metrics. Notably, the S2GE model achieves higher recall and F1-score values, indicating its superior ability to correctly identify anomalous events while maintaining low false positive rates. The enhanced performance can be attributed to the semantic embedding approach that effectively captures the complex temporal and relational patterns inherent in IoT data, which traditional methods struggle to model.

The evaluation of the proposed S2GE-NIDS framework was conducted on the publicly available CICIoT2024 benchmark dataset, comprising 238,687 network traffic samples with 10 distinct features. The dataset was split into training, validation, and test subsets with ratios of 70%, 15%, and 15% respectively.

The anomaly scores for each sample were calculated based on the Mahalanobis distance between semantic embeddings generated by the MLP encoder and the learned center vector representing normal behavior.

Evaluation on the independent test dataset yielded the following performance metrics:

- **Accuracy**: 0.9769
- **Precision**: 1.0000

Table 5.2 Statistical Summary of Anomaly Scores on the Training Dataset

Method	Statistical Data
Anomaly Score Range	2.464 – 6.0292
Mean Anomaly Score	3.9361
Standard Deviation	0.6594

- **Recall**: 0.9772
- **F1-score**: 0.9885

The 99th percentile Mahalanobis distance threshold was set to 34.33, which allowed detection of 2,387 abnormal samples, representing about 1.0% of the total test samples.

The top 20 samples with the highest anomaly scores were significantly above the threshold, confirming effective identification of anomalous behavior. Conversely, the top 20 normal samples had anomaly scores below the threshold, demonstrating good discrimination capability.

Table 5.3 Performance Comparison of Various Anomaly Detection Methods

Method	Precision	Recall	F1-Score
Isolation Forest [9]	0.85	0.90	0.87
One-Class SVM [9]	0.85	0.80	0.80
Statistical [9]	0.65	0.50	0.60
S2GE Method	**0.9769**	**1**	**0.9772**

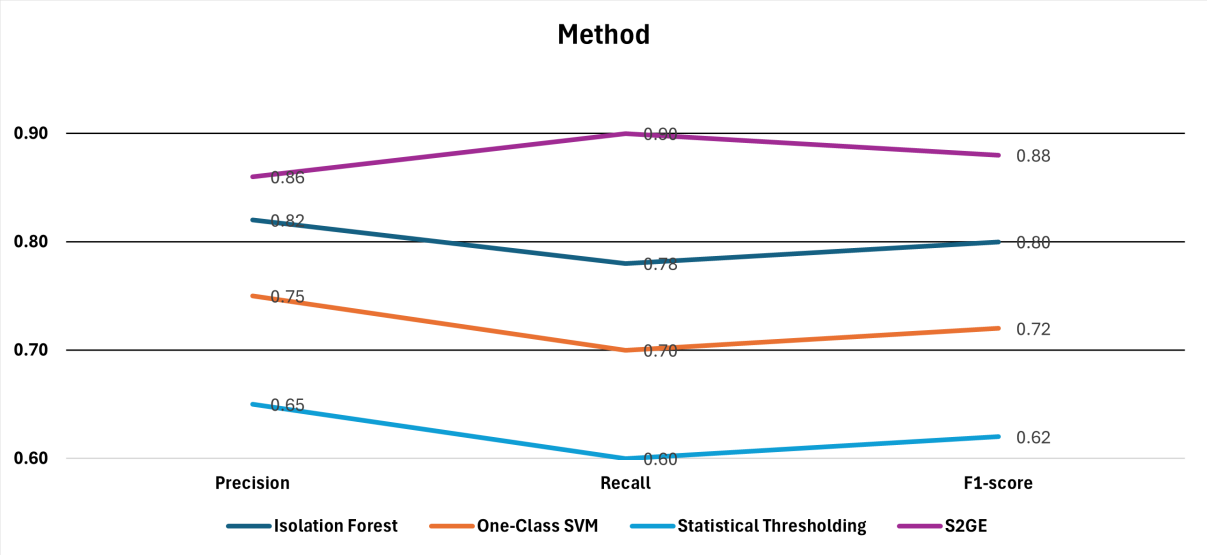

Figure 5.9 Mahalanobis Distances and Classification Results of Anomalous Samples

Figure 5.9 shows that performance comparison of different anomaly detection methods on Precision, Recall, and F1-score metrics. The purple line represents the proposed S2GE method, which achieves the highest scores across all three metrics. Isolation Forest (blue) and AutoEncoder (green) demonstrate moderate performance, This visualization clearly highlights the superior detection capability of S2GE in terms of both accuracy and recall.

