MIL-STD-1553 Intrusion Detection

Daniel Kim

September 21, 2018

Abstract

The goal for this project was to introduce an efficient method for detecting anomalies within the streaming data logs of an MIL-STD-1553 multiplex data bus system. Our current approach focuses on the incoming command words and the patterns in which they create to define a model creation pipeline comprising of training, testing, and exporting phases. Once exported, the project can be used to remotely or locally read data for real time data analysis and anomaly detection.

1 Introduction

The MIL-STD-1553 Bus Protocol outlines the method of communication and electrical interface requirements for subsystems connected to the data bus [1]. Using a Bus Controller (BC), the bus is able to transmit and receive byte commands to delegate functionality to multiple devices or Remote Terminals (RT) within the distributed system. As effective as the bus protocol is, an adversary may still be able to interfere with system processes through the replacement and deletion of benign data or through the injection of anomalous data. This is because the MIL-STD-1553 was developed during a period in which the area of cyber-security was a relatively new concept and attack threats regarding the MIL-STD-1553 Protocol had not been explored [6]. Now, the MIL-STD-1553 Protocol is a military standard whose current revision MIL-STD-1553B has become one of the basic tools and most widely applied systems as used today by the United States Department of Defense for integration in weapon systems [1].

Our 3-Phased Anomaly Detection Pipeline was thereby inspired by the security threat imposed on the MIL-STD-1553 Protocol and created in order to counteract the adversary's malicious attempts at impairing systems implementing the protocol. The first Training phase extracts preexisting patterns within a bus log representing the expected normal behavior of a genuine 1553 system. The second Testing phase is then used to determine whether the performance of the anomaly detection model built using the data extracted from the training phase is satisfactory. The final Exporting phase can then be used to create a project package to be deployed to begin receiving real-time streaming data for anomaly detection. The package can be installed either locally as part of a Bus Monitor (BM) module or remotely in a server that contains access to the data being logged in the BM.

2 Related Works

Due to the lack of focus in cyber-security measures against possible attack threats during the development of the MIL-STD-1553, there has been some research that have been conducted in order to enhance the security of the system. Proposed solutions include the creation of cryptographic components, a separation kernel, monitoring modules with automatic recovering services [6], and the replacement of the MIL-STD-1553 standard altogether [1]. However, implementing any of the mentioned physical hardware solutions is cost prohibitive since it requires altering various components of already implemented systems. Software solutions include the implementation of firewalls, intrusion and malware detection, data leakage prevention, and access control which are also unsuitable for the 1553 communication bus since they will require a considerable amount of adaptation and reconfiguration to the Operating Systems and communication protocols that have already been implemented. Therefore, any solution that attempts to protect against attack threats and attack vectors must be lightweight enough to be cost sensible and easily implemented while remaining flexible and adaptable enough to work on a wide range of 1553 systems.

2.1 Current State-of-the-Art

The current state-of-the-art uses a Sequence-Based Anomaly Detection Method for identifying cyber-attacks [6] on the MIL-STD-1553 Bus. The most important factor that skewed the development of their project is the importance of fulfilling the conditions of a lightweight, highly flexible and adaptable solution. Ease of implementation is achieved through their requirement of solely needing access to a continuous data stream of messages being transmitted over the bus. This allows it to be easily integrated as a Bus Monitor module without the need of altering any existing modules within the bus. There is no need for any hardware implementation nor software configuration since the Bus Monitor already listens to all messages and subsequently collects data from the data bus [1]. High flexibility and adaptability are achieved through the focus on following machine learning techniques during the design of their project model.

Before explaining the definition of what they consider to be an anomaly within the monitored bus, we must first examine the Data Link Layer which contains the Word and Message Protocols of the MIL-STD-1553 Bus. The 1553 network makes use of a time division, half duplex communication system where all transmissions run along a single cable [5]. This enforces that only one computer terminal (BC or RT) can talk/transmit data at any given time and the other computers will be listening/receiving the data currently being transmitted. A command/response system is required by the 1553 protocol where each RT must respond back to the BC that initiated the command through the use of a status message within a 4 to 12 microsecond window [2]. An adversary may utilize these windows of idle times to inject anomalous messages in the data bus. This can be achieved through the insertion of malicious code during the early phases of the component's life cycle or through the interception and modification of the component during its exportation in the supply chain [5]. Malicious code may additionally find its way into the system even after it is implemented during its maintenance period since the system may require a wireless communication or physical connection to a data medium such as a CD/DVD or a USB which may be infected [6]. The injection of anomalous messages, however, is not completely flawless as it leaves a significantly obvious trace which is detected by the Sequence-Based Anomaly Detection Method since it must be inserted in between already defined and consistent time intervals. Any disruption of consecutive messages and their normal time cycles should alert a system of anomalous messages.

A training phase is utilized using the logged activities of normal bus operations in order to conform their detection method around a specific system. A Markov Chain Model is used to create a basis for the detection mechanism to represent the normal activities of the monitored bus. Two separate Markov models are designed, one for periodic messages and another one for aperiodic messages. A message is classified as periodic if it is sent at fixed time intervals and classified as aperiodic if they are not sent in fixed time cycles due to the fact that these messages are event driven. They have also defined the term "Major Frame" which is a "predefined time frame in which all periodic messages are transmitted at least once... it is derived from the periodic message with the longest time cycle" [6]. After creating the two Markov Chain Models, state probability transitions between messages are calculated between only the periodic messages since their model does not allow aperiodic messages to have a time cycle. An anomaly threshold is then calculated as the minimum probability between two consecutive message sequences. A transition between messages is deemed anomalous if the probability between these two consecutive messages are less than the anomaly threshold. After the training phase, their model can be directly used to provide real time anomaly detection over the bus.

