Tensor Based Monitoring of Large-Scale Network Traffic

A Thesis Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Gerald Liso
December 2018

Copyright © 2018 by Gerald Liso
All rights reserved.

ACKNOWLEDGEMENTS

I would first and foremost like to thank Dr. Jens Gregor for not only being an outstanding mentor for my graduate and undergraduate degrees, but for also instilling in me the foundations of computer science in the Data Structures and Algorithms courses. Due to his great instruction, I found a passion for computer science that led me to achieve higher learning in my Masters degree.

I also would to thank my master's committee members, Dr. Michael Thomason and Dr. Audris Mockus. Dr. Thomason provided exceptional guidance for much of the mathematical concepts presented in this thesis, and without him, much of this work would not be possible. Dr. Mockus provided great instruction for developing analytical models and visualizations through his Fundamentals of Digital Archaeology course. That course gave me a passion for data science and allowed me to question the endless possibilities of the field.

Finally, I would like to thank Angel Kodituwakku for serving as a very good source of information to make the work in this thesis possible as well as ensuring the project's proper continuation and growth. He worked hard and long hours that provided me with the means to complete my research.

The work presented in this thesis was supported by the National Science Foundation under Grant No. IRNC-1450959. Any opinions, findings, and conclusions or recommendations expressed are those of the author and do not necessarily reflect the views of the National Science Foundation.

ABSTRACT

Network monitoring systems are important for network operators to easily analyze behavioral trends in flow data. As networks become larger and more complex, the data becomes more complex with increased size and more variables. This increase in dimensionality lends itself to tensor-based analysis of network data as tensors are arbitrarily sized multi-dimensional objects. Tensor-based network monitoring methods have been explored in recent years through work at Carnegie Mellon University through their algorithm DenseAlert. DenseAlert identifies events anomalous events in tensors through quick detection of dense sub-tensors in positive-valued tensors. However, from experimentation, DenseAlert fails on larger datasets. Drawing from RED Alert, we developed an algorithm called RED Alert that uses recursive filtering and expansion to handle anomaly detection in large tensors of positive and negative valued data. This is done through the use of network parameters that are structured in a hierarchical fashion. That is, network traffic is first modeled at low granular data (e.g. host country), and events detected as anomalous in lower spaces are tracked down to higher granular data (e.g. host IP). The tensors are built on-the-fly in streaming data, filtering data to only consider the parameters deemed anomalous in previous granularity levels. RED Alert is showcased on two network monitoring examples, packet loss detection and botnet detection, comparing results to DenseAlert. In both cases, RED Alert was able to detect suspicious events and identify the root cause of the behavior from a sole IP. RED Alert was developed as part of a greater project, InSight2, that provides several different network monitoring dashboards to aid network operators. This required additional development of a tensor library that worked in the context of InSight2 as well as the development of a dashboard that could run the algorithm and display the results in meaningful ways.

TABLE OF CONTENTS

1. Introduction 1
1.1 Overview of Network Monitoring 1
1.2 Tensor Definitions 2
1.3 Problem Motivation 6
1.3.1 Performance Monitoring 7
1.3.2 Security Monitoring 7
2. Network Anomaly Detection Through DenseAlert 11
2.1 DenseAlert Overview 11
2.1.1 DenseAlert Algorithm 11
2.1.2 DenseAlert Analysis 13
2.2 DenseAlert Results 14
2.2.1 Packet Loss Detection 15
2.2.2 Botnet Detection 17
3. REDAlert: Recursive Event Detection 20
3.1 Algorithm Overview and Analysis 20
3.1.1 Granularity and Filtering 20
3.1.2 RED Alert Algorithm 22
3.1.3 Time and Space Complexity 24
3.2 RED Alert Results 26
3.2.1 Packet Loss Detection 26
3.2.2 Botnet Detection 28
3.2.3 Choosing Threshold Values 30
3.2.4 Effects of Time Window 31
3.2.5 Robustness 33
3.2.6 Using Energy as Criteria 34
4. Modeling Results with InSight2 39
4.1 History and Motivation of InSight2 39
4.1.1 GLORIAD and InSight 39

4.1.2 From InSight to InSight2 40
4.2 Architecture Overview 41
4.2.1 Enrichment Module 41
4.2.2 Updater and Summarizer Modules 43
4.2.3 Plug-ins 44
4.2.4 Dashboards 44
4.2.5 InSight2 Workflow 45
4.3 Tensor InSight2 Framework 46
4.3.1 Overview of Python Tensor Libraries 46
4.3.2 Modifications to scikit-tensor 49
4.3.3 Interaction with Elasticsearch 50
4.4 RED Alert Plug-in within InSight2 52
4.4.1 Frontend 52
4.4.2 Backend 53
5. Conclusions 56
References 58
Vita 61

LIST OF FIGURES

Figure 1.1: Tensor Slices 4
Figure 1.2: Source Traffic Heatmap, Packet Loss 8
Figure 1.3: Destination Traffic Heatmap, Packet Loss 8
Figure 1.4: Total Packets Lost for Large Network 8
Figure 1.5: Source Traffic Heatmap, Number of Connections 10
Figure 1.6: Destination Traffic Heatmap, Number of Connections 10
Figure 1.7: Total Number of Botnet Connections for Large Network 10
Figure 2.1: DenseAlert Packet Loss, Country Data 16
Figure 2.2: DenseAlert Packet Loss, City Data 16
Figure 2.3: DenseAlert, Packet Loss, Institution Data 16
Figure 2.4: DenseAlert Packet Loss, IP Data 17
Figure 2.5: DenseAlert Botnet, Country Data 18
Figure 2.6: DenseAlert Botnet, City Data 18
Figure 2.7: DenseAlert Botnet, Institution Data 18
Figure 2.8: DenseAlert Botnet, IP Data 19
Figure 3.1: Default Granularity Levels 21
Figure 3.2: RED Alert Algorithm 23
Figure 3.3: Finding Maximum Change 23
Figure 3.4: RED Alert Packet Loss, Country Granularity 27
Figure 3.5: RED Alert Packet Loss, City Granularity 27
Figure 3.6: RED Alert Packet Loss, Institution Granularity 27
Figure 3.7: RED Alert Packet Loss, IP Granularity 28
Figure 3.8: RED Alert Botnet, Country Granularity 29
Figure 3.9: RED Alert Botnet, City Granularity 29
Figure 3.10: RED Alert Botnet, Institution Granularity 29
Figure 3.11: RED Alert Botnet, IP Granularity 30
Figure 3.12: 10-second Time Window, Packet Loss Detection 32
Figure 3.13: 5-minute Time Window, Packet Loss Detection 32
Figure 3.14: 10-second Time Window, Botnet Detection 32
Figure 3.15: 5-minute Time Window, Botnet Detection 33
Figure 3.16: RED Alert Packet Loss using Energy, IP Granularity 36
Figure 3.17: RED Alert Packet Loss using Energy, City Granularity 36
Figure 3.18: RED Alert Packet Loss using Energy, Institution Granularity 36
Figure 3.19: RED Alert Packet Loss using Energy, IP Granularity 37
Figure 3.20: RED Alert Botnet using Energy, Country Granularity 37
Figure 3.21: RED Alert Botnet using Energy, City Granularity 37
Figure 3.22: RED Alert Botnet using Energy, Institution Granularity 38

Figure 3.23: RED Alert Botnet using Energy, IP Granularity 38
Figure 4.1: InSight2 Architecture Workflow 46
Figure 4.2: COO Representation 49
Figure 4.3: InSight2 Sample Tensor 51
Figure 4.4: RED Alert Plug-in 54

1. INTRODUCTION

Parts of this thesis have been submitted to *ACM Transactions on Knowledge Discovery from Data* as of October 2018. I served as the author of this paper, additionally conducting the literature survey, developing the algorithm, portraying and analyzing the results, and writing the journal paper. Other authors, Angel Kodituwakku, Jens Gregor, and Michael Thomason served as sources of guidance and advisers to help locate other resources that were beneficial to the completion of the paper.

1.1 Overview of Network Monitoring

Network flow data is one source of data that can be used to facilitate real-time network performance monitoring and detection of malicious traffic [1, 2]. Network flow data is a stream of records that provide further metadata about a transaction in the network, including labels of actors in a transaction and performance metrics of each transaction. Due to the amount of flow data produced by even a moderately active network, automated detection of interesting events is needed to support the network administrators carry out their work [3]. The large number of parameters associated with each flow, such as the source and destination host IP addresses, the number of packets transmitted, lost and retransmitted, and the protocol and port numbers to mention a few, furthermore implies that the analysis might benefit from the use of multi-dimensional data representations.

Tensors offer a way to model multi-dimensional data effectively; tensors of order n are n-dimensional representations of data describing linear relationships [4]. Various algorithms have been developed to analyze patterns within tensors. For example, generalized singular value decomposition (GSVD) is used in genomics for classification of genes [5]. GSVD has been extended to tensors through the so-called higher-order GSVD (HOGSVD) [6]. Tensors have also been used in network analysis for anomaly detection of network packet activity using Canonical Polyadic (CP) decomposition [7]. Both of these methods project the data in a new space. In order to relate events to specific parameters and fields, it may be preferable that the analysis be carried out in the original data space.

DenseAlert [8] maintains a tensor model of the raw data. Events characterized by an increase in activity are detected by identifying dense sub-tensors. The algorithm uses sophisticated methods to carry out the needed computations quickly. The complex nature of the code makes the algorithm relatively difficult to implement. The Java implementation provided by the authors moreover builds on dedicated tensor support code, the efficiency of which may not be matched by a third-party. Combined, this makes porting the algorithm difficult. In addition, DenseAlert requires that the data be positive valued which could be a limitation for some applications. DenseAlert is analyzed and tested in Chapter 2 of this thesis.

The primary focus of this thesis is the introduction of an algorithm called RED Alert that uses recursive filtering and expansion to create a hierarchy of sub-tensors that represent the data at different granularity levels. The idea is to maintain a coarse-level tensor model that can quickly be updated when new data becomes available. When the activity being monitored exceeds a user-defined threshold, relevant sub-tensors are automatically expanded until an alarm can be raised which provides the most detailed information possible. Granularity is a refinement to the tensor dimensions. For example, low granularity data might be host country with high granularity data being the explicit host IP address. This is explained in more detail in Chapter 3 with some showcase examples.

The data used in this thesis comes from a large research and education (R&E) network called GLORIAD. The Global Ring Network for Advanced Applications Development (GLORIAD) was an R&E network aiming to provide a network infrastructure to connect institutes and universities at a global scale [9]. GLORIAD's successes can be seen through its ability to connect over 15 million endpoints and support from the NSF and universities worldwide. GLORIAD led to the development of a network performance tool called InSight2, a platform for which the work in this thesis was developed. The inner workings of InSight2 and the tensor processing infrastructure developed for InSight2 are described in Chapter 4. Concluding remarks are then provided in Chapter 5.

1.2 Tensor Definitions

Tensors are generalized *n*-dimensional objects that can provide a means for in-depth analysis of multiple parameters. Tensors offer a way to easily model multi-

dimensional data as they can contain an arbitrary number of dimensions. In this section, the basic tensor definitions and concepts that will be used in the remainder of this thesis are defined.

Modes: The dimensions of a tensor are referred to as modes. For example, a vector is a tensor with 1 mode, and a matrix is a tensor with 2 modes. Tensors can have any number of modes since the dimensionality of a tensor is unrestricted.

Furthermore, each mode can be indexed as one would expect. In a tensor with 2 modes, element (*i,* j*)* corresponds to the value stored in the i^{th} row and the j^{th} column in the tensor. In this thesis, each mode represents a parameter in a network flow record.

Sub-tensors: Tensors can be divided into subsets of a tensor by indexing a subset of the set of viable vertices in the original tensor. For example, suppose T is a tensor with 2 modes (i.e., a matrix) and dimensionality 4 × 4. Let S be the set {(4,4), (4,1), (1,4), (1,1)}. Then, T_S is the sub-tensor of T consisting of the first and fourth rows and columns of T.

Slices: Another method of dividing tensors (and sub-tensors) is a method called slicing. Slices are formed by holding one mode constant in the tensor, representing the various layers of a tensor. For example, consider the same tensor T from the previous example. A slice across mode 1 would represent a row. A slice across mode 2 would represent a column. Since T has dimensions 4 × 4, each mode has 4 possible slices. Figure 1.1 visualizes slices in a tensor of 3 modes [10]. These tensors can be sliced horizontally, laterally, or frontally as shown in the figure.

Tensor Subtraction: Both tensors and sub-tensors can be subtracted from one another when the objects contain the same dimensionality. This is useful for tracking changes in streaming tensors as discussed in the next chapter. This subtraction is done in an index-wise manner. This means that for two tensors of mode 3, T_1 and T_2, the difference $T_1 - T_2$ would be the difference for each element at (*i*, *j*, *k*) in T_1 and T_2 for all *i*, *j*, *k* $\in$ the dimension set for T_1 and T_2.

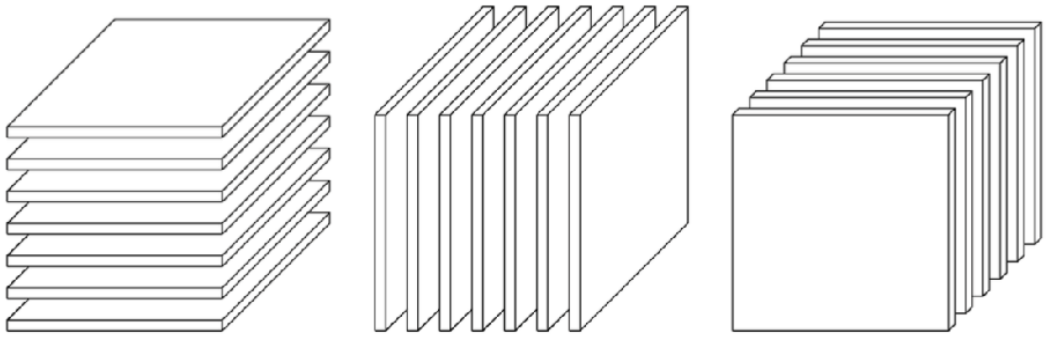


Figure 1.1: Tensor Slices, Source: [10]

Density: Tensors, sub-tensors, and slices have a density that can be calculated. Density can be interpreted as a quantified measure of influence created by the tensor, sub-tensor or slice. In the next chapter about the DenseAlert algorithm, density of a slice is used as the primary measurement. Density of a tensor T can be defined as

$$\frac{\sum_{(i.j,k)} T(i,j,k)}{|T|},$$

where (i, j, k) are valid indices with T, and $|T|$ is the number of elements in the tensor. This equation holds when T is a slice or sub-tensor as well. The summation would sum valid entries where (i, j, k) are valid indices for the slice or sub-tensor.

Change: Total change of a sub-tensor or a slice is defined as the sum of the element-wise absolute valued differences of two such entities. For two tensors, T_1 and T_2, total change is defined as

$$\sum_{(i,j,k)} |T_1(i,j,k) - T_2(i,j,k)|,$$

for all valid indices (i, j, k) in the tensors. Total change serves as the main metric used in the RED Alert algorithm described in Chapter 3. The slice in a tensor with the most change is detected in RED Alert, indicating anomalous activity.

Total change is inspired from the concept of total variation in Calculus. Total change is described by the equation

$$\int_a^b |f'(x)|dx,$$

which calculates the arc length of a differentiable function, *f(x)* for a given interval [*a*, *b*]. This equation can be represented as

$$\sum_{i=a}^{b} |f(i) - f(i-1)|,$$

which is similar to the definition of total change defined above.

Energy: Energy can be defined as the sum of the squares of the norms for a given sequence, normalized by the number of elements in this sequence [11]. Energy has been used to estimate the number of principal components needed through thresholding [11], but we use energy as an anomaly detection measure. In the context of our tensors, define energy can be defined as the squared Frobenius norm [12] for a given tensor, sub-tensor, or slice, divided by the number of non-zero entries. That is, for indices (i, j, k) in tensor T , we calculate the energy as

$$\frac{\sum_{(i,j,k)} |T(i,j,k)|^2}{|T|},$$

where $|T|$ is the number of non-zero components. In the context of slices or sub-tensors, the equation holds, but instead the equation only iterates on indices (i, j, k) for the slice or sub-tensor. Energy is used as a supplemental measurement for the anomaly detections in RED Alert. Just as the slice with maximum change is detected, the slice with the highest energy in the tensor representing total change can also be detected to indicate anomalous activity.

Total change and energy can be considered as the L1-norm and L2-norm for the change tensor. Total change is the L1-norm as it considers the values of the tensor in linear space, and energy is the L2-norm as it considers the tensor in

quadratic space. The Frobenius norm is equivalent to the Euclidean norm typically used for L2-norms [12].