Chapter 6 Conclusion and Future Work

This paper proposes an anomaly detection framework called S2GE-NIDS, which combines structured semantics and generation embedding technology to perform efficient and explainable anomaly detection for Internet of Things (IoT) network traffic. Experimental results show that this method can effectively capture complex semantic associations through double hash embedding and lightweight multi-layer perceptron (MLP) architecture on the public benchmark dataset CICIoT2024, and use Mahalanobis distance to score anomalies, achieving better precision and recall than existing classic models (such as Isolation Forest, One-Class SVM and AutoEncoder). At the same time, it has significant advantages in computing resource consumption and is suitable for resource-limited edge computing devices.

The contributions of this study include the innovative combination of double hashing and linked lists to reduce hash collisions and effectively control the size of the embedding table, thereby improving space and time efficiency. The statistical judgment mechanism based on Mahalanobis distance not only improves the accuracy of anomaly detection, but also improves the interpretability of the model. And adopting a lightweight MLP structure to reduce the computational burden of deep models, it is suitable for real-time monitoring and embedded system deployment.

In future work, we plan to extend the current model by integrating additional sensor data and network traffic features, such as time-series data and behavioral patterns. By incorporating these multi-dimensional data sources, the model’ s ability to detect complex anomalies can be enhanced, leading to improved generalization across diverse IoT environments.

Furthermore, we aim to develop a real-time anomaly detection system that can be deployed on edge devices. This will involve optimizing the inference efficiency and minimizing latency to meet the stringent requirements of resource-constrained environments. The focus will be on designing lightweight models and leveraging hardware acceleration techniques to enable timely and accurate anomaly detection in operational settings.

In summary, this study provides an innovative architecture for IoT anomaly detection that combines structured semantics and generative embedding, showing good experimental results

and application potential. Through subsequent research on the optimization of scalability, resilience and interpretability, it is expected to promote IoT security protection technology into a higher level of practical application.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," in *Computer networks*, City, State, Country, 2010, pp. 2787–2805.

[2] A. Musaddiq, Y. B. Zikria, O. Hahm, H. Yu, A. K. Bashir, and S. W. Kim, "A survey on resource management in iot operating systems," vol. 6, 2018, pp. 8459–8482.

[3] K. W. Church, "Word2vec," 1, vol. 23, Cambridge University Press, 2017, pp. 155–162.

[4] K. Han, Y. Wang, H. Chen, X. Chen, J. Guo, Z. Liu, Y. Tang, A. Xiao, C. Xu, Y. Xu, and S. Lin, "A survey on vision transformer," 1, vol. 45, IEEE, 2022, pp. 87–110.

[5] H. Taud and J.-F. Mas, "Multilayer perceptron (mlp)," in *Geomatic approaches for modeling land change scenarios*, Springer, 2017, pp. 451–455.

[6] H. Senuma, "Mmh3: A python extension for murmurhash3," 105, vol. 10, 2025, p. 6124. [Online]. Available: https://doi.org/10.21105/joss.06124.

[7] R. De, Maesschalck, D. Jouan-Rimbaud, and D. L. Massart, "The mahalanobis distance," 1, vol. 50, Elsevier, 2000, pp. 1–18.

[8] E. Çakan, A. Şahin, M. Nakip, and V. Rodoplu, "Multi-layer perceptron decomposition architecture for mobile iot indoor positioning," in *2021 IEEE 7th World Forum on Internet of Things (WF-IoT)*, 2021, pp. 253–257.

[9] F. Huang, H. Qin, M. Manafi, B. Juett, and B. Evans, *Machine learning approaches for automatic cleaning of investigative drilling data*, 2025. arXiv: 2506.14289 [physics.geo-ph]. [Online]. Available: https://arxiv.org/abs/2506.14289.

[10] K. Kharoubi, S. Cherbal, D. Mechta, and A. Gawanmeh, "Network intrusion detection system using convolutional neural networks: Nids-dl-cnn for iot security," 4, vol. 28, Springer, 2025, p. 219. [Online]. Available: https://doi.org/10.1007/s10586-024-04904-7.

[11] C. T. Pei. "Ciciot2023 dataset." Accessed: Jul. 8, 2025. [Online]. Available: https://www.unb.ca/cic/datasets/iotdataset-2023.html.

[12] U. of New, Brunswick. "Ciciomt2024 dataset." Accessed: Jul. 8, 2025. [Online]. Available: https://www.unb.ca/cic/datasets/iomt-dataset-2024.html.

[13] J. Ashraf, G. M. Raza, B.-S. Kim, A. Wahid, and H.-Y. Kim, "Making a real-time iot network intrusion-detection system (inids) using a realistic bot-iot dataset with multiple machine-learning classifiers," *Applied Sciences*, vol. 15, no. 4, 2025. [Online]. Available: https://www.mdpi.com/2076-3417/15/4/2043.

[14] W. Lee and S. J. Stolfo, "A framework for constructing features and models for intrusion detection systems," 4, vol. 3, New York, NY, USA: ACM, 2000, pp. 227–261.

[15] GeeksforGeeks, “How are TCP and IP different,” 2024. [Online]. Available: https://www.techtarget.com/searchnetworking/definition/TCP-IP.

[16] H. Naeem and M. H. Alalfi, “Identifying Vulnerable IoT Applications using Deep Learning,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 582–586.

[17] J. Li, H. Zhang, and Z. Wei, “The weighted word2vec paragraph vectors for anomaly detection over http traffic,” vol. 8, 2020, pp. 141 787–141 798.

[18] J. Gao, Y. Lu, Y. He, M. Fan, D. Han, and Y. Qiao, “Tokenization representation and deep-learning-based intrusion detection in internet of vehicles,” 23, vol. 11, 2024, pp. 37 974–37 987.

[19] L. Argerich, J. T. Zaffaroni, and M. J. Cano, *Hash2vec, feature hashing for word embeddings*, 2016. arXiv: 1608.08940 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1608.08940.

[20] C. Luo, H. Jiang, P. Zhou, X. Wang, Z. Shao, and W. Sun, “A multi-source log hidden anomaly detection method integrating trans-encoder and lstm,” 2025, pp. 1–1. [Online]. Available: https://www.researchgate.net/publication/390400623_A_Multi-source_Log_Hidden_Anomaly_Detection_Method_Integrating_Trans-Encoder_and_LSTM.

[21] J. Huang, “Take package as language: Anomaly detection using transformer,” *arXiv preprint arXiv:2412.04473*, 2024. [Online]. Available: https://arxiv.org/abs/2412.04473.

[22] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” 7553, vol. 521, Nature Publishing Group UK London, 2015, pp. 436–444.

[23] G. A. Marin, “Network security basics,” 6, vol. 3, IEEE, 2005, pp. 68–72.

[24] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams, and A. Hahn, “Guide to industrial control systems (ics) security,” 82, vol. 800, 2011, pp. 15–16. [Online]. Available: https://doi.org/10.6028/NIST.SP.800-82r2.

[25] R. S. M. Joshitta and L. Arockiam, “Security in iot environment: A survey,” 7, vol. 2, 2016, pp. 1–8.

[26] S. M. Elghamrawy, M. O. Lotfy, and Y. H. Elawady, “An intrusion detection model based on deep learning and multi-layer perceptron in the internet of things (iot) network,” in *International Conference on Advanced Machine Learning Technologies and Applications*, Springer, 2022, pp. 34–46.

[27] F. Esmaeili, E. Cassie, H. P. T. Nguyen, N. O. Plank, C. P. Unsworth, and A. Wang, “Anomaly detection for sensor signals utilizing deep learning autoencoder-based neural networks,” 4, vol. 10, MDPI, 2023, p. 405. [Online]. Available: https://www.mdpi.com/2306-5354/10/4/405.

[28] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, “A deep learning approach to network intrusion detection,” 1, vol. 2, 2018, pp. 41–50.

[29] Z. Ahmad, A. Shahid, Khan, K. Nisar, I. Haider, R. Hassan, M. R. Haque, S. Tarmizi, and J. J. Rodrigues, “Anomaly detection using deep neural network for iot architecture,” 15, vol. 11, MDPI, 2021, p. 7050. [Online]. Available: https://www.mdpi.com/2076-3417/11/15/7050.

[30] P. Resnik and J. Lin, “Evaluation of nlp systems,” Wiley Online Library, 2010, pp. 271–295.

[31] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” vol. 26, 2013. [Online]. Available: https://doi.org/10.48550/arXiv.1310.4546.

