Application of Machine Learning to Computer Network Security

by

Jason Haydaman

A Thesis submitted to the Faculty of Graduate Studies of
The University of Manitoba
in partial fulfillment of the requirements of the degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
University of Manitoba
Winnipeg, Manitoba, Canada

University of Manitoba

Abstract

Faculty of Engineering

Department of Electrical and Computer Engineering

Master of Science

Application of Machine Learning to Computer Network Security

by Jason Haydaman

Computer Security covers a wide array of topics, with much of the development in the field happening outside academia. We look at intrusion detection, and evaluate the effectiveness of machine learning in the development of a commercial intrusion detection system (IDS), and compare it with conventional IDS design approaches. We attempt to create novel data sets, and examine the difficulties of extracting new features from network traffic to aid machine learning based systems. Finally, we propose a novel, near-zero overhead method of associating network packets with the process identifier (pid) of their source in real-time and demonstrate a significant performance improvement over existing methods of pid labeling.

Acknowledgements

I would like to thank Prof. Bob McLeod for being my advisor for the past several years and a never ending source of moral support, and Colin Gilmore, who deserves a great deal of credit for both advice and for providing me with so many industrial research opportunities. This work would not be what it is without the both of them. Thank you.

Contents

List of Tables

List of Figures

Chapter 1

Overview of Research

1.1 Introduction and Scope of Research

My research focuses on the application of machine learning to computer network intrusion detection. Specifically, I focus on the detection of sophisticated attackers which are labeled by industry and academia alike as Advanced Persistent Threats (APTs). My threat model assumes nation-state level attackers with access to 0-day (unpatched and previously unknown) vulnerabilities and other low-level backdoors. The assumption therefore is that a security compromise is inevitable, and the focus turns from prevention to detection of compromise. Intrusion Detection Systems (IDS) deployed in industry rely on heuristics and signatures of known attacks [1] and there has not been any use of machine learning in this area outside of academia. My aim is to find an industrially applicable approach focused around machine learning that can make advances towards the detection of APTs. Industrially applicable in this context would mean that any alerts generated by the system would need to be actionable, timely, and with a low false positive rate. In addition, the system needs to be general in its ability to detect novel attacks, as systems to detect known attacks already exist and see deployment in industry.

1.1.1 Problem Statement

Does Machine Learning have anything to offer an IDS? Machine Learning seems suitable for the detection of 0-day attacks in the form of anomalies, but very little use of Machine Learning is found in industrially deployed IDS products. Why?

1.1.2 Why Machine Learning?

I specifically focus on methods around machine learning due to the necessity to adapt over time to changing attacks, and their resistance to analysis by human attackers. A system that has to be manually tuned and updated over time to detect known attack signatures (the current state of the art in industry) is expensive, and will never be able to detect 0-day attacks by definition. Machine learning, in contrast, is uniquely positioned to be able to learn of commonalities in attacks carried out by APTs, if any exist, and detect them in future attacks. Machine learning is also notorious, to the disdain of many, for its inability to explain why it classifies the way it does. In other words, given some trained model that classifies items into classes A or B, the classifier cannot tell you what about a particular item makes it more A-like than B-like, or vice versa. The trained model is opaque to analytic reductionist approaches to understand it, which is a feature in this case. If we cannot determine why our model classifies the way it does, then neither can an APT. This makes it hard for the APT to tune their attacks to avoid detection because they are unable to reason about the model.

1.2 Overview of Research

Recall our aim is to implement an industrially deployable Intrusion Detection System (IDS) based on machine learning classification or anomaly detection

algorithms. We take the following approaches:

1. Interface with existing IDS (Swarm network)
2. Replicate existing anomaly detection methods with SVM
3. Attempt to collect more data
4. Attempt classification of encrypted traffic
5. Propose a method of traffic labeling to potentially help solve the classification problem

1.2.1 Interface with Existing IDS

We first attempted to implement a system to improve the visibility of an ISP IDS to systems behind customer NAT firewalls, and the results proved successful. With this system in place, a new IDS could be used in place of the existing commercial IDS, or in tandem with it. The motivation here was to gain the ability to work with real network data, and have an existing IDS system in place to compare results with.

1.2.2 Replicate SVM Anomaly Detection

We then proceeded to attempt to replicate previous studies of anomaly detection algorithms in IDS applications using a Support Vector Machine (SVM) classifier trained with the ISCX 2012 data set [2]. We used features (Table 5.1) commonly found in the literature, but the results were mediocre. While the classifier could identify some malicious traffic, it required a lot of investigative effort on the part of human analysts to determine what was malicious about it. This was made worse when we implemented a competing IDS based on correlators which had much better results, both with a lower false-positive rate and able to provide context as to what was malicious about the traffic.

1.2.3 Attempt to Collect More Data

As we have no reason to believe that SVM should not work, only that the input data is insufficiently descriptive, we attempted to create our own labeled training data set. For this, a malware sandbox was constructed and 2000 malware samples were run through it. However, our malware was mostly inactive, and we were not able to obtain a sufficient data set from this approach.

1.2.4 Attempt Classification of Encrypted Traffic

Unable to improve our data sets, we decided to try to improve the features we extract from them. Since encryption destroys any features beyond the ones already considered, we decided to see if encrypted traffic identification was possible. Additional motivation here is provided by the realization that in many commercial networks there does not usually exist traffic that cannot be decrypted. Most encryption takes the form of TLS, which many organizations are able to decrypt by using self-signed certificates that are added to the root of trust of all corporate computers. While this is not true of all networks, and is not true of all traffic, being able to know the volume of encrypted traffic, and particularly encrypted traffic that does not match expected protocols like TLS or SSH, seems useful. This proved successful, if allowing for the existence of a filter that exhaustively removes all known compression algorithms from consideration to avoid false-positives. However, this feature is flawed in that an attacker can trivially avoid it using steganographic covert channels, as we demonstrated.

1.2.5 Traffic Labeling

Having exhausted ideas for how to extract more data out of network traffic, the scope of research was expanded to include obtaining information from

the host machines involved in communication. Not wanting to abandon the previous research entirely, we hypothesized that the defining reason why existing data sets do not work for anomaly detection is that traffic from different sources "looks different", i.e., comes from different distributions. Therefore, all traffic looks anomalous in some ways, and some literature supported this hypothesis [1]. Therefore, it would be useful if we could label traffic with its associated source in data sets. The answer to this is the proposed Linux kernel hooks to associate traffic with its source in real-time.

Chapter 2

Literature Review

Included here is a brief literature review. For a more expanded review, see Appendix B [3].

2.1 Intrusion Detection

2.1.1 Network vs. Host

Intrusion detection broadly falls into two categories (which may be used together): network-based and host-based. The difference is the source of available data, or features. A network approach can only provide data from network traffic, or anything derived from network traffic. A host based approach can provide (within technical constraints in implementation difficulty) any information that the operating system of the computer it is deployed on could know. The advantage of a network based approach is that it is much easier to deploy and develop than a host-based approach, because it only requires the installation of a "sensor" on a network at a point where it has visibility into all or the majority of the traffic on the network, such as on a router. This also provides you with a wider context as to what is going on with the entire network, rather than only the activity of an individual computer.

A combined network and host based IDS would amalgamate packet data from network taps with system logs or any other data that is being collected from host machines. This is the most powerful approach generally, as it has the most data available.

2.1.2 Methodologies

There are three common methodologies to intrusion detection: signature-based, specification-based, and anomaly-based. [4]

2.1.3 Signature Based

Signature based detection techniques have the advantage of being simple and effective at detecting known attacks, but are ineffective at detecting unknown attacks. They also require signatures to be updated over time, which requires a significant time investment. Therefore, a signature approach doesn't fit our threat model, because we assume that an APT will use previously unknown attacks. Signature based approaches are also not as applicable to network traffic as they are to traditional virus detection because of the common use of encryption. While malware payloads are also often encrypted, the payload must be decrypted at run-time before it can be executed. This is done using a packer, which is itself subject to signature based detection. Additionally, packer-agnostic detection techniques exist to detect similar malware samples that obfuscate themselves using different packers [5]. That said, the most well known open source network based IDS (SNORT) uses a signature based approach [6].

2.1.4 Specification Based

Specification approaches enhance signature approaches with stateful rules that can match more complicated patterns. For example, if one unsuccessful

login attempt isn't considered worthy of an alert, a specification-based rule could be written to alert on 10 failed login attempts in a short time period. Rules can be chained together in arbitrary ways to come up with heuristics that may be useful in intrusion detection. One rule alone may not prove sufficiently suspicious, but several taken together can paint an enlightening picture. Specification engines often use signature and anomaly based systems as event sources. This allows otherwise noisy rules to still be useful, because rules can be written to only trigger alerts if many rules are triggering at the same time.

Another large advantage of specification based approaches is that they are easy to understand and justify. When an alert is triggered, there is no mystery as to why. Noisy rules can be disabled, or filtered to only show up along with other rules. This provides a lot of opportunity for experts to tune the system, but still requires a lot of manual effort.

2.1.5 Anomaly Based

Anomaly detection methods revolve around trying to classify behavior as normal or anomalous, with the idea being that anomalous is synonymous with malicious. This is not true in general as it is easy to show examples of anomalous behavior that are not considered malicious, but since a previously unseen attack should be an anomaly, the set of all events flagged as anomalies should contain APTs. It is then a matter of filtering out the known-good portion of that set. This makes anomaly detection the method of interest for APT detection.

Existing literature surrounding anomaly detection systems focuses on statistical and machine learning approaches. A survey of IDS techniques [7] shows that there is literature on the use of artificial neural networks (ANN)

[8], Fuzzy Logic [9], Support Vector Machine (SVM) [10], and Genetic Algorithms [11]. The differences between the results of these approaches are minor and varry across different data sets. There seems no general trend towards any particular algorithm being "best" for any particular application.

Axelsson shows that anomaly based approaches are not yet viable: "The cited studies of intrusion detector performance that were plotted and compared indicate that anomaly-based methods may have a long way to go before they can reach these standards, since their false alarm rates are several orders of magnitude larger than what we demand." [12]. He argues that the base rate of malicious events is so low, that even with a false positive rate of less than 1% (which would be considered good in the literature), the number of actual false positives is high (by Bayes theorem).

2.2 Feature Selection

Without relevant features, no algorithm can ever produce useful results. Therefore it seems prudent to focus on feature selection first, before turning to tuning algorithms for performance improvements or incremental decreases in false positive rate. Gonzalez [13] produced a systematic approach to deriving relevant features for use in machine learning classification. He defines relevance as separability. In other words, features that tend not to cluster together, and are not correlated.

The features listed in Table 2.1 are the basic features readily available from network traffic data, without any protocol analysis applied beyond the TCP level. This makes these features general and suitable for application to most common data sets. There are many additional features that could be possible, both derived features (e.g. entropy, or features resulting from Principal Component Analysis (PCA)), and basic features from higher-level protocol analysis (e.g. DNS record response type, HTTP User-agent, TLS chosen

Source IP
Destination IP
Protocol
Source Port
Destination Port
Bytes Transferred[ab]
Packets Transferred[ab]
Number of Packets Containing Data[a]
Packet Inter-arrival Time (IAT)[ab]
Segment Size[b]
TCP Window Size[b]
Unique Byte Count[a]
RFC 1323[c]
TCP Flags Used[a]
Bulk Transmissions/Total Transmissions
Time Spent Idle[a]
Time Spent Transmitting[a]

[a]Including both directions.
[b]Including mean, standard deviation, and inter-quartile ranges
[c]Whether or not RFC 1323 TCP Extensions were being used.

TABLE 2.1: Network Traffic Features

[13][7][14][15]

cipher-suite/parameters, etc...). Based on a survey of the available literature, most research in this area seems to focus on the use of readily available, basic features in common data sets.

2.3 Supervised vs. Unsupervised Learning

Laskov shows that supervised learning outperforms unsupervised learning in the detection of known attacks, but both do poorly in the detection of unknown attacks [16]. He also notes only marginal differences between the results of different approaches to unsupervised or supervised learning. This further supports the notion that feature selection is the most important consideration in intrusion detection, not algorithm selection.

2.4 Datasets

Network traffic inherently contains data sensitive to the privacy of its users, and therefore there is a lack of available data sets in this field [1]. Furthermore, there is little evidence that a generic data set even makes sense given how different networks can be. What classifies as normal traffic on a small business network is going to be different from what classifies as normal on a classified military network. Imagine the problem of trying to detect malicious traffic in a military cyber offensive network, where malicious traffic is the norm. How would you differentiate friendly-malicious traffic from enemy-malicious traffic? Directionality will not work, because you would miss all of the trojans that managed to slip by the detectors and are now phoning home.

It is tempting to ignore that problem of poor data sets, as one can always create their own (an exercise almost universally left to the reader…). However, further problems await, because network traffic has the peculiarity of violating an important assumption of anomaly detection: outliers are not always attacks. Actually, outliers are rarely attacks. Actually, almost everything is an outlier on some time-scale as [1] points out in their measurements of self-similarity of Ethernet traffic, or the lack thereof. They suggest limiting the scope of machine learning algorithms to reduce misclassifications.

Nevertheless, there does exist a de facto standard data set, for lack of alternatives: the KDD data set [17][18]. KDD is a widely-criticized [18][19][20] synthetic data set originally generated in 1998 and refined in 1999. It is commonly used for comparative analysis of algorithms, and is mentioned here only for completeness. It remains a useful data set for performance benchmarking, but it cannot be used to train a classifier for use on real networks.

2.5 Conclusions

Many papers [21][22][8] compare false positive rates and training performance of various algorithms, such as Multi-Layer Perceptron (MLP) vs Self Organizing Map (SOM), but there seems to be less research focused on feature selection. This makes sense from an academic perspective, where the interest is in repeatability and comparability to existing research, but there is no justification for the features commonly used (protocol, source port, destination port, source IP address, destination IP address, ICMP type, ICMP code, raw data length, raw data) [7].

Furthermore, little consideration seems to be given for the base rate of success when talking about false positives. A classifier trained to 99% accuracy, or a 1% false positive rate, would be considered good, but when the base rate of a malicious event is low, Bayes theorem shows that the majority of actual detection's will be false positives. Axelsson provides an excellent analysis of this problem [12] and concludes that an actual false positive rate of 1/100,000 is required for an IDS to be useful.

Chapter 3

Data Collection

3.1 Introduction

A discussion of an application of machine learning is incomplete without a discussion of data sources. We combined a mixture of novel data sets that we collected, and static data sets publicly available. The primary source of novel test network data for this project was MERLIN (Manitoba Education, Research, and Learning Information Network). MERLIN has a large enough traffic volume and a commercial network IDS system (TippingPoint) such that we had plenty of data to test with, and the ability to correlate our results with TippingPoint's results. We also obtained a data set from University of Twente [23] that contained 14.2 million malicious flows (IP, port, protocol). The ISCX 2012 data set [2] was also used. We also attempted to generate a set of known-bad traffic that didn't rely on pre-labeled data or commercial IDS-labeled data by setting up a malware sandbox environment that could capture network traffic directly from malware.

3.2 Novel Data Sources

3.2.1 MERLIN Network

We installed an instance of the BRO Network Security Monitor on MERLIN's network, which provided us the following data:

1. Flow data (IP, port, protocol, duration, and size of transfer)
2. HTTP data (HTTP headers, referrer, user-agent)
3. DNS data (query and response, query type, TTL)
4. SSH data (raw traffic, and a successful login heuristic)
5. TLS data (certificate information, including validity)

For training data, we were able to use MERLIN's HP TippingPoint IDS to label the data we collected from BRO. Specifically, we collected a sub-set of the data that we knew represented one MERLIN customer with known-malicious traffic contained within it. We then dumped IDS logs from TippingPoint and correlated alerts (filtering out benign alerts such as bittorrent traffic) with flow data from our data set to label it.

3.3 Malware Sandbox

We setup a malware sandbox in a virtualised environment that automated the deployment of malware samples to virtual machines. These virtual machines were monitored for traffic volume and prematurely killed if traffic volume was large to prevent knowingly participating in a Distributed Denial of Service (DDoS) attack. Approximately 2000 samples of malware were obtained from [24]. Of these samples, only 11 showed any level of traffic above baseline. One started flooding traffic on port 53 UDP and was killed,

presumably a DNS reflection DDoS attack. Two more downloaded payloads from IP addresses in Russia and proceeded to install them, but the payloads after installation remained inactive. The rest either tried to connect to IP addresses that were unresponsive, presumably old, inactive command and control networks, or showed a higher than normal volume of SMB and NetBIOS (Windows file sharing protocols) traffic, presumably scanning the local network for shares to infect. In the end, we were not able to obtain a significant enough sample of data from this approach.

Chapter 4

Swarm Sensor Network

4.1 Introduction

The Swarm Sensor Network was the first step in the design of an industrially deployable IDS system based around network anomaly detection. We worked closely with MERLIN (Manitoba Education, Research and Learning Information Networks), an ISP for the majority of Manitoba's school divisions, with the idea that our anomaly detection system would be deployed on their networks. School networks are an excellent environment for an IDS because of the large user base and high volume of traffic.

4.1.1 Design Overview

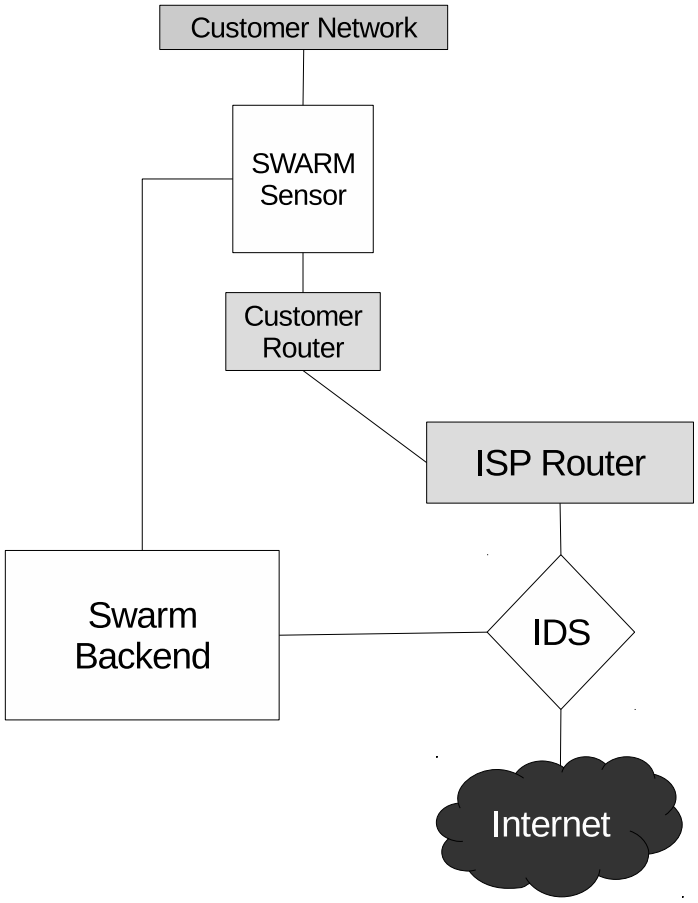

FIGURE 4.1: Swarm Sensor Network Overview

Figure 4.1 Shows a design overview of the Swarm Sensor Network. The two main components are the sensor and the backend. Sensors sit behind customer NAT firewalls or routers and have full layer-2 visibility into the customers network, while the backend sits on the ISP network and connects to the IDS to receive alerts. The sensors connect to the backend over the ISP's network, so the whole system is self-contained to the ISP-customer network topology and is never exposed to the entire internet. Swarm sensors are invisible on the network except for their phone-home connection to the backend. This would typically happen on a different network segment than the main customer network.

Swarm sensors have 3 Ethernet interfaces. One management interface, and two monitoring interfaces (in and out). The in side faces the customers internal network and the out side faces their router. Ideally, all traffic exiting and entering the network from the internet will go through these interfaces. The monitoring interfaces are provided by a NIC with hardware failover to passthrough. In other words, if the sensor stops responding, or loses power, the NIC throws a hardware switch that causes the line to turn into a straight passthrough wire, preventing disruption of customer traffic. The management interface works out-of-band from the main customer traffic, ideally, and connects to the backend over SSH.

Obviously, to obtain permission to install one of these sensors on a customers network requires considerable security considerations. None of the interfaces on the sensor respond to unsolicited traffic. There are no daemons listening on any ports, and so there is no network attack surface. The sensor connects to the backend over SSH, and refuses to connect if the backends host-key has changed (which is set before the sensor is installed on-premise), which prevents a Man-in-the-Middle (MitM) attack. The sensor passively records all packets that pass through the monitoring interfaces. To prevent sensitive information from leaking in the case of physical theft of the sensor, packets are logged to an encrypted partition of the hard drive. The keys to the partition are not stored on the device, and are randomly generated 64-character strings, unique to each sensor. The keys are passed once the SSH tunnel has been established, validating both ends, allowing the sensor to unlock its partition and begin logging packets. If the sensor attempts to connect from an unrecognized IP block (since we know the valid range for each customer), we refuse the connection and do not transmit keys.

The sensor serves two purposes. From MERLIN's perspective, the need was to help the network administrators of customers to pinpoint problem systems on their network. MERLIN's IDS was capable of detecting attacks

that were not being stopped, because the customer did not have the required knowledge or monitoring tools to know which of their systems were responsible. Because these systems were behind NAT routers, MERLIN couldn't help. With Swarm, we had the ability to associate net flows with MAC addresses and internal IP addresses of the systems responsible. Usually, this was enough to track down the problem systems and remove them from the network to be cleaned up. From our perspective, this could serve as a source of data and a testing facility for our IDS system. Initially we deployed it using MERLIN's IDS, but the idea was that at some point it could use our as-yet-undeveloped IDS as well to provide multiple alert sources.

4.1.2 Results

The system worked as advertised, and was deployed in many different school divisions, as well as Brandon University. MERLIN was able to help IT administrators clean up their networks, and the utility and value of MERLIN's IDS was enhanced greatly.

Chapter 5

Intrusion Detection

5.1 Introduction

With the swarm sensor network in place (see Chapter 4), the goal now is to replace the commercial IDS with our own, hopefully improved, IDS. We considered two approaches here, machine learning and extending an existing commercial IDS (OneStone) with modules that we call correlators (which can be thought of as a cognitive approach), that do not make use of machine learning, for a comparison. Correlators implement algorithms thought up by domain experts, which provide a nice contrast to machine learning algorithms. The goal is to see if machine learning can detect intrusions that we cannot come up with a simple heuristic to detect, as that would provide value over the commercially available systems.

5.2 Machine Learning

5.2.1 Choice of Algorithm

The main algorithm used for machine learning was Support Vector Machine (SVM). We chose SVM because of literature review indicating that SVM classifiers in this domain are able to achieve a low false positive rate, in one case on the order of 10^{-5} [25] by using an ensemble of one-class SVM classifiers.

Duration of Flow
Protocol Number
Source Port
Destination Port
Number of packets per flow
Number of bytes per flow
TCP flags
Number of flows from source to destination IP in the past 300 seconds
Number of flows from source port to destination port in the past 300 seconds
Number of flows from destination to source IP in the past 300 seconds
Number of flows from destination to source port in the past 300 seconds

TABLE 5.1: Network Traffic Features Used in SVM Classifier

Some work [26] has also been done on using one-class SVM for unsupervised anomaly detection. Ultimately, given our literature review showing that feature selection is likely more important than choice of algorithm [16], the decision on SVM vs Neural Networks is largely arbitrary and results with one algorithm are likely generalisable to other algorithms. We used libsvm [27] for our implementation.

5.2.2 Feature Selection

The features listed in Table 5.1 were used based on taking a cross section of the features available to us in our data sources and features commonly used in the literature [7].

5.2.3 Training

We focused most of our training on the ISCX 2012 data set [2] due to its recency and similarity to our own sources of test data. Best results were found using a Gaussian radial basis function as the kernel. For the SVM soft-margin parameter C and kernel parameter γ we used a grid search with cross-validation as the parameter selection method, and used F-score to find

	Classified non-malicious	Classified malicious
Test flow non-malicious	165091	2595
Test flow malicious	2328	7732

TABLE 5.2: SVM Confusion Matrix

optimal parameters to obtain the confusion matrix shown in Table 5.2. This results in an accuracy of 97% and a false positive rate of 1.46%.

5.2.4 Testing Methodology

The following process was used over several weeks to tune our IDS:

1. Raw data collected from MERLIN customer network via BRO network sensors.
2. Feature extraction from raw data into the feature set in Table 5.1 .
3. SVM classifies the data set as malicious or non-malicious.
4. For flows labeled malicious by SVM, gather additional information (MERLIN IDS logs, additional BRO data, DNS logs, HTTP logs).
5. Human analysis of malicious flows and additional information provides ground-truthing.
6. Develop heuristics to filter out observed false-positives after ground-truthing.
7. Repeat steps 3 – 6 until the false positive rate is acceptable.

5.2.5 SVM Results

Initial results were poor, with a false-positive rate of 80%. We found the addition of heuristics useful in culling common alerts that could be ignored such as port scans (which while malicious, are so common that they are not worth investigating, and therefore not an actionable alert), and white listing

domains such as NTP servers and content delivery networks. With the addition of these filters, the false positive rate dropped to 50%. This is actually not bad, because this is the false positive rate after taking the base rate into account, that is, 50% of flows flagged as malicious actually were malicious or worth investigating.

5.3 Traditional IDS

To compare against the machine learning work, we also implemented an IDS based on more traditional detection techniques that we refer to as correlators. These correlators were implemented as Python modules for a commercial IDS system developed by Seccuris Inc (bought by Above Security and now owned by Hitachi) called OneStone. Correlators work based on a pattern recognition system. Correlators are dormant until incoming events (e.g. firewall log, packet capture, DNS lookup, etc...) match a correlators pattern. On pattern match, the correlator activates, and all subsequent events of interest (based on event type and optionally filtered by event parameters) are passed to the correlator. The correlator is arbitrary Python code, and so it can maintain state however it wishes, and emit additional events into the system. After a set timeout of inactivity, or on demand, correlators deactivate until activated again by the pattern matching system. This design allows for flexible and extremely powerful alerts that target very specific behavior.

5.3.1 Reverse Geographic Communication Correlator

This proved to be one of our most useful correlators for detecting attacks. It has no false positives by design. The correlator activates on unsolicited inbound connection attempts from countries outside the '5 Eyes' (Canada, United States, UK, Australia and New Zealand) that are rejected or dropped.

The remote IP address is kept in the correlators state, and the correlator begins listening for all outbound connection attempts. If any outbound connection has a destination IP address contained within the correlators state, an alert is thrown. The idea behind this correlator is that unsolicited inbound traffic is indicative of port-scanning, and so these addresses are assumed to be malicious. This is even more likely to be true with the addition of the geographic filter. Triggering the alert on an outbound connection attempt is what makes this correlator so useful, because it implies that the attacker was successful in penetrating or otherwise influencing some target machine in the network to connect back, and this is always something that should be investigated.

5.3.2 User-Agent Deprecation Detector

The idea behind this correlator is to monitor a hosts HTTP traffic and throw an alert when it is noticed that a User-Agent has gone backwards in time, or deprecated itself. The most common behavior of malware after infecting a system is to make an outbound HTTP request to obtain a second-stage payload to install on the host machine. Since malware is usually self-contained to maximize portability (that is, contains statically-compiled libraries and doesn't rely on system libraries being present), the HTTP client used by malware will be different from what is commonly used by the users of the system. Sometimes malware User-Agent's will contain typographical errors or otherwise be obviously forged by human inspection. This correlator alerts on any noted deprecation's (decrease in version number of Browser), or strange User-Agents that don't fit common profiles.

This correlator becomes useless if looking at NAT-ted addresses because of the variety of User-Agent's coming out of an entire network. Windows Update also likes to false-positive this correlator by using older versions of

Internet Explorer. Also, any web developers that commonly test the corporate website in older browser versions also get flagged incorrectly.

5.3.3 HTTP Connection without DNS Lookup

This is self-explanatory. Not necessarily indicative of malicious behavior, and requires large grace-periods to account for DNS caching, but is an example of cross-protocol correlations that are possible.

5.3.4 DNS Tunnel Detection

A DNS tunnel works by using DNS as a two-way communication channel. Information can be encoded into the sub-domain of a DNS request, which will be routed by the DNS system to the authoritative name server. By using a TTL of 0, it is possible to ensure that intermediary name servers do not cache results. The authoritative name server then replies, and in addition to a well-formed reply to the request, information can be encoded in the response. This creates a communication channel that almost no networks block, because it is difficult to block DNS on most networks.

This correlator was based on work done by one of our team members in his M.Sc. thesis [28]. It works by recording a Domain Length Weighted Entropy (DLWE) metric for each top-level domain observed over DNS. DLWE is calculated by forming a probability distribution over 20 samples or 60 seconds of DNS requests to the same top-level domain, whichever comes first, calculating the entropy of the distribution, and then multiplying that by the average length of the DNS request. This metric provides a measure of DNS tunnel likelihood, with values below 50 indicating no likely DNS tunnel, and values over 150 indicating an almost certain DNS tunnel.

That it was possible to implement such a correlator demonstrates the power of the correlation engine of OneStone.

5.3.5 Correlation Results

The main benefit to using correlators instead of machine learning is that correlators provide descriptive alerts with no ambiguity as to their cause. Machine learning only tells you that, for reasons unknown, something was flagged as suspicious. Therefore, correlators by their nature have no false positives (though whether a correlator correlates perfectly with malicious behavior is not certain and in general not true). As mentioned, our Reverse Geographic Communication Correlator was extremely useful in finding compromised machines on customer networks, and was by far more useful than any other correlator or machine learning technique used.

5.4 Overall IDS Results

From the results of both the machine learning and correlator approach to IDS, we are forced to conclude that the machine learning approach, with the data sets we had available, was inferior to the correlation approach, for the detection of known attacks. While the false positive rate was acceptable, the amount of work required by human analysis to determine whether something was a false positive, and what exactly had taken place, was too great compared to the simplicity of a correlator being able to tell the analyst exactly what happened, with no ambiguity. For the detection of 0-day attacks, we can draw no strong conclusion as we do not have any data sets with 0-day attacks in them.

Chapter 6

Modeling

6.1 Introduction

Our previous efforts at anomaly detection with SVM were underwhelming compared to the results obtained using correlators. One potential reason for this is the feature set used for our classifier, which was based on available data as opposed to what "makes sense", or, what an expert would expect to be correlated with malicious behavior. In this Chapter we examine the difficulties of crafting new features that might be used to build better classifiers.

6.2 Problems with Feature Selection

By limiting data sets to including network traffic only, there is a relatively fixed amount of information available. If that information doesn't contain the features necessary to discriminate malicious traffic from normal traffic, then no algorithm, feature selection method, or any amount of cleverness can create that information out of nothing. Furthermore, most prior work in this area only uses the same ten or so features, and even efforts to differentiate all possible features (a few hundred) into a set of statistically meaningful features yields only 37 [13]. This is not outside the range of what modern machine learning can handle, and so the natural course of action would then be to obtain data sets of every feature and blindly pass that over to machine

learning systems to train and classify. This approach doesn't work in general, though it can be made to work for more specific applications such as classifying DNS tunnels [28]. We see then that the effort needs to be focused not on feature selection, but on designing new features, keeping in mind that we cannot create information where it does not exist.

6.3 Designing new Features

In [28] we saw that designing a metric DLWE (Domain Length Weighted Entropy) allowed the design of a novel DNS tunnel detection technique. Can we do something similar for intrusion detection? It is not clear how that could be done for all possible attacks. If we reduce the scope of consideration to only attacks that result in successful exfiltration of data from a network, then detecting that exfiltration becomes something we can focus on.

Since we still wish to consider only applications that commercial IDS has not already solved (unless we can do better), it seems sensible to not focus on data-loss prevention techniques (DLP). DLP is available as commercial products from a variety of vendors, and typically works by watermarking sensitive documents with a digital signature that is monitored for in outgoing email, HTTP traffic, printers, etc. DLP products tend to be invasive and require both host and network components in order for them to work. They also rely on a critical assumption that encryption isn't being used, or can be passively removed (by replacing authentic TLS certificates with manufactured certificates and then adding the corporate public key to your root of trust).

Encryption being a problem is a common theme. Any feature that you might hope to be able to extract from network traffic other than high-level flow data is destroyed by encryption. Maybe the presence of encryption itself might be a useful metric? Attackers are not likely going to use unencrypted

connections, and they will also tend to prefer connections that cannot be decrypted by subverting TLS. Therefore, the presence of encrypted data leaving the network that cannot be decrypted may be a useful metric. We then direct our focus to determining whether some arbitrary data is encrypted.

6.4 Detecting Encrypted Traffic

A symmetric cipher e takes plain-text p and some key k and performs the operation of encryption by some means transforming p into cipher-text c which should only be able to be transformed back into p by some decryption function d, requiring the same key k. Note that this is not the only kind of cipher that is possible, but symmetric ciphers are used to encrypt the bulk of data on the internet. Specifically, AES is one of the most widely used symmetric ciphers available today and is the cipher that we will be dealing with.

An interesting property of encrypted data is that cipher-text c should be statistically independent of both key k and plain-text p. Furthermore, c is always statistically identical to uniformly distributed random noise, with each byte of c being independent of every other byte. These properties are covered in detail by Shannon as confusion and diffusion [29]. While this property is hard to prove analytically, an empirical study of AES has shown it to be true for this cipher [30]. Since this property holds for AES, and should hold for every symmetric cipher that is used today, a good test for whether something has been encrypted is to check if it is statistically indistinguishable from independent and identically distributed uniform random noise. The assumption we make here is that legitimate messages which are not encrypted will not be random noise.

Symmetric ciphers operate in what are known as modes. A mode of encryption defines how the plain-text gets converted into cipher-text. The simplest method is directly in what is known as ECB mode (electronic codebook). The problem with ECB is that identical plain-texts turn into identical cipher-texts for the same key, which is easily detectable. The mode that we use is known as counter mode, or CTR. CTR mode encrypts successive integers with AES, and then XORs the output with the plain-text to produce cipher-text. If a 128-bit key is used, then this mode is abbreviated as AES-128-CTR.

6.4.1 Entropy Estimation

Since our aim is to detect if our message is random, it is natural to employ the use of entropy. The entropy of the bit-stream produced by some truly random process is 1 bit per sample, i.e.

$$H(\mathbf{X}) = -\sum_{x \in \mathbf{X}} p(x) \log_2 p(x) \tag{6.1}$$

$H(\mathbf{X}) = 1$ bit when $p(x) = (1/2, 1/2)$. The problem is that we cannot use the definition of entropy to estimate the entropy of our bit-stream, because the probability distribution is unknown. Furthermore, any attempts to estimate the distribution by averaging over a large number of packets will fail because the distribution may not be stationary or ergodic, and packets may not be independent or even identically distributed. Fortunately, there is a large quantity of literature on testing the randomness of finite length bit-streams for the purposes of testing the quality of pseudo-random number generators (PRNGs). The National Institute of Standards and Technology (NIST) has published a large document containing numerous tests that can be applied to a candidate bit-stream [31]. The output of these tests is formulated in terms of a p-value. NIST recommends a p-value of 0.01, meaning

that we have 99% confidence that a sequence that is claimed to be random is actually random. The two tests that we are considering here are Approximate Entropy and Maurer's Universal Test. Both are known to identify AES output as random [30]. If p-value > 0.01 then $\mathbf{X}$ is considered to be random. Otherwise, $\mathbf{X}$ is considered to be the output of some Markov process of order $\kappa < d$

Approximate Entropy

As the name suggests, Approximate Entropy is a test designed to estimate the true entropy of a source that produced a given sample sequence. For an m-bit sample $\mathbf{X} = (x_0, x_1, \ldots, x_{m-1}), x_i \in \{0, 1\}$, let $1 < n \leq m$ be a sample size and with $0 \leq i < n - 1$ let $\tilde{x}_i^d$ be the ith overlapping d-tuple.

$$\tilde{x}_i^d = (x_i, x_{i+1}, \ldots, x_{i+d-1}) \tag{6.2}$$

Then, if $\mathbf{a} = (a_1, \ldots, a_d) \in \{0, 1\}^d$ let $\tilde{\pi}_{\mathbf{a}}^d$ be the relative frequency of $\tilde{x}_i^d = \mathbf{a}$.

$$\tilde{\pi}_{\mathbf{a}}^d = \frac{1}{n} |\{0 \leq i < n : \tilde{x}_i = \mathbf{a}\}| \tag{6.3}$$

For a sample $\mathbf{X}$ from a stationary ergodic source, an asymptotically unbiased estimator for the entropy H can be found

$$\hat{H}_f^d = -\sum_{\mathbf{a} \in \mathcal{A}^d} \tilde{\pi}_{\mathbf{a}}^d \log_2 \tilde{\pi}_{\mathbf{a}}^d + \sum_{\mathbf{a} \in \mathcal{A}^{d-1}} \tilde{\pi}_{\mathbf{a}}^{d-1} \log_2 \tilde{\pi}_{\mathbf{a}}^{d-1} \tag{6.4}$$

Provided that the order of the source is less than d, $\hat{H}_f^d$ will converge to H as $n \to \infty$. A statistic for $\hat{H}_f^d$ is $\hat{I}^d$, which has a chi-squared distribution of order $2^d - 2^{d-1}$.

$$\hat{I}^d = 2n(1 - \hat{H}_f^d) \xrightarrow{D} \chi^2_{2^d - 2^{d-1}} \tag{6.5}$$

$\hat{H}_f^d$ is referred to as approximate entropy. The p-value for approximate entropy can be shown [31] to be

$$P_{\text{entropy}} = \frac{1}{\Gamma(2^{d-2})} \int_{\hat{I}^d/2}^{\infty} t^{2^{d-2}} \mathrm{e}^{-t} \mathrm{d}t \tag{6.6}$$

Maurer's Universal Test

Maurer's Universal Test aims to measure the relative compressibility of a sequence. A sequence which fails to be able to be significantly compressible is deemed to be random. For an m-bit sample $\mathbf{X} = (x_0, x_1, \ldots, x_{m-1}), x_i \in \{0, 1\}$, let $1 < n \leq m$ be a sample size and with $0 \leq i < n - 1$ let $\bar{x}_i^d$ be the ith non-overlapping d-tuple, and $m \geq d \cdot n$.

$$\bar{x}_i^d = (x_{i \cdot d}, x_{i \cdot d+1}, \ldots, x_{i \cdot d+d-1}) \tag{6.7}$$

Then, if $\mathbf{a} = (a_1, \ldots, a_d) \in \{0, 1\}^d$, let $\bar{\pi}_{\mathbf{a}}^d$ be the relative frequency of $\bar{x}_i^d = \mathbf{a}$.

$$\bar{\pi}_{\mathbf{a}}^d = \frac{1}{n} |\{0 \leq i < n : \bar{x}_i = \mathbf{a}\}| \tag{6.8}$$

Let $T(i)$ be the return time to $\bar{x}_i^d$. In other words, if $\bar{x}_i^d = \mathbf{a}$, then $T(i)$ is the smallest value for which $\bar{x}_{i-T(i)}^d = \mathbf{a}$.

$$T(i) = \min\{1 \leq j \leq i + 1 : \bar{x}_{i-j}^d = \bar{x}_i^d\} \tag{6.9}$$

Then, with $Q = 10 \cdot 2^d$ being a "warm-up" time,

$$\hat{H}_r^d = \frac{1}{d \cdot n} \sum_{i=Q}^{Q+n-1} \log_2 T(i) \tag{6.10}$$

This test needs $Q + n$ non-overlapping tuples. A statistic for $\hat{H}_r^d$ is

$$\hat{N}^d = \frac{\hat{H}_r^d - \mathrm{E}\{\hat{H}_r^d\}}{\sqrt{\mathrm{Var}\{\hat{H}_r^d\}}} \xrightarrow{D} N[0, 1] \tag{6.11}$$

In general, finding the expected value and variance here is difficult. Precomputed tables and approximations exist [32] for various values of d. It can be shown that

$$P_{\text{maurer}} = \text{Erfc}\left(\frac{|\hat{H}_r^d - \text{E}[d]|}{\sqrt{2} \cdot (0.7 - (0.8/d) + (1.6 + (12.8/d))(\lfloor n/d \rfloor - Q)^{-4/d}) \cdot \sqrt{\text{V}[d]/(\lfloor n/d \rfloor - Q)}}\right) \tag{6.12}$$

6.4.2 Implementation and Results

The two entropy estimators described were implemented in C++. The Cephes math library was used for numerical computation of p-values, and the pcap library was used to read in packet data. The program also provides the ability to read in raw data from any source, allowing it to be used to compute the randomness of arbitrary data, not only packet data. The input byte-stream from either source is first converted into an input bit-stream in order to ensure faithful reproduction of the tests as described.

Test 1 – Secure Copy

The first test was copying a 32MB file over scp (secure copy, a common Unix application that copies files over an encrypted connection), which in this case was encrypting the connection with AES-128-CTR. We show the results for various values of d, with $n = 271,335,304$ bits. The largest value of d in both cases was selected based on NIST recommendations [31]. The results are shown for Approximate entropy in Table 6.1 and Maurer's test in Table 6.2.

Since this connection was encrypted, it's worth investigating why the tests disagree. The approximate entropy for smaller values of d is very close

d	$\hat{H}_f^d$	$1 - \hat{H}_f^d$	p-value
3	0.999999966537	3.3 x 10^{-8}	0.0011
4	0.999999960918	3.9 x 10^{-8}	0.0066
5	0.999999939104	6.1 x 10^{-8}	0.0073
6	0.99999981646	1.8 x 10^{-7}	7.3 x 10^{-9}
7	0.999999694224	3.1 x 10^{-7}	5.4 x 10^{-11}
8	0.999999322518	6.8 x 10^{-7}	5.1 x 10^{-25}
23	0.988490359896	0.012	0

TABLE 6.1: Test 1 – Approximate Entropy Results

d	$\hat{H}_r^d$	E[·]	σ	p-value
3	2.40139590766	2.401607	0.000063	0.000789
4	3.31104289578	3.311225	0.000093	0.051136
5	4.25317755531	4.253427	0.000121	0.038842
6	5.21754085234	5.217705	0.000145	0.256502
7	6.19604145991	6.196251	0.000166	0.208490
8	7.18380291046	7.183666	0.000186	0.459370
14	13.1677329848	13.167693	0.000280	0.885252

TABLE 6.2: Test 1 – Maurer's Universal Test Results

to 1, however, it's not close enough given the very large sample size n. Therefore, there should be something about this sequence that is lowering the entropy. Maurer's test indicates that this sequence is random for $d > 3$, indicating that, for $d > 3$, this sequence cannot be significantly compressed. Examining the sequence in detail, shown in Figure 6.1 reveals the likely source of the problem:

The first several packets of an scp connection include an initial handshake where both sides agree on a cipher to use to encrypt the payload. This information is transmitted in a plain-text format which is clearly not random. The inclusion of this in the test is likely why the Approximate Entropy results indicate that the sequence is not random, because the first several thousand bytes are not. This also explains why Maurer's test is insensitive to this deviation. Since there are only on the order of 10^4 bits of non-random data, but the entire sample size is on the order of 10^9 bits, the relative compressibility of the sequence is very small.

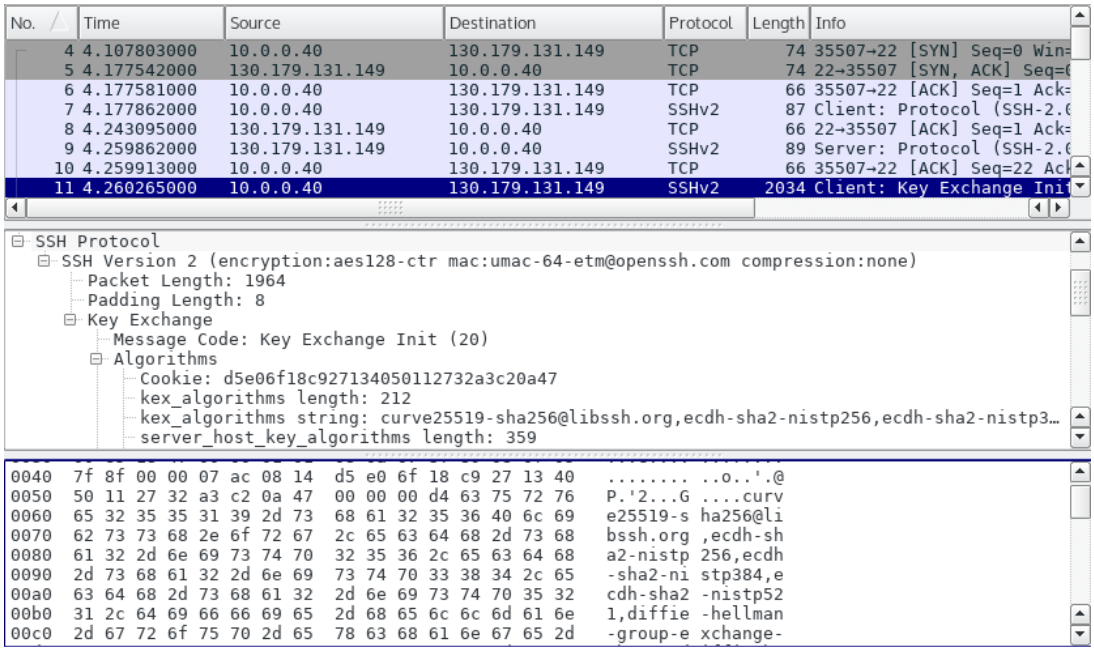

FIGURE 6.1: Packet-level view of sequence used for Test 1

The remaining tests concentrate on files instead of network traffic. We have already shown that the presence of non-random headers and protocol negotiation preambles in network traffic skews our entropy estimates, but that Maurer's test resists them. Outside of protocol-specific headers, there is little difference between a bit-stream in packet payloads and the bit-stream in the files sent over a network.

Test 2 – Random Number Generator

Motivated by the results of Test 1 and noticing that Approximate Entropy is sensitive to any deviations in randomness, we propose a second test where we use the direct output of the high quality PRNG `/dev/urandom` on Linux. Here, we keep the sample size as close as possible to Test 1 to eliminate any deviations in results due to differences in the sample size. Here, we use $n = 271,335,304$ bits. Again, the largest value of d used here comes from NIST recommendations. The results are shown for Approximate entropy in Table 6.3 and Maurer's test in Table 6.4.

d	$\hat{H}_f^d$	$1 - \hat{H}_f^d$	p-value
3	0.999999996353	3.6 x 10^{-9}	0.74
4	0.999999983472	1.7 x 10^{-8}	0.34
5	0.999999976466	2.4 x 10^{-8}	0.69
6	0.999999944182	5.6 x 10^{-8}	0.55
7	0.999999852098	1.5 x 10^{-7}	0.082
8	0.999999695656	3.0 x 10^{-7}	0.015
23	0.98876176956	0.011	0

TABLE 6.3: Test 2 – Approximate Entropy Results

d	$\hat{H}_r^d$	E[·]	σ	p-value
3	2.40151779016	2.401607	0.000063	0.16
4	3.31127405977	3.311225	0.000093	0.60
5	4.25345058195	4.253427	0.000121	0.84
6	5.21760394486	5.217705	0.000145	0.48
7	6.19622467094	6.196251	0.000166	0.88
8	7.1836726432	7.183666	0.000186	0.97
14	13.1680352142	13.167693	0.000280	0.22

TABLE 6.4: Test 2 – Maurer's Universal Test Results

As expected, Maurer's test continues to perform well as this sequence is completely incompressible. Approximate Entropy also works well until d becomes large. While this may be an artifact of the test or perhaps a problem due to lack of precision in the numbers, I was unable to find any large ($n > 10^9$) sequence which could satisfy the Approximate Entropy test at NIST-recommended d.

Test 3 – 64KB Random Number Generator

So far, Maurer's test has worked well for large sample sizes, and Approximate Entropy has worked well provided there are no non-random sub-sequences present in the file. Now we test for smaller sample sizes. Here, we use a 64KB file generated by `/dev/urandom`, leading to $n = 524,288$. The results are shown for Approximate entropy in Table 6.5 and Maurer's test in Table 6.6.

Since Maurer's test requires more samples than Approximate Entropy,

d	$\hat{H}_f^d$	$1 - \hat{H}_f^d$	p-value
3	0.999997978564	2.0 x 10^{-6}	0.71
4	0.999994838575	5.2 x 10^{-6}	0.71
5	0.999970024324	3.0 x 10^{-5}	0.012
6	0.999943425199	5.7 x 10^{-5}	0.0023
7	0.999901136018	9.9 x 10^{-5}	0.0013
8	0.999819549399	1.8 x 10^{-4}	0.00036
14	0.988685273705	0.011	0

TABLE 6.5: Test 3 – Approximate Entropy Results

d	$\hat{H}_r^d$	E[·]	σ	p-value
3	2.40101871682	2.401607	0.000063	0.68
4	3.31284285698	3.311225	0.000093	0.45
5	4.25427222556	4.253427	0.000121	0.76
6	5.22207645428	5.217705	0.000145	0.19

TABLE 6.6: Test 3 – Maurer's Universal Test Results

NIST recommends using a maximum of $d = 6$ for this length sequence. Despite that, the test still performs well.

Test 4 – Plain-text

We have evaluated the performance of the tests against encrypted and random data, and they have overall performed well. But in order for these tests to be useful, they must correctly reject unencrypted, non-random data. Here, we use a sample sequence of The Divine Comedy [33], giving $n = 1,940,728$ bits. The p-value for all Approximate Entropy tests was 0 regardless of d, and Maurer's Test only had $p > 0.01$ when $d = 3$, $p = 0$ otherwise. Therefore, plain-text is correctly rejected.

Test 5 – Large File Compression

A potential problem for both tests is compressed files. Compressed files should have high entropy, and since they have already been compressed, Maurer's test may fail to find any further compressibility in them. This would falsely classify compressed files as encrypted or random, and would

limit the utility of these tests for this application. For this test, we use an xz-compressed archive of the Linux kernel available from `https://kernel.org`. This file contains the Linux kernel source code compressed with `xz`, which internally uses LZMA/LZMA2 compression. For this test, $n = 653,609,600$ bits. The results are shown for Approximate entropy in Table 6.7 and Maurer's test in Table 6.8.

d	$\hat{H}_f^d$	$1 - \hat{H}_f^d$	p-value
3	0.999999998802	1.2 x 10^{-9}	0.81
4	0.999999995822	4.2 x 10^{-9}	0.71
5	0.999999985961	1.4 x 10^{-8}	0.30
6	0.999999963463	3.7 x 10^{-8}	0.036
7	0.999999928765	7.1 x 10^{-8}	0.010
8	0.999999837575	1.6 x 10^{-7}	4.1 x 10^{-6}
24	0.990672011756	0.0093	0

TABLE 6.7: Test 5 – Approximate Entropy Results

d	$\hat{H}_r^d$	E[·]	σ	p-value
3	2.40162543192	2.401607	0.000063	0.65
4	3.31118660912	3.311225	0.000093	0.53
5	4.25337048717	4.253427	0.000121	0.47
6	5.21761450852	5.217705	0.000145	0.33
7	6.19624124132	6.196251	0.000107	0.93
8	7.18352704939	7.183666	0.000120	0.25
15	14.167318356	14.167488	0.000188	0.37

TABLE 6.8: Test 5 – Maurer's Universal Test Results

Indeed, both tests completely fail to distinguish encryption from compression. Approximate Entropy correctly rejects the sample for larger values of d, but since that test seems to always reject for large values of d that should not be considered a success here.

6.5 Covert Channels

We have shown that it is possible to determine, with some caveats, whether arbitrary data is encrypted. Encryption is not generally distinguishable from

compression, except that we can assume that it is possible to enumerate all compression algorithms in common use and exclude them from consideration. While both encrypted and compressed messages are indistinguishable from a purely theoretical perspective, in practice compressed messages are meant to be decompressed, which means that compressed messages contain a structure to them that allows them to be identified. Therefore, in practice, one could with some effort filter out all known compression algorithms, and if the resulting data still has high entropy as determined by an unbiased estimator, then it is safe to conclude that the data is encrypted (if you make the assumption that nobody intentionally sends actual random noise).

There is a related problem to consider from the perspective of data exfiltration in the form of covert channels. Even though it is theoretically possible to detect encrypted traffic, it may be impossible to do so in the face of an adversary that is attempting to fool the detector. For example, if encrypted traffic is compressed, then it would be removed from consideration by the compression detector, and not be flagged. So, we would need to perform encryption detection on the decompressed data stream. This is where covert channels come into play, because covert channels are able to hide messages in otherwise benign data.

Steganography is the practice of embedding a secret message M inside of an innocuous looking message C (cover-text) such that no one except the intended recipient of M is able to determine what M is, or that it even exists. The combination of cover-text and secret message is called stegotext S. Intuitively, this seems like it should always be possible. Such a scheme to do so [34] could be taking an uncompressed digital video file as a cover-text, and making the kth bit of M the least significant bit of the nth pixel of the kth frame, with n being the kth integer on a shared one-time-pad (a one-time-pad is a list of randomly generated numbers shared by the sender and receiver of M, but nobody else). We expect that M would be completely lost

in the natural noise of the video frames.

Information Theory tells us that any information source can be arbitrarily compressed as close as we wish to entropy, removing all redundancy. Real compression schemes used in practice often fall short of this asymptotic limit. Assuming that M is random (which it could easily be if we first encrypted it), then entropy is additive.

$$H(S) = H(C) + H(M) \tag{6.13}$$

Therefore, to detect the presence of M, we would have to detect the deviation in $H(C)$ of $H(M)$ in S. Since a careful attacker will keep $H(M)/H(C)$ very small, any hope of detecting M requires extremely accurate knowledge of $H(C)$. The entropy estimators evaluated in the previous section have asymptotic limits that converge to true entropy for infinite length messages, but an accurate evaluation of their errors for finite lengths [32] is beyond the scope of research here.

6.6 Results

We conclude that it is impossible to say with certainty that data exfiltration has not occurred by observation of network traffic only. The best we can do is provide an upper bound on the amount of data that may have been exfiltrated, but since such a bound would grow with total traffic volume over time, this is not a useful bound, as all someone can conclude from it is that you should prevent all communication from occurring if you wish to prevent data exfiltration from happening. This is not a useful result.

We have shown that it is possible to detect encrypted traffic that is not being intentionally hidden in a covert channel, which does account for the vast majority of encrypted traffic seen in practice. While it could be argued

that DNS tunnels are a form of covert channel, they seem more often used as a universal means of subverting firewalls as opposed to trying to hide the fact that communication is taking place.

Chapter 7

Enhanced Data Collection Tools

7.1 Introduction

In the previous sections, we've explored the possibilities of network based intrusion detection. In this section, we expand the scope of research to look into what is possible with a combination of host and network based IDS. One of the common problems with only examining network traffic is a lack of context as to what host process is responsible for that traffic. It would be extremely helpful if the network packets were labeled to identify which host process was responsible for them, and, even better, if there was an additional label that maps to some representation of the process state. As we saw in Chapter 6, the presence of unencrypted handshakes in otherwise encrypted connections shows that even for a single connection for a single process, traffic output is not stationary. However, if some notion of state could be developed, such that we could identify SSH in the handshake state vs SSH in the encrypted state, then perhaps the traffic distributions within those states would be stationary. No operating system has this feature, unfortunately, so the intent here is to determine how it may be possible to build such as system.

7.2 Host Based Context Mining

Park et al. [35] published a study where they generated a system call graph via introspection of running malware samples in a sandbox. This graph could then be run though graph mining algorithms to extract subgraphs. By comparing these subgraphs against other malware samples, it becomes possible to identify distinct malware samples that come from the same family of malware by their common behavior graphs. Can this idea can be adapted to a host+network based IDS to provide context to network traffic?

7.3 Design

Consider the behavior graph system [35] as a given. Can we map that into a state label for machine learning applications? In general, the state of a program can be thought of as a snapshot of its code and data in memory, and CPU registers, but this is too granular and would yield far too many states. What we want is some fuzzy/qualitative method of saying that the program is currently exhibiting some kind of "behavior", and call that its current state. Examples could be reading from the filesystem, waiting for user input, running heavy computational work (e.g. a browser executing Javascript), performing DNS resolution, or, in our example from Chapter 6, undergoing cipher suite negotiation.

If we assume that [35] makes it possible to distinguish between those states, then we hypothesize that machine learning would start being useful. Example, if under normal circumstances DNS lookups happen with a small amount of associated network IO, and no file IO, but in the future we observe a program in the DNS lookup state exhibiting large network IO and file IO at the same time, maybe that means it's exfiltrating data over DNS? From a pure network perspective, all we see is a large amount of traffic. We

might see a large amount of traffic over DNS, but if the attacker decides to do DNS over a SOCKS5 proxy over a SSH tunnel, then we would have no way of knowing that this large outbound transfer is supposed to be DNS. With the added context of a behavior graph state, it would immediately become suspicious.

Assuming that has been solved already, we need only consider how to map individual packets to processes in a system, since once we can associate packets with a process, the process can be introspected to determine its current behavior state. How can we achieve this?

7.3.1 Linux Kernel Networking Hooks

The background to design here is covered in Appendix A. We want to provide a relatively high performance way of associating packets with the process that emitted it. Performance matters here, because if the method has too much per-packet overhead then we will either drop packets or be unable to identify the source of all of them. Since all processes in Linux have a unique ID, their `pid`, we aim to map packets to their source `pid`. With existing tools, the best we can currently do is to use conntrack as an event source for connection state changes, and on state change re-scan `/proc` for sockets, and map those socket `fd`s to `pid`s (again via `/proc`). The problem with this approach is that scanning `/proc` is slow, particularly having to re-scan all sockets on every connection change. Also, when dealing with malicious binaries, scanning on connection change may not be sufficient as binaries can fork and allow their children to assume control of the socket, or, we may miss very short lived connections. Since `tcpdump` is the data source we used for most of our experiments, it's natural to want to augment `tcpdump` output with `pid`s. However, `libpcap` hooks into the kernel at too low of a level for `pid` information to exist. `libpcap` uses the `ETH_P_ALL` protocol type, and thus hooks in

at the `__netif_receive_skb_core` and `dev_queue_xmit_nit` levels (pretty much right after/before the packet leaves or enters the driver). In the send case, this is ok, as the `skb` carries sock information with it until it's eventually deallocated, but in the receive case this doesn't work as we don't yet even know if this packet is destined for the system, or if netfilter is going to drop it before it hits any applications. We would need to process the connection hash tables early on in order to get the required information at this stage.

A better approach is to hook into netfilter. Netfilter provides hooks at very granular points in the network stack. For outgoing packets, we can hook at the postrouting stage, right before packets egress the NIC, and capture full `skb` and `sk` information there. For ingress, we can hook right before packets are allowed for delivery to applications, at which point we know where they're going. Netfilter also already provides a kernel to userspace communication protocol called netlink. We can use that protocol to create a kernel to userspace communication socket, similar to the socket that `libpcap` uses, except with an additional structure containing `pid` information.

This sounds like a good approach, but it has a problem. The `skb`s that netfilter deals with don't have `pid` information in them. They do have socket information, and that socket information does map to `struct file`, but the `struct file` does not contain a pointer to `pid`. Also, there is no existing lookup table in the kernel that holds such a map. This makes sense, because it is not information that the kernel needs to do its job, and Linux data structures are optimized around the kernel's functionality. When sending a packet, the `pid` doesn't matter because the packet is about to be sent to the hardware and dropped from memory, so who cares who sent it. On receive, the packets destination is to a receive buffer, not a process. A process reads the receive buffer, and so there must exist a map from `pid` to `fd` (and there is, via `current->files->fdt`), but there is no map in the other direction. Therefore, we have two options. Implement such a map, or be forced to iterate

over all processes, check their open file descriptors, and see if any match the file descriptor we have for the socket from the `skb`. This is expensive and not very elegant, but simple, easy to implement, and self-contained within a kernel module. Adding the lookup table requires more work outside the scope of a simple kernel module, but simplifies lookup and adds functionality to the kernel.

Since we aim for industrial deployment of this system, requiring a custom kernel seems too demanding. A kernel module is more palatable as it can be loaded and unloaded on demand. So, what is the best we can do with a kernel module? Since we need `pid` information, hooking netfilter doesn't work without having a reverse map that would require kernel modifications to introduce. But, we can hook the read and write system calls. We can even do this from userspace, but since we must make the assumption that userspace is compromised (otherwise we don't need an IDS), we note that hooking from userspace is easy to bypass for malware. The common way of hooking from userspace would be to override `LD_PRELOAD` in the processes environment before running it to inject our own hooks into its library path. When the dynamic linker at run-time tries to load the shared object that allows is to call `libc`'s `read` or `write` syscalls, our hook would come up first and be used instead. Malware can trivially bypass this by asking for the `libc` symbols explicitly, or clearing its `LD_PRELOAD` environment variable and then forking itself. We also want this to be a global hook for the entire system, preferably even the kernel, in the event of a kernel rootkit. Therefore, the hook should happen in kernel space.

How do we hook from a kernel module without using netfilter? After some experimentation, it was found that a netfilter hook is sufficient for outbound packets. While the `skb` doesn't contain `pid` information, in testing we find that evaluating `current->pid` in a netfilter `NF_INET_POST_ROUTING` hook always correctly returned the `pid` of the process that called send. So, we can

use a netfilter hook in the outgoing direction.

Incoming is much harder. `current->pid` is 0 in this case because the kernel is receiving the packet, not userspace directly. Furthermore, nowhere does the kernel actually call userspace to tell it to come and get its packet. Instead, all that happens is the scheduler, as it iterates over all processes deciding which processes are to be scheduled to run, will notice that if a process is blocked on a read operation on a file descriptor that now has data in it available to be read, that it should schedule that process to run. When the process resumes, its read operation completes, and it has now read the data from the socket. This is a very indirect process, because all the kernel sees when the packet comes in is the receive buffer for this file descriptor. Multiple file descriptors may be pointing to the same receive buffer, each of which may belong to different processes, so the information of which processes are waiting on this buffer to have data is only contained in the processes themselves, not the buffer. This prevents, without modification of the kernel, us from knowing who can read inbound packets from netfilter.

It's actually worse than that though, because the process may fork itself before reading the packet, and so even if we had a map of who could read the packet when it came in, that map could be different by the time the packet is actually read. In general, we don't even know that it is ever read. So it's an ill-defined question to ask which process received a packet when the packet arrives on the NIC. At best, that information would be delivered asynchronously at some later point in time.

So how can we hook on read? All of the operations that are performed on file descriptors are stored in function pointer tables. If we overwrite these function pointer tables to point to our own hooked functions, then we can hook whatever calls we wish. The problem with that is these tables are marked as `const` which means the compiler puts them into a section of the executable that is mapped into read-only memory at runtime, and so trying

to override them causes the kernel to segfault. We find through experimentation that the following allows disabling segfault checks on x86_64 CPUs.

```
/* returns true if write protection was enabled */
static bool disable_write_protect(void)
{
    unsigned long eflags;

    /* btr doesn't work on control registers. */
    asm volatile (
         "mov %%cr0, %%rax\n"
         "btr $16, %%rax\n"
         "pushf\n"
         "pop %0\n"
         "mov %%rax, %%cr0\n"
         : "=r" (eflags)
         :
         : "rax"
    );

    return (bool)(eflags & 1);
}

/* returns true if write protection was disabled */
static bool enable_write_protect(void)
{
    unsigned long eflags;

    /* bts doesn't work on control registers. */
    asm volatile (
         "mov %%cr0, %%rax\n"
         "bts $16, %%rax\n"
         "pushf\n"
         "pop %0\n"
         "mov %%rax, %%cr0\n"
         : "=r" (eflags)
         :
         : "rax"
    );

    return (bool)(~(eflags & 1));
}
```

This is extremely dangerous code and it must be used with pre-emption disabled and write protection must be re-enabled before returning to userspace. It works by flipping the write-protect bit in register `cr0` (control register)

that the memory management unit (MMU) of the CPU checks to determine whether it should consult `rwx` permissions in the page table. Here is an example of using it to overwrite `inet_recvmsg`:

```
preempt_disable();
enable_write_protection = disable_write_protect();

((struct proto_ops *)&inet_stream_ops)->recvmsg = inet_recvmsg_patched;

if (likely(enable_write_protection))
        enable_write_protect();

preempt_enable();
```

The netfilter hook for outgoing packets is far more straightforward.

```
static struct nf_hook_ops ops = {
    .hook = recv_from_netfilter,
    .hooknum = NF_INET_POST_ROUTING,
    .priority = NF_IP_PRI_FIRST,
    .pf = PF_INET
};

unsigned int recv_from_netfilter(void *priv, struct sk_buff *skb,
        const struct nf_hook_state *state)
{
    send_skbuff(skb);  /* handles userspace communication */
    return NF_ACCEPT;
}

nf_register_hook(&ops);
```

7.4 Results

Using the netfilter hook discussed above, we are able to successfully map `pids` to outgoing packets in real-time in the kernel. We have three implementations here for comparison, and a summary of the performance of each, measured as processing time in userspace per packet, is tabulated in Table 7.1. Also included are graphs of the processing times per packet for the kernel method (Figure 7.1), the userspace method (Figure 7.2), and the control

method (no lookup) (Figure 7.3). The graphs are also labeled with whether or not the correct `pid` was resolved by the method. The connection being monitored is a file transfer over scp, and each experiment used the same file, the same source, and the same destination. The kernel method is the netfilter hook already explained, the userspace method is obtaining the `pid` mappings in userspace using the `/proc` filesystem, and the control method provides no mapping as a baseline comparison. The results greatly favor the in-kernel netfilter hook over userspace. The three implementations here varry only in their `pid` resolution technique and packet source. For the kernel module approach, we used a netlink socket to communicate with userspace, using a new, unused netlink protocol definition: `socket(AF_NETLINK, SOCK_RAW, 22)`. For the userspace and no resolution techniques (provided for baseline timings), we used the same method that libpcap uses: `socket(PF_PACKET, SOCK_RAW, ETH_P_ALL)`. The userspace component of all methods was implemented in python2.7, using the `scapy` library for packet parsing. All methods used two threads, one that consumed data from the socket and wrote it into a queue, and the main thread that read data from the queue and processed it. This was in an attempt to not drop packets due to socket buffers filling up.

The userspace lookup was implemented using the following algorithm (pseudocode)

```
inode_map = build_lookup_table()  // from /proc/net/{tcp,udp}
// inode_map maps socket fd inodes to (src, sport, dst, dport)
sock_map = {}
for pid in /proc:
    for fd in /proc/pid:
        s = stat(/proc/pid/fd)
        if s.st_mode & S_IFSOCK:  # from include/uapi/linux/stat.h
            sock = inode_map[s.st_ino]  # st_ino is inode
            if sock:
                sock_map[sock] = pid
```

This algorithm was run on every incoming packet to associate the `pid` with the observed four-tuple. It may seem inefficient to run this algorithm

on every packet, and it is. However, it would be incorrect to make optimizing assumptions about the nature of socket communication here when dealing with potentially compromised systems. A process that begins communicating on a socket may not continue (if, for instance, it forks itself, and a child assumes responsibility for the socket). Alternatively, malware may reuse existing source ports when making new connections. Caching previously seen mappings may lead to errors in identification that malware could take advantage of. Furthermore, as can be seen in Figure 7.2, a sufficiently short-lived connection may evade being labeled entirely.

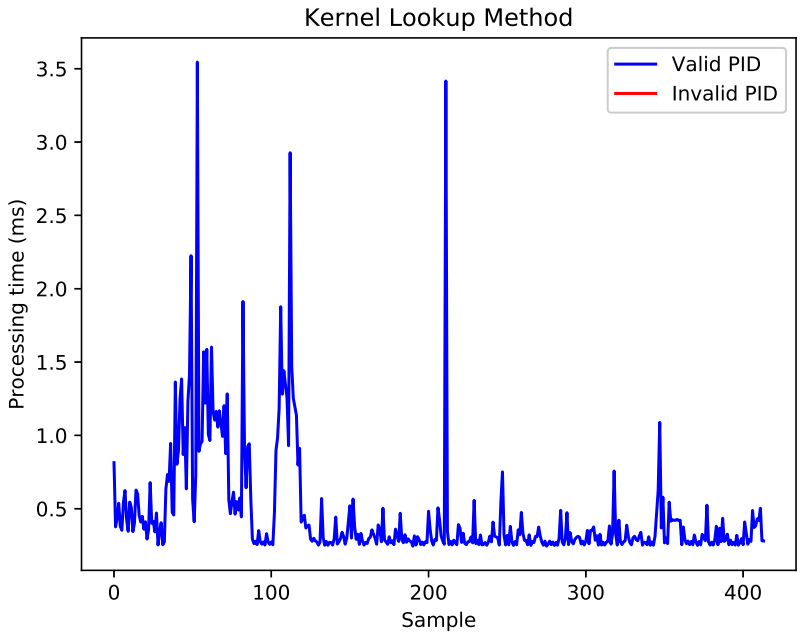

FIGURE 7.1: Per-packet processing time using in-kernel pid resolution

method	min (ms)	max (ms)	mean (ms)	stdev (ms)
Kernel	0.245	3.55	0.463	0.394
Userspace	8.54	33.2	9.59	1.74
Control	0.259	6.765	0.418	0.377

TABLE 7.1: Packet Processing Times

The main problem with the userspace mapping method, aside from being slow, is that the mapping is built asynchronously with respect to the actual

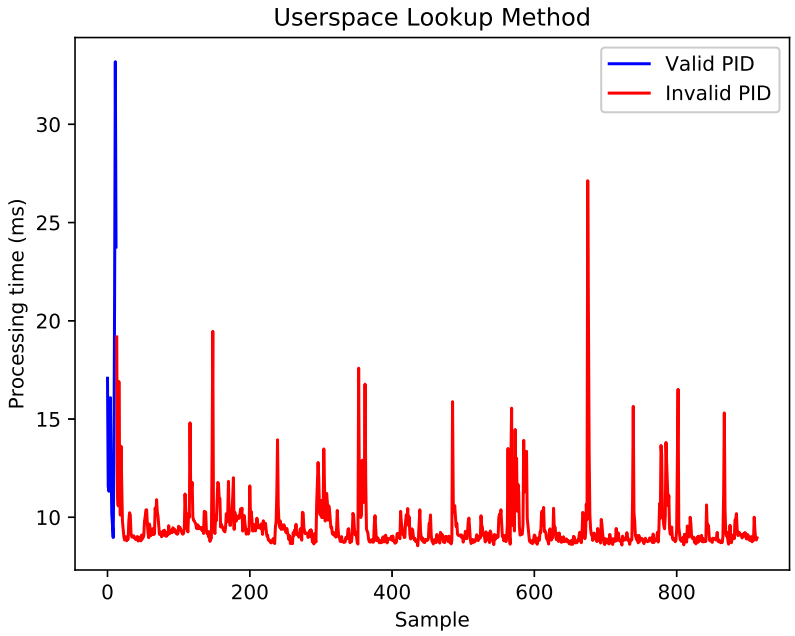

FIGURE 7.2: Per-packet processing time using userspace pid resolution

communication taking place. That is, while our socket is still receiving incoming packets, the connection that we are currently trying to lookup has already disappeared. This is made worse by the lookup method being so slow, which is unavoidable due to the number of syscalls involved. Even with a more efficient implementation, there is no guarantee that we will see the correct results using the userspace lookup method, if the connection is sufficiently short in duration. This explains the presence of invalid `pids` in Figure 7.2, by the time we got around to reading those packets from our queue, the connection had already finished.

The noise in the graphs can be explained by our primitive collection method. During the time the connection was active, a large burst of packets entered the system. This overwhelmed our program causing it to spend most of its time in the packet collection thread. Since this was implemented in python, the Global Interpreter Lock (GIL) prevented the processing thread from running during this time. Time was measured as the time delta between reading a packet from the queue, until just before outputting the mapping. If we got

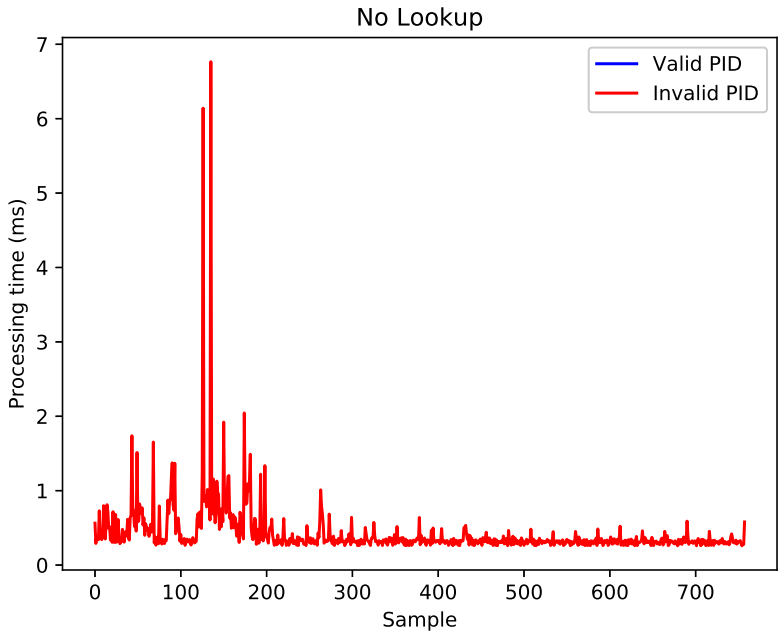

FIGURE 7.3: Per-packet processing time with no pid resolution

interrupted by the socket consumer thread in the middle of this (as would happen often during the beginning of the run), that explains the spike in timing at the beginning. A better userspace implementation would smooth out the timings, but the average timings during non-bursty periods would likely remain unchanged.

As can be seen, our proposed method is not only faster (35 times faster best case, 20 times faster average case, 9 times faster worst case), but robust against short lived connections as well. Since the netfilter hook lives in the kernel, it sees everything. Our approach can also be observed to have near-zero overhead compared to the baseline control experiment. There are improvements to be made in the userspace implementation, as well as in the netlink communication channel, as the buffer is prone to filling up during periods of high packet load, but the technique is fundamentally sound. A more robust implementation would solve those problems.

Chapter 8

Discussion

8.1 Conclusions

The main inference we can draw from this research is that correlators based on simple rules work better than traditional SVM for IDS applications, given the restrictions imposed by available data and the required false positive rate for human consideration, which is reflected by the techniques used in industrially deployed IDS systems. However, we must admit that there are still some 0-day attacks that a correlator would not be able to detect, and a general anomaly detection system would be preferable in these cases. In our experimentation with SVM, the trained model had reasonable performance (false positive rate) when tested with data from the same data set, but performed poorly on data from a different network, i.e. the model did not generalize. Likely, the main reason for this is that there is too much variance in network traffic to define what "normal" means, and therefore, cannot identify anomalies (or rather, that everything appears anomalous). As we saw in Chapter 6, even when considering the output of a single connection of a single instance of a single program, there was no fixed probability distribution for the traffic, which made obtaining an estimate of its entropy, as an attempt to classify it as encrypted or not (a potentially useful feature), somewhat ill-defined. This is the main justification for the hypothesis that simply throwing more data at

the problem may not help traditional SVM perform better. However, if considerably more data were available, it cannot be ruled out that methods such as deep learning may perform better. In the absence of much larger data sets, it seems necessary to improve the available features used for training, which we suggest take the form of mapping process state to network traffic.

8.2 Pid Labeling

We developed a novel `pid` labeling technique, motivated by the need to associate process information with packets in data sets. We demonstrate our method as between 9 and 35 times faster than existing methods, with near-zero overhead compared to regular packet capturing, while being more robust against short-lived connections than the existing methods. We believe this approach, in combination with some additional future work, could make anomaly detection in network traffic a viable option in the future.

8.3 Future Work

We hypothesize that if the input data sets could be labeled based on their source and the current "output distribution" of the source, then we could attempt to define what is normal and anomalous both within distributions, and perhaps the relative frequency of distributions. We define output distribution as the hypothetical probability distribution or Markov process that models the packet output of a program at some point in time. Possibly [35] could be a starting point for future research in this area, if the graph clusters their method produces map to output distributions of traffic. It is likely this approach will face problems in practice, both because the graph clustering algorithms are too slow to run in real-time, and the program needs to have already completed in order to obtain a complete system call graph, which

makes point-in-time "state" labeling difficult. Nevertheless, if graph behaviors do map to stationary output distributions, that would be useful to know, as we then have reduced the problem to something that we need to solve faster, instead of not knowing how to solve at all. Alternatively, it may be worth considering if a different notion of state could be developed. If either case proved promising, then in combination with the `pid` labeling approach we demonstrate, anomaly detection would be worth a revisit.

Bibliography

[1] R. Sommer and V. Paxson, "Outside the closed world: On using machine learning for network intrusion detection", in *Security and Privacy (SP), 2010 IEEE Symposium on*, IEEE, 2010, pp. 305–316.

[2] A. Shiravi, H. Shiravi, M. Tavallaee, and A. A. Ghorbani, "Toward developing a systematic approach to generate benchmark datasets for intrusion detection", *computers & security*, vol. 31, no. 3, pp. 357–374, 2012.

[3] C. Gilmore and J. Haydaman, "Anomaly detection and machine learning methods for network intrusion detection: An industrially focused literature review", in *Proceedings of the International Conference on Security and Management (SAM)*, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2016, p. 292.

[4] H.-J. Liao, C.-H. R. Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review", *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 16–24, 2013.

[5] G. Jacob, P. M. Comparetti, M. Neugschwandtner, C. Kruegel, and G. Vigna, "A static, packer-agnostic filter to detect similar malware samples", in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, Springer, 2012, pp. 102–122.

[6] *Snort*, `https://www.snort.org`, Accessed: 2017-07-11.

[7] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A survey of intrusion detection techniques in cloud", *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 42–57, 2013.

[8] L. M. Ibrahim, "Anomaly network intrusion detection system based on distributed time-delay neural network (dtdnn)", *Journal of Engineering Science and Technology*, vol. 5, no. 4, pp. 457–471, 2010.

[9] P. Tillapart, T. Thumthawatwor, P. Santiprabho, *et al.*, "Fuzzy intrusion detection system", 2015.

[10] W.-H. Chen, S.-H. Hsu, and H.-P. Shen, "Application of svm and ann for intrusion detection", *Computers & Operations Research*, vol. 32, no. 10, pp. 2617–2634, 2005.

[11] W. Li, "A genetic algorithm approach to network intrusion detection", *SANS Institute, USA*, vol. 15, pp. 209–216, 2004.

[12] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection", *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 3, pp. 186–205, 2000.

[13] J. A. Gonzalez, "Numerical analysis for relevant features in intrusion detection (narfid)", AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH GRADUATE SCHOOL OF ENGINEERING and MANAGEMENT, 2009.

[14] S. Chebrolu, A. Abraham, and J. P. Thomas, "Feature deduction and ensemble design of intrusion detection systems", *Computers & security*, vol. 24, no. 4, pp. 295–307, 2005.

[15] S. Mukkamala and A. Sung, "Feature selection for intrusion detection with neural networks and support vector machines", *Transportation Research Record: Journal of the Transportation Research Board*, no. 1822, pp. 33–39, 2003.

[16] P. Laskov, P. Düssel, C. Schäfer, and K. Rieck, "Learning intrusion detection: Supervised or unsupervised?", *Image Analysis and Processing–ICIAP 2005*, pp. 50–57, 2005.

[17] *Kdd cup 1999 data*, `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`, Accessed: 2017-07-12.

[18] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the kdd cup 99 data set", in *Computational Intelligence for Security and Defense Applications, 2009. CISDA 2009. IEEE Symposium on*, IEEE, 2009, pp. 1–6.

[19] M. Sabhnani and G. Serpen, "Why machine learning algorithms fail in misuse detection on kdd intrusion detection data set", *Intelligent data analysis*, vol. 8, no. 4, pp. 403–415, 2004.

[20] J. McHugh, "Testing intrusion detection systems: A critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory", *ACM Transactions on Information and System Security (TISSEC)*, vol. 3, no. 4, pp. 262–294, 2000.

[21] J. Cannady, "Artificial neural networks for misuse detection", in *National information systems security conference*, 1998, pp. 368–81.

[22] Á. Grediaga, F. Ibarra, F. García, B. Ledesma, and F. Brotóns, "Application of neural networks in network control and information security", *Advances in Neural Networks-ISNN 2006*, pp. 208–213, 2006.

[23] R. R. R. Barbosa, R. Sadre, A. Pras, and R. Meent, "Simpleweb/university of twente traffic traces data repository", Centre for Telematics and Information Technology, University of Twente, 2010.

[24] *Virusshare*, `https://virusshare.com/`, Accessed: 2017-07-31.

[25] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee, "Mcpad: A multiple classifier system for accurate payload-based anomaly detection", *Computer networks*, vol. 53, no. 6, pp. 864–881, 2009.

[26] P. Laskov, C. Schäfer, I. Kotenko, and K.-R. Müller, "Intrusion detection in unlabeled data with quarter-sphere support vector machines", *Praxis der Informationsverarbeitung und Kommunikation*, vol. 27, no. 4, pp. 228–236, 2004.

[27] C.-C. Chang and C.-J. Lin, "Libsvm: A library for support vector machines", *ACM transactions on intelligent systems and technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[28] M. Himbeault, "A novel approach to detecting covert dns tunnels using throughput estimation", Master's thesis, University of Manitoba, 2014. [Online]. Available: `https://mspace.lib.umanitoba.ca/handle/1993/23550`.

[29] C. E. Shannon, "Communication theory of secrecy systems*", *Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

[30] P. Hellekalek and S. Wegenkittl, "Empirical evidence concerning aes", *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 13, no. 4, pp. 322–333, 2003.

[31] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, and E. Barker, "A statistical test suite for random and pseudorandom number generators for cryptographic applications", DTIC Document, Tech. Rep., 2001.

[32] J.-S. Coron and D. Naccache, "An accurate evaluation of maurer's universal test", in *Selected Areas in Cryptography*, Springer, 1999, pp. 57–71.

[33] D. Alighieri, *Divine Comedy, Longfellow's Translation, Hell*, trans. by H. W. Longfellow. Project Gutenberg, 1997. [Online]. Available: `http://www.gutenberg.org/ebooks/1001.txt.utf-8`.

[34] R. J. Anderson and F. A. Petitcolas, "On the limits of steganography", *Selected Areas in Communications, IEEE Journal on*, vol. 16, no. 4, pp. 474–481, 1998.

[35] Y. Park, D. S. Reeves, and M. Stamp, "Deriving common malware behavior through graph clustering", *computers & security*, vol. 39, pp. 419–430, 2013.

[36] *The rcu api, 2010 edition*, `https://lwn.net/Articles/418853/`, Accessed: 2017-07-30.

[37] *Napi*, `https://wiki.linuxfoundation.org/networking/napi`, Accessed: 2017-07-30.

Appendix A

Overview of the Linux Kernel Networking Subsystem

This is an overview of the internals of the Linux networking subsystem. It's meant to be followed side-by-side with the kernel source code and aims toward directing the reader to the relevent bits of code, pointing out where things are defined and provides context to aid in understanding of the code base. This was written against the 4.4.x series kernel, and follows the path taken to setup a new UDP connection, send a packet, and close the connection. The motivation for this follows from Chapter 7 and the overall goal is to hook into the kernel to enhance packet collection capabilities to include process information with each packet, so in addition to an overview of the kernel subsystems, it is also mentioned where various hooks into the kernel are possible.

Here is the example we are considering:

```
$ echo helloworld | strace nc -u 10.0.0.50 1234 -c
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
setsockopt(3, SOL_SOCKET, SO_LINGER, {onoff=1, linger=0}, 8) = 0
setsockopt(3, SOL_SOCKET, SO_REUSEADDR, [1], 4) = 0
connect(3, {sa_family=AF_INET, sin_port=htons(1234),
    sin_addr=inet_addr("10.0.0.50")}, 16) = 0
read(0, "helloworld\n", 1024) = 11
write(3, "helloworld\n", 11) = 11
shutdown(3, SHUT_RDWR) = 0
close(3) = 0
```

Here, we use netcat `nc` to do the work of creating the connection and sending helloworld over the socket, and we use `strace` to show the system calls that this uses. System calls are the API that the kernel exposes to userspace programs. We will now follow the system calls above and trace through what happens in the kernel.

A.1 Socket System Call

```
socket(PF_INET, SOCK_DGRAM, IPPROTO_IP) = 3
```

`socket` is defined in `linux/net/socket.c` via `SYSCALL_DEFINE3` as

```
int socket(int family, int type, int protocol);
```

The `PF_INET` family specifies that this is an IPv4 connection. The type is `SOCK_DGRAM` which means that this is a connectionless, datagram socket where each datagram has a fixed maximum length. Protocol is more complicated. `man 2 socket` specifies that if a particular (type, family) combination has only one acceptable protocol choice, then you can specify 0 for protocol and the socket will be defined with the only possible protocol for this type. If there is more than one possible protocol, an unspecified default will be chosen. As we will see later, the default for `SOCK_DGRAM` is UDP. `IPROTO_IP` is defined as 0 in `linux/include/uapi/linux/in.h`. A return value of 3 indicates that the file descriptor associated with this socket is 3.

A.1.1 Socket Initialisation

In the system call, the first thing the kernel does after some checks is to allocate a `struct socket`, by calling `sock_create`. `struct socket` is Linux's implementation of classic BSD sockets. This allocation happens in sockfs which is a virtual filesystem in kernel memory, acting as a translation layer between files and sockets. With sockfs, a userspace application can operate on a socket

file descriptor in the same way that it operates on any file descriptor, and sockfs handles the translation into packets.

`sock_create` calls `__sock_create` which calls `sock_alloc()` to perform the actual allocation of the socket object in the vfs. `sock_alloc()` allocates the inode with `new_inode_pseudo(sock_mnt->mnt_sb)` where `sock_mnt` is the `struct vfsmount` that represents the virtual mountpoint of the sockfs, and `mnt_sb` points to the superblock of the filesystem, from which the filesystem can be searched or enumerated by the relevent filesystem kernel functions. `new_inode_pseudo` calls `alloc_inode(sock_mnt->mnt_sb)` which performs the actual allocation by way of the function pointer `mnt_sb->s_op->alloc_inode`. This function pointer is set in `sock_init()` indirectly, by way of passing `struct file_system_type sock_fs_type` to `kern_mount()`. An in-depth explanation of the kernel filesystem internals is omitted for brevity, but `sock_fs_type` contains a function pointer `.mount = sockfs_mount`, which in turn calls `mount_pseudo` passing it a pointer to `struct super_operations sockfs_ops`, containing function pointers to `sock_alloc_inode` and `sock_destroy_inode`. Therefore, `mnt_sb->s_op->alloc_inode = sock_alloc_inode`. `sock_alloc_inode` initializes a `socket_alloc` and `socket_wq` structure.

`socket_alloc` contains both the socket and inode structures, and is retrieved from `kmem_cache_alloc`, which is part of the Linux memory management subsystem that is able to cache unused objects to be returned quickly rather than having to perform `kmalloc` allocations on every new socket creation. The first such object is created in `sock_init` by calling `init_inodecache()`. `socket_wq` is a wait queue structure which contains a linked list of threads that are waiting on I/O for this socket. It's used by the scheduler and a detailed coverage of the scheduler is omitted for brevity.

How does `sock_init()` get called? During kernel bootup in `linux/init/main.c` the kernel calls `do_initcalls()` which in turn loops over an array of function pointers to init routines. This array of function pointers is in turn stored in

an array `initcall_levels`, and the kernel loops over the levels executing low-level initcalls before higher level initcalls. Kernel modules and subsystems can register themselves to be called in this manner via macros defined in `linux/include/init.h`. In the case of the socket subsystem, `core_initcall(sock_init)` is called. `core_initcall` is a macro which takes the pointer to the passed function and stores it in the ELF binary `.initdata` section with the help of the linker and compiler directives. In this case, the initcall is defined as level 1, or "core" and hence `core_initcall` is the appropriate macro name.

After the socket is allocated on sockfs via `new_inode_pseudo`, a macro `SOCKET_I(inode)` gets a pointer to the socket structure. `SOCKET_I` is

```
#define &container_of(inode, struct socket_alloc, vfs->inode)->socket
```

`container_of` is another commonly used macro in the kernel, which "casts a member of a structure out to the containing structure". In other words, given some member of a structure, the name of the member, and the structure type, return the actual structure it's contained in. It's defined in `linux/include/linux/kernel.h` and works with the help of the `offsetof` compiler directive. The socket and inode are allocated together in a `socket_alloc` structure, and `new_inode_pseudo` returns a pointer to the inode in that structure. `container_of` in that case returns a pointer to the `sock_alloc` structure, which `SOCKET_I` then returns a pointer to the socket element of by using the `->` operator and taking the address of the result. `sock_alloc()` then sets some inode fields and returns the socket.

It's important to note that at this point in time, the socket is still not fully created. `sock_alloc()` has merely ensured that the memory necessary for the structures has been claimed, and taken care of the inode initialization in the vfs layer. `sock_alloc_inode` specifically sets `socket.ops = NULL`, `socket.sk = NULL`, and `socket.file = NULL`, and the instantiation is not yet complete.

The next line of interest in `__socket_create` is

`pf = rcu_dereference(net_families[family]).`

RCU stands for Read-Copy-Update, which is a mechanism in the kernel that allows lockless concurrency to work without race conditions. A detailed description of RCU is omitted here for brevity. `pf` is declared as `const struct net_proto_family *pf`, where `net_proto_family` is a simple structure whose main element of interest is a function pointer

```
int (*create)(struct net *net, struct socket *sock, int protocol, int kern)
```

`net_families` is populated by `sock_register`, which is called by each socket family that registers itself in the kernel. In the case of IPv4, it's called in `linux/net/ipv4/af_inet.c` which sets the function pointer in `inet_family_ops.create = inet_create`. `pf->create` is then called. If `pf->create` is successful, then success is propogated back up the stack, and `sock_create` returns success. `sock_map_fd` is then called on the socket, which calls `sock_alloc_file`, which obtains an unused file descriptor for the socket in sockfs, sets `sock.file` to the `struct file` (defined in `linux/include/linux/fs.h`) and calls `fd_install(fd, newfile)` where `struct file *newfile` was obtained from `sock_alloc_file`. `fd_install` calls `__fd_install(current->files, fd, newfile)` which is defined in `linux/fs/file.c`. `__fd_install` takes the file descriptor table `struct fdtable *fdt = current->files->fdt` and then assigns via RCU `fdt->fd[fd] = newfile`. `current` is a macro commonly used in the Linux kernel to return the `struct task_struct` of the current active userspace process (the process that called the syscall). This struct (defined in `linux/include/linux/sched.h`) contains a wealth of information about the process such as the pid, open files, a pointer to its stack, and many others. the file descriptor `fd` returned to userspace momentarily is simply an index into a `struct fdtable` array which is unique to each process. Therefore, fds are not globally unique but are rather per-process. Thus, when a process gives a fd to a system call, the kernel only looks into that processes fdtable

to determine whether or not it is a valid fd. After the file descriptor has been installed in the fdtable for the current process, and is correctly mapped to the actual struct file that was allocated for it, the file descriptor is returned to userspace.

A.1.2 Network Layer

Sockets make little sense without talking about networking, and in order to understand the final parts of socket creation in more detail it is necessary to delve into the network layer. The implementation here will be different depending on what network protocol is in use, but for this discussion we will stick with IPv4 due to it's popularity.

since `pf->create = inet_create`, `__sock_create` calls `inet_create` from `linux/net/ipv4/af_inet.c`. But, how does `af_inet.c` mange to set that function pointer? `inet_init()` is called using the same initcall method as `sock_init()` is called with, but registers itself using the `fs_initcall` macro instead of `core_initcall`, which is a level 5 initcall and thus is called later in the boot cycle.

`inet_init()` does many things, including registering TCP, UDP, RAW, and ping protocols, which are part of the kernel transport layer and will be covered later. `sock_register(&inet_family_ops)` follows immediately, which registers IPv4 with the kernel socket interface and is where the `inet_create` function is assigned as the creation function for this protocol. `inet_init` also populates a list `inetsw` from the `inetsw_array`. This contains operations that `inet_create` will need depending on the protocol type specified. In our case, `type` is `SOCK_DGRAM` and `protocol` is `IPPROTO_UDP` and so we end up with `.prot = &udp_prot` and `.ops = &inet_dgram_ops`, whose purpose will become clear momentarily.

`inet_init()` then calls `arp_init()`, `ip_init()`, `tcp_v4_init()`, `tcp_init()`, `udp_init()`, `udplite4_register()`, `ping_init()`, `icmp_init()`, `ipv4_proc_init()`, `ipfrag_init()`, `dev_add_pack(&ip_packet_type)`, and `ip_tunnel_core_init()`. I will cover many of these later, to avoid getting sidetracked from the main topic of discussion at the moment which is finishing up the instantiation of a socket structure.

`inet_create` loops over the `inetsw` list by using a macro `list_for_each_entry_rcu`. This is a common construct in the Linux kernel, but it can be confusing at first glance, and so I will cover it in detail here. The exact call is:

```
static struct list_head inetsw[SOCK_MAX];
...
struct inet_protosw *answer;
list_for_each_entry_rcu(answer, &inetsw[socket->type], list) {
    ...
}
```

With `inet_protosw` defined in `linux/include/net/protocol.h`.

The way linked lists work in the Linux kernel (actually a doubly linked list) is that instead of the list itself containing pointers to structures of interest, the structures themselves hold pointers to the nodes of the list. Looping over the linked list then does not involve looping over nodes pointing to list structures, but instead involves directly jumping to each structure in the list. This is an optimization in two ways. First, it saves a pointers worth of memory for each element in the list. Secondly, and more importantly, it only requires following one pointer per node. This is a huge cache optimization, because going to the next node in the list automatically loads the surrounding memory into CPU cache. With the Linux implementation of linked lists, the data that was just loaded into the cache is the structure that we are interested in, and all it takes is some pointer magic using the `container_of` macro covered earlier to get a pointer to the parent struct of the element.

In this case, each `inet_protosw` struct contains a `struct list_head list`. `struct list_head` in turn contains a `*next` and `*prev` pointer to more `struct list_head` objects. The pointers point to the next and previous list member (hence the 3rd argument to `list_for_each_entry_rcu` being `list`, the member name), and the address of the member is simply `offsetof(struct inet_protosw, list)` bytes ahead of the address of the beginning of the object. Pointer arithmetic computed in `container_of` returns the base addresss, and `list_for_each_entry_rcu` returns that address to `answer`. The rest of the macro is setup so that it can be used in the same manner as a for loop.

The body of the loop is checking for protocol used with this socket type, and assigns the first match. This is required because it is possible to specifiy `IPPROTO_IP` (the wildcard protocol) and it is then up to kernel to pick a default real protocol. Since `IPPROTO_UDP` comes before `IPPROTO_ICMP` in `inetsw_array`, the default for `SOCK_DGRAM` will be UDP.

`inet_create` then sets the `ops` field of our socket to `inet_dgram_ops` (`answer->ops`) and proceeds to allocate a `sock` structure. This is somewhat confusing, but it's a convention in the kernel networking code to refer to `struct socket` as `sock` and `struct sock` as `sk`. The `sock` structure is allocated with `sk_alloc`, which calls `sk_prot_alloc` to do the actual allocation. `sk_prot_alloc` uses `kmem_cache_alloc` to fetch an unused `sock` from the slab created for this particular protocol. The slab cache is created when `proto_register` is called in `inet_init`. Once `sk_prot_alloc` retrieves memory for the `sock`, either through the slab or directly with `kmalloc`, `sk_alloc` sets the family, protocol, and other fields not necessary to understand here.

When `proto_register` is called for the UDP case, the call is `proto_register(&udp_prot, 1)`. `struct proto udp_prot` is defined in `linux/net/ipv4/udp.c` which contains mostly a collection of udp-specific function pointers to various socket operations. The important element of note

here is that `udp_prot.obj_size = sizeof(struct udp_sock)`. `struct udp_sock` is defined in `linux/include/linux/udp.h` and contains a `struct inet_sock` and some other fields that aren't important right now. `struct inet_sock` is defined in `linux/include/net/inet_sock.h` and contains a `struct sock`, and inet information such as source and destination ports and addresses, tos, TTL, etc. Therefore: `sizeof(struct udp_sock) > sizeof(struct inet_sock) > sizeof(struct sock) > sizeof(struct socket)`. This is important to remember for memory allocation purposes, and is helpful to illustrate which structs contain which other structs as members. `struct socket` contains a pointer to `struct sock`, which in this case is contained within an `inet_sock`, which in this case is contained within a `udp_sock`. When the slab allocator is allocating the `struct sock` in `sk_prot_alloc`, it requires `prot->obj_size` bytes of memory. Therefore, even though we are allocating a `struct sock` here, we are actually allocating enough memory for the entire `struct udp_sock`. It is easiest to think of this as object oriented programming. If the kernel had been written in C++, `class udp_sock` would inherit from `class inet_sock`, which would inherit from `class sock`. When allocating a `udp_sock` object, enough memory would be allocated for `udp_sock` and all of its parent classes. With that understanding, the next line of `inet_create` makes sense.

```
struct inet_sock *inet = inet_sk(sk)
```

This takes the `struct sock *sk` and casts it to `struct inet_sock *`, which as we just explained is possible because `sk` points to enough memory for itself, `inet_sock`, and `udp_sock`. Since the first element of `inet_sock` is `struct sock`, this works as expected.

`sock_init_data(sock, sk)` is then called, which sets `sock->sk = sk`, and initializes many of the sk fields. Finally, `inet_create` will call `sk->sk_prot->init(sk)` if it exists. In our case, `struct proto udp_prot` does not define a function pointer for init, and so by the C standard it will be initialized to `NULL`.

Our socket structure is now initialized. `inet_create (pf->create)` returns, `__sock_create` returns, `sock_create` returns, and control returns to the socket syscall which calls `sock_map_fd` as described above. If no errors are encountered, the systemcall returns the file descriptor from `sock_map_fd` to userspace and program control reverts to the caller process.

A.2 Setsockopt System Call

Now that we have an established socket, before calling connect netcat sets some parameters of our socket using `setsockopt`. The syscall is defined in `linux/net/socket.c` as

```
int setsockopt(int fd, int level, int optname, char *optval, int optlen)
```

In our case, level = `SOL_SOCKET`, optname = `SO_LINGER` and `SO_REUSERADDR`, and optval = [`onoff=1`, `linger=0`] and [`1`]. fd and optlen are set according to the file desriptor returned from socket and the length of optval.

A.2.1 Implementation Details

`setsockopt` first needs to look up the struct socket that is associated with the fd that was passed to it. To do this, it uses `sockfd_lookup_light(fd, &err, &fput_needed)` where `err` and `fput_needed` are int. `sockfd_lookup_light` is defined in `linux/net/socket.c` and calls `fdget(fd)` to get a `struct fd` from the integer file descriptor. `fdget` returns `__to_fd(__fdget(fd))` which is inlined as

```
return (struct fd){(struct file *)(fd & ~3), fd & 3}
```

Do not be alarmed at the bitwise operations, they will be explained shortly. `__fdget` is defined in `linux/fs/file.c` as return `__fget_light(fd, FMODE_PATH)` and `__fget_light` (also defined in `file.c`) performs the actual work. `__fget_light` gets `struct files_struct *files = current->files` and then attempts to get the `struct file` associated with the passed fd. This is done by calling

`__fcheck_files(files, fd)` which will attempt to get the struct file via RCU `files->fdt->fd[fd]` if the mask (`FMODE_PATH` in this case) works `(file->f_mode & mask)` and we're able to obtain a lock on the file with `get_file_rcu`. `get_file_rcu` is a macro defined as `atomic_long_inc_not_zero(&(x)->f_count)` which detects atomically if the reference count for the file object is zero, or goes to zero during an attempt to increment it. If it is able to obtain a reference count, then it returns the structure. Before returning however, `__fget_light` bitwise ORs the `FDPUT_FPUT` flag onto the returned pointer. This flag is defined to be 1, and if needed, signals to functions up the call chain that they will need to `fput` the file. `fput` is part of the reference counting. When we grabbed the file structure, the reference count gets incremented. Later, we decrement it when releasing it, and it can be freed if we held the last reference to it. Since this flag is encoded into the low bits of the pointer to the struct file, and those bits will always otherwise be 0 because of memory alignment, the bitwise operations in `__to_fd` seperate the flags from the pointer and return the pointer to its original state before inserting it into `struct fd`. This is an optimization that saves a pointer dereference.

Now that `sockfd_lookup_light` has a `struct fd`, it is able to call `sock_from_file` with the file object. `sock_from_file` checks if `file->f_op == &socket_file_ops`. This is set in `socket.c` by `sock_alloc_file` when calling `alloc_file` (which is ultimately called by the socket syscall with `sock_map_fd`). If it is, it then returns `file->private_data` which also gets set in `sock_alloc_file` as a pointer to the socket structure passed to it.

With the socket structure available, the level passed to the syscall is checked. `SOL_SOCKET` means that `sock_setsockopt` is called, otherwise `sock->ops->setsockopt` is called. The difference is that `sock->ops->setsockopt` will call whichever function `.setsockopt` points to in the `struct proto_ops` that was specified when the socket was created. In our case, it's specified as `inet_dgram_ops` in `af_inet.c`

since this is an IPv4 socket, which sets `.setsockopt = sock_common_setsockopt`. In this case however, `setsockopt` is called with level set to `SOL_SOCKET`, and so we call `sock_setsockopt`.

`sock_setsockopt` is defined in `linux/core/sock.c` and is designed to be generic and apply to all sockets, not just those with a particular family. It gets the `struct sock *sk = sock->sk reference`, and then checks the option passed to it in a switch statement. The first time we call this function, we're passing it `SO_LINGER`, with onoff=1 and linger=0. This corresponds to setting `sk->sk_lingertime` to `0 * HZ` where `HZ` is an internal kernel timer frequency determined at compile time based on the configuration options set, and setting the `SOCK_LINGER` flag in sk. This gets checked in `inet_release` which is called by `sock_release` when `close()` gets called on a file descriptor belonging to a socket. In the case of UDP, timeout is ignored, and the socket is closed immediately. This is because UDP is connectionless, so there is no need to wait for packets that may be in transit but not arrived yet, as UDP makes no guarantees about the delivery of packets.

The next option is `SO_REUSEADDR`, which is set to 1. This correspons to `sk->sk_reuse = SK_CAN_REUSE` which is a flag checked during socket binding, which will be discussed later.

A.2.2 Connect System Call

Connect is defined in `socket.c` as

```
int connect(int fd, struct sockaddr *uservaddr, int addrlen)
```

Even though UDP is a connectionless protocol, it still uses the connect syscall. The way TCP and UDP use the word connection is different from what is meant by it in this context. In a socket model, a connection can be thought of as a pipe between two sockets. It can also be thought of as a globally unique five-tuple (protocol, source address, source port, destination

address, destination port). UDP refers to itself as connectionless on a higher level. It does not have the TCP connection handshake and concept of connection state. When talking about connections here, we refer to the socket model of a connection rather than the TCP model unless otherwise specified.

There is a new structure here that we haven't talked about before, `struct sockaddr`. It contains an address family `sa_family_t sa_family` and `char sa_data[14]`. `sa_family_t` is a `typedef` of `__kernel_sa_family_t` which in turn is a `typedef` of `unsigned short`. These definitions can be found in `linux/include/linux/socket.h` and `linux/include/uapi/linux/socket.h`. This is a generic socket address structure that's used to store a protocol number and a single address. A historical note: since sockets are a generic interface that can be used for more than just internet communication, the potential address size is larger than it would need to be if it were only storing an IPv4 address. The socket interface also pre-dates the common IP internet, for example X.25 networks use a 14 character binary coded decimal addressing system, and the socket interface would still support it today. `struct sockaddr_storage` is a struct large enough to hold any kind of specialization of `sockaddr`. For example, `sockaddr_in6` for IPv6 inet sockets are 16 bytes and would not fit in a `struct sockaddr`. Thus, `struct sockaddr_storage` was created. Currently, it is defined to contain 2 bytes of family and 126 bytes of data. This should be large enough for the foresable future.

connect uses the familiar `sockfd_lookup_light` function to retrieve the socket from the fd, and a `move_addr_to_kernel` function to copy the sockaddr from userspace into a `sockaddr_storage` in kernel space. It then calls the `sock->ops->connect` function pointer which was defined previously by `inet_create` during socket creation. `sock->ops` points to `inet_dgram_ops`, and `inet_dgram_ops.connect = inet_dgram_connect`, as defined in `linux/net/ipv4/af_inet.c`.

`inet_dgram_connect` grabs `struct sock *sk = sock->sk` and after some checks calls `sk->sk_prot->connect`. `sk_prot` was defined previously by `sk_alloc` to be

`struct proto udp_prot`. Remember, `sk_alloc` is called by `inet_create` which determines the protocol type by `inetsw[sock->type]`, which in this case was `SOCK_DGRAM`. `inetsw[SOCK_DGRAM].prot = &udp_prot` as defined in `af_inet.c`, and `struct proto udp_prot` is defined in `linux/net/ipv4/udp.c`. `udp_prot` defines the connect function pointer to be `ip4_datagram_connect` which in turn is defined in `linux/net/ipv4/datagram.c`. `ip4_datagram_connect` obtains a lock on the `struct sock *sk` passed to it, then calls `__ip4_datagram_connect(sk, uaddr, addr_len)`. The call chain is thus:

```
net/socket.c: connect(): sock->ops->connect() ->
net/ipv4/af_inet.c: inet_dgram_connect(): sk->sk_prot->connect() ->
net/ipv4/datagram.c: ip4_datagram_connect() -> __ip4_datagram_connect()
```

`__ip4_datagram_connect` performs the actual work.

A.2.3 Transport Layer

We now enter the kernel network transport layer, which is responsible for the concept of connections and routing of packets. `inet_dgram_connect` first needs to bind the socket before proceeding to connect. In this context, binding means setting the source address for the socket since there may be many interfaces, each with many IP addresses. It first checks to make sure that `inet_sk(sk)->inet_num` is 0 and then calls `inet_autobind(sk)` to perform the actual binding. `inet_num` will be 0 in the case of unbound UDP as it is set in `inet_create` when `SOCK_RAW` is used, not `SOCK_DGRAM`. `inet_autobind` calls `sk->sk_prot->get_port` which is the function pointer `udp_prot.get_port` which maps to `udp_v4_get_port`. `udp_v4_get_port` computes two hashes, a nulladdr hash and a partial hash. The hash is a function of the source address, source port, and network namespace. For the purposes of our discussion here, we can ignore the namespace parameter and assume it is a constant (as it is unless you have explicitly created namespaces). In this case, the two hashes should be equal since we have not explicitly bound the socket to any source

address yet, and the passed port is 0 which will indicate that any available port can be used. We then pass our sock, source port (0), the null address hash, and a comparison function (`ipv4_rcv_saddr_equal`) to `udp_lib_get_port`.

`udp_lib_get_port` looks up an available port for us to use. To understand how it accomplishes this, we have to detour into an explanation of udp hashtables in the Linux kernel.

```
struct udp_table *udptable = sk->sk_prot->h.udp_table
```

`sk_prot` as explained earlier points to `struct proto udp_prot`, defined in `net/ipv4/udp.c`, but declared in `include/net/sock.h`. In `udp.c` we see `h.udp_table = &udp_table`, where `udp_table` is declared, defined and exported all within `udp.c`. The struct element `h` is declared in `sock.h` as a union of hashtables of different types, in this case we're using the udp hash table. This is a global table that all UDP connections will refer to. It is initialized in `udp_table_init` which is called by `udp_init` which is called by `inet_init`, which finally is called via the initcall system explained earlier.

`udp_table_init` calls `alloc_large_system_hash`, passing it several parameters. One parameter that at first appears to have not been set is `uhash_entries`. `uhash_entries` is set though, it is called during the kernel boot sequence by the macro `__setup("uhash_entries=", set_uhash_entries)`. This sets the function `set_uhash_entries` to be called during bootup, optionally passed a pointer to a string if `uhash_entries=` was set on the kernel boot command line. In absence of this string, it defaults to 0.

`alloc_large_system_hash` is defined in `mm/page_alloc.c`, and from there we can see that `udp_table_init` is defining a hash table that contains 2 `udp_hslot` structs per bucket, 0 buckets (though this will be overridden by a lower limit parameter), with a minimum of 256 buckets (unless overridden by a compile option for lower memory usage, in which case it uses 128 buckets as a minimum) and a maximum of 64*1024 buckets. Within each bucket, a list is initialized using `INIT_HLIST_NULLS_HEAD`. `hlist_nulls_head` is a different kind

of list than was used earlier for `inet_create`. `hlist_nulls_head` uses a non-null terminating sentinel instead of a null. These lists make use of the fact that all nodes of the list will be aligned to 4 or 8 byte boundaries, meaning that the last bit of their address will always be 0. Therefore, the null node in these lists has the last bit of its address set to 1. This frees up the other 31 or 63 bits to be used by the programmer to store identifying information about the list. Why would you want to store identifying information in a list sentinel? Since these lists are updated using the lockless RCU mechanism in the kernel, there is a potential race condition where one CPU could free and reallocate a node while another CPU is travsering that list. In this case, the traversing CPU would then begin traversing a different list without realizing it. An identifier in the sentinel node allows that CPU to realize this has occured and search the list again [36]. This should happen infrequently enough that it is a net performance increase to use RCU intead of locks here.

Back to `udp_lib_get_port`, since we passed it 0 for snum, we are asking the function to find any port for us to use. `DECLARE_BITMAP` is defined in `include/linux/types.h` and declares an array of unsigned longs, the size of which is sufficient to store as many bits as passed in the second parameter. In this case, the second parameter `PORTS_PER_CHAIN` is `65536 / (CONFIG_BASE_SMALL ? 128 : 256)`. Normally, `CONFIG_BASE_SMALL` is false, and so in normal circumstances we need 256 bits, corresponding to 8 unsigned longs on x86_64 systems. `udp_lib_get_port` then picks a random port in the range allowed by `inet_get_local_port_range`. This range is set in `af_inet.c` in the `inet_init_net` function, defaulting to [32768, 60999).

`last = first + udptable->mask + 1` sets `last = first + 256` in the default case. `udptable->mask` is set in `alloc_large_system_hash` as `log2(numentries) - 1`. This is setting up a loop to loop over all of the buckets in the hashtable. The hash function is constructed such that `log2(numentries)` consecutive integers will hash to different values. For each hash bucket, `udp_lib_get_port` will

check every possible port number in that bucket for whether or not it is in use.

`udp_lib_lport_inuse` sets all of the bits in bitmap that correspond to the ports in use by that hash slot. Then, for each snum in the hash slot, iterating over the range in a random order (for security reasons, you want ephemeral source ports to be unpredictable), the inner loop checks to see if the bit is set. If the bit is not set, the port is within the allowed range, and not reserved, we select this port for use. After finding an available port we then set `inet_sk(sk)->inet_num = snum`, `udp_sk(sk)->udp_port_hash = snum`, and add this port to `udp_table` so that nobody else can use it while we have it bound.

Assuming a port was found, successs propogates up and the stack unwinds back to `inet_autobind`, which then sets `inet->inet_sport = htons(inet->inet_num)` and returns success in turn. `inet_dgram_connect` then proceeds to call `ip4_datagram_connect` via the function pointer `sk->sk_prot->connect`, which locks the `sk` before calling `__ip4_datagram_connect`, which performs the actual work of establishing a connection.

`__ip4_datagram_connect` calls `sk_dst_reset(sk)` which clears `sk`'s transmit queue buffer via `sk_tx_queue_clear` and then swaps the passed `struct dst_entry` for the old `dst_entry`, freeing the old entry before returning. In this case, the passed pointer is `NULL`. `struct dst_entry` is used as part of the kernel transport layer to store the routing path of the socket and will be explained in detail later.

`ip_route_connect` is called to get a `struct rtable`. `rtable` is used to determine which route a connection uses to send packets, with the majority of the information being stored in a `struct dst_entry` within `rtable`. At this point in time, `oif` is 0, so the first condition is skipped. `dst` is non-zero, but `src` is 0, so `__ip_route_output_key` is called. `__ip_route_output_key` calls

__ip_route_output_key_hash, which is defined in net/ipv4/route.c, which resolves a route for us to use. Since we don't have a source address, and are not bound to an interface yet at this point, and do have a destination, we end up skipping over the first 3 conditions and call fib_lookup.

fib_lookup calls fib_get_table, asking for RT_TABLE_MAIN, which is the default routing table in Linux. The routing tables can be viewed from userspace by using the ip utility: ip rule list shows all current routing tables and their priority in the table lookup system. ip route show table main will show the contents in RT_TABLE_MAIN. fib_get_table then calls fib_table_lookup, which is defined in net/ipv4/fib_trie.c. This system is rather complicated, but the end result is that the kernel figures out which device and IP address on that device should be used as the source for this connection, and in turn sets oif and saddr respectively.

Back to __ip4_datagram_connect, once ip_route_connect returns after figuring out the source device and IP address to use for this connection, __ip4_datagram_connect finishes assigning the source address to the relevent locations in inet->inet_saddr, inet->inet_rcv_saddr, sets the destination address and port in inet, and sets the sock state to established. No packets have been sent in this connection yet, but since this is only a UDP connection, no packets need to be sent in order to declare a connection as established since there is no handshake. Success propogates back up the call stack and the connect syscall returns.

A.3 Write System Call

The write syscall is defined in fs/read_write.c. As explained earlier, the networking subsystem uses a VFS layer to expose network functionality to plain file system call operations. write calls fdget_pos to obtain a struct fd from the int fd that was passed, sets the write position to the seek position

set on the fd, and then calls `vfs_write`. `vfs_write` checks permissions and that we have passed a valid user space buffer to write from. `rw_verify_area` ensures that we are writing to something that makes sense, i.e. a positive number of bytes to a valid, positive offset in the file. `file_start_write` then waits to obtain a semaphore for this inode that allows us to actually perform the I/O. `__vfs_write` is then called to perform the actual write by calling `file->f_op->write`, which was set earlier in `sock_alloc_file (socket.c)`. `f_op` points to `struct file_operations socket_file_ops`, and write in turn is `NULL`. This in turn causes `__vfs_write` to instead use `file->f_op->write_iter`, which points to `sock_write_iter`. `fsnotify_modify` notifies anyone who has registered to be notified if this file has changed that it has now changed, and some file accounting statistics are updated. `file_end_write` then releases the semaphore, and `vfs_write` returns. The file position pointer is then updated by `write`, and `write` returns.

Back to the networking subsystem, we now examine `sock_write_iter` which is defined in `net/socket.c`. It obtains the socket from `file->private_data`, which was set earlier in `sock_alloc_file` to point to the struct socket, and then calls `sock_sendmsg` to write the buffer to the socket. `sock_sendmsg` in turn calls `sock_sendmsg_nosec`, which in turn calls `sock->ops->sendmsg`. Recall that `inet_create sets sock->ops = inet_dgram_ops`, and that struct sets sendmsg to point to `inet_sendmsg` defined in `net/ipv4/af_inet.c`. `inet_sendmsg` ensures that the socket is bound if it isn't already, though in our case it is because we called connect explicitly, and then calls `sk->sk_prot->sendmsg`. Recall that `sk_prot` was defined by `sk_alloc` to be `struct udp_prot`, which in turn defines `sendmsg` to be `udp_sendmsg` in `net/ipv4/udp.c`.

`udp_sendmsg` first checks to see if there are any pending ipv4 frames in this socket. If there is, it appends the current data to the queued frames. This happens because Linux has the option to cork data in socket buffers. To cork means literally to stop from flowing, and in the context of sockets in Linux it

means to queue data up into a buffer so that as much data as possible is sent in one frame. This prevents an application that calls write very often with small amounts of data from generating an unreasonable amount of packets. Corking can be disabled or enabled depending on the flags passed to sockets, and if disabled then there will be no pending frames. We assume the no corking case in the discussion of the code here.

`udp_sendmsg` then adds `sizeof(struct udphdr)` to ulen, to account for the size of the udp header in the packet, and checks to see if we have a `msg->msg_name` set. Since `sock_write_iter` did not set it in the `struct msghdr` that we were passed, we instead find our destination address and port from the `inet_sock`. After several more branches that will be skipped over in our case, we come across a call to `sk_dst_check(sk, 0)`. This ends up calling `ipv4_dst_check` in our case, as `ipv4_dst_ops` defines it in `ipv4/route.c`. For IPv4, this just checks to see if the route has expired or has otherwise been invalidated, in which case it returns `NULL`. Otherwise, and in our case, it returns a `struct rtable` pointer and therefore we skip the next check that sets up the `struct rtable` if it doesn't exist.

The next notable code path for us is the check for `if (!corkreq)` and the call to `ip_make_skb`, which is defined in `net/ipv4/ip_output.c`. This function takes all outstanding IP fragments for this socket and combines them into one `struct sk_buff`. `struct sk_buff` is the socket buffer structure defined in `include/linux/skbuff.h` and is a linked list of buffers that contain actual udp packets. This buffer is passed to `udp_send_skb`, defined in `net/ipv4/udp.c`. `udp_send_skb` calculates the udp checksum and writes it into the header, and then passes it on to `ip_send_skb -> ip_local_out`.

`ip_local_out` is for outgoing packets that originate from this machine (as opposed to being forwarded from somewhere else). It first calls `__ip_local_out` which sets the ethernet protocol type to IPv4, calculates the IP checksum and

length, and then enters netfilter with the hook NF_INET_LOCAL_OUT. A discussion of netfilter (the Linux ip filtering subsystem) is beyond the scope covered here, but if netfilter allows this packet to pass, then nf_hook returns 1, and so does __ip_local_out. In that case, err = 1 (which is not actually an error, despite the unhelpful name), and the packet is then forwarded to dst_output, which is a wrapper function that calls skb_dst(skb)->output. This is set for IPv4 in net/ipv4/route.c in rt_dst_alloc as rt->dst.output = ip_output. ip_output finds the physical device that this packet is set to egress from, sets the device and protocol in the socket buffer, and enters netfilter again, this time with the INET_POST_ROUTING hook. The macro NF_HOOK_COND will call ip_finish_output if netfilter allows the packet to pass. If the length of the socket buffer is larger than our MTU, we first detour to ip_fragment to fragment our packet at the IP level. In our case, the length is within the MTU size and ip_finish_output2 is called. ip_finish_output2 checks to see if we know the physical address (ethernet MAC) of the next hop for this packet, and if not, calls __neigh_create to populate the ARP table. After the MAC address is found, dst_neigh_output is called, which calls dev_queue_xmit, in turn calling __dev_queue_xmit defined in net/core/dev.c.

__dev_queue_xmit interfaces with the driver for your NIC to copy the socket buffer into the appropriate hardware buffers, and from our point of view the packet is now queued by the hardware and will be sent.

A.4 Read Syscall

Now that we've taken a deep dive into networking internals for what happens in the simplest of cases to open a UDP connection and send a packet, we will take a more abbreviated look at how the other direction works: receiving a packet.

A.4.1 NAPI - New API

Unlike `write`, the logical place to begin examination of `read` is at the driver level, since packets originate from the hardware instead of from userspace in the `read` case. The simplest conceptual driver model for a network card is to trigger an IRQ when a packet arrives on the PHY. This is, however, a terrible design from a performance perspective because of the overhead generated by each IRQ, and the fact that a busy network interface can easily be receiving hundreds of thousands of packets per second. NAPI was devised as a way for drivers to partially disable interrupts during periods of high network activity [37], and instead use polling. The driver that I'm going to be using here as an example is the intel e1000 driver found in `drivers/net/ethernet/intel/e1000/`.

The initialization function for e1000 is `e1000_probe`, and what we're looking for in here is

```
netdev = alloc_etherdev(sizeof(struct e1000_adapter))
```

This line allocates the `struct net_device` which is a generic network device struct used by all network device drivers. The function also registers the device with the napi subsystem with `netif_napi_add()`, passing in the function `e1000_clean` as the polling function to use. The polling function cleans the tx and rx buffers and if not enough work was done, reverts to using interrupts. The functions that clean the buffers, in turn, re-enable polling if they're doing a lot of work.

The function of interest here is `e1000_clean_rx_irq`. It fetches data from the hardware, and of particular interest constructs a skb using `e1000_alloc_rx_skb`. `e1000_alloc_rx_skb` calls `napi_alloc_skb` which in turn calls `__napi_alloc_skb`. This function largely just reserves memory for the skb, but it also sets `skb->dev = napi->dev`, and sets up the linked list. No other skb fields are set here. After filling in the skb with data from the hardware and adjusting any checksums if necessary (for e.g. hardware checksum offloading), the skb is passed

to `e1000_receive_skb`, which sets the `skb->protocol` field, and passes it off to `napi_gro_receive`. `napi_gro_receive` passes skb to `napi_skb_finish`, which in turn in the normal case passes it to `netif_receive_skb_internal`.
`netif_receive_skb_internal` checks to see if RPS (receive packet steering) is being used, and if it is then it enqueues the skb to a cpu-specific backlog for processing. Here I will assume that RPS is disabled, in which case the skb is sent to `__netif_receive_skb` and then to `__netif_receive_skb_core`. This function sets the interface field in skb, removes vlan tags if present, and delivers the skb to any network taps present `(&ptype_all)`.

`ptype_all` is a `struct list_head` contained in `struct net_device` and is used as a list of `struct packet_type` that defines handlers for incoming packets of a particular type. Any handlers that are registered in `ptype_all` are passed all packets. Handlers are registered using `dev_add_pack` in core/dev.c, which gets the list to put the `packet_type` into by calling `ptype_head`. If the type is set to `ETH_P_ALL` then this `packet_type` gets put in the `ptype_all` list, otherwise it's put in a `ptype_base` hash table that's indexed by the type.

If netfilter ingress filtering is enabled (which it typically is), then we pass the skb to `nf_ingress` to see if we're allowed to proceed, and if not, `goto out`. Finally, `deliver_ptype_list_skb` is called to deliver the packet to a handler specific to its type, using the `ptype_base` hash table. The `struct packet_type` for IPv4 has its receive handler set as `ip_rcv` in `net/ipv4/af_inet.c`, which is defined in `net/ipv4/ip_input.c`. `ip_rcv` validates the packet and passes it to netfilter's `NF_INET_PRE_ROUTING` hook, indicating to it to call `ip_rcv_finish` if the packet is allowed to pass. `ip_rcv_finish` calls `ipprot->early_demux`, where `ipprot = inet_protos[protocol]`. In the case of UDP, this is set as `early_demux = udp_v4_early_demux` in `struct net_protocol udp_protocol` in `net/ipv4/af_inet.c` with `inet_add_protocol(&udp_protocol, IPPROTO_UDP)`. If the `skb->pkt_type` is set to `PACKET_HOST`, then we call `__udp4_lib_demux_lookup`. Note that the `pkt_type` is not explicitly set to `PACKET_HOST` yet, however since it is defined to be 0, and

__build_skb (part of __napi_alloc_skb) explicitly memsets the skb to zero, by default all incoming packets are marked as type PACKET_HOST, unless marked otherwise. Early demuxing is a performance optimisation that can be disabled, and may not always return a socket. I will assume here that a socket was not found, in which case, the packet first has to pass through the routing layer.

If early demux failed to return a valid destination for the packet, then ip_route_input_noref is called, which in turn for non-multicast packets calls ip_route_input_slow. ip_route_input_slow calls fib_lookup, and if it is determined that the destination type is RTN_LOCAL, then rt_dst_alloc is called which sets rt->dst.input = ip_local_deliver. This function then returns with a valid dst_entry, and ip_rcv_finish calls ip_local_deliver with a call to dst_input(skb).

ip_local_deliver calls netfilter with the hook NF_INET_LOCAL_IN, passing it ip_local_deliver_finish to call if the packet is allowed in, which will lookup the protocol in inet_protos, and call ipprot->handler(skb), which for UDP is set to udp_rcv which calls __udp4_lib_rcv. The main function of interest here is obtaining the socket for this packet, which is done through __udp4_lib_lookup_skb, passing it the udp_table hash table where the sockets are kept. Once the socket is found, the skb is enqueued into the sockets receive queue using udp_queue_rcv_skb, and the function returns. The packet will sit in queue until it is eventually returned to userspace with a call to read() on the socketfd, through the same vfs layer as write().

Appendix B

Expanded Literature Review

This paper [3] has been included as an expanded literature review to provide additional references to the main body of work. We originally published it in the 2016 Conference on Security and Management.

Anomaly Detection and Machine Learning Methods for Network Intrusion Detection: an Industrially Focused Literature Review

Colin Gilmore and Jason Haydaman
TRTech
100-135 Innovation Drive, Winnipeg, Canada.
colin.gilmore@trtech.ca, jason.haydaman@trtech.ca

SAM Track: Security Algorithms

***Abstract*—This paper outlines a literature review undertaken towards the goal of creating an industrial viable (real world) anomaly detection/machine learning based network intrusion detection system. We develop a taxonomy of available methods, and outline the pros and cons of each. This review leads to several important conclusions: (1) There are a large number of algorithms in the literature with significant level of overlap; (2) given the state of the literature today, it is not possible to objectively select the best algorithm; (3) there is a lack of research on the feature selection process needed for machine learning approaches; and (4) the low base-rate of attacks on computer networks compared with benign traffic means that effective detection systems will consist of many detection algorithms working simultaneously.**

Keywords— Network Intrusion Detection, Machine Learning, Anomaly Detection

I. Introduction

The amount of data stored on personal, industrial, and government computer networks is constantly growing. This creates a large incentive for other actors to attempt to illegitimately access these data. The economic cost of these attacks is notoriously difficult to quantify, however in 2011 the British Office of Cyber Security and Information Assurance estimated that cyber-crime cost the United Kingdom £27 billion per year, of which £21 billion was lost to espionage and intellectual property theft. The Canadian Cyber Security Strategy estimates identity theft losses (which, for the UK data represent only 6.3% of the total) costs Canadians $1.9 billion each year. In addition to the direct economic costs, an undetected network attack can have affects that go beyond economic, including the loss of confidence in a major government department or program.

The goal of our research team is to use anomaly detection and machine learning type approaches to improve the state-of-the-art for network intrusion detection. Our approach is an industrial one: we seek only to implement operationally viable algorithms (i.e. in operation in the real world). Thus, our approach to the problem is different than past reviews [3, 10, 16, 17, 18, 19, 20, and 21]. There has been significant effort in the academic literature relating to anomaly detection and data mining techniques for network intrusion detection (see [10, 17,26]). However, this has not resulted in widespread industrial deployment. Two researchers note [26] "… despite extensive academic research one finds a striking gap in terms of actual deployments of such systems: compared with other intrusion detection approaches, machine learning is rarely employed in operational "real world" settings."

Compared to other machine learning/data mining/anomaly detection applications, such as detecting credit card fraud, or recommending new purchases, these anomaly detection and machine learning techniques have not seen industrial deployment for detecting network intrusions – they have not made it into the 'open world' of real-world deployment.

This work is the first part of taking up the challenge of creating a real-world deployment for an anomaly-detection/machine learning based network intrusion system.

The first step of such a process is to undertake a literature review of the available algorithms, and this paper outlines that process.

II. Intrusion Detection Overview:

Network intrusion detection can be divided into three types: signature based, specification based, and anomaly based [6]. We outline these types in Fig. 1.

1.1 Signature Based

Signature based techniques use a 'signature' – typically a hash – associated with a particular malicious activity. The most common signature based technique is an anti-virus program, which checks the signature of all files traversing a network, or being downloaded onto a computer. If the file being checked is a known virus/Trojan/worm, etc. then an alert is triggered. Signature based techniques have the advantage that there is very rarely a false alarm, but the disadvantage that they can (by definition) only detect known attacks. Significant effort must be made for signature management and subscription. Advanced attackers can readily avoid signature based detection because they are often capable of writing their own software, or may operate by taking over legitimate accounts.

1.2 Specification Based

Specification based techniques rely on the listing of network behaviors which are considered to be malicious. An example might be a clear brute-force attempt on a publically available account (e.g. 100's of unsuccessful login attempts). An example program that (for the most part) runs a specification-based detection engine is the program SNORT [7]. Specification based techniques offer a more generalized way of detecting threats on a network, and may detect attacks not seen before, but often take a large amount of expert effort to specify. Unlike signature based detection, specification based detection can have false alarms. They can also be bypassed as these attackers will avoid obvious malicious behavior (like brute-force attempts).

1.3 Anomaly/Machine Learning Based

Anomaly based techniques rely on detecting 'abnormal' or anomalous behavior. We include in this definition the various machine learning algorithms. These techniques take inputs from numerous network features, and label these features as 'anomalous' or 'normal' output. These techniques are the hardest for an attacker to avoid, as they are so general. However, they have the disadvantage of having high-false positive rates, which can make the detector useless in practical areas (discussed in Section III).

1.4 Notes on Features and Classification

In any detection system, there exist two main issues which need to be solved: feature selection and classification.

The features are the inputs which are selected as inputs to the algorithm. Features can include things like the Internet Protocol (IP) addresses, and much more. Classification is another word for 'which algorithm is used to determine which input data comes from a malicious source'.

1.5 Network Based Vs. Host Based Intrusion Detection

Network based anomaly detection algorithms depend only on data which is collected from network devices like firewalls, routers, IPS's, etc. Host based anomaly detection systems can include programs running on individual computers, which allows for more features to be added to the anomaly detection system. It is also possible to have a combined network and host-based system. Network based systems have the advantage of simplicity – there does not need to be a program running on every individuals computer (in some networks, it may be impossible to install a host-based agent on every computer). We consider only network-based systems in this review.

III. Base rates and the problem with False Positives

Axelsson [8] gives an excellent discussion about the problem with false positives for network intrusion detection. This issue is best illustrated with an example: Imagine that we have a network intrusion detection method which is 99% accurate. Thus, if the detection technique sends an alarm, it has a 99% chance that it is a true network intrusion, and a 1% chance of a false alarm (positive). Next, assume that on a particular network, only 1 in 10,000 input features comes from a malicious source (it could be even lower than this for many networks). The exact input feature can vary from method to method, but could be, for example, a Domain Name Service (DNS) request to a command-and-control server.

Now, assume that the anomaly detection system signals that it has detected an intrusion. What is the probability that the system has actually detected a real network intrusion?

The answer comes from Bayes theorem, and is quite low: 0,98% (or about 1%). Thus, when this system triggers, 99 times out of 100, it is a false alarm. This number is known as the positive predictive value of the test. Axelsson claims that the positive predictive value of the test must be above 50%, or most human operators will completely disregard the detector [8]. This problem is due to the low rate of the base problem we are trying to detect (1 in 10,000).

Due to this base rate problem, and the fact that the vast majority of network traffic is not malicious [8], care must be taken to keep false positives to a minimum. To obtain reasonable positive predictive values, the test must have a false positive rate on the order of the event we are trying to detect. In practice, this will likely mean that we need to combine several forms of detectors to reduce false positives.

IV. Feature Selection

The selection of features is perhaps the most important part of any anomaly detection process. If the right features can be found, then there is no need for further detection processes. For example, the ideal 'feature' for network intrusion detection would be a feature that was in one state when there was an attack and another state when no attack was occurring. If this ideal feature existed, there would be no 'anomaly' style classification algorithm necessary.

To be at all useful, the selected features must vary when there is an intrusion on a computer network. If the features we use as inputs to the anomaly detection system do not vary, then the best algorithm in the world will not be able to detect the intrusion. We can imagine the worst feature in the world – in this case, the feature would not change at all when an attack occurs (it would have zero sensitivity to an attack).

In practice, of course, the features extracted from network traffic will rest between these two extremes. The features used for network intrusion will vary somewhat for both normal and malicious traffic. The art of feature extraction to find features that vary a small amount for normal traffic, and then vary more significantly when an attack occurs. If there is a large enough difference in these features, then we can detect the attack with a suitable algorithm.

Ideally, we would like to select the minimal set of features that allow us to appropriately classify network behavior into normal and malicious categories. In practice, most features are selected on an ad-hoc basis based on expert domain knowledge. Automated feature selection procedures are available, but they will rely on a large data set of labelled training data [9] which is generally not available.

Based on [10], we have shown a taxonomy of features in Fig. 2. The features are split into network traffic based features, as well as network taxonomy features. Network taxonomy refers to the structure of a network (number of hosts, network organization). Network traffic can be split into three sub-groupings: flow data, protocol analysis, and derived features.

Flow data are data which capture which two computers are talking to each other at what time. A flow datagram will have source/destination IP and port numbers as well as protocol used for the communication.

Derived Features: We use the term derived features to mean features which are not readily available from the more basic data. A good example is Principal Component Analysis [11,12] – where a large number of (typically correlated) basic input variables are processed into a (much) smaller number of uncorrelated input variables. The outputs of this technique are variables which have no easy-to-grasp relationship to the more basic inputs. Other examples include syslog information, authorization logs, etc. [13].

Protocol analysis: many of features we will use in this project fall under the protocol analysis label. Any feature which is not part of the flow data, or a 'derived' feature will be of the protocol analysis type. Examples include the browser agent used or the particulars of a DNS query.

Gonzalez [14] notes that "Identification of cyber attacks and network services is a robust field of study in the machine learning community. Less effort has been focused on understanding the domain space of real network data in identifying important features for cyber attack and network service classification." Gonzalez presents a systematic way of making a set of derived features (13-27 input features) from a much larger set of basic features (over 200). The top features set include port numbers, packet length, number of truncated packets etc. Other research in this area includes [13,57].

V. Anomaly Detection and Machine Learning Methods

The use of anomaly detection algorithms to for network intrusion detection has a long history. To the best of our knowledge, the use of anomaly detection for network intrusion detection began with Denning in 1987 [15].

There exists a large number of papers on anomaly detection: a thorough review of the experimental methods used between 2000-2008 found 276 peer-reviewed papers [2]. The huge number of papers in this research area means that we will almost certainly miss many papers. However, our approach has relied on both reading systematic reviews/taxonomies, and finding examples of each class of anomaly detection method.

Given our goal of creating a practical anomaly detection which is useful in the real world, we must be aware that many academic papers may have been published simply for the sake of being a novel approach. We are entering this process expecting that many papers will have been written for this 'novelty' reason, not because they solve the network intrusion detection problem in a better way.

In our literature search, we have come across 8 separate survey papers that relate to anomaly detection [3, 10, 16, 17, 18, 19, 20, and 21]. Most are focused on anomaly detection for network intrusion detection, and one more general, e.g. [3]. Much of this remaining section relies on the results of these surveys. We have summarized the methods discussed in this review (with their pros and cons) are outlined in Table 1.

Although we would like ideally quickly find the 'best' anomaly detection techniques for network intrusion detection, it has been shown that this will be very difficult. A review of *evaluation* techniques used for proposed anomaly detectors [2] concludes that '[anomaly detection] studies from all categories fail to follow basic principles of scientific experimentation.' That is, the 'tests' used to prove the usefulness of a particular anomaly detection technique for intrusion detection are not useful for actually evaluating the technique in an objective way. From our perspective this means it is impossible from to objectively decide which anomaly detection algorithm performs the best.

A. Anomaly Detection Categories

Garcia-Teodoro *et. al*. [17] split anomaly detection into several categories. We have created our own taxonomy, based on [10] and [17]. This is shown in Fig. 2. We note that these categories can have significant overlap. For example, some 'knowledge based' systems are also machine learning, and vice-versa. Further, almost all machine-learning and data mining approaches can be called statistical. Thus, this taxonomy is very loose at best.

Statistical Based: network traffic is captured and a profile representing its stochastic behavior is created. Two datasets are considered: current and trained (or previous, or whatever). Histograms, single variable, multiple variables.

Knowledge Based: (e.g. an expert system). Classify the audit data according to a set of rules: Finite state machine is an example. It is likely that we are coming up with an 'expert system'.

Machine Learning Based: Based on establishing an explicit or implicit model that enables the patters analyzed to be categorized. Need for labelled data in all cases. Examples: Bayesian Network, Markov Models, Neural Networks, Fuzzy Logic Techniques

B. Statistical Based Examples

Statistical based models rely on creating some type of underlying model about a particular variable such as traffic volume, or number of connections per hour. One method would be to assume a particular probability density function (e.g. Gaussian) for a variable, then calculate parameters such as mean and standard variation [15]. If these parameters are outside of some threshold, then an anomaly is triggered. For example, if the mean level of packet size goes high (or low) for some reason, this is a reason for suspicion.

Statistical techniques may also be practiced on more complicated derived features [22].

1) Single-variate, Multi-variate based

Single-variate and multi-variate models can be used for these statistical models. In the multi-variate case, correlations between multiple variables can be considered. [23, 24]

Despite all the research into mathematically more complicated detection methods, some claim that these simple techniques with thresholds often out-perform the more complicated methods. [17,25].

2) Histogram based

In most cases, one cannot assume an underlying statistical model for computer network traffic – e.g. variables do not fall into a simple Gaussian pattern. Considerer, e.g. a probability density function of the first 8 bits of an IP address (e.g. 196.xxx.yyy.zzz). [22] The probability density function will be dominated by the internal private IP addresses of the network, with large spikes located at '172', 192' and '10' (depending on how the internal network is configured).

In cases like these, a histogram-based approach is a viable solution. One can create a statistical model through histograms of the training data, and then compare the test data on the same histogram. Various metrics can then be used to generate 'normal' and 'anomalous' labels [22, 27, 28].

Histograms may also be constructed over various time-scales and for datasets with large differences in sizes. In this case, there is a significant problem of comparing histograms with different numbers of bins. This leads to a non-linear optimization problem [27,28].

C. Knowledge Based Examples

Knowledge based intrusion techniques rely on the use of a human expert to define a set of rules or process that are 'normal' or allowed. Deviations from these processes are flagged as anomalies. To a certain degree, our use of feature selection could be viewed as a knowledge- based approach to anomaly detection. We are encoding the expert knowledge in the feature selection process, rather than in the classification (anomaly detection) algorithm.

1) Finite State Machine / Markov Chain

Finite state machines (which also can be viewed as Markov Chain) are models are an abstract system that transitions from one state to another with a certain probability [29,30]. Within the concept of a knowledge-based intrusion detection system, the model can be constructed by a human expert – who knows the states which should exist. The probabilities of transitions between states can then be determined from training data [15].

If the testing data exhibit low-probability state transitions, this can be flagged as an anomaly. In the context of machine learning, the same state-based can be used, but in the machine-learning case, the states, and the probability of transitions, are created via automated processes – not human experts.

2) Expert Systems

Expert systems are systems which attempt to emulate expert human reasoning via machines [31]. While a strict application of the term expert system involves the creation of a knowledge base and an inference engine [31], we feel that the term expert system could be applied to almost any anomaly detection system, as the goal is to replace/assist a skilled human operator.

The Next Generation Intrusion Detection Expert System (NIDES) is an example of an expert system used for network intrusion detection [32], which has since evolved into a project called Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) [33]. EMERALD offers a suite of anomaly detection tools and a signature engine.

D. Machine-Learning and Data Mining Examples

1) Association Rules/inductive rules

Association rules are a data-mining approach to anomaly detection. It takes a set of variables and discovers relations that commonly exist between the values. For example, it may be common in intrusion situations to have the events {virus detected, ftp attempt} -> {data loss}. Almost everyone has had experience with association rule algorithms via the product recommendation algorithms ('people who bought this book also bought ....') on websites like Amazon.

Examples of association rules for use in network intrusion detection include [34, 35], where the authors use an association rule algorithm to detect abnormal record activity via unsupervised learning.

Association rules can work with both supervised and unsupervised data, and have the pros of being able to find their own rules. The cons of this type of method include the fact that rare events which cannot be trained for will trigger an 'anomalous' reading. This is likely to lead to a large number of false positives in a live computer network, which has much larger diversity than most people intuitively expect [26].

2) Bayesian Networks

A Bayesian network is a graphical model of a set of random variables, and the conditional decencies of those variables on each other. Bayesian networks are directional – that is they imply causality by the direction of the relationships between variables. They are similar to Markov models, but Markov models do not have directional links between the states. Bayesian networks have been used for network intrusion detection. Examples include [36, 37, and 38].

3) Markov Models

The Markov model-based intrusion detection systems try to calculate the likelihood of system in an anomalous state based on a sequence of observations. For example, a sequence of alerts from an IDS system such as Snort can be used to calculate the probability of system being under attack.

To understand this technique better, a brief description of Markov model is presented. (from [39]) A Markov model is a statistical model with N states S_1, S_2, ..., S_N and discrete timestamps. On a given timestamp, system is in exactly one of N states and between timestamps states are chosen randomly. An important property of a Markov model is that being at state S_t on timestamp t only depends on the system's state on timestamp t-1 and all the earlier states do not have any effect in selecting state S_t.

An extension to the Markov model is Hidden Markov Model (HMM) in which a system's states and state transitions are not visible and only events from these states are observable. Formally speaking, an HMM is a five-tuple (N,M,Π,A,B) where N is the number of states, M is the number of possible observations, Π is the starting state probabilities, A is the state transition probability matrix, and B is the observation

probability matrix. Since HMM is a statistical model, the initial values of Π, A, and B are selected randomly and a training process is required to make this model ready for actual data.

For example in [40], a computer network can be in one of Normal, Attempt, Progress, or Compromise states. These states are not known at a given time and only Snort events are observed. Snort events can be categorized into different groups based on their severity to limit the number of observable events. Initially we can assume the system is in Normal state and initialize state transition matrix and observation probability matrix with random numbers. Once the HMM trained, an intrusion detection system can find the probability of system being in one of states based on sequence of observed Snort events.

4) Neural Networks

In cases where a relationship between inputs and outputs is expected, but the exact relationship between the inputs and outputs is unknown, Artificial Neural Networks (ANN) (or just Neural Networks) are useful. ANN's are based on a simple model of how neurons work in the human brain. There are many nodes (or neurons) operating in parallel. Each node is connected to other neurons in the next layer by a specific weight. Changing the weights allow the network to model very complex, multidimensional functions – in practice this means that the differences between normal and anomalous behavior can be quite complex . In order to set the weights of each node, ANN's require large sets of supervised learning.

An example of a valid use for an ANN is the prediction of the real-estate value for houses given a large amount of pertinent information: e.g. neighborhood location, size of house, age of house, #of bedrooms, # of bathrooms, bungalow vs. two-story, etc. Given that large amounts of historical data for the values of houses are known, we could make a neural network that attempts to predict the value of a house given all of the relevant information – even though we don't know the exact relationships between the inputs and outputs.

Neural networks have been used in various ways for network intrusion detection [41-44]. In some works, neural networks are used to create self-organized maps, e.g. [45,46].

5) Fuzzy Logic

Some authors have used fuzzy logic approaches to network intrusion detection e.g. [47,48]. In fuzzy logic, other categories – rather than true or false – can be used for a set. The truth value of a fuzzy set lies somewhere between 0 and 1 [49]. Fuzzy logic has received some criticism as a tool [50], and some defense [51]. Pros of fuzzy logic include the ability to have non-conclusive outputs, but cons include high computational costs, and a lack of clarity on the proper approach with fuzzy logic (e.g. why not just use a percent output from the classification algorithm, with a probability from 0 to 1?).

6) Genetic Algorithms

Genetic algorithms are an optimization technique which is modelled on the process of biological evolution. Advantages of this optimization technique include the fact that it is capable of finding a global minimum in an optimization function with a large number of local minima, and the lack of assumptions required to use this algorithm. The largest disadvantage is that the huge amount of computational resources required to find the global minimum. These techniques have been used with respect to intrusion detection [52]. Genetic algorithms could be used to optimize the parameters for other detection algorithms. [53].

7) Clustering (K-means)

The k-means algorithm [54] is a centroid-based partitioning technique that takes the input parameter *k* and partitions *n* objects into *k* clusters so that an object is 'similar' to other objects within a cluster and different than objects in the other clusters. This algorithm thus groups like objects together automatically, in an unsupervised learning environment. To find the similarity between objects, a set of features are selected and distance between these features are measured. Initially, the k-means algorithm randomly selects k objects representing k clusters. As an example, Munz et al. [55] used network flow as the source for anomaly detection and selected total number of packets sent from/to a given port, total number of bytes sent from/to a given port, and number of source-destination pairs matching the given port number as features to compute similarity between different flows.

8) Support Vector Machines

Support Vector Machines operate as a group classifier by constructing a hyperplane or set of hyperplanes in a high dimensional space using training data to separate data into two groups of normal and abnormal (or malicious) classes. They generally require labelled training data.

One of the main advantages of SVM is its ability to discriminate data sets which are not readily separable by simpler techniques. If the 'normal' and 'abnormal' data sets to discriminate are not linearly separable or variations of features for two classes have overlaps, it maps the original data space into much higher space to make the separation easier. The function for mapping of features and the other parameters would be optimized under optimization methods. In practice, this means that input features which first appear to be poor indicators of malicious or abnormal activity may be good indicators with the SVM.

Another advantage of Support Vector Machines is that other applications have seen a low false positive rate [63]. Perhaps the biggest cost with SVM's are their complexity – in order to perform the non-linear mapping of the input feature space into the higher dimensional space, a particular mapping function must be selected and this often requires optimization techniques. This leads to long training times for the SVM.

VI. Conclusions

Through this literature review, we have reached several conclusions about anomaly detection and machine learning algorithms for use in real-world network intrusion detection systems.

- While many different anomaly detection approaches are outlined in the literature, there is significant overlap between many of them. For example, an 'expert system' could describe virtually any algorithm that has its initial inputs generated by a human expert, and Finite State Machines, Bayesian Networks, and Markov Networks are all extremely similar (and in some cases are sub-classes of each other)

- Due to the false positive problem when detecting rare events, and the fact that anomaly detection systems commonly have high false-positive rates, it is likely that a functional detection system will be comprised of several correlated detectors. This could include correlations with signature and specification based intrusion detection techniques.
- Given the lack of literature on feature selection, ad-hoc, expert supervised feature selection based on previous records of attacks will be the best method to generate relevant features.
- We suspect that a significant number of publications suggest algorithms which are not industrially viable.
- The evaluation of intrusion detection techniques is a problem from a scientific perspective [2]. This makes it very difficult to objectively determine which algorithm is the 'best'.

ACKNOWLEDGEMENTS

The authors would like to thank the Canadian Safety and Security Program at Defence Research and Development Canada for project funding.

REFERENCES

[1] "Data Mining". Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. Accessed March 2016., http://en.wikipedia.org/wiki/Data_mining,

[2] Tavallaee, Mahbod, Natalia Stakhanova, and Ali Akbar Ghorbani. "Toward credible evaluation of anomaly-based intrusion-detection methods." *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 40.5 (2010): 516-524.

[3] Chandola, Varun, Arindam Banerjee, and Vipin Kumar. "Anomaly detection: A survey." *ACM Computing Surveys (CSUR)* 41.3 (2009): 15.

[4] Eric M. Hutchins, Michael J. Clopperty, Rohan M. Amin, Ph.D."Intelligence-Driven Computer Network Defense Informed by Analysis of Adversary Campaigns and Intrusion Kill Chains". Lockheed Martin Corporation Abstract. Retrieved March 13, 2013

[5] Advanced Persistent Threats: A Decade in Review Command Five Pty Ltd June 2011.

[6] Liao, Hung-Jen, et al. "Intrusion detection system: A comprehensive review."*Journal of Network and Computer Applications* (2012).

[7] www.snort.org, accessed March 2016.

[8] Axelsson, Stefan. "The base-rate fallacy and the difficulty of intrusion detection." *ACM Transactions on Information and System Security (TISSEC)*3.3 (2000): 186-205.

[9] Dash, Manoranjan, and Huan Liu. "Feature selection for classification."*Intelligent data analysis* 1.3 (1997): 131-156.

[10] Estevez-Tapiador, Juan M., Pedro Garcia-Teodoro, and Jesus E. Diaz-Verdejo. "Anomaly detection methods in wired networks: a survey and taxonomy."*Computer Communications* 27.16 (2004): 1569-1584.

[11] [Lakhina, Anukool, Mark Crovella, and Christophe Diot. "Mining anomalies using traffic feature distributions." *ACM SIGCOMM Computer Communication Review*. Vol. 35. No. 4. ACM, 2005.

[12] Wang, Wei, and Roberto Battiti. "Identifying intrusions in computer networks with principal component analysis." *Availability, Reliability and Security, 2006. ARES 2006. The First International Conference on*. IEEE, 2006.

[13] Chebrolu, Srilatha, Ajith Abraham, and Johnson P. Thomas. "Feature deduction and ensemble design of intrusion detection systems." *Computers & Security*24.4 (2005): 295-307.

[14] Jose Andres Gonzalez, Numerical Analysis for Relevant Features in Intrusion Detection. MSc. Thesis, Department of the Air Force, Air University, Air Force Institute of Technology, Write-Patterson Air Force Base, Ohio, 2009.

[15] Denning, Dorothy E. "An intrusion-detection model." Software Engineering, IEEE Transactions on 2 (1987): 222-232.

[16] Axelsson, Stefan. *Intrusion detection systems: A survey and taxonomy*. Vol. 99. Technical report, 2000.

[17] Garcia-Teodoro, Pedro, et al. "Anomaly-based network intrusion detection: Techniques, systems and challenges." *computers & security* 28.1 (2009): 18-28.

[18] Lazarevic, Aleksandar, et al. "A comparative study of anomaly detection schemes in network intrusion detection." *Proc. SIAM* (2003)

[19] Wegner, Ryan. Multi-Agent Malicious Behaviour Detection,. PhD Thesis, Department of Computer Science, University of Manitoba, 2012.

[20] Patcha, Animesh, and Jung-Min Park. "An overview of anomaly detection techniques: Existing solutions and latest technological trends." *Computer Networks* 51.12 (2007): 3448-3470.

[21] Tsai, Chih-Fong, et al. "Intrusion detection by machine learning: A review."*Expert Systems with Applications* 36.10 (2009): 11994-12000.

[22] Kind, Andreas, Marc Ph Stoecklin, and Xenofontas Dimitropoulos. "Histogram-based traffic anomaly detection." *Network and Service Management, IEEE Transactions on* 6.2 (2009): 110-121.

[23] Ye, Nong, et al. "Multivariate statistical analysis of audit trails for host-based intrusion detection." Computers, IEEE Transactions on 51.7 (2002): 810-820.,

[24] Ye, Nong, et al. "Probabilistic techniques for intrusion detection based on computer audit data." Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on 31.4 (2001): 266-274.

[25] Kruegel, Christopher, et al. "Bayesian event classification for intrusion detection." Computer Security Applications Conference, 2003. Proceedings. 19th Annual. IEEE, 2003

[26] 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, 2010.

[27] Beigi, Mandis S., et al. "Anomaly detection in information streams without prior domain knowledge." *IBM Journal of Research and Development* 55.5 (2011): 11-1.

[28] Beigi, Mandis, et al. "Muti-scale temporal segmentation and outlier detection in sensor networks." Multimedia and Expo, 2009. ICME 2009. IEEE International Conference on. IEEE, 2009.

[29] "Finite-state Machines." Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. Accessed March 2016., http://en.wikipedia.org/wiki/Finite-state_machine

[30] "Markov Chain." Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. Accessed March 2016., http://en.wikipedia.org/wiki/Markov_chain

[31] "Expert System." Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. Accessed March 2016., http://en.wikipedia.org/wiki/Expert_system

[32] Anderson, Debra, Thane Frivold, and Alfonso Valdes. Next-generation intrusion detection expert system (NIDES): A summary. SRI International, Computer Science Laboratory, 1995.

[33] http://www.csl.sri.com/projects/emerald/, accessed March 2016

[34] Das, Kaustav, Jeff Schneider, and Daniel B. Neill. "Anomaly pattern detection in categorical datasets." Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2008.

[35] Das, Kaustav, and Jeff Schneider. "Detecting anomalous records in categorical datasets." Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2007.

[36] Bronstein, Alexandre, et al. "Self-aware services: Using bayesian networks for detecting anomalies in internet-based services." Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on. IEEE, 2001

[37] Kruegel, Christopher, et al. "Bayesian event classification for intrusion detection." Computer Security Applications Conference, 2003. Proceedings. 19th Annual. IEEE, 2003.

[38] Barbara, Daniel, Ningning Wu, and Sushil Jajodia. "Detecting novel network intrusions using bayes estimators." First SIAM Conference on Data Mining. 2001.

[39] Moor, Andrew. "Statistical Data Mining Tutorials", http://www.autonlab.org/tutorials/, accessed March 2016.

[40] Shameli Sendi, Alireza, Michel Dagenais, Masoume Jabbarifar, and Mario Couture. "Real Time Intrusion Prediction based on Optimized Alerts with Hidden Markov Model." Journal of Networks 7, no. 2 (2012): 311-321.

[41] Ryan, Jake, Meng-Jang Lin, and Risto Miikkulainen. "Intrusion detection with neural networks." Advances in neural information processing systems. MORGAN KAUFMANN PUBLISHERS, 1998.,

[42] Mukkamala, Srinivas, Guadalupe Janoski, and Andrew Sung. "Intrusion detection using neural networks and support vector machines." Neural Networks, 2002. IJCNN'02. Proceedings of the 2002 International Joint Conference on. Vol. 2. IEEE, 2002.,

[43] Zhang, Zheng, et al. "HIDE: a hierarchical network intrusion detection system using statistical preprocessing and neural network classification." Proc. IEEE Workshop on Information Assurance and Security. 2001.

[44] Cannady, James. "Artificial neural networks for misuse detection." National information systems security conference. 1998.

[45] Depren, Ozgur, et al. "An intelligent intrusion detection system (IDS) for anomaly and misuse detection in computer networks." Expert systems with Applications 29.4 (2005): 713-722.

[46] Ramadas, Manikantan, Shawn Ostermann, and Brett Tjaden. "Detecting anomalous network traffic with self-organizing maps." Recent Advances in Intrusion Detection. Springer Berlin Heidelberg, 2003.

[47] Dickerson, John E., and Julie A. Dickerson. "Fuzzy network profiling for intrusion detection." Fuzzy Information Processing Society, 2000. NAFIPS. 19th International Conference of the North American. IEEE, 2000.

[48] Gomez, Jonatan, and Dipankar Dasgupta. "Evolving fuzzy classifiers for intrusion detection." Proceedings of the 2002 IEEE Workshop on Information Assurance. Vol. 6. No. 3. New York: IEEE Computer Press, 2002.

[49] "Fuzzy Logic." Wikipedia, The Free Encyclopedia. Wikimedia Foundation, Inc. Accessed March 2016., http://en.wikipedia.org/wiki/Fuzzy_logic

[50] Haack, Susan. "Do we need "fuzzy logic"?." International Journal of Man-Machine Studies 11.4 (1979): 437-445.

[51] Zadeh, Lotfi A. "Is there a need for fuzzy logic?." Information Sciences 178.13 (2008): 2751-2779

[52] Li, Wei. "Using genetic algorithm for network intrusion detection." Proceedings of the United States Department of Energy Cyber Security Group (2004): 1-8.

[53] Bridges, Susan M., and Rayford B. Vaughn. "Fuzzy data mining and genetic algorithms applied to intrusion detection." Proceedings twenty third National Information Security Conference. 2000.

[54] Hartigan, J. A.; Wong, M. A. (1979). "Algorithm AS 136: A K-Means Clustering Algorithm". Journal of the Royal Statistical Society, Series C 28 (1): 100–108. JSTOR 2346830.

[55] Münz, Gerhard, Sa Li, and Georg Carle. "Traffic anomaly detection using k-means clustering." Proc. of Leistungs-, Zuverlässigkeits-und Verlässlichkeitsbewertung von Kommunikationsnetzen und Verteilten Systemen 4 (2007).

[56] Horng, Shi-Jinn, Ming-Yang Su, Yuan-Hsin Chen, Tzong-Wann Kao, Rong-Jian Chen, Jui-Lin Lai, and Citra Dwi Perkasa. "A novel intrusion detection system based on hierarchical clustering and support vector machines." Expert systems with Applications 38, no. 1 (2011): 306-313.

[57] Mukkamala, Srinivas, and Andrew H. Sung. "Feature selection for intrusion detection with neural networks and support vector machines." *Transportation Research Record: Journal of the Transportation Research Board* 1822.1 (2003): 33-39.

[58] Kim, Dong Seong, and Jong Sou Park. "Network-based intrusion detection with support vector machines." Information Networking. Springer Berlin Heidelberg, 2003.

[59] R. Lippmann, R. K. Cunningham, D. J. Fried, I. Graf, K. R. Kendall, S. E. Webster, and M. A. Zissman, "Results of the 1998 DARPA Offline Intrusion Detection Evaluation," in Proc. Recent Advances in Intrusion Detection, 1999.

[60] "KDD Cup Data," http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.

[61] M. Tavallaee, E. Bagheri, W. Lu, and A. Ghorbani, "A Detailed Analysis of the KDD CUP 99 Data Set," Submitted to Second IEEE Symposium on Computational Intelligence for Security and Defense Applications (CISDA), 2009.

[62] McHugh, John. "Testing intrusion detection systems: a critique of the 1998 and 1999 DARPA intrusion detection system evaluations as performed by Lincoln Laboratory." ACM transactions on Information and system Security 3.4 (2000): 262-294.

[63] R. Perdisci, D. Ariu, P. Fogla, G. Giacinto, and W. Lee, "McPAD: A Multiple Classifier System for Accurate Payload based Anomaly Detection," Computer Networks, vol. 53, no. 6, pp. 864—881, Apr. 2009.

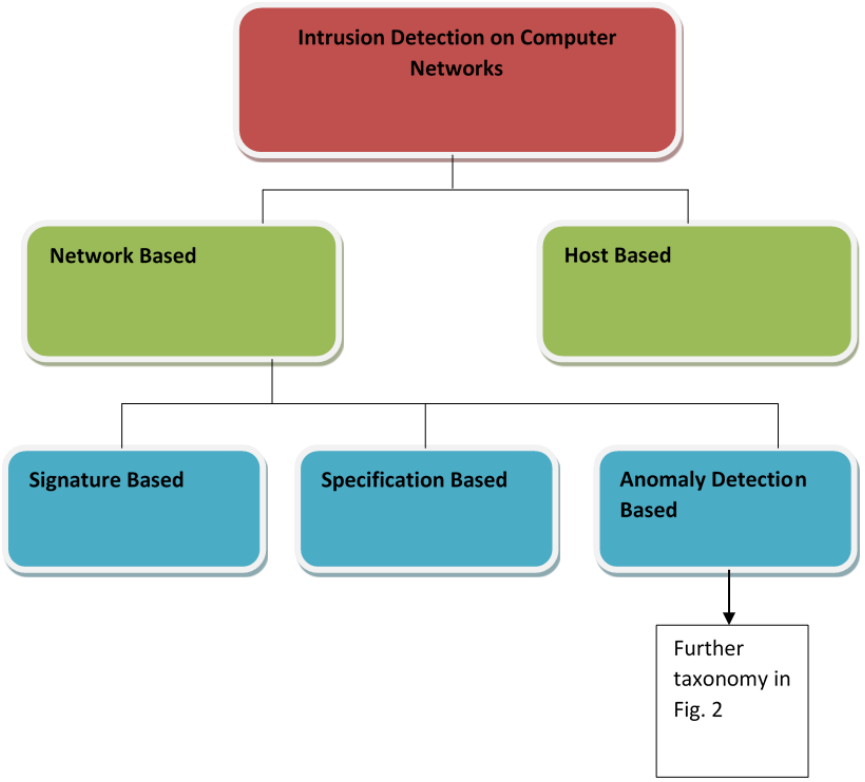

Fig. 1: Taxonomy of Intrusion Detection on Computer Networks. We are considering network-based Anomaly Detection based methods. Note that host based methods have the same sub-taxonomy (not shown).

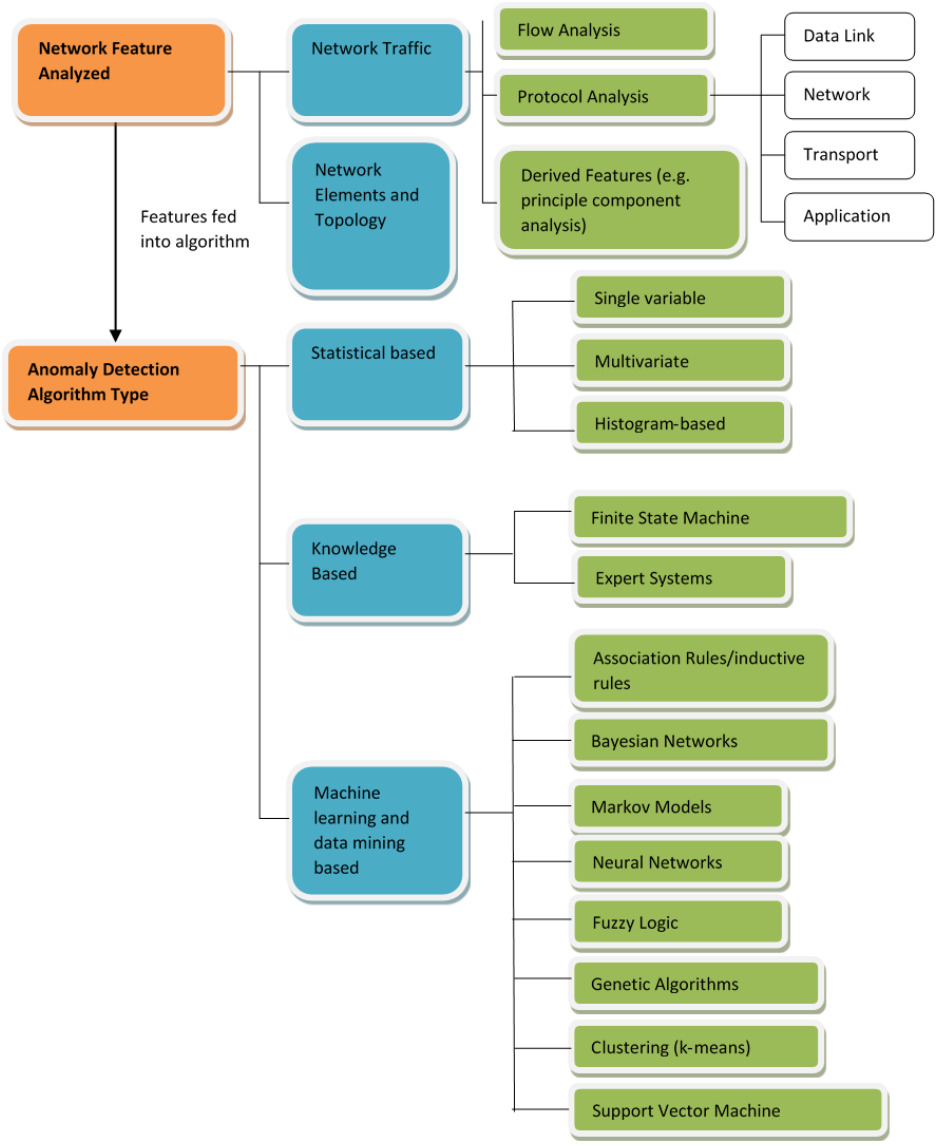

Fig. 2. A Taxonomy of Anomaly Detection methods. Adapted from [10] and [17].

Table 1: Pros and Cons of Various Anomaly Detection Methods

Method Name	PROS	CONS
Univariate/Multivariate	• Simple to use quick to implement, • Can use on numerous parameters and correlate • Unsupervised learning possible	• High false positive rate, • Unreasonable assumptions about underlying probability density functions • Likely to miss small changes in data
Histogram	• Simple to use • Can deal with multiple time-scales quickly • Makes no assumptions about underlying statistics • Unsupervised learning possible	• Due to averaging nature, likely to miss small changes in data • Need large amounts of data to work with
Finite State Machine	• Contribution of human expert can limit number of states	• Finite State Machines are a subset of Markov models • Need to define states, when states may be unclear
Expert System	• Exact definition of this system unclear: any type of human-influenced detector could be called an 'expert system'	• Vagueness of definition makes this hard to decide 'expert system' represents a method or not.
Association Rules/ Inductive Rules	• Useful for identifying new patterns in data set • Unsupervised learning • Find patterns without human intervention	• For rare events, high false positive rate • Need significant amount of data to reliably determine associations
Bayesian Networks	• Good algorithms exist for training • Can encode causal relationships • Widely used • Unsupervised learning	• Results sometimes outperformed by much simpler methods • States must be defined by user • Similar to Markov Models •
Markov Models	• Can use Hidden Markov Models so states do not need to be known (only outputs) • Unsupervised learning	• Similar to other state-based approaches
Neural Networks	• Unsupervised learning possible, but best results with • No assumptions about statistical model	• Supervised learning is desirable • Lots of data is required
Fuzzy Logic	• Uses linguistic variables	• Other ways of expressing uncertainty • Not widely used in literature
Genetic Algorithms	• Capable of finding local minima in optimization problems • Possible to use to solve sub-problems of the overall anomaly detection algorithm	• Very slow • Lots of computational resources • More efficient methods available
Clustering	• Simple algorithms exist to create clusters • Unsupervised learning	• False positives are likely for outlier events: as network traffic more diverse than typically thought
Support Vector Machines	• Excellent method at finding separable classes • Low false positive rate • No assumptions about statistical model	• Must have labelled training data (supervised learning) • Difficulty in finding the function to map decision space onto new dimensions

Appendix C

Presentation Slides

Application of Machine Learning to Computer Network Security

Jason Haydaman

August 17, 2017

University of Manitoba

Introduction

Problem Statement

Does Machine Learning have anything to offer an Intrusion Detection System (IDS)?

- Anomaly detection – Possible APT or 0-day attack?
- Classification – Security compromise or benign traffic?

Very little Machine Learning to be found in commercial IDS products. Why?

Why Focus on Machine Learning?

- 0-days and APTs.
- Creating signatures is expensive and doesn't scale.
- Machine Learning is theoretically harder to evade (no model visibility).

Overview of Research

- Swarm Sensor Network
- Intrusion Detection
- Data Collection
- Modeling
- Traffic Labeling

Swarm Sensor Network

Design Overview

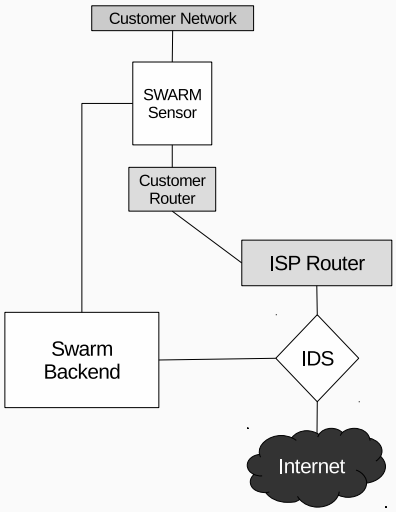

Figure 1: Swarm Sensor Network Overview

Intrusion Detection

Replication Study

With Swarm in place, we could now focus on Machine Learning.

- Obtained ISCX 2012 Data set.
- Trained an SVM classifier with ISCX 2012 as training data.
- 97% accuracy against ISCX 2012 test data.
- 80% false positive rate against our own test data.

5

Revision

Enormous discrepancy between real world accuracy and the confusion matrix. How do we do better?

1. For flows labeled malicious by SVM, gather additional information (IDS logs, DNS logs, HTTP logs).
2. Human analysis of malicious flows and additional information provides ground-truthing.
3. Develop heuristics to filter out observed false-positives after ground-truthing.
4. Repeat until the false positive rate is acceptable.

After many iterations of the above algorithm, new real world false positive rate: 50%.

6

Comparison

How does this compare to existing IDS?

We developed a python module for the OneStone IDS that we call the Reverse Geographic Communication Correlator. It alerts when machines on the network connect back to IP space that has unsuccessfully attempted to connect to our network over some past time window (hours – days).

No false positives. Very useful for identifying malicious traffic.

Data Collection

Malware Sandbox

Clearly the ISCX data set was not working out for us. Can we develop a better data set?

- Didn't want to rely on IDS-labeled flow data, then we would only be training an IDS as good as what we already have
- Needed a source of data that was known malicious: A malware sandbox.
- Ran 2000 samples of malware through a network of Windows 7 VMs.
- Most were inactive on the network, not enough data collected to be useful.

8

Modeling

A Detailed Look at Traffic

Without good data sets or the ability to create good data sets, can we determine what about network traffic makes the classification problem so hard?

Features we used:

- Duration of flow
- Protocol number
- Source port
- Destination port
- Number of packets per flow
- Number of bytes per flow
- TCP flags
- Number of flows from matching (src, dst) and (dst, src) IP and port in past 300 seconds

9

Feature Selection

The features we selected were based on what we had available rather than what we would like to have.

Can we do better?

Encryption destroys most higher-level features that we might try to use. Can we detect encrypted traffic in a generic way?

Encrypted Traffic Detection

Encrypted data looks like uniform random noise

$$H(\mathbf{X}) = -\sum_{x \in \mathbf{X}} p(x) \log_2 p(x) \approx 1 \tag{1}$$

What is $p(x)$? In general, we don't know. Network traffic doesn't exactly meet the criteria of independent and identically distributed, and $p(x)$ may not be well-defined.

Can we estimate the entropy of a data source based on a finite length sample of its output?

PRNG Testing

There is a large body of literature on evaluating PRNGs for randomness. Approximate Entropy is defined as:

$$\hat{H}_f^d = -\sum_{\mathbf{a}\in\mathcal{A}^d} \tilde{\pi}_{\mathbf{a}}^d \log_2 \tilde{\pi}_{\mathbf{a}}^d + \sum_{\mathbf{a}\in\mathcal{A}^{d-1}} \tilde{\pi}_{\mathbf{a}}^{d-1} \log_2 \tilde{\pi}_{\mathbf{a}}^{d-1} \tag{2}$$

$$\tilde{\pi}_{\mathbf{a}}^d = \frac{1}{n} |\{0 \leq i < n : \tilde{x}_i^d = \mathbf{a}\}| \tag{3}$$

$$\tilde{x}_i^d = (x_i, x_{i+1}, \ldots, x_{i+d-1}) \tag{4}$$

Where $\mathbf{a} \in \mathcal{A}^d$ and we consider $\mathcal{A} = \{0, 1\}$.

PRNG Testing

Then, approximate entropy converges to entropy as $n \to \infty$.

$$H = \lim_{n \to \infty} \hat{H}_f^d \tag{5}$$

Assumptions:

- $\mathbf{X}$ is a markov process of order $< d$.
- $\mathbf{X}$ is a stationary ergodic process.

Do these assumptions hold?

Network Traffic

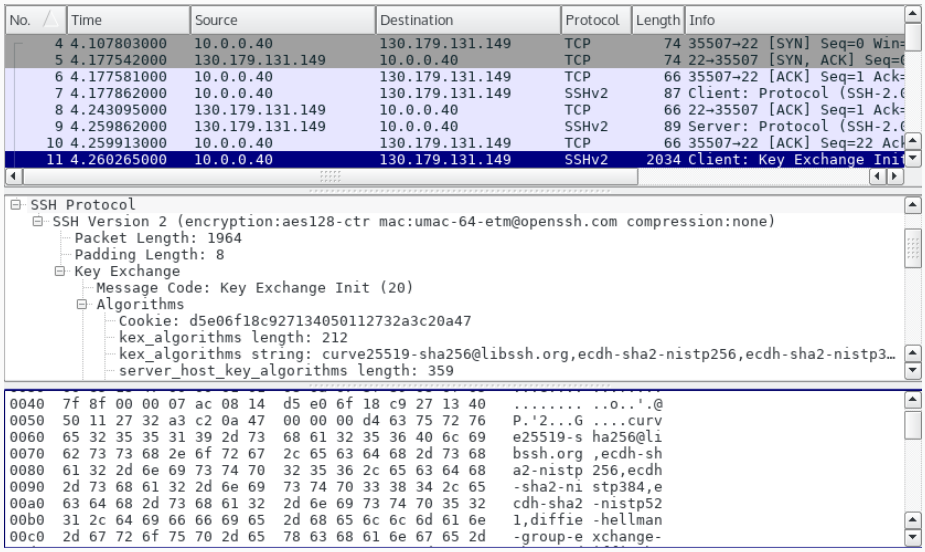

Figure 2: Packet-level view of scp transfer

Traffic Labeling

Stationary Distributions

Network traffic is not a stationary ergodic source, even for a single connection from one program.

Machine Learning assumes that training data and testing data belong to the same distribution.

There is research into dealing with Covariate Shift, i.e. distributions that drift between train and test, but I found no sound method of dealing with distributions that are not stationary even during train.

15

Answers to the Problem Statement

Q2: Very little Machine Learning to be found in commercial IDS products. Why?

A2: No self-similarity of traffic makes anomaly detection not work, and no stationary datasets makes classification not work.

Q1: Does Machine Learning have anything to offer an Intrusion Detection System (IDS)?

A1: Maybe. New Question:

Can we turn a data set of non-stationary distributions into n data sets of stationary distributions, to then train n models?

16

Traffic Distributions

- Network traffic is the output of a program.
- A program can be in many states.
- Any state can influence the distribution of network traffic arbitrarily.
- Conjecture: If we can label traffic by the state of the program that produced it, maybe the set of all traffic with the same state label would be stationary.

17

State Labeling

What is a program state?

- Most generally: Some member of the set of all combinations of the memory and CPU registers, and all of the inputs to a program. This is an astronomical set, not helpful.
- No clear answer. What is clear is that whatever the answer is, if to be used in a data set, requires we have the ability to map packets to that program state, and so therefore, to the program itself.

New Question: How do we map packets on a network to their source process?

18

Packet Source Mapping

Requirements:

- Fast, little overhead.
- Visibility into all packets in a system.
- Visbility into all processes in a system.
- Hooked into the network stack such that it cannot be bypassed by malware.

Sounds like it needs to be in the kernel.

19

Packet Source Mapping

- Introduced a netfilter hook that userspace can communicate with via netlink.
- Netlink channel acts like a PF_PACKET raw socket with an additional structure containing pid information.
- Userspace can be anything that speaks sockets.
- Test application was written in python.
- Performance benchmarks measured using packet capture of 10MB scp transfer.

20

Performance – Baseline

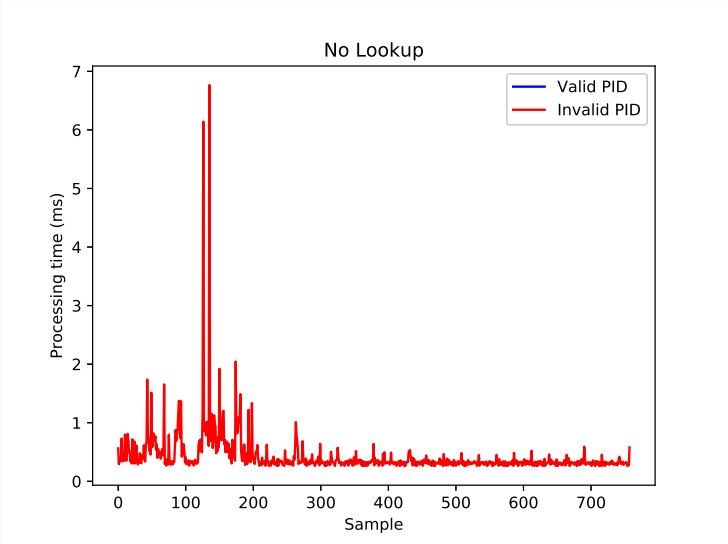

Figure 3: Per-packet processing time with no pid resolution

Performance – Userspace Method

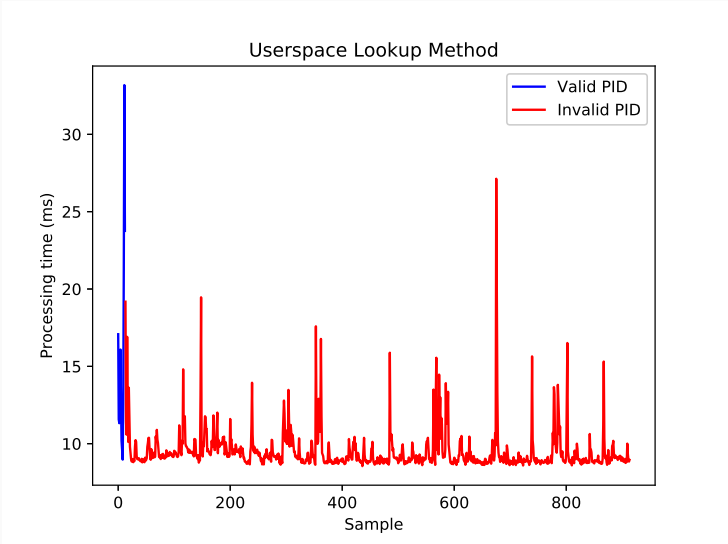

Figure 4: Per-packet processing time with userspace pid resolution

Performance – Kernel Method

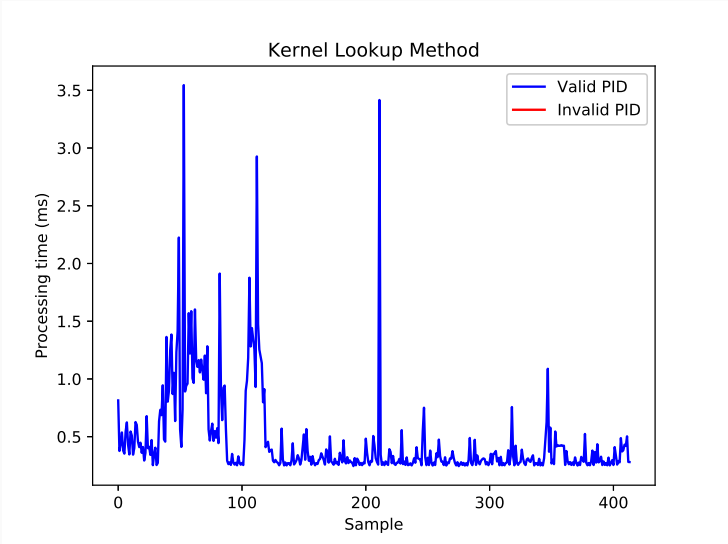

Figure 5: Per-packet processing time with kernel pid resolution

Performance – Summary

method	min (ms)	max (ms)	mean (ms)	stdev (ms)
Kernel	0.245	3.55	0.463	0.394
Userspace	8.54	33.2	9.59	1.74
Control	0.259	6.765	0.418	0.377

Table 1: Packet Processing Times

Conclusions and Future Work

Conclusions and Future Work

We are left with one unanswered question: Under what conditions can it be said that the output of a program is stationary?

I conjecture that the answer to that question yields the answer to what Machine Learning has to offer IDS. