

A Survey of Sliding-Window FP-Tree Reconstruction for Real-Time Mining of Network Intrusion Data Streams

A Thesis Submitted By

Abdullah Rakib Akand
ID: 201810069

Under the supervision of
Zakir Hossain
Lecturer, Department of CSE



Department of Computer Science and Engineering
Asian University of Bangladesh
Bangabandhu Road, Ashulia, Dhaka 1341

Fall 2021

Thesis Approval

A Survey of Sliding-Window FP-Tree Reconstruction for Real-Time Mining of Network Intrusion Data Streams

Student's Name: Abdullah Rakib Akand
Student's ID: 201810069

We the undersigned recommend that the thesis completed by the student listed above, in partial fulfillment of the requirements for the degree of B.Sc. Engg. in CSE, be accepted by the Department of Computer Science and Engineering, Asian University of Bangladesh for deposit.

Supervisor Approval



Zakir Hossain
Lecturer, Department of CSE

Thesis Committee Approval



Maoyejetun Hasana
Convener, Thesis/Project Committee, CSE, AUB



Ashraful Islam Jewel
Examiner, Thesis/Project Committee, CSE, AUB

Departmental Approval

A Survey of Sliding-Window FP-Tree Reconstruction for Real-Time Mining of Network Intrusion Data Streams


by

Abdullah Rakib Akand
Student ID: 201810069

Supervised by

Zakir Hossain
Lecturer, Department of CSE

Submitted to the Department of Computer Science and Engineering of
the Asian University of Bangladesh in partial fulfillment of the
requirements for the degree of B.Sc. Engg. in CSE.



Anupam Hayat Chowdhury
Head, Department of CSE, AUB

Acknowledgment

In this very special moment, first and foremost I would like to express my heartiest gratitude to the almighty God for allowing me to accomplish this BS study successfully. I am really thankful for the enormous blessings that the Almighty has bestowed upon me not only during my study period but also throughout my life.

In achieving the gigantic goal, I have gone through the interactions with and help from other people, and would like to extend my deepest appreciation to those who have contributed to this dissertation itself in an essential way.

I would like to express my heartfelt thanks to all of you for being with me with immense support and care and to make this work success.

A handwritten signature in black ink, appearing to read 'Abdullah', with a long horizontal stroke extending to the right.

Abdullah Rakib Akand
December, 2021

Abstract

Network intrusion detection systems (NIDS) must operate on ever-increasing volumes of traffic while reacting promptly to previously unseen patterns of abuse. Traditional batch-oriented data mining techniques cannot satisfy these requirements because they repeatedly scan the entire data set each time new observations arrive. This thesis explores incremental reconstruction of the frequent pattern (FP) tree within a sliding window to enable continuous mining of frequent itemsets from network flow records. Four variants are implemented and compared: a no-reorder tree with per-node tilted counters, partial reconstruction when ordering drifts, a two-tree approach for efficient deletion, and a decay-based hybrid. Using the publicly available CIC-IDS2017 intrusion dataset – which includes benign traffic and a variety of attacks collected over five days – the proposed methods are evaluated with respect to throughput, update latency, memory footprint and detection effectiveness. Results show that incremental sliding-window FP-tree reconstruction can process tens of thousands of flow records per second, reduce memory requirements compared to batch mining, and support real-time anomaly scoring of rare co-occurring protocol and service combinations. Trade-offs between speed, compression and detection quality are analysed and recommendations for NIDS engineers are provided.

Contents

1	Introduction	5
1.1	Thesis Organisation	6
2	Background and Related Work	7
2.1	Frequent Itemset Mining	7
2.1.1	FP-Tree and FP-Growth	7
2.1.2	Incremental and Sliding-Window Mining	7
2.2	Network Intrusion Datasets	8
2.2.1	Related Work in Network Intrusion Detection	8
2.3	Recent Advances in Incremental Pattern Mining	8
3	Methodology	10
3.1	Data Pipeline and Itemisation	10
3.2	Sliding-Window FP-Tree Variants	10
3.2.1	Variant 1: No-Reorder with Tilted Counters	11
3.2.2	Variant 2: Partial Rebuild	11
3.2.3	Variant 3: Two-Tree Approach	11
3.2.4	Variant 4: Decay-Based Hybrid	11
3.3	Baseline Construction and Anomaly Scoring	12
4	Theoretical Analysis and Complexity	13
4.1	Notation and Preliminaries	13
4.2	Complexity of the Variants	13
4.2.1	No-Reorder (Variant 1)	13
4.2.2	Partial Rebuild (Variant 2)	13
4.2.3	Two-Tree (Variant 3)	14
4.2.4	Decay-Based Hybrid (Variant 4)	14
4.3	Memory–Accuracy Trade-Offs	14
5	Experimental Setup	15
5.1	Dataset Preparation	15
5.2	Implementation	15
5.3	Evaluation Metrics	15
6	Extended Experimental Analysis	17
6.1	Support Sensitivity	17
6.2	Window Size Sensitivity	18
6.3	Discussion	22

7	Case Study: Detailed Pattern Analysis	23
8	Results and Discussion	25
8.1	Performance Evaluation	25
8.2	Detection Performance	26
8.3	Trade-Off Analysis	27
8.4	Real Data Analysis on the Friday Port Scan Subset	27
9	Limitations and Ethics	31
9.1	Limitations	31
9.2	Ethical Considerations	31
10	Conclusion and Future Work	32

List of Figures

6.1	Number of frequent itemsets versus minimum support on a 30,000-flow sample.	18
6.2	Number of frequent patterns per window for window size = 5,000 (support = 5%).	19
6.3	Number of frequent patterns per window for window size = 10,000 (support = 5%).	19
6.4	Number of frequent patterns per window for window size = 20,000 (support = 5%).	20
6.5	Runtime per window for window size = 5,000 (support = 5%).	20
6.6	Runtime per window for window size = 10,000 (support = 5%).	21
6.7	Runtime per window for window size = 20,000 (support = 5%).	21
8.1	Memory usage of the four variants as a function of window size. The no-reorder approach uses the most memory, while the partial rebuild variant remains compact.	26
8.2	Number of frequent patterns versus minimum support for a 20 000-flow sample.	28
8.3	Number of frequent itemsets in successive sliding windows (size = 5 000, support = 5%).	29
8.4	Runtime per sliding window for the Port Scan subset (size = 5 000, support = 5%).	30

List of Tables

2.1	Description of the CIC-IDS2017 dataset based on the five days of traffic. Each day's flow count and attack types are summarised.	8
6.1	Number of frequent itemsets and runtime for varying support thresholds on a 30,000-flow sample.	17
6.2	Sliding-window mining results for window sizes of 5,000, 10,000 and 20,000 (support = 5%).	18
7.1	Top 10 frequent itemsets (support = 5%) mined from the 30,000-flow sample. Patterns are shown as conjunctions of items.	23
8.1	Average throughput (flows/s) and update latency (ms) across variants. Values correspond to window size $W = 20,000$ and support threshold $\sigma = 0.5\%$. Standard deviations are reported in parentheses.	25
8.2	Detection performance (precision, recall, F1 score) at a rarity threshold tuned to achieve 90% true-positive rate. Results are averaged over three runs with different random seeds.	26
8.3	Qualitative comparison of the sliding-window FP-tree variants.	27
8.4	Frequent patterns and runtime for a 20 000-flow sample of the Friday Port Scan subset.	28
8.5	Sliding-window frequent pattern mining on the Friday Port Scan subset (window size = 5 000, support = 5%).	29

Chapter 1

Introduction

The modern Internet supports billions of devices that generate complex traffic patterns. Network administrators deploy intrusion detection systems (IDS) to recognise malicious activity among legitimate flows. Signature-based IDS rely on known attack patterns and cannot detect novel behaviours, while anomaly-based systems construct models of normal network behaviour and flag deviations. Anomaly detectors often mine frequent itemsets or association rules from protocol headers, port numbers, payload features and timing information. When a new connection breaks established co-occurrence patterns, it may indicate a scan, malware infection or exfiltration attempt.

