Improving Large Graph Visualization Using a Paging Mechanism

by

Fatemeh Jafarrangchi

A Report Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF ENGINEERING

in the Department of Electrical and Computer Engineering

© Fatemeh Jafarrangchi, 2023

University of Victoria

All rights reserved. This report may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author

Supervisory committee

Improving Large Graph Visualization Using a Paging Mechanism

by

Fatemeh Jafarrangchi

University of Victoria 2023

Supervisory Committee

Dr. Issa Traore, Department of Electrical and Computer Engineering
Supervisor

Dr. Isaac Woungang, Department of Electrical and Computer Engineering
Co-Supervisor

List of Contents

Supervisory committee ii

List of Contents iii

Glossary vi

Abstract vii

Acknowledgments viii

Dedication ix

Chapter 1 : Introduction 1

1.1 Context 1

1.2 Objective and Approach 2

1.3 Report Outline 5

Chapter 2 : Background and Approach 6

2.1 The Activity and Event Network Model 6

2.2 AEN Graph Visualization 8

2.3 Current Issues and Proposed Solution 9

Chapter 3 : Proposed Paging Graph Model 11

3.1 Graph Paging 11

3.2 System Architecture 12

3.3 Graph Paging Visualization 13

3.4 Graph Saving Algorithm 17

3.5 Saving Multiple Files 23

3.6 Graph Reading Algorithm 24

3.7 Reading existing graph paging 28

Chapter 4 : Implementation and Performance Evaluation 29

4.1 Dataset 29

4.2 Evaluation Environment and Procedures 30

4.3 Evaluation Results 31

Chapter 5 : Conclusion 39

References 41

List of Tables

Table 4.1: Comparing RAM and CPU usage evaluation in different file sizes. 32

List of Figures

Figure 1.1 Percentage of surveyed companies compromised by at least one successful attack in 2021 by countries [2] 2

Figure 2.1: A visualized graph sample in NetFlow 8

Figure 3.1 Graph Paging reading/saving process 13

Figure 3.2: Paging Graph on page 5 14

Figure 3.3: Slider dynamic range set function 15

Figure 3.4: Paging graph on page 175 16

Figure 3.5: Saving graph paging flow chart 20

Figure 3.6: Graph Paging model 21

Figure 3.7: Actual saved data as a graph paging in a JSON file. 22

Figure 3.8: Sample JSON of actual saved nodes and edges 23

Figure 3.9: reading graph paging flow chart 27

Figure 4.1: Comparing response time for the first five files of Tuesday-WorkingHours.pcap file in three different reading formats 33

Figure 4.2: Comparing CPU usage for the first five files of Tuesday-WorkingHours.pcap file in three different reading formats 35

Figure 4.3: Comparing RAM usage for the first five files of Tuesday-WorkingHours.pcap file in three different reading formats 36

Figure 4.4: CPU and RAM usage when using the timeline. 38

Glossary

AEN: Activity and Event network graph.

NetFlow: Network Flow.

CPU: Central Processing Unit.

RAM: Random Access Memory.

IP: Internet Protocol.

ISOT: Information Security and Object Technology Research lab at the University of Victoria.

PCAP: Packet Capture.

Syslog: System Logging Protocol.

DDoS: Distributed Denial-of-Service.

Dos: Denial of Service.

SQL: Structured Query Language.

JSON: JavaScript Object Notation lightweight data transfer format.

XSS: Cross-Site Scripting

Abstract

The activity and event network (AEN) model captures the network activities and events using a large random dynamic graph that is continuously maintained and updated as new information and data arrive. The AEN engine leverages extensive graph database technology in creating, maintaining, and visualizing the produced graph. Because the graph can become very large (e.g., have millions of nodes) over time, a visual analysis by a security analyst can be unwieldy, overwhelming, and thus counterproductive. This thesis presents an extension of the AEN graph engine visualization module, which consists on developing a timeline feature that improves the visualization process by allowing the analyst to access and work on segments or portions of the graph as needed. A graph paging mechanism was developed to implement the timeline feature, where a graph is structured into multiple pages that enable navigating back and forth and other related functionality. To reduce memory/storage usage, the proposed graph paging mechanism supports consolidating fine-grain changes into coarser-grain ones without losing the timeline integrity and altering the order in which the changes occurred. An experimental evaluation using the CIC 2017 IDS evaluation dataset yielded improved results in visualizing and handling large graphs while achieving low performance overhead in terms of response time, CPU time, and memory utilization.

Acknowledgments

I want to thank my supervisor, **Dr. Issa Traore**, for his constant guidance, support, and motivation throughout my project and throughout my program.

I would like to thank my co-supervisor, **Dr. Isaac Woungang**, for all his valuable advice and feedback.

I would also like to thank **Dr. Paulo Quinan** for his continuous support and assistance throughout my project.

Dedication

I dedicate this report to my husband, Ali, for his unwavering support and motivation during my studies. I also dedicate it to my beloved family and cherished friends, who have been a constant source of inspiration throughout this journey.

Chapter 1 : Introduction

1.1 Context

In the last decades, there has been a significant increase in hacking incidents against organizations and individuals. As time progresses and new technologies arise, cyberattack techniques become more sophisticated. The solutions to the latest hacks or the discovery of new hacking methods might take a long time to be revealed. On the other hand, in recent years, the number of applications released into the market and installed on devices has risen. These applications require minimal credentials and configurations, which increase the network's attack surface and the probability of successful cyber-attacks [1]. According to a study by Comparitech Corporation in 2021, many companies worldwide fell victims to cyber-attacks. More than 85% of the surveyed companies in Canada were affected by some breach [2]. Figure 1.1 shows the percentage of companies affected by cyber-attacks during 2021, categorized by countries based on the study by Comparitech Corporation.

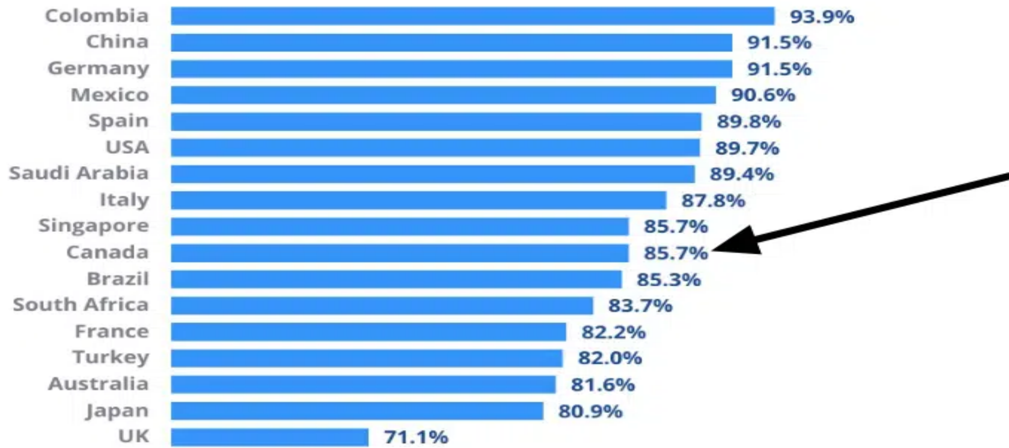

Figure 1.1 Percentage of surveyed companies compromised by at least one successful attack in 2021 by countries [2].

1.2 Objective and Approach

Fortunately, in response to the current threat landscape, a broad range of technologies are available or are being developed to help organizations strengthen their network defense and protect their assets and users. These technologies cover an entire spectrum, ranging from incident prevention, detection, and investigation.

Intrusion detection systems (IDS) are generally divided into two prominent categories: signature-based techniques, and anomaly detection methods [3]. Signature-based detection involves identifying the attacks by comparing patterns with those observed in known malicious events. In contrast, anomaly detection operates under the premise that events deviating from established normal systems or user behavior may indicate potential malicious activities [4].

The Activity and Event Network (AEN) is a new security knowledge graph model that provides the foundation for developing cyber incident detection and investigation techniques [5]. The AEN framework captures network activities and events by using a large, random dynamic graph that is continuously maintained and updated as new information and data arrive.

The AEN framework includes the AEN engine, a tool responsible for maintaining the AEN graph and providing key functionalities. These functionalities encompass processing and aggregating incoming data through data receivers, attack fingerprint matching, proactive third-party data collection to supplement other incoming data, and various threat detection models [6].

To build the graph, the engine uses as inputs a variety of data sources, including network traffic data, raw or flow data, Syslogs, IDS alert logs (e.g., from Snort, Zeek, and Kitsune's), and pre-collected IP address information derived from DNS and WHOIS queries [7].

Malicious traffic detection is vital in cybersecurity to prevent threats. Machine learning algorithms are commonly used for this purpose, analyzing the network packets. In extensive networks, analyzing every packet is impractical. Instead, a sampled flow data is examined for effective malicious traffic detection purposes [8].

Snort is a lightweight network intrusion detection system developed in the C language, that is originated in 1998. It has continuously evolved and improved over more than a decade as an open-source tool. Today, it stands as globally recognized network intrusion detection and prevention software. Snort's capabilities include real-time analysis of data flow and protocols [9]. Snort's workflow consists of six essential components, namely: data packet capture, data analysis code, packet preprocessing, rule parsing, detection engine, and logging [10].

In 1994, Vern Paxson, affiliated with the International Computer Science Institute's Center for Internet Research (ICIR), developed a platform called Bro. Over time, this platform evolved as more research findings were integrated, leading to a change of its name to Zeek. Zeek serves a dual purpose, i.e., it functions as both a network security monitoring tool and a network intrusion

detection system. It is primarily coded in C++ and is compatible with multiple operating systems, including Linux, FreeBSD, and MacOS [11] [12].

On another note, Kitsune, known as a plug-and-play Network Intrusion Detection System (NIDS), is capable of autonomously learning to identify threats within a local network in a highly efficient and unsupervised online fashion. At its core, Kitsune employs the KitNET algorithm, which harnesses an ensemble of neural networks, known as autoencoders, to collaboratively distinguish between regular traffic patterns and anomalous ones [13].

After discussing the above-mentioned intrusion detection systems, our focus in this thesis is on elucidating the operational behavior of the Activity and Event Network (AEN) engine in graph analysis. This engine stores the graph in a custom-made, in-memory graph database with capabilities to add, update, remove, and search for graph elements. The AEN engine also has an administration dashboard that, besides standard administrative tasks, can be used to (i) visualize the graph fully or partially depending on its size, and (ii) manually add data and perform forensic analysis.

Because the graph can be huge (e.g., millions of nodes), a visual analysis by an analyst can be unwieldy. The objective of this thesis is to extend the AEN graph engine by developing a timeline feature that can improve the visualization by allowing the analyst to access and work on segments or portions of the graph as desired. The timeline feature will enhance the forensics capabilities of the system by allowing the analysts to move the graph visualization back and forth in time to analyze how the graph changes with time. It should efficiently support fine movement and direct access to any point in the timeline. To reduce memory/storage usage, it should also support consolidating fine-grain changes into coarser-grain ones without losing the timeline integrity and altering the order in which the changes occurred.

A graph paging model is also developed to implement the timeline feature [14], where a graph is structured into multiple pages that enable navigating back and forth, and other related functionalities. The work in this thesis involved designing and implementing the underlying data model and algorithms, and performing an experimental evaluation of the model's performance using a public cybersecurity dataset.

1.3 Report Outline

The thesis is organized as follows.

Chapter 2 provides background on the AEN graph engine and underlying gaps.

Chapter 3 presents our proposed paging graph model.

Chapter 4 presents the performance evaluation of the proposed model.

Chapter 5 provides concluding remarks.

Chapter 2 : Background and Approach

2.1 The Activity and Event Network Model

The Activity and Event Network (AEN) [5] is a security knowledge graph model used for identifying and analyzing security-related information in networks in real-time. Its components include several types of nodes and edges, as well as the related algorithms used to construct and maintain the graph. It provides a foundation to deploy various threat detection and investigation models, enabling effective monitoring, analysis, and network management, improving network performance, delivering enhanced security, and facilitating more efficient operations.

By following multiple approaches in the AEN methodology, the practical feature in this report for the AEN graph would be the ability to represent the temporal aspect of the network, capture the changes, updates, and deletions in real-time. This allows for identifying the patterns and trends over time, which can be used for predictive analysis and proactive network management [5].

The graph model is continuously updated and maintained with historical information preserved for long-term attack detection purposes. This means that the graph is expected to grow as more data is added over time, resulting in an extensive graph.

As the graph updates continuously, the graph engine, in uploading the AEN graph, needs to load the entire graph into memory for efficient malicious pattern detection. Loading the graph into memory allows for faster querying and processing of graph-based algorithms. The AEN graph engine needs to handle large-scale graph data efficiently, with optimizations for memory usage, query performance, and scalability. Mostly, these graphs are too large to fit entirely in memory

and should handle continuous updates and maintenance of the graph model without sacrificing performance.

The AEN model transforms the inputs from different data sources into graph formats, such as network packets in PCAP format, intrusion alerts from Zeek alerts, network traffic from NetFlow, and record events from syslogs. Various adaptors are used to make this data usable for the graph engine. These adaptors transform the data from each source into a readable format that can be integrated into the graph model. They parse, extract, and convert the relevant data from each source into a unified structure compatible with the graph model. These adaptors aim to bridge the gap between the diverse data sources and the graph model, enabling the graph engine to process the data from different sources and construct a consistent and meaningful graph representation. This allows one to perform a graph-based analysis, modeling, and visualization of the modeled system/network [5].

The AEN graph supports multiple graph database technologies. Currently, the implemented engines include a custom engine developed at the ISOT lab and the PGX Oracle graph engine. PGX is a comprehensive toolkit designed for graph analysis, offering support for a wide range of graph algorithms such as PageRank, advanced SQL-like pattern-matching graph queries, and graph machine learning capabilities. With PGX, the latent information embedded in the graphs can be effectively extracted as machine learning features, enabling a robust and efficient graph data analysis [15].

2.2 AEN Graph Visualization

In Figure 2.1, a part of the graph generated from processing the network traffic flow data (NetFlow) is illustrated. This graph depicts the features as nodes and edges representing the connections between them.

Figure 2.1: A visualized graph sample in NetFlow.

The graph model exhibits several key characteristics. The nodes are labeled and capable of having multiple simultaneous relationships, meaning that they can have multiple edges. The relationships

also possess their properties and defined sources and destinations. Additionally, all the nodes, edges, and attributes, are subject to continuous changes over time [5].

In this study, the graph visualization shows the latest changes, and as long as the graph gets bigger, the result of the visualization changes dynamically.

2.3 Current Issues and Proposed Solution

As mentioned in this Chapter, the visualization of the graph updates itself over time by means of the changes in the graph. Here, the term “change” means each of the inserts, updates, or deletes of any node or edge.

One of the primary limitations of the existing framework is the potential concealment of the edge updates over time. As the graph visualization only presents the last change being processed, any intermediate changes to the edge may go unnoticed by the analyst. This can result in a loss of critical information, which can thereby hinder a comprehensive understanding of the graph's dynamics. By requiring the ability to track and visualize the incremental updates to the edges, the framework limits the analyst's ability to observe and interpret the evolving relationships within the graph.

Another area for improvement with the existing framework is the challenges involved in handling large graphs. Loading the entire graph into the memory can quickly exhaust the available resources, leading to memory constraints and potential system instability. This limitation arises due to the inability of the framework to efficiently manage and store the graph data on the hard drive instead of solely relying on the memory. Consequently, the backend encounters some difficulties in sustaining the graph processing and analysis as the memory reaches its capacity.

By failing to employ appropriate memory management strategies, the existing framework inhibits the ability to handle the graphs of significant size. Implementing a graph paging mechanism that leverages the hard drive storage for data persistence would address this issue, allowing for the seamless handling of large graphs and mitigating the strain on the system resources.

As previously discussed, the graph paging solution poses some challenges in saving the entire dataset into the memory due to its large size. However, it is crucial to ensure fast results when navigating through different graph pages. To address this issue, our approach focuses on achieving an efficient and rapid performance by leveraging some alternative storage methods. Indeed, instead of loading the entire dataset into the memory, we propose to store the graph data using a disk-based storage such as multiple JSON files.

JSON, short for JavaScript Object Notation, is a lightweight and text-based format for exchanging data. JSON is frequently employed when data needs to be sent from a server to a web page or browser for display. Despite its origin in JavaScript, JSON is not limited to JavaScript usage and operates as a text-based format independently. Numerous programming languages, including Python, Java, C++, and others, can work with and handle JSON data. [16]

Additionally, our approach allows the users to upload the network security logs once and then reuse the resulting file for future analysis, promoting the reusability and minimizing the redundant computation. The proposed paging graph model is presented in-depth in Chapter 3.

Chapter 3 : Proposed Paging Graph Model

3.1 Graph Paging

In defining how the paging graph works, first, the definition of a graph should be considered. A graph consist of sets of nodes and edges, and the nodes are connected by those edges. As discussed in Chapter 2, the AEN engine ingests the network data from data sources and converts them to a graph, where nodes represent the users, the hosts, to name a few, and edges represent the relationships or connections between the nodes [6]. The derived graph shows how network traffic flows evolve through sequences of events.

The graph paging contains the sequences of pages (or steps). Each page has a subset of nodes and edges in the graph. Each node contains the information about the devices, routers, or servers. On the other hand, each edge shows the connections between the entities, and the connections between pages ensure that the graph remains cohesive and that data can be accessed and analyzed across multiple pages while maintaining the integrity of the graph representation. Graph paging can be used to highlight key features or patterns that happen in the graph and to understand how it is evolving.

A network security analyst might identify a malicious activity. The analyst needs to know when the malicious activity occurred on the network and from which entities it has started propagating into the network. In the presence of a large graph, this traffic might become lost and hidden in the system or overwritten by new traffics. A process in the graph engine is needed to show how and when this malicious traffic is added to the graph. For this reason, the system must be able to

paginate the graph into multiple graph sections, such that each section shows the propagation of the graph until reaching the page that the user has chosen. For each page, the user must be able to visualize the nodes and edges that connect those nodes, as well as all the attributes and features related to those nodes and edges. Also, the system must maintain and provide the time series duration since this page was built.

The challenges in designing graph paging primarily revolve around visualization, particularly in breaking down a graph into multiple pages while maintaining the coherence of the graph structure. This requires ensuring smooth transitions between pages. Additionally, a key challenge lies in enhancing the system performance to optimize graph paging operations.

3.2 System Architecture

The new graph paging framework is integrated into the AEN engine to save the converted graph as multiple pages in snapshots. The design is the same for any file format, such as PCAP, CSV, or text files. These files could represent packets, NetFlow, Syslog, or IDS alerts. After loading these files to extract the graphs, the AEN graph generation process starts automatically. The nodes and edges are added to the graph one after the other simultaneously; and the graph paging framework divides each defined interval as a page and saves the snapshots. Here, the interval refers to a predetermined number used to divide each page based on the occurrence of updates, deletes, or inserts in the graph. The process of reading and saving a graph paging is illustrated in Figure 3.1.

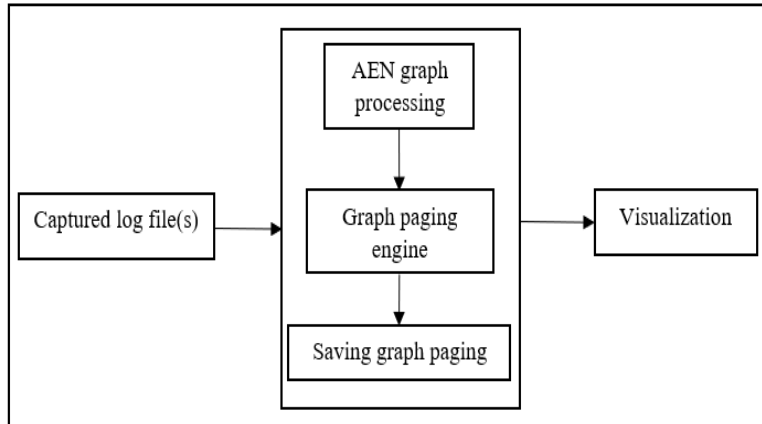

Figure 3.1 Graph Paging reading/saving process.