2.2 Limitations of the Current State-of-the-Art

Although the Sequence-Based Anomaly Detection Method presented is able to successfully distinguish between anomalous and benign messages with a very low false alarm rate, it is only able to do this using the command word structure in the 1553 network. It is unable to detect anomalies within the data words and status words structures. Data words are simply the information being passed between computers in a 1553 network and status words are the responses generated from the RTs to the BC as part of the command/response protocol. Their design is unable to extract features from these two word structures and therefore unable to classify them as anomalous or benign. Furthermore, their anomaly detection method is solely focused on the detection of injected messages. Two additional scenarios are not accounted for which include an attempt of replacing and deleting a message within a major frame. It is very critical to note that any malware that is able to precisely delete and replace a message with the correct time differences within a major frame would not be detected by the Sequence-Based Anomaly Detection Method. This is due to the fact that their algorithm examines the time differences in between messages and any successful replacement of a message with the same time difference will not be deemed as anomalous.

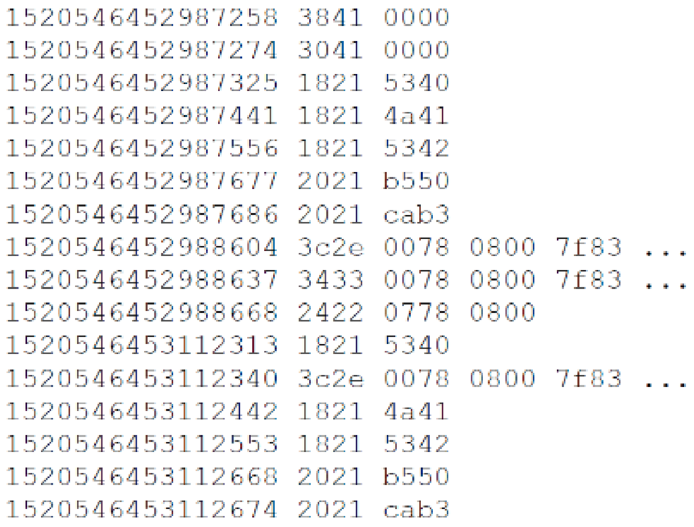
Their proposed solution also requires a large amount of data to be stored and retrieved on a continuous basis for comparisons and probability calculations. The amount of information that must be stored would increase exponentially as the number of possible command words within a 1553 network increases in a complex system due to the nature of their sequence mining algorithm and message to message transition calculations. This would require additional hardware to be installed abandoning the requirements of a lightweight system as previously mentioned.

2.3 Method Comparison

As for our implementation, not only do we require a proposed solution to be able to be easily implemented and cost sensible to be considered lightweight, but also require the solution to prevent any installation of additional memory modules in the system. Our Minimum Scoring Graph Based Detection algorithm (herein MSG algorithm) follows the same principles established by the Sequence-Based Anomaly Detection Method in that it solely requires access to the continuous stream of data being transmitted over the MIL-STD-1553 bus. Therefore, our method is able to also be optionally integrated as a simple BM module. We further increase the flexibility and reduce the reliance of additional memory of our MSG algorithm through the Exporting Phase where the BM has the option to, instead of directly computing through its own local computing system, transfer responsibility to a remote server where memory and computational power is of no issue. We were able to even further reduce the usage of memory through the calculation of pattern transition probabilities instead of message to message transition probabilities. The MSG algorithm does not rely on the amount of command words existing in a complex system, but rather on the patterns created through the calculation of time-gaps within a training bus log. Therefore, the required amount of memory stays constant as a 1553 system increases in complexity and the amount of messages that can be transmitted within it. For example, out of a bus log containing close to 1 million lines of data, only 65 patterns and 0 non-patterns have been found. A transition matrix created out of these requires less than a megabyte of data.

High flexibility and adaptability are achieved through the adherence of machine learning practices. We also use training and testing phases as part of our 3-Phased Anomaly Detection Pipeline. During our training phase, our model is able to quickly conform to the existing 1553 system using a training dataset consisting of data logs reflecting the expected normal behavior of the system. Instead of focusing on the single aspect of message injections, we have built our algorithm to be able to also handle message replacements along with future plans to implement data anomaly detection. This was achieved by looking at a collection of messages as a repeatable pattern instead of analyzing individual messages.