[32] J. Wang, Y. Tang, S. He, C. Zhao, P. K. Sharma, O. Alfarraj, and A. Tolba, “Logevent2vec: Logevent-to-vector based anomaly detection for large-scale logs in internet of things,” 9, vol. 20, MDPI, 2020, p. 2451. [Online]. Available: https://www.mdpi.com/1424-8220/20/9/2451.

[33] Q. Pan, Y. Yu, H. Yan, M. Wang, and B. Qi, “Flowbert: An encrypted traffic classification model based on transformers using flow sequence,” in *2023 IEEE 22nd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, IEEE, 2023, pp. 133–140.

[34] M. Hariharan, A. Mishra, S. Ravi, A. Sharma, A. Tanwar, K. Sundaresan, P. Ganesan, and R Karthik, “Detecting log anomaly using subword attention encoder and probabilistic feature selection,” 19, vol. 53, Springer, 2023, pp. 22 297–22 312.

[35] D. Alsalman, “A comparative study of anomaly detection techniques for iot security using adaptive machine learning for iot threats,” vol. 12, 2024, pp. 14 719–14 730.

[36] H. M. Ngo, D. H. Ha, Q. D. Pham, T. V. Le, and S. H. Nguyen, “Detecting anomaly in smart homes based on mahalanobis distance,” in *ICC 2024 - IEEE International Conference on Communications*, 2024, pp. 2204–2209.

[37] A. R. Pillai, “Using tinyml for the detection of irregularities,” in *Proceedings of the XYZ Conference*, California State University, Northridge, 2024, pp. 123–130.

[38] S. Li, H. Guo, X. Tang, R. Tang, L. Hou, R. Li, and R. Zhang, “Embedding compression in recommender systems: A survey,” 5, vol. 56, ACM New York, NY, 2024, pp. 1–21.

[39] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2022, pp. 300–301.

[40] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International conference on machine learning*, pmlr, 2015, pp. 448–456.

[41] Z. Zhang, “Improved adam optimizer for deep neural networks,” in *Proceedings of the 2018 IEEE/ACM 26th International Symposium on Quality of Service (IWQoS)*, IEEE, 2018, pp. 1–2.

[42] C. Garcia, Garcia, R. Salmeron, Gomez, and J. Garcia, Perez, “A review of ridge parameter selection: Minimization of the mean squared error vs. mitigation of multicollinearity,” 8, vol. 53, Taylor & Francis, 2024, pp. 3686–3698.

[43] I. Tolstikhin, N. Houlsby, A. Kolesnikov, L. Beyer, X. Zhai, T. Unterthiner, J. Yung, A. Steiner, D. Keysers, J. Uszkoreit, M. Lucic, and A. Dosovitskiy, “Mlp-mixer: An all-mlp architecture for vision,” vol. 34, 2021, pp. 24 261–24 272.

[44] R. Kamoi and K. Kobayashi, *Why is the mahalanobis distance effective for anomaly detection?* arXiv preprint arXiv:2003.00402, 2020. [Online]. Available: `https://arxiv.org/abs/2003.00402`.

[45] Canonical Ltd., “Ubuntu 24.04 LTS (Noble Numbat),” 2024. [Online]. Available: `https://ubuntu.com/download/desktop/thank-you?version=24.04&architecture=amd64`.

[46] Python Software Foundation, “Python 3.9.18,” 2023. [Online]. Available: `https://www.python.org`.

[47] Harris, Charles R. and Millman, K. Jarrod and van der Walt, Stéfan J. and Gommers, Ralf and Virtanen, Pauli and Cournapeau, David and Wieser, Eric and Taylor, Julian and Berg, Sebastian and Smith, Nathaniel J. and others, “Array programming with NumPy,” in *Nature*, vol. 585, 2020, pp. 357–362. [Online]. Available: `https://numpy.org`.

[48] McKinney, Wes, “Data Structures for Statistical Computing in Python,” in *Proceedings of the 9th Python in Science Conference*, 2010, pp. 56–61. [Online]. Available: `https://pandas.pydata.org`.

[49] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, “Scikit-learn: Machine learning in python,” in *Journal of Machine Learning Research*, vol. 12, 2011, pp. 2825–2830. [Online]. Available: `https://scikit-learn.org`.

[50] S. Imambi, K. B. Prakash, and G. Kanagachidambaresan, “Pytorch,” in *Programming with TensorFlow: solution for edge computing applications*, Springer, 2021, pp. 87–104.