3.3 Graph Paging Visualization

For the graph paging visualization, it is expected that the paging of the graph be visible and the various parts of the timeline be easily accessible to the users. It would not be rational to track the pages using page numbers, especially when dealing with large page numbers. Digit paging can make the visualization hard to manage – for instance, that is the case when referring to thousands of pages like the one used in the Google search result for pagination [17]. For instance, in the graph paging process, the user might want to access a specific part of the middle of the graph while the current graph shows the first page. In this case, it would not be effective for the user to go several steps back and forth to reach that specific page.

Based on the above-mentioned considerations, the best way to develop this pagination is to use a slider to allow the user to easily slide any part of the graph. Even if the graph has millions of pages, it would be easy for the user to navigate through the pages and drag the slider to the required position on the timeline. This enables the user easily access any point of the graph. Another feature that should be accessible for the analyst is the forward and backward the moves through the graph. The user might want to visualize the changes that occur to the graph when the graph moves forward

or backward by one page, and it would be handier for the user to navigate forward and backward using buttons instead of trying to navigate with the slider.

The slider is generated using JavaScript in conjunction with HTML tags to make it dynamic. Therefore, some libraries must be added to the client side to develop the paging graph visualization. As a result, it's necessary to define new CSS libraries for creating the slider. Figure 3.2 shows the visualization for a sample packet file. Additionally, this figure displays the visualization result, including the slider and the LIVE Graph button. Also, it shows how long ago this page was built. The Host with address 192.168.10.5 is displayed on this page, with five edges with this source. As shown by the slider, this graph shows the page 5 of the paging graph.

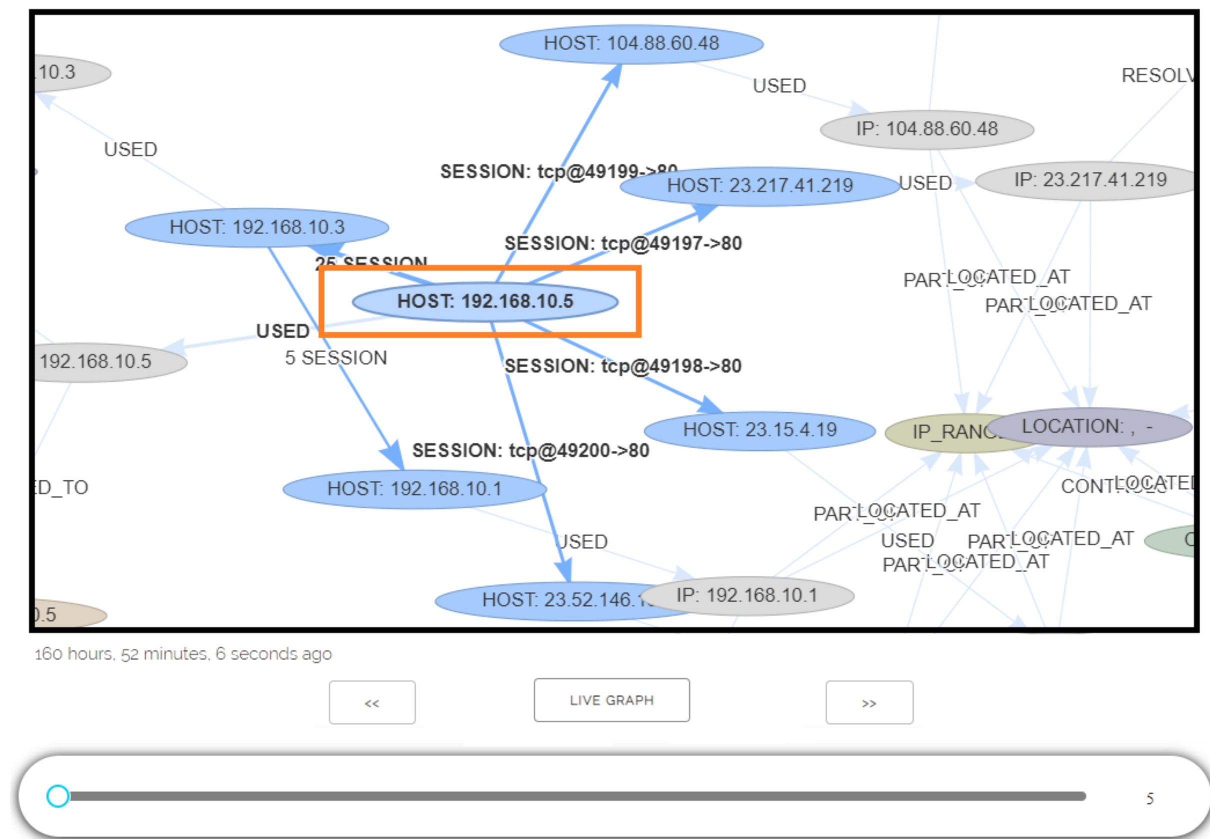

Figure 3.2: Paging Graph on page 5.

In this work, the slider CSS element, a predefined library, is used to prepare the range slider for the graph. To make this graph dynamic, a JavaScript function in the primary source of code is required [18]. As part of the procedure, the slider range is configured to match the total number of pages in the graph paging. The JavaScript source code for this function is shown in Figure 3.3.

```
1 function dynamic_slider (total_page){
2         //Set the maximum of the range in the slider as total_page
3         var html =
4         ['<div class="slider">', //the slider CSS class we used here for the div
5         '<input type="range" max="'+total_page+'">', //here we set the range of the slider
6         '</div>'];
7         selected_page = this.value;
8 }
```

Figure 3.3: Slider dynamic range set function.

The same host as the one in Figure 3.2 is shown on page 175 in Figure 3.4. As the visualization shows, approximately 30 minutes after the time shown in Figure 3.3 on this page, the same Host with the same address has much more edges to multiple destinations. Although, as s time passes, the numbers of such nodes and edges get larger and larger, it is still possible to analyze the behavior in the network using the paging graph.

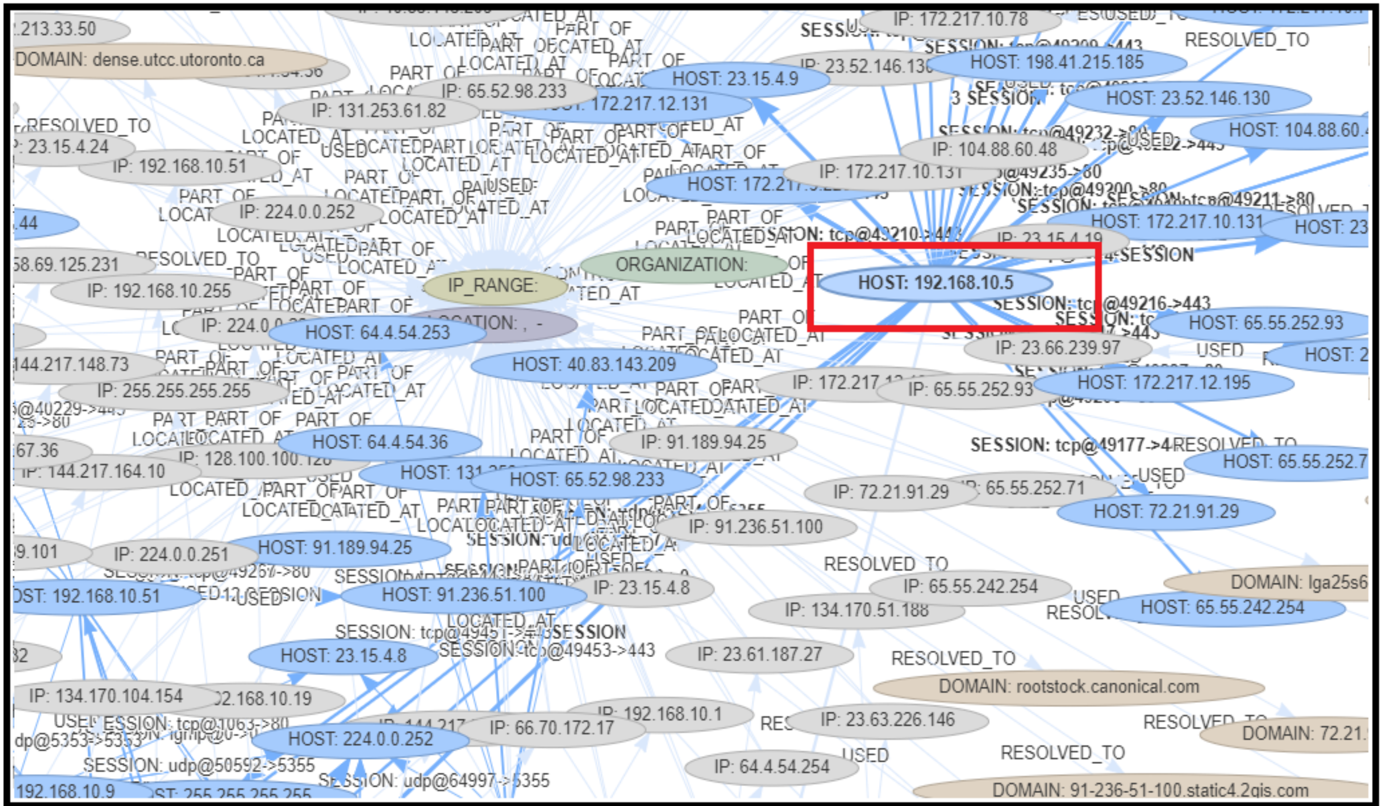

Figure 3.4: Paging graph on page 175.

In this scenario, the "LIVE GRAPH" button is a toggle switch that switches the graph mode between paging mode and continuous mode. When the graph is in continuous mode, the slider behaves differently than in the paging mode. Instead of displaying the individual pages, the slider always shows the most recent page of the graph. This allows the users to continuously view and analyze the latest updates in real-time without manually changing the slider position. However, the users can manually adjust the slider if they navigate to a specific page, triggering a transition

back to the sliding paging graph mode. This allows the users to switch between continuous viewing and specific page navigation based on their preferences and analysis needs.

3.4 Graph Saving Algorithm

The graph paging framework enhances the existing graph functionalities by introducing a variable that maintains the live graph data. Without the graph paging framework, this variable is responsible for storing and providing access to the final graph data at each point in time as the network security data proceeds. As new data is read from the network logs, the final graph is updated to incorporate the changes, reflecting the evolving state of the graph. With the introduction of the graph paging framework, the variable continues to store the live graph data.

As discussed earlier, one of the challenges is the saving process while reading the data. The reason is that each page of the graph paging is a continuation of the previous page, and the conventional approach of saving the complete page data each time raises several issues. Firstly, it can lead to redundant data being repeatedly stored, resulting in inefficient usage of storage resources. Secondly, reading and saving the entire page data for each update can be complex, time-consuming, and that demands significant available memory, which is not always guaranteed.

To overcome these challenges, an alternative approach is required. Rather than saving the complete page data every time, an incremental saving mechanism is implemented. This approach ensures that only the changes made on each page are saved, eliminating redundancy and optimizing the storage usage. Keeping only the incremental changes makes the reading and saving process more streamlined and efficient, requiring less memory and reducing the processing overhead. The proposed approach is described as follows.

- First, each element must be temporarily stored in a variable based on the element status. Each element could be a node or edge with the status of being an insert, update, or delete element. For updates and inserts, the elements are added to a temporary list. The model has a separate category for deleted items that encompasses the elements that will be deleted on each page, and stores the deleted elements separately from the nodes and edges, but within the same paging category. While the data is being read after the nodes and edges of each page are fetched, the algorithm checks if any elements that need to be deleted exist on the current page.

- Secondly, the number of elements is checked to ensure that the number of rows in the temporary variable containing the elements does not exceed the predetermined limit for each page. For this thesis, 100 elements have been setup per page, including the nodes and edges. During this step, the elements are examined to determine if they have already been saved on the same page. If an element is found to be duplicated, it is replaced since it is redundant to have the same element multiple times within a single page. When the page is read, any repeated elements are only returned up to the latest change, making it unnecessary to save them multiple times. Ignoring the saving of duplicate elements has been proven to be advantageous in terms of storage efficiency and maintaining a coherent graph representation. By implementing this check and replacement mechanism, the graph paging framework ensures that each page contains a unique set of elements, thereby eliminating the redundancy and optimizing the storage space that is required.

- Third, when the algorithm proceeds to the next stage once the number of elements reaches the limitation. At this point, the saved elements undergo a process of retrieving the page-specific details from the ongoing live graph for each element. This ensures that the

temporary variable, where the elements are temporarily stored before being saved as a snapshot in the persistent data file, contains the most up-to-date and accurate information reflecting the final changes made in the live graph. By fetching the details from the live graph variable, the graph paging framework ensures that the elements saved within each page accurately represent the current state of the graph, while incorporating the latest modifications and achieving the updates that are specific to that page.

- Fourth, the data prepared for a page is serialized and stored as a complete record within the JSON file. This process involves writing the collected information into a structured format, including the nodes and edges. It should be recalled that the JSON file is a repository for the entire page's data, enabling efficient retrieval and utilization when needed. In this step, to provide the user with the information about the creation time of each page, the user's local machine's time and date are captured and stored as separate variables within the same row of the JSON file. This ensures that the temporal context of each page is preserved, allowing for meaningful analysis and reference to specific points in the graph's timeline.

The flowchart of this algorithm is shown in Figure 3.5.

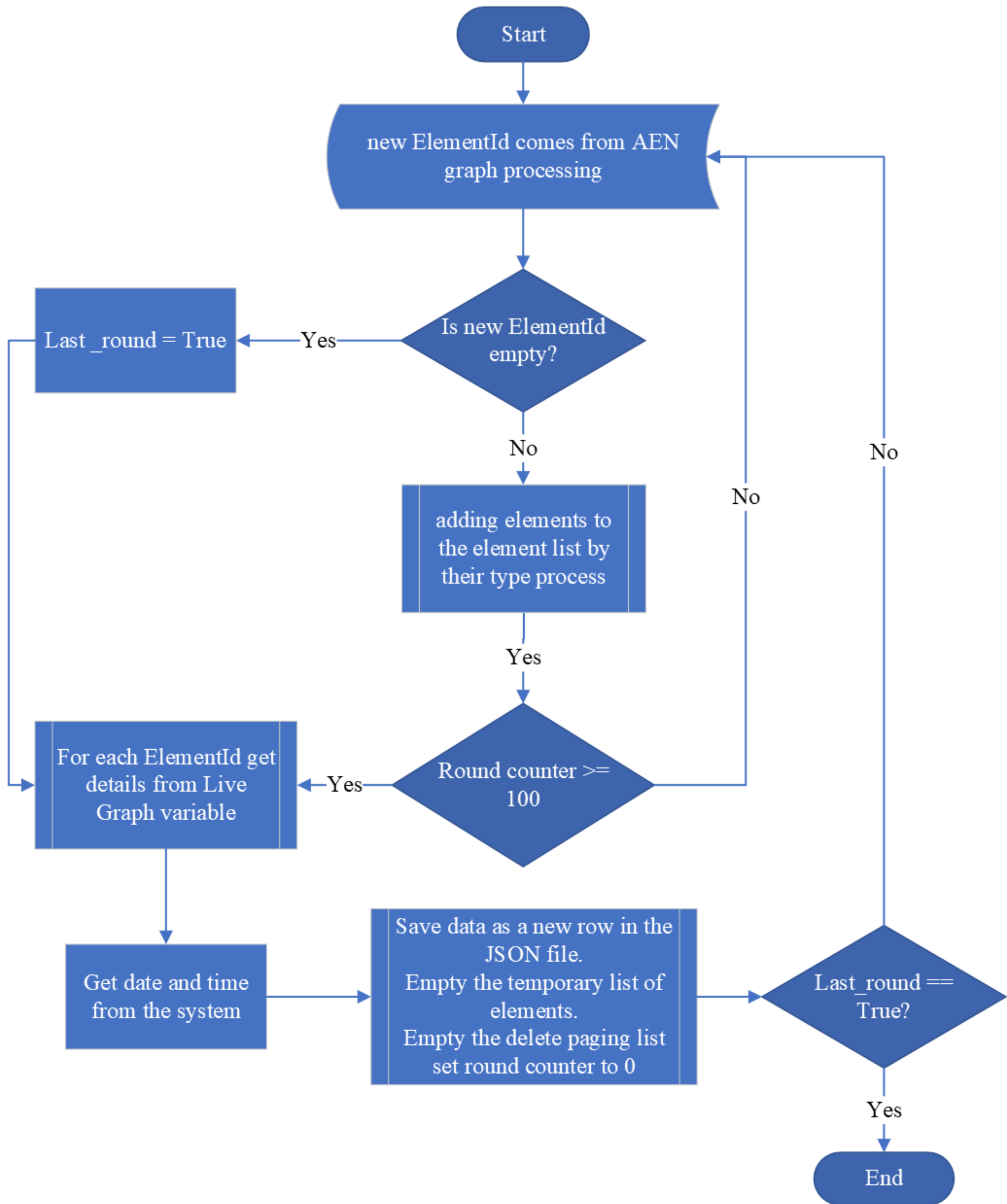

Figure 3.5: Saving graph paging flow chart.

The model to save each page in the JSON file is represented in Figure 3.6

```
 1 class GraphDtoInTimeProgress:
 2       time: LocalDateTime
 3       graph: GraphDto
 4       set<DeletePagingDto>: deletePagingDtos
 5


 6  class  GraphDto:
 7       nodes: Set<NodeDTO> //Here Set is used for
 8       edges: set<EdgeDTO> //having unique elements
 9
10 class NodeDTO:
11       id: long
12       label: String
13       properties: Map<String, Object> //This Map is used
                              //for setting key-value pairs
14
15 class EdgeDTO:
16               id: long
17               label: String
18               source: long
19               destination: long
20       properties: Map<String, Object>

21 class DeletePagingDto:
22               elementId: long
23               nodeOrEdge: String //shows from which category element
                               // should be deleted
```

Figure 3.6: Graph Paging model.