Streaming Tensors: Since network flow data can be viewed as a continuous stream of data, in this thesis, dynamic streaming tensors are used instead of static tensors of fixed dimensionality. Streaming tensors are a sequence of tensors aggregated into a single tensor for a fixed time. This allows for data to only be considered for a specific time window, and old data can be discarded as new tensor streams come in. This concept will be touched on throughout this thesis.

1.3 Problem Motivation

We were given access to several terabytes of GLORIAD's network flow data to develop network analysis tools. Using Argus [13] as a network flow aggregator, over 40 different fields describing the network flow data could be extracted. Examples of such fields include bytes per transaction, number of packets per transaction, and packets lost. Additionally, online databases were used to gather additional metadata about geolocation of the flow data. This is useful since GLORIAD was a global network, so understanding how the network functioned in different regions is essential for network monitoring. Varying levels of metadata were gathered about each transaction from these online databases. This includes country, city, and institution name for each IP discovered from the flow data. This primary focus of this thesis is exploring how performance at these varying levels of metadata, referred to as granularity levels, can be tracked and explored to find anomalous trends within the data.

Although network monitoring can be conducted from varying perspectives, the primary focus of this thesis is in the context of performance monitoring through packet loss detection and security monitoring through botnet activity detection. These two problems are described in this section.

1.3.1 Performance Monitoring

One important aspect of network performance is minimizing the number of packets lost per transaction. With each packet lost, network users either must re-fetch the request to obtain the lost data or suffice with incomplete data. Large amounts of packet loss indicate potential issues in network infrastructure that warrant investigation. Thus, one focus of this thesis is detection times of high packet loss within GLORIAD. This can be detected by building tensors of the number of packets lost between a given source and destination connection.

Figures 1.2 and 1.3 provide heat maps of packet losses during an 11-hour time period for a select subset of source and destination countries. Dark cells indicate low losses while bright cells indicate large amounts of packet loss. Figure 1.4 plots the corresponding percent of packets lost overall. The goal is to detect the high loss events and determine whether they can be attributed to many hosts, in which case they may indicate a general network problem, or if they are specific to a small number of identifiable hosts, in which case the problem likely has to do with the specific hosts. In Figure 1.4, the sudden increase in packet loss around 14:30 is one of primary interest explored in this thesis. Throughout the thesis, the events marked in green circles in the figures represent anomalous events that could be detected.

1.3.2 Security Monitoring

Botnets consist of a collection of compromised hosts which are used to carry out network attacks including but not limited to stealing data, sending spam, and performing distributed denial-of-service (DDoS) [14]. As these attacks can greatly affect the well-being of a network and its hosts (not to mention the users), it is important to discover when botnets are active. Because of this, another focus of this thesis considers analyzing network flow data from a security perspective to detect botnets in large networks.

Typical botnet behavior includes sudden broadcasting of connections after being dormant for some time. This lends itself to detection by considering the number of connections from a given source to a destination in a specific time window. This way, a sudden increase in connections can be detected as the values in this tensor will change drastically in a short period of time.

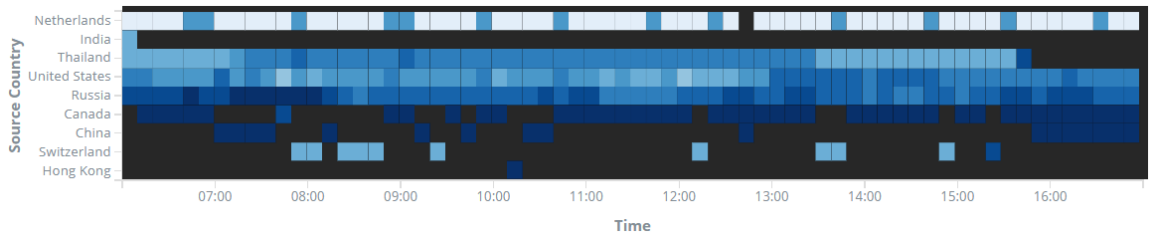


Figure 1.2: Source Traffic Heatmap, Packet Loss

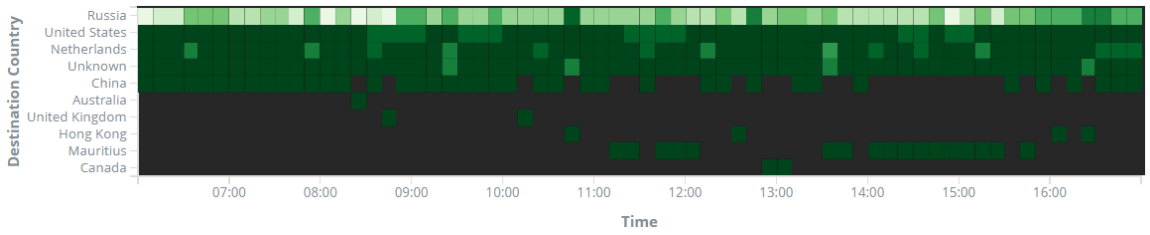


Figure 1.3: Destination Traffic Heatmap, Packet Loss

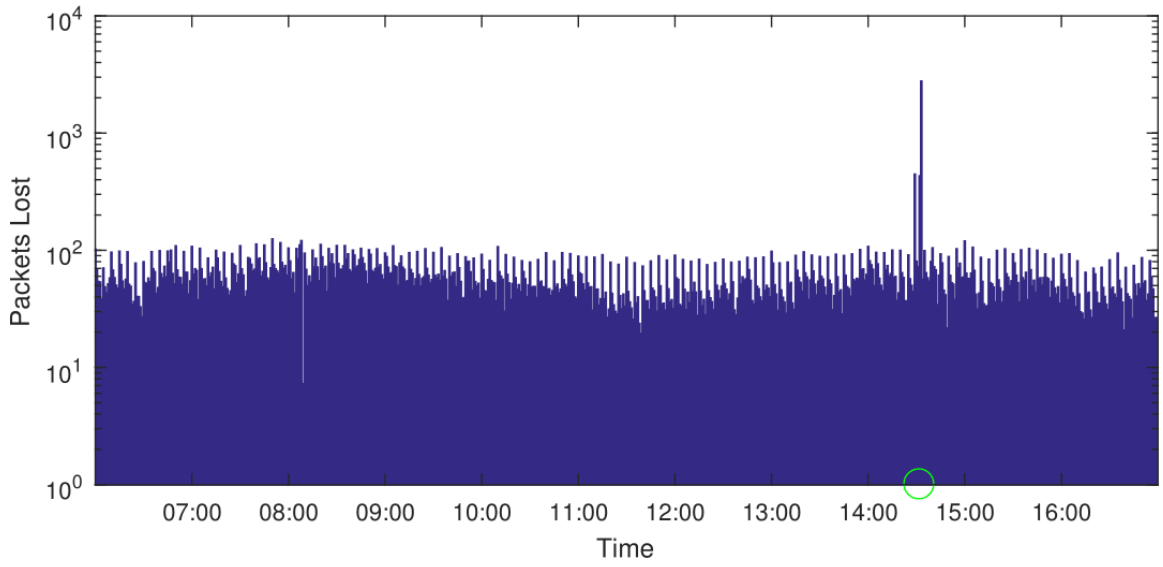


Figure 1.4: Total Packets Lost for Large Network

Figures 1.5 and 1.6 provide heat maps of the number of connections made per source and destination during an 11-hour period for a select subset of countries. Dark cells indicate few connections while bright cells indicate many connections. Figure 1.7 plots the number of known botnet hosts active during the same time period. The term “known botnet host” refers to IP addresses labeled as compromised in the Zeus [15], Palevo [16], and Feodo [17] botnet databases. As seen in the figures, the large increase in botnet activity around 9:00 corresponds to the large increase in network activity from Russia to the Netherlands.

In the next two chapters of this thesis, both DenseAlert and RED Alert process these two datasets of packet loss and number of connections to attempt to detect the performance and security anomalies. These algorithms use completely different methodologies and evaluation metrics, providing varying results. Again, the primary goal in these detections is the ability to be able to detect the anomalous behavior at any level of metadata (country, city, institution, or IP). That is, the spikes seen in Figures 1.4 and 1.7 should be detected independent of the labels given in the tensor. DenseAlert fails to recognize spikes using IP labels, which led to the development of RED Alert, an algorithm that can detect anomalous behavior regardless of level of metadata through recursive filtering and expansion.

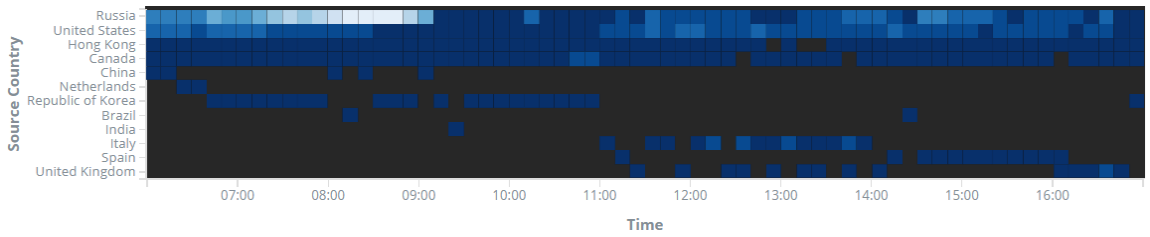


Figure 1.5: Source Traffic Heatmap, Number of Connections

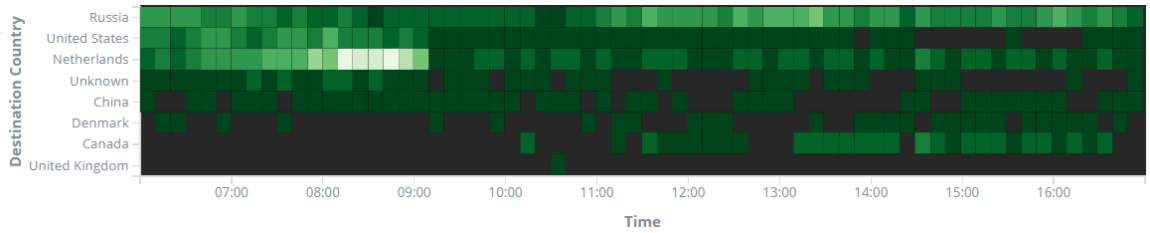


Figure 1.6: Destination Traffic Heatmap, Number of Connections

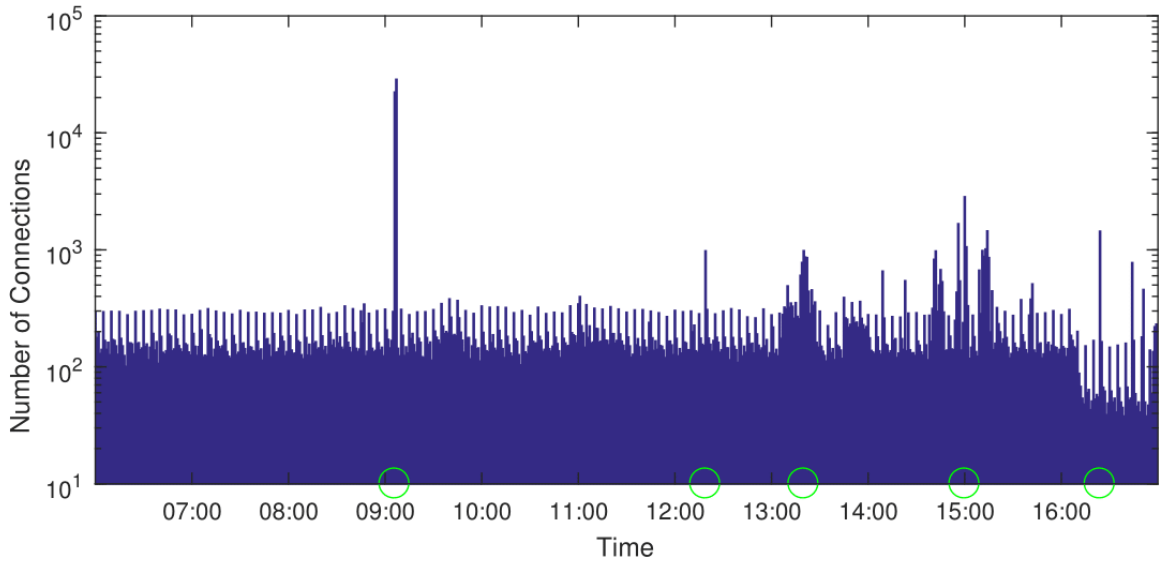


Figure 1.7: Total Number of Botnet Connections for Large Network

2. NETWORK ANOMALY DETECTION THROUGH DENSEALERT

One recently developed algorithm that showcases botnet detection in network data using tensors is DenseAlert [8]. This algorithm uses the concept of density to maintain the densest sub-tensor of a streaming tensor to detect sudden changes, reporting them as anomalous. The DenseAlert paper showcases the algorithm on a number of anomaly detection uses, including Yelp spam reviews, Wikipedia edit wars, and the botnet detection. Thus, it would seem that DenseAlert would be a suitable algorithm to use for our analyses of large-scale network traffic data. However, some problems arose when using DenseAlert with larger tensors. This chapter describes DenseAlert's algorithm and demonstrates the shortcomings of the algorithm for GLORIAD data in the context of the performance and security monitoring problems.

2.1 DenseAlert Overview

DenseAlert is an algorithm that claims to be "fast and any time," "provably accurate," and "effective" [8]. This is done through the maintenance of a tensor consisting of entries that represent the increment or decrement values in a tensor for a given time window. This means that DenseAlert does not actually maintain an entire tensor; rather it tracks multi-dimensional streaming data for increments or decrements to values in the tensor. The idea is that a sudden change in the maintained dense sub-tensor should indicate anomalous activity. This section describes this algorithm in detail and provides some analyses of the algorithm. Although DenseAlert works for static and streaming data, we focused on the streaming implementation of the algorithm due to the use of network flow data.

2.1.1 DenseAlert Algorithm

DenseAlert's main goal is to detect suddenly appearing dense sub-tensors within streaming or static tensors, which indicates anomalous activity. This is done by projecting a sequence of data streams into a tensor space and performing a series of re-orderings and calculations to maintain the densest block. The algorithm does not actually calculate the exact densest sub-tensor as that is

computationally expensive, taking high powers of polynomial time [18]. Instead, the algorithm estimates the densest sub-tensor with provable confidence to quell the large polynomial time.

DenseAlert uses a variety of concepts and notations throughout its description of the algorithm. The most important component of DenseAlert is their idea of “D-Ordering.” This idea is what provides significant speedup to DenseAlert compared to its competitors. A D-Ordering is obtained by repeatedly re-ordering slice indices so that the slice with the minimum sum is chosen first. That is, when a slice *s* is chosen for the D-Ordering, the slice index is added to the D-Ordering, and all elements from *s* are removed from the tensor. The process is then repeated until all slices appear in the D-Ordering. The paper claims that using D-Ordering reduces the search time for finding the densest sub-tensor.

The DenseAlert algorithm works as follows. Every ΔT time ticks, the change for a given tensor is calculated by finding the difference between the tensor of the previous time window and the current time window. For each value in the tensor that changed, DenseAlert performs re-ordering and updating based on whether there was an increase or decrease in the value. The first iteration of the algorithm compares the current tensor against the zero tensor of the same size, meaning the first iteration will always be the case of increment.

In the case of an increment in a tensor value, DenseAlert first finds a region that needs to be re-ordered by the definition of D-Ordering. Anything outside of this region can be left alone as the paper guarantees that the D-Ordering will be maintained after the detected region is updated. Once this region is found, the reordering of the calculated region is conducted to maintain the D-Ordering, and the slice sum of this region is saved. After the re-ordering, the dense sub-tensor needs to be updated. To do this, first the slice sum calculated by the D-Ordering is compared to the densest sub-tensor density from the previous tensor. If the maximum slice sum is less than the sub-tensor density, the paper proves that the densest sub-tensor is guaranteed to not need updating. Otherwise, DenseAlert recalculates the dense sub-tensor, updating it if it has changed. After this, DenseAlert is done with the re-ordering and updating. However, the algorithm then schedules a decrement to this tensor value at the next ΔT time. This is done to undo the increment to the value experienced here and return the tensor back to its prior state in case of sudden and sharp increases.

In the case of a decrement in a tensor value, the algorithm again finds a region that needs to be re-ordered and re-orders the region to meet the D-Ordering definition. This region is determined in a slightly different fashion than in the case of an increment, but the idea is the same – re-order the region so that D-Ordering is maintained. Again, after the re-order process is completed, DenseAlert checks to see if the densest sub-tensor needs to be updated. Instead of comparing the maximum slice sum as done in the case of an increment, DenseAlert checks if the decremented tensor value is in the densest sub-tensor. If the value is a part of this region, the densest sub-tensor is updated if it has changed. If the tensor value is not in the densest sub-tensor, the authors guarantee that the sub-tensor would be unaffected and, thus, does not need to be updated. Unlike in the case of increment, DenseAlert does not schedule any event in the case of decrements.