3 The Dataset



```
1520546452987258 3841 0000
1520546452987274 3041 0000
1520546452987325 1821 5340
1520546452987441 1821 4a41
1520546452987556 1821 5342
1520546452987677 2021 b550
1520546452987686 2021 cab3
1520546452988604 3c2e 0078 0800 7f83 ...
1520546452988637 3433 0078 0800 7f83 ...
1520546452988668 2422 0778 0800
1520546453112313 1821 5340
1520546453112340 3c2e 0078 0800 7f83 ...
1520546453112442 1821 4a41
1520546453112553 1821 5342
1520546453112668 2021 b550
1520546453112674 2021 cab3
```

Figure 1: The log from an MIL-STD-1553 Bus

The above figure shows the structure of the incoming data from an MIL-STD-1553 bus. The first value in each line is an Epoch timestamp which indicates the date and time, in microseconds, in which the corresponding data was sent. The second value is a 16-bit hexadecimal command word immediately followed by 1 to 32 16-bit hexadecimal data words [4].

4 Theory

Here, we familiarize the reader with a brief summary of the formulas and concepts that we have used for this project.

4.1 Time-Gaps

We utilize time-gaps to determine the beginning and end of a command grouping, a series of consecutive commands. Time-gaps are significant time differences between the occurrence of a command and its subsequent one which allows us to break a bus log into a series of command groupings. The time-gap threshold is calculated using the winsorized mean difference between commands over the entire bus log given during the training phase. The winsorized mean is used to compensate for outlier values while still achieving a reasonable estimate of central tendency. [7] Depending on the limit used to calculate the winsorized mean, the pattern extraction process can either hinder or aid the anomaly detection process.

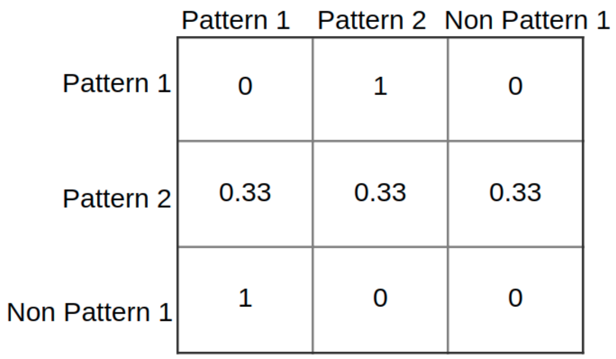
4.2 Patterns, Sub-Patterns, and Non-Patterns

A pattern is any command grouping that occurs more than once while a non-pattern is a command grouping which has been observed to occur only once throughout the entire bus log given during the training phase. A sub-pattern is a pattern that occurs only after we have split a command grouping and have either found a matching sub-pattern or previously found pattern. Each pattern is assigned a unique Pattern Symbol in the format of "PID_#" and each non-pattern a unique Non-Pattern Symbol "NID_#"

4.3 N-Grams

N-grams, a concept found in Natural Language Processing, is a sequence of consecutive 'values'. These values can range from complete words delimited by spaces to sentences delimited by periods. In our application, we consider n-gram values in terms of consecutive command groupings. For example, in a command grouping of c_1, c_2 and c_3, a 2-gram would be c_1, c_2 and c_2, c_3.

4.4 Transition Matrix



	Pattern 1	Pattern 2	Non Pattern 1
Pattern 1	0	1	0
Pattern 2	0.33	0.33	0.33
Non Pattern 1	1	0	0

Figure 2: A example of a transition matrix

A transition matrix contains the probabilities of transitioning from one pattern or non-pattern to another. For example, according to Figure 2 all the occurrences of Pattern 1 directly transition to Pattern 2.

4.5 State Diagram

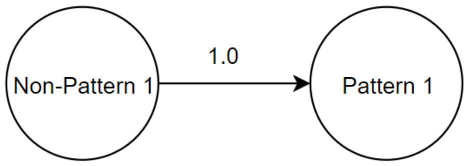


Figure 3: A example of a State Diagram

A state diagram is simply a visual representation of the transition matrix using nodes to represent patterns and non-patterns and valued arrows that correspond to the probability of transitioning from one pattern/non-pattern or state to another.

5 Minimum Scoring Graph Based Anomaly Detection

As previously mentioned, any proposed solution that provides enhanced cyber-security measures against attack threats for the MIL-STD-1553 Bus must be lightweight, highly flexible, and highly adaptable. We propose a Minimum Scoring Graph Based Anomaly Detection algorithm that fulfills these requirements as explained in the following subsections.

