Title
Anomaly Detection for the Science DMZ

Author
Gegan, Ross Kieran

Publication Date
2021

Peer reviewed|Thesis/dissertation

Anomaly Detection for the Science DMZ

By

ROSS K. GEGAN
DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Dipak Ghosal

Matt Bishop

Karl Levitt

Committee in Charge

2021

Contents

Abstract v

Acknowledgments vi

Chapter 1. Introduction 1

1.1. Science DMZ 1

1.2. Outline of Contributions 3

Chapter 2. Anomaly Detection Using System Performance Data 8

2.1. Introduction 8

2.2. Science DMZ 10

2.3. System Performance Metrics 10

2.4. Machine Learning for Anomaly detection 11

2.5. DBSCAN 12

2.6. Experimental Setup 12

2.7. Evaluation and Discussion 15

2.8. Conclusions and Future Work 17

Chapter 3. Insider Attack Detection Using System Performance Data 19

3.1. Introduction 19

3.2. Background and Related Work 21

3.3. Data Obfuscation Attack Scenario 23

3.4. Data Sabotage Detection 25

3.5. Experimental Setup 28

3.6. Results and Evaluation 30

3.7. Conclusions and Future Work 33

Chapter 4. Covert Timing Channel Detection 36
4.1. Introduction 36
4.2. Timing Channels and Detection 39
4.3. Corrected Conditional Entropy 43
4.4. System Design and Experimental Setup 45
4.5. Results and Discussions 49
4.6. Related Work 53
4.7. Conclusion and Future Work 55

Chapter 5. Denial of Service Detection on High-Throughput Research Networks 57
5.1. Introduction 57
5.2. Characterization of DDoS Attacks 59
5.3. DDoS Detection Algorithms 62
5.4. Performance Evaluation 64
5.5. Related Work 71
5.6. Conclusion and Future Work 72
5.7. Sharing Statement 72

Chapter 6. Unusual Protocol Monitoring with Zeek 74
6.1. Introduction 74
6.2. Background and Related Work 76
6.3. Zeek Plugins and Scripts 78
6.4. Anomaly Detection 80
6.5. Conclusions and Future Work 84

Chapter 7. Conclusion 85
7.1. Anomaly Detection Using System Performance Data 85
7.2. Insider Attack Detection Using System Performance Data 87
7.3. Covert Timing Channel Detection 88
7.4. Unusual Protocol Monitoring with Zeek 88
7.5. Conclusion 89

Publications 91

Bibliography 92

Ross K. Gegan
Computer Science

Abstract

The primary focus of this dissertation is on evaluating anomaly detection for Science DMZ networks. Compared to other anomaly detection problems, this introduces some unique challenges. First, if real-time threat detection and response is desired, there is a smaller time window for reaction in high speed networks such as Science DMZs. Anomaly detection, and real-time detection in particular, is already a challenging problem, but in the case of high speed networks it becomes even more difficult. Different approaches are required in order to keep up at high rates, such as either simplifying the detection methods, or finding creative means of optimizing existing methods, either through different hardware or modifying the algorithms. In some cases the best option is finding an entirely new detection technique, utilizing new or previously neglected metrics. Determining the specific techniques best suited for a given environment is another challenge. The ideal methods for anomaly detection will vary depending on the particular network and how it is used. Science Demilitarized Zone (DMZ) networks are generally made to perform a limited range of tasks, primarily facilitating high speed data transfers between research sites. Therefore, the behavior seen will be more predictable compared to more general purpose networks, which helps allow for more practical anomaly detection. This work contains research into different methods of anomaly detection on Science DMZs, providing evaluations of new real-time detection methods and tools for more effective monitoring.

Acknowledgments

First, I would like to express my extreme gratitude to Dipak Ghosal for guiding and mentoring me throughout my PhD, as well as to Matt Bishop and Karl Levitt for their guidance and serving on my dissertation committee. I would also like thank Michael Dopheide and Sean Peisert, along with all the others I've worked together with over the years. Thank you to my family for having my back through it all, and thank you to everybody else who has helped me along the way.

CHAPTER 1

Introduction

1.1. Science DMZ

The Science DMZ is a model designed for scaling scientific research, ensuring reliable performance and high rate data transfers [**1**] between sites. Though the DMZs differ depending on their purpose, DMZs share certain key features. Figure 6.1 presents a typical science DMZ configuration and its components. A science DMZ is typically connected with a site at the network perimeter, through a border router linking the Science DMZ and the site. Assuming their security policies allow for it, multiple organizations can share access to their Science DMZs [**2**]. Therefore, the slower site or campus network gains access to the high performance resources of the Science DMZs, allowing high performance data transfers over the wide area network. A critical component is the data transfer node (DTN), which is dedicated to managing the efficient data transfers. For these transfers, the Science DMZ model prioritizes correctness, consistency, and performance, in that order [**1**]. Different security measures such as data encryption during transfers, and stateless firewalls controlling which DTNs are communicating, help to prevent data exfiltration [**2**]. Protecting the DTN will be one of the main focuses of this work.

The DTN connects directly to the Science DMZ router, serving as a high-performance server responsible for managing all of the incoming and outgoing data. As such, it is a critical component, and the security of the Science DMZ depends on protecting the DTN and its data. The DTNs do not typically run many applications, they are used almost solely for parallel data transfers (commonly performed using GridFTP [**3**]), along with some performance monitoring performed by tools like perfSONAR [**4**], and system maintenance [**1**, **2**]. This simplicity not only improves the efficiency of file transfers, it also improves the security of the DMZ by allowing strict access control to be implemented, minimizing the attack surface. The Science DMZ model is flexible, meaning no two DTNs will be identical in terms of hardware and software [**5**], and more or less

user access to the DTNs is possible depending on the security policy. For example, shell access might be restricted on the DTN in some cases [**1**]. However, the basic usage can be expected to remain similar across organizations. This is helpful to keep in mind when considering detection of attacks, as the range of normal and acceptable behavior is much more limited and predictable than a general purpose system. This narrower range of normal network and host activity makes practical anomaly detection based on system performance metrics. Access to the DTN is strictly controlled, meaning insider attacks from within an organization might be the primary concern. Detecting these attacks is one area of research we will focus on.

There are many key differences between Science DMZ networks and standard enterprise networks that have implications on how anomaly detection is performed [**6**]. This includes both the type of traffic seen, and the types of devices within the network. A typical enterprise network contains a wide variety of devices, while the Science DMZ has much fewer devices. While enterprise traffic tends to consist of many short flows, Science DMZ traffic usually consists of a few large and long-lasting flows. The types of common applications are also different, with Science DMZs being limited largely to data transfers and performance monitoring. This narrower scope greatly helps to simplify anomaly detection and changes which threats we must consider.

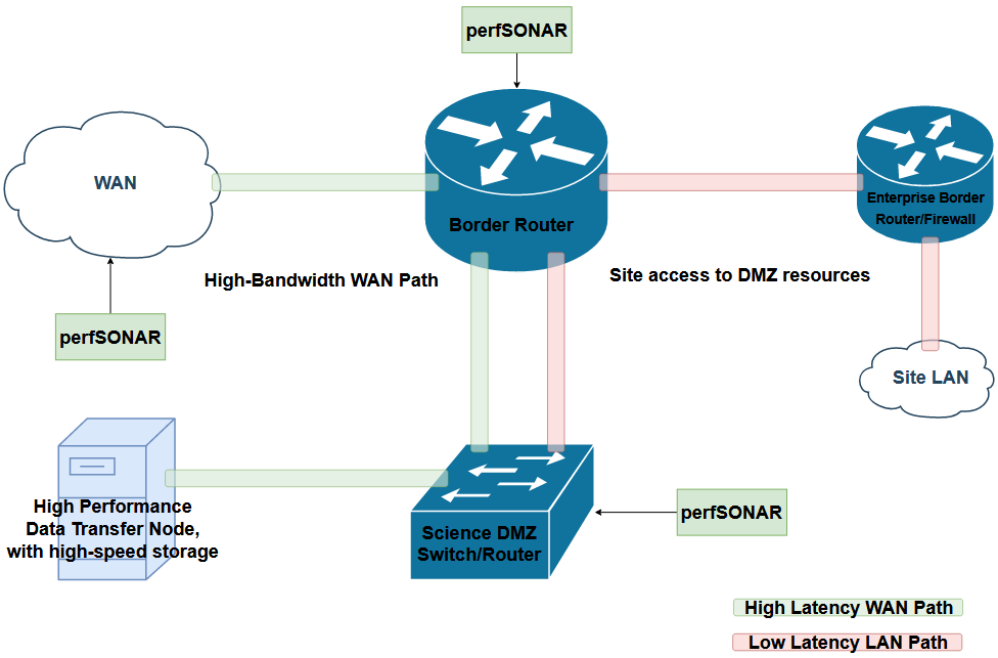


FIGURE 1.1. A typical Science DMZ [**7**].

1.2. Outline of Contributions

In order for anomaly detection to be practical, there are many concerns we must address. Sommer and Paxson [**8**] describe some common concerns across prior machine learning-based anomaly detection research. They discuss the necessity of avoiding *closed world assumptions* when applying anomaly detection based on machine learning. A *closed world assumption* is defined by Witten et al. [**9**] as *the idea of specifying only positive examples and adopting a standing assumption that the rest are negative is called the closed world assumption.* Sommer et al. argues that in many cases where machine learning is used for anomaly detection, there is too broad of a scope and that anomalies are inappropriately considered attacks by default (the semantic gap), leading to excessive false positives. Anomaly detection is better applied towards detecting known attacks versus novel ones. In our work, an important goal is finding detection methods that are realistic and applicable to real environments. Therefore, we will attempt to address these concerns in our work. Due to their narrower use, the Science DMZ and other specialized network models are well-suited for applying machine learning-based anomaly detection. Throughout our research, we will consider a variety of different detection methods and evaluate their effectiveness. For our science DMZ work, we focus on machine learning-based approaches, primarily clustering. We will demonstrate how clustering algorithms such DBSCAN [**10**] can be effective, and show how the algorithms might be modified for real-time detection in the Science DMZ environment. However, we also consider some statistical threshold-based anomaly detection methods, such as the corrected-conditional entropy test [**11**]. This test can be effective for anomaly detection, but is difficult to perform in real-time. Therefore, we utilize different architectures (MPPA, GPUs) and techniques to attempt a more practical implementation of the algorithm for detection in high speed networks.

1.2.1. External Threats to the Science DMZ. We begin by considering external threats to the Science DMZ, focusing on monitoring the data transfer nodes, a critical component of the model. Our interest is in determining whether or not we can use commonly available system and network metrics together to detect external threats, such as denial of service attacks. Although denial of service attacks might be unlikely on a DTN, we chose to consider TCP-SYN floods as a starting point for future research on Science DMZ threats. DTNs are often monitored with network

intrusion detection systems (NIDS). However, NIDS do not consider system performance data, such as network I/O interrupts and context switches, which can also be useful in revealing anomalous system performance potentially arising due to external network based attacks or insider attacks. We will demonstrate how system performance metrics can be applied towards securing a DTN in a Science DMZ network. Specifically, we evaluate the effectiveness of system performance data in detecting TCP-SYN flood attacks on a DTN using DBSCAN (a density-based clustering algorithm) for anomaly detection. Our results demonstrate that system interrupts and context switches can be used to successfully detect TCP-SYN floods, suggesting that system performance data could be effective in detecting a variety of attacks not easily detected through network monitoring alone.

1.2.2. Insider Threats to the Science DMZ. In the next chapter, we will expand on this idea, considering insider threats to the Science DMZ. In particular, we choose to consider the detection of data exfiltration and data sabotage attacks by insiders. Although some limited network intrusion detection systems (NIDS) are deployed to monitor DTNs, this alone is not sufficient to detect insider threats. Monitoring for abnormal system behavior, such as unusual sequences of system calls, is one way to detect insider threats. However, the relatively predictable behavior of the DTN suggests that we can also detect unusual user activity through monitoring system performance, such as CPU and disk usage, along with network activity. In this paper, we introduce a potential insider attack scenario, and show how readily available system performance metrics can be employed to detect data tampering within DTNs, using DBSCAN clustering to actively monitor for unexpected behavior. We also consider a potential scenario for exfiltrating data through obfuscated SSH sessions, and show that this obfuscation can also be detected through system performance metrics. Some of the key contributions are as follows:

- We consider a new SSH obfuscation system using PDF files, describing how it could be exploited by insiders as part of the overall attack chain.
- We present a real-time detection method which can quickly identify unexpected file editing on the DTN using DBSCAN, as well as the obfuscated SSH sessions.
- We demonstrate the value of system performance metrics for anomaly detection on DTNs for protecting data integrity.

1.2.3. Detecting Exfiltration Through Covert Timing Channels. In the following chapter, we consider a different method for potential data exfiltration - covert timing channels. Specifically, covert timing channels which use inter-packet delays in network packets to communicate. Although DTNs typically transmit large amounts of data, and the bandwidth of these channels is generally low, it is possible they could be used to extract small amounts of important data, or that a long-lasting channel could be used to extract larger sets of data. We begin with our research on the effectiveness of entropy-based detection of various covert timing channels. In particular, we focus on a more complex entropy calculation known as corrected-conditional entropy (CCE). The CCE test is useful for anomaly detection, but the complexity makes it difficult to apply in real-time, particularly for higher traffic rates. We attempt to make the CCE detection more practical for real-time detection by modifying the algorithm and using different architectures to improve the effectiveness of the detection. First, we consider an MPPA architecture, the TILEPro64, then we consider a method using GPUs. The GPU method was considerably more effective, so we put more emphasis on it in this work. GPU-based packet processing provides one means of scaling the detection of CTCs and other anomalies in network flows. We implement a GPU-based detection tool, capable of detecting model-based covert timing channels (MBCTCs). The GPU's ability to process a large number of packets in parallel enables more complex detection tests, including the corrected conditional entropy (CCE) test, which has a variety of applications outside of covert channel detection. In our experiments, we evaluate the CCE test's true and false positive detection rates, as well as the time required to perform the test on the GPU. Our results demonstrate that GPU packet processing can be applied successfully to perform real-time CTC detection at near 10 Gbps with high accuracy. Some of the important contributions of this covert timing channel research are as follows:

- Our detection tool marks a significant improvement over previous CTC detection work. In our experiments, we manage to detect nearly 100% of our sample's CTCs.
- In addition to confirming the CCE test's effectiveness, we achieve higher rates compared to previous real-time detection experiments [**12**]. We achieve close to 10 Gbps using medium-sized packets.

- By performing our tree transformation and completing the calculation using arrays, we compute the CCE scores in less than 1 ms per flow, an order of magnitude faster than previous results [**13**].
- Our work also demonstrates how GPU packet processing can efficiently calculate complex individual flow statistics using packet batches.

1.2.4. Creating Modular Detection Tools. In the next chapter, we present new detection tools we have created to assist in anomaly detection. First, we present a modular tool for detecting distributed denial of service attacks. Using this tool, we perform experiments evaluating the effectiveness of various detection techniques in large-scale science networks. The detection methods tested include common entropy and volume-based techniques. Using our tool, we evaluate the true and false positive rates of these techniques for detecting known DDoS attack samples. In addition, we merged these attack samples with large science flow traffic to determine whether the attacks could still be detected in the presence of legitimate periods of high volume traffic. The tool used is highly configurable, and could be used to quickly evaluate new detection tests. The second portion of this chapter presents new plugins and scripts created for the Zeek security monitor [**14**] to improve the monitoring of protocol usage. By default, Zeek lacks the ability to log much information about L3 and L4 protocol usage. These new tools help to address this hole in visibility. The scripts are highly configurable and are capable of logging statistics about protocol distribution and how it changes over time. To demonstrate the utility, we apply these tools towards detecting DDoS attacks as well, using real attack data from the recent CIC-DDoS2019 dataset. All of the tools discussed in the chapter have been made available on public GitHub repositories.

This dissertation consists of our completed work on the topic of anomaly detection for Science DMZ and other related high speed networks. The dissertations is organized into the following chapters. In chapter 2, we introduce our research work on Science DMZ threat detection, describing our work on detecting external threats, using TCP-SYN floods as a starting point. In chapter 3, we will show more of our work on Science DMZ threat detection, describing how system and network performance metrics can be applied towards detecting insider threats. Chapter 4 will discuss methods for detecting covert timing channels, which could be used for data exfiltration in the context of both Science DMZs as well as other high-speed networks. Chapters 5 will discuss

our experiments with DDoS detection in high-throughput networks. In Chapter 6, we will discuss our work on expanding the Zeek IDS for monitoring for unusual protocol usage. Finally, Chapter 7 will summarize the contributions of these works, along with providing discussion on how they could be expanded upon by future work.

CHAPTER 2

Anomaly Detection Using System Performance Data

2.1. Introduction

Modern research often requires the efficient and reliable movement of vast amounts of data, with some organizations generating terabytes of data daily [**6**]. To help facilitate the movement of petabytes of data, organizations utilize Science Demilitarized Zone (DMZ) networks - a specialized network model intended to maximize data transfer efficiency. Through a combination of network organization, performance tuning, and dedicated data transfer nodes (DTNs), the Science DMZ model helps guarantee reliable and high performance data transfers [**6**]. This also implies that protecting the performance of such networks is an important security concern, and security measures must be considered with this in mind [**1**]. Therefore, Science DMZs avoid typical firewalls to maximize network transfer efficiency, instead relying on various detection systems and Access Control Lists (ACLs) [**1**]. Typically, network intrusion detection systems (NIDS), such as Zeek(Bro) [**14**] or Snort [**15**], tend to rely solely on network metrics to identify abnormal traffic or attacks. However, we believe system performance metrics can also reveal the type of traffic being received, including malicious traffic [**16**] [**17**].

DTNs can become a critical point of failure in a Science DMZ if performance becomes compromised due to a denial-of-service (DoS) attack. Our work evaluates how system performance metrics might be used to identify a standard DoS attack such as a TCP-SYN flood attack, directed toward a DTN. In our study, we configured a server with a 10 Gbps backbone link to our campus Science DMZ as a DTN following the best practices outlined in [**18**]. Scientific workloads are emulated on the DTN by transferring files from the Energy Sciences Network's (ESnet) test DTNs, to generate real network traffic and system activity, while logging system performance metrics such as interrupts, CPU utilization, memory utilization, context switches as well as packets received/transferred

and bytes transferred/received. We used an anomaly detection system based on DBSCAN clustering [**19**] to analyze these metrics for anomalies. The results given by the detection system are then evaluated to gauge the effectiveness of these metrics in detecting TCP-SYN floods as anomalies. By validating this detection method using a well-known and heavily studied attack, we establish the effectiveness of this method and that it can be extended upon and generalized to detect other types of external network-based attacks and insider attacks.

In section 2.2, we provide some additional background on the Science DMZ. Section 2.3 discusses the system performance metrics we have selected for detection. Section 2.4 discusses machine learning for anomaly detection. Section 2.5 discusses the DBSCAN clustering algorithm. Sections 2.6 and 2.7 explain our experimental setup and evaluation, respectively. Lastly, section 6.5 provides a conclusion and possibilities for future work.

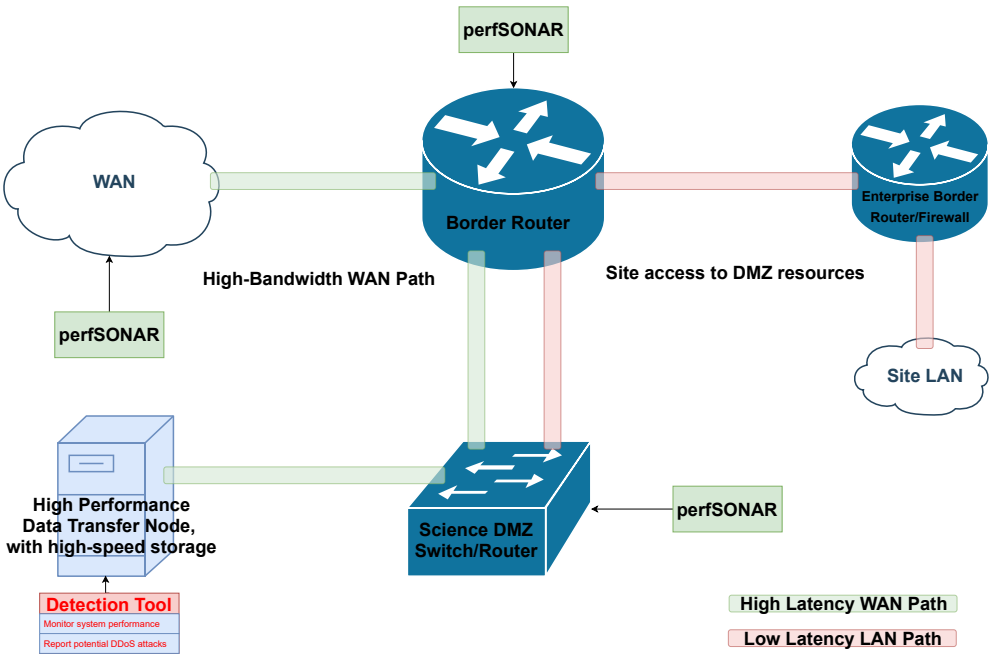


FIGURE 2.1. The detection tool monitors system performance metrics on a data transfer node. Using clustering-based anomaly detection, we can monitor for attacks by checking for abnormal system performance.

2.2. Science DMZ

A Science Demilitarized Zone (DMZ) is a network paradigm designed for enabling large-scale scientific research by providing a scalable environment for data transfers and reliable performance [**1**]. It has been recognized by the National Science Foundation (NSF) as a best practice and adopted at a number of research institutions in the United States to facilitate data-intensive scientific research [**20**]. Although Science DMZs vary among organizations depending on their purpose [**7**], there are some shared features across all DMZs. The Science DMZ is usually placed at the organization's network perimeter, with a border router connecting the site with the Science DMZ resources and typically has dedicated network components tuned to maximize data transfer performance. The most critical component of a Science DMZ is the data transfer node (DTN), servers provisioned for and dedicated to efficient high-speed data transfers between sites. As dedicated systems, DTNs do not have general computing applications installed (e.g. email clients, media players, document editors) [**1**], and are limited to parallel data transfers tools such as GridFTP, and performance monitoring tools such as perfSONAR [**21**] [**1**] [**3**].

2.3. System Performance Metrics

Predictions about the network activity can be made by considering the system metrics. For example, increased network I/O is associated with increased CPU usage [**22**]. Previous work has demonstrated that certain features can be used to identify different types of network traffic, including attacks such as SYN floods or port scanning [**16**]. In particular, the relationship between SYN floods and significant increases in interrupts has been demonstrated [**17**]. Interrupts signal to the CPU that I/O needs to be performed, and each new SYN request triggers a new interrupt. A large number of interrupts, coupled with an unexpectedly small increase in other metrics that indicate meaningful work, such as context switches, can be used to identify flooding attacks [**17**].

Because DTNs perform a relatively narrow range of network activities, they generate predictable network and disk I/O activity [**1**] [**7**], and make them ideal environments for host-based anomaly detection. Therefore, we believe that SYN floods and other attacks can be detected through anomalous patterns of system metrics. The relationship between system metrics such as interrupts and the network metrics such as packets received can also be combined to enhance detection. For

example, abnormal relationships between the interrupts per second and packets or bytes received per second could potentially identify malicious traffic. We focus primarily on three different per-second metrics in our experiments - interrupts, packets received, and context switches.

2.4. Machine Learning for Anomaly detection

Machine learning algorithms, including clustering, excel at finding pattern similarities and are thus classically applied to predictive classification problems [**23**]. The anomaly detection problem (also known as the outlier problem) seeks to find *new* patterns in data that do not conform to expected behavior [**24**] [**25**]. It too can be modeled as a classification problem with "normal" and "abnormal" as classes. Sommer and Paxson [**8**] argue that the application of machine learning for anomaly detection must be used with care when using a "closed world assumption." Witten et al. [**9**] defined a closed world assumption as: *The idea of specifying only positive examples and adopting a standing assumption that the rest are negative is called the closed world assumption.* We address some of those points of concern here:

- *Bridging the Semantic gap*: Anomalous activity does not necessarily translate to an attack, and anomaly detection might be better suited for known attacks over novel ones. In our experiments, we carried out both normal baseline activity and the attacks so that the ground truth is known.
- *Narrow Scope*: DTNs have limited system functionality, as discussed in Section **??**, resulting in predictable network and system activity. This context makes it suitable for anomaly detection via machine learning.
- *Real Data*: We generated our own data using a test DTN and emulated scientific workflows with other test DTNs (as we were unable to obtain campus-level DTN data).

Anomaly detection systems often give critical, actionable information, so a good system should provide this information as soon as possible with the ability to predict anomalies in real-time with streaming data. DBSCAN clustering is one such an unsupervised machine learning approach that exhibits properties ideal for anomaly detection with noisy data streams in real-time [**10**]. The next section gives a broad overview of the algorithm and its derivative anomaly detection system.

2.5. DBSCAN

DBSCAN, short for density-based spatial clustering of applications with noise, is a well-established clustering algorithm [**19**]. Clusters are grouped together based on the density of the points, with low density points labeled as noise. Unlike k-means clustering, DBSCAN does not require you to specify the number of clusters beforehand, and the clusters can be any shape. DBSCAN only requires two parameters, ϵ (the greatest distance allowed between points in each cluster) and **MinPts** (the minimum number of points required for a cluster). Clustering algorithms like DBSCAN are a form of unsupervised learning, meaning learns without the need for training nor labeled data, both of which are needed with supervised approaches. This makes it a strong choice for monitoring real-time data, since DBSCAN allows us to differentiate between normal patterns and anomalous ones despite not knowing the exact thresholds beforehand. In addition, DBSCAN is designed for clustering noisy data [**10**], which makes it well-suited for monitoring the sometimes unpredictable network and system performance metrics. These properties make DBSCAN a good candidate for our anomaly detection system.

2.6. Experimental Setup

For our experiments, we set-up and monitored a machine, denoted as D, to act as a Science DMZ DTN, like that shown in Figure 6.1. D is a PowerEdge T630 server with a 10 Gbps NIC, two Intel Xeon E5-2637 v3 3.5GHz processors, a RAID-10 set of 8 1TB 7.2K RPM SATA 6 Gbps hard drives, and 32GB 2133MT/s RDIMM memory capacity. D has a 10 Gbps backbone link to CENIC, a 100 Gbps wide-area network, as is true for our campus DTN. As in a true Science DMZ [**1**], this machine requests data from other DTNs and receives files through the 10 Gbps link using Globus GridFTP [**3**] transfers. We emulate real scientific workloads by having D receive test data via GridFTP transfers from three read-only test DTNs provided by ESnet [**26**]. For each day of experiments, we continuously receive randomly selected files from one of these three test DTNs at randomized time intervals. The potential file sizes are shown in table 3.1. The actual workload of a DTN varies significantly depending on the organization and type of scientific workflow. Therefore we also consider two additional cases - a large number of small file transfers, and a small number of large file transfers. This simulates the different distributions that have been observed on a real

NERSC DTN [**27**]. In each case, the time between file transfers has been chosen such that the total expected data transferred is roughly 750 GB per day.