In both the case of the increment and decrement, the density of the densest sub-tensor is returned. As seen in the results section, spikes in the density of this region typically correspond to anomalies in the data.

2.1.2 DenseAlert Analysis

DenseAlert has time complexity $O(S + RlogR)$ plus some relatively small time for other operations such as calculating slice sums and searching for slices. In this, S is the number of slices in the tensor, and R is the number of slices in the re-ordered region. The $O(S)$ time is due to the search for slices forming the dense sub-tensor, and the $O(RlogR)$ time is from the re-ordering of the tensor to meet the requirements of D-Ordering. DenseAlert uses a Fibonnaci heap to search for the minimum slices in the re-ordering leading to this linear logarithmic time complexity. Otherwise, the search would be exponential as the search space requires updating as each slice is re-ordered. Through plots shown in the DenseAlert paper, the algorithm is unarguably much faster than its competitors.

The algorithm described previously is a summary of DenseAlert, and the inner workings of the algorithm are complicated and require a detailed code base. Their tensor implementation alone consists of over 600 lines of code, and the actual DenseAlert algorithm requires over 1700 lines of code. The code uses sparse tensor implementations to lead to a small space complexity that is linearly proportional to the number of non-zero values stored in the tensor. That is, for N

non-zero values in the tensor, the space complexity is *O(N)*. This makes understanding and porting the algorithm to other frameworks cumbersome for those who cannot work in the Java framework provided by the authors and do not have the resources to create such a detailed code base for their own implementation. Without a dedicated code base, implementing re-orders and searching of large multi-dimensional objects is costly.

The paper showcases DenseAlert on a variety of datasets. These datasets include Yelp, Android, and Yahoo reviews for detecting fake reviews, Wikipedia edit history for detecting edit wars, bots or vandals, and social networks for spam detection. The paper also shows successful detection of botnets in TCP dump data where the dimensions were source IP × destination IP × timestamp, with values within the tensor being the number of connections. This would indicate that DenseAlert would be a suitable algorithm to use for our network data. In the next section, some shortcomings of DenseAlert are shown in the context of network data.

2.2 DenseAlert Results

In network anomaly detection, it is important to discover the root cause of the anomaly. Network operators rely on detailed information properly diagnose and solve the root cause of an issue. Since flow data has varying levels of granularity, it is important to be able to determine the cause of anomalies regardless of the level of network data being observed. For example, a time of large network performance issues should be able to be determined from a coarse perspective of country to country traffic and at a fine perspective of host IP to host IP traffic. This way, a network operator can understand if an issue is widespread across a region or fault for a specific IP. The DenseAlert algorithm is stated to work for large datasets, but in some cases DenseAlert fails to detect any anomalies and returns unhelpful and noisy results. This section shows the results of DenseAlert when attempting to process datasets of varying sizes, demonstrating some potential issues when using DenseAlert. To do this, minor modifications were made to the DenseAlert example code (https://github.com/kijungs/densealert) in order to work with our data. In both examples, a time window of 1 minute was used.

2.2.1 Packet Loss Detection

Although DenseAlert's network monitoring example focused on botnet detection, the algorithm could be used on any type of positive-valued data. Since number of packets lost consists of positive values, DenseAlert should be able to detect sudden increases packets lost. The GLORIAD data was aggregated into a CSV file representing a tensor to be read by DenseAlert. The tensor itself consisted of dimensions source information × destination information × time (in second increments) where the source information and destination information varied in granularity level (country, city, institution, or IP). The values within the tensor were the number of packets lost for that connection, summed by seconds. DenseAlert was run on tensors from the four different granularity levels to see if it could detect the large increase in packet loss regardless of tensor size and labels.

With reference to Figures 2.1 – 2.4, DenseAlert is able to detect the major spike in packet loss around 14:45, but as the granularity levels become finer, the reported densities slowly begin to have more variation until the algorithm completely fails at the IP granularity level. The country level detection (Figure 2.1) provides the most descriptive results where the large spike is clearly defined with relatively consistent densities otherwise. At the city level (Figure 2.2), the large spike is still seen, but the algorithm reports more variation in the densities, notably between 10:30 and 12:30 as well as 15:00 to 17:00. This seems to indicate anomalous activity with the sudden and frequent changes in the densities, but according to the ground truth, the number of packets lost at this time is rather constant. A similar variation in the densities can be seen at the city level (Figure 2.3). Finally, at the IP level (Figure 2.4), the algorithm falls apart and reports noisy densities that provide absolutely no correlation to the data. The densities at this level are also about 5 magnitudes smaller than the densities reported at the country, city, and institution levels. If kept plotting using the same y-axis scale as the first 3 plots, the densities in Figure 2.4 would be unnoticeable due to the size of their magnitudes.

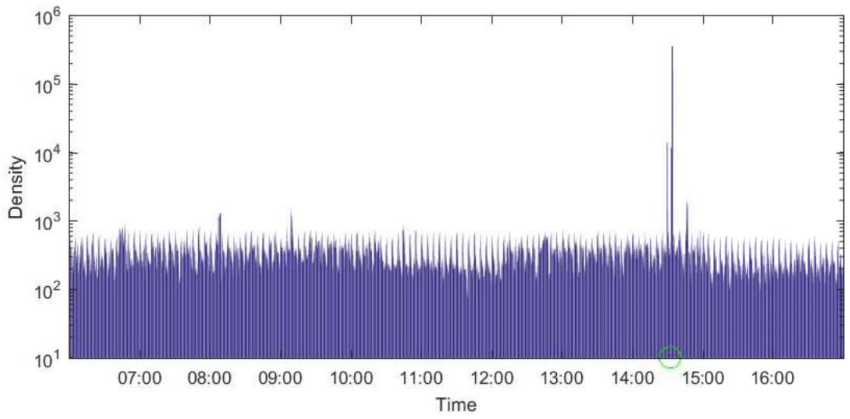


Figure 2.1: DenseAlert Packet Loss, Country Data

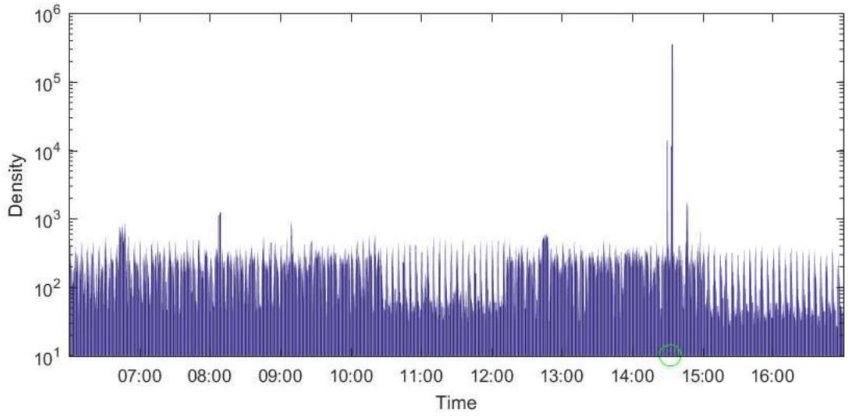


Figure 2.2: DenseAlert Packet Loss, City Data

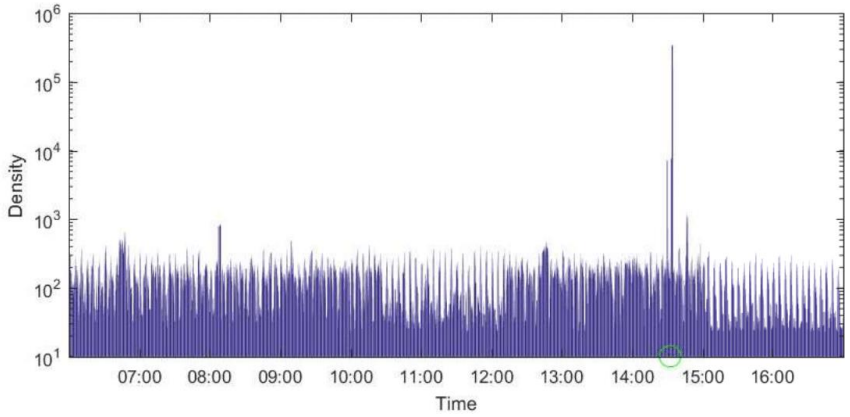


Figure 2.3: DenseAlert, Packet Loss, Institution Data

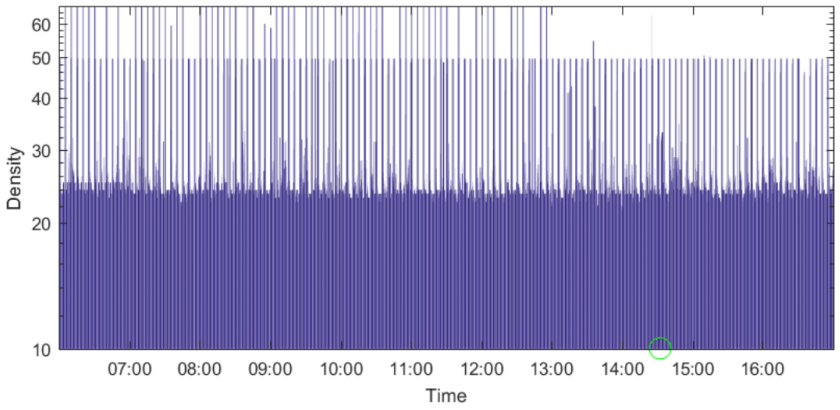


Figure 2.4: DenseAlert Packet Loss, IP Data

2.2.2 Botnet Detection

In its debut paper, DenseAlert is showcased on a botnet detection example. As proposed in the paper, a CSV file containing number of connections in GLORIAD was aggregated and processed by DenseAlert for detection of increases in botnet activity. With reference to Figures 2.5 – 2.8, DenseAlert is again able to detect the botnet activity at the country, city, and institution granularity levels, but fails at the IP granularity level. As seen in the packet loss example, the decline in performance is gradual through the increasing granularity levels. The densities once again slowly become noisier in each subsequent level until nothing useful is returned at the IP granularity level. DenseAlert also only detects the major spike seen around 9:00 but misses the two spikes seen around 13:00 and 15:00.

Nothing in the DenseAlert paper points to why the algorithm would fail on the IP level dataset in both cases. The IP level dataset consists of 4.7 million non-zero entries for the packet loss example and 7.4 million non-zero entries for the botnet example. The DenseAlert paper shows results from datasets containing up to 82.8 million non-zero entries. Similarly, the dimensionalities of the datasets are of sizes DenseAlert should be able to handle. It seems that the size of these datasets leads to the algorithm's failures. Since DenseAlert works on the smaller datasets of lower granularity and slowly performs worse as the datasets become larger and finer, the most reasonable explanation for the decline in performance would be the increases in granularity.

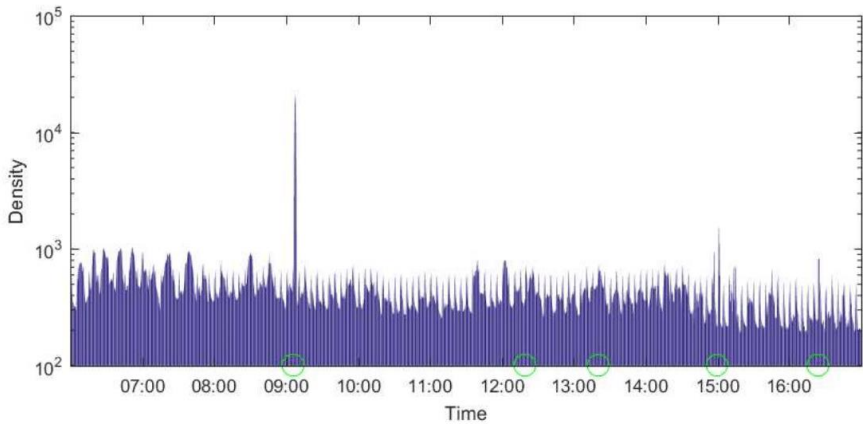


Figure 2.5: DenseAlert Botnet, Country Data

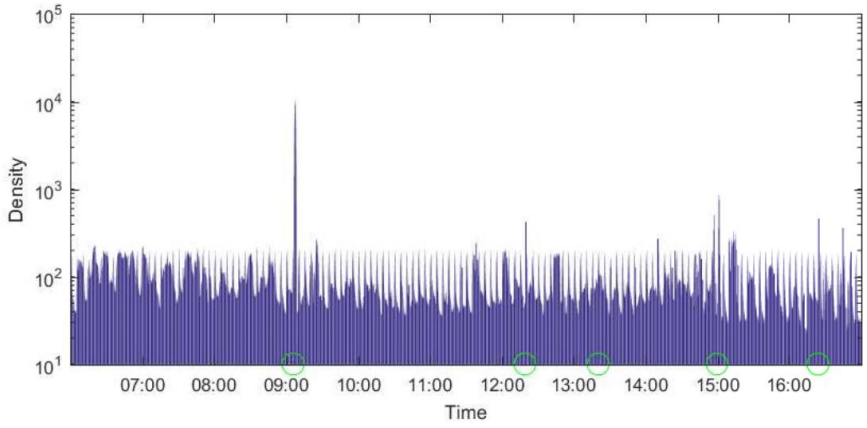


Figure 2.6: DenseAlert Botnet, City Data

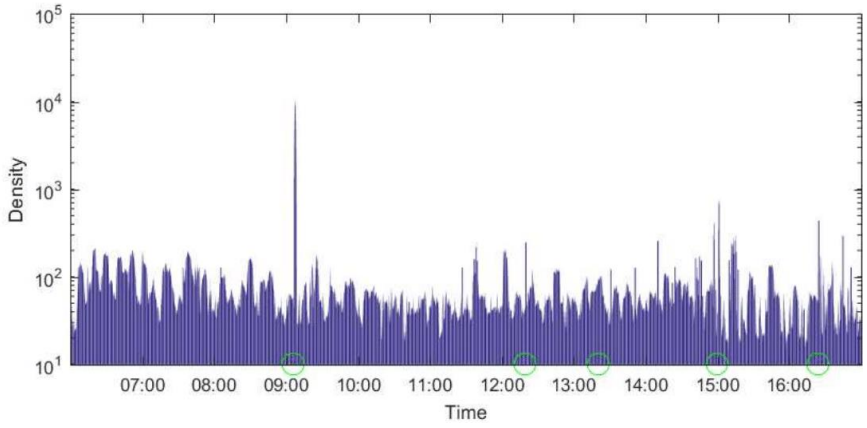


Figure 2.7: DenseAlert Botnet, Institution Data

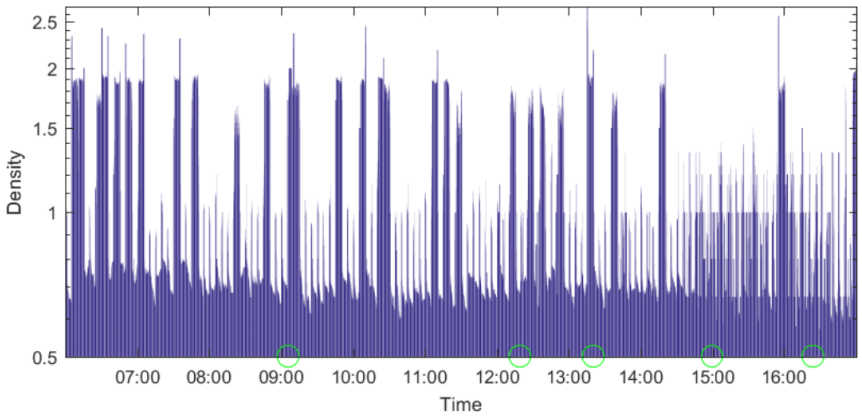


Figure 2.8: DenseAlert Botnet, IP Data

3. REDALERT: RECURSIVE EVENT DETECTION

As seen in the previous chapter, DenseAlert fails to recognize anomalous activity for large datasets using IP address labels. Because of this, we created an algorithm called RED Alert to attempt to improve on the shortcomings of DenseAlert. RED (Recursive Event Detection) Alert uses recursive tensor expansion and exploration to detect and classify anomalous network behavior. The algorithm tracks anomalies detected at a high granularity level down to the lowest granularity level, discarding unimportant data. This chapter introduces RED Alert's main algorithm, analyses of the time and space effects of the algorithm, and some case studies used to showcase RED Alert.

3.1 Algorithm Overview and Analysis

RED Alert finds the most changed slice in a sub-tensor by the total change or energy measures outlined previously. The algorithm works recursively in that when high change spikes are detected, the algorithm finds the most changed slice in the next finer level of granularity after filtering the data. This section explains the primary steps of RED Alert as well as some analyses of the algorithm's runtime and space complexity.