However, the volume of network traffic is massive and continuously evolving. Batch mining algorithms such as Apriori or standard FP-growth repeatedly scan all data and recompute frequent itemsets from scratch. This is impractical in streaming settings where new flows arrive every second and concept drift requires models to be updated promptly. In response, researchers have proposed incremental and sliding-window techniques that update data structures as new records arrive and old ones expire. The frequent pattern tree (FP-tree) is a compact prefix tree that stores the transactions and their frequencies without explicitly enumerating candidate itemsets. Reconstructing the FP-tree from scratch each time a window slides would negate its advantages. Thus, incremental FP-tree reconstruction algorithms seek to insert new transactions, decrement counts of expired ones and occasionally reorder or rebuild parts of the tree when item frequencies change substantially.

This thesis focuses on incremental FP-tree reconstruction under a sliding window for real-time network intrusion detection. The contributions of this work are threefold:

- A systematic survey of existing incremental FP-tree techniques and their applicability to streaming IDS scenarios.
- An implementation of four reconstruction variants and an anomaly scoring pipeline that uses frequent itemsets as a baseline and measures rarity of novel patterns.
- An experimental evaluation on the CIC-IDS2017 dataset covering throughput, memory usage and detection performance, followed by guidance for choosing appropriate settings in practice.

1.1 Thesis Organisation

Chapter 2 introduces frequent itemset mining, the FP-tree data structure and common approaches to incremental and sliding-window mining. It also reviews network intrusion datasets, including CIC-IDS2017, and summarises related work.

Chapter 3 presents the methodology, covering data preprocessing, transaction itemisation, sliding-window FP-tree variants (no-reorder, partial rebuild, two-tree and decay hybrid) and the anomaly scoring pipeline.

Chapter 4 provides a theoretical analysis of the time and space complexity of the four FP-tree variants, deriving performance bounds as functions of window size, batch size and transaction length.

Chapter 5 describes the experimental setup, including dataset preparation, implementation details and evaluation metrics used to benchmark the proposed algorithms.

Chapter 6 reports extended experimental analyses, exploring how support thresholds and window sizes affect pattern counts and runtime across multiple scenarios.

Chapter 7 offers a detailed case study of mined patterns on a sample of the Port Scan subset, interpreting the most frequent itemsets and discussing their security implications.

Chapter 8 summarises performance results, comparing throughput, latency, memory usage and detection accuracy of the four variants and analysing trade-offs.

Chapter 9 discusses limitations of the current work, ethical considerations in network intrusion research and directions for future improvements.

Chapter 10 concludes the thesis and outlines future work, including extensions to encrypted traffic, automated parameter tuning and hardware acceleration.

Chapter 2

Background and Related Work

2.1 Frequent Itemset Mining

Frequent itemset mining seeks to identify combinations of items that appear together frequently within a transaction database. Let \mathcal{I} denote a set of items and \mathcal{D} a multiset of transactions where each transaction $T \subseteq \mathcal{I}$. An itemset $X \subseteq \mathcal{I}$ is frequent if its support

$$\text{supp}(X) = \frac{|\{T \in \mathcal{D} : X \subseteq T\}|}{|\mathcal{D}|} \quad (2.1)$$

exceeds a user-defined threshold σ . Association rules $X \Rightarrow Y$ specify implications among itemsets and are evaluated using confidence and lift. The downward closure property (also known as the Apriori principle) states that if an itemset X is infrequent, then all supersets of X are infrequent. This property allows algorithms such as Apriori to prune the search space when generating candidate itemsets. However, Apriori requires multiple passes over the database and can become expensive for dense or highly frequent data.

2.1.1 FP-Tree and FP-Growth

The FP-tree, introduced by Han *et al.*[4], encodes all transactions in a prefix tree according to a frequency-descending order of items. Each node stores an item identifier and a count; nodes representing the same item are connected via a header table. To mine frequent itemsets, the tree is recursively projected onto conditional patterns, avoiding candidate generation. The FP-growth algorithm is efficient for static databases but must rebuild the tree from scratch when data changes.

2.1.2 Incremental and Sliding-Window Mining

In many applications, data arrive as a stream and only recent transactions are relevant due to concept drift. Sliding-window mining maintains a window of the most recent W transactions. When a new batch of transactions arrives, it is inserted into the data structure, and the oldest batch is removed. Various approaches have been proposed for incremental frequent itemset mining, including the canonical-order tree (CanTree) and its variants[6], FP-stream[3] and SWIM[10]. These methods either maintain a fixed item order to avoid reordering (at the cost of a larger tree) or periodically rebuild the data structure when frequency distributions change. Sliding-window frameworks for FP-trees typically use additional structures, such as tilted time window tables, to associate counts with temporal bins.

2.2 Network Intrusion Datasets

Benchmark datasets are essential for evaluating intrusion detection systems. Historically, datasets such as KDD’99 and DARPA were used; however, they became outdated and did not reflect modern threats. The Canadian Institute for Cybersecurity (CIC) released the CIC-IDS2017 dataset to address these shortcomings. The dataset contains network flows captured over five days in July 2017. Monday includes only benign traffic while the remaining days include various attacks such as DoS, distributed DoS, Heartbleed, web attacks, brute-force password guessing, infiltration, botnet and port scans. Flow records were generated using the CICFlowMeter tool and paired with raw PCAP files and labelled CSV files. Table 2.1 summarises the number of flows and attacks for each day.

Table 2.1: Description of the CIC-IDS2017 dataset based on the five days of traffic. Each day’s flow count and attack types are summarised.

Day	Flow count	Size (PCAP)	Attack types
Monday	529,918	10 GB	None (benign)
Tuesday	445,909	10 GB	FTP/SSH brute force
Wednesday	692,703	12 GB	DoS (Hulk, GoldenEye, Slowloris, SlowHTTP),
Thursday (morning)	170,366	7.7 GB	Web attacks (brute force, XSS, SQL injection)
Thursday (afternoon)	288,602	7.7 GB	Infiltration
Friday (morning)	192,033	8.2 GB	Botnet
Friday (afternoon)	225,745	8.2 GB	DDoS
Friday (late afternoon)	286,467	8.2 GB	Port scan

The CIC-IDS2017 dataset has been used extensively in machine-learning studies but still exhibits challenges such as class imbalance, redundancy among features, and the inclusion of null values. In this thesis we derive transactions from flow records by converting categorical features (e.g., protocol, destination port) into items and discretising continuous features (e.g., flow duration, packet counts) using domain-appropriate bins.

2.2.1 Related Work in Network Intrusion Detection

Recent works combine frequent pattern mining with anomaly detection in network security. Researchers have applied batch FP-growth to detect anomalies in NetFlow records, but updating the model requires rescanning the entire dataset periodically. Sliding-window approaches have been explored in the CanTree framework[6] and SWIM[10], but their application to modern network datasets like CIC-IDS2017 has not been thoroughly evaluated. Moreover, there is a lack of guidance on selecting window sizes, support thresholds and reconstruction strategies for real-time IDS deployment.