5.1 Phase 1: Training

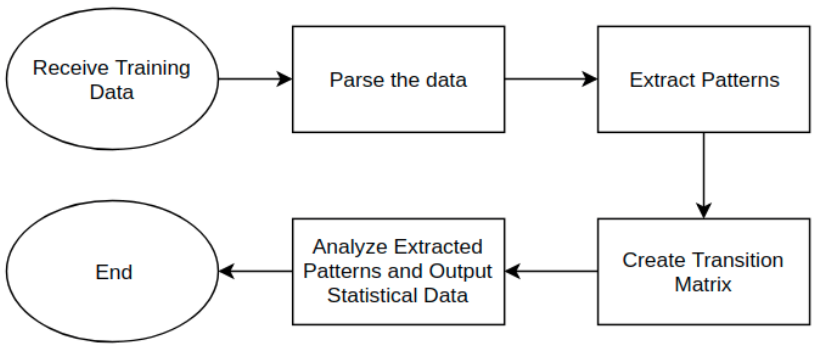


Figure 4: Training Phase Flowchart

Training begins by first parsing the bus log and creating a timestamp, command word, and data word list whose index correspond to the order in which their row was read in. Currently, our proposed pipeline is solely focused on the command words and the time intervals in which the command words are processed. A time-gap threshold value is utilized to form command groupings throughout a given training bus log. For example, assume that c_1, c_2, c_3 are commands. If the time difference between the occurrence of c_2 and c_3 is larger than the time-gap threshold, then c_1 and c_2 are considered a command grouping. This is done to allow us to eventually define Patterns/Non-Patterns or the states for a Markov Model. [3] It is important to note the winsorized-limit hyperparameter which controls the manner in which pattern extraction process will occur. This value controls how many outlying time-difference values will contribute to calculating the mean or time-gap for the entire training bus log. The lower this value is, the more do outliers affect the time-gap value, and the higher it is, the less they affect the overall time-gap value. Although singular, this value has a significant effect on the performance of the anomaly detection model as it is shown in the figure below. The winsorized-limit hyperparameter determines how many patterns and transitions we are able to record from the training bus log. In our dataset, we have found that a winsorized-limit value of 0.19 achieves a false positive rate of 0.029%.

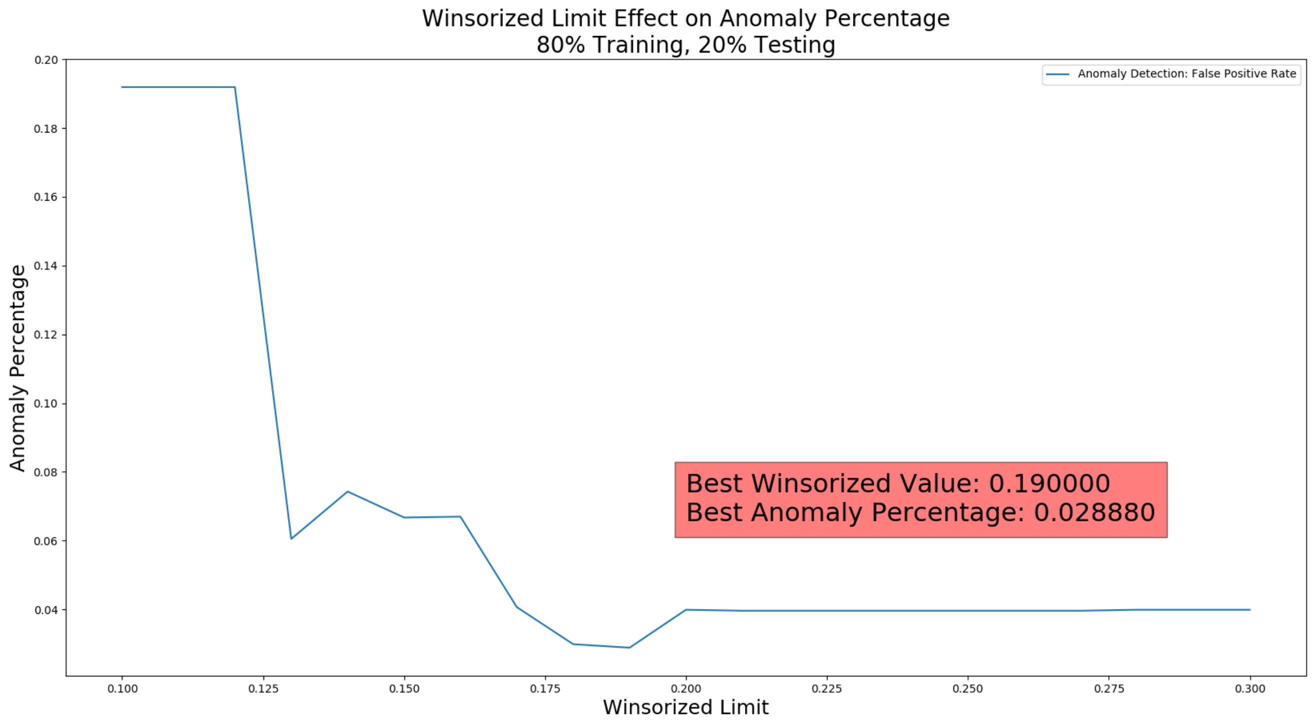


Figure 5: Winsorized Limit Performance Graph

Once the time-gap threshold is calculated, we split the entire bus log into a series of consecutive command groupings. We then perform a single pass through the entire list of command groupings to find repeating patterns within it. Any command grouping that occurs more than once is considered a pattern and is assigned a unique *Pattern_ID* symbol while also recording the following information: the occurrence count of the pattern, the index location and timestamp values corresponding to the beginning of each pattern occurrence, and the average time between occurrences of the pattern. After the first pass through, any pattern that does not occur at least twice is put into a non-pattern list.

Now that we have a list of command groupings that occur only once through the whole bus log, we begin the sub-pattern routine. We first iterate through the entire non-pattern list to find the largest command grouping within it and designate the length subtracted by one as the current *length_restriction* variable. For example, if the largest non-pattern contains 10 commands within it, *length_restriction* is initialized to the value 9. We then attempt to find repeating occurrences of *n-grams*, where n is equal to the current value of *length_restriction*, through the non-pattern list by splitting any command grouping larger or equal to the value of *n*. If an *n-gram* matches a previously found pattern, we simply update the information already logged for the pattern. If the *n-gram* matches another *n-gram* and not a previously found pattern, we assign it a new *Pattern_ID* symbol while recording the same set of information as we did for the patterns that we have found before. After running through all of the non-patterns and attempting to find repeating *n-grams*, the *n* value is then decremented, and the process is repeated until either we do not have any non-patterns left or the value of n becomes 0.