3.1.1 Granularity and Filtering

For the source and destination parameters, different levels of granularity are used to explore the root cause of density spikes. Granularity organizes different levels of metadata in a hierarchical fashion. In our implementation, the lowest granularity level is the host country with a maximum of about 200 fields. The next two levels of granularity are city and institution, respectively with about 10,000 and 35,000 fields. Institution is considered a finer granularity than city since multiple institutions may exist in one city. The finest granularity is the host IP addresses, having a magnitude of over 100 million fields in hours of data. Figure 3.1 shows the four granularity levels used in RED Alert. Additionally, the granularities outlined are not finite. More levels could potentially be included if the labels exist in a dataset. The labels obtained for country, city and institution were

gathered using online databases that map IP addresses to these fields. Fields such as port number that can be included in the tensor data do not undergo the recursive filtering as no varying levels of metadata exists for these types of fields.

To separate fields with the same name (i.e. same city name, different country, or same institution, different location), field names are appended together to as the granularity levels become finer. For example, a city is labeled as “country.city”, and an institution is labeled as “country.city.institution.” This avoids any labeling different places with the same name at a level as the same label. IP address labels are solely the IP address since only one instance of an IP address can exist.

A coarse granularity tensor is, by nature, small. Finding the densest sub-tensor slice(s) therefore takes relatively little time. In an attempt to keep the search time low in the finer granularity tensors, the data is filtered prior to being expanded. Filtering consists of discarding all but the densest sub-tensor slice(s) from further consideration.

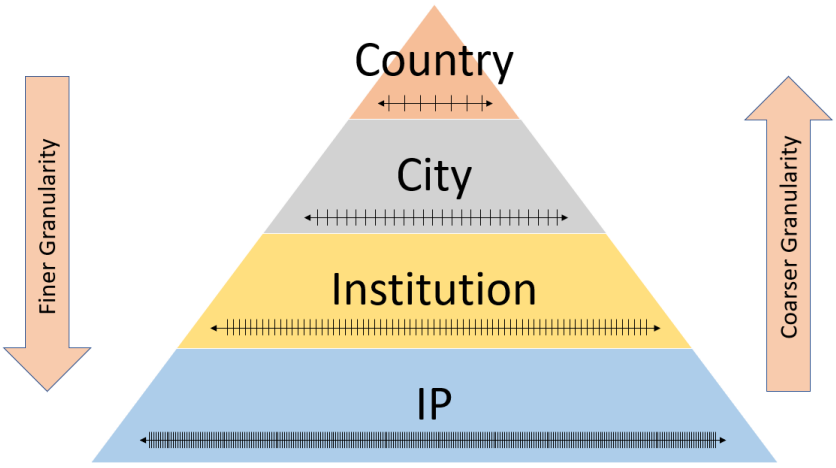


Figure 3.1: Default Granularity Levels

3.1.2 RED Alert Algorithm

Rather than building a detailed tensor model for the data at the fine granularity level ultimately of interest, RED Alert initially models the data at a coarse granularity level, moving to finer granularity spaces as needed. The RED Alert algorithm works as follows. With reference to Figure 3.2, every *N* time ticks, a tensor is created by aggregating the network flow data captured during that time period. This tensor is created for the coarsest granularity level of country. Then, the tensor is compared against the tensor for the previous time period. That is, the difference between the two tensors is computed and passed to the function described in Figure 3.3 that finds and returns a list of (mode, index)-pairs identifying the slices that have a total change value approximately as much as the most changed slice. The search is based on change magnitudes to allow negative change values, which correspond to decreases in network activity, to be detected on par with positive change values, which correspond to increases in network activity.

If the change magnitude exceeds a user-defined threshold, then a recursive search is carried out, during which copies of the original tensors and extracted sub-tensors are filtered and expanded as described in the previous section. Again, (mode, index)-pairs that do not exceed the threshold are ignored whereas the (mode, index)-pairs returned are expanded to consider the next level of granularity. For example, if the maximum (mode, index)-pairs returned were (source country, United States) and (destination country, Canada), then the tensor of the next granularity would be one of city to city traffic consisting only of data from the United States to Canada.

Thresholds vary based on granularity, so that false alarms can be canceled at a finer granularity. That is, as each maximum change value is calculated, it is compared against the threshold at the given granularity level to see if the filtering and expansion should continue or be terminated. When the recursive search reaches the final level and terminates, an alarm is raised an information is shared with the user about the event. A modified version of RED Alert might use the sign of the maximum to raise alarms on positive values and clear alarms on negative values when the traffic returns to normal.

Algorithm 1 Performing Recursive Event Detection

```
procedure RED ALERT
    T_0 = tensor(t_0, ..., t_{N-1})
    for k = N, 2N, ... do
        T_k = tensor(t_k, ..., t_{k+N-1})
        ⟨change, slice_list⟩ = find_most_changed_slices(T_k − T_{k−N})

        while not at finest granularity level do
            if |change| < threshold_granularity then
                break
            end if

            S_k = filter_and_expand(T_k, slice_list)
            S_{k−N} = filter_and_expand(T_{k−1}, slice_list)

            ⟨change, slice_list⟩ = find_most_changed_slices(S_k − S_{k−N})
        end while

        if change > threshold_ALERT then
            ALERT with param_fields(slice_list)
        end if
    end for
end procedure
```

Figure 3.2: RED Alert Algorithm

Algorithm 2 Find the Slice(s) with Most Change in a Given Tensor

```
function FIND_MOST_CHANGED_SLICES(T)
    for each ( ⟨mode, index⟩ ) ∈ T do
        change = sum(T, ⟨mode, index⟩)
        change_list.append(⟨change, mode, index⟩)
    end for

    set max_change = max(change ∈ change_list)

    for each ( ⟨change, mode, index⟩ ) ∈ change_list do
        if change ≥ .99 · max_change then
            append (⟨mode, index⟩) to slice_list
        end if
    end for
return ⟨max_change, slice_list⟩
end function
```

Figure 3.3: Finding Maximum Change

RED Alert differs significantly from DenseAlert. RED Alert finds the tensor slice for which the change is the greatest. DenseAlert, on the other hand, maintains and detects suddenly emerging dense sub-tensors using sophisticated re-ordering techniques. RED Alert can handle positive and negative valued data. DenseAlert requires that the data be positive valued. RED Alert only expands the data to the finest granularity level when need be. DenseAlert stores data and carries out all analysis at the finest granularity level which involves large tensors.

3.1.3 Time and Space Complexity

RED Alert with no optimizations is able to run and process data in real-time. This section provides time and space analyses of RED Alert for each of the major components of the algorithm.

Tensor Creation: Creating a tensor takes *O(N)* time for *N* non-zero entries in the tensor. This creation is independent of dimensionality used since each non-zero value will need to be processed regardless of the dimensionality. Likewise, tensor creation cannot be sped up since the values again must all be processed and assigned an index. The tensor creation process will be explained more deeply in the next chapter in the context of InSight2.

Computing Total Change and Energy: For *N* non-zero entries in a tensor, computing the total change and energy for the tensor takes *O(N)* time. This is because each of the *N* non-zero entries must be subtracted (in the case of total change) and additionally squared and summed (in the case the of energy). Unfortunately, there are no speed-ups to make multi-dimensional addition/subtraction quicker.

Finding Max Anomalous Slice: Finding the maximum change or energy for a tensor takes *O(S)* time for *S* total slices in a tensor, $S = \sum_{d \in D} d$, where *D* is the dimensionality of the tensor. For example, for a tensor with dimensions *a* × *b* × c, *S* = *a* + *b* + *c*. This process cannot be sped up since finding the maximum value in a list of values always takes linear time for an unsorted list. Sorting the list of changes and then finding the maximum change would take *O(SlogS)* time, longer than the original linear time.

The overall time complexity of RED Alert is *O(N + S)* since for each iteration of the algorithm, the *N* non-zero value differences must first be accumulated and calculated, *O(N)* time. This is followed by the detection of maximum slice, taking *O(S)* time. RED Alert runs in strictly linear time, and DenseAlert runs in log-linear time. However, DenseAlert's leading big-O coefficient, *RlogR*, is much smaller than *N* + *S*. *R*, the re-order region, consists of a subset of *S* and guaranteed to be much smaller than *S*. In turn, *S* is a subset of *N*, and guaranteed to be much smaller than *N*. Because of this, RED Alert's linear time complexity is a bit misleading when compared to DenseAlert. The latter algorithm is in fact faster since it only ever considers a subset of the entire tensor.

RED Alert Space Complexity: If stored directly without any improvements, storage of a tensor takes a maximum of *O(M)* for $M = \prod_{d \in D} d$, where *D* is the dimensionality of the tensor. This product can be exponentially large as the dimensionality increases. The tensors typically used are sparse, so sparse tensor representations are often useful to reduce the space complexity. RED Alert uses a sparse tensor representation called COO format, discussed in detail in the next chapter. For *N* non-zero entries and *d* dimensions, sparse tensor storage in COO format only requires *O(N + Nd)* space. This is because in COO format, only a list of the *N* non-zero values and a list of each non-zero value's index (*Nd* total space) is stored. This has a tradeoff of increasing time complexity of accessing each slice as the index list must be searched to find all entries in a specific slice. That is, to find all elements in slice *S* of the tensor, it takes *O(Nd)* time to locate all values for that slice since the entire index list must be searched. Although this may seem long, the sizes of tensors used in RED Alert are large and sparse, so sparse tensor representations are needed. Thus, the increased time complexity is necessary to be able to hold the data in memory.

Other than tensors, RED Alert uses constant memory to store temporary variables and constants. From this, the overall space complexity of RED Alert (assuming sparse tensor storage) is *O(N + Nd)*, which is equivalent to *O(N)*. RED Alert and DenseAlert have the same space complexity through the use of sparse tensor implementations in both algorithms.

3.2 RED Alert Results

RED Alert is able to detect and identify anomalous behavior from both a security and network performance standpoint. The results of these detections are shown in this section using both total change and energy. This section also explores the parameters of RED Alert, time window and threshold values, and provides a look at the robustness of the algorithm in terms of processing for long periods of time.

3.2.1 Packet Loss Detection

As done in the DenseAlert example, a tensor consisting of packet loss values between a source and destination at a given time (3-mode tensor) was used for the packet loss detection through RED Alert. Again, a time window of 1 minute was used, so each minute, the streaming tensor was aggregated and compared to the tensor corresponding to the previous minute. The results of running RED Alert can be seen in Figures 3.4 – 3.7 in order to attempt the packet loss detection from Figure 1.4

RED Alert uses its recursive filtering and expansion to detect a spike in packet loss at the country granularity level and trace it down all the way to the IP granularity level. Looking at Figure 3.4, the major spike around 14:30 is visible among other noise in the data. Using a threshold of 3000 at this level, 111 events were detected (shown through the red X's in the figure). These 111 events were then expanded to focus on city to city traffic, shown in Figure 3.5. Once again, the major spike was detected and alerted on, along with 29 other events, at the city granularity level using a threshold of 3500. This removed 81 insignificant events that were triggered at the country granularity level, saving computational time. Then at the institution granularity level, only 7 events were alerted with a threshold of 4000 as seen in Figure 3.6. Finally, only 5 events were alerted at the IP granularity from a threshold of 5000, shown in Figure 3.7. These events corresponded to the major spike in packet loss seen in Figure 1.4, shown through the green circle in Figure 3.7. At this level, the algorithm reported the IP addresses experiencing the extreme packet loss. In this case, a destination IP identified was one losing 40% of its packets through over 1000 connections in the time window. This indicates potential performance issues that warrants investigation.

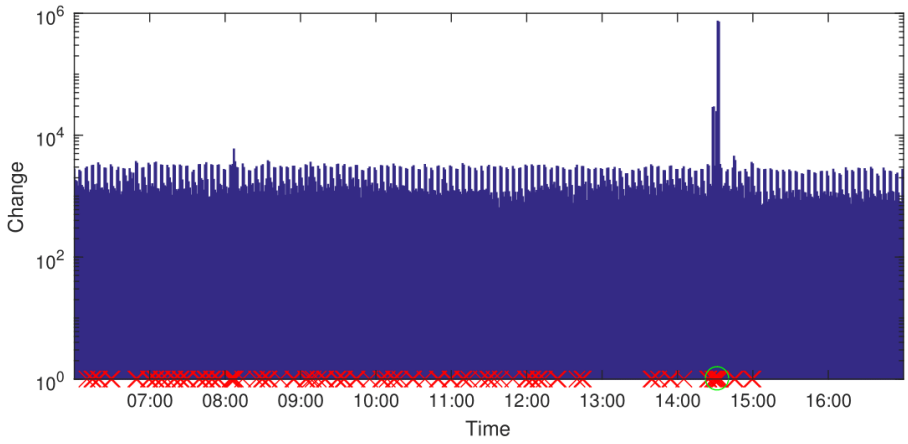


Figure 3.4: RED Alert Packet Loss, Country Granularity

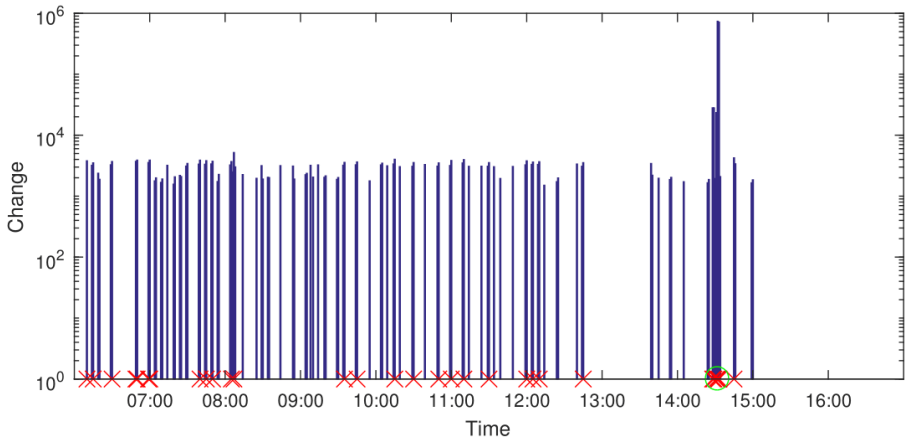


Figure 3.5: RED Alert Packet Loss, City Granularity

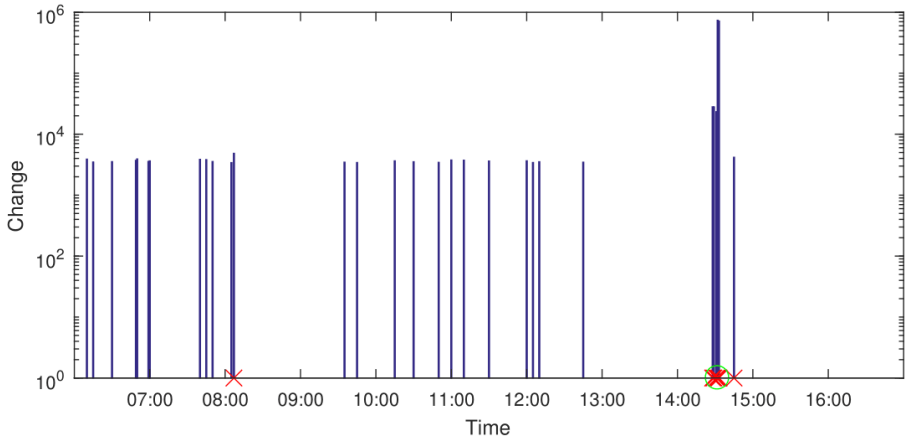


Figure 3.6: RED Alert Packet Loss, Institution Granularity

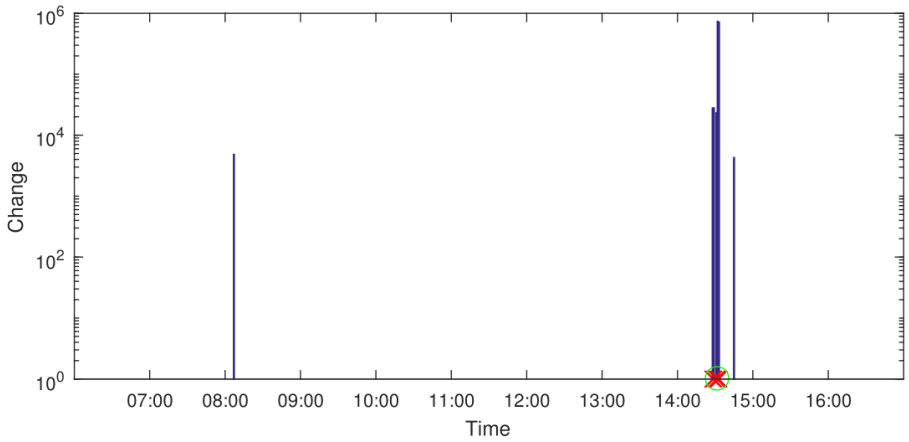


Figure 3.7: RED Alert Packet Loss, IP Granularity

3.2.2 Botnet Detection

Again, RED Alert ran through the same tensor passed to DenseAlert in its botnet detection. This consisted of a time × source × destination tensor with number of connections as its value. As shown in Figures 3.8 – 3.11, RED Alert is able to detect periods of high botnet activity at the country, city, institution, and IP granularity levels. At each transition to a finer level, uninteresting data is removed by the filtering process. 130 events were identified as being of interest at the country level (threshold of 1000), followed by 46 events at the city level (threshold of 1500), and 6 events at the institution level (threshold of 2000). Ultimately, 5 events resulted in alerts at the IP level (threshold of 2500). Any change value greater than 2500 at the IP granularity was alerted.

The alerts at the IP granularity level correspond to the increases in botnet activity seen in Figure 1.6. RED Alert identified the host responsible for the major spike seen around 9:00 in Figure 1.6. The detected IP address, which indeed was labeled as a botnet, went from no requests at 9:06 to over 5,000 requests at 9:07. Additionally, the IP identified at the spike around 15:00 was a group of botnets that increased activity from 0 requests at 14:59 to over 3,000 requests at 15:00. RED Alert was able to detect these events at the country granularity level and successfully track them to the offending hosts at the IP granularity level. The spike at 10:00 was traced down to the final granularity level, but was not alerted as it did exceed the threshold.

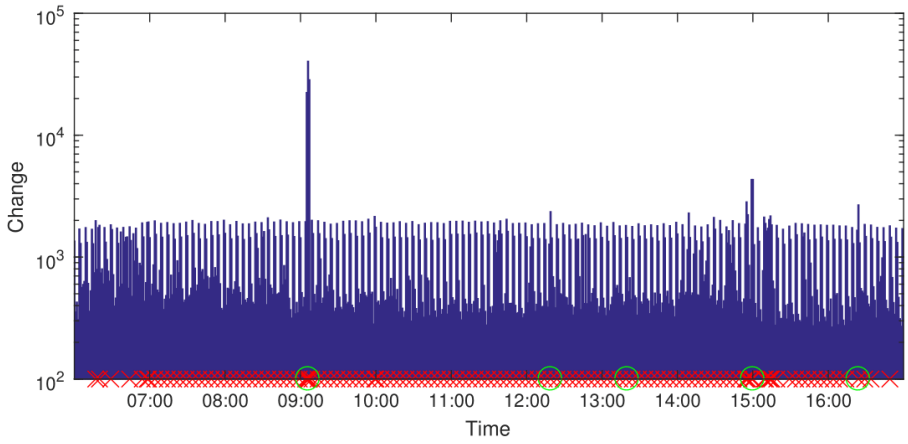


Figure 3.8: RED Alert Botnet, Country Granularity

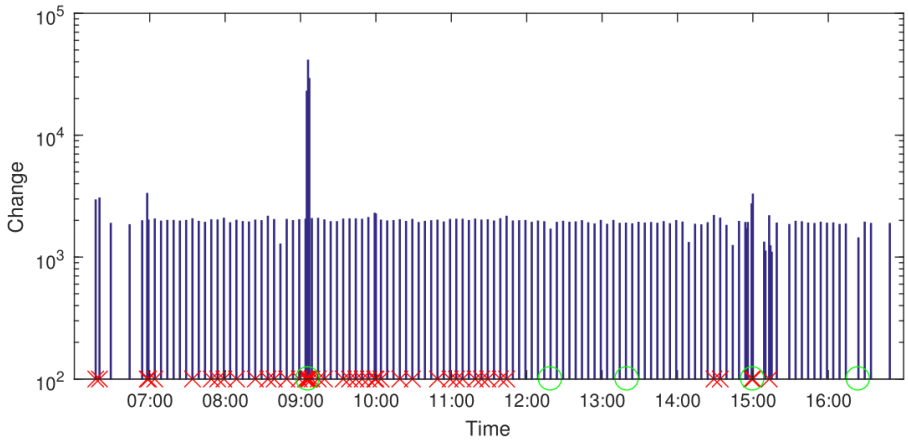


Figure 3.9: RED Alert Botnet, City Granularity

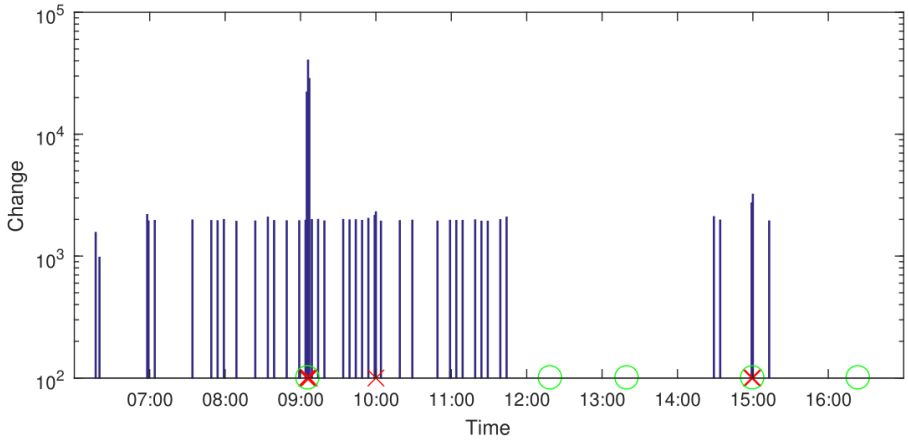


Figure 3.10: RED Alert Botnet, Institution Granularity

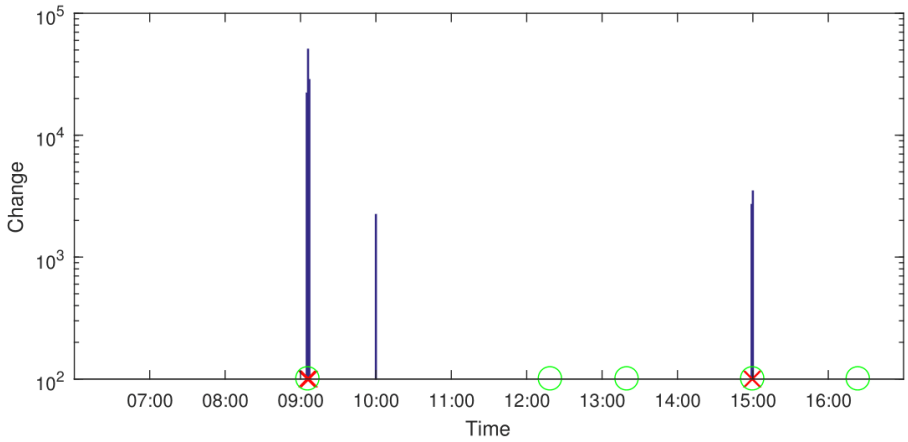


Figure 3.11: RED Alert Botnet, IP Granularity

3.2.3 Choosing Threshold Values

In both the botnet detection and packet loss examples, a periodic noise can be seen in the data with a total change value around 1000 for the botnet example and 3000 for the packet loss example. This noise represents the normal “hum” of the network traffic from users connecting and disconnecting, or the change in packet loss in the network for each time window. Thus, it is important that the first threshold for spawning the recursive search of RED Alert is at least this normal “hum” value (leading to the initial thresholds of 1000 and 3000 in our results). This way, any activity that is outside of this range would be explored through the remaining steps of RED Alert. The other threshold values should scale according to the user’s desires. It is up to the user to determine which threshold values provide the most meaning in the context of their network, so some trial and error with the threshold values is necessary to properly utilize RED Alert.

As seen in the botnet example in Figure 3.11, three events were not alerted at the final granularity level. This was due to the threshold chosen. If the threshold was lowered at a coarser granularity, the events would have been tracked down to the IP granularity level and alerted on. However, the threshold was set to detect the larger spikes at 9:00 and 15:00 and thus did not consider the other anomalous events at 12:15, 13:15, and 16:00.

3.2.4 Effects of Time Window

As is important with any machine learning algorithm, finding the best value for parameters is key to an algorithm's success. RED Alert has two main types of parameters, the time window and the thresholds. As just described, the thresholds are more of a user-configurable parameter for what is being detected. On the other hand, the time window could have drastic results on the detection in the algorithm. Too small of a time window could cause a large amount of noise being detected in the data whereas too large of a time window could lead to events being undetected when they are washed out by other noise.

RED Alert was run using varying time windows for the detections shown previously, focusing on the effects of the country level detection as it is what ultimately leads to the recursive event detection. As shown earlier, a time window of 1-minute provided meaningful and conclusive results. At the 10-second time window seen in Figures 3.12 and 3.14, the data is noisy, but still detects the spikes in packet loss for the network and the spikes in botnet activity. From a data analytics perspective, this time window performs well, but is a bit more visually unpleasing than the 1-minute time window. The periodic noise seen here is a bit more noticeable than in the 1-minute time window.

When using a 5-minute, shown in Figures 3.13 and 3.15, the plots fail to indicate some activity that was evident in the 1-minute and 10-second time windows. For example, in Figure 3.15, anomalous activity from 15:00 to 16:00 is washed out when using the 5-minute window, but not in the 1-minute and 10-second windows. This is because when looking at the larger time window of 5-minutes, the change may not be as drastic since more values are being aggregated even though the same slice may be detected as the most changed. For example, in the botnet detection, Russia was alerted as the most changed slice at the country level at 15:00. However, in the 5-minute time window, Russia increased from 23,400 requests at 14:55 to 24,700 at 15:00. Even though this was the most changed slice, the total change is only 1300, much lower than the orders of 10,000 typically calculated for the total change for this time-window. At the 1-minute scale, Russia increased from 2500 requests at 14:59 to 10,000 requests at 15:00. This caused the total change value to be about 7500 and noticeable in the 1-minute plots. Although the label of the most changed slice does not tend to change between different time windows, the value calculated for larger time windows may not be as meaningful when plotted.

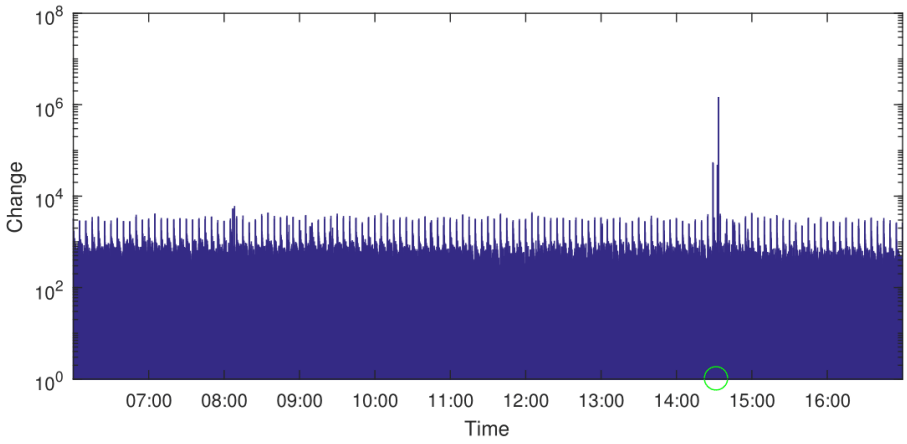


Figure 3.12: 10-second Time Window, Packet Loss Detection

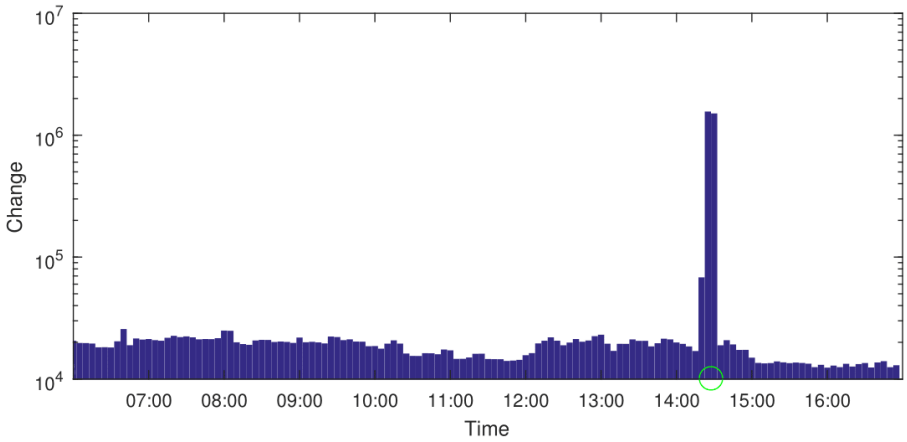


Figure 3.13: 5-minute Time Window, Packet Loss Detection

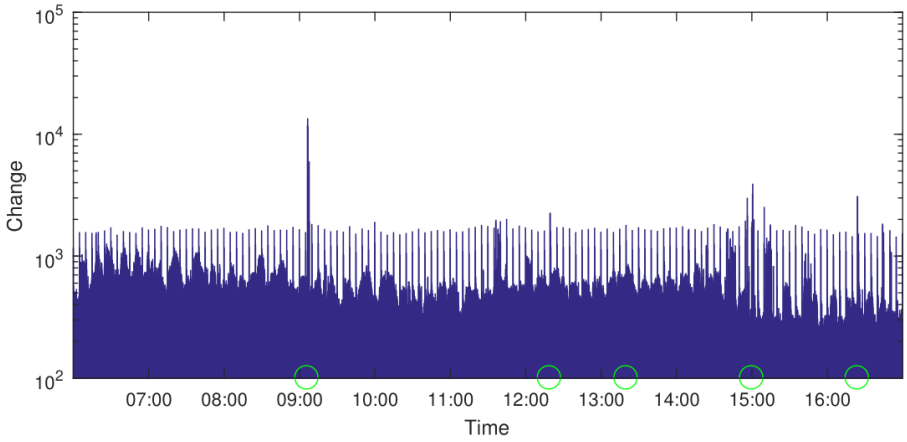


Figure 3.14: 10-second Time Window, Botnet Detection

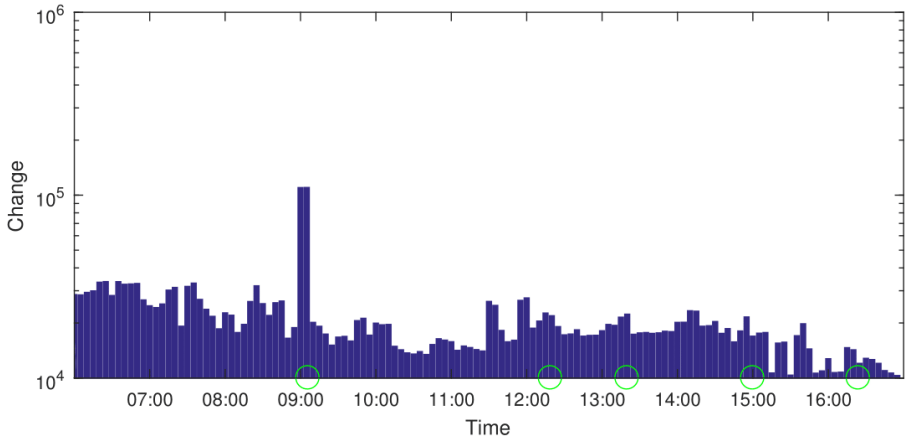


Figure 3.15: 5-minute Time Window, Botnet Detection

Another key importance about selecting the time window is the effect on computation time. With an increase in the time window, tensors will be larger due to the increase in the number of data points. Since RED Alert processing time scales linearly with respect to the number of non-zero values, an increase in data size leads to an increase in the execution time of the program. For example, for the packet loss detection, RED Alert takes about 50 minutes total for the 10-second time window, about 56 minutes for the 1-minute time window, and about 74 minutes for the 5-minute time window. Likewise, RED Alert takes about 36 minutes total for the 10-second time window, 37 minutes for the 1-minute time window, and about 57 minutes for the 5-minute time window in the botnet detection. Thus, there is a tradeoff with selecting the appropriate time window that works well for the data and also does not cause tremendous increases in execution time.

3.2.5 Robustness

To test the robustness of RED Alert, RED Alert was run on a month of data to see if the algorithm could handle long stretches of time without crashing and still properly filter out uninteresting data.

RED Alert ran through this month of data for both the packet loss example and botnet detection example. The dataset consisted of a total of over 590 million records. In the packet loss example, thresholds of 3000, 3500, 4000 and 5000 respectively for each of the granularity levels were used again as they provided conclusive results in the previous example. From this, a total of 6371 events was alerted at the country level, 980 events at the city level, 276 events at the institution level, and finally, 229 events were deemed anomalous at the IP granularity level.