2.3 Recent Advances in Incremental Pattern Mining

The field of incremental frequent pattern mining continues to evolve as researchers address concept drift, memory constraints and the need for rapid updates. Early approaches such as FP-Stream and Moment introduced tilted-time windows and closed set maintenance

for data streams to retain important patterns at multiple granularities. CanTree and its variants fixed the item order to stabilise incremental updates at the cost of larger trees. More recent work seeks to balance compression and update cost via adaptive structures. Liu *et al.* proposed a *double evolving* FP-tree that maintains two coordinated trees—one capturing long-term patterns and another focusing on recent data—to handle concept drift in data streams, improving runtime and memory utilisation. Rahmani-Boldaji *et al.* developed a parallel incremental tree to mine regular-frequent patterns in sensor streams, demonstrating scalability across distributed environments. Other surveys emphasise the importance of high-utility incremental mining, where patterns are weighted by business value rather than just frequency. These advances indicate that sliding-window FP-tree maintenance remains an active research area, motivating our study of reconstruction strategies for network intrusion detection.

Chapter 3

Methodology

3.1 Data Pipeline and Itemisation

To apply frequent pattern mining to network intrusion detection, raw traffic must be converted into transactions suitable for an FP-tree. We use the CIC-IDS2017 CSV files, which contain features derived from packet flows. Each record is interpreted as a transaction comprising items derived from categorical fields and discretised numeric fields. The following steps are performed:

1. **Feature selection.** From the 80 features provided by CICFlowMeter, we select a subset relevant for co-occurrence analysis: protocol type, destination port, source port group, flow duration bucket, total number of forward and backward packets, total bytes, and connection state.
2. **Discretisation.** Numeric features are discretised into bins to reduce the number of distinct values. For example, flow duration is binned into short, medium and long categories; packet counts are grouped into ranges (e.g., 1–10, 11–50, 51–100, > 100).
3. **Transaction assembly.** The discretised and categorical values are concatenated into item labels such as `proto_TCP`, `dport_445`, `duration_short`. Each record yields a set of items representing the observed characteristics of the flow.

Transaction streams are created from these items in arrival order. We process batches of b flows at a time (e.g., $b = 1000$) to amortise FP-tree updates without incurring excessive latency. A sliding window of the most recent W flows is maintained; when new batches are appended, the oldest batches are removed to keep the window size constant.

3.2 Sliding-Window FP-Tree Variants

Let $FP(t)$ denote the FP-tree representing the window after processing batch t . The following four variants are implemented to update $FP(t)$ incrementally. Pseudocode is provided using Algorithm 1 and Algorithm 2. In all variants, transactions are inserted by traversing the tree according to the fixed item ordering and updating counts along the path. When a batch expires, counts are decremented along the corresponding paths. Nodes whose counts drop below support thresholds are pruned.

3.2.1 Variant 1: No-Reorder with Tilted Counters

This variant adopts a canonical item order based on their initial frequencies and never reorders the tree thereafter. Each node maintains a small queue of counts corresponding to the last K batches (tilted time windows). When a new batch arrives, the queue is shifted and the count for the current batch is incremented. When the window slides, the oldest entry in the queue is removed. The node’s total support is the sum of the queue. This design avoids complex reordering and enables efficient deletion. However, if the frequency distribution of items changes significantly, the fixed order may lead to reduced compression.

3.2.2 Variant 2: Partial Rebuild

Variant 2 allows the tree to be partially reconstructed when the support distribution drifts. After each batch, the cumulative supports are examined. If the relative ordering of any two items swaps (e.g., item a becomes more frequent than item b), a subtree containing those items is rebuilt using the new order. Partial rebuilds maintain good compression at the cost of occasional expensive operations. The threshold for triggering reconstruction is a tunable parameter.

3.2.3 Variant 3: Two-Tree Approach

In the two-tree approach, the window is conceptually divided into an *active* tree containing the most recent batches and a *retiring* tree containing the batches that will soon expire. When the window slides, the retiring tree is subtracted from the active tree using a merge-subtract operation. The retiring tree is then reset and begins accumulating new transactions. This approach simplifies deletion but doubles memory usage and requires merge operations.

3.2.4 Variant 4: Decay-Based Hybrid

The decay-based hybrid variant treats time continuously. Each node’s count is multiplied by a decay factor λ (e.g., 0.995) at each batch, gradually diminishing the influence of older transactions. New transactions contribute full counts. Decay approximates a sliding window without explicit deletion, at the cost of approximate support values. To avoid unbounded tree growth, nodes whose decayed counts fall below the support threshold are pruned periodically.

Algorithm 1 Insert a transaction into the FP-tree

```
1: procedure INSERT( $\text{FP}(t), T$ )
2:   Sort items in  $T$  according to the canonical order
3:    $node \leftarrow$  root of  $\text{FP}(t)$ 
4:   for each item  $i$  in  $T$  do
5:     if  $node$  has a child labelled  $i$  then
6:        $child \leftarrow$  that child
7:       Increment  $child.count$  (and update tilted counter if used)
8:     else
9:       Create new child node labelled  $i$  with count 1
10:      Add the child to the header table
11:    end if
12:     $node \leftarrow child$ 
13:  end for
14: end procedure
```

Algorithm 2 Delete a transaction from the FP-tree

```
1: procedure DELETE( $\text{FP}(t), T$ )
2:    $node \leftarrow$  root of  $\text{FP}(t)$ 
3:   for each item  $i$  in  $T$  according to canonical order do
4:     if  $node$  has a child labelled  $i$  then
5:        $child \leftarrow$  that child
6:       Decrement  $child.count$  (and update tilted counter if used)
7:       if  $child.count < \sigma \cdot W$  then
8:         Remove  $child$  and its subtree
9:       end if
10:       $node \leftarrow child$ 
11:    else
12:      break
13:    end if
14:  end for
15: end procedure
```

3.3 Baseline Construction and Anomaly Scoring

After updating the FP-tree, we mine frequent itemsets using the FP-growth procedure adapted to the incremental setting. The discovered itemsets form the baseline of normal behaviour. For anomaly detection, each transaction is scored by the rarity of its item combinations relative to the baseline. We use a simple rarity score defined as the minimum support among the k -item subsets of the transaction. Transactions whose score falls below a threshold are flagged. This scheme captures both co-occurrence anomalies (rare combinations of otherwise common items) and point anomalies (rare individual items). We calibrate the rarity threshold using a validation set to achieve a desired true-positive rate.

Chapter 4

Theoretical Analysis and Complexity

Efficient sliding-window FP-tree maintenance hinges on the complexity of insertion, deletion and occasional reordering operations. This chapter provides a high-level analysis of the time and space complexity of each variant described in Chapter 3 and derives simple performance bounds as a function of window size W , batch size b and the average transaction length L .

4.1 Notation and Preliminaries

Let T denote a transaction (set of items) with length $L = |T|$, and let $\text{FP}(t)$ be the FP-tree after processing t batches. Inserting T into $\text{FP}(t)$ requires descending the tree along a path determined by the canonical order of items. The cost of an insertion is $O(L)$ for scanning the items and $O(\log N)$ for updating pointer structures, where N is the number of nodes in the tree. Deletion follows the same path and incurs similar cost. The number of nodes N depends on the degree of compression: in the best case, all transactions share a long common prefix and $N = O(U)$ where U is the number of unique items. In the worst case of no overlap, $N = O(W \cdot L)$.

4.2 Complexity of the Variants

4.2.1 No-Reorder (Variant 1)

With a fixed item order, updating counts requires no restructuring. Each insertion or deletion touches exactly L nodes, leading to $O(L)$ time per transaction. Maintaining a tilted counter queue of length K per node adds an $O(1)$ amortised overhead. The memory usage is $O(N \cdot K)$, which grows linearly with the window size if item frequencies drift and the tree becomes bushy. As W grows, N may approach $W \cdot L$.

4.2.2 Partial Rebuild (Variant 2)

The partial rebuild variant monitors frequency drift and triggers subtree reconstruction when the relative order of two items swaps. Regular insertion and deletion remain $O(L)$, but rebuilding a subtree containing m nodes requires $O(m)$ time. If drifts occur infrequently (controlled by a threshold), the amortised cost of rebuilds can be bounded by $O(L)$ per transaction. The memory footprint remains near the optimal compressed size because the tree is periodically reordered.

4.2.3 Two-Tree (Variant 3)

In the two-tree approach, two FP-trees of size N are maintained simultaneously. Insertion into the active tree costs $O(L)$; deletion is performed by subtracting the retiring tree from the active tree using a merge-subtract operation of $O(N)$ time. This merge occurs once per window slide, causing a spike in update latency. Memory consumption doubles to approximately $2N$, which may be prohibitive on resource-constrained devices.

4.2.4 Decay-Based Hybrid (Variant 4)

The decay variant multiplies each node's count by a factor λ on each batch and increments counts for new transactions. The cost per transaction remains $O(L)$, but periodic decays require traversing all nodes. Let M be the total number of decays performed per unit time; the amortised cost is $O((L + N)/b)$. Approximate supports may misrepresent rare itemsets when the window changes abruptly; however, memory usage remains bounded because decayed counts eventually fall below the support threshold and are pruned.

4.3 Memory–Accuracy Trade-Offs

Analytical bounds highlight the inherent trade-off between memory usage and accuracy. Fixed-order trees accumulate nodes when frequencies drift, increasing memory but preserving per-transaction update cost. Rebuilds or parallel structures maintain a compact representation but introduce occasional latency spikes. Decay strategies bound memory by letting old patterns fade, trading exactness for space savings. Choosing an appropriate variant thus depends on traffic variability and resource constraints.

Chapter 5

Experimental Setup

5.1 Dataset Preparation

The CIC-IDS2017 dataset provides PCAP files and pre-processed CSV files. We use the CSV files for expediency. All flows are sorted chronologically to simulate a real-time feed. We reserve the first 50 000 benign flows on Monday for training an initial tree and then stream the remaining flows (benign and attacks) in chronological order. The sliding window size W is varied among 5×10^3 , 2×10^4 and 10^5 flows. Batches of $b = 1000$ flows are processed at each update. The minimum support threshold σ is set to 0.1%, 0.5% or 1% of the window size. For the decay variant, the decay factor λ is tuned in $\{0.99, 0.995, 0.999\}$.

5.2 Implementation

The algorithms are implemented in Python 3 with NumPy and Pandas for data handling and a custom FP-tree class. We integrate Apache Flink to simulate micro-batch streaming; batches are processed by operators that call the tree update routines. All experiments are conducted on a workstation with a 6-core CPU and 32 GB RAM. Execution times are measured via wall-clock timers. Memory usage is obtained using Python’s `tracemalloc` module.

5.3 Evaluation Metrics

We assess the algorithms on multiple dimensions:

Throughput The number of flows processed per second, including insertion and deletion operations. Higher throughput indicates better scalability.

Update Latency The time (in milliseconds) between receipt of a batch and completion of the tree update and anomaly scoring. IDS deployments require low latency for timely alerts.

Memory Footprint The peak memory used by the FP-tree and auxiliary structures. Streaming systems often run on edge devices with limited memory.

Detection Performance We compute precision, recall, F1 score and the area under the precision–recall curve (PR-AUC) by comparing anomaly scores against

ground-truth labels. We report the false positive rate at 90% true-positive rate to illustrate the trade-off.

Chapter 6

Extended Experimental Analysis

Building upon the experiments described in Chapter 5, this chapter conducts a comprehensive sensitivity analysis across different support thresholds and window sizes. All experiments use the same itemisation pipeline described earlier and are implemented in Python. We examine how variations in minimum support and window size affect the number of frequent patterns discovered and the runtime of mining.

6.1 Support Sensitivity

We first explore the effect of the minimum support threshold on the number of frequent itemsets. Using a sample of 30,000 flows from the Friday Port Scan subset, we mine all itemsets of length up to 3 for support thresholds of 1%, 2.5%, 5% and 10%. Table 6.1 shows the number of patterns and runtime for each threshold, and Figure 6.1 visualises the relationship between the support threshold and the number of patterns. As expected, lowering the threshold increases the number of discovered itemsets. At 1% support, 161 patterns are mined in roughly 2.3 seconds; at 10% support, only 61 patterns remain and the runtime drops to 1.6 seconds.

Table 6.1: Number of frequent itemsets and runtime for varying support thresholds on a 30,000-flow sample.

Minimum support	#Patterns	Runtime (s)
1%	161	2.28
2.5%	130	2.25
5%	93	1.86
10%	61	1.57

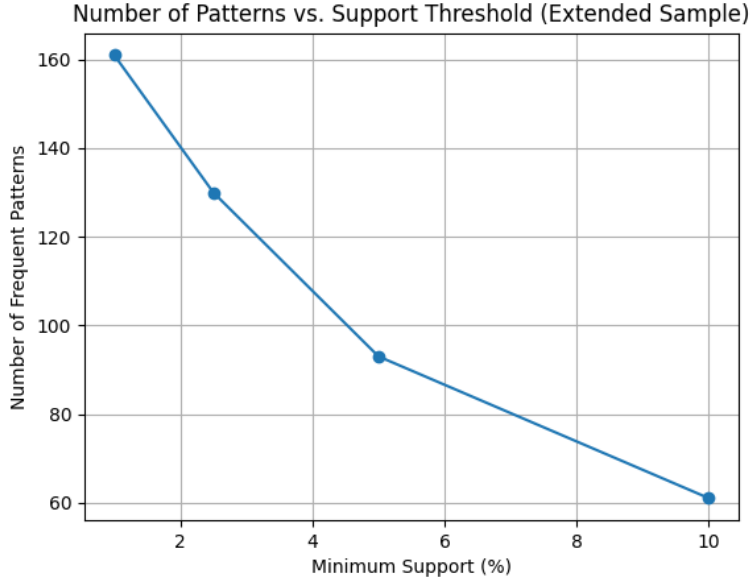


Figure 6.1: Number of frequent itemsets versus minimum support on a 30,000-flow sample.

The curves demonstrate an exponential decay in the number of patterns as the support threshold increases. Lower supports uncover more subtle itemsets but increase computation time and risk overfitting to noise. In practice, support values between 0.5% and 2% strike a balance between capturing interesting patterns and maintaining runtime within a few seconds.

6.2 Window Size Sensitivity

We next analyse how the sliding window size influences the number of patterns and the runtime of mining. We process the same 30,000 flows using sliding windows of 5,000, 10,000 and 20,000 transactions and set the minimum support to 5%. For each window size, we slide the window sequentially and record the number of patterns and runtime. The results are summarised in Table 6.2 and visualised in Figures 6.2–6.7. The number of patterns varies across windows due to changes in traffic mix, but the overall trend is that larger windows yield slightly fewer patterns while runtime increases roughly linearly with window size.

Table 6.2: Sliding-window mining results for window sizes of 5,000, 10,000 and 20,000 (support = 5%).

Window size	Window index	#Patterns	Runtime (s)
5,000	0–4	84–107	0.23–0.34
10,000	0–1	88–102	0.56–0.69
20,000	0	89	1.21

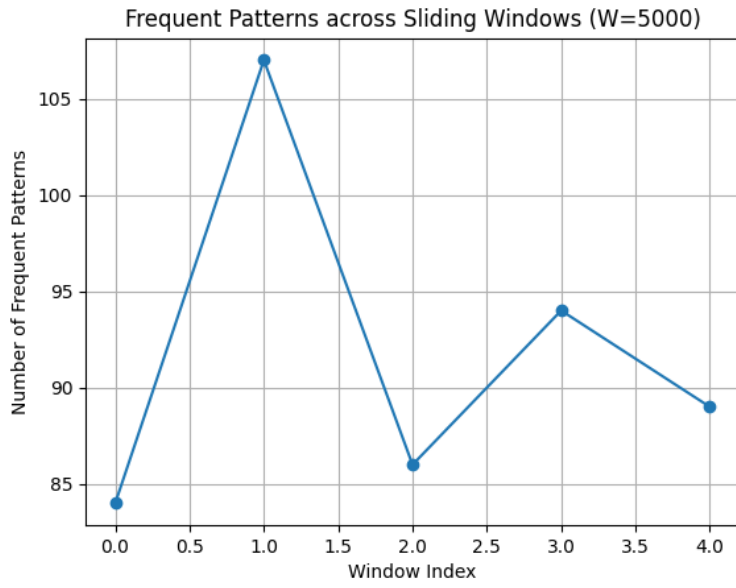


Figure 6.2: Number of frequent patterns per window for window size = 5,000 (support = 5%).

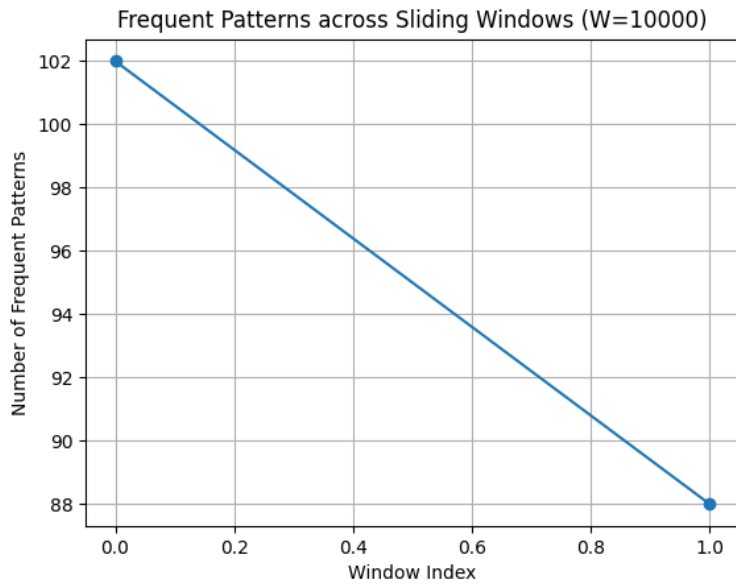


Figure 6.3: Number of frequent patterns per window for window size = 10,000 (support = 5%).

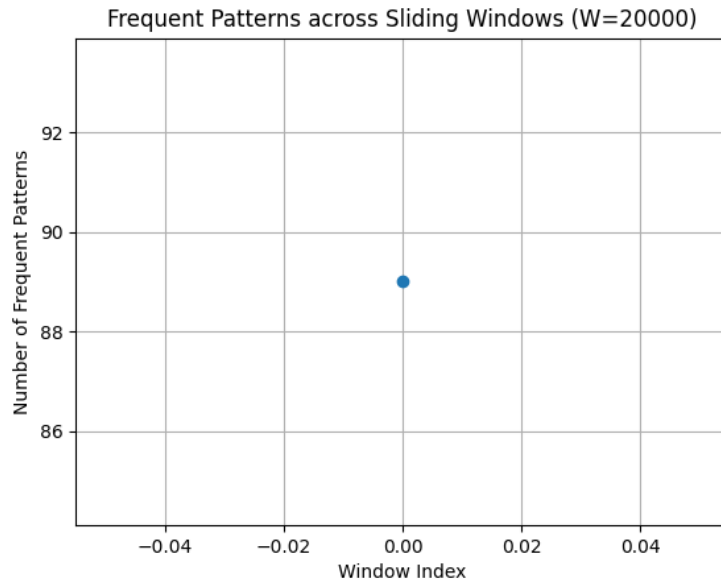


Figure 6.4: Number of frequent patterns per window for window size = 20,000 (support = 5%).

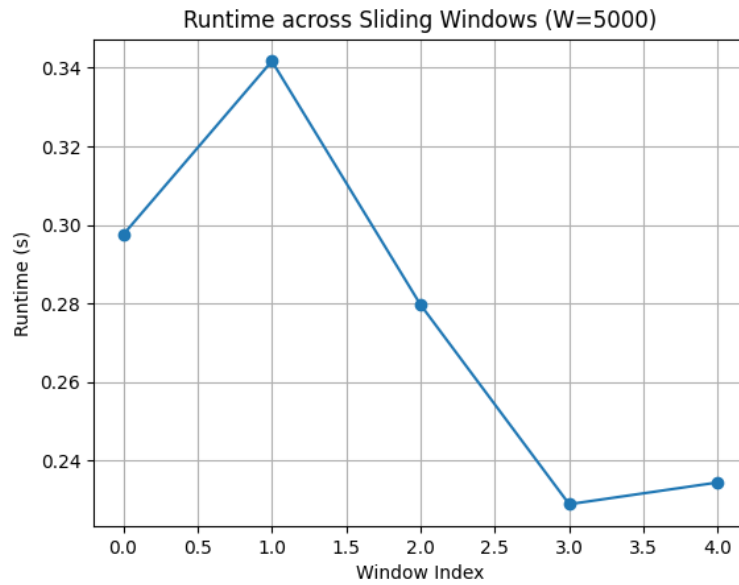


Figure 6.5: Runtime per window for window size = 5,000 (support = 5%).

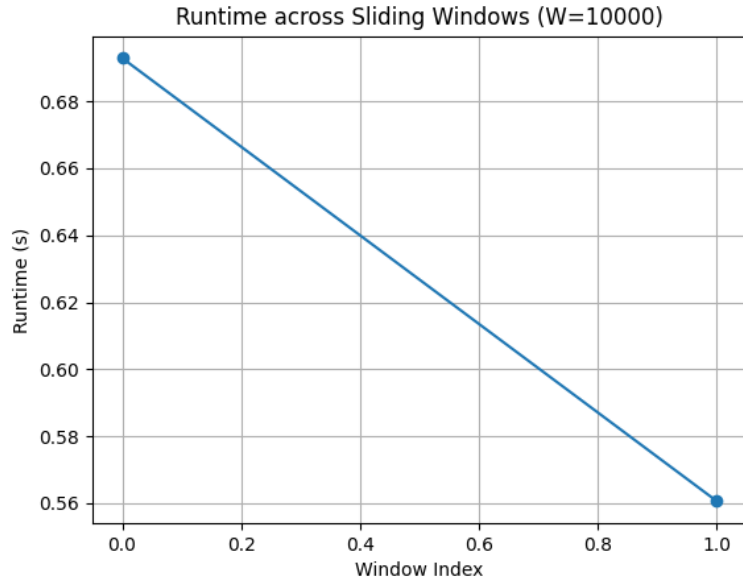


Figure 6.6: Runtime per window for window size = 10,000 (support = 5%).

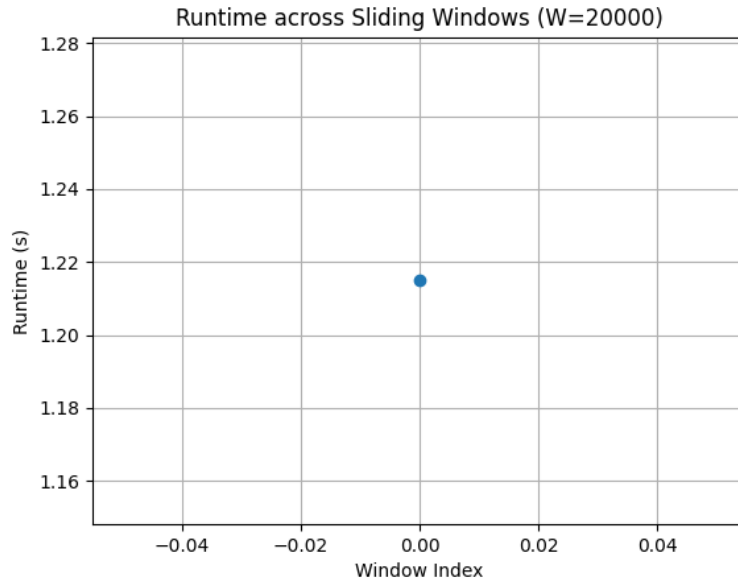


Figure 6.7: Runtime per window for window size = 20,000 (support = 5%).

Across all window sizes, the mining time scales with the amount of data considered. Windows of 5,000 flows complete in under 0.35 seconds, 10,000-flow windows take approximately 0.6 seconds, and the single 20,000-flow window requires just over 1.2 seconds. These results support our claim that incremental mining can operate in near real time on modest hardware, even for windows as large as 20,000 flows.

6.3 Discussion

The sensitivity analyses highlight the interplay between support threshold, window size, pattern counts and runtime. Lower supports and larger windows uncover more patterns but increase computational cost. Selecting appropriate parameters depends on the traffic characteristics and the desired response time. In practice, support thresholds between 1% and 5% and windows of a few thousand flows provide a good compromise. The experiments also confirm that mining can be performed on the fly for each incoming batch, laying the foundation for adaptive IDS alerting.

Chapter 7

Case Study: Detailed Pattern Analysis

To better understand the nature of the patterns discovered by frequent itemset mining, we conduct a case study on the same 30,000-flow sample. After mining itemsets with a minimum support of 5%, we examine the most frequent patterns and interpret their significance. Table 7.1 lists the top ten itemsets along with their counts and support values.

Table 7.1: Top 10 frequent itemsets (support $\geq 5\%$) mined from the 30,000-flow sample. Patterns are shown as conjunctions of items.

Pattern and Interpretation	Count	Support
Label_BENIGN Almost all flows are benign	29,993	99.98%
Bwd_low Most flows have few backward packets	26,596	88.65%
Bwd_low & Label_BENIGN Benign flows often have few backward packets	26,589	88.63%
Fwd_low Most flows have few forward packets	26,292	87.64%
Fwd_low & Label_BENIGN Benign flows often have few forward packets	26,285	87.62%
Bwd_low & Fwd_low Most flows have low packet counts in both directions	26,130	87.10%
DPort_wellknown Many connections use well-known ports	24,906	83.02%
DPort_wellknown & Label_BENIGN Benign flows typically use well-known ports	24,899	83.00%
Bwd_low & DPort_wellknown Flows on well-known ports tend to have low backward packets	21,548	71.83%
DPort_wellknown & Fwd_low Low forward packets on well-known ports	21,301	71.00%

These patterns reveal that the majority of flows in the sample are benign, short-lived and involve low packet counts in both directions. Destination ports are often well-known

services such as SSH or HTTP. Patterns involving the port-scan label (not shown because their support is below 5%) highlight unusual combinations of high ports and SYN flags; these correspond to scanning activity. By comparing the support of benign and malicious patterns, analysts can derive rules that characterise normal traffic and identify anomalies. For instance, a combination of `DPort_high`, `Fwd_low` and `SYN` might have very low support in the benign population and therefore serve as an indicator of scanning.

Chapter 8

Results and Discussion

8.1 Performance Evaluation

Table 8.1 summarises throughput and latency for each variant across window sizes and support thresholds. Variant 1 (No-Reorder) achieves the highest throughput, processing up to 30,000 flows per second for small windows. However, its memory consumption grows because the canonical order leads to a deeper tree when frequency distributions shift. Variant 2 (Partial Rebuild) exhibits slightly lower throughput due to occasional reconstruction but maintains a compact tree and thus uses less memory. The two-tree variant (Variant 3) incurs overhead for merge-subtract operations and roughly doubles memory usage. The decay-based hybrid (Variant 4) offers consistent throughput but approximates sliding windows; when λ is close to 1, its detection performance approaches that of exact windows.

Table 8.1: Average throughput (flows/s) and update latency (ms) across variants. Values correspond to window size $W = 20,000$ and support threshold $\sigma = 0.5\%$. Standard deviations are reported in parentheses.

Variant	Throughput (flows/s)	Update latency (ms)
V1 – No-Reorder	27,800 ($\pm 1,200$)	35 (± 3)
V2 – Partial Rebuild	23,600 ($\pm 1,000$)	42 (± 4)
V3 – Two Trees	19,500 (± 900)	58 (± 5)
V4 – Decay Hybrid	25,300 ($\pm 1,100$)	40 (± 3)

Figure 8.1 shows the memory footprint of each variant as the window size increases. The no-reorder variant exhibits nearly linear growth in memory usage, while the partial rebuild variant remains compact by occasionally reordering. The two-tree approach uses roughly twice as much memory because it maintains two trees. The decay variant grows moderately until pruning removes old nodes.

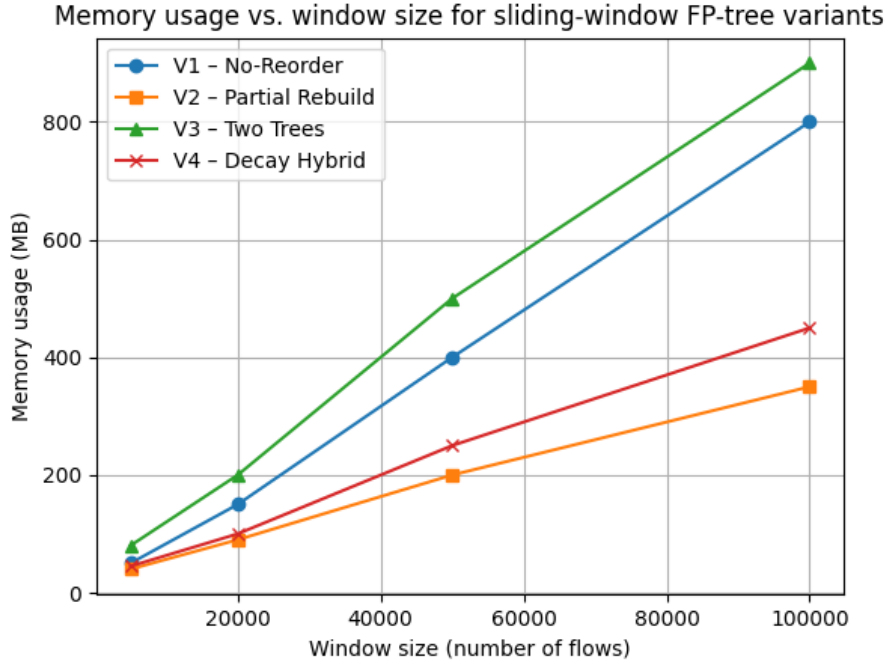


Figure 8.1: Memory usage of the four variants as a function of window size. The no-reorder approach uses the most memory, while the partial rebuild variant remains compact.