TABLE 2.1. GridFTP transfer distributions

Distribution	Potential File Sizes	Interval
Normal	10M, 50M, 100M, 1G, 10G, 50G	1-30 minutes
Large	10G, 50G	60-75 minutes
Small	10M, 50M, 100M, 1G	5-40 seconds

In addition to D, we have a machine designated as the attacker, denoted as A, with the same specifications as D. A is connected to D by a 40 Gbps Mellanox ConnectX-3 network adapter. To generate our TCP SYN flood attacks, we use hping3 [**28**], a freely available packet generator used for penetration testing. Using hping3, we have A generate SYN packets and send them to D over the 40 Gbps link. In order to determine the point at which unusual patterns of SYN packets becomes detectable, we vary the attack intensity by increasing or decreasing the number of microseconds between SYN packets sent. Each attack lasts for 15 minutes, with the intensity gradually increasing by reducing the delay between packets by 5 μs every 90 seconds. At the max intensity, hping3 just sends a flood of SYN packets with no delays in-between for 15 minutes. Six different intensities, with varying inter-packet delay ranges, are used in our experiments – **lowest** (150-100 μs), **low** (125-75 μs), **medium** (100-50 μs), **high** (75-25 μs), **higher** (50-0 μs), and **max** (0 μs). We performed four experiments using this system configuration and set-up as shown in figure 2.3. Three of the experiments involved periodic SYN floods occurring every two hours starting at noon, with each intensity level used once. The final experiment sends the medium, higher, and max intensity floods once, each at random times during the day. While our experiments run, we log system (interrupts, context switches) and network data (packets received, bytes received) per second on D using the `proc` filesystem.

Finally, we use DBSCAN clustering [**19**] to detect the anomalies in each time-series performance data set collected. First, we used DBSCAN to cluster the 24 hours worth of logged performance data gathered for each of the experiments shown in figure 2.3. The standard method of detecting TCP-SYN floods is to check whether or not the number of SYN packets received per second exceeds a predetermined threshold [**17**]. As figure 6.4 shows, looking at the packets received alone is likely insufficient on the DTN, particularly for low volume attacks, as file transfers can be mistaken for

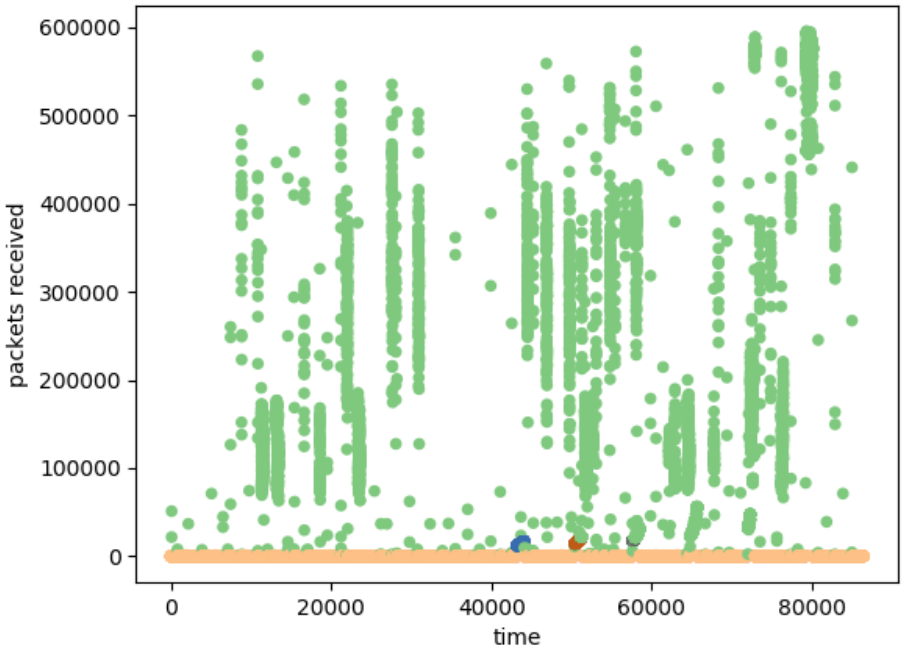
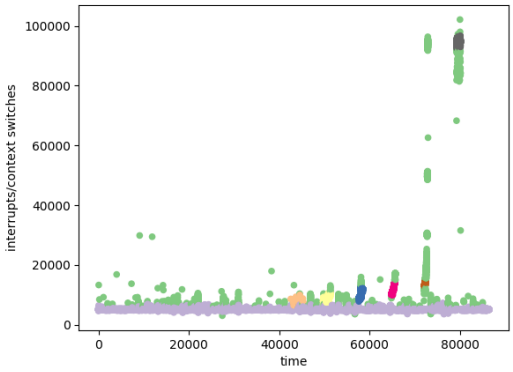
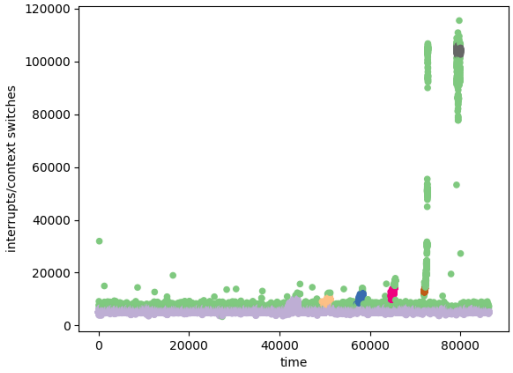


FIGURE 2.2. DBSCAN clustering using packets received per second on the DTN. The green dots represent noise, while the orange dots represent the primary cluster. The other colors are random clusters formed from the noise. The SYN floods here are generally indistinguishable based on packet volume alone.

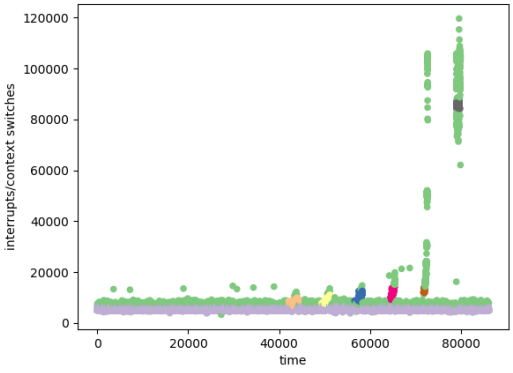
TCP-SYN floods. Instead, we consider the results of clustering interrupts over context switches vs. time for each of the four experiments over the entire day. We set the parameters such that, under normal conditions, there will only be a single cluster representing the normal range of interrupts over context switches. Therefore, if another cluster is discovered, it indicates a period of time where there is a high density of abnormal behavior. Then, we consider the effectiveness of a real-time detection system using DBSCAN clustering. The real-time streaming DBSCAN detection system continually adds data from the `proc` filesystem to a `pandas` dataframe. Once the dataframe grows large enough for clustering, we run DBSCAN and count the number of resulting clusters to check for anomalies. We consider the time required to detect SYN floods of different rates, and trade-offs associated with the various parameters.



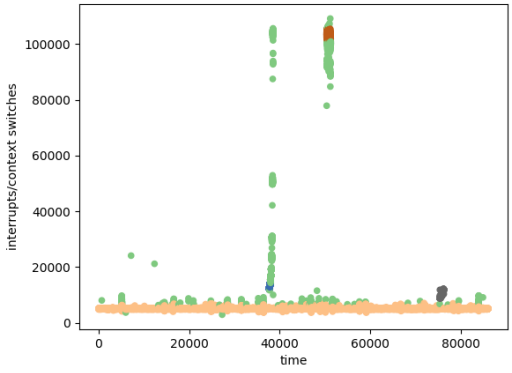
(a) Experiment 1 (periodic floods, random file transfers)



(b) Experiment 2 (periodic floods, large file transfers)



(c) Experiment 3 (periodic floods, small file transfers)



(d) Experiment 4 (random floods, random file transfers)

FIGURE 2.3. DBSCAN clustering results for the normalized interrupts over context switches vs. time (in seconds). In each subfigure, the largest cluster represents standard activity, while the green dots represent un-clustered noise. The other small clusters occur during SYN floods. In all cases, the SYN floods can be distinguished from ordinary traffic.

2.7. Evaluation and Discussion

In figure 2.3, we show the results of DBSCAN clustering on 24 hour DTN experiments. In these figures, the interrupts over context switches values have been normalized by multiplying them by 10,000 to improve the readability of the charts and simplify the choice of the DBSCAN distance parameter ϵ. To obtain these figures, we set ϵ to 910, and **MinPts** to 200. During ordinary conditions on the DTN (file transfers or inactivity), we found that the interrupts over context switches remained fairly consistent around 0.5 (5000 in the figure 2.3), with some noise occurring

throughout the day. However, ordinary spikes occur infrequently, compared to densely packed clusters which appear during the attacks. The small clusters shown in figure 2.3 occur only during the hping3 SYN floods. A large number of interrupts coinciding with a relatively low number of context switches generally indicates that less meaningful work is being performed [**17**]. Therefore, it is sensible that this is a good indicator of SYN floods, where a large amount of "useless" packets are handled within a short time frame. As expected, the higher intensity floods showed significantly higher interrupt over context switch ratios.

2.7.1. Real-time Anomaly Detection. Following the analysis of the full days worth of traffic, we consider how DBSCAN could be applied to detect attacks in real-time. A streaming anomaly detection system was created using Python, which gathers performance data and creates new clusters every 10 seconds. If an unexpected number of clusters is detected during a time slice, then an anomaly is reported. In our experiments, we found that accurate detection could be performed even clustering a small number of points. The recommended number for DBSCAN's **MinPts** parameter is one more than the number of dimensions [**29**]. In our case, the number of dimensions is two − the interrupts over context switches ratio and the time. Since there are only two dimensions, we don't need a large sample to perform reliable clustering. We found that the minimum recommended value of three points per cluster was sufficient to detect the attacks without false positives. Random spikes caused by noise never occurred closely enough together in a short time frame to trigger false alerts. The minimum time for detection is determined by how many points we choose to gather before re-clustering. Our experiments used 10 data points, making 10 seconds the minimum detection time. Figure 2.4 shows that most attacks were detected in 10 seconds using the minimum number of points, even at very low rates. In our experiments, we managed to detect SYN flooding at as low as 4,800 packets per second. Although we detected all attacks with no false positives, the bandwidth measuring tool iperf produces a similarly high interrupts over context switch ratio. As a common application used to test DTN performance [**6**], this could result in false positives. However, even if the scheduled iperf tests are not recognized beforehand, we can eliminate any false positives by also considering the number of bytes received per second in our analysis. We found that the bytes per second during an iperf test is at least

one order of magnitude higher than during the SYN floods we tested. Therefore, we can improve detection further by combining network and performance metrics.

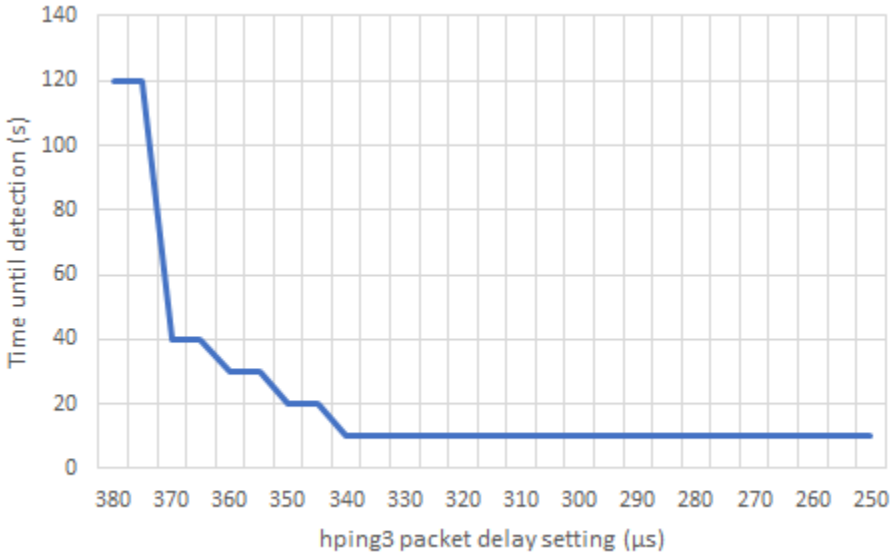


FIGURE 2.4. Time spent before detection during low intensity TCP-SYN floods. Over the course of an attack, the number of interrupts gradually increases, until it becomes detectable. All but the lowest rate attacks were detected after the minimum 10 seconds required to gather data for clustering.

2.8. Conclusions and Future Work

Our work demonstrates that system performance data can be effective in detecting TCP-SYN floods, an attack traditionally detected by a network IDS, on a Science DMZ DTN. The limited variety of tasks typically performed on a DTN helps us use system performance metrics to reliably detect threats. Interrupts over context switches consistently detected the SYN floods, even at very low intensities. Combining this with network metrics such as the number of packets or bytes received per second can improve detection. DBSCAN can applied in a real-time detection system, clustering performance metrics over time to detect attacks. Successfully detecting SYN floods on the DTN suggests that this system could be generalized for other threats, such as port scanning or insider attacks which are unlikely to be detected with network activity alone. With that in mind, these metrics could be valuable being incorporated into an intrusion detection system such as Zeek

(Bro). Future work could also involve using both network and system performance metrics (e.g. context switches, etc.) to detect insider attack scenarios on a DTN.

CHAPTER 3

Insider Attack Detection Using System Performance Data

3.1. Introduction

Scientific research depends on the safe transfer of huge quantities of data, in some cases terabytes worth [**6**]. Research organizations use Science Demilitarized Zone (DMZ) networks, a network model designed to guarantee optimized and reliable transfers through performance tuning and efficient network organization, as well as custom built data transfer nodes (DTNs). This Science DMZ model enables higher performance and more reliable data transfers [**1**], and help connect research sites to each other as well as cloud computing resources. Science DMZs often avoid typical defense measures such as firewalls in order to optimize performance, instead using Access Control Lists (ACLs) and other forms of detection [**30**]. However, these defenses are insufficient for handling insider threats. Insider threats are a serious concern, as ensuring data integrity is critical to scientific research [**31**]. In particular, protecting data confidentiality and preventing data exfiltration are important concerns [**31**]. Therefore, monitoring for insider data tampering is important to provide the DTNs with extra protection, both to protect their performance and the integrity of the transferred data. DTNs can also be useful in contexts outside of scientific research, such as transfers between cloud service providers [**32**].

This work examines a method of detecting insider attacks targeting the data stored on or moving through a DTN. Since the insider attack category can be broadly defined, we focus our efforts primarily on detecting data sabotage, considering a possible attack scenario involving SSH obfuscation. We consider a novel method of detecting data tampering which monitors the host performance data to detect file editing events, distinguishing between user file modification and the modification occurring as a result of the DTN's file transfers. The limited range of legitimate DTN operations [**2**] allows for effective anomaly detection. The anomaly detection method used for this utilizes DBSCAN clustering [**19**] of host metrics such as CPU utilization, along with checking disk

writes and network activity. To recreate the DTN environment, we set up an experimental DTN using a server with a 10 Gbps backbone link to the UC Davis Science DMZ. Our experimental DTN follows the best practices described in ESnet's DTN tuning guide [**18**]. To emulate scientific data being transferred to the DTN, real traffic and DTN activity is generated by files continually transferred from the Energy Sciences Network (ESnet) test DTNs. As the DTN operates, we continually log and monitor system performance and network behavior, performing real-time monitoring by continually re-clustering the CPU activity and checking the disk and network activity.

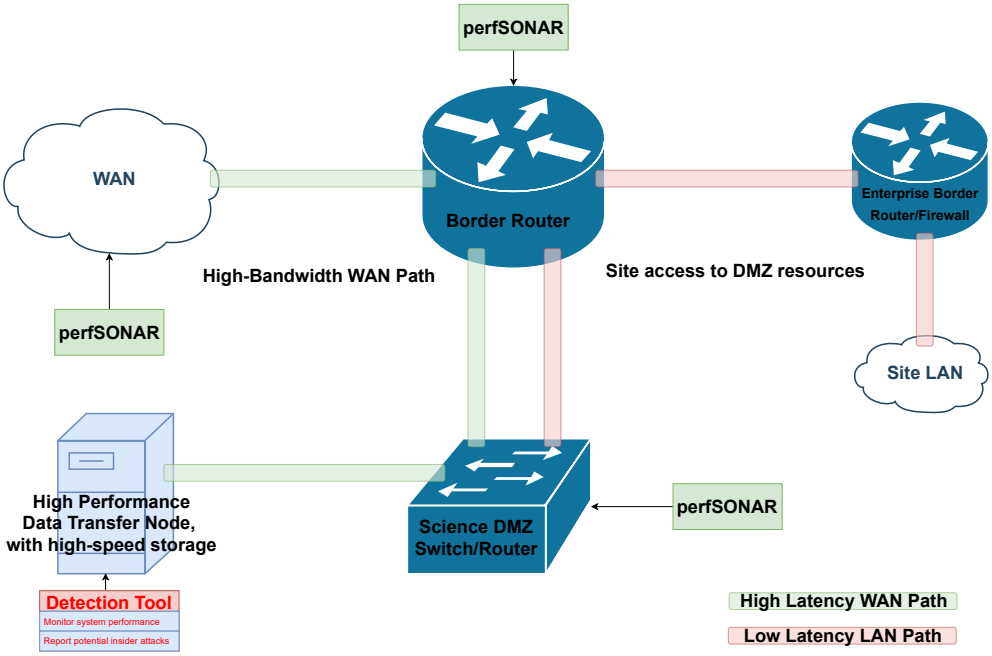


FIGURE 3.1. The detection tool monitors system performance metrics on a data transfer node. Using clustering-based anomaly detection, we can monitor for insider attacks such as data tampering or exfiltration.

The key contributions are as follows:

- We consider a new SSH obfuscation system using PDF files, describing how it could be exploited by insiders as part of the overall attack chain.
- We present a real-time detection method which can quickly identify unexpected file editing on the DTN using DBSCAN, as well as the obfuscated SSH sessions.
- We demonstrate the value of system performance metrics for anomaly detection on DTNs for protecting data integrity.

In Section 3.2 we provide background information on the Science DMZ, insider attacks, and system performance metrics, along with related work. In Section 3.3 we describe a potential attack scenario we have envisioned where an insider can establish an obfuscated SSH session on the DTN, giving them the ability to sabotage data. Section 3.4 we provide details on our detection system, in addition to some background on clustering and anomaly detection. In Section 3.5 we describe our experimental setup in-depth and in Section 3.6 will discuss and evaluate our results. Finally, in Section 3.7 we summarize our conclusions and discuss future work.

3.2. Background and Related Work

3.2.1. Insider Attacks. An insider attack can describe any case where the attack is performed by somebody with legitimate access to a system [**33**]. Clearly, this encompasses a wide range of attack types, from data sabotage and leaking to blackmail and fraud. Many different insider attack taxonomies have been created [**34**] [**35**]. Generally speaking, we need to first consider the insider's profile, whether they are intentionally malicious or if they are unintentionally causing damage, if they are masquerading as authorized or legitimately authorized, and what is their intended role within the system [**34**]. The attacks also vary in terms of the level of sophistication and the attacker's knowledge of the target, their personal motivation and goals. We need to consider the scope and targets of the attack - are they attacking the network, the operating system, applications, or stored data? In some cases, the attacks will occur across multiple levels but become more noticeable on a particular level. For instance, data tampering is noticeable at the application or data level, while exfiltration can be observed at the network level [**34**].

Figure 3.2 shows the insider attack chain, and some actions that might be taken during the different steps of an attack. Attackers can be classified into one of three categories depending on their actions and role in the chain of an attack [**34**]. Masqueraders perform reconnaissance and imitate legitimate users to set up an attack, while traitors and unintentional perpetrators execute the attack through extracting data, sabotaging data or some other part of the overall system. In our experiments, we focus primarily on the "actions on objectives" portion of the attack sequence, looking at data tampering detection in the DTN environment. However, in Section 3.3, we will consider an SSH obfuscation method which could be used as part of a full insider attack chain.

FIGURE 3.2. The cyber attack chain [**34**]. In the first five steps, the insider acts as a "masquerader" to gain control of the system. The final actions can also be executed by a traitor or an unintentional perpetrator within the organization, skipping the initial steps.

This method can be used to conceal the SSH protocol, making the exchange of protocol related packet data within PDF files. To a network IDS monitoring the DTN, this activity would appear as normal file transfers. We will explain its functioning at a high level, and consider how we can detect the obfuscated SSH sessions before any data tampering occurs.

As expected from the wide variety of attacks, there are many categories of insider attack defense. Our focus is on the "Detection and Assessment" branch of defense [**34**], in particular we consider anomaly-based detection and unsupervised detection of insider attacks. In this paper, one of our goals will be to try detecting insider threats on the DTN using novel data sources and techniques. Therefore, we will try to detect insider threats using host performance data such as CPU usage and disk writes, applying DBSCAN clustering [**29**] to create a detection scheme well-suited for the DTN environment.

3.2.2. Related Work. Detecting insider attacks is a very broad research area, considering a range of topics based on the types of attackers and the form of the attacks. For further reading, Homoliak, Ivan, et al. [**35**] provides a survey of the large variety of taxonomies for insider attacks, covering taxonomies both for attacks and defense techniques. Liu et al. [**34**] provides another look at insider attack taxonomies. A CERT guide [**33**] gives an in-depth description of insider attacks and best practices for mitigating or eliminating them. In our case, the focus is primarily on data

sabotage by an insider who at one point was granted legitimate access to a system, and using machine learning to detect that anomalous behavior. A number of different data sources have been used for this form of insider detection [**35**]. Many other insider threat detection papers have relied on anomaly detection based on system calls [**36**] [**37**]. However, using performance data for insider threat detection like our method appears less common.

Some recent papers have been written on securing Science DMZs specifically. Nagendra et al. [**30**] introduces a tool called SciMon, designed to protect Science DMZ DTNs. Trivedi et al. [**38**] describes a high-speed network IDS for Science DMZs which utilizes Zeek (formerly known as Bro) [**14**]. Machine learning based anomaly detection appears uncommon, and these papers do not consider system performance metrics for detection. However, these metrics have been applied to detecting insider attacks in other contexts. Nikolai et al. [**39**] describes a method of detecting insider data theft in IaaS cloud environments using a mixture of system metrics such as CPU and memory usage and network metrics such as the number of network bytes sent or received. Oppermann et al. [**40**] describes how a simulated insider attack in a cloud environment can be detected by monitoring CPU usage along with network traffic.

3.3. Data Obfuscation Attack Scenario

In order to establish how an insider might gain the ability to tamper with data on the DTN, we consider one potential attack scenario using data obfuscation. This can be considered as the "Masquerader" step shown in Figure 3.2. We make the assumption that the attacker is an insider who at one point had been granted legitimate access to the system, and the privileges necessary to modify data on the DTN. In this case, it is possible that the insider could setup a backdoor, allowing them to continue remotely modifying the DTN files after officially losing access. However, the attacker would like to avoid being caught running an SSH session, which might be blocked by the DTN [**1**]. Therefore, we consider a method of SSH obfuscation which could be used to remotely execute the attack. By hiding the raw SSH data in another file type, we can have a hidden SSH session. Many types of files could be used, including image files. In our case, the SSH sessions are obfuscated by hiding SSH data within PDF files.

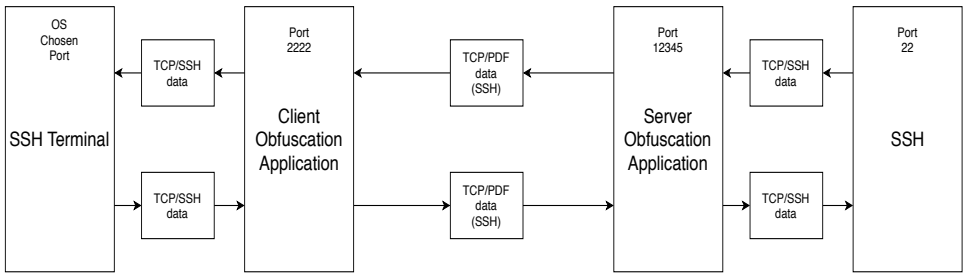


FIGURE 3.3. PDF obfuscation overview. SSH protocol data is hidden within PDF data sent over the network. This obfuscation method is one possible method an insider could use to stealthily establish an SSH session and perform data sabotage on a DTN. Although we chose to use PDF files, the SSH data could be placed into other data types as well, such as image files.

This PDF obfuscation method involves creating a stealth SSH connection by concealing the protocol data within PDF files before transmitting on the network. If both the sender and receiver are able to decode the obfuscation, a covert connection can be setup. Assuming the network IDS does not perform deep packet inspection to block packets containing PDF data, the SSH session with the target will appear as normal file transfers over GridFTP. Since we know DTNs do not perform this type of check, and GridFTP file transfers are the expected behavior, it is likely this method would allow the attacker to establish a stealth SSH session, giving them the capability to execute their attack.

The basic functioning of the PDF obfuscation method is shown in Figure 3.3. The obfuscation server is setup on the DTN, while the insider is running the client on their own machine. As the endpoints exchange packets, the obfuscation program checks if the packet contains obfuscated data. If it does, then it will deobfuscate the data by inserting it into a PDF and reading from it, sending that raw data to the original application - SSH in this case. If the packet does not contain obfuscated data, the program inserts the raw SSH protocol data into the PDF to obfuscate it, before sending it through the tunnel. By masquerading as legitimate traffic, the insider gains the ability to perform their tampering on the DTN while evading detection by network intrusion detection systems. Between this and tampering with user logging, the insider could evade detection through traditional means. Therefore, alternative means of detection such as monitoring system performance metrics could improve upon traditional insider attack defense.

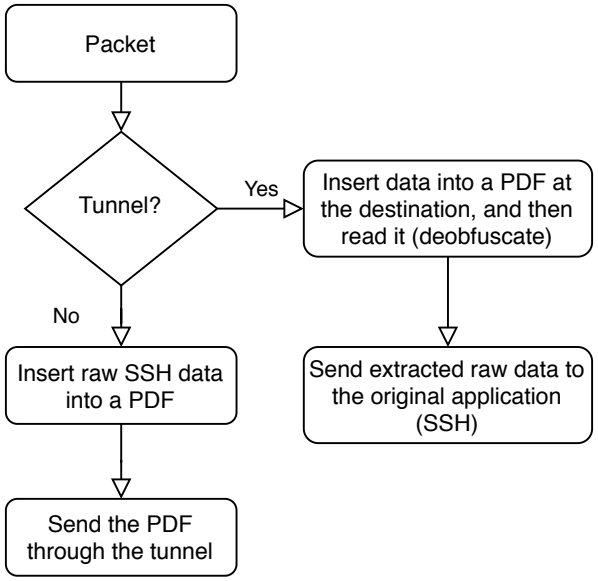


FIGURE 3.4. PDF obfuscation tunneling method. The PDF obfuscation program either places the raw SSH protocol data into a PDF before sending it through the tunnel, or extracts the raw protocol data arriving through the tunnel. The tunneled data (PDF data containing obfuscated SSH data) is inserted into a PDF at the destination. The obfuscation application then extracts the raw SSH data and sends it to SSH.