In Figure 3.7 an example of the saved data in the JSON file is given. As shown, each row has graphTime, graphDTO, and deletedPagingDtos, as fields. In each graphDTO field, the information about the nodes and edges in that row (a page in graph visualization) is given. Each JSON file has one hundred rows, each row representing a separate graph page. Limiting each file to one hundred rows is justified by the need to prevent the file size from becoming excessively large. By maintaining a manageable file size, the process of reading and accessing the data is made more efficient and facilitated by the buffers. The buffer mechanism ensures that the data can be read and processed in smaller, more manageable chunks, hence optimizing the system's performance.

```
[
{"graphTime":{"date":{"year":2023,"month":3,"day":23},"time":{"hour":22,"minute":7,"second":15,"nano":624788300}},
 "graphDTO":{
    "nodes":[{"id":3871810368739712711,"label":"DOMAIN","properties":{"name":"192.168.10.5"}},{"id":435433887312481
    "edges":[{"id":-5988210145712461342,"label":"IP_RES_DOM","source":-107305210130788307,"destination":38718103687
 "deletePagingDtos":[]},
{"graphTime":{"date":{"year":2023,"month":3,"day":23},"time":{"hour":22,"minute":7,"second":31,"nano":27368000}},
 "graphDTO":{
    "nodes":[],
    "edges":[{"id":8593916135139200578,"label":"SESSION","source":-4181233445582162238,"destination":30760904714718
 "deletePagingDtos":[]},
{"graphTime":{"date":{"year":2023,"month":3,"day":23},"time":{"hour":22,"minute":7,"second":34,"nano":443660900}},
 "graphDTO":{
    "nodes":[],
    "edges":[{"id":-5714871482538588102,"label":"SESSION","source":-4181233445582162238,"destination":30760904714718
 "deletePagingDtos":[]},
{"graphTime":{"date":{"year":2023,"month":3,"day":23},"time":{"hour":22,"minute":7,"second":35,"nano":480291600}},
 "graphDTO":{
    "nodes":[{"id":3999157850501481142,"label":"IP_RANGE","properties":{"cidr":""}},{"id":2166010918664031629,"labe
    "edges":[{"id":5239137677215005983,"label":"SESSION","source":-4181233445582162238,"destination":30760904714718
 "deletePagingDtos":[]},
{"graphTime":{"date":{"year":2023,"month":3,"day":23},"time":{"hour":22,"minute":7,"second":36,"nano":484542300}},
 "graphDTO":{
    "nodes":[{"id":-5426568496156666703,"label":"ORGANIZATION","properties":{"name":""}},{"id":2780074772380241575,
    "edges":[{"id":-8751324207754967101,"label":"IP_PART_RANGE","source":-1138839507777431303,"destination":3999157
 "deletePagingDtos":[]},
```


Figure 3.7: Actual saved data as a graph paging in a JSON file.

Figure 3.7 illustrates that the nodes and edges exist on each of the pages. Within the graph paging framework, each page represents a specific subset of data and captures 100 changes. These changes encompass various operations such as insertions, updates, and deletions. Each page contains up to one hundred elements consisting of nodes and edges. In Figure 3.8, 26 elements in 9 nodes and 17 edges are shown. However, for these 26 elements, 74 updates have been made to the same nodes or edges within this page. It is important to note that, despite the occurrence of these 74 updates, the saving paging graph algorithm does not save each update on the page. Therefore, the 74 updates on this page have not been stored individually as the paging graph framework employs some mechanisms to keep the latest change of each node or edge.

```
[{"graphTime":{"date":{"year":2023,"month":3,"day":23},"time":{"hour":22,"
"graphDTO":{"nodes":[
{"id":3871810368739712711,"label":"DOMAIN","properties":{"name":"192.168.1
{"id":4354338873124818843,"label":"ORGANIZATION","properties":{"name":"Int
{"id":3076090471471850113,"label":"HOST","properties":{"malicious":false,"
{"id":7128423359328452096,"label":"LOCATION","properties":{"tag":"Los Ange
{"id":-2257014382250022190,"label":"DOMAIN","properties":{"name":"192.168.
{"id":7143889664883696731,"label":"IP_RANGE","properties":{"cidr":"192.168
{"id":-107305210130788307,"label":"IP","properties":{"ip":"192.168.10.5"}}
{"id":4968668182114427654,"label":"IP","properties":{"ip":"192.168.10.3"}}
{"id":-4181233445582162238,"label":"HOST","properties":{"malicious":false,
"edges":[{"id":-5988210145712461342,"label":"IP_RES_DOM","source":-1073052
{"id":-8102870633015006489,"label":"IP_RES_DOM","source":49686681821144276
{"id":-2904704631738962696,"label":"SESSION","source":-4181233445582162238
{"id":6700652168816056619,"label":"SESSION","source":-4181233445582162238,
{"id":901952854402985456,"label":"IP_PART_RANGE","source":-107305210130788
{"id":6374989873391077751,"label":"IP_LOC_AT","source":-107305210130788307
{"id":-6215881316194415904,"label":"ORG_LOC_AT","source":43543388731248188
{"id":7438604270233780017,"label":"HOST_USED_IP","source":3076090471471850
{"id":3433591142846792191,"label":"IP_LOC_AT","source":4968668182114427654
{"id":8593916135139200578,"label":"SESSION","source":-4181233445582162238,
{"id":-5496509415953422668,"label":"IP_PART_RANGE","source":49686681821144
{"id":-2642186098089445495,"label":"HOST_USED_IP","source":-41812334455821
{"id":5240352804976821672,"label":"SESSION","source":-4181233445582162238,
{"id":5563655078922665923,"label":"SESSION","source":-4181233445582162238,
{"id":6181998448569162079,"label":"SESSION","source":-4181233445582162238,
{"id":4070203016130282628,"label":"ORG_CTRL_RANGE","source":43543388731248
{"id":2169297505237947969,"label":"SESSION","source":-4181233445582162238,
"deletePagingDtos":[]},
```


Figure 3.8: Sample JSON of actual saved nodes and edges.

3.5 Saving Multiple Files

As stated earlier, the paging graph framework uses multiple JSON files to save the graph paging snapshots, and this type of files are primarily used to store or transmit the data. This type of file also has a key-value pair format of data, which makes it easily readable, as it does not have a data type, which makes it easy to parse. Another benefit of JSON file type is that it is lightweight compared to many other data formats such as CSV or XML. These reasons make it adequate for storing the pages of massive graphs and efficient when reading them. For instance, an already

processed graph might have more than a million pages, and the user might choose page 500 to see what the graph looks like on that page. In this case, it would not make sense for the system to load as many as a million-page graph data and then display its first 500 pages. For this reason, the graph data access and retrieval scalability should be improved, for instance by dividing the data into multiple JSON files. Also, instead of loading a large graph content into the memory, by allocating multiple files to the enormous data, small contents are pushed into the buffer, then get processed, and the buffer will be released. Each file contains 100 pages, and each page has up to 100 elements in a graph, so each file has at maximum 10000 elements in the graph.

3.6 Graph Reading Algorithm

In the process of saving graphs into the data storage files, the corresponding file names are captured and stored in a dedicated local variable. The file names are sorted in ascending order based on the chronological order of their creation. This ensures that the file names are organized sequentially, with the earliest created file appearing first in the sorted list. Subsequently, when the user specifies a particular page number, the algorithm undertakes a search operation to identify the file name that corresponds to the desired page. Due to the fact that there are 100 pages stored in each file, the page number is divided by 100 to determine how many files should be read completely. The next step is to determine how many pages of the last file would be read; and the remainder of the division provides such value.

Once the algorithm has determined the file name and line number where the selected page number is located, it proceeds to read all the preceding files until the algorithm reaches the file containing the desired page. This sequential reading process is to receive all the necessary data to build the graph up to the selected page.

The graph reading process starts after getting the data from each graph snapshot file. Once the JSON data is parsed and represented as Java objects, the algorithm extracts the nodes and edges from the parsed data. Then, it checks if each node or edge already exists in the graph. If true, the algorithm updates the existing node or edge with the new data from the current page. If not true, the algorithm adds the new node or edge to the graph. By iterating through the previous pages and updating the graph with any repeated data encountered in subsequent pages, the algorithm ensures that the selected page accurately reflects the state of the evolving graph. For each page, a list of deleted elements is saved. After retrieving the page's content, the algorithm examines the deleted objects. The function removes the corresponding object from the final graph for each element that matches an entry in the deletion list.

For the last file, mostly the whole data would not be needed. Therefore, the system calculates the remainder of the selected page number and goes into a loop to extract that specific page number of the file.

In saving the paging graph, a separate file stores a complete snapshot of every 1000 pages. This approach effectively reduces the algorithm's complexity while minimizing the need to navigate back and forth through multiple files. The algorithm periodically creates snapshot files to preserve the entire graph as a distinct entity at specific intervals. This approach provides several benefits, including improved performance by reducing the time required to access and process the graph data. Additionally, it simplifies the retrieval and analysis of specific snapshots since each file represents a self-contained snapshot of a segment of the graph's timeline. Overall, the snapshot-based file organization strategy optimizes the handling and management of the paging graph data. While reading the graph data by choosing a page number, whenever the page number selected is more extensive than that measurement, the algorithm reads the whole graph snapshots until that

point and continues the process as before. The flow chart of the reading paging graph process is shown in Figure 3.9.

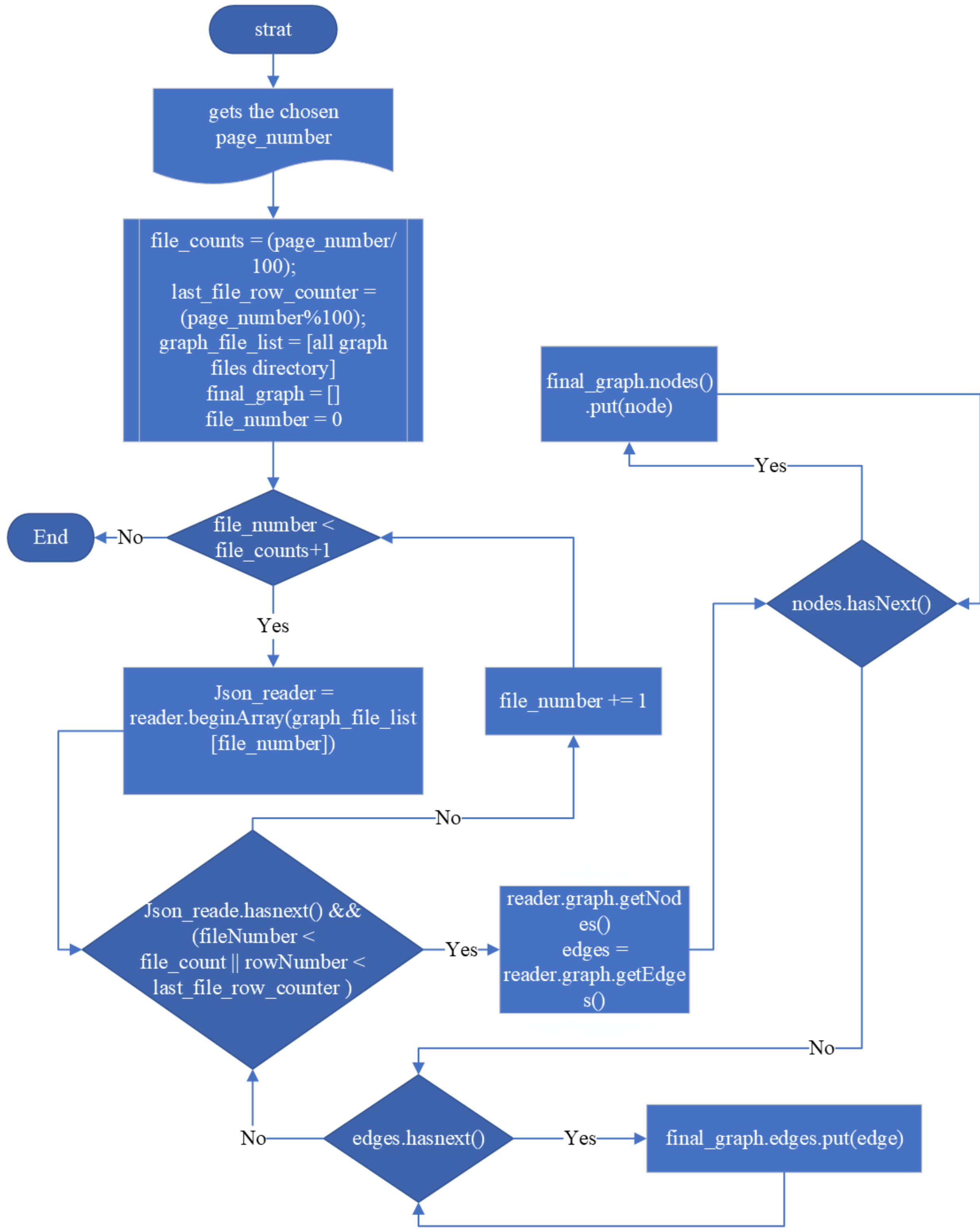

Figure 3.9: Reading graph paging flow chart.

3.7 Reading existing graph paging

As the system saves the result of the graph paging into the client-side local hard drive, whenever the graph paging framework executes and receives the first request from the client, it automatically checks the folder which stores the snapshots of the graphs to see if there exist any paging graph files and if this is the case, it executes the same process of reading the graph and rendering it in pages. But whenever the user uploads new data into the system, the previous data should be replaced with the new one. This feature allows stopping and restarting the system without needing to start the paging process from scratch, which reduces the resource and time consumed to render the graph.

Chapter 4 : Implementation and Performance Evaluation

4.1 Dataset

The dataset used in this thesis is a subset of CICIDS2017 [19], a publicly available dataset released by the Canadian Institute for Cybersecurity (CIC) at the University of New Brunswick (UNB). It contains a diverse range of network traffic samples, including benign data (normal human activities in the network) and different types of cyber-attacks, including Cross-site scripting (XSS), denial of service (DOS), distributed denial of service (DDOS), port scan, botnet, surreptitiously gaining unauthorized access to a system. The data was captured between morning and afternoon from Monday, July 3, 2017, to Friday, July 7, 2017. The dataset is in PCAP format, and the specific subset used in this thesis was captured on Tuesday, July 4, 2017, and the file name is Tuesday-WorkingHours.pcap of size 10.78 Giga bytes.

Due to limited hardware resources, dividing 10.78 GB within this thesis's scope was necessary. The Pcap file was divided into smaller, more manageable segments. This partitioning was undertaken to facilitate the handling and analysis of the network packet traffic logs contained within the file. Each segmented file had a subset of the original data, allowing for efficient processing and examination. After partitioning the .Pcap file into smaller segments, each segment was loaded into the graph paging framework individually. Once loaded, the segments were converted into graph snapshots, representing the network traffic data. Additionally, each segment was added to the previously loaded segments, resulting in a cumulative representation of the network traffic within the graph paging framework. This approach allowed for a comprehensive analysis of the network data by considering the entirety of the segment Pcap file. This thesis

utilizes a subset of the original data extracted from the .Pcap file. Specifically, the first 10 blocks, each consisting of 100,000 entries, are selected as the representative subset. This approach ensures that a manageable portion of the data, totaling 1,000,000 entries, is analyzed to draw meaningful conclusions. By focusing on this subset, this thesis aims to provide some insights and findings that can be generalized to the larger dataset from which it was derived.

4.2 Evaluation Environment and Procedures

The performance of the graph paging visualization and graph processing framework was evaluated by measuring the RAM utilization, response time, and CPU utilization. This evaluation was conducted on an Intel Core i7 processor with 1.8 GHz CPU speed and 8GB of RAM.

The response time is calculated from the moment when the packets started began loading into the system until the completion of the loading process, and the graph renders completely. The CPU usage in terms of percentage of total CPU (respectively the RAM utilization in terms of Giga bytes) measures the amount of CPU utilized by loading the packets (respectively the amount of RAM used for rendering the graph entirely). Also, the average response time is measured in seconds.

The evaluations were categorized into three groups. The first category focused on the framework without a paging visualization layer, the second category one cantered on the graph paging framework, and the third category involved a separate assessment, specifically loading the JSON results of the snapshots.

To evaluate the performance of the graph paging framework, the same Pcap segments utilized in the previous segment graph operation were selected. Specifically, the evaluation focused on the first 5 segments chosen for analysis. This approach allowed for a comparative assessment of the

framework's performance across multiple segments, providing valuable insights into its efficiency, scalability, and effectiveness in handling the selected network data. Each segment was submitted as input to the system 5 to 7 times. Here, reputation was required to obtain reliable results, and a repeated submission was performed to ensure robustness and accuracy in the computed average outputs.

4.3 Evaluation Results

Table 4.1 presents the collected data from loading the first 5 segments of the Tuesday-WorkingHours.pcap file into two computational frameworks: one incorporating the graph paging mechanism and the other devoid of this functionality. This comparative analysis aims to assess the performance characteristics and operational efficiencies of the two frameworks (i.e., with and without graph paging) in handling the CPU and RAM usage before loading any segment, and after loading a segment and observing the final graph. Before loading, the engine is started, but no file is loaded into the system. Figures 4.1, 4.2, and 4.3 summarize the results from Table 4.1 as histograms.

Section (Size)	With Graph paging	Before loading	After loading	
Segment1 (2 Megabytes)	Yes	1	30	CPU usage (%)
	No	0.4	40	
	Yes	1.6	3.8	RAM Usage (Giga bytes)
	No	2	3.6	
Segment2 (2.5 Megabytes)	Yes	0.8	46	CPU usage (%)
	No	0.3	51	
	Yes	1.7	3.3	RAM Usage (Giga bytes)
	No	1.7	2.9	
	Yes	0.6	55	CPU usage (%)
	No	0.4	63	

Segment3 (4.6 Megabytes)	Yes	1.8	4	RAM Usage (Giga bytes)
	No	1.8	3.7	
Segment (6 Megabytes)	Yes	0.4	58	CPU usage (%)
	No	0.5	50	
	Yes	1.8	4	RAM Usage (Giga bytes)
	No	1.65	2.9	
Segment5 (6.5 Megabytes)	Yes	0.3	60	CPU usage (%)
	No	0.9	70	
	Yes	1.85	3.8	RAM Usage (Giga bytes)
	No	1.8	3.6	

Table 4.1: Comparison of RAM and CPU usage evaluation for different file sizes.

Figure 4.1 shows the response times for the same 5 files. The response time for the “Without Graph paging” and “With Graph paging” computational frameworks are measured from when the packet capture is loaded until the complete graph is built and rendered. The “JSON file snapshot” framework does not require the user to upload any segment or network traffic file. In this framework, the network traffic is loaded once into the graph paging computation framework, and snapshots of the graph related to the same network traffic are saved to the local machine’s hard drive, and each time the users wants to refer to the graph, they do not need to upload the network traffic logs again. In this framework, the response time is the duration from when the graph paging framework is started until when the entire graph is rendered. Also, the time required for loading and converting a dataset into a graph does not depend on the file size. Instead, it depends on the graph processing and the amount of computation that it needs.

Figure 4.1 demonstrates a notable reduction in response time when using the framework with the paging graph mechanism compared to the framework responsible for graph calculation alone. Specifically, the response time for processing the same files loaded into both frameworks is halved when the paging graph mechanism is employed. The significant disparity in performance can be

attributed to the distinct operational characteristics of the two frameworks. In the framework with the paging graph mechanism, the data loading process enables the visualization to present one page of the graph at a time without requiring real-time requests for the graph results. Consequently, this approach substantially reduces the frequency of data transfers between the frontend and backend components. Moreover, as the visualization remains static without any user-initiated actions, there is no need for continuous updates. Figure 4.1 further reveals a significant disparity in the average response times between the graph paging mechanism and the mechanism that generates the graphs using the JSON snapshots. Notably, the average response time (in seconds) for the graphs generated through the JSON snapshots approach is up to 7 times lower than that obtained for the graphs generated using the graph paging mechanism. This involves just the time it takes for the framework to submit a request to the backend and render the result through the visualization module. This is because the time it takes for the JSON file to load into the buffer is near 0.