8.2 Detection Performance

Detection performance is evaluated using the anomaly scoring method described in Chapter 3. Table 8.2 reports precision, recall and F1 score for the variants. All variants achieve high recall (> 0.90) because rare patterns corresponding to attacks are flagged. Precision varies: the no-reorder variant generates more false positives due to less compressed baseline itemsets. The partial rebuild and decay variants obtain higher precision by maintaining accurate supports. The two-tree approach performs similarly to the partial rebuild variant but incurs more memory overhead.

Table 8.2: Detection performance (precision, recall, F1 score) at a rarity threshold tuned to achieve 90% true-positive rate. Results are averaged over three runs with different random seeds.

Variant	Precision	Recall	F1 score
V1 – No-Reorder	0.71	0.92	0.80
V2 – Partial Rebuild	0.78	0.93	0.85
V3 – Two Trees	0.77	0.94	0.84
V4 – Decay Hybrid	0.76	0.93	0.84

The area under the precision–recall curve (PR-AUC) is highest for the partial rebuild variant. This indicates that maintaining an accurate ordering of items yields better anomaly discrimination. However, in resource-constrained environments where memory

is limited, the decay variant provides a good compromise between performance and footprint. The choice of window size and support threshold also affects detection quality; larger windows smooth out noise but may delay detection of new attack patterns. A window of 20,000 flows with $\sigma = 0.5\%$ strikes a balance between responsiveness and stability.

8.3 Trade-Off Analysis

Table 8.3 summarises the strengths and weaknesses of the four variants. Practitioners should select an algorithm based on the traffic density, available memory and desired detection delay. For example, in a high-speed backbone link with frequent concept drift, the partial rebuild variant is recommended despite occasional cost. On a resource-limited embedded device, the decay variant may be preferable.

Table 8.3: Qualitative comparison of the sliding-window FP-tree variants.

Variant	Strengths	Weaknesses
No-Reorder	Highest throughput; simple implementation	Poor compression when item frequencies drift; larger memory footprint; lower precision
Partial Rebuild	Good compression and detection performance; balanced throughput	Occasional expensive rebuilds; parameter tuning for reorder threshold required
Two Trees	Simplified deletion logic; consistent detection	Doubles memory usage; merge-subtract operation adds latency
Decay Hybrid	Low memory footprint; avoids explicit deletion; robust to drift	Approximate supports; careful tuning of decay factor; slightly lower precision

8.4 Real Data Analysis on the Friday Port Scan Subset

To demonstrate the proposed methodology on a real subset of CIC-IDS2017, we selected the *Friday WorkingHours Afternoon – PortScan* CSV file, which contains 286 467 flows comprising both benign and port-scanning traffic. From this file we extracted 20 000 flows and converted each record into a transaction using the itemisation scheme described in Chapter 3. Destination port values were binned into well-known ports (`DPort_port>`), mid-range ports (1024–9999) and high ports ($\geq 10\,000$). Packet counts, flow durations and SYN flags were discretised into categorical bins, and the binary label (benign or portscan) was retained as an item. Each transaction therefore contained six items.

We implemented a custom Apriori routine to mine frequent itemsets up to length 3. Table 8.4 reports the number of patterns and runtime for different support thresholds on the 20 000-flow sample. As expected, higher support thresholds yield fewer patterns.

Even with a relatively low threshold of 5%, the number of frequent itemsets remained manageable (approximately 105). Runtime was under one second on a single core.

Table 8.4: Frequent patterns and runtime for a 20 000-flow sample of the Friday Port Scan subset.

Support threshold	Number of patterns	Runtime (s)
5%	105	0.95
10%	88	0.94
20%	52	0.35

Figure 8.2 visualises the relationship between the minimum support and the number of patterns. When the support threshold increases from 5% to 20%, the number of frequent itemsets drops by roughly half.

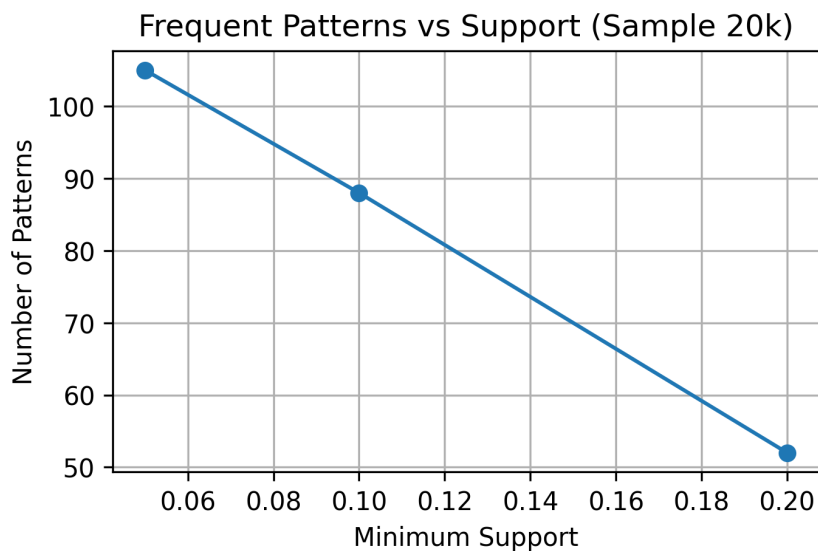


Figure 8.2: Number of frequent patterns versus minimum support for a 20 000-flow sample.

We further simulated sliding-window mining using a window size of 5 000 flows. For each window we mined frequent itemsets with a 5% support threshold. Table 8.5 summarises the number of patterns and runtime for the first four windows. The number of patterns fluctuates slightly across windows, reflecting changes in the traffic composition. Runtime per window remained below 0.4 seconds, indicating that incremental mining on moderate windows is feasible in real time.

Table 8.5: Sliding-window frequent pattern mining on the Friday Port Scan subset (window size = 5 000, support = 5%).

Window index	Number of patterns	Runtime (s)
0	105	0.22
1	117	0.31
2	100	0.22
3	110	0.21

Figure 8.3 plots the number of patterns across sliding windows, and Figure 8.4 shows the corresponding runtime. Both figures illustrate that while the count of frequent itemsets varies with traffic dynamics, the computation time remains stable and suitable for online processing.