Regardless of the precise method used to gain or maintain access, the attacker has the ability to modify data, putting any files stored on the DTN during the transfer process at risk. The following section describes our method of detecting file editing on the DTN, and how file editing can be distinguished from the disk writes caused by standard DTN transfers.

3.4. Data Sabotage Detection

Previous work has demonstrated how certain host performance metrics can be linked to network activity, and can be used to detect some insider attacks [**40**]. In addition, logs of host activity (system calls, user command histories) and network data are common data sources used to perform anomaly detection for different forms of insider attacks [**34**]. However, detection schemes relying on forensic logging can be hampered if the attacker is able to modify these logs [**41**], and interpreting the log data can be difficult or time-consuming [**42**]. Therefore, we want to consider additional data sources. Normally, as a dedicated component of the Science DMZ, a DTN is expected to perform only a narrow range of tasks, mostly related to moving data and performance monitoring. The

performance metrics can be expected to remain within a consistent range, which allows us to more easily predict typical system performance [**1**] [**7**]. Therefore, we take advantage of the difference in system performance during normal activity and file modification to help detect unexpected file editing events which might occur during an insider attack, as well as detecting obfuscated SSH sessions.

3.4.1. Performance Metrics Monitored. In order to achieve this, we monitored host performance metrics stored in procfs [**43**] - **system CPU usage**, **user CPU usage**, **disk writes**, **interrupts**, and **context switches**. Figure 3.6 shows how the performance metrics change over the course of a 10GB file transfer to the DTN, and figure 3.8 shows how the CPU changes during file editing. Standard file editing increases CPU usage noticeably, along with the obvious spike in the data written to the disk. However, a large file transfer to the DTN causes a similar increase in total CPU usage and disk writes which can mask the file editing when the two events overlap, and a clever attacker could arrange for editing to coincide with large transfers. Fortunately, the effects on CPU usage are distinguishable by looking at the user and system CPU usage. Network file transfers increased the system CPU (often significantly weighted on one core), while the user CPU usage remains stable. Meanwhile, if the files are large enough, file editing will noticeably increase the user CPU usage while the system CPU usage remains stable, as shown in figure 3.9. Based on this, CPU usage is the primary means used for detecting file editing events.

3.4.2. Clustering Performance Metrics. To actively monitor these performance metrics, we continually cluster the user CPU usage using **DBSCAN** (density-based spatial clustering of applications with noise), a widely used clustering algorithm [**10**]. Although other clustering methods could also be applied, this particular clustering algorithm was selected for anomaly detection because it is specifically designed for clustering noisy data [**29**]. This is necessary when monitoring network and system performance metrics, which will have noticeable noise even in a relatively predictable environment. As a form of unsupervised learning, DBSCAN allows us to distinguish between normal and unusual performance data without performing training or using predefined threshold values.

Although an in-depth comparison of clustering methods is beyond the scope of this paper, it is worth noting that DBSCAN is simple and flexible compared with other classic clustering methods such as k-means clustering. Any cluster shape is possible, and there is no need to define the number of clusters beforehand. The only necessary parameters are the maximum distance between the points within a cluster, ϵ, and the minimum points required to form a cluster, **MinPts**. There are three types of points - core points, border points, and noise. Core points are within ϵ distance of two or more points, while border points are within ϵ distance of just one other point. Noise points are not within ϵ of any other points. If at least **MinPts** points are connected as core or border points, then that becomes a cluster.

The relative simplicity and ability to handle noise well also allows us to easily cluster data in real-time. Using Python, we created two scripts which continually create new small clusters every 10 seconds using the per-second data we gather. The first script clusters **user CPU usage** to detect unexpected file modification, while the second script clusters **disk writes** to detect obfuscated SSH sessions. With ϵ=4 and **MinPts**=4, the baseline user CPU usage values will be placed into the same cluster. However, during a file editing event, outliers or additional clusters of higher values will appear. Once this is detected, we check if the disk writes are greater than 1MB. If outliers or extra clusters appear alongside disk writes, we report an anomaly indicating file editing. Figure 3.5 shows the clustering results when 10MB of file editing coincides with a 10GB file arriving on the DTN.

3.4.3. Machine Learning for Anomaly Detection. Clustering and other forms of machine learning-based anomaly detection are effective at detecting new behavior which does not match the expected data patterns [**25**]. Therefore, machine learning is often applied towards predictive classification problems. However, anomaly detection based on machine learning must address a classification problem, defining properly what is "normal" and what is "abnormal". Sommer et al. [**8**] discusses the necessity of avoiding *closed world assumptions* when applying anomaly detection based on machine learning. A *closed world assumption* is defined by Witten et al. [**9**] as *the idea of specifying only positive examples and adopting a standing assumption that the rest are negative is called the closed world assumption.* Sommer et al. argues that in many cases where machine learning is used for anomaly detection, there is too broad of a scope and that anomalies are inappropriately

considered attacks by default (the semantic gap), leading to excessive false positives. Anomaly detection is better applied towards detecting known attacks versus novel ones. In our experiments, we attempt to bridge the semantic gap by carrying out normal baseline activity alongside the attack in order to establish a ground truth. In addition, we focus on a narrow scope - the limited functionality of DTNs results in more predicatable activity. Therefore, machine learning is suitable for anomaly detection in this context. Furthermore, although we did not have access to campus-level DTN data, we generate real data using a test DTN, emulating scientific workflows by receiving data from other test DTNs. Our experimental setup is discussed in the following section.

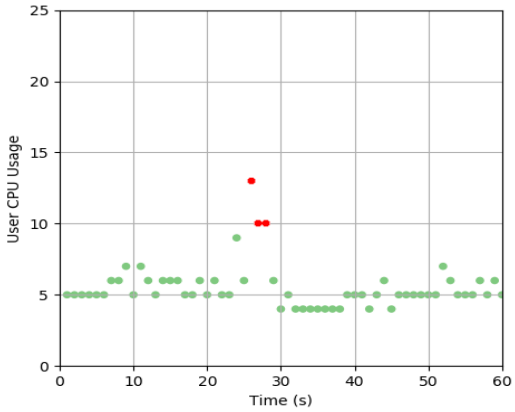


FIGURE 3.5. DBSCAN clustering of the user CPU usage during a 10MB file editing event coinciding with a 10GB file transfer. Although the CPU usage fluctuates, it will not normally increase beyond a certain value during a transfer. If the user CPU usage spikes without being clustered, this suggests unusual user activity. The red spike coinciding with disk writes indicates file editing, while the green represents ordinary CPU usage.

3.5. Experimental Setup

3.5.1. DTN Testbed. To simulate a real Science DMZ data transfer node (DTN), we setup an experimental system acting as a DTN, a PowerEdge T630 server which we refer to as D. D has a RAID-10 set of 8 1TB 7.2K RPM SATA 6 Gbps hard drives, 32GB 2133MT/s RDIMM memory capacity, and contains two Intel Xeon E5-2637 v3 3.5GHz processors. Just like our campus DTN, D is connected to a 100 Gbps wide-area network, called CENIC, through a 10 Gbps backbone link. D continually requests and receives data from three different Energy Science Network (ESnet) test DTNs through Globus GridFTP. To simulate the behavior of an ordinary DMZ, D randomly

requests different sizes of files from these test DTNs. These file transfers continue over the course of the day. Table 3.1 shows the sizes of the files requested from the test DTNs. To simulate the different distributions seen on a real NERSC DTN [**27**], we have two potential file size distributions, with an equal value for the total expected data received (750 GB per day). The first case is a large number of small files, while the second case is a small number of large files.

TABLE 3.1. GridFTP transfer distributions

Distribution	Potential File Sizes	Interval
Normal	10M, 50M, 100M, 1G, 10G, 50G	1-30 minutes
Large	10G, 50G	60-75 minutes
Small	10M, 50M, 100M, 1G	5-40 seconds

3.5.2. Data Sabotage. Through the PDF obfuscation method discussed in Section 3.3, the attacker can establish an SSH session with the target, while hiding the SSH session to secretly execute the data tampering script. Once the attacker gains access, either through masquerading or legitimate access, they gain the ability to modify data. The specific nature of the data is not significant. We choose to use `tstat` logs, which are commonly stored during network monitoring [**44**]. Using familiar data, we can make assumptions about how an attacker might want to alter the data set. The attacker might want to completely destroy the stored data, or they might want to selectively alter the data by changing the entries in one field. Selectively changing the data could be useful, allowing the attacker to extract the legitimate data sets while sabotaging the data left on the system. Modifying particular data fields could also be employed to manipulate research results, in the case of science data. Therefore, we consider different cases of data tampering in our experiments, by varying the degrees to which the data is altered. A knowledgeable attacker, familiar with the data, might only need to alter a few lines to get their desired result. Alternatively, they might alter large portions of the dataset. The difficulty of detecting these changes depends on the amount of files changed, but also on distinguishing between the attacker's file modifications and the changes caused by the file transfers. The nature of the DTN implies that files will not normally be modified outside of file transfers. However, the attacker could potentially send traffic while the data sabotage script runs, or, if the attacker is aware of when file transfers are occurring,

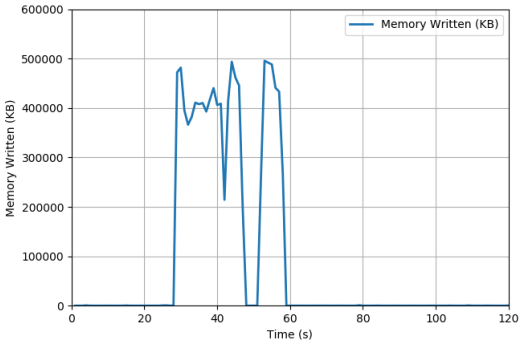
they might set their script to coincide with transfers. In addition, some log files could be routinely written on the DTN for monitoring [**2**].

3.5.3. Detection. We consider different sources of data in our experiments. Host and network performance data is gathered from **procfs**, as well commonly available tools such as **collectl** [**45**]. In addition to logging these values over a 24 hour period, we created two Python scripts to cluster the data in real time - one script for detecting file modification, and another script for detecting obfuscated SSH sessions. Each scripts works by gathering 10 seconds worth of per-second data, clustering it using DBSCAN clustering [**19**] to look for anomalies, then repeating. We cluster **disk writes** to detect obfuscated SSH sessions, and cluster **user CPU usage** to detect file modification. All of the points should cluster under normal conditions, meaning an anomaly is detected when a second cluster appears. When monitoring for file detection, we also considered noise outside the baseline cluster as an anomaly, since user CPU usage remains within a small range under normal conditions.

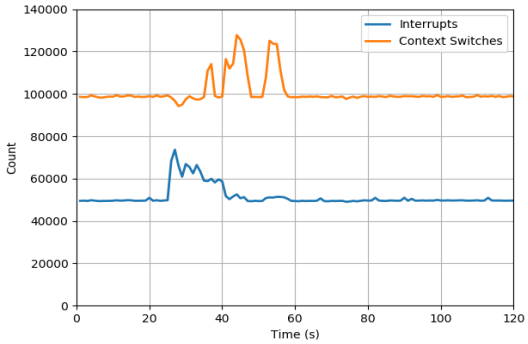
3.6. Results and Evaluation

We considered two insider attack cases - **data sabotage** through file editing, and **data exfiltration** through obfuscation. By monitoring the performance metrics and taking into account network activity, we can detect both of these cases. We will discuss the various performance metrics we measured, and how they were affected by file transfers, file editing, and PDF obfuscation. In addition, we will explain our detection results and some limitations of our approach.

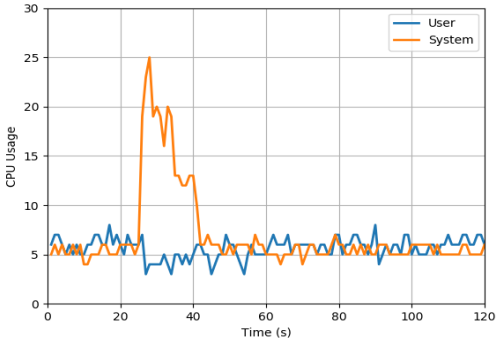
3.6.1. Data Sabotage. The most obvious impact on the performance metrics during file editing will be an increase in disk reads and writes, and increased CPU usage. Figure 3.9a. demonstrates the effects file editing had CPU had on user and system CPU usage. Other metrics, such as interrupts and context switches, did not increase significantly while observing file editing. Figure 3.8 shows how the user CPU usage, representing the time spent on user level processes, increases significantly as the amount of overwritten data increases, though it stabilizes around 30MB. Meanwhile, the system CPU usage (the time spent on kernel tasks) only slightly increases, remaining relatively stable regardless of increased disk writes. By observing increased disk writes coinciding with increased user CPU usage, we can easily identify a file editing event.



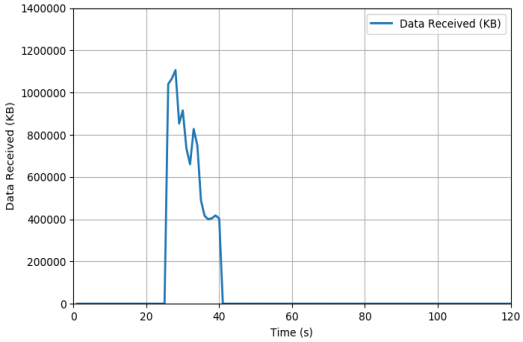
(a) Data written to disk



(b) Interrupts and context switches



(c) User and System CPU usage



(d) Data received

FIGURE 3.6. Host and network performance during a 10GB GridFTP transfer. Interrupts and context switches do not increase significantly. CPU usage, primarily system CPU usage, increases noticeably.

This is more complicated while file transfers are occurring. Small file transfer events don't impact the disk writes or CPU usage significantly enough to affect this detection. However, larger file transfers similarly cause large spikes in both CPU usage and disk activity. Figure 3.6 demonstrates the effect of a 10GB file transfer to D using GlobusFTP on various performance metrics over the course of two minutes. Since there are generally few programs running on the DTN, the baseline CPU and disk activity on the DTN remains low. When the transfer occurs around 20 seconds, we see a large spike in data being written to the disk and CPU usage, as expected. Other metrics like interrupts and context switches increase slightly during a large file transfer, and more than during file editing, but not significantly enough to be useful for detection. Therefore, these transfers

can produce similar activity and it's conceivable that an attacker might attempt concealing file tampering by editing files while a large file transfer is ongoing.

In order to detect data tampering while a large file transfer event is occurring, we need to consider how the host performance differs. The most obvious difference is the insider must edit the files through user processes, meaning user CPU usage will increase noticeably during these file editing events. We apply DBSCAN to cluster normal user CPU activity and detect outliers. Figure 3.5 shows how the clustering appears when the `tstat` files are edited during a 10GB file transfer. When a small amount of file data is modified, short spikes in the user CPU usage will create outliers, while normal variations form the largest, primary cluster. Sustained periods of file editing will form a smaller clusters above the primary cluster. The appearance of either outliers or an unexpected cluster, combined with a spike in disk reads and writes, indicates files are being overwritten. Clustering can reliably identify on-going data modification, even if it coincides with a large GlobusFTP transfer.

Although the detection is effective, it is unable to detect small file modifications which do not significantly increase the CPU usage. Looking at figure 3.8, we see that if the total data overwritten is below 10MB, we cannot reliably detect that event, because it falls within the normal performance range. Therefore, it is possible an attacker could evade detection by only editing small portions of data at a time. Future work should consider what additional metrics could be leveraged to detect this form of data tampering. However, most of the science data arriving on the DTN is likely to be in the form of large files, and modifying these files will necessitate disk activity above the threshold for detection.

3.6.2. Data Exfiltration. Data exfiltration can be performed through ordinary GridFTP transfers, or by sending data through an obfuscated channel. In the first case, it is likely that the transfer could be detected by ordinary security measures, such as Zeek network intrusion detection. However, if it is being leaked through obfuscated protocols, an insider could extract the data while evading detection. Therefore, we must be able to detect the PDF obfuscated SSH sessions.

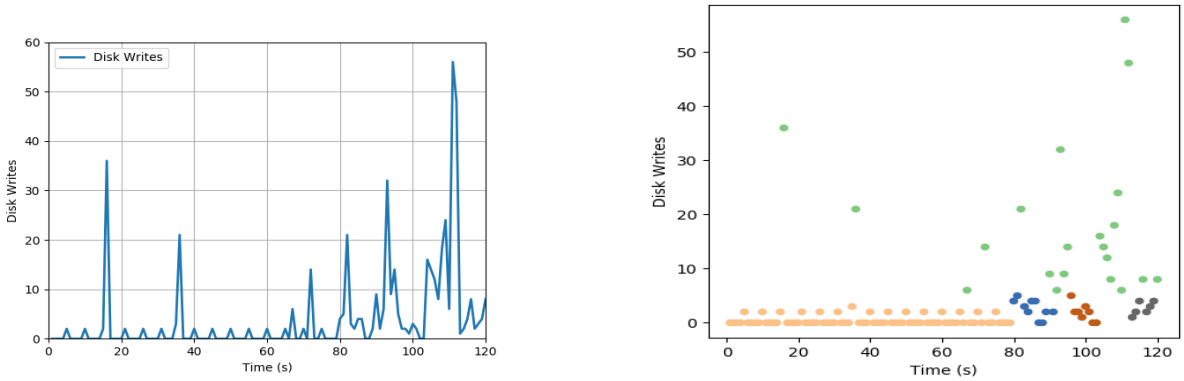
The PDF obfuscation method has a clear impact on the system, because communication between the client and server depends on frequent writes to the PDF files. Figure 3.7a shows how the disk writes increase during the obfuscated SSH session. Figure 3.7b shows how this increase

in disk writes appears when clustered. The consistently higher disk writes per second leads to new clusters being formed in addition to the primary cluster representing the baseline disk writes. During periods with no file transfers to the DTN, this is simple to detect. However, files transfers could also cause sustained spikes in disk writing. If an attacker can ensure their activity on the DTN coincides with these transfers, detection could become more difficult. We can achieve more reliable detection by considering the ratio of disk writes to the amount of data written per second. Since the PDF obfuscation program writes only small amounts of data to the disk each time, the ratio will be much larger than during most file transfers.

3.6.3. Real-Time Detection. Since the user CPU usage and disk writes remain within a small enough range under normal conditions, we do not need a large sample size to identify file modification or the obfuscated SSH channel. In our experiments, we found that 10 data points was sufficient to detect these anomalies through clustering. Therefore, DBSCAN can use a low **MinPts** values, and the detection can be performed quickly. After collecting 10 seconds worth of data, the values are clustered and checked for anomalies. If noise points or more than one cluster appears, we report an anomaly. This method was able to reliably identify file modification without false positives past 10MB, where the increase in user CPU usage becomes large enough to form outliers and new clusters (Figure 3.8). By clustering the ratio of disk writes to data written to disk, we could also reliably identify an on-going obfuscated SSH session. The obfuscated SSH session performs more disk writes when characters are being typed, which will lead to a new small cluster being formed. If two or more total clusters appear, we can identify the anomaly. Sending three characters at a normal pace in the obfuscated SSH session is sufficient to create a new cluster and trigger the alert. Therefore, the operations the attacker can perform using the obfuscated SSH session are severely limited, and exfiltrating large amounts of data through the obfuscated channel is no longer feasible.

3.7. Conclusions and Future Work

This work suggests that system performance metrics such as CPU usage and disk writes, along with network performance metrics such as the amount of incoming traffic, can be used together to help identify unwanted data modification in a DTN environment. The limited range of applications



(a) Disk writes over time

(b) Disk write clusters

FIGURE 3.7. Disk write activity can detect an obfuscated SSH session. The obfuscated SSH session begins at 80 seconds. Although this obfuscation method can evade traditional detection methods, the frequent writes to PDF files can be used for detection. The three new small clusters appearing after 80 seconds indicate obfuscated SSH activity.

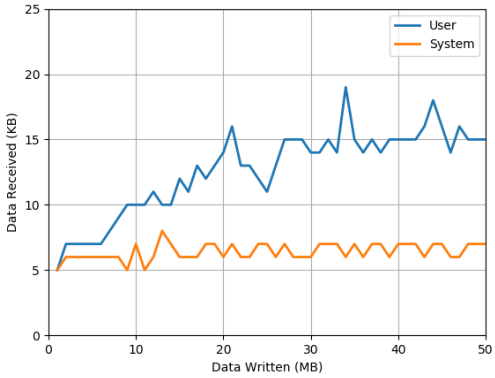
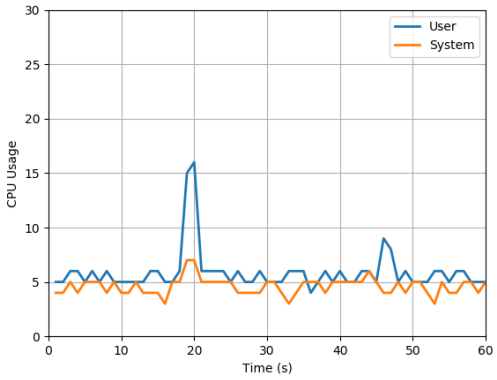
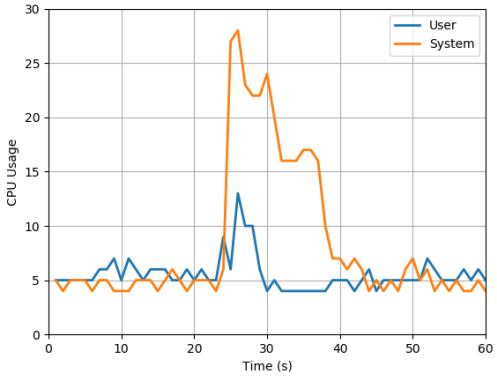


FIGURE 3.8. Change in CPU usage during file editing events. We can see that user CPU usage increases as more file data is overwritten, while system CPU usage remains relatively stable.

and predictable system performance of the DTN environment allows clustering based anomaly detection using DBSCAN to function effectively. Using DBSCAN clustering to detect abnormal CPU activity, along with checking for coinciding disk and network activity, we can predict when unexpected file editing is occurring and distinguish it from a large file transfer, even if the insider has taken efforts to conceal it from other means of detection. Furthermore, this detection can be used in real-time by repeatedly clustering the performance metrics gathered by common tools such as `collectl`. In the future, this form of detection could be enhanced by considering additional



(a) CPU activity during data editing



(b) CPU activity during data editing alongside file transfer

FIGURE 3.9. CPU Usage over the course of editing 50MB worth of files. The user CPU usage increases significantly more than the system CPU usage. This spike in user CPU usage is visible even if the editing occurs during a large data transfer.

performance metrics, as well as combining it with other methods of insider attack detection, such as system call monitoring.

CHAPTER 4

Covert Timing Channel Detection

4.1. Introduction

A covert channel allows unauthorized data transfers through authorized channels. This can allow harmful exploits such as controlling botnets or leaking private data [**12**]. In this work, we will focus on detecting network covert timing channels, which function by encoding data inside the inter-packet delays (IPDs) of a network flow. A very basic channel type would be Cabuk's IP covert timing channel (IPCTC) [**46**], a simple on/off channel, where a packet transmission during a set time interval will be interpreted by the receiver as a 1, while no transmission during that interval will be interpreted as a 0 [**46**]. This encoding scheme, although functional, creates traffic where the shape and regularity differs greatly from the original, overt traffic, making detection simple [**13**]. More advanced timing channels attempt to mimic real traffic statistics to bypass detection. For example, Time-Replay Covert Timing Channels (TRCTC) uses two legitimate IPD sets, producing a 0-bit or a 1-bit by replaying according to one of the two sets. This encoding scheme makes the new traffic appear similar to overt traffic, complicating detection [**13**]. CTCs are one possible means of exfiltrating Science DMZ data. Although the capacity of these channels is low, a short-lived CTC could be exploited to extract a small amount of sensitive data passing through the DTN, while a long-running CTC could be used to slowly leak data over time. Therefore, we want to consider methods of detecting these channels in high-speed networks.

Network security applications must utilize new techniques in order to process packets at network line rates of 10 Gbps, and eventually 40 and 100 Gbps. For scaling these tasks, multi-core CPUs and other parallel systems such as Massively Parallel Processing Array (MPPA) or Field Programmable Gate Arrays (FPGA) architectures can be applied. However, these approaches each have their own limitations in terms of programmability, performance, and flexibility. Containing thousands of cores, Graphics Processing Units (GPUs) possess much greater thread-level parallelism and memory

bandwidth compared to CPUs [**47**]. Therefore, using the GPU presents another possible way to scale real-time software-based packet processing, and several papers have already demonstrated its effectiveness for tasks such as software routing [**48**] and firewall packet classification [**49**]. Covert timing channel (CTC) detection is one possible network security application which can benefit from GPU packet processing. CTCs exploit the timing of the inter-packet delays (IPDs) in authorized network flows to embed hidden data transfers. Detecting CTCs in real-time requires performing statistical tests on a large number of incoming flows and comparing the results with those expected for legitimate traffic. Since the test scores for each flow are independent, a GPU can process many flows in parallel. Furthermore, the GPU can also perform the tests on individual flows in parallel, allowing real-time usage of more complex and effective detection tests such as the corrected conditional entropy of the IPD sequence, which could not be included in our previous CTC detection experiments [**12**].

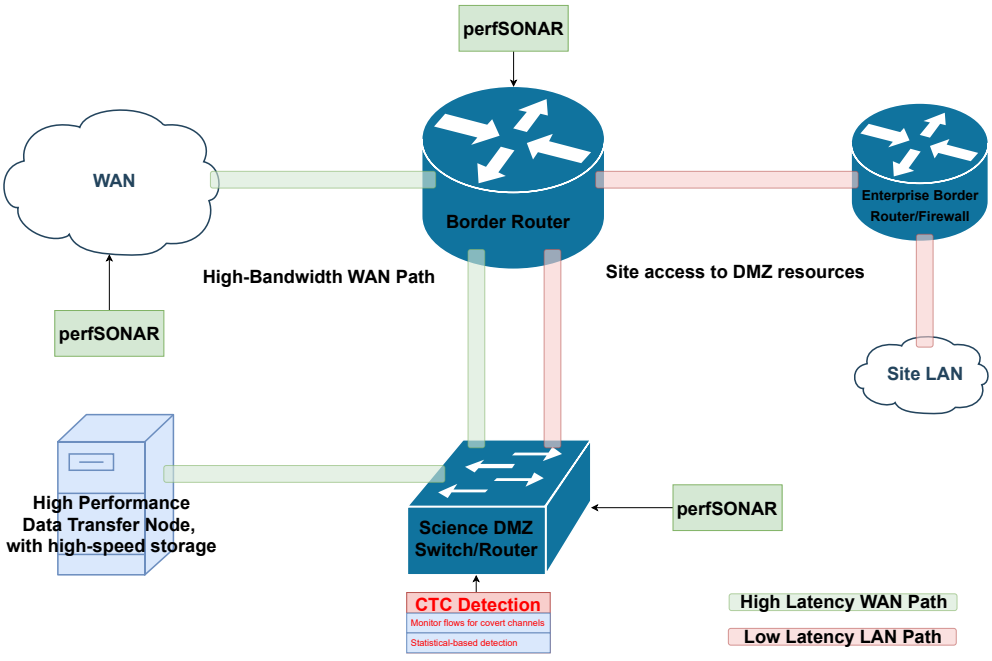


FIGURE 4.1. A covert timing channel detection tool can be placed at the Science DMZ router. Using statistical-based anomaly detection, we can monitor for flows containing suspicious inter-packet delays. [**7**].

Although previous work shows that the first-order entropy alone can be somewhat effective for detection, the false positive rate is still high [**12**]. A more effective detection method requires calculating the corrected conditional entropy (CCE), which is the conditional entropy calculation plus a corrective term accounting for the number of unique subsequences in the sample. The CCE test has proven effective for detecting a variety of CTCs with minimal false positives [**13**]. Outside of CTC detection, the CCE test has a variety of applications, particularly in medical imaging applications, such as analyzing heart rate variability data and other biological processes [**11**]. However, calculating the CCE score for a large sequence is an expensive calculation computationally, requiring the construction of a tree for each individual flow [**13**]. For larger traffic rates with a large number of flows arriving each second, we need to calculate the CCE score for each flow more efficiently. In order to perform the calculation quickly, we propose that packets be processed on the GPU. The corrected conditional entropy formula and our GPU algorithm will be explained in detail in section 4.3.

To evaluate the large number of incoming flows without packet loss, we implement a detection tool that performs the CCE test using a NVIDIA Tesla K20C GPU. Our tool sniffs incoming traffic and gathers packet data into large batches, which are then tested on the GPU. The GPU will use the CCE calculation to report which flows are likely to contain covert channels. In our work, we consider a well-known CTC variety known as model-based covert timing channels (MBCTCs), which avoid detection by fitting the CTC's packet timings to a statistical model based on natural traffic [**50**]. By testing our tool against a traffic sample injected with MBCTCs, we confirm the CCE test's effectiveness as a classifier established in previous results [**13**]. We also evaluated the maximum performance of our tool, establishing that it can handle close to a full 10 Gbps line rate assuming average sized packets.

Implementing real-time CTC detection at high data rates requires capturing and storing large amounts of packet data by flow, and then performing the detection test in a small amount of time (less than 67.2 ns per packet for minimum-sized packets) to avoid packet loss. The limited time available to process each flow makes it especially difficult to perform more complex calculations such as the CCE test. While GPU packet processing offers good potential performance, mapping the problem is particularly difficult because the CCE test requires us to construct and process a k-ary

tree for every flow, every few seconds. Since tree construction is a difficult task for GPUs [**51**, **52**], we need an alternative method of calculating CCE scores. Since the CCE calculation uses k-ary trees, each tree can be interpreted as an array. Therefore, our solution involves transforming the tree structures into arrays, a form well-suited for GPU processing.

This work presents multiple new contributions. The most significant are as follows:

- Our detection tool marks a significant improvement over previous CTC detection work. In our experiments, we manage to detect nearly 100% of our sample's CTCs.
- In addition to confirming the CCE test's effectiveness, we achieve higher rates compared to previous real-time detection experiments [**12**]. We achieve close to 10 Gbps using medium-sized packets.
- By performing our tree transformation and completing the calculation using arrays, we compute the CCE scores in less than 1 ms per flow, an order of magnitude faster than previous results [**13**].
- Our work also demonstrates how GPU packet processing can efficiently calculate complex individual flow statistics using packet batches.

The rest of the chapter will provide further background on the CCE test and CTC detection, as well as our experimental results and discussion. In Section 5.1 we discuss our motivations for this project. In Section 4.4 we discuss our tool design in-depth, the CCE algorithm, and the experimental setup. In Section 4.5 we discuss the experimental results. Section 5.5 describes some related work on GPU-based packet processing and CTC detection. Finally, in Section 5.6 we give the conclusions and describe the potential for future work.

4.2. Timing Channels and Detection

In this work, we focus on Model-Based Covert Timing Channels (MBCTC). MBCTCs evade detection by mimicking legitimate traffic's shape and regularity. To achieve this, the channel first analyzes outgoing flows to determine an appropriate statistical model, then encodes the message inside the flow's IPDs using that model's inverse distribution function. The channel can then be decoded by the receiver using the cumulative distribution function [**13**]. Figure 4.2 illustrates the

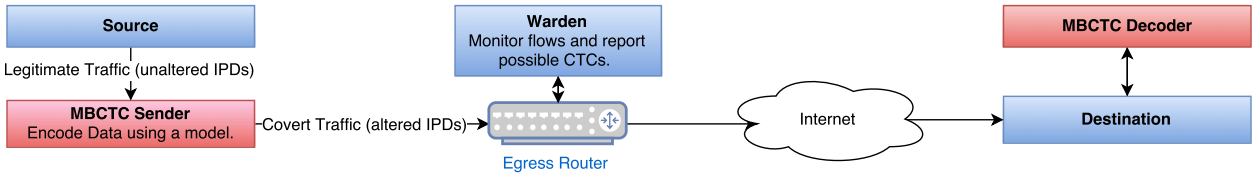


FIGURE 4.2. Model-based covert timing channel detection model. A warden placed at a router monitors traffic and reports unusual flows.

process. While this encoding prevents detection by common detection methods such as Kullback-Liebler Divergence and the Kolmogorov-Smirnov Test, CCE detects MBCTCs with near-perfect accuracy [**13**].

4.2.1. Detection Methods. Ideally, we would want to eliminate all possible CTCs. Existing methods such as the network pump [**53**] and fuzzy time [**54**] introduce noise that alters packet timings, reducing a covert channel's reliability and capacity. However, applying these techniques to all incoming flows will harm legitimate traffic performance as well. Therefore, we want to first detect likely CTC flows, then disrupt them selectively. Basic channels—such as IPCTCs—are relatively simple to detect, but other channels can closely resemble the unaltered traffic. CTCs can be detected by measuring the shape and regularity of network traffic and comparing it to the expected legitimate statistics [**46**].

CTC detection tests can be grouped into regularity tests and shape tests. Shape tests represent first-order traffic statistics, including the Shannon entropy or Kolmogorov-Smirnov score, which measures the greatest difference between two IPD distributions. Regularity tests represent higher order statistics, such as conditional entropy [**13**]. Different detection tests will be effective depending on the channel type [**12**]. IPCTCs alter both traffic shape and regularity, allowing detection by a variety of tests. TRCTCs closely resemble the natural traffic's shape, but since the packet timings do not correlate naturally, the regularity scores will be significantly different than normal traffic. MBCTCs closely resemble both the regularity and shape of unaltered traffic, making most detection tests ineffective [**12**]. However, the CCE test, which is a modified conditional entropy measurement, effectively identifies MBCTCs [**13**]. Our work focuses on efficiently implementing the CCE test for real-time detection.

4.2.2. Motivation.

Improved Real-Time Detection. Although many papers have been written on CTC detection, few consider real-time detection in streaming data. Our previous real-time CTC detection work used an MPPA architecture [**12**] (Tilera TilePro64 NIC). The incoming flows were equally assigned to the different cores. Each core, acting independently, used sample-and-hold [**55**] to identify large flows. After a flow was identified as large, and potentially harboring a CTC, a histogram was constructed representing that flow's inter-packet delays (IPDs). Once enough packets were gathered for a given flow (typically 1,000), a detection test such as the first-order entropy test was performed using the histogram. If the score exceeded a certain threshold, the flow was reported as a CTC flow. Our results demonstrated that distributing the network flows evenly to the individual cores of the Tilera TilePro64 MPPA architecture significantly reduced the number of packet drops at high data rates and consequently improved the covert timing channel detection rates. With the exception of the first order entropy test for detecting model-based covert timing channels, the detection tests included in our initial implementation are mostly poor classifiers. The work provided a relatively functional and scalable framework for real-time covert timing channel detection at relatively high data rates. However, the TilePro64 architecture proved to be overly complicated to use, not widely available, and lacked the flexibility of multicore CPUs and GPUs, making it poorly suited for widespread detection applications.

Although the MPPA detection tool could detect real-time CTCs with some success at line rates around 2 Gbps, there were multiple limitations. Since the memory on a core is fairly small, only a relatively small amount of packet data could be kept on each one. In addition, by having each core handle different sets of flows, the tests themselves could not be parallelized across cores. Both of these factors limited the types of possible detection tests to simple ones such as first-order entropy, Kullback-Liebler divergence, and the Kolmogorov-Smirnov test. While the results show that the first-order entropy test is a decent classifier for MBCTCs, it is outclassed by the corrected-conditional entropy (CCE) test, which could not be implemented on the MPPA tool. Previous results have shown that the CCE test is an excellent CTC classifier, capable of identifying multiple covert channel types with low false positive rates [**13**].

Compared with specialized hardware like FPGAs or MPPAs, GPUs are more commonly available in existing systems. Therefore, GPU-based packet processing would be more valuable for

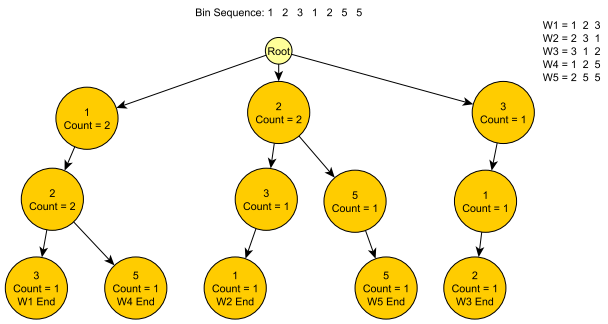


FIGURE 4.3. CCE k-ary tree [**50**]. $k = 5$ bins, window size = 3. The bin sequence is divided into 5 windows, each representing a path through the tree. Each node maintains a count of how many windows have passed through it. The counts are then used to calculate the CCE score. Algorithm 1 describes how this same structure can be represented using arrays.

implementing real-time CTC detection. Some existing GPU packet processing tasks include pattern matching and packet routing [**47**]. In addition, GPUs have previously been used to accelerate mutual information calculations, a measurement that shares similarities with conditional entropy [**56**]. Calculating the CCE scores for a large numbers of incoming flows in real-time should also demonstrate more complex GPU calculations on streaming packet data are plausible. The primary motivation behind this project was to determine whether or not CCE-based detection could be effectively implemented using GPU packet processing.

Efficient Entropy Calculation. In addition to detecting covert channels, entropy measurements have a variety of applications. Corrected conditional entropy in particular is useful for evaluating heart rate data and other bioinformatics [**11**]. Conditional entropy alone is useful for a variety of applications, including analyzing financial time series data [**57**]. Transfer entropy has applications in data mining and neuroscience, among other uses. For example, it can be used to measure a person's influence on social media such as Twitter, or to measure the connectivity in brain regions [**57**]. However, for a large value series, entropy calculations can become very time-consuming [**57**]. Therefore, improving the efficiency of entropy measurements using GPU processing is a worthwhile pursuit beyond its applications for covert channel detection.

4.3. Corrected Conditional Entropy

The corrected conditional entropy (CCE) test is simply the conditional entropy with a corrective term consisting of the Shannon entropy multiplied by the percentage of unique subsequences added. The Shannon entropy, or first-order entropy, measures the amount of randomness of a random variable. The formula is as follows:

$$H = -\sum_{i=1}^{n} P(x_i) \log P(x_i) \tag{4.1}$$

with $P(x_i)$ referring to the probability of selecting the value x_i. The conditional entropy (CE) test, which measures the randomness of a variable given the value of another variable, can be calculated as follows for a sequence of random values:

$$H(X_i|X_1..X_{i-1}) = H(X_1..X_i) - H(X_1..X_{i-1}) \tag{4.2}$$

The CCE formula is as follows:

$$CCE(X_m|X_1..X_{m-1}) = H(X_m|X_1..X_{m-1}) + P \times H \tag{4.3}$$

with P representing the percentage of uniquely occurring sequences and H being the Shannon Entropy [**13**]. This corrective term is necessary, because with finite sequences the conditional entropy tends towards zero as the sequence grows longer, while the corrective term will increase [**13**]. The reason for this is that the conditional entropy calculation requires finding the Shannon entropy values for each subsequence. As the sequence windows get longer, the more likely we will have no repeating subsequences, giving us a final entropy value of 0. With CCE, the corrective term ensures that the score will not continue to decrease towards 0. The maximum CCE score possible is equal to the first-order entropy score [**13**]. In the context of CTC detection, the sequence we use to calculate the CCE is the sequence of bin numbers determined by the inter-packet delays (IPDs). Each IPD is compared to a range of values based on training data and assigned a bin between 0 and 4 inclusive. The minimum CCE scores can be used to reliably detect common varieties of CTCs, including IPCTC, TRCTC, and MBCTC.

To calculate the minimum CE score, we calculate the Shannon Entropy for each subsequence length from 1 through N, where N is the maximum subsequence length. Then, after calculating all the entropy values, we find the minimum entropy score. The typical way to calculate the CE score is to create a perfect k-ary tree, where k is the total number of bins. The IPD bin sequence for a flow is divided into subsequence windows, which have sizes equal to the tree height. Figure 4.3 demonstrates the tree used to calculate the CE for a small sequence of seven IPD bin values. Each node of the tree contains a count representing how many times a subsequence has passed through that node. For example, the leaf nodes represent a full window sequence, while the root's children represent the first value in a sequence. These counts are used to calculate the Shannon entropy for each level of the tree, and the minimum value gives the CE score for that network flow. The tree-based minimum CCE calculation functions the same way, but the corrective term is added to the entropy score for each tree level.

Algorithm 1 Steps for calculating the corrected conditional entropy for the tree shown in Figure 4.3 using only arrays. The window size is 3, equal to the height of the tree.

input: IPD sequence array $A = [1, 2, 3, 5, 1, 2, 5]$.
output: IPD sequence's corrected conditional entropy.
1. Convert the sequence into windows of size 3. $[[1, 2, 3], [2, 3, 5], [3, 5, 1], [1, 2, 5]]$
2. Convert each sub-sequence to a single base 4 value by combining the value in that sub-sequence. For example, in the first window, the sub-sequence of length 3, $[1, 2, 3]$ will become $1 + 2 * 4 + 3 * 16 = 57$. The windows will now be $[[1, 9, 57], [2, 14, 94], [3, 23, 39], [1, 9, 89]]$
3. Arrange the values such that tuples of equal length are together. We now have $[[1, 2, 3, 1], [9, 14, 23, 9], [57, 94, 39, 89]]$
4. Sort the values. $[[1, 1, 2, 3], [9, 9, 14, 23], [39, 57, 89, 94]]$
5. Count how many times a number appears in the same group. $[[2, 1, 1], [2, 1, 1], [1, 1, 1, 1]]$
6. Now, each group corresponds to the counts at a level of the tree. Using these counts, we can calculate the corrected conditional entropy at each level as we would using the tree.
7. Finally, take the minimum of these values to obtain the final CCE score.

4.3.1. GPU Entropy Calculation. Although using trees to calculate entropy works, it requires too much time to calculate the CCE score for long sequences of packets. After dividing the sequence of values into windows, the standard k-ary tree-based CCE calculation requires updating each node's count as it is visited while moving in a path from the root to the leaf nodes, repeating for each window in the sequence. Once the counts have been calculated, the CCE score is calculated for each level of the tree and the minimum value is selected as the final score. A previous paper

showed this method requires 16 ms to calculate the CCE score for a single flow using a 3.4 GHz Intel Pentium D [**13**]. For high data rates, the need to process a large number of new flows constantly arriving necessitates a more efficient means of calculating the CCE score for each flow.

For this reason, we chose to perform the CCE calculation using the GPU. However, constructing thousands of large trees in real time on a GPU is difficult to perform efficiently. Rather than dynamically constructing a tree for each flow on the GPU, we instead represent each flow's tree within a single large array. Since the number of bins and window size for the trees is predetermined, the CCE calculation can be performed by first dividing each flow's portion of the array into windows of size N, then counting the number of matching sub-arrays of different lengths from 1 to N. Each sub-array represents a partial path through the tree, and is stored as a single base-4 value for comparison. Although some different sub-arrays will have the same base-4 translation, it is less likely for longer sub-arrays and simplifies the matching process. The conversion of a CCE tree into an array representation is further explained in Algorithm 1, which calculates the CCE score using the same IPD bin sequence used to create the tree shown in Figure 4.3.

This approach simplifies performing the CCE test on the GPU and has multiple advantages. Since the calculation can be performed entirely using arrays, we could implement the CCE calculation using NVIDIA's CUDA Thrust template library. Thrust is more manageable than raw CUDA kernels and easily portable with multicore CPUs using OpenMP or TBB [**58**]. In its current state, the algorithm expresses the nodes and counts for a k-ary tree in array form. However, with modifications, other types of data can be stored at each node. Other calculations that require binary or k-ary trees could benefit from using similar methods. Assuming we already know each node's maximum number of children, a tree could similarly be flattened into an array.

4.4. System Design and Experimental Setup

Our detection tool is a heterogeneous system, using both the CPU and GPU to calculate flow statistics for an incoming packet stream. Similar to PacketShader and other GPU-based packet processing systems [**48**, **59**], our detection tool gathers first gathers packets into large batches on the CPU, then sends those batches to the GPU for processing. The tool uses two threads to allow an overlap between CPU and GPU operations. One thread receives packets, timestamping them

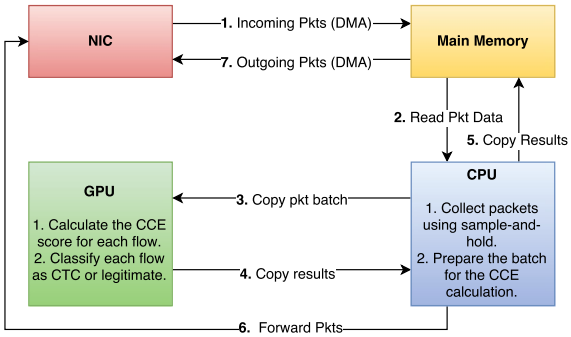


FIGURE 4.4. Basic GPU-based packet processing model, adapted from Mukerjee et al. [**59**]. The GPU receives packet batches from the CPU, processes them and returns a result.

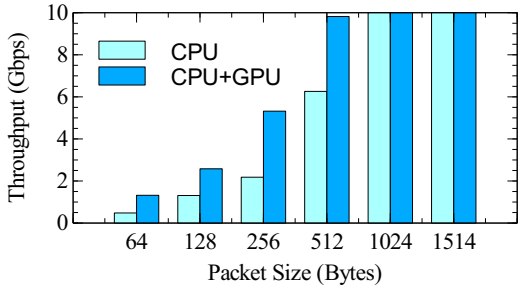


FIGURE 4.5. CPU vs. GPU maximum throughput achieved for different packet sizes. Although the GPU does not make much difference for large packet sizes, it significantly improves throughput for small and medium packet sizes.

and placing them into a lockfree queue, while the other removes packets from the queue and stores them into batches. Once enough packets have been obtained to form a complete batch, that batch is then copied to the GPU for processing. The processing consists of two steps. First, the GPU gathers converts the batch array into a smaller arrays consisting of only packets belonging to flows with enough packets to accurately perform the CCE test. Second, the GPU takes this modified array and calculates the CCE score for each flow it contains. If a flow's reported CCE score is under a certain predetermined threshold, then that flow will be reported as containing a model-based covert timing channel (MBCTC). Figure 4.5 compares the performance with the OpenMP CPU-only version of the tool. Figure 4.4 gives an overview of how the system processes packet data.

4.4.1. PF_RING Packet Capture. Our CTC detection tool prototype uses PF_RING ZC (zero-copy), a NUMA-aware packet processing framework developed by ntop to receive packets at line rate on a 10 Gbps ethernet link [**60**]. Similar to Intel's DPDK [**61**] or netmap [**62**], PF_RING ZC bypasses the standard network stack, accelerating packet processing. Using DMA, the NIC copies packet data directly to memory, rather than copying between the kernel and user space. When beginning an application, PF_RING ZC establishes a packet buffer to avoid any memory allocation during execution. Rather than receiving packets through interrupts, PF_RING ZC polls for packets [**63**]. According to ntop [**60**], PF_RING ZC performs better than DPDK for smaller packets. Our packet sniffing code builds on ntop's zcount example program, which receives and counts packets at 10 Gbps line rates regardless of packet size.

4.4.2. Batch Processing. The packet I/O code is based on a modified version of pf_ring's zcount example. One thread is dedicated to collecting and timestamping incoming packets. The thread puts the raw packet pointers and timestamps into a lockfree queue. The processing thread continuously reads packet data from this queue, obtains the four-tuple identifying the flow (source ip and port, destination ip and port), and stores them in a buffer. Since we assume only large flows contain CTCs, and 80 percent of flows contain no greater than 20 packets [**64**], we use "sample-and-hold" [**55**] to reduce the amount of packet data stored. For each incoming packet from a new flow, there is a small chance (about 0.5% in our case) that it and all further packets in that flow will be stored. Therefore, only large "Elephant" flows are likely to be stored in the buffer [**65**], reducing memory usage and creating a buffer containing mostly flows capable of carrying a high capacity CTC. Since the CTC flows we can detect will be very large (thousands of packets or more), we can afford to use a very low sample-and-hold probability. This processes continues until enough packets are gathered in the buffer to send a large batch to the GPU for testing. By default, the batch size is set to 12,500,000 packets. Larger batch sizes will increase bandwidth, but also latency.

Once enough new packets are stored, the packet data in the buffer array is prepared for the CCE calculation. First, the array containing the IPDs is converted into a new array containing only flows with enough data to obtain an accurate CCE score, typically between 500 and 2,000 packets. Then, we calculate the adjacent difference for the packet timestamps to obtain the inter-packet delays (IPDs). Using equiprobable binning, the IPDs are converted to a bin value between 1 and 5,

with 5 representing the largest IPDs. If there are enough eligible flows, this modified batch array is copied to the GPU to perform the CCE calculation using Thrust. Although our approach will accurately detect CTCs in a batch, one issue is that processing packets in batches will inevitably capture only a fraction of the large flows. For example, assume we set the CCE test to process flows with 2000 packets or more, and the batch only contains the first 1000 packets of that flow. If the flow is only slightly larger than 2000 packets, the flow will not be reported, because not enough of its packets were present in any given batch. Therefore, our current approach can only sample a fraction of the overall large flows. Using our trace file with a 500 IPD threshold, we captured around 98.4% of the large flows, and 91% when replaying it together with near 10 Gbps traffic. However, we assume CTC flows are large and long-lived [**12**], and therefore a more significant portion of potential CTC flows are likely to be sampled in their lifetime. CUDA Thrust allows GPU code to be updated and tested quickly, while also being portable with multi-core CPUs.

4.4.3. Experimental Setup. For our setup, two PowerEdge T630 machines (a sender and a receiver) were connected by a 10 Gbps Ethernet connection. Both the sender and receiver contained two Intel Xeon E5-2637 v3 3.5GHz processors. The receiver contains a PowerEdge T630 GPU along with a NVIDIA Tesla K20C GPU accelerator, which is used for our experiments. Designed for general purpose computing, the Tesla K20C has 2496 CUDA cores, 208 GB/s memory bandwidth, and 5 GB GDDR5 memory [**66**]. The sender will transmit flows to the receiver, which sniffs incoming traffic and processes the packets in batches using the CCE test. By running our tool in this way, we performed a variety of tests, evaluating the CCE test's effectiveness of as a classifier, measuring the packet processing time, and the various trade-offs in our implementation between memory usage, latency, and throughput on the GPU. From a CAIDA repository [**67**], we used a real traffic trace containing one minute of traffic from a 10 Gbps San Jose OC-192 link. The trace file has been anonymized, meaning it only contains packet headers and therefore minimum-sized packets. This same trace was used in our previous real-time detection experiments [**12**]. We modify the pcap file by replacing roughly 10% of the flows containing greater than 1000 packets with MBCTC flows. The threshold choice of 1000 IPDs was chosen based on previous detection experiments [**12**,**13**]. Table 4.1 describes the CTC-containing trace file we used for our experiments.

TABLE 4.1. MBCTC trace file statistics. Large flows contain 1000 packets or more.

Total Packets	Total Flows	Large Flows	Legitimate	CTCs
33581932	631089	3377	3056	321

TABLE 4.2. Experimental Parameters

Parameter	Definition
True Positive Rate	Percentage of tested CTC flows correctly classified as CTCs.
False Positive Rate	Percentage of tested legitimate flows incorrectly classified as CTCs.
Window Size	The IPD bin sequence window length (CCE tree height).
IPD Threshold	The number of IPD bin values per flow for calculating CCE scores.
Batch Latency	The time spent gathering packets before processing a batch.
Maximum Throughput	The maximum data rates achievable without dropping packets.
Batch Threshold	The number of new packets required before testing a batch.
Batch Setup Time	The time spent preparing a packet batch for the CCE test.
CCE Test Time	The time required to calculate CCE scores and report flows as CTCs or not after preparing a batch.

For sending packets, we used two different tools—tcpreplay and zsend. Depending on the experiment, we send packets using either tcpreplay, zsend, or both. Tcpreplay replays pcap files while maintaining accurate timing information, meaning we can use it for our detection rate tests. PF_RING ZC's zsend tool allows us to blast random packets at 10 Gbps line rate for testing the maximum data rates our system can handle without packet loss. By setting the packet size, zsend lets us test the maximum data rate at varying packet sizes. The two tools can be combined by replaying with tcpreplay while running zsend. By combining the two, we embedded our trace file while sending larger amounts of traffic to test detection at high data rates.

4.5. Results and Discussions

We performed tests on the tool's ability to identify CTCs using the CCE test, as well as the performance. Table 4.2 defines the important system and experimental parameters.

4.5.1. Classification Results. To ensure our tool functions properly, we measured the true and false positive rates for the CTCs reported after receiving all the packets from our trace file. In this case, a true positive refers to a flow being correctly reported as containing a CTC. As Figure 4.6 demonstrates, the corrected conditional entropy test score performs well at classifying MBCTCs. Even while sending our trace file embedded within 10 Gbps traffic, the CCE scores

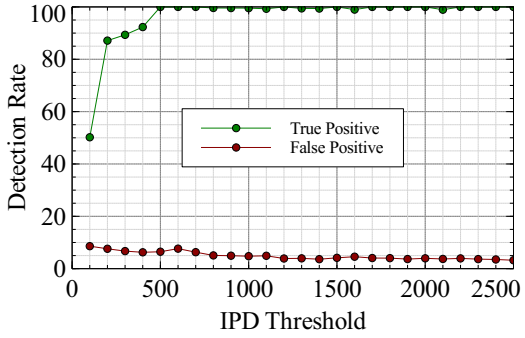


FIGURE 4.6. MBCTC true and false positive rates vs. the amount of IPDs tested per flow. Flows with CCE scores < 0.4 were reported as CTCs. Setting the IPD threshold to 500 or more is sufficient.