For the botnet example, the same thresholds were used of 1000, 1500, 2000 and 2500. This led to of 13,500 events alerted at the country level, 293 events at the city level, 179 events at the institution level, and 173 events at the IP granularity level. Processing this month of data took about 3600 minutes (2.5 days) for the packet loss example and 4900 minutes (3.5 days) minutes for the botnet detection example. This shows that RED Alert is able to keep up with large amounts of data in “faster than real-time” allowing for quick response when used on real-time network data.

One potential limitation to using these constant thresholds is the changes in network behavior through days of the week. Network traffic on a weekend is much less active than network traffic on a weekday, so it would make sense to have varying threshold levels based on the day of the week and time of the year to provide more accurate detection. Some events on a weekend may be considered anomalous, but lower than the thresholds used for weekdays. This would cause some events to go unnoticed. However, using time-varying thresholding requires extensive understanding of the network behavior through a large amount of statistic collections and analysis.

3.2.6 Using Energy as Criteria

Total change was one evaluation criteria explored for detection through RED Alert. Another value RED Alert used was the energy metric explained in the introduction. For this, slices with the most energy were detected and alerted in the algorithm using thresholds based on slice energy. With respect to the RED Alert algorithm shown in Figures 3.2 and 3.3, nothing in regards to the algorithm’s process change besides calculating the energy of each slice rather than the total change. The algorithm merely detects the slice with the most

energy, alerting on energy-defined thresholds. Since values are squared in the energy calculations, larger values for thresholds are necessary.
For the packet loss example shown in Figures 3.16 - 3.19, using energy as the threshold criteria led to conclusive results. From Figure 3.16 at the country granularity level, an energy threshold of 1,000,000 led to 130 events being detected. At the city granularity in Figure 3.17, a threshold of 20,000,000 led to 8 events being detected. Then using a threshold of 40,000,000 at the institution granularity level led to 6 events continuing on to the IP granularity level (Figure 3.18). Finally, as displayed in Figure 3.19, a threshold of 160,000,000 led to 5 events being alerted as anomalous at the IP granularity level. RED Alert still alerted with the IP addresses leading to the events detected at this level. These final 5 alerts corresponded to the same IP addresses as the ones alerted in the total change example discussed previously.

With reference to the plots in Figures 3.20 – 3.23, RED Alert was also successful in its detection of botnets using the energy criteria. The threshold values used here were much smaller than the ones in the packet loss example, but still larger than the thresholds from the total change example. For the country granularity level in Figure 3.20, a threshold of 150,000 resulted in 76 events detected. These events were filtered down to 26 events at the city granularity level from a threshold of 600,000, shown in Figure 3.21. At the institution level, a much larger threshold of 3,500,000 was needed to filter these events to the 7 events seen in Figure 3.22. Finally, in Figure 3.23, a threshold of 9,700,000 was used to alert on 6 IP addresses. 5 of the IP addresses alerted at the final level corresponded to the same IP addresses detected from total change. The sixth IP address corresponds to the event that was detected around 16:20 that was overlooked when using total change. This is because energy is an L2-norm, so the change values will increase more drastically and can be detected with lower thresholds. The lack of detection for the botnet at event at 12:15 is due to a botnet that was already active becoming more active, and the event at 13:15 is undetected due to a botnet becoming gradually more active over time. RED Alert is only sensitive to sudden changes in the network as opposed to gradual changes.

Using energy takes slightly more time than using total change due to the extra calculations in calculating and squaring the norm of the slices. RED Alert took about 56 minutes for the packet loss example using total change and 60 minutes using energy. Similarly, using total change took 37 minutes for the botnet detection, but total energy took 46 minutes. These differences become starker when used on longer periods of time and larger networks.

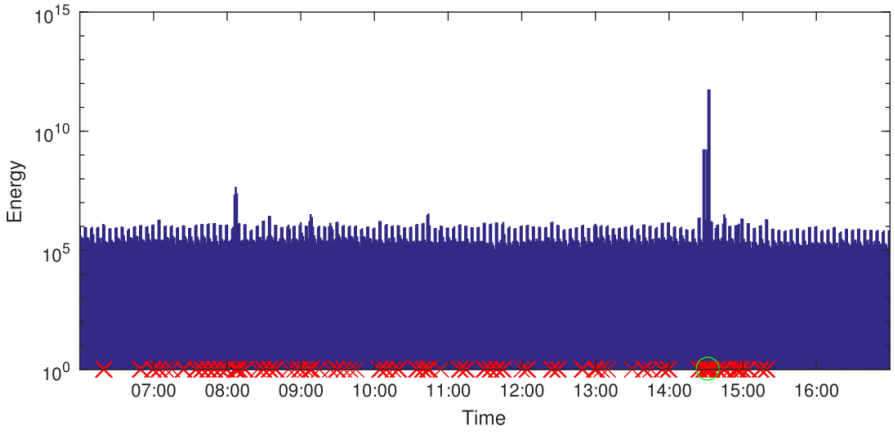


Figure 3.16: RED Alert Packet Loss using Energy, IP Granularity

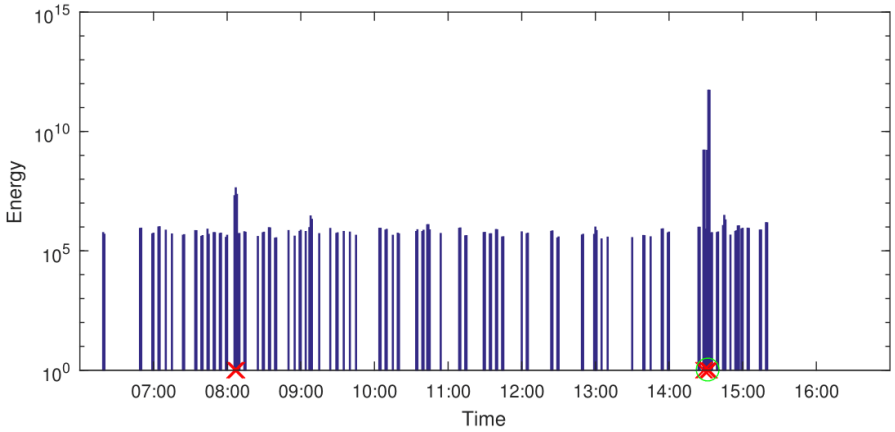


Figure 3.17: RED Alert Packet Loss using Energy, City Granularity

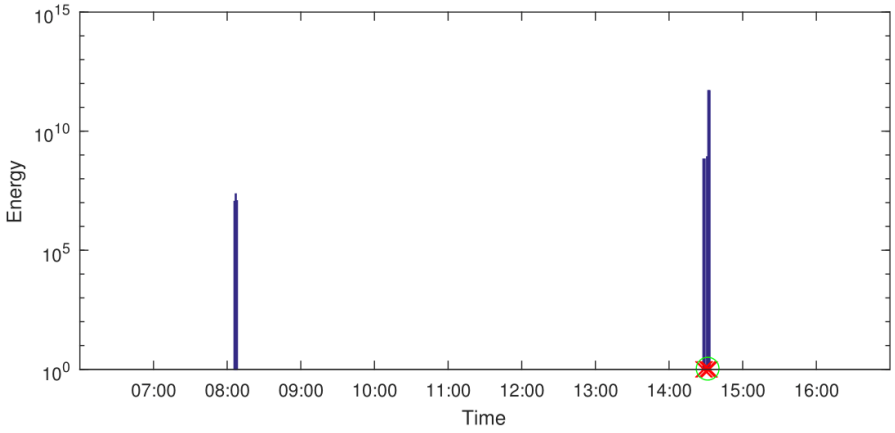


Figure 3.18: RED Alert Packet Loss using Energy, Institution Granularity

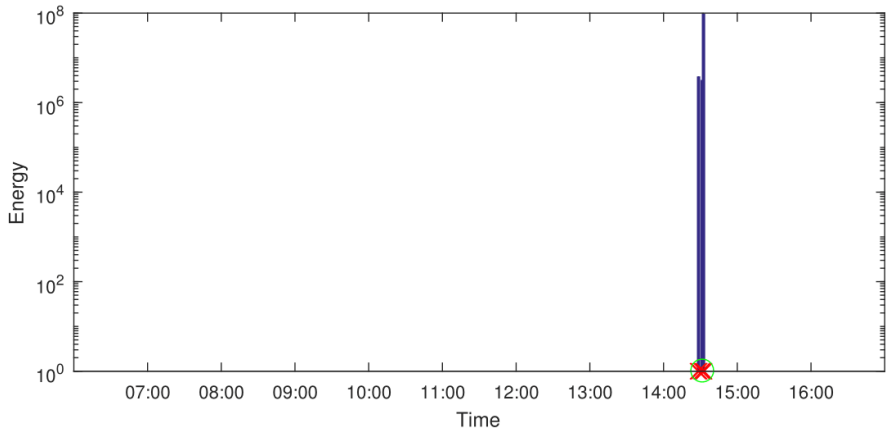


Figure 3.19: RED Alert Packet Loss using Energy, IP Granularity

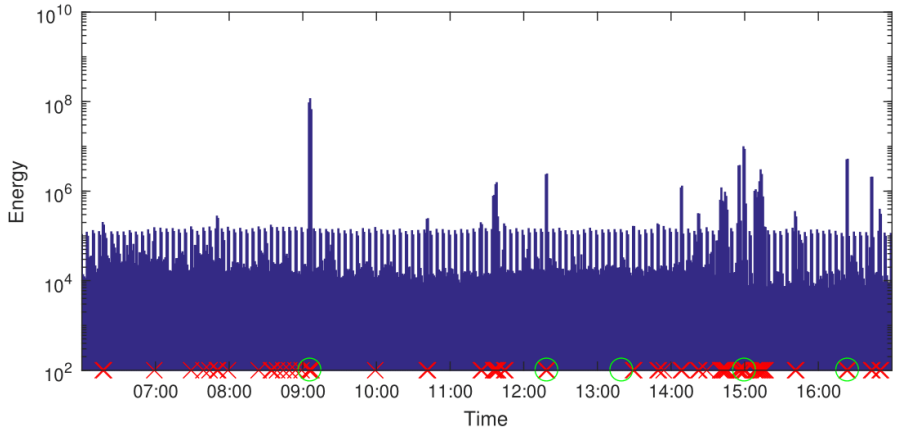


Figure 3.20: RED Alert Botnet using Energy, Country Granularity

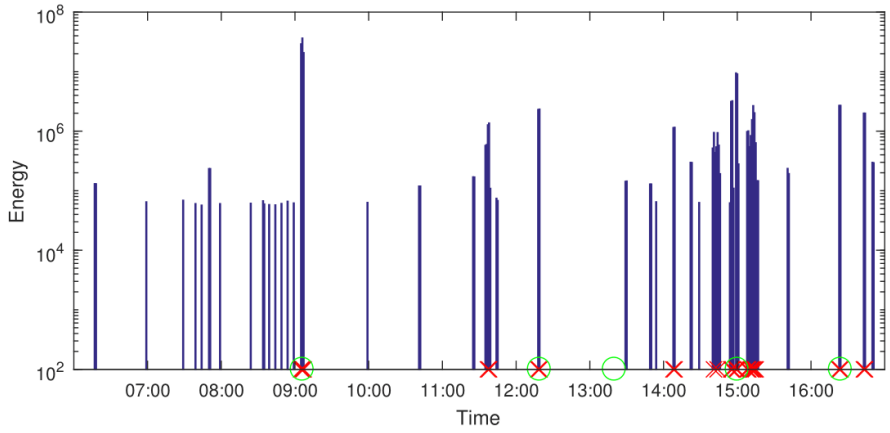


Figure 3.21: RED Alert Botnet using Energy, City Granularity

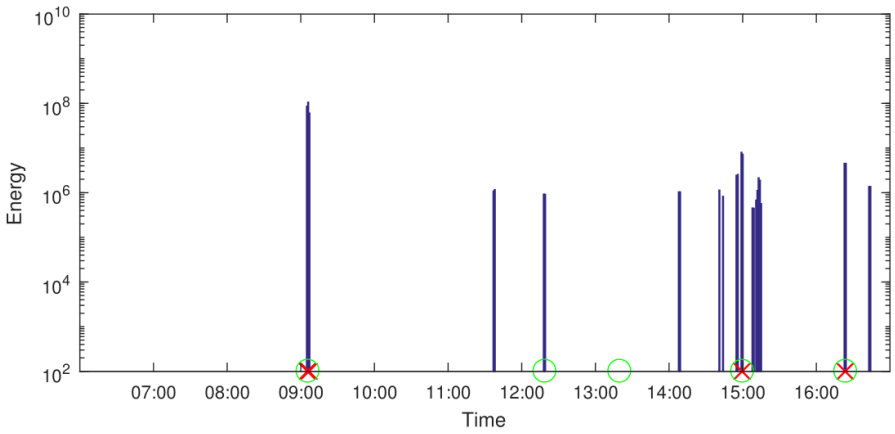


Figure 3.22: RED Alert Botnet using Energy, Institution Granularity

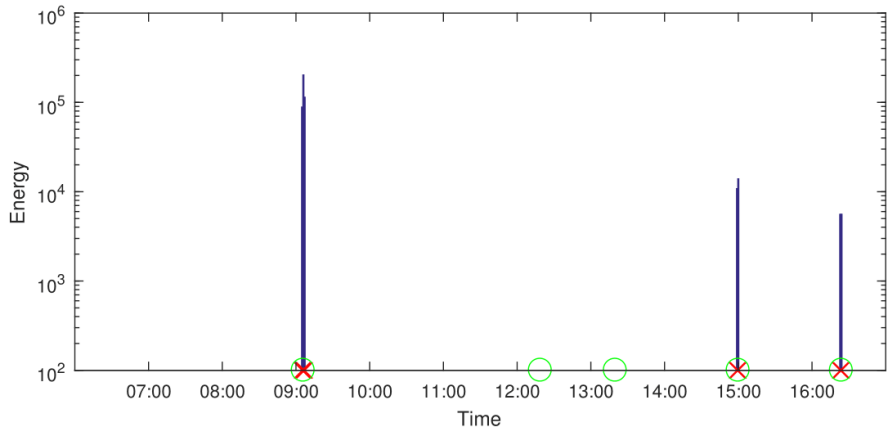


Figure 3.23: RED Alert Botnet using Energy, IP Granularity

4. MODELING RESULTS WITH INSIGHT2

InSight2 is a network monitoring system developed at the University of Tennessee to provide a platform for data analytics and visualization of large-scale network data. In this chapter, the history, motivation and current architecture of InSight2 is outlined as well as how RED Alert interacts with the platform.

4.1 History and Motivation of InSight2

InSight2 serves as a continuation of the GLORIAD InSight project developed in 2013. InSight solely monitored the GLORIAD research and education network described below, but InSight2 serves as a generic, open framework capable of providing analytics for any network. This section explains the transition from InSight to InSight2.

4.1.1 GLORIAD and InSight

Due to the sheer size of GLORIAD and the data associated with it, InSight was created in 2013 as a network monitoring tool to ensure high performance and security in the international network [19]. Its focus was to provide in-depth analyses on network flow data such as jitter, packet loss, and various security measurements.

However, in 2014, GLORIAD ended and with it came the termination of InSight. InSight's dependencies changed and updated after InSight's termination causing it to be useless. Updating the original code base of InSight seemed to be near impossible due to the complex components with little documentation and no modularity. Additionally, it was unclear as to what types of database backends were used to host the GLORIAD data for display. For these reasons, modernizing InSight would require much more work than anticipated. From NSF sponsorship under the project "The InSight Advanced Performance Measurement System" with sponsorship from IRNC-AMI (International Research Network Connections - Advanced Measurement Network Infrastructure) program,

development for a completely new InSight platform (aptly named InSight2) began.

4.1.2 From InSight to InSight2

InSight2 is a complete redesign from the original InSight framework [1]. InSight2 strived to have the modularity and cohesiveness that InSight lacked in a modern framework with less moving parts. The goals of InSight2 remained the same as those of InSight – providing a large-scale network visualization tool that allows for network operators to quickly determine the health of their system. Moreover, InSight2 was developed with extendibility in mind so that users can define their own dashboard interfaces and provide their own data to be used with the tool. All tools and databases used in the framework are free or open-source so that InSight2 can be shared with anyone without licensing issues.

InSight2 consisted of some notable changes from the original InSight. The first change was to use a Python code base rather than Perl. Python is a very robust and user-friendly language that would easily allow for extensions to the platform without rewrites in the code base. Although Python is a scripting language, it contains object-oriented design mechanisms that allow for modularity in software design. Moreover, InSight2 used Elasticsearch [20] as the sole data storage database. Python has libraries that interact directly with Elasticsearch databases (elasticsearch-py and elasticsearch-dsl-py) allowing for quick and easy upload and retrieval of data stored in the database. Another useful component of having Elasticsearch access is the ability to create visualizations of the data within the same framework with Kibana. Kibana users can access the various indices within Elasticsearch and create a variety of different visualizations of the data. Examples of highly used visualizations are area plots, line plots, heat maps, and bar graphs among many others. Kibana also has the ability to aggregate multiple visualizations together to create dashboards. Using these Kibana dashboards and an HTML frontend, the InSight2 visualization platform can be seen anywhere from the web. This differs from the original InSight in that multiple different database types (MySQL, SQLite, and Elasticsearch) were aggregated with a customized Kibana front-end for display. The InSight2 method is much more compartmentalized with only the Elasticsearch and Kibana interaction.

Some components from InSight stayed the same due to their performance. Argus was kept as the network flow aggregating tool since it is versatile and fast, supporting the collection of up to 127 different fields for each flow. Currently, work is underway to support other network flow aggregators such as NetFlow. Additionally, the majority of the dashboards were kept the same since the dashboards used in InSight provided meaningful displays in regards to the health of the network. In the next section, the changes made are described in detail.

4.2 Architecture Overview

The InSight2 architecture was developed with simplicity and modularity in mind [21]. Each component of the design is independently developed and maintained so that add-ons and changes can be made to the design without disrupting the flow of InSight2. These modules work synchronously together to provide a better user experience as well as having more efficient disk usage. The main modules are the Enrichment Module (EM), Updater Module (UM), Summarizer Module (SM), and user-defined plug-ins. These modules are described below as well as how they work together in the InSight2 framework.

4.2.1 Enrichment Module

The Enrichment Module (EM) serves as the core component in InSight2. Its purpose is to collect the output from the Argus flow data and provide further metadata from other database sources. The EM then uploads the aggregated data to the Elasticsearch database. The EM works both for archived flow data, such as the GLORIAD data, or can be used on live streaming data. The database sources used to provide further metadata are described below. The Global Science Registry and MaxMind GeoIP databases are the ones used for RED Alert's granularity levels.

Global Science Registry (GSR): The Global Science Registry is a SQL database from the original InSight that is uploaded to Elasticsearch. The GSR is composed of tags for over 14,000 different institutions that used GLORIAD. For InSight2, GSR is used to collect information about each unique IP address's domain name server (DNS) information, the labels for each IP node, the IP

name, the address to domain ID matching, location information, autonomous system (AS) numbers, the IP address's ISP name, and the organization the IP belongs to. Since this database is so large, it is stored in its own index in the Elasticsearch database to provide quick access and searching.

MaxMind GeoIP: MaxMind's GeoIP database contains information about IP address locations. The location data varies in granularity level including information about the IP's city, province, zip code, country, latitude and longitude. Using the Python module pygeoip, IP addresses are quickly tagged with the appropriate geolocation data. The GeoIP database is stored on disk since it is implemented using the Python library.

Threats Databases (TD): Various online databases listing malicious IP addresses are aggregated into the Threats Database (TD) upon InSight2's creation to tag potential suspicious activity in the network. The TD contains information on emerging threats, CYMRU Bogons, and IP addresses belonging to the Zeus, Feodo, Palevo, and Spyeye botnets. Using the Updater Module described in the next section, these threat databases are guaranteed to be up-to-date and contain the most recent lists of malicious IP addresses. This database is also stored in Elasticsearch to provide quick searching and tagging of malicious IP addresses.

The EM is parallelized to provide quick collection and upload of flow data by distributing the enrichment across all available cores. This is done by using Python's multiprocessing library and splitting the data into chunks to be processed and uploaded individually by each core. Each core has its own thread from the main program with each thread processing one flow record at a time. Race conditions are avoided using the multiprocessing Manager class in Python that provides shared data types across threads. This ensures that two threads do not try to index to the same point in Elasticsearch and cause overwrites. Moreover, these threads are lightweight, containing only the necessary values needed for enrichment – the data to be enriched, access to the Elasticsearch database, access to the enrichment databases, and other small data types such as verbose flags and file names.

The EM works as follows. The first step in the enrichment is to spawn off the Argus client. Using the subprocess Python module, an Argus thread is started to pipe flow data into the EM. If archived data is used, the data is broken up into smaller sets so that the Argus call does not hold up the enrichment. If live data is

used, the output buffer from Argus is read every few seconds to gather any new flow data that has appeared. The flow data typically amounts to a large quantity of records since the records are stored by milliseconds. When the Argus call finishes (or when the live stream buffer is read), the resulting flow data is chunked into smaller pieces to begin the parallelized parsing of the flow data for further enrichment. This parsing works by accessing the various fields returned by Argus and adding them to a Python dictionary for temporary storage. Checks are made to ensure that no data fields are left empty or of the wrong data type. If a field is left empty, a default value is used. Using the source and destination IP address fields, information from the previously mentioned databases are aggregated as well and added to the dictionary. Once all fields have been added to the dictionary, it is converted to a JSON object for upload to the Elasticsearch database within the appropriate index. Once a thread finishes parsing and uploading its data, it joins back to the main thread and is given a new set of data to parse.

4.2.2 Updater and Summarizer Modules

The EM is useful for uploading the data to the Elasticsearch database, but it is important to ensure that the data in the database is up to date and aggregable. This is the motivation for the Updater Module (UM) and Summarizer Module (SM). The UM is used to periodically check for updates in the databases used in the EM to see if any new information is available. The SM is able to “summarize” data in longer time periods than the millisecond granularity used in the EM. Both of these modules are described in this section.

Updater Module*:* The UM ensures up-to-date enrichment databases by polling the sources of each database periodically. If a database changed in the past time period, the UM deletes the old database from Elasticsearch and uploads the more modern one in its place. The UM is run as a background daemon with logging capabilities to log changes to the databases and alert the user if problems with the module arise. This time period is configurable by the user, and each database can be individually configured to be checked at different time intervals. Examples of information updated through the UM include newly discovered botnet IP addresses and organizational labels for IP addresses.

Summarizer Module: From the EM, data is collected at the time granularity of milliseconds. In some instances, this granularity is too fine for network operators. The SM is designed to fix this issue by summarizing numerical data in larger time windows such as minute-to-minute scale. The SM summarizes data in the following ways. Source and destination IP addresses are summarized by using least specific subnet prefixes. Number of bytes and packet counts are individually summed. Jitter and inter-packet arrival times are individually averaged per time window. This allows for a quick snapshot of what the network traffic looks like at a specific time without using the large data at such a fine granularity.

4.2.3 Plug-ins

The dashboards generated by InSight2 only provide information about the raw data collected in the EM. For proper data analysis, intelligent algorithms are useful to provide further insight as to underlying trends in the data. Plug-ins are user-created modules that have access to the InSight2 Elasticsearch database to collect flow data and provide more informative network analytics. Plug-ins are designed to work by accessing the Elasticsearch database to quickly obtain the dataset, perform some sort of analytics, and upload the resulting data to a new Elasticsearch index. The resulting data in the new index can then be visualized in a Plug-in dashboard with the visualization design suited for the analysis. Since InSight2 uses an Elasticsearch backend, users can develop Plug-ins in their programming language of choice to interact with the database. Currently, Elasticsearch has APIs in Java, Javascript, Perl, Python, PHP, and Ruby. Despite InSight2's internal development in Python, any of these languages can be used to create Plug-ins. Examples of Plug-ins developed already are Markov chain analysis and RED Alert.

4.2.4 Dashboards

As briefly mentioned earlier, dashboards are the ways in which the data created by these different modules are visualized. The dashboards used in InSight2 were inspired by the dashboard interface from the original InSight, but with a modernized interface. Dashboards group related visualizations together, separating each dashboard in different tabs on the InSight2 webpage. These

dashboards are iFrames from Kibana stored in an HTML file for display on the web. This means that any browser with Javascript capabilities can load the dashboards for viewing from any location. The three main dashboards are the overview dashboard, the performance dashboard, and the connections dashboard. As mentioned in the previous section, additional dashboards may be created for plug-in modules.

4.2.5 InSight2 Workflow

InSight2 has modularized the framework, so it is important that the individual components interact in a cohesive fashion to provide a reliable and efficient system. Each module works independently such that a problem in one module does not disrupt the process of the overall framework. Additionally, as modules are added or removed in potential future updates, the whole framework does not need to be redesigned as it is robust to changes.

The workflow for the InSight2 framework proceeds as follow. The EM is spawned to collect the Argus flow data and upload the enriched data to Elasticsearch. From the EM, each additional Plug-in is started to collect the data from Elasticsearch and perform their analyses. The EM periodically checks that each Plug-in is still running and not using too much computational power. The results of the Plug-ins are uploaded back to Elasticsearch and stored there. Separately, the UM and SM interact with Elasticsearch. The UM daemon is inactive for most of the time and updates the enrichment databases when needed. The SM periodically accesses the Elasticsearch database and summarizes data, uploading the summarized data back to Elasticsearch in a new index. Using a webserver with embedded Kibana visualizations, the various dashboards are visible to network operators. The Kibana visualizations are built on top of Elasticsearch which allows for direct access to create visualizations without the need for a middleman. Figure 4.1 displays a diagram depicting the workflow for InSight2 and how each module interacts with one another.

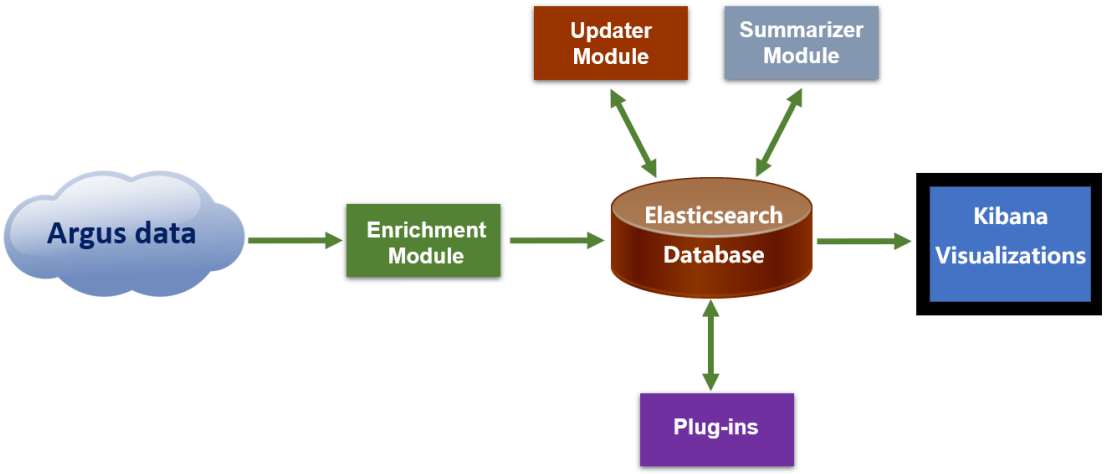


Figure 4.1: InSight2 Architecture Workflow

4.3 Tensor InSight2 Framework

InSight2 is designed for large-scale networks, so the data aggregated from the platform is large in volume. The dimensionality of the tensors can reach several thousands of indices per mode. The tensors are also quite sparse due to the nature of the data. Thus, an efficient method for storing and accessing large sparse tensors is necessary for tensor analyses. In this section, Python libraries for tensor computations are surveyed and analyzed. Then, the implementation used for the RED Alert plug-in is described.

4.3.1 Overview of Python Tensor Libraries

Since the development of InSight2 consisted of using the Elasticsearch APIs in Python, the RED Alert plug-in was designed in a Python framework. However, Python has limited resources for tensor development. Many libraries exist for the creation and processing of high-dimensional data, but many lack key functionalities or documentation for proper use. This section summarizes some of the tensor libraries tested for RED Alert's InSight2 plug-in.

NumPy and SciPy: NumPy [22] is perhaps the most popular Python multi-dimensional computation framework. The library is part of the SciPy numerical computation library in Python. NumPy is a lightweight library contain only the necessary tools to create and manage multi-dimensional objects with some computational tools, whereas SciPy contains much more computational packages such as statistics and linear algebra tools wrapped around BLAS and LAPACK frameworks. These libraries are often compared to MATLAB due to the similar uses and capabilities between the two.

NumPy is popular due to its easy and efficient use of *n*-dimensional arrays. Python lists can easily be converted to NumPy arrays by using simple calls to the library. Retrieval, filtering, and searching of data in these arrays is fast as well. NumPy even contains functions for addition, subtraction, dot product, division, inverse, and many other useful math tools for their *n*-dimensional arrays so that the user does not have to iterate over the entire structure to compute some result.

The fact that NumPy supports an arbitrary number of dimensions seems to suggest that it could be used easily as the tensor framework for RED Alert, but there are some notable limitations to using NumPy as a tensor framework. For instance, modifying the multi-dimension data such as appending rows or columns and adding dimensions is extremely slow in NumPy. Network data is dynamic and may require deletions and additions to the tensors, so quick tensor manipulation is essential for robust computation. Additionally, as mentioned earlier, network flow data in a tensor framework is quite sparse. The data can grow very large at times and exceed Python memory capacities. Thus, some sort of sparse tensor compression is necessary for our analyses. NumPy itself does not support any sort of compression and requires all data to be stored in RAM at a given time. This solution will not work for all of our data. Although SciPy does contain support for sparse matrix representations, it does not extend to *n*-dimensional data objects as needed in our algorithm.

TensorLy: TensorLy [23] is a Python library designed for tensor storage and computation with minimal dependencies. It contains not only support for tensor storage and manipulation such as accessing sub-tensors and slices, but it also comes with high-end tensor computations such as tensor decompositions and regressions. TensorLy supports three different Python libraries as its internal backends for storage, NumPy, PyTorch and MXNet. PyTorch and MXNet are

both large-scale deep learning frameworks that are capable of GPU usage that contain *n*-dimensional data objects as NumPy does.

TensorLy seems to have most of the functionality necessary for RED Alert. Though, as of late 2017 when this tensor analysis work began, no support for sparse tensors existed for TensorLy (the authors mentioned it was on their roadmap for the future). Once again, the memory problems of NumPy and SciPy arose again where large tensors could not be stored in RAM when using Python. As mentioned, TensorLy does support high performance backends that support GPU use, but since RED Alert was designed for generic computational frameworks without GPU support, these backends were not an option. TensorLy may be revisited in future work once sparse tensor support is released.

scikit-tensor: Scikit-tensor [24] is a Python library built on top of SciPy and NumPy to provide tensor support through multilinear algebra and tensor factorization functions. Some of the same high-end functions from TensorLy were available in scikit-tensor as well. More importantly though, scikit-tensor consisted of support for both sparse and dense tensors. For dense tensors, scikit-tensor used the NumPy *n*-dimensional arrays. For sparse tensors, scikit-tensor used the Coordinate format (COO) from SciPy that stores only non-zero data.
COO works as follows. *n* different lists representing the *n* dimensions hold the coordinates of the various non-zero values. For a tensor of 3 modes, the coordinates (*i, j, k*) would be split into three different lists all at the same index. Another list is stored that holds the non-zero values in the sparse data structure. If there are *m* non-zero values in the sparse data structure, each of these *n+1* lists would contain *m* values.

For example, consider a sparse matrix (tensor of two modes). The COO representation of this matrix would contain a list for the rows with non-zero values and a list with the corresponding columns with non-zero values. The COO representation would then consist of a third list containing these non-zero values. Thus, the i^{th} non-zero value, stored in *data[i]* would be located at (*rows[i], columns[i]*) in the matrix. Figure 4.2 shows a depiction of COO for a matrix of dimensions 3 × 2.

Even though Scikit-tensor contained the functionalities needed for RED Alert, the documentation was scarce and difficult to find, and the project had not been updated in almost 2 years.

$$\begin{bmatrix} 0 & 1 & 2 \\ 0 & 0 & 8 \\ 3 & 0 & 0 \\ 0 & 7 & 0 \end{bmatrix}$$

Rows	0	0	1	2	3
Columns	1	2	2	0	1
Data	1	2	8	3	7

Figure 4.2: COO Representation

4.3.2 Modifications to scikit-tensor

Since scikit-tensor contained the core functionalities necessary for our tensor processing, the library was selected as the basis for our toolkit with some modifications to fully meet the requirements of our algorithm. Scikit-tensor lacked functionality in accessing only subsections of the tensor such as slices and sub-tensors. Consequently, the four following functions were added to scikit-tensor for use in our tensor toolkit:

- *slice(mode, i)*: Return the list of non-zero values at slice *i* in mode *mode*.
- *slice_toarray(mode, i)*: Return the entire slice as an *n*-dimensional NumPy array at slice *i* in mode *mode* including the zero values.
- *delete(mode, i)*: Delete the non-zero values in the slice *i* of mode *mode*.
- *subtensor_toarray(mode, begin, end)*: Return the subtensor as a NumPy array from slice *begin* to slice *end* in mode *mode*.