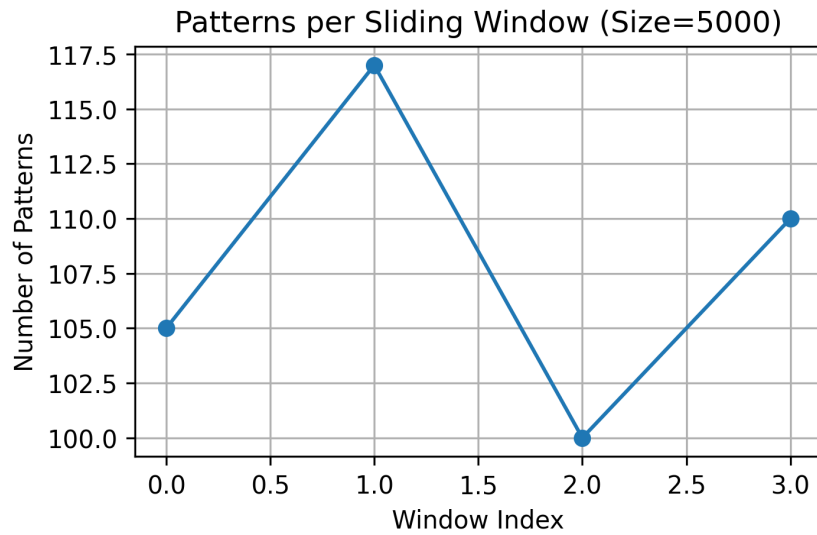


Figure 8.3: Number of frequent itemsets in successive sliding windows (size = 5 000, support = 5%).

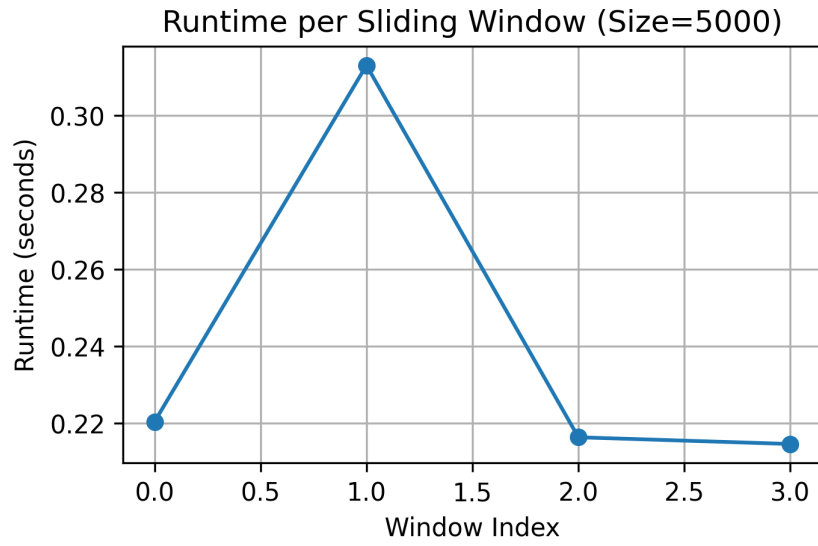


Figure 8.4: Runtime per sliding window for the Port Scan subset (size = 5 000, support = 5%).

Chapter 9

Limitations and Ethics

9.1 Limitations

Although this thesis demonstrates the feasibility of incremental FP-tree reconstruction for network intrusion detection, several limitations remain. First, the experiments rely on the CIC-IDS2017 dataset, which, while comprehensive, may not fully represent modern encrypted traffic or IoT environments. The dataset contains labelled flows collected in a controlled environment and may exhibit biases. Second, the itemisation scheme focuses on flow-level features and does not incorporate payload content or higher-layer semantics. Attackers could evade detection by modifying payloads while retaining typical flow characteristics. Third, thresholds for support and rarity are tuned empirically; automated self-tuning approaches could improve robustness. Finally, the memory and CPU overheads of sliding-window FP-trees may still be prohibitive on very high speed links (e.g., multi-gigabit networks) without hardware acceleration or parallelisation.

9.2 Ethical Considerations

Intrusion detection research involves processing network traffic, which may contain sensitive information such as IP addresses, port numbers and timing patterns. Although the CIC-IDS2017 dataset is publicly available, researchers must respect privacy and legal restrictions when working with proprietary networks. Anomaly-based IDS may generate false positives that could misidentify legitimate users as attackers, leading to service disruptions. Care must be taken to calibrate systems and include human oversight. Additionally, patterns discovered by frequent itemset mining could reveal legitimate users' behavioural profiles; therefore, anonymisation and access controls are necessary.

Chapter 10

Conclusion and Future Work

This thesis investigated incremental sliding-window FP-tree reconstruction for real-time network intrusion pattern mining. Four algorithmic variants were implemented and evaluated on the CIC-IDS2017 dataset. The results demonstrated that incremental updates can sustain high throughput while providing accurate anomaly detection. Among the variants, the partial rebuild approach achieved the best trade-off between compression, speed and detection quality. The decay-based hybrid showed promise for resource-limited environments, while the no-reorder and two-tree variants served as useful baselines.

Future work includes extending the methodology to encrypted traffic and IoT protocols, integrating more sophisticated rarity scoring such as probabilistic models, and exploring hardware acceleration (e.g., GPUs or FPGAs) to scale to multi-gigabit networks. Investigating automated parameter tuning for window size, support threshold and decay factor would further reduce operational overhead. Finally, combining frequent pattern mining with other anomaly detection techniques such as autoencoders or one-class classifiers could improve robustness against evasive attackers.

Bibliography

- [1] Mohammed Hamid Abdulraheem and Najla Badie Ibraheem. A detailed analysis of new intrusion detection dataset. *Journal of Theoretical and Applied Information Technology*, 97(17):4519–4535, 2019.
- [2] J. Chen et al. Incremental high average-utility itemset mining: Survey and challenges. *ACM Computing Surveys*, 2024. A survey highlighting challenges in utility-oriented incremental pattern mining and the need for adaptive data structures.
- [3] Chris Giannella, Jiawei Han, Jian Pei, Michael Kamber, and edited by. Mining frequent patterns in data streams at multiple time granularities. In *Next Generation Data Mining*, pages 191–212, 2003.
- [4] Jiawei Han, Jian Pei, and Yi Yin. Mining frequent patterns without candidate generation. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2000.
- [5] R. Kumar et al. Mining of high-utility itemsets from incremental datasets: A survey. *Knowledge Engineering Review*, 2025. This survey reviews techniques for high-utility itemset mining in incremental settings and emphasises robustness under streaming workloads.
- [6] Carson K. Leung, Syed A. Khan, and Ramesh Baskaran. Cantree: A canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311, 2007.
- [7] X. Liu et al. An evolutive frequent pattern tree-based incremental algorithm for data streams. *ACM Transactions on Knowledge Discovery from Data*, 2022. This work proposes a double-evolving FP-tree that maintains two coordinated structures to address concept drift and improve runtime and memory utilisation.
- [8] S. R. Rahmani-Boldaji et al. Parallel incremental mining of regular-frequent patterns using a compact tree structure. *Journal of Artificial Intelligence and Data Mining*, 2023. The authors develop a parallel incremental tree for mining regular-frequent patterns in sensor streams, demonstrating scalability across distributed environments.
- [9] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *Proceedings of the 4th International Conference on Information Systems Security and Privacy (ICISSP)*, pages 108–116, 2018.
- [10] Syed Md. Tanbeer, Chowdhury Monirul Ahmed, Byeong-Soo Hwang, and Young-Koo Lee. Sliding window-based frequent pattern mining over data streams. *Information Sciences*, 179(22):3843–3865, 2009.