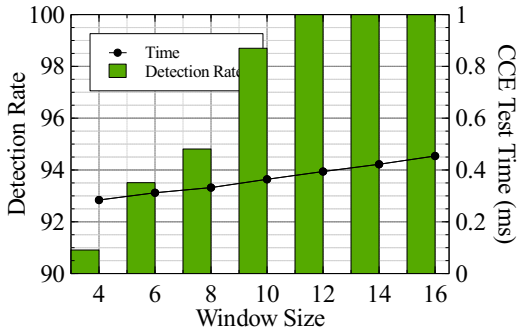


FIGURE 4.7. CCE test time per flow and MBCTC true positive rates vs. the window size used to calculate the CCE score. The false positive rate for these values was roughly 2.5% and the IPD threshold was set to 2000.

remain accurate. The percentages nearly match previous results [**13**], which reported a 95% true positive rate with a 1% false positive rate when testing a sample of 2000 packets. By tweaking the score threshold for reporting CTCs to reduce the false positive rate to the same value, we also obtain a true positive rate around 95%. For our results, we report a possible CTC if the CCE score is less than 0.4, ignoring outliers with CCE scores near zero. However, this threshold can be altered depending on how many false positives can be accepted. The response to a reported false positive could be to add noise to that flow through fuzzy time or other techniques [**53**, **54**]. Since that flow's packets will still arrive, albeit at a reduced rate, some false positive may be allowed depending on the application. However, a stricter false positive rate could be necessary for some applications such as VoIP.

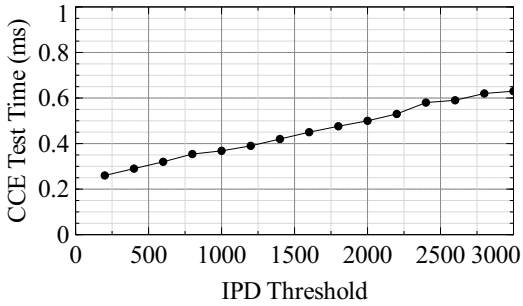


FIGURE 4.8. CCE test time per flow vs. the IPD threshold used to calculate the CCE score. Window Size was set to 10. Increasing the number of IPDs used in the CCE calculation increases the time to complete the test.

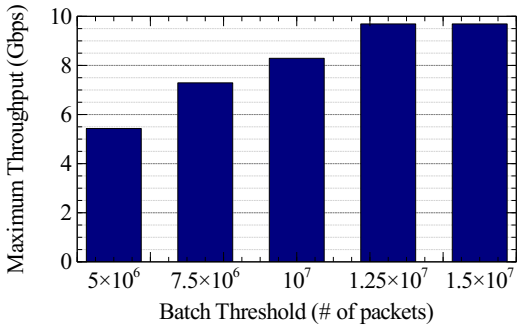


FIGURE 4.9. Maximum throughput achieved with the GPU using different batch testing thresholds.

There are trade-offs to consider when selecting the window size and IPD threshold used for calculating the CCE score. Using a larger window size and testing more IPDs will give more accurate detection (Figure 4.7). However, that will also increase the time required to process the flows, and require more memory per flow. Figure 4.7 shows that the time per flow can increase significantly. Ideally, the smallest possible sample and window size should be used to handle higher data rates without packet loss. The detection rate for our sample stopped improving significantly at 2000 IPDs and window size 10. With those parameters, it takes around 0.4 ms per flow to calculate the CCE score for a batch.

4.5.2. GPU Packet Processing Results. In order to test for CTCs, we gather packets into large batches. Once enough packets are gathered, the packet batch is copied to GPU memory, and each flow in the batch is tested for CTCs. Before copying to GPU memory, the batch must

be setup for the CCE calculation. This involves culling all flows below the IPD threshold, and ensuring that all remaining flows have an equal number of IPD bin values. The batch threshold affects the test result latency and throughput. A larger batch threshold increases the throughput by processing more flows per batch (Figure 4.9). Larger batch thresholds also increase the latency between receiving enough packets to perform the CCE test on a flow and reporting whether or not it contains a CTC (Figure 4.11). Assuming we test 2,500 IPDs per flow, we can test 3,000 large flows per batch without running out of memory during the Thrust CCE calculations. This translates to a maximum batch size of 7,500,000 packets. However, it is improbable that all the flows in a batch will each contain exactly 2,500 IPDs. Therefore, batch sizes larger than 7,500,000 are possible if larger throughput is desired, provided the number of flows to be tested is limited to 3,000 or less. We obtained the best results using a threshold of 12,500,000 or more, as shown in Figure 4.9.

In Figure 4.5, we show the highest possible rates we could achieve with different packet sizes on a CPU and GPU implementation of our detection tool. To measure this, we have our sender machine blast packets to the receiver using PF_RING's zsend application for five minutes, then report whether or not any packets were dropped. Since zsend cannot specify the number of packets per flow, we assign a random flow id between 0 and 14999 to each incoming packet, ensuring that flows are large enough to pass the sample-and-hold test and be stored in a batch. In order to simulate a heavy workload with hundreds of large flows being processed every batch, each batch is randomly assigned between 0 and 1,500 flows over the IPD threshold per batch for which to arrange and calculate the CCE score. On average, around 750 flows containing over 2500 IPDs will be tested per batch using this method. Assuming average 512 byte packets, our tool can handle 10 Gbps traffic at near line rate (about 9.69 Gbps, or 2,200,000 pps). However, higher rates should be possible when testing with real traffic samples, depending on how many flows pass the sample-and-hold test. Since code written in Thrust is portable between CUDA and OpenMP, comparing the performance is simple [**58**]. Figure 4.5 shows that the GPU version performs significantly better than the parallel OpenMP CPU version running on eight Intel Xeon E5-2637 cores.

The total time to complete the two batch processing steps depends primarily on a few factors—the number of eligible flows, the window size, and the IPD threshold value. We obtained our best

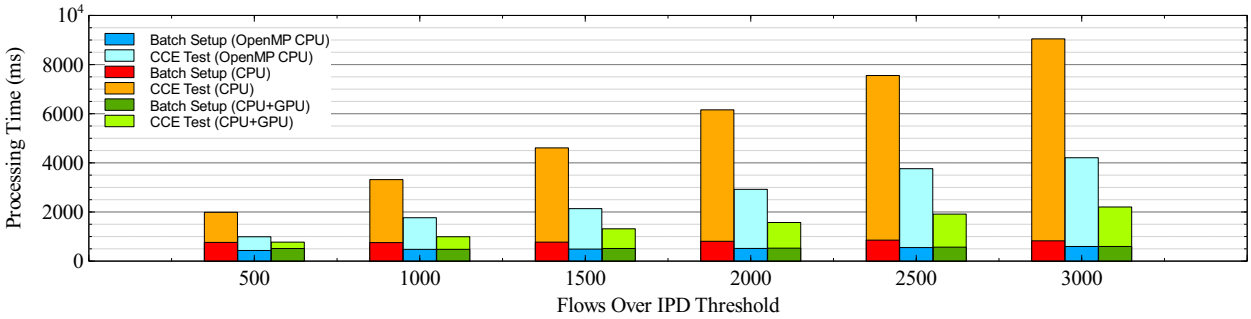


FIGURE 4.10. CPU Batch processing time vs. the numbers of flows tested.

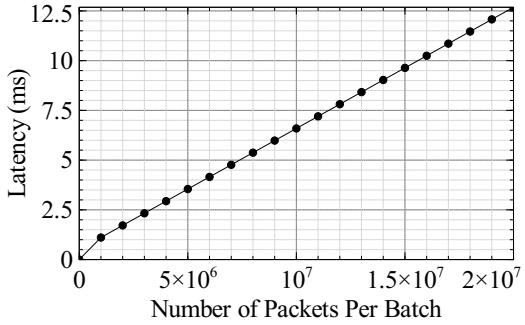


FIGURE 4.11. Batch Latency vs. Batch Threshold.

results by having Thrust perform the batch setup on the CPU and the CCE calculation on the GPU. If the batch contains the maximum number of flows possible (3000), the calculation will require slightly over 2 seconds to classify the flows (Figure 4.10). However, since a batch will almost never contain only eligible flows, packet loss is unlikely. The CCE calculation time per flow remains below 0.5 ms regardless of how many flows are being tested. Even including the batch setup time, this is significantly faster than the 16 ms per flow CCE calculation described in previous results [**13**].

4.6. Related Work

Covert Timing Channel Detection. Although many papers describe techniques for either limiting channel capacity or eliminating network covert timing channels completely [**53**, **54**], these techniques usually hurt legitimate traffic performance as well, making covert timing channel detection the more appealing choice [**12**]. Cabuk et al. [**46**] introduced two covert timing channel detection techniques—the regularity test and the ϵ-similarity test. Gianvecchio and Wang [**13**] provide background on a variety of covert timing channel detection techniques, and also introduce

entropy-based detection using measurements of first-order entropy and the corrected-conditional entropy to identify covert traffic. Many other measurements have been used for detection, such as mean-max ratio [**13**]. Commonly, detection techniques are created to counter a particular covert timing channel, but have limited effectiveness in detecting other channel types [**13**]. Examples include the ϵ-similarity test (effective for detecting IPCTC traffic), and measuring the data and acknowledgement packet timing intervals (effective for detecting the Cloak CTC) [**13**]. After new detection techniques are introduced, new covert timing channel types designed to counter those techniques tend to follow [**13**, **50**].

GPU Packet Processing. There have been many papers showing GPU packet processing as an effective means of scaling network packet processing applications using commodity hardware [**47**]. Given their higher memory bandwidth, GPUs have been shown to perform high data rate software packet processing more efficiently than CPUs alone [**48**]. PacketShader is a GPU-based processing framework that can perform OpenFlow flow matching and ipv4/ipv6 packet forwarding at multi-10Gbps rates [**59**]. Similarly, Snap is a framework built on top of Click, a modular software router. Snap performs SDN forwarding and other processing tasks at 30 Gbps with minimum-sized packets, and can reach 40 Gbps with packet sizes starting at 128-bytes [**68**]. In addition to packet forwarding, GPU-based processing has been used to quickly perform pattern matching for intrusion detection systems, such as Kargus and Gnort [**69**, **70**]. GPU-based processing has also been applied to software-defined networks (SDNs). For example, GSwitch is a recent system that performs packet classification using the GPU to improve packet searches. Their Bloom search algorithm outperforms a CPU-based equivalent by a factor of 12, processing 64-byte packets at 10 Gbps [**71**].

Depending on the application, much of the benefits of GPU processing come from the advantages of writing algorithms in languages such as OpenCL or CUDA, which has inherent advantages such as vectorization and hiding memory latency [**72**]. By optimizing memory latency in CPU software packet processing applications, the authors significantly closed the gap between CPU and GPU performance [**72**]. Therefore, GPU-based packet processing could be more worthwhile for tasks that benefit more from vectorization than reducing memory latency, since programming for CPU-based vectorization is difficult [**72**], although compilers such as Intel's ispc provide extensions for

single-instruction multiple data (SIMD) programming [**72**]. For this reason, CTC detection could be expected to benefit significantly from GPU processing, since certain detection tests—including the CCE entropy test—require processing a large number of flows, each with corresponding vectors of inter-packet delays (IPDs).

4.7. Conclusion and Future Work

Our results confirm that covert timing channel detection can be performed efficiently in real-time by calculating the corrected conditional entropy scores for network flows on a GPU. Our tool manages to detect CTCs more accurately and at higher data rates when compared to previous results [**12**], even while running purely on the CPU. As predicted by earlier results [**13**], the CCE test is a reliable classifier for model-based covert channels, identifying suspicious flows with a low false positive rate. By converting the conditional entropy tree structures into arrays, batches of packet data can be converted into a format that is easily parallelized and translated into GPU code. This technique could potentially be applied to other entropy measurements, such as those used to evaluate financial transfers or connectivity in the brain [**57**]. Our results demonstrate that, in addition to tasks such as firewall rule lookups and packet forwarding [**48**], GPU packet processing can be applied for improving statistical analysis of individual network flows at the packet level.

There are multiple ways in which our tool could be expanded upon and improved. One obvious way would be to include additional CTC detection tests that can identify channel types that evade CCE detection, such as including Welch's t-test for Jitterbug channels [**73**]. Although our detection tool could handle traffic at 10 Gbps line rates with average sized packets, 40 Gbps line rates are becoming more common. Although using raw CUDA kernels might increase the maximum throughput, using CUDA Thrust provides multiple advantages, notably the ease of coding that allows new detection tests to be added quickly, as well as portability between CPUs and GPUs. One potential direction to expand this work would be to integrate it with the Bro intrusion detection system by writing policy scripts that respond to reported CTCs by disrupting the flow's packet timing to reduce or eliminate the channel's capacity. Finally, although our implementation uses only a single GPU, the CCE computation should scale well for handling higher data rates. The more

memory available, the more batches can be created. On a system with multiple GPUs, multiple batches can be processed in parallel, allowing our tool to handle much higher rates of traffic.

CHAPTER 5

Denial of Service Detection on High-Throughput Research Networks

A broad range of techniques exists for detecting distributed denial of service (DDoS) attacks under different circumstances. The type of techniques used will depend on the network traffic and typically avoiding false positive alerts is a priority. In this paper, we consider the problem of detecting DDoS attacks in large-scale science networks. Science networks possess certain features, such as sudden large volume increases, which can trigger false positive alerts when applying common anomaly-based detection techniques. Therefore, we evaluate the effectiveness of detection techniques for detecting attacks within science network traffic, including common entropy and volume-based techniques. In our experiments, we evaluate the true and false positive rates of these techniques against known DDoS attack samples. In addition, we combine these attack samples with science flow traffic to determine whether or not the detection remains effective. By analyzing known attack samples and DTN traffic, we hope to learn more about the effectiveness of DDoS detection techniques on high performance science networks, including which techniques might be effective and potential challenges. In addition, we have created a modular tool for performing our DDoS detection tests, which we have made available in a public GitHub repository [1].

5.1. Introduction

Distributed denial of service attacks (DDoS) are a common form of attack which make the target unusable by depleting some resource or reducing availability [**74**]. The methods used for DDoS attack detection and mitigation depends heavily on the type of network being protected and where the detection takes place [**74**]. Recent work has considered the problem of defending high performance science networks from attacks [**30**], and therefore it is natural to consider the problem of DDoS detection in that environment. Science networks pose a number of issues. For example,

[1] `https://github.com/lbnl-cybersecurity/ddos-detection`

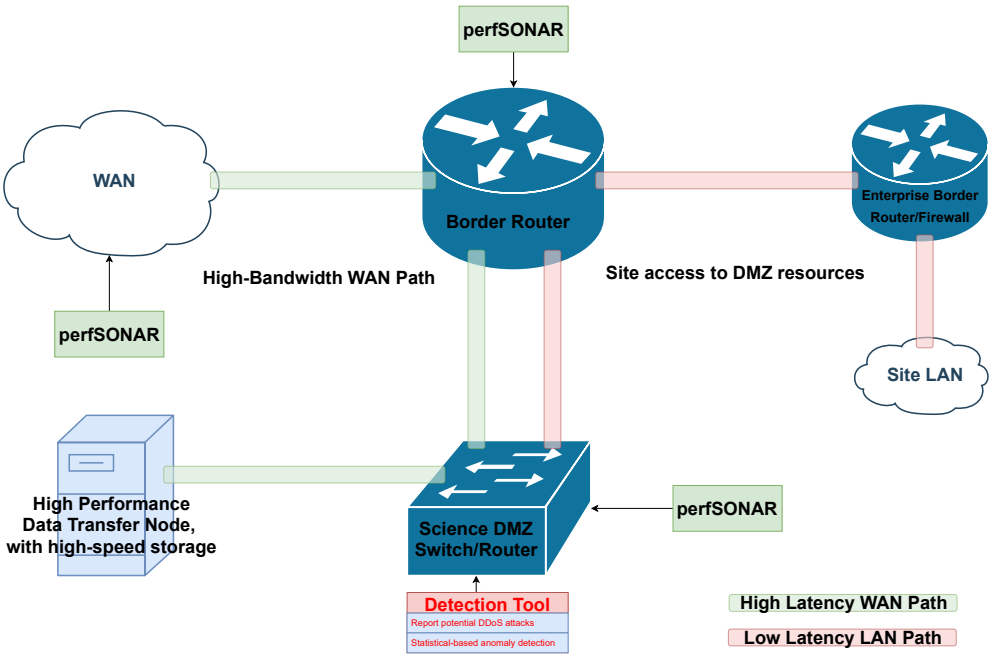


FIGURE 5.1. In this chapter, we examine various methods for detecting DDoS traffic in the Science DMZ environment. A modular tool for testing different DDoS detection techniques has been created.

the sudden appearance of large data transfers might be mistaken for a DDoS attack. Therefore, existing statistical anomaly-detection methods such as entropy [**75**] or wavelet analysis [**76**] might be ineffective in this environment. In addition, the high rates found in science networks, such as the 100G NERSC border router [**77**] presents another challenge for defending against attacks. Our goal in this project was primarily to lay the groundwork for DDoS detection and prevention in large-scale science flow environments by learning more about science traffic in general and how well traditional methods work. To achieve this, we performed DDoS detection tests on both known attack samples and science flow traffic, and noted other observed characteristics of science flows. The goal was to help answer which types of detection tests function in these environments without creating false positives. In our experiments, we consider two sources of science flows - the 10 Gbps ESnet and the NERSC Data Transfer Nodes (DTN). We developed detection tests which were capable of detecting attacks observed in real traffic samples obtained from IMPACT Cyber Trust [**78**]. Following that, we studied the effectiveness of these methods to determine how to reduce potential false positives.

TABLE 5.1. Summary of the datasets used in this study.

Dataset	Collected At	Start Date	Duration	Description
DS-1 [2] [**79**]	Merit border router in SFPOP	07/21/2015	24 hours	DNS-based reflection and amplification DDoS attack.
DS-2 [**80**]	Merit border router in SFPOP	11/24/2016	24 hours	CharGen-based reflection and amplification DDoS attack.
DS-3 [**81**]	Merit border router in Detroit	12/09/2015	2 hours	SSDP-based reflection and amplification DDoS attack.

The rest of the paper is organized as follows. Section 5.2 provides background on the attacks included in our experiments. Section 5.3 describe the DDoS detection algorithms we used. Section 5.4 describes our analysis evaluating the performance of these algorithms in the science environments. Section 5.5 provides a description of related work relevant to DDoS detection and protecting science flows. Finally, Section 5.6 gives our conclusion and a discussion of future work planned for this project.

5.2. Characterization of DDoS Attacks

The design of our detection framework is inspired by our analysis of real-world DDoS case studies. In this section, we analyze a number of real-world DDoS events that are of different attack types and are collected from different sources. Table 5.1 summarizes various DDoS events we have analyzed.

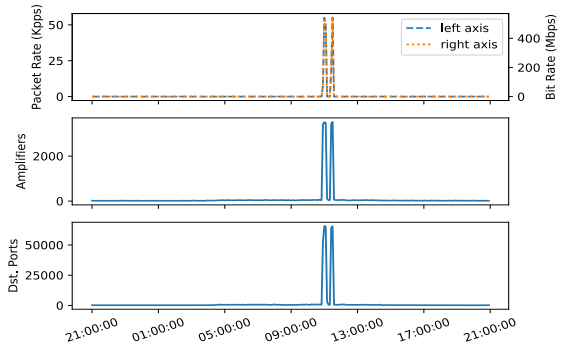
5.2.1. Reflection and Amplification DDoS Attack. In a typical reflection and amplification DDoS attack, the attacker sends a number of requests to a group of amplifiers that run services vulnerable to an amplification attack. The attacker intentionally specifies the victim's IP address as the source of the requests, causing the amplifiers to send their responses to the victim, even though the victim never asked for it. Due to the amplification factor, the size of responses (number of packets and bytes) are significantly larger than the requests, causing a DDoS attack at the victim side.

DNS, CharGen, SSDP, NTP are the top 4 services that are abused most frequently for reflection and amplification DDoS attacks [**82**]. Table 5.2 summarizes the typical UDP listening port for the top 4 services. This section illustrates how a clear attack pattern is detected by monitoring incoming

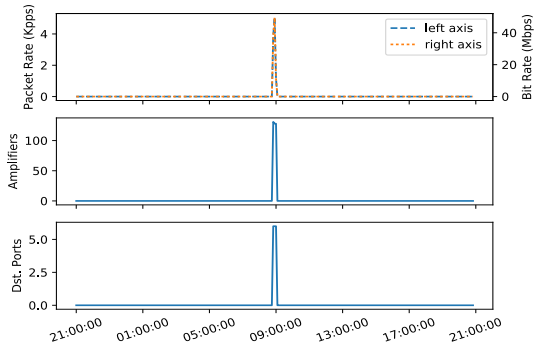
TABLE 5.2. Overview of the analyzed protocols.

Protocol	Port	Description
DNS	53	The Domain Name Service
CharGen	19	The Character Generator Protocol
SSDP	1900	The Simple Service Discovery Protocol

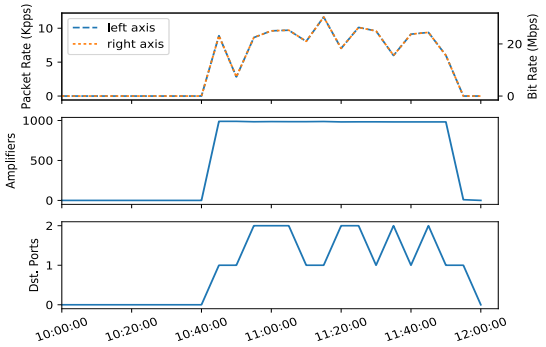
traffic from the corresponding source port. In the following we will characterize the attack case study per protocol from the victim's perspective:



(a) DNS attack



(b) CharGen attack



(c) SSDP attack

FIGURE 5.2. From left to right are the case studies of DNS/CharGen/SSDP-based reflection and amplification DDoS attacks. In each figure, we monitor the incoming DNS/CharGen/SSDP traffic to the victim and plot from top to bottom: the packet/bit rate, the number of amplifiers sending replies, and the number of destination ports being accessed over time. The results are calculated per 5-minute time window.

[2]Dataset 1-3 are collected by Merit Network, Inc., who operates Michigan's research and education network. In dataset 1-3, all IPs are anonymized by zeroing the last 11 bits.

5.2.1.1. *DNS.* The dataset DS-1 contains a DNS-based reflection and amplification DDoS attack. In figure 5.2, we monitor the following metrics over time: a) the DNS traffic volume towards the victim, b) the number of unique source addresses (amplifiers) found in the DNS traffic, and c) the number of the destination ports being accessed by the DNS traffic. For part a), we consider all packets that are sent to the victim, over UDP protocol using source port 53. We can clearly detect the attack from Figure 5.2a based on: a) dramatic increase in the DNS traffic volume received by the victim during the attack, b) dramatic increase in the number of source addresses that send DNS replies to the victim, and c) the fact that almost all 65535 ports at the victim side are accessed by the DNS traffic.

5.2.1.2. *CharGen.* The DS-2 dataset contains a CharGen-based reflection and amplification DDoS attack. From the victim's perspective, we monitor the following metrics over time a) the CharGen traffic volume, b) the number of source addresses (amplifiers) found in the CharGen traffic, and c) the number of destination ports being accessed in the CharGen traffic. To estimate CharGen traffic volume, we consider all packets sent to the victim over UDP protocol using source port 19. As shown in Figure 5.2b, the DDoS attack can be easily detected based on: a) dramatic increase in the CharGen traffic received by the victim, and b) dramatic increase in the number of source addresses (amplifiers) sending replies to the victim. However, the number of destination ports DDoSed is only 6, opposed to the 65535 ports exploited in the DNS DDoS attack.

5.2.1.3. *SSDP.* The dataset DS-3 contains a SSDP-based reflection and amplification DDoS attack. Similarly, the attack can be clearly detected from Figure 5.2c based on: a) dramatic increase in the SSDP traffic volume received by the victim, and b) dramatic increase in the number of source addresses that send SSDP replies to the victim. However, the number of destination ports being targeted during DDoS is only up to 2, in contrast to the 65535 ports exploited in the DNS DDoS attack.

5.2.2. SYN Flooding DDoS Attack. The dataset DS-5 contains a SYN-flooding DDoS attack. During the SYN-flooding, the victim sees an unusually high volume of SYN packets and an alarmingly high percentage of *syn-to-tcp packet ratio*. We define the metric *syn-to-tcp packet ratio* as the number of syn packets over the total number of tcp packets received by a destination address within a time interval. As shown in Figure 5.4, the SYN-flooding attack signature is self-evident.

During the attack, the victim receives huge amount of TCP-SYN packets from a number of source addresses. Note that DS-5 dataset is collected using Netflow [**83**] with 1:1000 sampling rate. The true scale of source addresses are expected to be at most 1000 times larger than what we observe after sampling. In this attack, all SYN packets are targeted at one destination port: port 80. We specifically calculate the syn-to-tcp packet ratio. During the SYN-flooding DDoS attack, the ratio is even higher than 50%.

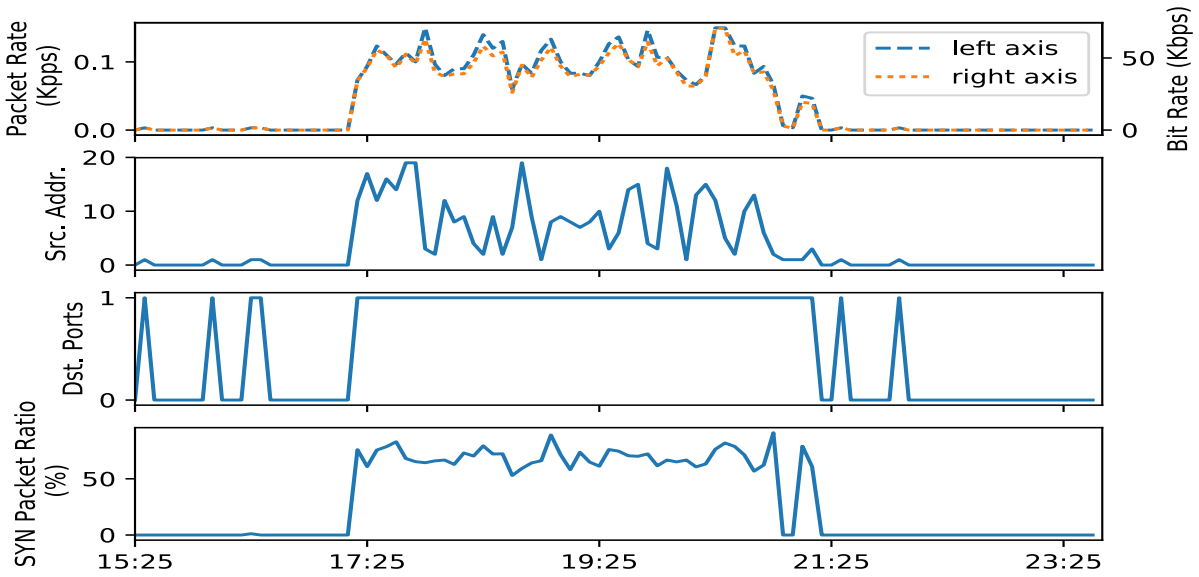


FIGURE 5.3. Standard Deviation

FIGURE 5.4. From top to bottom: the SYN packet rate/bit rate, the number of unique source addresses (botnet IPs or spoofed source IPs) found in the SYN traffic, the number of destination ports being accessed by the SYN traffic, and the syn-to-tcp packet ratio over time.