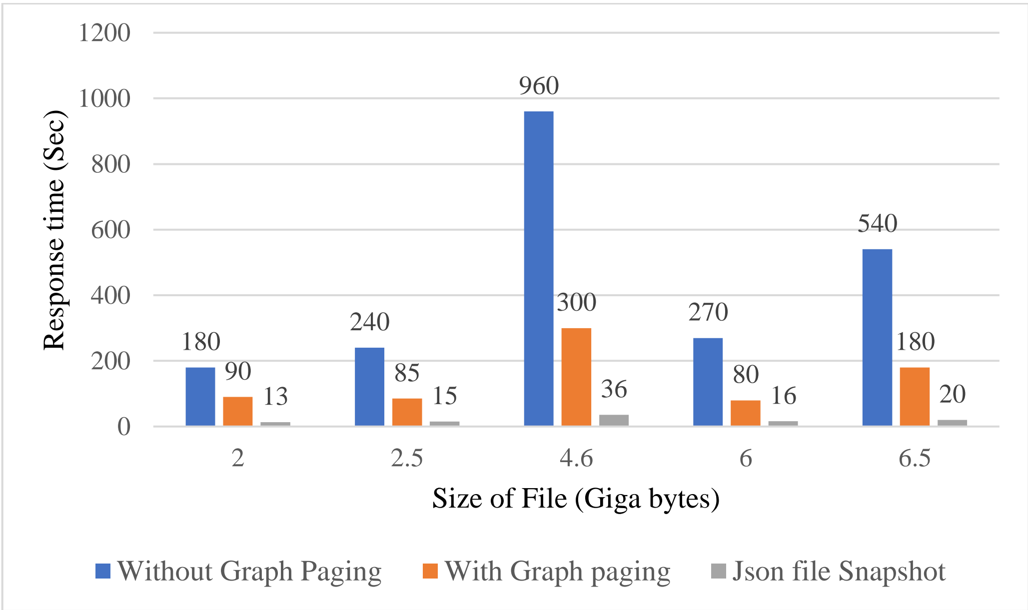

Figure 4.1: Comparing response time for the first five files of Tuesday-WorkingHours.pcap file for three different reading formats.

Figure 4.2 shows the average CPU usage from when the input files are selected until the complete graph is generated and rendered. For the graph fetched from the JSON file using the snapshot graph reading mechanism, the average CPU usage is calculated from when the engine starts until the last page of the graph is finally shown.

A comparative analysis between the "Without Graph paging" and "With Graph paging" graph computation frameworks reveals that the average CPU performance exhibits a modest improvement, albeit not as substantial as the observed enhancement in response time. This discrepancy can be attributed to the underlying characteristics of the calculation process, which remain unchanged between the two frameworks. The process of converting the packets to the graph follows the same algorithm in both cases, ensuring some consistency in the network activity capture. The marginal improvement in CPU performance can be attributed to the alteration in the visualization graph update mechanism within the new graph paging framework. Indeed, unlike the previous framework, in the graph paging framework, the visualization does not need to request an updated graph each time the graph gets updated. Consequently, the relatively small improvement in CPU performance can be attributed to the absence of the visualization graph update, resulting in a reduction in the computational overhead. Figure 4.2 also illustrates a substantial reduction in the CPU utilization when reading the graph from the JSON file snapshots. A comparison of CPU usage between the graph paging mechanism, which involves converting the network traffic to the graph, and the mechanism that solely reads and saved the JSON graph snapshots of the packet files, reveals a noteworthy shift in the CPU usage range. It is observed that the range has expanded from 30% - 60% for the graph paging mechanism to a significantly lower range of 0.2% to 3.7% for the mechanism utilizing JSON graph snapshots. This significant decrease in CPU utilization can be attributed to the contrasting processes employed by the two mechanisms. In the graph

paging mechanism, the conversion of the network traffic to the graph incurs substantial computational demands, resulting in higher CPU usage. However, the mechanism that directly reads the pre-saved JSON graph snapshots eliminates the need for extensive computation, resulting in a significantly reduced CPU utilization.

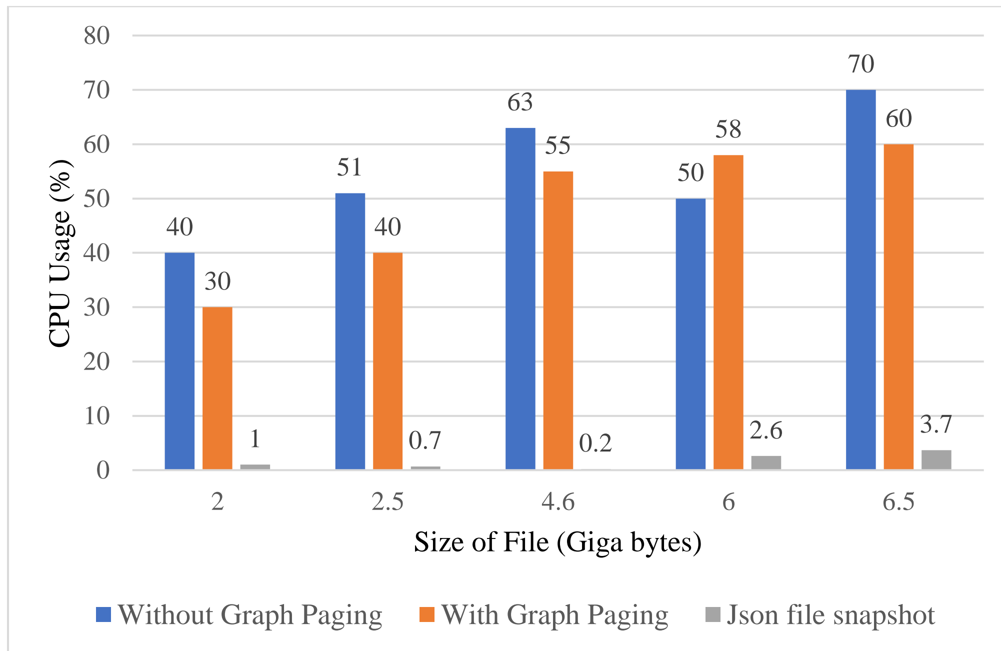

Figure 4.2: Comparing CPU usage for the first five files of Tuesday-WorkingHours.pcap file for three different reading formats.

Figure 4.3 shows the RAM utilization by loading the same files and converting the network traffic packets to graphs with and without the graph paging framework, and provides the RAM utilization in reading the graph snapshots from the JSON files. The amount of RAM used in the graph paging framework increased slightly. The reason for this increase in RAM utilization can be attributed to the additional step of saving the paging graph nodes and edges to the hard drive in the graph paging framework. Simultaneously, saving the live graph, employed in the framework without graph paging, is also in progress. Consequently, these concurrent processes contribute to a slight rise in the amount of RAM utilized within the graph paging framework. It is essential to acknowledge

that a better RAM utilization performance could be achieved if the live graph process was eliminated from the graph paging framework. However, there is no significant change in the RAM usage. Reading the graphs from the JSON file snapshot, the snapshot still performs better than the framework generating the graph in parallel with the visualization of the graph.

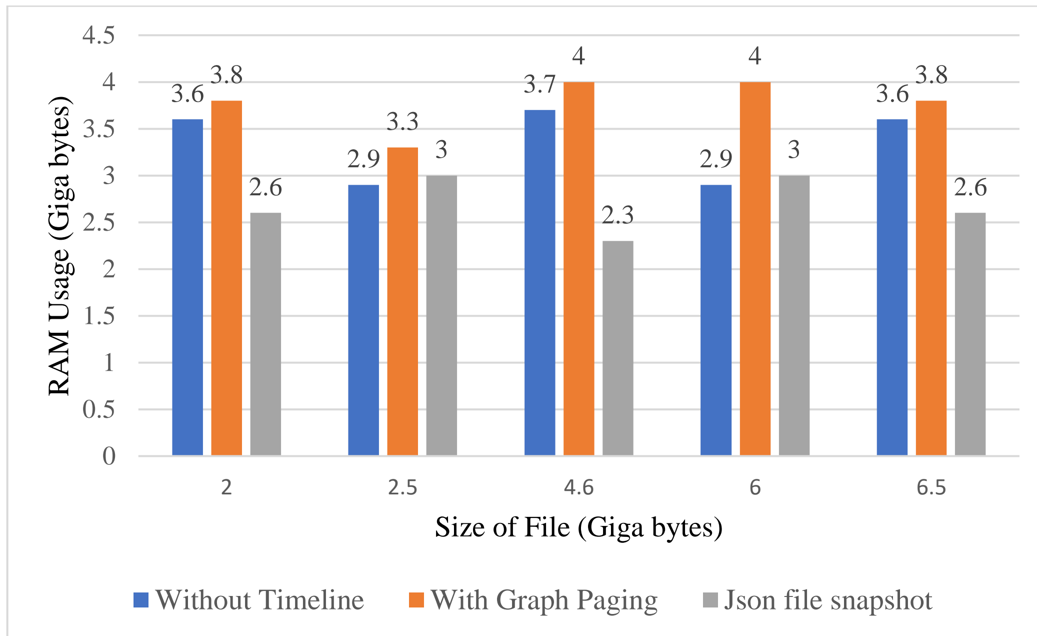

Figure 4.3: Comparing RAM usage for the first five files of Tuesday-WorkingHours.pcap file for three different reading formats.

This thesis's primary focus was the generation of a comprehensive timeline visualization using graph paging techniques. The primary dataset under examination was the Tuesday-WorkingHours.pcap file. The study aimed to assess the performance characteristics across different graph pages by implementing the timeline for the graph paging approach. The key outcome of this research is the successful creation of a graph timeline visualization that provides some valuable insights into the performance dynamics of the graph, particularly on different pages. Through analysis and evaluation, this thesis effectively captures and presents the variations in

performance within the context of the Tuesday-WorkingHours.pcap dataset, contributing to a deeper understanding of the graph's behavior and resource utilization.

Figure 4.4 shows the CPU and RAM usage while sliding through the timeline. 100 changes mean that the paging graph is on the first page (each page contains 100 changes indicating insert, update, and delete). The second trend is for 10,000 changes corresponding to page number 100, and it goes on until a million changes, page number 10,000.

In Figure 4.4, it is observed that the CPU and RAM usage gradually increases over time. This trend can be attributed to the specific methodology employed in the thesis, where every 1000 changes within the graph are saved as separate files. In the worst-case scenario, the most computationally intensive calculations occur between the 999th and 1000th pages. Consequently, this heavier processing workload contributes to the incremental rise in CPU and RAM usage observed in the graph visualization. In this scenario, the graph is ready for use, and the workload for both the RAM and CPU primarily involves data transfer. Since all the required data has already been stored in a JSON file, the CPU usage is allocated mainly to reading and processing the final dataset. On the other hand, the RAM is predominantly used for transferring the necessary data to the visualization module. As a result, the CPU's role revolves around efficiently extracting and analyzing the data, while the RAM's function centers on facilitating the smooth transfer of the data for visualization purpose.