These additional functions provided easy access to subsets of the tensors without having to expand the entire tensor and running into memory issues as was required in the original scikit-tensor library. Using these functions, tensors could also be quickly resized and modified which was slow in NumPy. The code for the tensor library used in RED Alert can be seen at https://www.github.com/jtliso/scikit-tensor.

4.3.3 Interaction with Elasticsearch

As a plug-in for InSight2, functionality was needed to interact with the Elasticsearch backend to create the tensors and perform the analyses. Using the elasticsearch-py library, the InSight2 database could be queried to collect data.

Tensors are built from an *n*-tuple of data gathered from Elasticsearch. These *n*-tuples consisted of (*n-1*) parameters (e.g., time, source information, destination information, protocol, etc.) and a value (e.g., packets, number of connections, bytes, etc.) as the n^{th} item in the tuple. This corresponds to a tensor of *n-1* modes with each mode corresponding to one of the *n-1* parameters collected from Elasticsearch. The user needs to specify a time range of data to collect, the parameters of RED Alert (time window and thresholds) and the value to be collected within the tensor. Figure 4.4 depicts a tensor of mode 3 collected using the 4-tuple (time, source country, destination country, number of connections). Spikes in the tensor represent values at each cell in the tensor.

The tensors are built in the following way. A query to Elasticsearch is sent using the Search API from elasticsearch-py. For longer time ranges (greater than 1 hour), the searches are broken into one-hour chunks to prevent stress on the Elasticsearch server and preserve RAM. These searches are sorted by the *n-1* dimensions that are passed using Elasticsearch's Sort API. This guarantees that tuples containing the same *n-1* parameters will be returned in succession. This is important for creating the tensor in COO format as explained later in this section.

The n^{th} entry in the *n*-tuple represents the value, and this value is stored in a list of these values from Elasticsearch. This works well with the COO format since it only needs to store the non-zero values (i.e., any value obtained from Elasticsearch). By appending the value with the appending of the *n-1* indices, COO format is maintained by having the i^{th} value correspond to the i^{th} indices.

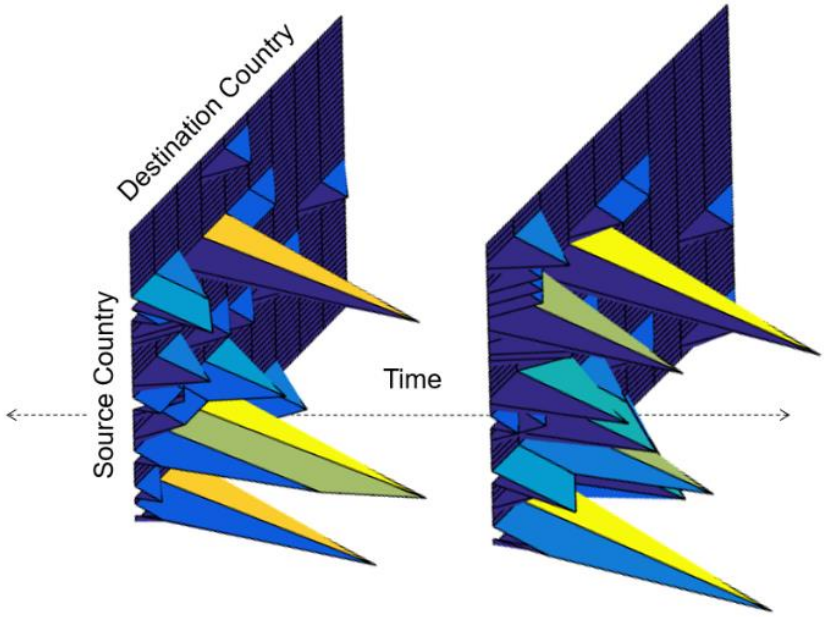


Figure 4.3: InSight2 Sample Tensor

As mentioned earlier in this section, these Elasticsearch results are sorted by the parameters passed by the user. This means it is possible to sum the values returned from the search until a new tuple is returned from the query. For example, in building a tensor of time × source IP address × destination IP address with number of connections as the value, the number of connections between a specific source IP to destination IP can be summed until a new tuple is reached before appending it to the non-zero value list. This prevents look-up in the *n* different lists to determine which value needs to be incremented when a repeated tensor entry is seen.

Once the searches in the specified time range have been completed, the scikit-tensor model is ready to be created. The index list, the non-zero value list, and the shape (which is determined by the number of unique parameters per mode) are stored in a class, creating the sparse tensor representation.

4.4 RED Alert Plug-in within InSight2

As one of the premiere plug-ins of InSight2, RED Alert needed to be ported to work with the InSight2 framework. This means that RED Alert must be able to be initialized from an InSight2 dashboard and provide visualizations back to the user on this dashboard. Kibana has no functionality of starting an outside script from a dashboard, so a non-Kibana dashboard was needed. Kibana also prevents the development of third-party dashboards within their web application, so this required some engineering to work around their limited dashboard features. Thus, the Python library Flask [25] was used to develop the application side of the RED Alert plug-in. This allows for RED Alert script initialization from the InSight2 user and display the results of RED Alert through Kibana visualizations.

Flask is an open-source library in Python that provides server management for web applications. Users create a Flask-based HTML file that interacts with the Flask server management through a Python file. Flask is well-documented with a large userbase allowing for flexible development for any Python project. Since InSight2 and RED Alert were both built in Python frameworks, it made sense to use Flask to create the plug-in using RED Alert. This section summarizes the development of the Flask application created for RED Alert in the context of InSight2.

4.4.1 Frontend

The RED Alert plug-in can be seen in Figure 4.4. The plug-in features a form to enter in the parameters used in RED Alert (date from, date to, time window, thresholds, additional dimensions and tensor value) with a “Refresh” button and a “RED Alert” button. The “Refresh” button refreshes the plots to match the date range specified by the user, and the “RED Alert” button spawns the RED Alert algorithm for the provided parameters. The backend workings of these buttons and the interactions with the RED Alert script are explained in more detail in the next section.

Below the form contains the plots generated for the RED Alert data using Kibana. Since Kibana does not allow for direct manipulation of their dashboards, the iFrame of the RED Alert visualization dashboard is stored in the application’s

HTML file to display the plots. Three different plots were used in this dashboard to aid in the visualization of RED Alert's results. The example below shows the packet loss detection example discussed throughout this thesis. The first plot in the dashboard shows a bar plot with positive and negative values. This displays the "change of the change" through the use of Kibana's difference plot. A positive bar indicates a sudden increase in the change from the previous window (an alert) and the negative bars indicate the "cancelling" of an alert, meaning the network has returned back to normal behavior. The second plot is a line plot of the overall health of the network representing the total change at a given timestamp. Large increases and decreases in this plot correspond to the positive or negative spikes seen in the difference plot above it. Finally, the third "plot" in the dashboard is a table listing the values that were alerted in RED Alert. This table lists the label of the alerted field, the mode that the alert occurred in and the count for how many times it occurred in a time window (shown is a 5-minute time window). This way, the network operator can search on these terms in the Elasticsearch database to gather more information about the root cause and outcome of the alert.

4.4.2 Backend

The backend of the RED Alert plug-in works as follows. The plug-in consists of only one page at the root directory of the application. From there, the Flask app handles both GET requests and POST requests to this page. The GET requests handles reloading the page and fetching the iFrame from Kibana with a default time range. To change the date range of the visualized data in the plug-in, a POST request is committed through the "Refresh" button shown in Figure 4.4. This POST request first ensures that a valid date from and date to are provided by the user. Then the POST request reads the values of the date from and date to fields to re-fetch the iFrame from Kibana that displays the tensor dashboard. This is done through URL manipulation of the iFrame. The Kibana iFrame value contains the date from and date to values within the string. Using a regular expression, the date fields in this string are updated to match Kibana's pattern with the valid dates. Once this URL has been modified correctly, the URL for the iFrame is then passed to the HTML file that overlays the iFrame. Flask allows for static variables in HTML files that can be passed through functions in a Python file, so updating this iFrame string in the HTML file is simple.



Figure 4.4: RED Alert Plug-in

Other than the “Refresh” button to change the date range, the plug-in form also contains a “RED Alert” button that is used to spawn the RED Alert algorithm. This is also done through a POST request, but in this case more validation and processes must be completed than in the “Refresh” case. Not only do the date range fields need to be validated and stored from the POST request, but also the other fields necessary for RED Alert must be collected as well. The user must input a valid integer time window greater than 0. Additionally, the user must provide the desired threshold for each of the granularity levels. The selection of other dimensions besides source, destination and timestamp are also included since exploration of higher dimensionality is currently being looked into. Finally, the user needs to indicate whether to use number of connections (botnet detection) or total packets lost (packet loss detection) within the tensor being used in RED Alert. Once the user inputs these values, pressing the “RED Alert” button spawns a POST request to validate these input values and spawn the RED Alert process. If a value is missing or invalid, the Flask application returns a meaningful error message pointing the user to the issue.

After the POST request is completed, the RED Alert process is started as a new process from the Flask application so that the page can be reloaded while the algorithm is running. This allows for the user to periodically refresh the iFrame with the “Refresh” button to see the results of RED Alert uploaded and displayed in real-time. The RED Alert script uploads the results from each time tick back to Elasitcsearch so that the data can be visualized through Kibana. This process allows for the user to have no coding knowledge as the backend handles all of the parameter passing and checking to execute the script. The user needs to navigate to the RED Alert plug-in page and enter the desired parameters for proper functionality.

5. CONCLUSIONS

Analyzing network trends is an important aspect of network operator's responsibilites. Creating ways to extract and process network flow data in meaningful ways is a challenge due to the volume and high-dimensionality of the data. In this thesis, two main algorithms that use tensors as a means to process and analyze large-scale network traffic were explored. Network data from the GLORIAD provided the means for the analyses.

Using DenseAlert as a network monitoring tool proved to be problematic in large, finely defined tensors. The algorithm begins to report unhelpful results as the datasets become larger. Specifically, DenseAlert failed to provide any meaningful results from datasets representing IP network traffic, although smaller tensors of lower granularity led to more conclusive results. Since IP is the level of traffic that provides the most meaning to network operators, it is important that a network monitoring algorithm provides conclusive results on these datasets. Even though DenseAlert provides quick processing through its re-ordering and update strategies, its shortcomings can be seen through these results.

Thus, RED Alert was created as a tensor-based monitoring tool for network traffic to provide means of anomaly detection at any level of granularity. RED Alert offers a useful alternative to DenseAlert by exploiting the hierarchical nature of network data, filtering out unimportant data and exploring interesting events in more detail. It is a simple, but effective algorithm for recursively filtering and expanding sparse tensors representing time-varying network flow data. We have provided empirical evidence that the algorithm is able to detect and track the modeled activity and attribute the changes to host IPs. This is shown through detection of increases in packet loss and detection of high botnet activity for a network. Through the use of change magnitude, RED Alert can be used on a wide variety of data to provide real-time event detection, be that for interesting benign or critical malicious events. RED Alert also used the idea of slice energy to show that other tensor metrics can be used to monitor the behavior of the network. Both total change and energy provide conclusive results with detailed alerts from the data. RED Alert was shown to be robust in its performance, processing a month of data in merely a few days. RED Alert can further be extended to other problems that have multi-dimensional data that can be separated in a hierarchical fashion.

Although the steps of RED Alert's algorithm are easy and straightforward, creating or implementing tensor libraries in Python proved to be cumbersome. Python provides numerical processing libraries through SciPy and NumPy, but these libraries require extensive memory that becomes problematic when dealing with large, sparse tensors. SciPy and NumPy are useful for their computational capabilities through their wide variety of linear algebra functions, but they are not designed for large multi-dimensional storage. This requires the use of third-party libraries that provide more efficient tensor processing. However, most libraries lacked the functionality desired at the time. As the DenseAlert creators had to do, we created a tensor library that fit the needs of RED Alert through modifications to scikit-tensor, a third-party tensor library on GitHub.

This work ties into the large overall project of InSight2, a network monitoring platform developed here at the University of Tennessee. The tensor library used was modified to work in the context of this platform, so that RED Alert could effectively access and process the data. RED Alert was built as an additional plug-in to InSight2 using Flask to allow for the execution of the algorithm through its backend management and meaningful displays in the frontend. From this, a useful tool for tensor-based network performance monitoring has been created for internal and external use.

REFERENCES

[1] H.A.D.E. Kodituwakku. InSight2: An Interactive Web Based Platform for Modeling and Analysis of Large Scale Argus Network Flow Data. Master's thesis, University of Tennessee, 201B.

[2] D. Phan, J. Gerth, M. Lee, A. Paepcke, and T. Winograd. Visual analysis of network flow data with timelines and event plots. In John R. Goodall, Gregory Conti, and Kwan-Liu Ma, editors*, VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security*, pages 85–99, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[3] R. A. Becker, S. G. Eick, and A. R. Wilks. Visualizing network data. IEEE Transactions on Visualization and Computer Graphics, 1(1):16–28, March 1995.

[4] C. Van Loan et al. Future Directions in Tensor-Based Computation and Modeling. NSF Workshop, 2009.

[5] O. Alter, P.O. Brown, and D. Botstein. Generalized singular value decomposition for comparative analysis of genome-scale expression data sets of two different organisms. *Proceedings of the National Academy of Sciences*, 100(6):3351–3356, 2003.

[6] L. De Lathauwer, B. De Moor, and J. Vandewalle. A Multilinear Singular Value Decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[7] K. Maruhashi, F. Guo, and C. Faloutsos. MultiAspectForensics: Pattern Mining on Large-Scale Heterogeneous Networks with Tensor Analysis. In *2011 International Conference on Advances in Social Networks Analysis and Mining*, pages 203–210, July 2011.

[8] K. Shin, B. Hooi, J. Kim, and C. Faloutsos. DenseAlert: Incremental Dense-SubTensor Detection in Tensor Streams. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1057–1066. ACM, 2017.

[9] G. Cole, and N. Bulashova. "GLORIAD: a ring around the Northern Hemisphere for science and education connecting North America, Russia, China, Korea and Netherlands with advanced network services", 2005.

[10] T.G. Kolda and B.W. Bader. Tensor Decompositions and Applications, SIAM Review 2009 51 no. 3 455-500.

[11] S. Papadimitriou, J. Sun, and C. Faloutsos. 2005. Streaming pattern discovery in multiple time-series. In Proceedings of the 31st international conference on Very large data bases (VLDB '05). VLDB Endowment 697-708.

[12] E.W. Weisstein. "Frobenius Norm." From MathWorld--A Wolfram Web Resource. Retrieved from http://mathworld.wolfram.com/FrobeniusNorm.html, 17 October 2018.
[13] QoSient Argus. Retrieved from http://qosient.com/argus/, 4 September 2018.
[14] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-scale Botnet Detection and Characterization. In *Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets*, HotBots'07, pages 7–7, Berkeley, CA, USA,2007. USENIX Association.
[15] Zeus Tracker. Retrieved from https://zeustracker.abuse.ch/, 8 August 2018.
[16] Palevo Tracker. Retrieved from https://palevotracker.abuse.ch/, 8 August 2018.
[17] Feodo Tracker. Retrieved from https://feodotracker.abuse.ch/, 8 August 2018.
[18] A.V. Goldberg. 1984. Finding a maximum density subgraph. Technical Report.
[19] "GLORIAD InSight". Retrieved from http://insight.gloriad.org/, 4 September 2018.
[20] Elasticsearch. Retrieved from http://www.elastic.co/, 12 October 2018.
[21] A. Kodituwakku, J.T. Liso, and J. Gregor. Insight2: An Interactive Web Based Platform for Modeling and Analysis of Large Scale Argus Network Flow Data. FloCon, 2018 (Tuscon, AZ).
[22] NumPy. Retrieved from http://www.numpy.org/, 18 October 2018.
[23] TensorLy. Retrieved from http://tensorly.org/stable/index.html, 18 October 2018.
[24] Scikit-Tensor. Retrieved from https://github.com/mnick/scikit-tensor, 18 October 2018.
[25] Flask. Retrieved from http://flask.pocoo.org/, 12 October 2018.

VITA

Gerald Liso was born in Hackensack, New Jersey in April 1996, moving to Franklin, Tennessee in October 1997. J.T. graduated with a Bachelors of Science in computer science in December of 2017, joining the University of Tennessee's 5-year BS/MS program in computer science. As an undergraduate, J.T. worked as an intern at Cisco and as a research assistant both at Oak Ridge National Lab and the University of Tennessee. As a graduate student, J.T. continued working as a research assistant at the University of Tennessee developing the tensor plug-in model of InSight2 described in this thesis. J.T. will graduate with a Masters of Science in December 2018 and is excited to begin working in the technology industry.