Any remaining non-patterns after the previous algorithm has terminated is assigned an *Non_Pattern_ID* symbol and the occurrence index and corresponding timestamp of the first element within the non-pattern is logged. Throughout this algorithm, we have attempted to continuously provide the best methods to find patterns and non-patterns as efficiently as possible through the reduction of sorts, loops, and string comparisons while implementing methods such as hashing and custom list searching algorithms. We have found that it only takes a few seconds for the algorithm to find all of the patterns and non-patterns within a significantly large bus log containing one million lines.

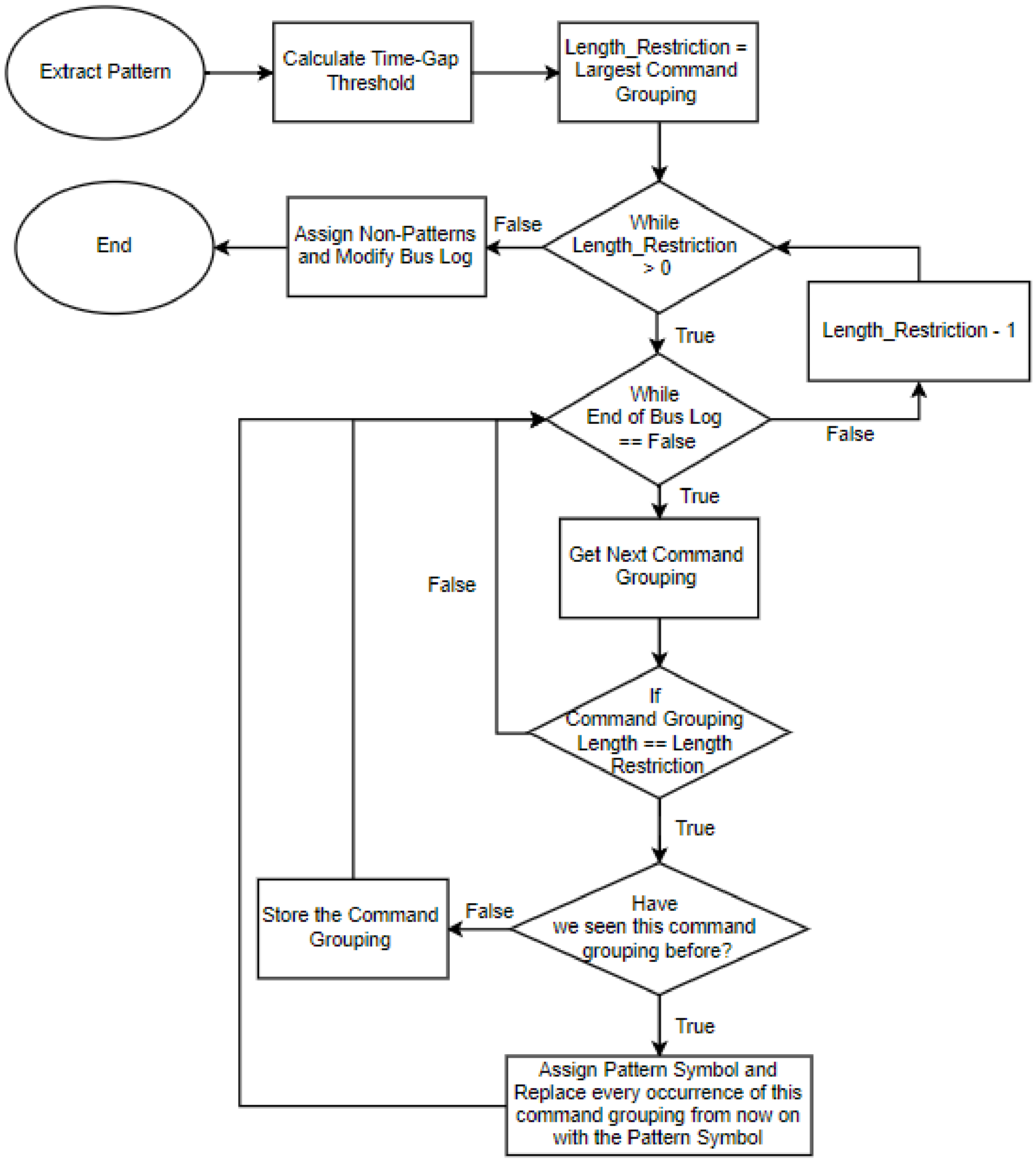
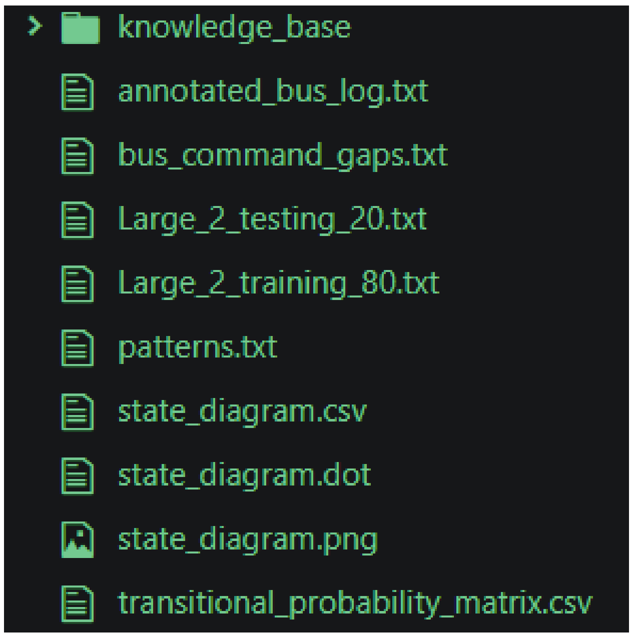