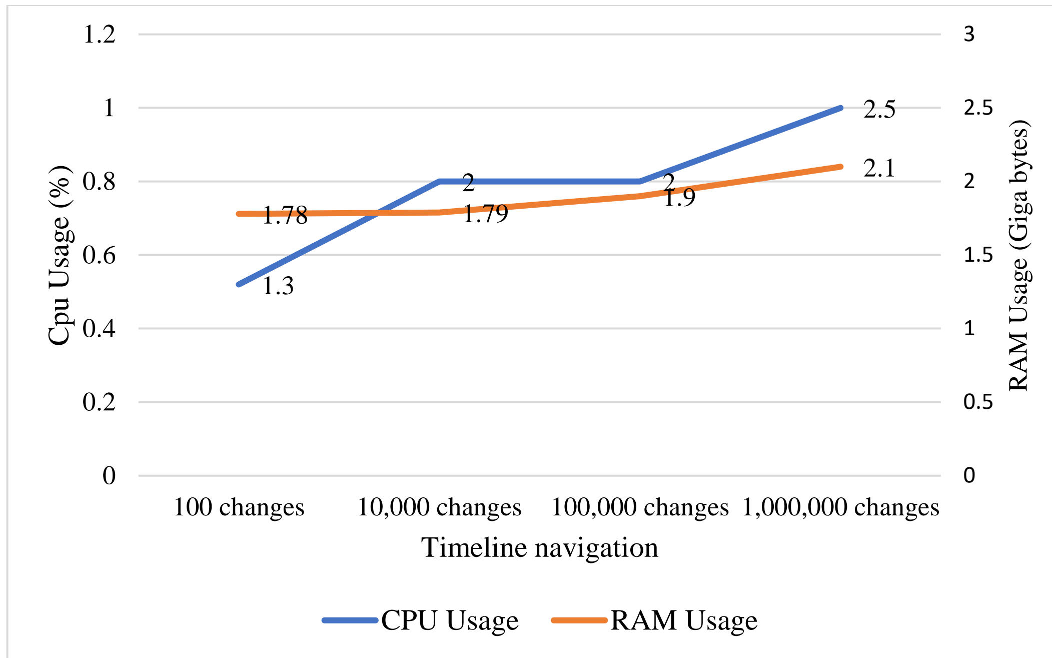

Figure 4.4: CPU and RAM usage when using the timeline.

From the results, the proposed graph paging framework is shown to improve the performance in terms of CPU utilization and response time compared to the system without the use of this framework. Also, the results for reading JSON files individually show much better performance than using the graph paging framework, which renders the JSON snapshots for the graph and reads the captured network packets to convert them to the graph in parallel. Based on the evaluation results, it is recommended to load the entire graph without utilizing the visualization feature on a device equipped with high-performance hardware. Indeed, this approach allows for efficient handling of the graph data, leveraging the capabilities of robust hardware resources. The system can benefit from faster and more streamlined operations by prioritizing the loading process and bypassing the visualization component. Following the preparation of the JSON files, the users can utilize them in any system, even those with lower-level hardware, for visualizing the results.

Chapter 5 : Conclusion

As the volume and complexity of network security data continue to grow exponentially, there is a pressing demand for visualization tools to analyze the network activities and related data. More relevant and detail-oriented visualization tools help the analysts in finding the source of the malicious data more efficiently and avoid spending lots of time on unnecessary or irrelevant data. In this thesis, a paging mechanism is developed to provide a detailed view of the AEN graph, which not only exposes more details of the progress of the requests and data transferred through the network, but also makes the visualization much faster and performs better on complex and expansive graph structures. The new visualization layer helps the analyst to identify the engagement of other IP addresses from outside the network into the network system and how traffic flows and back forth the network security perimeter. It also helps in capturing and highlighting the timeline for these changes to the system. Besides, the age of each graph element could help the analyst follow the evolution of the attacking processes. With the help of the paging visualization layer, analysts could now handle the entire graph in live proceedings by working on the individual pages. The paging visualization breaks down the graph and reduces the time needed to analyze where a specific node comes into the system and when it terminates. The performance evaluation demonstrates a clear improvement in the overall performance of the generating graphs when utilizing the graph paging framework compared to the framework without graph paging. Also, it is found that including the graph paging framework enhances the usability of graph visualization files and makes them significantly faster in their execution. This improvement is primarily achieved by eliminating the excessive calculations and time-consuming operations.

As future work, some improvements can be made to the proposed graph paging mechanism as follows. The utilization of page-to-page changes offers a viable solution for expediting the analysis processes. Instead of recalculating the entire graph from the first page whenever a new page is selected, this approach leverages the existing results and selectively incorporates or removes the differences in nodes and edges. Leveraging these incremental changes makes the analysis process significantly faster and more efficient. This methodology effectively capitalizes on the previous calculations, allowing for a more streamlined analysis experience while maintaining the integrity of the graph representation. Overall, adopting page-to-page changes contributes to faster analysis and facilitates a more efficient exploration of the graph data. Also, another tab could be added to the system to show the difference that happened between the pages and enable the analyst to determine whether the malicious request is happening at the page and should engage in deeper analysis on that page. An additional helpful feature would be to give the analysts some flexibility in defining the range of changes they wish to observe within each page instead of having it as a fixed instance variable. This customizable option empowers the analysts to fine-tune their analysis process according to their specific requirements and preferences. By allowing the analysts to specify the range of changes, such as a particular timeframe or a subset of modifications, they can focus on the relevant portions of the graph data and tailor their analysis accordingly. This feature can enhance the user's experience by providing more control and adaptability. Finally, it should be possible for the analyst to view the time-series statistics based on the stored timeline data. This would require (i) adding the state to timeline waypoints in order to keep the time-series statistics and other state data not observed in the graph, and (ii) identifying and employing some algorithms to extract the time-series statistics from the stored timeline data calculated from waypoint states.

References

[1] Samson Ho, Saleh Al Jufout, Khalil Dajani, Mohammad Mozumdar. (2021). "A Novel Intrusion Detection Model for Detecting Known and Innovative Cyberattacks Using Convolutional Neural Network." IEEE Open Journal of the Computer Society, January 2021.

[2] Canada cyber security and cybercrime statistics (2020-2022), https://www.comparitech.com/blog/information-security/canada-cyber-crime-statistics/ (Last visited Oct. 9, 2023)

[3] Delafontaine, V., Schiano, F., Cocco, G., Rusu, A., Floreano, D.: Drone-aided localization in LoRa IoT networks. In: IEEE International Conference on Robotics and Automation (ICRA) (2020)

[4] Paulo Gustavo Quinan, Issa Traore, Ujwal Reddy Ghondi, Isaac Woungang, "Unsupervised Anomaly Detection using a new Knowledge Graph Model for Network Activity and Events," 4th International Conference on Machine Learning for Networking (MLN), 2021, Paris, France.

[5] Issa Traore, Paulo Gustavo Quinan, Waleed Yousef, "The Activity and Event Network (AEN) Model: Graph Elements and Construction," Technical report, ISOT lab, ECE Department, University of Victoria, January 2020.

[6] Paulo Gustavo Quinan, Issa Traoré & Isaac Woungang, "Activity and Event Network Graph and Application to Cyber-Physical Security" Part of the Engineering Cyber-Physical Systems and Critical Infrastructures book series (ECPSCI, Vol. 2), 2022.

[7] Hadi Shiravi, Ali Shiravi, and Ali A. Ghorbani, “A Survey of Visualization Systems for Network Security”, IEEE Transactions on Visualization and Computer Graphics, Vol. 18, No. 8, Aug. 2012.

[8] Adrián Campazas-Vega, Ignacio Samuel Crespo-Martínez, Ángel Manuel Guerrero-igueras, Claudia Álvarez-Aparicio & Vicente Matellán , Analysis of NetFlow Features’ Importance in Malicious Network Traffic Detection, 14th International Conference on Computational Intelligence in Security for Information Systems and 12th International Conference on European Transnational Educational (CISIS 2021 and ICEUTE 2021).

[9] Zhou Zhimin, Chen Zhongwen, Zhou Ti echeng, Guan Xiaohui, The Study on Network Intrusion Detection System of Snort, Department of Computer Science Zhejiang Water Conservancy And Hydropoeer College Hangzhou, China, 2010.

[10] JX Xu. The analysis of intrusion detection software. Journal of Southwest Guizhou Teachers College for Nationalities. [J]. 2008.

[11] Vern Paxson, “Bro: A System for Detecting Network Intruders in Real-Time”, In: Network Research Group, Lawrence Berkeley National Laboratory, Berkeley, CA, 1999.

[12] Zeek scripting tutorial, available at https://zeek.org/ (Last visited Oct. 9, 2023).

[13] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, Asaf Shabtai, Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection, Cornel University, 2018.

[14] Robert Morawa, Tom Horak, Ulrike Kister, Annett Mitschick, Raimund Dachselt, Combining Timeline and Graph Visualization, Proceedings of the Ninth ACM International Conference on Interactive Tabletops and Surfaces, November 2014.

[15] Oracle. Oracle Labs Pgx: Parallel Graph Analytics, https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html (Last visited Oct. 9, 2023).

[16] "Abhishek Jaiswal", "JSON: Introduction, Benefits, Applications, and Drawbacks", available at https://www.turing.com/kb/what-is-json (Last visited Oct. 9, 2023).

[17] Google, Pagination, "incremental page loading, and their impact on Google Search", https://developers.google.com/search/docs/specialty/ecommerce/pagination-and-incremental-page-loading (Last visited Oct. 9, 2023)

[18] "Danielle Ellis", "Creating a Range Slider in HTML + CSS", available at https://blog.hubspot.com/website/html-slider, published 2023, (Last visited Oct 9, 2023).

[19] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani, "Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization," In Canadian Institute for Cybersecurity (CIC), University of New Brunswick (UNB), Canada, 2018.