5.3. DDoS Detection Algorithms

In this section, we present two DDoS detection algorithms. We refer to the first one as modified adaptive change detection (MACD) algorithm. It is an easy yet efficient algorithm that detects DDoS incidents based on violations of a threshold that is adaptively updated with recent traffic measurements. Direct application of the adaptive change detection algorithm is subject to a high

false positive rate. In this section, we also elaborate on our proposed methods to reduce false positive rate. The second one is an entropy-based detection algorithm.

5.3.1. Modified Adaptive Change Detection Algorithm. Inspired by the analysis in Section 5.2, detection of different DDoS attack types requires monitoring different types of traffic. For instance, we focus on (incoming) DNS traffic to detect DNS-based amplification DDoS attacks. In the later context, we use the term "traffic of interest" to refer to the traffic that are monitored to detect a particular type of DDoS attack.

In this algorithm, we first profile the normal traffic of interest to get the baseline volume. The baseline volume is adaptively updated to account for traffic variations and trends. We test whether the current traffic measurement over a given time interval exceeds a particular threshold that is set based on the baseline number.

Let $x_{da,n}$ denote the measurement for destination da in the $n-th$ time interval, $\mu_{da,n-1}$ is the mean count estimated from measurements to n for destination da. The mean $\mu_{da,n}$ can be computed using an exponential weighted moving average (EWMA) of previous measurements [**84**]

$$\mu_{da,n} = \beta * \mu_{da,n-1} + (1-\beta) * x_{da,n} \tag{5.1}$$

where β is the EWMA factor.

Alarm Condition. An alarm is triggered in the $n-th$ time interval for destination da, if

$$x_{da,n} \geq \alpha * \mu_{da,n-1} \tag{5.2}$$

where α is the amplification factor.

The tuning parameters of the above algorithm are the amplification factor α that is used to set the alarming threshold, the EWMA factor β for updating average count, and the length of the time interval over which traffic measurements are taken. Direct application of the adaptive change detection algorithm would yield high false positive rate. We discuss various methods to reduce false alarms in the following context.

5.3.1.1. *Absolute Volume Threshold.* The absolute volume threshold provides a lower bound on DDoS attacks we detect in our algorithms. The vanilla adaptive change detection algorithm only

considers relative change, therefore has a high false positive rate for variations of small-size flows. We set this value according to the flow size distribution of background traffic. The absolute volume threshold helps eliminate false positives induced by small flows.

5.3.1.2. *Time Aggregate.* The vanilla adaptive change detection algorithm easily mistakes flash-crowds in the traffic as DDoS attacks. We observed that flash-crowds tends to be short peaks in time. An effective modification we deploy in our MACD algorithm is to hold on an alarm, until the number of consecutive violations reach an *time aggregate* threshold.

5.3.2. Shannon Entropy. Entropy measures the amount of randomness in a set of data, and is commonly used for DDoS detection [**75**]. The formula for Shannon entropy is as follows:

$$E = -\sum_{i=0}^{n} [p_i * \log_2 p_i] \tag{5.3}$$

In this formula, p_i is the probability of some event occurring, such as the probability that a particular destination address occurs in some interval. For our implementation, we actively measure the amount of randomness for incoming traffic flows every five minutes. For many DDoS attacks, certain features will repeat more often than normal, giving an abnormal entropy scores [**75**]. Depending on the chosen features, the entropy scores will vary, as well as their effectiveness for detecting attacks. Due to its consistency, we found it best to use the destination IP entropy, with each flow containing that IP address weighted by its average packet size. In this case, E is calculated every five minutes, each p_i corresponds to the probability of a particular destination IP appearing during that interval. This setting tended to give the most consistent baseline scores, along with the most noticeable dips in entropy when attacks occur.

5.4. Performance Evaluation

We compare the performance of MACD algorithm against (a) direct application of Adaptive Change Detection (ACD) algorithm and (b) Absolute Threshold algorithm. In absolute threshold algorithm, we consider an IP address is under DDoS attack as long as the measurement in a time interval is larger than the absolute volume threshold. We will show later that (a) or (b) would yield large false positive rates. Direct application of ACD algorithm identifies small flows as DDoS

attacks. On the other hand, the absolute threshold algorithm simply treats all large flows as DDoS attacks. We also consider the performance of entropy-based detection for our attack samples as a comparison.

5.4.1. Experiment Design. We use two sets of real-world network traffic as the background traffic. The first trace is taken from ESnet. This dataset contains sampled netflow data collected continuously from 01/31/2016 to 02/13/2016. The sampling rate is 1 in 1000 [**85**]. The second trace is taken from the NERSC DTN nodes.

In order to see how our tests performed with different kinds of science flows, we merge various real-world DDoS attacks (described in section 5.2) with the science network background traffic to test the effectiveness of our detection algorithms. The attack flows are extracted from the attack samples and then tested alongside another source of background traffic. In some cases the attack samples were taken from networks where the traffic was orders of magnitude higher. In other cases, such as the ESnet sample, the recorded flows were sampled at a 1 to 1000 rate. Therefore, we also include a method of amplifying one of the samples. For example, if the attack sample comes from a higher volume network, we can choose to multiply the DTN background traffic by a constant factor to mimic a proportionally large number of background flows.

In the first investigation, we merge the UDP-based reflection and amplification DDoS attacks with ESnet background traffic. Following that, we merge attacks with the NERSC DTN traffic. We also analyzed features such as the packet and flow volume, comparing the science flow traffic to our Merit samples.

We use two metrics to evaluate the performance of detection algorithms. The detection rate measures the percentage of true attacks that are detected by the algorithm. The false positive rate (FPR) is the number of false alarms that do not correspond to an actual attack, over the total number of alarms reported.

5.4.2. DDoS Detection in ESnet. In this experiment, we use traffic traces collected from ESnet as the background traffic, and merge various real-world DDoS attacks into the background traffic. We run the algorithms over the merged traffic trace to evaluate their detection performance for the SSDP and CharGen samples.

TABLE 5.3. Comparison of Detection Results for SSDP DDoS Attacks

Algo.	Detection Rate	FPR
MACD	100%	2.10%
Direct ACD	100%	73.9%
Absolute Threshold	100%	8.50%

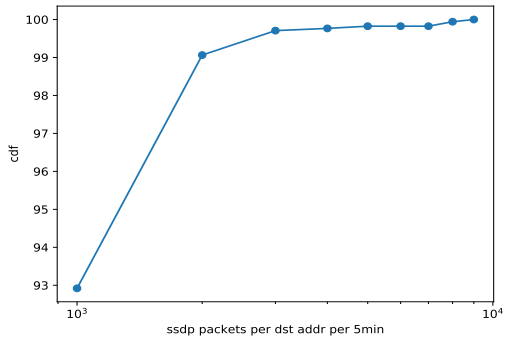
5.4.2.1. *SSDP.* We use the real-world DDoS attack found in [**81**] and merge it into the ESnet background traces. Figure 5.5 (right) shows a snapshot of the attack and (left) plots the distribution of SSDP packets in the background traffic. We can see that the volume of the attack is three orders of magnitude larger than the traffic volume in background. We scaled down the attack volume and merge into the background multiple instances of the attack with different scale factors, at randomly picked time slots.

We use the first-day's traffic data as the training data to build the average volume baseline for each destination address and run our detection algorithm afterwards. In Figure 5.7, from top to bottom, we show the traffic trace with merged attacks, the original background traffic trace, the merged attacks only, and the detection results of our proposed algorithm. Table 5.3 compares the detection performance of our proposed algorithm with direct adaptive change detection algorithm and absolute threshold algorithm discussed above. We also run an entropy-based detection algorithm on the merged trace and the result is plotted at the bottom of figure 5.7. We note that the entropy score is constantly jumping from 0 to 1, making it not helpful in detection. This might arise from the fact that the number of flows in SSDP traffic is limited.

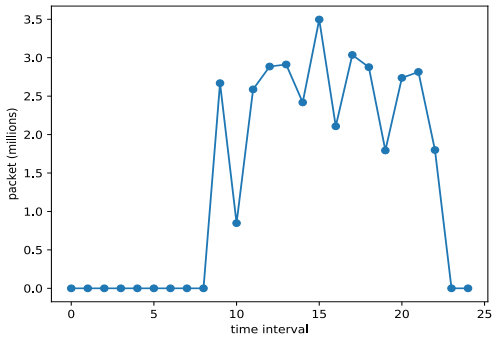
5.4.2.2. *CharGen.* Similarly, we merge the real-world CharGen-based DDoS attack of [**80**] into the ESnet background trace. Figure 5.8 (right) plots a snapshot of the attack and (left) shows the distribution of CharGen packets per destination address per 5min bin in the background traffic.

We repeat the same experiments as in 5.4.2.1 (SSDP), and the results are shown in table 5.4 and Figure 5.10. We can see that our proposed algorithm achieves 100% detection rate as well as the least false positive rate, compared to the other three algorithms.

5.4.3. DDoS Detection in DTN. In this section we will cover the results of our analysis of DDoS detection in the NERSC DTN. Although we do not have any samples for attacks occurring in the DTN traffic, we consider how the characteristics of our DTN traffic might affect established



(a) Standard Deviation



(b) Entropy

FIGURE 5.5. Left: Distribution of SSDP packets in the ESnet background traffic. SSDP packets are aggregatively counted for each destination address within a 5-min interval. Right: A snapshot of the real-world SSDP attack. SSDP packets are counted within every 5-min time interval.

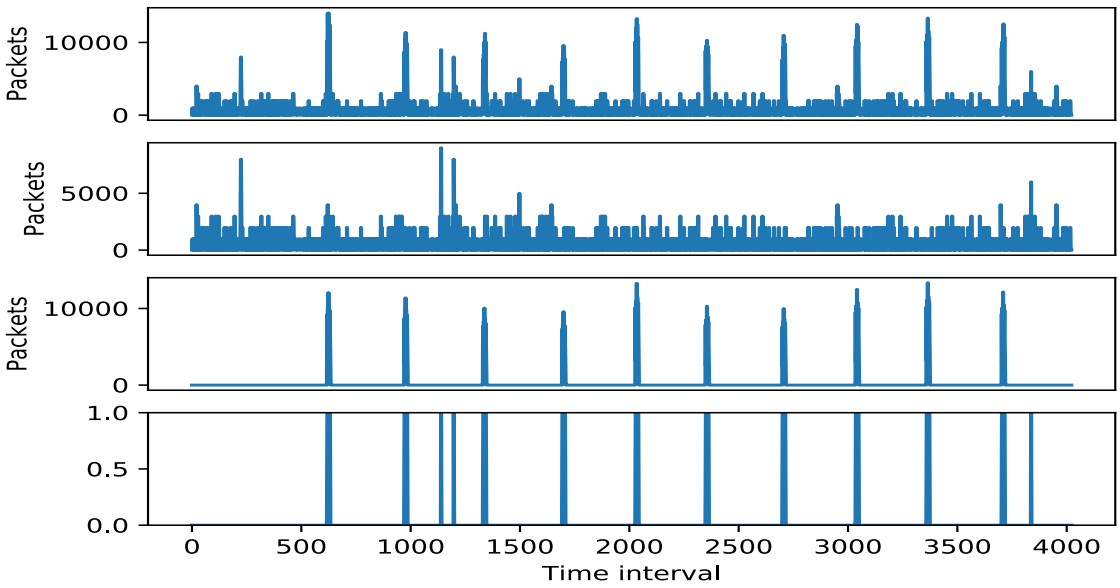
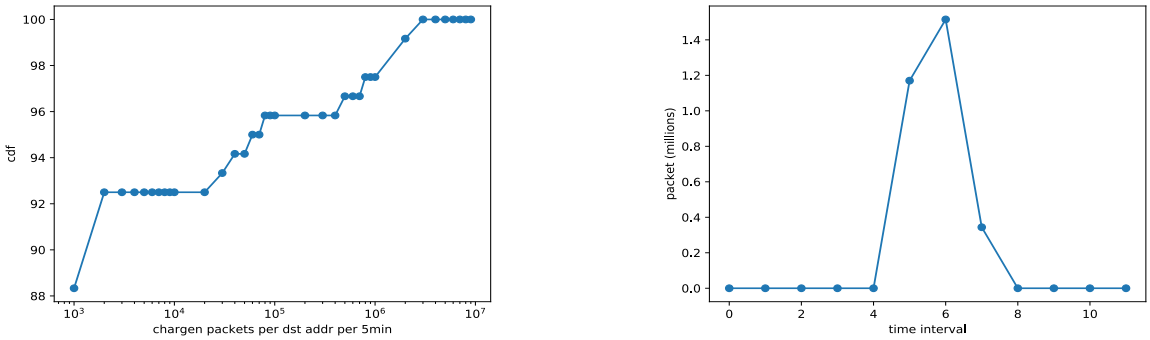


FIGURE 5.6. Standard Deviation

FIGURE 5.7. From top to bottom, the time-varying traffic trace with merged attacks, the original background traffic trace, the merged attacks only, the time intervals when an alert is reported. The bottom figure shows the time-series of entropy score for traffic trace with merged attacks.

TABLE 5.4. Comparison of Detection Results for CharGen DDoS Attacks

Algo.	Detection Rate	FPR
MACD	100%	23.1%
Direct ACD	100%	53.8%
Absolute Threshold	100%	23.1%



(a) Standard Deviation

(b) Entropy

FIGURE 5.8. Left: Distribution of CharGen packets in the ESnet background traffic. CharGen packets are aggregatively counted for each destination address within a 5-min interval. Right: A snapshot of the real-world CharGen attack. SSDP packets are counted within every 5-min time interval.

detection methods. Merging DTN traffic with our attacks did not seem to affect the results significantly, possibly because during periods of low activity the attack would dominate the traffic. This indicates a potential flaw in the usefulness of our merging technique for the DTN sample. Rather than considering merged data, we begin by comparing the entropy results for the DTN flows with normal border router traffic seen in the DS-2 attack sample. The characteristics of both our research network samples make certain detection tests less effective, meaning alternative methods of calculating entropy should be used.

5.4.3.1. *DTN Entropy Detection.* Table 5.5 demonstrates that entropy was able to reliably detect all our attack samples except the SSDP attack. This includes an observed ESnet SYN flood attack we examined. In order to detect attacks, a consistent entropy score range is required for non-attack traffic to establish a baseline. This proved problematic when measuring the DTN science flows, because no consistent baseline exists when using the total packet size entropy (figure 5.11a). This means a normal entropy range cannot be established for detecting anomalies. However,

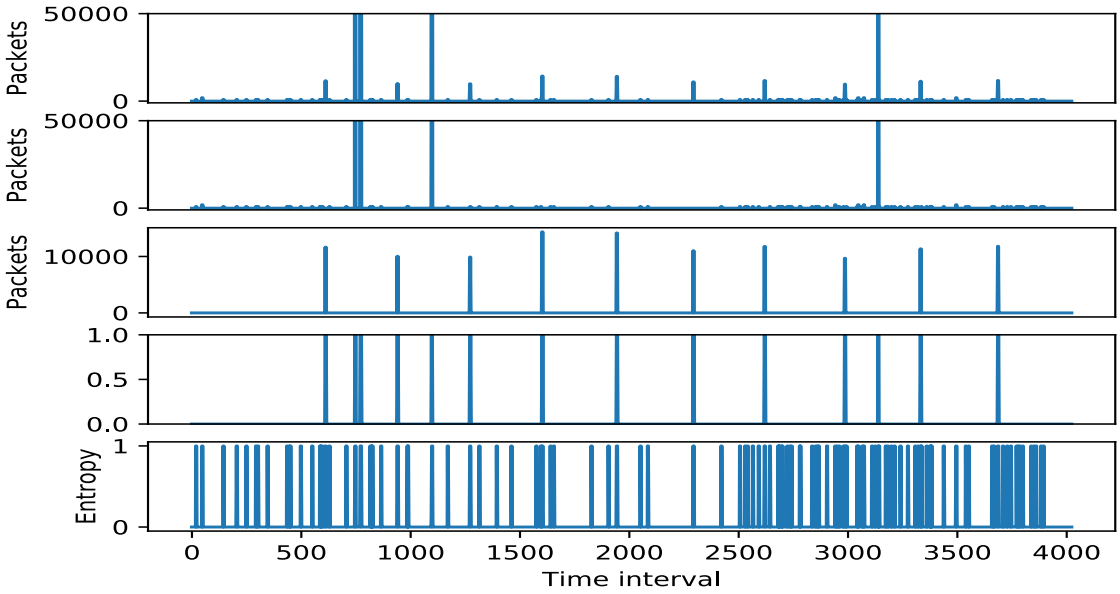


FIGURE 5.9. Standard Deviation

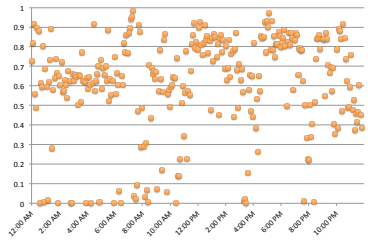
FIGURE 5.10. From top to bottom, the time-varying traffic trace with merged attacks, the original background traffic trace, the merged attacks only, the time intervals when an alert is reported. The bottom figure shows the time-series of entropy score for traffic trace with merged attacks.

using the average packet size entropy (figure 5.11b), gives more consistent results. Therefore, the consistency of the entropy scores varies heavily depending on the background traffic and the features chosen. Figure 5.10 demonstrates the difference between the CharGen (DS-2) scores and the DTN entropy scores (total packet size and average packet size entropy). In the character generation attack sample (CharGen), we see an obvious drop from baseline entropy around 12:00 pm, matching where the attack begins according to the provided metadata. Therefore, entropy is effective for detecting CharGen's attack without significant false positives. As seen in Figure 5.11b, the average packet size gives more consistent entropy scores, letting us establish thresholds for anomaly detection. However, the DTN results are still inconsistent. This inconsistency is due to the DTN traffic being less regular compared to other forms of traffic, such as the border routers used in our IMPACT Cyber Trust samples. Figure 5.12 compares the DTN volume on July 31st to the dns_ampl attack sample volume. For long stretches, the DTN has low levels of activity, with traffic on the order of

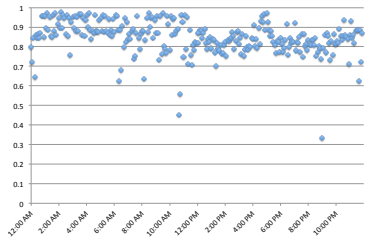
TABLE 5.5. Comparison of Entropy Results for DDoS Attacks.

Sample	Detection Rate	FPR
DNS (DS-1)	100%	3.1%
Chargen (DS-2)	100%	0%
SSDP (DS-3)	0%	0%
ESnet SYN Flood (DS-5)	100%	4.2%

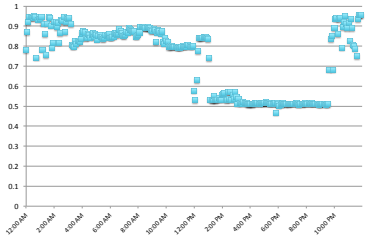
100s of flows. However, due to intermittent large data transfers, we also see sudden increases in traffic volume. These sudden traffic increases skew the entropy results, as well as other anomaly-based detection methods such as wavelet detection [**76**]. This is similar to the problems caused by flash events [**86**], where large amounts of legitimate traffic resembles a DDoS attack. Although entropy is normally effective for detecting attacks, for research networks we must use carefully select features which give consistent results under normal conditions in spite of the large variations in volume.



(a) NERSC DTN (total packet size)

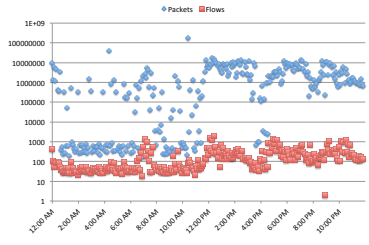


(b) NERSC DTN (average packet size)

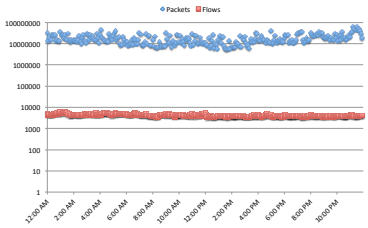


(c) chargen (DS-2)

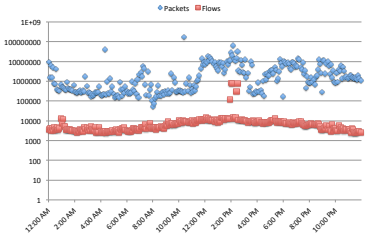
FIGURE 5.11. Entropy Score Comparison. Using the average packet size per flow to calculate the entropy gives more consistent entropy scores, making anomalies easier to detect. However, it is still less consistent than the DS-2 Merit border router traffic.



(a) NERSC DTN Volume (July 31st)



(b) ESnet Sample Volume



(c) DS-1 Volume

FIGURE 5.12. Packet and Flow Volume. The NERSC DTN traffic fluctuates more than the ESnet traffic or the Merit border router traffic.

5.5. Related Work

There is a rich literature in the study of DDoS detection. Koay et al. [**75**] discussed the commonly used entropy-based detection, describing effective features for calculating the score. Wang [**87**] proposed to use the ratio of SYN packets to FIN/RST packets to detect SYN floods. However the attackers can easily avoid detection by sending FIN/RST packets along with SYN packets. Siris [**84**] proposed an adaptive threshold algorithm to detect SYN flooding. But the algorithm is only tested against synthetically generated attacks. The LADS system [**88**] focuses on DDoS detection on the coarse level of egress interface using SNMP feeds from routers, as contrary to our fine-grained per-IP level DDoS detection. The MACD algorithm proposed in our work bears a similar idea to the above proposals. But we generalize its application to different types of DDoS attacks, and evaluate its effectiveness against real-world DDoS attacks with science background traffic. We adapt the algorithm to accommodate the new environment to reduce false positive rate.

Although no other papers have currently addressed the specific problem of identifying DDoS attacks on large scale science networks, a number of recent papers have described the challenges of

protecting such networks. A recent article [**89**] summarizes important differences between protecting high-performance computing (HPC) networks and protecting standard environments. A key observation is that, in addition to higher speeds, HPC networks usually have more specific purposes, leading to more predictable behavior. The paper describes some currently used solutions for monitoring high-rate science flows, such selectively shunting flows to analyze with the Bro network intrusion detection system [**90**]. The broader issue of analyzing the content of large high-speed flows found in science networks is also getting attention. Mandal et al. [**91**] discusses challenges with monitoring and understanding the normal behavior of science networks.

5.6. Conclusion and Future Work

Our experiments demonstrate that some special considerations are necessary when monitoring science flows. The activity in science flow environments like the NERSC DTN can vary largely, and any detection method must account for this. The primary complications we discovered were long periods of low activity in the DTN, combined with the sudden appearances of large, sometimes long lasting flows. Although these factors make anomaly-based detection more difficult, we believe that common methods such as Shannon entropy are potentially effective if the chosen feature can produce consistent results in legitimate traffic. In addition, our MACD algorithm showed high detection rates even with the ESnet background traffic. There are many directions in which our current work can be expanded. So far, we have looked primarily at layer 3 and 4 attacks, such as TCP-SYN floods and UDP reflection and amplification attacks (DNS Amplification, Character Generation, SSDP). Therefore, we would like to continue looking at a wider variety of DDoS attack types to further test the effectiveness of our detection methods. We also intend to experiment further with other DDoS detection methods, such as wavelet analysis [**76**]. Although real attack data is valuable, it can be difficult to base broad conclusions on individual attacks. Therefore, we would also test detection using synthetic attacks where we have more control over different parameters.

5.7. Sharing Statement

The data in this project from the IMPACT repository [**78**] is available to qualified researchers at DHS-approved institutions, as indicated in the IMPACT Terms of Use. The data in this project

from ESnet and NERSC is available on a case-by-case basis to qualified researchers, and in accordance with those institutions privacy policies, application[3] to ESnet's Security Team and NERSC's Security Team,[4] respectively. The code for this project, including our DDoS detection tools are also available in a public GitHub repository [5].

[3] https://www.es.net/about/governance/data-privacy-policy/
[4] http://www.nersc.gov/users/accounts/user-accounts/computer-security/
[5] https://github.com/lbnl-cybersecurity/ddos-detection

CHAPTER 6

Unusual Protocol Monitoring with Zeek

The Zeek network security monitor provides a variety of ways to monitor and log protocol usage. However, there are holes in the visibility of Zeek's standard logging. By default, Zeek does not report extensive data on all lower level protocols. For example, many transport layer protocols outside of the common TCP, UDP, and ICMP are not visible to Zeek. No events are created when they appear, making it difficult to handle them in scripts. In this work, we extend Zeek by creating modified plugins and various scripts to assist in analyzing lower level protocol usage. These extensions allow Zeek to monitor and log more protocol usage information, along with logging anomalies which could indicate malicious behavior. In addition to the Zeek scripts, we provide Python scripts for more in-depth analysis of the log data. In this paper, we explain our the extensions, how our scripts work, and demonstrate how this protocol usage data can assist with intrusion detection through detecting some common network anomalies such as DDoS attacks.

6.1. Introduction

Protocol specifications and usage statistics has been a common method for network anomaly detection [**92**]. However, the widely used Zeek network intrusion detection system (IDS) provides limited support for monitoring and analyzing lower level protocols. By default, Zeek supports roughly 50 different protocols, and new protocol analyzers are required for adding new protocols. [**93**] In some cases, the unrecognized protocols won't be logged by Zeek at all, not even as unknown protocols in the weird.log, which logs unexpected behavior. [**94**] This work introduces modified plugins for IP and Ethernet layer protocols, which call Zeek events for each new packet. These events contain the information such as the packet timestamp and the protocol seen. Zeek scripts handle these events, logging the protocol data according to the user's chosen settings. Some Python scripts are also provided for analyzing these log files by providing additional analysis such as

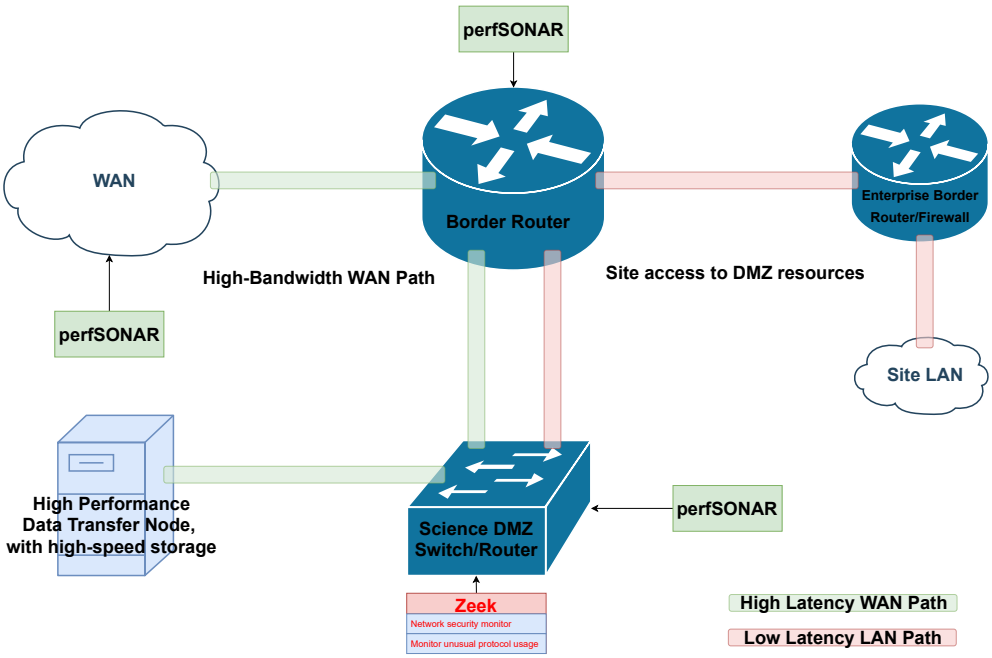


FIGURE 6.1. The Zeek network security monitor is a common tool used for protecting Science DMZs. In this chapter, we expand on Zeek's protocol monitoring capabilities.

DBSCAN clustering. If the distribution changes significantly in different measurement periods, the clusters could reveal anomalous behavior on the network. By extending Zeek for this form of anomaly detection should improve detection by making possible to create scripts based on previously applied detection methods using protocol data. The Zeek scripts can be applied for both real-time monitoring and analyzing trace files. This should be particularly useful for monitoring Science DMZs and other networks where the traffic remains relatively predictable over time.

The tools we've created for monitoring protocol usage can log a variety of results, including when new protocols appear for the first time, when protocols have appeared past a preset threshold count, and statistics about the protocol usage distribution. This data can be used to detect a variety of anomalies, and in this work we demonstrate its utility in detecting some common network anomalies. In particular, we show how these scripts can be used together to detect some common varieties of DDoS attacks captured in a recent test set [**95**]. Our initial results suggest that protocol usage is effective for detecting DDoS attacks and other threats. The scripts can also log new top protocols, the protocol distribution statistics such as standard deviation and Shannon entropy, and

when common L7 protocols appear over unexpected L4 protocols. For example, a protocol such as HTTP or SSH might be riding on top of a protocol other than the expected TCP. Logging this occurrence could be useful in detecting covert traffic that evades standard detection. The code is made available in a public GitHub repository [**96**], and the various options and scripts will be explained in a later section.

The following are some key contributions:

- We present modified Zeek plugins for creating new L3 and L4 protocol events.
- We provide Zeek scripts using these events for monitoring and logging anomalies in protocol usage, along with Python scripts for additional analysis of the log files.
- We demonstrate the potential value of these detection tools through experimental results, detecting attacks in the CIC-DDoS2019 dataset [**95**].

In section II, we will provide some background information on Zeek and examine related work using protocol data for network anomaly detection. In section III, we describe the plugins and detection scripts in detail, explaining the different log files and options available. Section IV will provide some results for examples of how these tools could be used for detecting common network threats. Finally, Section V discusses our conclusions and some plans for future work.

6.2. Background and Related Work

6.2.1. Zeek. Zeek (formerly Bro) [**14**] is an open-source platform for network monitoring, analysis, and intrusion detection. The Zeek framework monitors network traffic (live or reading from trace files), using two primary components - an event engine which calls events based on network events, and policy scripts which handle these events. For example, an event is when a new TCP connection begins, which can then be handled in different ways by the policy scripts. Zeek provides a wide variety of events and scripts by default, logging connections and information on application layer protocols, such as DNS requests [**97**]. However, unlike some other network intrusion detection systems (NIDS), Zeek is not limited to predefined analysis functions. Instead, Zeek's goal is to provide a framework for network analysis through this extendable system of events and scripts. Through the Zeek scripting language, users can define any sort of analysis to be performed in response to the events, including calling external processes. Therefore, it can be

configured for any variety of intrusion detection methods, such as signature-based or anomaly-based detection. Figure 6.2 illustrates how Zeek's architecture functions.

In addition to this flexibility, Zeek is lightweight and suitable for a variety of systems. Zeek clusters can be created to handle higher rates of traffic. The platform is commonly deployed for monitoring Science DMZs and other high speed networks [**97**], effectively monitoring 10 Gbps and higher networks in real-time. Furthermore, certain functionality such as calling events for L7 protocols regardless of the underlying protocols allows for the potential detection of obfuscation attempts similar to those discussed in our previous insider attack detection work [**98**].

6.2.2. Related Work. As a basic feature of network traffic, monitoring protocol usage is a critical feature for many network intrusion detection systems, and is used to detect a variety of anomalies [**99**]. Different intrusion detection systems will vary in the number of protocols supported and the extent to which they can be analyzed. A recent paper by Holkovič et al [**93**] proposes a framework for automating network security analysis, and discusses the challenge of needing to create new protocol dissectors whenever a previously unsupported protocol must be analyzed. In their paper, Zeek is discussed as being close to their proposed framework, but Zeek's limited range of protocols supported by default is mentioned as a weakness, along with missing support for some protocol fields. Grashöfer et al. [**100**] describes another weakness, that current methods of identifying protocols in IDS can be evaded. Based on these papers, we can see some of the motivation to expand Zeek by providing more methods of easily analyzing protocols.

One recent tool which helps alleviate this issue is Spicy [**101**], a parser generator. With its Zeek plugin, Spicy simplifies the process of creating new network protocol parsers, allowing the user to define their own grammar determining which events are called in response to an input. For example, a recent paper by Chromik et al. [**102**] makes use of Spicy to implement an extensible parser for IEC-104 protocol traffic. Similarly, Rice et al. [**103**] uses Spicy to parse the BACnet protocol. Though Spicy satisfies some of our goals, we instead opted modify the default Zeek plugins by providing events with new information about all seen protocols at lower levels, allowing scripts to more readily handle previously unrecognized protocols.

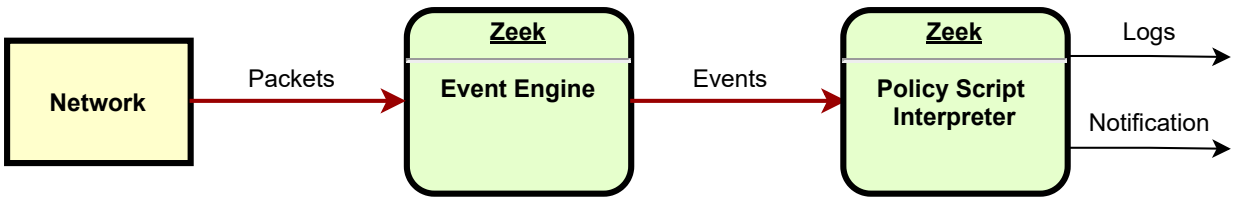


FIGURE 6.2. Zeek architecture [**97**].

TABLE 6.1. File descriptions

Filename	Description
IP.cc	L4 protocol analyzer plugin, creates events for new IP packets.
Ethernet.cc	L3 protocol analyzer plugin, creates events for new Ethernet packets.
unusual_protocols.zeek	Script for monitoring L4 protocols and logging unusual events.
ethernet_protocols.zeek	Script for monitoring L3 protocols and logging unusual events.
l7_unusual.zeek	This script is unusual_protocols.zeek, with an added logging for L7 protocols appearing over uncommon L4 protocols.
thresholds.file	CSV file for the Zeek scripts, containing a list of protocols and their threshold values. If a protocol is observed in more packets than the threshold value, the protocol is logged.
nfdump_parser.py	Parses nfdump files, printing the protocols and the number of packets and flows containing each protocol.
parse_router_results.py	Takes nfdump files from the router directory, and outputs protocol statistics.
protocol_clustering.py	Uses DBSCAN clustering to cluster a given protocol's timestamps and totals.

6.3. Zeek Plugins and Scripts

In this section, we will describe the updated plugins and the different Zeek scripts used. In addition, we will explain the various events and functions. All of the described files are available in a public GitHub repository. [**96**] Table I lists the various files and gives a brief overview for each. In the following subsections we will discuss each in more detail.

6.3.1. Zeek Plugins. Previously, many protocols outside of the application layer did not have Zeek analyzers available, making it more difficult to create plugins for lower level protocols. However, a recent addition to Zeek [**104**] made pluggable lower layer analyzers available. This recent release simplified the process of getting creating and modifying plugins for new L3 and L4 protocols. For example, by modifying the provided Ethernet and IP plugins to create events for each new packet, we can gain access to a wide range of different protocol data for analysis with our scripts. Whenever a new Ethernet or IP packet is seen, we call an event which contains the packet timestamp, the source and destination addresses, and the protocol number. This event is

TABLE 6.2. Zeek log files

Log name	Source	Fields
unusual_protocols.log	unusual_protocols.zeek, l7_unusual.zeek	timestamp, source IP, destination IP, protocol number, protocol name, threshold, cycle count
unusual_eth_protocols.log	ethernet_protocols.zeek	timestamp, source IP, destination IP, protocol number, protocol name
protocol_totals.log	unusual_protocols.zeek, l7_unusual.zeek	timestamp, message type, protocol number, protocol name, protocol total, standard deviation, entropy
new_protocols.log	unusual_protocols.zeek, l7_unusual.zeek	timestamp, protocol number, protocol name
over_unusual.log	l7_unusual.zeek	timestamp, source IP, destination IP, protocol number, protocol name

TABLE 6.3. Zeek script options

Option Name	True	False
log_all	Logs every protocol that appears past the shared threshold value value (log_all_thresh)	Only logs the protocols listed in thresholds.file.
log_once	Only log a protocol the first time it passes the threshold value.	Log every time the protocol appears past the threshold.
log_distribution	Write the protocol distribution along with statistics periodically, with log_distr_pkts as the cycle length.	Don't log the distribution or distribution statistics.
reset	Reset the value such as packet counts each cycle.	Don't reset the values for new cycles.
check_esp	Check for the protocol encapsulated in ESP protocol packets and log it.	Report ESP protocol only, not the encapsulated protocol.

then handled by the various Zeek scripts, which monitor and analyze protocol usage through this information.

6.3.2. Zeek Scripts. The two main Zeek scripts we use for monitoring and logging the lower layer protocol events are **unusual_protocols.zeek** and **ethernet_protocols.zeek**. These scripts handle the events created by **IP.cc** and **Ethernet.cc** respectively. Therefore, **unusual_protocols.zeek** monitors new IP packets and **ethernet_protocols.zeek** monitors new Ethernet packets. Optionally, multiple log files can be created with these scripts. A third Zeek script called **l7_protocols.zeek** is essentially **unusual_protocols.zeek** with an additional logging function. This script will monitor SSH, HTTP, and FTP connections to log when they are found running over a lower level protocol besides the typical TCP, which can indicate anomalous behavior. As most of our work focused on L4 protocols, **unusual_protocols.zeek** has more options available and creates additional log files for more in-depth analysis. These Zeek scripts have a variety of different settings controlling the analysis and how the log files are created. The different log files created are listed in Table II, and the various options are explained in Table III.

6.3.3. Python Scripts. In addition to the Zeek scripts, we've provided Python scripts for further log files analysis. The scripts **nfdump_parser.py** and **parse_router_results.py** work together to breakdown protocol statistics for a directory containing Zeek logs for different routers.

First, **nfdump_parser.py** takes an nfdump file and parses the protocol data, printing the protocols, and the number and percentage of flows and packets appearing with that protocol. Meanwhile, **nfdump_parser.py** reads in the text file created by a bash script (**router_protocols_script.sh**), and outputs the protocol percentages per router, unique and unnamed protocols per router, and unique protocols (appearing only on a single router). The script **protocol_clusters.py** takes protocol-totals.log (which logs the packet distribution periodically) and clusters the packet counts for that protocol along with the timestamps. This is one of the scripts we use for anomaly detection, because a large change in the clusters will indicate that the distribution has changed significantly. For traffic that maintains a relatively consistent distribution under normal conditions, this could indicate a serious anomaly. In the next section, we will show how this clustering can be used along with the Zeek log files to detect potential DDoS attacks.

6.4. Anomaly Detection

In order to show some potential use cases for our protocol monitoring, we demonstrate how the scripts can be used to detect some common network anomalies. For our anomaly detection experiments, we used the recent CIC-DDoS2019 dataset [**95**]. This dataset contains trace files for two separate days of traffic, during which a variety of common DDoS attacks occur. The dataset emphasized creating realistic background traffic along with the attacks, which helps in verifying the detection results. Our goal was to test whether or not we could detect some of the DDoS attacks contained in these trace files using our scripts. We consider how these attack can be detected through the information logged in **protocol_totals.log**, both by using the measured statistics (Shannon entropy and standard deviation) or by clustering the protocol totals gathered each cycle. In the next sections, we present and discuss the anomaly detection results.

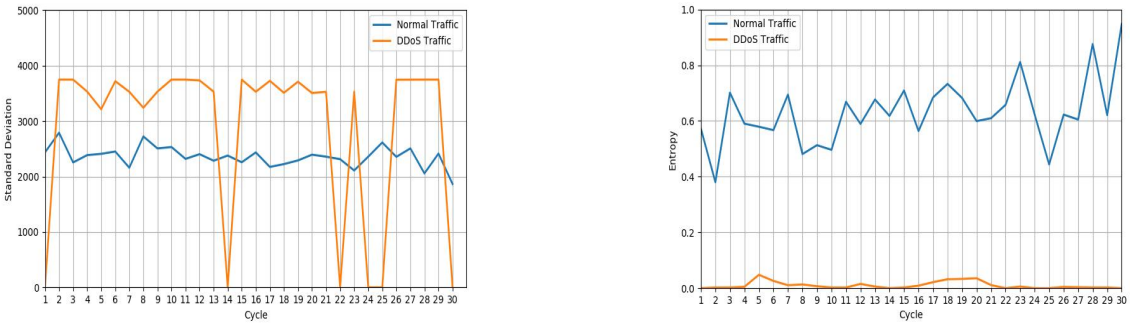
6.4.1. Statistical anomaly detection. First, we consider how the statistics in **protocol_totals.log** can be used to detect the attacks. We use two separate statistics, which are recorded each time a packet cycle ends and the protocol totals distribution is logged. The two values we log are the standard deviation of the protocol totals, and the Shannon entropy of the protocol totals. The Shannon entropy measures the randomness of a variable, with a higher entropy score indicating the

variable is less predictable. It is given by the following formula:

$$H = -\sum_{i=1}^{n} P(x_i) \log P(x_i) \tag{6.1}$$

with $P(x_i)$ referring to the probability of selecting the value x_i. Assuming we know the network will have relatively consistent types of traffic, the protocol totals per cycle should also maintain predictable entropy and standard deviation values. For this experiment, each cycle is set to 10,000 packets (**log_distr_pkts** = 10,000), so new statistics are logged after every 10,000 packets.

Figure 6.3 compares the standard deviation and entropy values over time for both normal traffic and DDoS traffic in the CIC-DDoS2019 dataset. As we can see, the standard deviation increases during the DDoS attack, and the entropy drops significantly (indicating less randomness in the protocol totals).



(a) Standard Deviation

(b) Entropy

FIGURE 6.3. The standard deviation and entropy values for 30 cycles of protocol distributions at different times during day 2 of the CIC-DDoS2019 dataset. The normal traffic is taken after 17:15, when all the attacks have finished. The DDoS traffic here is the SYN flood occurring at 13:29 - 13:34. In this case, each cycle lasted 7,500 packets. During normal traffic conditions, the standard deviation of the protocol usage distribution remains fairly stable. However, during an attack, the standard deviation will either increase due to one protocol becoming significantly more common, or drop to zero because only one protocol is seen during that entire cycle. Since the protocol distribution is more predictable during the attack, we see that the entropy becomes very low.

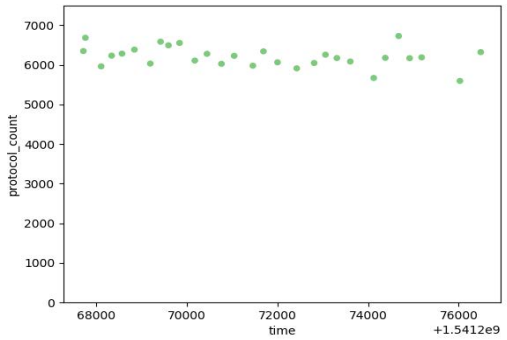
6.4.2. Clustering-based anomaly detection. Next, we show how clustering can be applied to identify the DDoS attacks. The timestamps and protocol totals per cycle are clustered using DBSCAN clustering, a density-based clustering method designed for handling noisy datasets [**29**].

DBSCAN requires two parameters − **minPts**, the minimum number of points constituting a cluster, and ϵ, the maximum distance between points within a cluster. The choice for these parameters varies depending on the dataset, though generally **minPts** should be at least one more than the number of variables being clustered. For our usecase, ϵ will depend primarily on the chosen **log_distr_pkts** value, which determines how many packets are logged in each cycle. We previously applied DBSCAN for detecting threats using system performance data. More details on the DBSCAN algorithm can be found in Chapter 2.

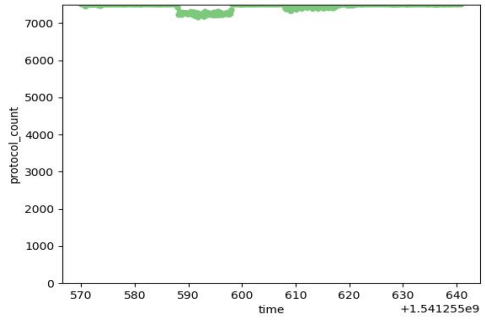
Figure 6.4 compares examples of the clusters formed during a SYN flood with those formed during typical traffic. As we can see, the clusters are noticeably different. In particular, it seems that there is a large difference when clustering the entropy per cycle. Therefore, it is likely that new entropy clusters will appear during the DDoS attacks.

6.4.3. Further anomaly detection. In addition to these examples of detecting DDoS attacks, these scripts could be helpful in detecting a variety of other threats. For example, when Science DMZs or other networks with relatively predictable behavior, we can expect to mostly know which protocols will appear. When a previously unseen protocol is logged in **new_protocols.log**, it could indicate an anomaly. Along with the distribution statistics, there are additional logging options in **packet_totals.log** can be used to detect unusual changes in the distribution. For example, we create a log entry whenever the top seen L4 protocol changes. Furthermore, we monitor for some common application layer protocols riding on top of uncommon lower layer protocols. Specifically, we log HTTP, FTP, and SSH packets which are not using TCP. Running common applications like SSH over uncommon protocols could be an indication of an attacker attempting to obfuscate their behavior.

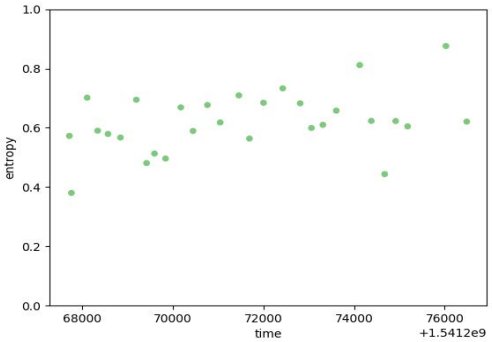
6.4.4. Detecting Low and Slow DDoS Attacks. "Low and slow" DDoS attacks [**105**] are performed at low rates and slowly exhaust resources over time. Unlike high rate DDoS attacks, low and slow attacks are not likely to significantly alter the protocol usage statistics. Therefore, it is unlikely that the detection method discussed earlier will be effective for these types of attacks. However, there are effective options for detecting these types of attacks which can be implemented alongside the other forms of detection. We considered how to modify our Zeek tools for detecting



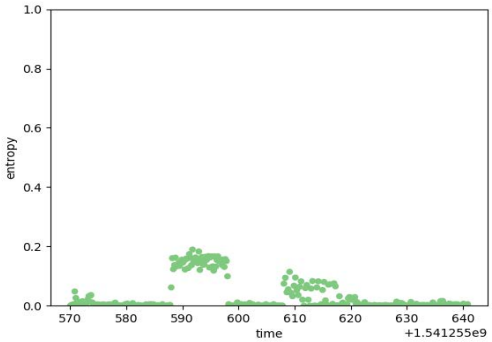
(a) Ordinary TCP cluster



(b) DDoS TCP cluster



(c) Ordinary entropy cluster



(d) DDoS entropy cluster

FIGURE 6.4. Clustering the TCP protocol counts and entropy per 7,500 packet cycle at different times during day 2 of the CIC-DDoS2019 dataset. The normal traffic is taken after 17:15, when all the attacks have finished. The DDoS traffic here is the SYN flood occurring at 13:29 - 13:34. The clusters formed during normal traffic conditions will be separate from the DDoS clusters. (Need to update these graphs and get more results for clustering, but this gives an idea. Currently the CIC-DDoS2019 dataset is broken up in a way that makes creating the graphs difficult.)

Slowloris [**106**], a common "low and slow" attack, testing with a pcap file containing a Slowloris attack [**107**]. The Slowloris attack functions by sending incomplete HTTP GET requests [**106**], keeping HTTP connections open until their are no resources to handle legitimate HTTP connections. One method we considered for detecting Slowloris attacks was monitoring for HTTP packets appearing over uncommon protocols. Slowloris attacks can be performed using HTTP over protocols besides TCP, such as UDP using QUIC [**105**]. Therefore, a sudden increase in HTTP packets over non-TCP protocols could indicate suspicious activity. Using our **l7_unusual.zeek**

script, we can log whenever common application layer protocols such as HTTP appear over unusual protocols. Our sample Slowloris attack contained a large number of HTTP over HOPTOPT (Hop-by-Hop IPv6) packets. Although seeing protocols like this is not unusual by itself, they can be used exploited in certain servers which are not configured for handling them [**108**]. In addition to looking at the protocol data, there are some known generalized detection methods for Slowloris and similar HTTP stalling attacks. One such method using Zeek [**109**] involves monitoring HTTP request and reply events for requests going for significant periods of time without receiving a reply. If too many suspicious connections are flagged, then there is a possible Slowloris attack occurring. This detection method has been incorporated into our Zeek scripts in addition to the protocol usage-based DDoS detection. However, there is no one-size-fits-all solution, and in some cases it would be necessary to include more specialized detection methods for specific attacks. In many cases, this involves monitoring the resource targeted by the particular low and slow attack [**110**]. New Zeek scripts for detecting specific DDoS attacks of this variety could be added when desired.

6.5. Conclusions and Future Work

In this work we have expanded on Zeek's protocol monitoring capabilities, updating the lower layer protocol plugins and adding new scripts for logging protocol data. Taken together, these scripts should be a useful tool for analyzing traffic and performing anomaly detection. Using the recent CIC-DDoS2019 dataset [**95**]. Our tests showed that the logged protocol usage distributions and related statistics can be used to identify DDoS attacks in networks where the protocol distributions remain relatively consistent under normal conditions, such as Science DMZs. In addition, other network traffic anomalies are likely to be found through these logs. By adding to Zeek's protocol monitoring capabilities, we also expand the range of possible anomalies we can detect. Future work should provide a more comprehensive look at which anomalies can be detected through more in-depth monitoring of lower layer protocols. Furthermore, the scripts themselves can be expanded upon, adding new features and logging more complex statistics. The tools created are available in a public GitHub repository [1]

CHAPTER 7

Conclusion

In this final chapter, we will summarize our work and contributions. In addition, we will provide some thoughts on future work, including new potential projects in addition to expanding our completed research. The emphasis of our work has been improving anomaly detection for protecting the Science DMZ. Anomaly detection for high speed networks such as Science DMZ is a challenging problem, in part due to the difficulty of obtaining realistic data for experiments. However, anomaly-based detection has been shown to be more effective in cases where the detection is focused on known threats in environments with a narrow range of normal behavior. The relatively predictable behavior compared to general purpose environments makes anomaly-based detection well-suited for Science DMZ networks. Our work considered various ways of detecting potential threats to the Science DMZ, both insider threats such as data tampering and outsider threats such as denial of service attacks. We developed and tested various methods of detection suited for protecting the Science DMZ environment, particularly the data transfer nodes (DTNs). We considered potential threats such as covert timing channel, and studied how they can be detected effectively in high-speed networks. In addition, we expanded on the Zeek network intrusion detection system to assist with this goal, creating new plugins and scripts for monitoring protocol usage more effectively. In the following sections, we will summarize the completed projects, their important contributions, and suggestions for future work.

7.1. Anomaly Detection Using System Performance Data

In this work [**111**], we considered how host performance metrics could be applied towards anomaly detection in the data transfer nodes (DTNs). More specifically, we demonstrated how interrupts per second and context switches per second could be used to discover TCP-SYN floods in a DTN environment. In addition, we showed how system performance metrics could be used to help detect insider threats such as data exfiltration and data tampering. We created a detection

method using continuous re-clustering of these performance metrics using DBSCAN [**10**]. In our experiments, we found that by routinely clustering context switches and interrupts per second together, we could detect all the generated TCP-SYN floods within 10 seconds. Previous work has shown similar results in other environments [**17**] [**16**], indicating that certain attack varieties create distinct and reliably detectable patterns in system performance. The primary contributions of this work were the creation of the DBSCAN-based real-time anomaly detection method, and demonstrating that certain system performance metrics such as interrupts and context switches can be effective for anomaly detection in the Science DMZ environment.

TCP-SYN floods are just one simple form of attack which can be detected, and it is likely that similar methods can be applied to detect a wide variety of attacks. Therefore, this work should be expanded by attempting to detect a wider variety of threats to DTNs. Building on the completed work, experiments should investigate a variety of common network threats, and take a deeper look at which types of outsider attacks are the most realistic threats to the DTNs. In addition to the TCP-SYN floods we've already considered, this could include detecting port scans, brute force/dictionary attacks, and so forth. Although simulating these attacks is possible as in our experiments, ideally real data samples could be used. Preferably, the detection could then be tested on an active DTN, measuring the effectiveness of detecting them through the various logged network and host performance metrics. In our previous experiments, TCP-SYN floods were detected using our streaming DBSCAN clustering method. This method should be capable of detecting a variety of other attacks by distinguishing between normal and abnormal clusters in the logged data, and future experiments should apply this method to a wider variety of attacks. For some common external threats to the network, it is likely the Science DMZ already has some detection or prevention mechanisms in-place. Therefore, another aspect of future work should be performing an in-depth comparison between this detection method and common network intrusion detection methods. Finally, it would be useful if the clustering-based detection described in our work were implemented in a standard network intrusion detection systems such as Zeek [**14**], allowing it to be more easily tested and expanded upon.

7.2. Insider Attack Detection Using System Performance Data

Following the external attacks, we considered how this system might be applied towards the detecting insider threats [**98**]. The main goal was to determine network and system performance on the host given normal conditions, and then determine whether or not certain insider attacks cause recognizable patterns of performance activity diverging from ordinary behavior on the DTN, through which the attacks can be detected. In the Science DMZ environment, it is likely that We considered varieties of insider threats targeting important data − data sabotage, exfiltration, and obfuscation. To accomplish this, created a tool which obfuscates the packets through PDF files to run an ssh session on the target, and through that session we tampers with or exfiltrates the targeted data. The primary contribution of this work was demonstrating the value of system performance metrics for detecting data tampering, as well as detecting the usage of obfuscation methods which an attacker could exploit such as our PDF obfuscation tool, which could allow the attacker to exfiltrate data while appearing to perform legitimate behavior. For detection, we used our real-time detection method, which continually re-clusters the performance metrics using DBSCAN. Through this approach, we could reliably detect both data tampering and obfuscated SSH sessions. Given the relatively predictable range of behavior on the Science DMZ, we can more effectively detect whether or not a user (authorized or not) is performing legitimate actions. Even if an attacker attempts to conceal their malicious behavior, changes in the expected system behavior could be observed.

Considering our results, it appears that a wide variety of insider attacks could be detected through watching for anomalies in system performance data. Future work should begin by expanding on this detection by considering additional performance metrics. In addition, this type of detection might be combined with other common methods of insider attack detection, such as monitoring system calls. One possibility for improving detection would be to create a more complete model of the expected system behavior. For a system like a DTN, it might be feasible to gather performance metrics for the majority of possible common operations and traffic loads, such as during varying levels of traffic or while performance monitoring applications are running. Currently, our clustering-based anomaly detection looks for major changes in the performance metrics, which we assumed will occur only in the case of likely illegitimate behavior. However, it's possible that

some unusual, but legitimate, situations could produce similar changes in performance. Therefore, it would be beneficial to complete an in-depth analysis of system behavior on a real DTN, considering both during different transfer loads (large number of small files vs. small number of large files), and whether or not performance monitoring is occurring during those transfers. It would also be necessary to consider what types of legitimate user activity on the DTNs is generally accepted, though this is likely to vary between different Science DMZs, as some will allow more access than others [**1**]. Future work expanding on this project should take these factors into consideration.

7.3. Covert Timing Channel Detection

Network covert timing channels (CTCs) exploiting the inter-packet delays in network packets represent another potential means through which an insider could exfiltrate data. Although their capacity is often low [**112**], these channels are likely to be effective in environments such as a Science DMZ where long-lasting flows are commonplace. In our work [**12**], we considered how such channels could be detected in real-time at high data rates. Primarily, we focused on methods of performing the correct-conditional entropy test (CCE) in real-time. Prior work demonstrated the effectiveness of this test for detecting a variety of CTCs, however it was prohibitively time-consuming to perform. By performing a novel tree transformation to calculate the CCE score more efficiently, we were able to compute the CCE scores efficiently enough to perform the detection test in real-time. As a result, we could handle traffic rates close to 10 Gbps using both a GPU-based implementation and an OpenMP multicore CPU-based implementation. In addition to the improved CCE algorithm, this work considered some advantages and disadvantages of using different parallel architectures for CTC detection. Initially, we attempted CTC detection using the Tilera TilePro64 [**113**] MPPA architecture, but found issues with the flexibility compared to implementing detection on multicore CPUs or GPUs using CUDA Thrust [**58**]. Overall, the GPU implementation performed the best for testing the batches of packets.

7.4. Unusual Protocol Monitoring with Zeek

In addition to the anomaly detection itself, we wanted to provide tools for intrusion detection that could be shared and expanded upon. In this work, we created a set of plugins and scripts to improve the Zeek [**14**] network security monitor's ability to monitor L3 and L4 protocol usage.

Although Zeek provides some protocol monitoring by default, it has limited visibility. A large number of protocols are either disregarded or reported as "unknown protocols." Therefore, the primary contribution of this work was to provide these tools to improve unusual protocol monitoring with Zeek. The plugins and script have been provided in a publicly available GitHub repository. In addition, we provided extra Python scripts for analysis such as clustering the data contained in the new protocol log files. A variety of options are provided, such as logging new top protocols, logging the distribution of protocols periodically, and protocols appearing for the first time. To demonstrate a possible use case for these tools, we demonstrate the effectiveness of detecting DDoS attacks using new logs. We found that clustering the statistics recorded in the protocol distribution logs was effective for detecting DDoS attacks, assuming the protocol distributions remained relatively consistent. The Science DMZ is more predictable in terms of protocols seen than general purpose networks, making it well-suited for this form of detection.

There are a variety of ways in which this work could be expanded. First, the code we provide could be updated with new features. Currently the tools primarily log L3 and L4 protocol usage statistics, as these were the protocols least visible to Zeek by default, but the plugins and scripts could be expanded to log L7 statistics as well. In addition to adding more features, future work should try to apply these tools for detecting a wider variety of attacks. Although we've shown that the statistics gathered can be useful for detecting DDoS attacks, future experiments should try detecting a wider variety of attacks. This could include other varieties of DDoS attacks such as Crossfire [**114**], or even insider attacks and obfuscation attempts. There is already an option available for logging case where application layer protocols are observed over atypical L4 protocols (for example, SSH over UDP). This feature could possibly be applied to detect hidden SSH sessions as seen with our PDF obfuscation tool. Overall, the tools provided in this work should help improve detection with Zeek and serve as a starting point for future anomaly-detection using protocol usage statistics.

7.5. Conclusion

Throughout our research, we have found effective methods of anomaly detection for the Science DMZ which could reliably detecting common threats. In general, consistent anomaly detection is

challenging to perform due to a wide variety of possible threats and the difficulty of interpreting the anomalies [**8**]. However, the relatively limited behavior seen in the Science DMZ networks makes practical anomaly detection more feasible. In our work, we've demonstrated that we can reliably detect common threats such as denial of service attacks through real-time anomaly-based detection. We've also demonstrated that similar methods can be used to detect insider threats such as data tampering or data exfiltration through obfuscation or covert channels. The projects discussed in this dissertation should help demonstrate the potential for anomaly-based detection in this environment, and serve as a starting point for future experiments.

Publications

(1) R. K. Gegan, "Unusual protocol monitoring with zeek," (In preparation), 2020

(2) R. K. Gegan, R. Archibald, M. K. Farrens, and D. Ghosal, "Performance analysis of real-time covert timing channel detection using a parallel system," in *International Conference on Network and System Security*, pp. 519–530, Springer, 2015

(3) R. K. Gegan, V. Ahuja, J. D. Owens, and D. Ghosal, "Real-time gpu-based timing channel detection using entropy," in *2016 IEEE Conference on Communications and Network Security (CNS)*, pp. 296–305, IEEE, 2016

(4) R. Gegan, C. Mao, D. Ghosal, M. Bishop, and S. Peisert, "Anomaly detection for science dmzs using system performance data," in *2020 International Conference on Computing, Networking and Communications (ICNC)*, pp. 492–496, IEEE, 2020

(5) R. K. Gegan, R. Archibald, M. K. Farrens, and D. Ghosal, "Performance analysis of real-time covert timing channel detection using a parallel system," in *International Conference on Network and System Security*, pp. 519–530, Springer, 2015

Bibliography

[1] E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The science dmz: A network design pattern for data-intensive science," *Scientific Programming*, vol. 22, no. 2, pp. 173–185, 2014.

[2] S. Peisert, E. Dart, W. Barnett, E. Balas, J. Cuff, R. L. Grossman, A. Berman, A. Shankar, and B. Tierney, "The medical science dmz: a network design pattern for data-intensive medical science," *Journal of the American Medical Informatics Association*, vol. 25, no. 3, pp. 267–274, 2018.

[3] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridFTP framework and server," in *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, p. 54, IEEE Computer Society, 2005.

[4] PerfSONAR, "PerfSONAR," 2019.

[5] Y. Qin, A. Simonet, P. E. Davis, A. Nouri, Z. Wang, M. Parashar, and I. Rodero, "Towards a smart, internet-scale cache service for data intensive scientific applications," in *Proceedings of the 10th Workshop on Scientific Cloud Computing*, pp. 11–18, 2019.

[6] J. Crichigno, E. Bou-Harb, and N. Ghani, "A comprehensive tutorial on Science DMZ," *IEEE Communications Surveys & Tutorials*, 2018.

[7] S. Peisert, W. Barnett, E. Dart, J. Cuff, R. L. Grossman, E. Balas, A. Berman, A. Shankar, and B. Tierney, "The medical Science DMZ," *Journal of the American Medical Informatics Association*, vol. 23, no. 6, pp. 1199–1201, 2016.

[8] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection," in *2010 IEEE symposium on security and privacy*, pp. 305–316, IEEE, 2010.

[9] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[10] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, p. 19, 2017.

[11] A. Porta, G. Baselli, D. Liberati, N. Montano, C. Cogliati, T. Gnecchi-Ruscone, A. Malliani, and S. Cerutti, "Measuring regularity by means of a corrected conditional entropy in sympathetic outflow," *Biological Cybernetics*, vol. 78, no. 1, pp. 71–78, 1998.

[12] R. K. Gegan, R. Archibald, M. K. Farrens, and D. Ghosal, "Performance analysis of real-time covert timing channel detection using a parallel system," in *Network and System Security*, pp. 519–530, Springer, 2015.

[13] S. Gianvecchio and H. Wang, "An entropy-based approach to detecting covert timing channels," *IEEE Transactions on Dependable and Secure Computing*, vol. 8, no. 6, pp. 785–797, 2011.

[14] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[15] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks.," in *Lisa*, vol. 99, pp. 229–238, 1999.

[16] A. Visconti, N. Fusi, and H. Tahayori, "Intrusion detection via artificial immune system: A performance-based approach," in *IFIP International Conference on Biologically Inspired Collaborative Computing*, pp. 125–135, Springer, 2008.

[17] A. Aqil, A. O. Atya, T. Jaeger, S. V. Krishnamurthy, K. Levitt, P. D. McDaniel, J. Rowe, and A. Swami, "Detection of stealthy TCP-based DoS attacks," in *MILCOM 2015-2015 IEEE Military Communications Conference*, pp. 348–353, IEEE, 2015.

[18] E. Lawrence Berkeley National Lab, "DTN tuning," 2019.

[19] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise.," in *Kdd*, vol. 96, pp. 226–231, 1996.

[20] N. S. Foundation, "Campus cyberinfrastructure (cc*) program solicitation," 2019.

[21] A. Hanemann, J. W. Boote, E. L. Boyd, J. Durand, L. Kudarimoti, R. Łapacz, D. M. Swany, S. Trocha, and J. Zurawski, "Perfsonar: A service oriented architecture for multi-domain network monitoring," in *International conference on service-oriented computing*, pp. 241–254, Springer, 2005.

[22] T. Park, Y. Kim, J. Park, H. Suh, B. Hong, and S. Shin, "Qose: Quality of security a network security framework with distributed nfv," in *2016 IEEE International Conference on Communications (ICC)*, pp. 1–6, IEEE, 2016.

[23] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[24] V. Chandola, V. Mithal, and V. Kumar, "Comparative evaluation of anomaly detection techniques for sequence data," in *2008 Eighth IEEE international conference on data mining*, pp. 743–748, IEEE, 2008.

[25] V. Chandola, A. Banerjee, and V. Kumar, "Anomaly detection: A survey," *ACM computing surveys (CSUR)*, vol. 41, no. 3, p. 15, 2009.

[26] ESnet, "ESnet," 2019.

[27] A. Giannakou, D. Gunter, and S. Peisert, "Flowzilla: A methodology for detecting data transfer anomalies in research networks," in *2018 IEEE/ACM Innovating the Network for Data-Intensive Science (INDIS)*, pp. 1–9, IEEE, 2018.

[28] K. Tools, "hping3. ICMP or SYN flooding tool," 2014.

[29] M. Hahsler, M. Piekenbrock, and D. Doran, "dbscan: Fast density-based clustering with r," *Journal of Statistical Software*, vol. 25, pp. 409–416.

[30] V. Nagendra, V. Yegneswaran, and P. Porras, "Securing ultra-high-bandwidth science dmz networks with coordinated situational awareness," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pp. 22–28, ACM, 2017.

[31] P. J. Hawrylak, G. Louthan, J. Hale, and M. Papa, "Practical cyber-security solutions for the science dmz," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pp. 1–6, 2019.

[32] W. Hong, J. Moon, W. Seok, and J. Chung, "Enhancing data transfer performance utilizing a dtn between cloud service providers," *Symmetry*, vol. 10, no. 4, p. 110, 2018.

[33] D. M. Cappelli, A. P. Moore, and R. F. Trzeciak, *The CERT guide to insider threats: how to prevent, detect, and respond to information technology crimes (Theft, Sabotage, Fraud)*. Addison-Wesley, 2012.

[34] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: A survey," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 2, pp. 1397–1417, 2018.

[35] I. Homoliak, F. Toffalini, J. Guarnizo, Y. Elovici, and M. Ochoa, "Insight into insiders and it: A survey of insider threat taxonomies, analysis, modeling, and countermeasures," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–40, 2019.

[36] N. Nguyen, P. Reiher, and G. H. Kuenning, "Detecting insider threats by monitoring system call activity," in *IEEE Systems, Man and Cybernetics SocietyInformation Assurance Workshop, 2003.*, pp. 45–52, IEEE, 2003.

[37] A. Awad, S. Kadry, G. Maddodi, S. Gill, and B. Lee, "Data leakage detection using system call provenance," in *2016 International Conference on Intelligent Networking and Collaborative Systems (INCoS)*, pp. 486–491, IEEE, 2016.

[38] S. Trivedi, L. Featherstun, N. DeMien, C. Gunlach, S. Narayan, J. Sharp, B. Werts, L. Wu, C. Ellis, L. Gorenstein, *et al.*, "Pulsar: Deploying network monitoring and intrusion detection for the science dmz," in *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning)*, pp. 1–8, 2019.

[39] J. Nikolai and Y. Wang, "A system for detecting malicious insider data theft in iaas cloud environments," in *2016 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, IEEE, 2016.

[40] A. Oppermann, F. G. Toro, F. Thiel, and J.-P. Seifert, "Anomaly detection approaches for secure cloud reference architectures in legal metrology.," in *CLOSER*, pp. 549–556, 2018.

[41] D. M. Cappelli, A. P. Moore, and E. D. Shaw, "A risk mitigation model: Lessons learned from actual insider sabotage," tech. rep., CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 2006.

[42] M. Bishop, D. Gollmann, J. Hunker, and C. W. Probst, "08302 abstracts collection–countering insider threats," in *Dagstuhl Seminar Proceedings*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2008.

[43] E. Mouw, "Linux kernel procfs guide," 2001.

[44] A. F. M. M. M. Meo, M. Munafo, and D. Rossi, "10-year experience of internet traffic monitoring with tstat," 2020.

[45] M. Seger, "Collectl," 2014.

[46] S. Cabuk, "Network covert channels: Design, analysis, detection, and elimination," 2006.

[47] K. Tsiamoura, "A survey of trends in fast packet processing," *Network*, vol. 41, 2014.

[48] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: a GPU-accelerated software router," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 195–206, 2011.

[49] J. Zheng, D. Zhang, Y. Li, and G. Li, "Accelerate packet classification using GPU: A case study on HiCuts," in *Computer Science and its Applications*, pp. 231–238, Springer, 2015.

[50] K. Kothari and M. Wright, "Mimic: An active covert channel that evades regularity-based detection," *Computer Networks*, vol. 57, no. 3, pp. 647–657, 2013.

[51] R. Strzodka, M. Doggett, and A. Kolb, "Scientific computation for simulations on programmable graphics hardware," *Simulation Modelling Practice and Theory*, vol. 13, no. 8, pp. 667–680, 2005.

[52] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. S. Deng, and S. Zhang, "IP routing processing with graphic processors," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 93–98, European Design and Automation Association, 2010.

[53] M. H. Kang, I. S. Moskowitz, and S. Chincheck, "The pump: A decade of covert fun," in *Computer Security Applications Conference, 21st Annual*, pp. 7–pp, IEEE, 2005.

[54] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of computer security*, vol. 1, no. 3–4, pp. 233–254, 1992.

[55] Y. Lu, M. Wang, B. Prabhakar, and F. Bonomi, "ElephantTrap: A low cost device for identifying large flows," in *15th Annual IEEE Symposium on High-Performance Interconnects*, HOTI 2007, pp. 99–108, IEEE, 2007.

[56] Y. Lin and G. Medioni, "Mutual information computation and maximization using GPU," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, CVPRW'08, pp. 1–6, IEEE, 2008.

[57] S. Shao, C. Guo, W. Luk, and S. Weston, "Accelerating transfer entropy computation," in *International Conference on Field-Programmable Technology*, pp. 60–67, IEEE, 2014.

[58] L.-t. Lo, C. Sewell, and J. P. Ahrens, "PISTON: A portable cross-platform framework for data-parallel visualization operators," in *EGPGV*, pp. 11–20, 2012.

[59] M. Mukerjee, D. Naylor, and B. Vavala, "Packet processing on the GPU,"

[60] "Ntop." `http://www.ntop.org`.

[61] "Impressive packet processing performance enables greater workload consolidation." Intel Solution Brief, 2013. Accessed: Whitepaper.

[62] L. Rizzo, "Netmap: a novel framework for fast packet I/O," in *21st USENIX Security Symposium (USENIX Security 12)*, pp. 101–112, 2012.

[63] S. Gallenmuller, P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle, "Comparison of frameworks for high-performance packet IO," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 29–38, IEEE, 2015.

[64] I. Matta and L. Guo, "Differentiated predictive fair service for TCP flows," in *Network Protocols, 2000. Proceedings. 2000 International Conference on*, pp. 49–58, IEEE, 2000.

[65] K. Psounis, A. Ghosh, B. Prabhakar, and G. Wang, "SIFT: A simple algorithm for tracking elephant flows, and taking advantage of power laws," in *43rd Allerton Conference on Communication, Control and Computing*, Citeseer, 2005.

[66] "Nvidia." `http://www.nvidia.com/object/tesla-workstations.html`. Accessed: 2015-03-28.

[67] "Caida data.." `http://www.caida.org/data/overview/`.

[68] W. Sun and R. Ricci, "Fast and flexible: parallel packet processing with GPUs and click," in *Proceedings of the Ninth ACM/IEEE symposium on Architectures for Networking and Communications Systems*, pp. 25–36, IEEE Press, 2013.

[69] M. A. Jamshed, J. Lee, S. Moon, I. Yun, D. Kim, S. Lee, Y. Yi, and K. Park, "Kargus: a highly-scalable software-based intrusion detection system," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pp. 317–328, ACM, 2012.

[70] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis, "Gnort: High performance network intrusion detection using graphics processors," in *Recent Advances in Intrusion Detection*, pp. 116–134, Springer, 2008.

[71] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman, "Multi-layer packet classification with graphics processing units," in *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, pp. 109–120, ACM, 2014.

[72] A. Kalia, D. Zhou, M. Kaminsky, and D. G. Andersen, "Raising the bar for using GPUs in software packet processing," in *12th USENIX Symposium on Networked Systems Design and Implementation*, NSDI 15, pp. 409–423, 2015.

[73] R. Archibald and D. Ghosal, "A comparative analysis of detection metrics for covert timing channels," *Comput. Secur.*, vol. 45, pp. 284–292, Sept. 2014.

[74] J. Mirkovic and P. Reiher, "A taxonomy of ddos attack and ddos defense mechanisms," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 2, pp. 39–53, 2004.

[75] A. Koay, A. Chen, I. Welch, and W. K. Seah, "A new multi classifier system using entropy-based features in ddos attack detection," in *Information Networking (ICOIN), 2018 International Conference on*, pp. 162–167, IEEE, 2018.

[76] L. Li and G. Lee, "Ddos attack detection and wavelets," *Telecommunication Systems*, vol. 28, no. 3-4, pp. 435–451, 2005.

[77] E. D. Dart, K. A. Antypas, G. R. Bell, E. W. Bethel, R. Carlson, V. Dattoria, K. De, I. T. Foster, B. Helland, M. C. Hester, *et al.*, "Advanced scientific computing research network requirements review: Final report 2015," 2016.

[78] "Impact Cyber Trust." `https://www.impactcybertrust.org`.

[79] "DNS AMPL DDOS." `https://www.impactcybertrust.org/dataset_view?idDataset=580`.

[80] "DDoS CHARGEN." `https://www.impactcybertrust.org/dataset_view?idDataset=693`.

[81] "SSDP DDoS Attack." `https://www.impactcybertrust.org/dataset_view?idDataset=572`.

[82] M. Jonker, A. King, J. Krupp, C. Rossow, A. Sperotto, and A. Dainotti, "Millions of targets under attack: a macroscopic characterization of the dos ecosystem," in *Proceedings of the 2017 Internet Measurement Conference*, pp. 100–113, ACM, 2017.

[83] "Cisco NetFlow." `https://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.html`.

[84] V. A. Siris and F. Papagalou, "Application of anomaly detection algorithms for detecting syn flooding attacks," in *Global Telecommunications Conference, 2004. GLOBECOM'04. IEEE*, vol. 4, pp. 2050–2054, IEEE, 2004.

[85] T. Jin, C. Tracy, and M. Veeraraghavan, "Characterization of high-rate large-sized flows," in *Communications and Networking (BlackSeaCom), 2014 IEEE International Black Sea Conference on*, pp. 73–76, IEEE, 2014.

[86] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites," in *Proceedings of the 11th international conference on World Wide Web*, pp. 293–304, ACM, 2002.

[87] H. Wang, D. Zhang, and K. G. Shin, "Detecting syn flooding attacks," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 3, pp. 1530–1539, IEEE, 2002.

[88] V. Sekar, N. G. Duffield, O. Spatscheck, J. E. van der Merwe, and H. Zhang, "Lads: Large-scale automated ddos detection system.," in *USENIX Annual Technical Conference, General Track*, pp. 171–184, 2006.

[89] S. Peisert, "Security in high-performance computing environments," *Communications of the ACM*, vol. 60, no. 9, pp. 72–80, 2017.

[90] V. Paxson, R. Sommer, S. Hall, C. Kreibich, J. Barlow, G. Clark, G. Maier, J. Siwek, A. Slagell, D. Thayer, *et al.*, "The bro network security monitor," 2012.

[91] A. Mandal, P. Ruth, I. Baldin, D. Król, G. Juve, R. Mayani, R. F. Da Silva, E. Deelman, J. Meredith, J. Vetter, *et al.*, "Toward an end-to-end framework for modeling, monitoring and anomaly detection for scientific workflows," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*, pp. 1370–1379, IEEE, 2016.

[92] P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based network intrusion detection: Techniques, systems and challenges," *computers & security*, vol. 28, no. 1-2, pp. 18–28, 2009.

[93] M. Holkovič, O. Ryšavỳ, and J. Dudek, "Automating network security analysis at packet-level by using rule-based engine," in *Proceedings of the 6th Conference on the Engineering of Computer Based Systems*, pp. 1–8, 2019.

[94] A. Khan, *A feature taxonomy for network traffic*. PhD thesis, University of Delaware, 2019.

[95] I. Sharafaldin, A. H. Lashkari, S. Hakak, and A. A. Ghorbani, "Developing realistic distributed denial of service (ddos) attack dataset and taxonomy," in *2019 International Carnahan Conference on Security Technology (ICCST)*, pp. 1–8, IEEE, 2019.

[96] https://github.com/rgegan/unusual protocols, "Uncommon protocol statistics and detection," 2020.

[97] Zeek, "Faq," 2020.

[98] R. Gegan, B. Perry, D. Ghosal, and M. Bishop, "Insider attack detection for science dmzs using system performance data," in *2020 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2020.

[99] M. Szmit, R. Weżyk, M. Skowroński, and A. Szmit, "Traffic anomaly detection with snort," *Information Systems Architecture and Technology. Information Systems and Computer Communication Networks, Wydawnictwo Politechniki Wrocławskiej, Wrocław*, pp. 181–187, 2007.

[100] J. Grashöfer, C. Titze, and H. Hartenstein, "Attacks on dynamic protocol detection of open source network security monitoring tools," in *2020 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–9, IEEE, 2020.

[101] Spicy, "Generating parsers for protocols files," 2020.

[102] P. Drakos, "Implement a security policy and identify advance persistent threats (apt) with zeek anomaly detection mechanism," 2020.

[103] T. R. Rice, G. Seppala, T. Edgar, E. Choi, D. Cain, and S. Mahserejian, "Development of a host-based intrusion detection and control device for industrial field control devices," in *2019 Resilience Week (RWS)*, vol. 1, pp. 105–111, IEEE, 2019.

[104] https://github.com/zeek/zeek/issues/248, "Generating parsers for protocols files," 2020.

[105] J. Iyengar and M. Thomson, "Quic: A udp-based multiplexed and secure transport," *Internet Engineering Task Force, Internet-Draft draftietf-quic-transport-17*, 2018.

[106] E. Damon, J. Dale, E. Laron, J. Mache, N. Land, and R. Weiss, "Hands-on denial of service lab exercises using slowloris and rudy," in *proceedings of the 2012 information security curriculum development conference*, pp. 21–29, 2012.

[107] "Slowloris attack sample." `https://kb.mazebolt.com/knowledgebase/slowloris-attack/`. Accessed: 2021-04-10.

[108] "Ip null attack description." `https://ddos-guard.net/en/terminology/attack_type/ip-null-attack`. Accessed: 2021-04-10.

[109] S. Hall, "Http stalling detector." https://github.com/corelight/http-stalling-detector/, 2018.

[110] L. Cadalzo, C. H. Todd, B. Obayomi, W. B. Moore, and A. C. Wong, "Canopy: A learning-based approach for automatic low-and-slow ddos mitigation," 2021.

[111] R. Gegan, C. Mao, D. Ghosal, M. Bishop, and S. Peisert, "Anomaly detection for science dmzs using system performance data," in *2020 International Conference on Computing, Networking and Communications (ICNC)*, pp. 492–496, IEEE, 2020.

[112] F. Rezaei, M. Hempel, P. L. Shrestha, and H. Sharif, "Achieving robustness and capacity gains in covert timing channels," in *2014 IEEE International Conference on Communications (ICC)*, pp. 969–974, IEEE, 2014.

[113] "Tilera tilepro64 overview." http://www.tilera.com/sites/default/files/productbriefs/.

[114] M. S. Kang, S. B. Lee, and V. D. Gligor, "The crossfire attack," in *2013 IEEE symposium on security and privacy*, pp. 127–141, IEEE, 2013.

[115] R. K. Gegan, "Unusual protocol monitoring with zeek," (In preparation), 2020.

[116] R. K. Gegan, R. Archibald, M. K. Farrens, and D. Ghosal, "Performance analysis of real-time covert timing channel detection using a parallel system," in *International Conference on Network and System Security*, pp. 519–530, Springer, 2015.

[117] R. K. Gegan, V. Ahuja, J. D. Owens, and D. Ghosal, "Real-time gpu-based timing channel detection using entropy," in *2016 IEEE Conference on Communications and Network Security (CNS)*, pp. 296–305, IEEE, 2016.