Extract Pattern
Calculate Time-Gap Threshold
Length_Restriction = Largest Command Grouping
While Length_Restriction > 0
False
Assign Non-Patterns and Modify Bus Log
End
True
Length_Restriction - 1
While End of Bus Log == False
False
True
Get Next Command Grouping
False
If Command Grouping Length == Length Restriction
True
Have we seen this command grouping before?
False
Store the Command Grouping
True
Assign Pattern Symbol and Replace every occurrence of this command grouping from now on with the Pattern Symbol

Figure 6: Pattern Extraction Flowchart

A modified bus log is created through the above implementation where each command grouping is replaced with their respective Pattern and Non-Pattern Symbols. We can now create the representation of our Markov Model using a Transition Matrix by moving through the modified bus log and documenting each transition. Once it is created, we collect all of the information that we have gained throughout the Training Phase process, serialize the objects, and place them into a folder titled *knowledge_base* inside of the user specified save directory or default project folder. This folder contains all of the information that we need in order to continue onto the Testing Phase. Other supplementary files created throughout the ongoing process of the Training Phase are also located inside of the user specified directory as shown in Figure 7.



```
knowledge_base
annotated_bus_log.txt
bus_command_gaps.txt
Large_2_testing_20.txt
Large_2_training_80.txt
patterns.txt
state_diagram.csv
state_diagram.dot
state_diagram.png
transitional_probability_matrix.csv
```

Figure 7: Training Phase output

The *annotated_bus_log.txt* demonstrates the start and end of an identified pattern. The file *bus_command_gaps.txt* is the original bus log file containing lines where a time-gap has been found. *Large_2_testing_20.txt* and *Large_training_80.txt* are files created by splitting the original training bus log into their respective training and testing portions. The file *patterns.txt* contains all of discovered patterns throughout the training bus log alongside some statistical informations about each pattern. All three *state_diagram.extension* files contain different representations of the state diagram. The final *transitional_probability_matrix.csv* is simply the transitional matrix that we have created during the Training Phase for the user's convenience.

5.2 Phase 2: Testing

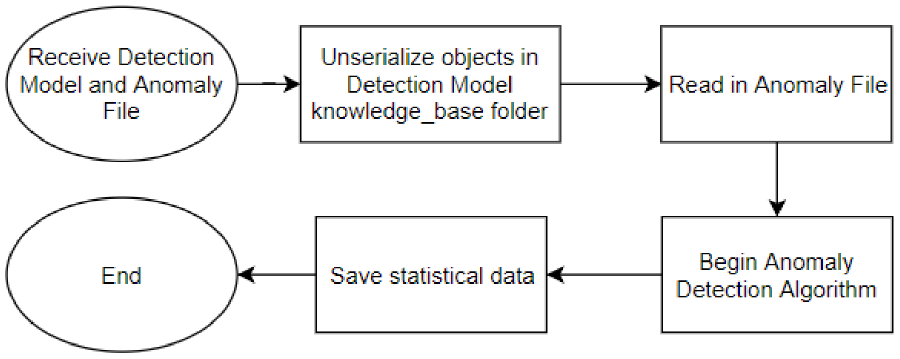

Receive Detection Model and Anomaly File
Unserialize objects in Detection Model knowledge_base folder
Read in Anomaly File
Begin Anomaly Detection Algorithm
Save statistical data
End

Figure 8: Testing Phase Flowchart

Above we provide the implementation of the Testing Phase. Once the executable program receives a properly configured detection model and anomaly file, it will begin the Testing Phase. Objects created and stored in the detection model's *knowledge_base* are unserialized and the given anomaly file is read in line by line to simulate continuously streaming data.

It is important to note the criteria in which our detection algorithm will label a set of command groupings as anomalous. Assume $S_i = \{c_0, c_1, ..., c_n\}$ where S_i defines the i^{th} command grouping found within the test bus log. If S_i is not a pattern or non-pattern as discovered during the Training Phase, S_{i-1}, S_i, S_{i+1} are defined as anomalous. However, if S_i is a previously discovered pattern or non-pattern, but the transition from $S_i \rightarrow S_{i+1}$ is invalid, then S_{i-1}, S_i, S_{i+1} are also defined as anomalous. Now that the two criteria have been discussed, we are now able to demonstrate the algorithm for detecting any combinations and repetitions of the two above anomaly conditions.

Consider the case in which S_1 is an unknown pattern, this would obviously mean that the transition from $S_0 \rightarrow S_1$ is invalid. After adding them to an *anomaly_list* we then search to find the next valid transition, but we happen to find that S_2 is also unknown. Since both S_0 and S_1 have already been added to the *anomaly_list*, we simply append S_2 to the list also. If S_3 and S_4 are known patterns but the transition from $S_3 \rightarrow S_4$ is invalid, we also add these two into the *anomaly_list*. Finally, we find a valid transition from $S_5 \rightarrow S_6$, but we know that since S_4 is an anomaly, that the transition from $S_4 \rightarrow S_5$ is anomalous and therefore add S_5 into the *anomaly_list*. See that once an anomaly is found, we continue until we find a valid transition of patterns/non-patterns. The *anomaly_list* is then sent to the Minimum Scoring Graph Based Algorithm.

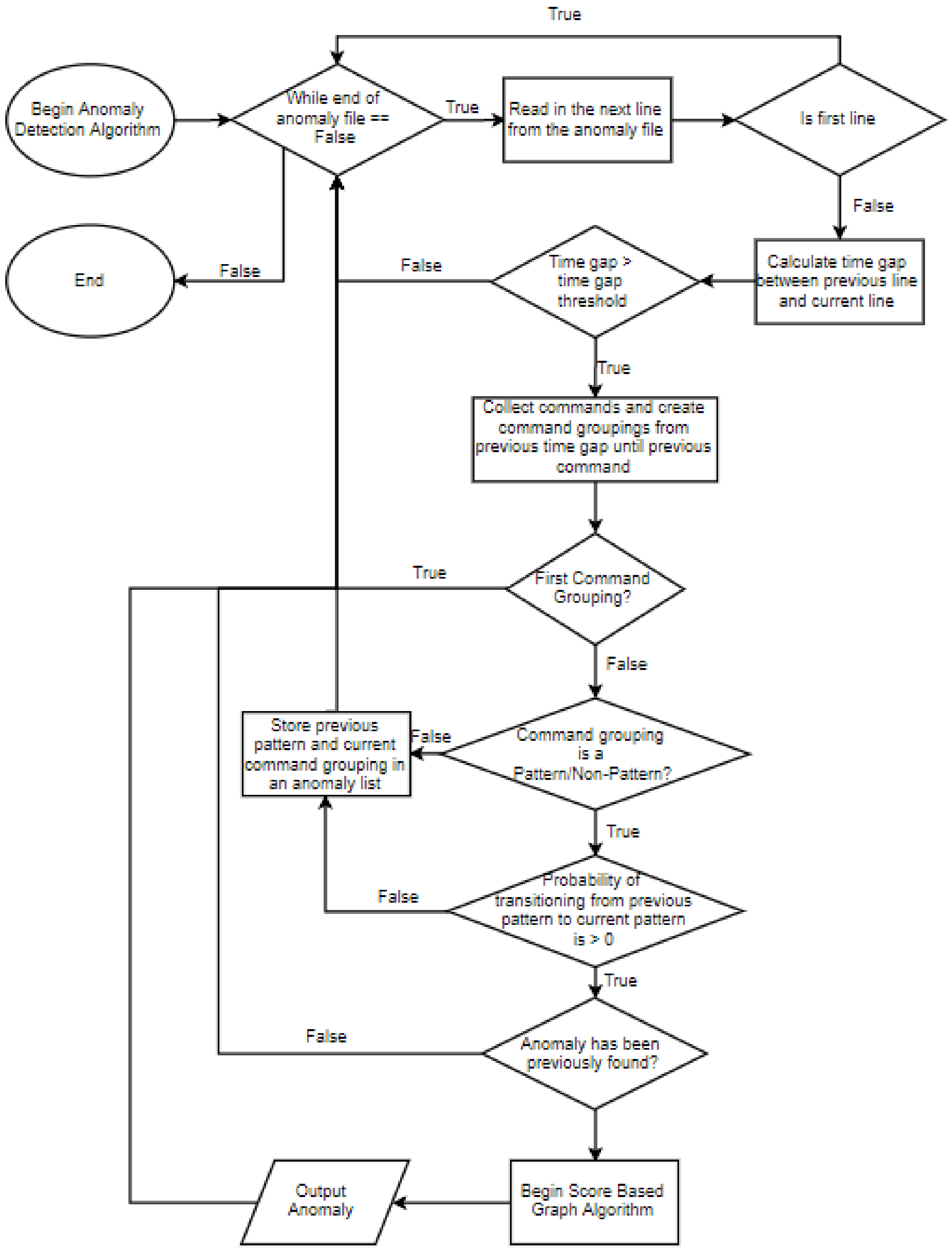

True
Begin Anomaly Detection Algorithm
While end of anomaly file == False
True
Read in the next line from the anomaly file
Is first line
False
End
False
False
Time gap > time gap threshold
Calculate time gap between previous line and current line
True
Collect commands and create command groupings from previous time gap until previous command
True
First Command Grouping?
False
Store previous pattern and current command grouping in an anomaly list
False
Command grouping is a Pattern/Non-Pattern?
True
False
Probability of transitioning from previous pattern to current pattern is > 0
True
False
Anomaly has been previously found?
Output Anomaly
Begin Score Based Graph Algorithm

Figure 9: Anomaly Detection Flow Chart

The Minimum Scoring Graph Based Algorithm allows us to find the most probable set of consecutive patterns and non-patterns that have been altered within an *anomaly_list*. Figure 10 shows a brief overview of how the algorithm works.

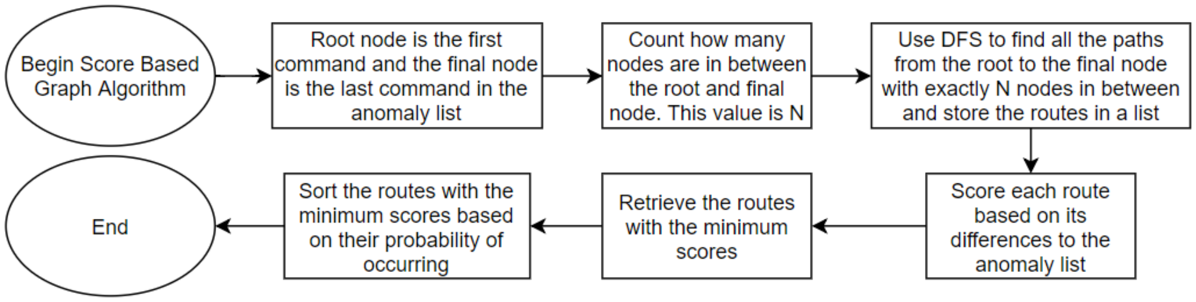

Begin Score Based Graph Algorithm
Root node is the first command and the final node is the last command in the anomaly list
Count how many nodes are in between the root and final node. This value is N
Use DFS to find all the paths from the root to the final node with exactly N nodes in between and store the routes in a list
Score each route based on its differences to the anomaly list
Retrieve the routes with the minimum scores
Sort the routes with the minimum scores based on their probability of occurring
End

Figure 10: Score Based Graph Flowchart

Continuing on from the previous example, suppose that we have $anomaly\_list = A$ where $A = \{S_0, S_1, S_2, S_3, S_4, S_5\}$ and $N = length(A) - 2$, the number of all of the command groupings in between the first and last command groupings. S_0 is classified as the *root node* and S_5 is classified as the *final node* and everything else in between are simply classified as *nodes*. Once we have all of these defined, we begin a recursive Depth First Search Algorithm which attempts to find all of the routes from the *root node* to the *final node* after exactly N *nodes*. The DFS algorithm is slightly altered to allow the visiting of the same node multiple times. Once we have established all of the possible routes from the *root node* to the *final node* after exactly N steps, we begin to score each individual route based on their differences as compared to *anomaly_list*.

Before scoring each route, we first vectorized $anomaly\_list = X \in R^{NxF}$ where $N = length(anomaly\_list)$ and $F = max(command\ grouping\ length\ within\ anomaly\_list)$ into a Matrix and each command corresponds to an integer value. Then, each route is also vectorized where $route\_matrix = Y \in R^{NxF}$ and subtracted from Matrix X, outputting a difference Matrix $Z \in R^{NxF}$. All of the non-zero values within the subsequent Matrix Z define an element that differ within route Matrix Y and *anomaly_list* Matrix X. These non-zero values are the scores of the current route Matrix Y. The probability of occurrence for the routes containing the minimum scores are then calculated using the Transition Matrix. Finally, we sort the routes based on the route with the greatest probability of occurring to the least and output them to the console window along side the original *anomaly_list*.

Once the Testing phase is finished, the user will be presented with some statistical data about the process that had just run. Once the user is satisfied with the results of the Testing Phase, they may continue on to the final Exporting Phase.

5.3 Phase 3: Exporting

Once the project has been exported, a package will be created in the location specified by the user. A server will be created on port 7000 on the machine that executes the *main.py* file of the package that will allow a system to start streaming data to the port and begin the anomaly detection process.

6 Analysis

Training	Testing	Possible Anomalies Found
80% 216,000 Lines	20% 53,999 Lines	0.00394% 213 Possible Anomalies
85% 229,500 Lines	15% 40,499 Lines	0.00222% 90 Possible Anomalies
90% 243,000 Lines	10% 26,999 Lines	0.00089% 8 Possible Anomalies

Table 1: Results

Above, we are able to see the results of the Training Phase depending on how the input bus log has been split.

References

[1] ILC Data Device Corporation. *MIL-STD-1553 Designer's Guide.* ILC Data Device Corporation, 6th edition, December 2003.

[2] Alta Data Technologies LLC. Mil-std-1553 tutorial and reference. Alta Data Technologies LLC.

[3] Markov chain. Markov chain — Wikipedia, the free encyclopedia, 2018. [Online; accessed 08-July-2018].

[4] MIL-STD-1553. Mil-std-1553 — Wikipedia, the free encyclopedia, 2018. [Online; accessed 08-July-2018].

[5] Thuy D. Nguyen. Towards MIL-STD-1553B Covert Channel Analysis. January 2015.

[6] O. Stan, Y. Elovici, A. Shabtai, G. Shugol, R. Tikochinski, and S. Kur. Protecting Military Avionics Platforms from Attacks on MIL-STD-1553 Communication Bus. In *eprint arXiv:1707.05032*, July 2017.

[7] Winsorized mean. Winsorized mean — Wikipedia, the free encyclopedia, 2018. [Online; accessed 08-July-2018].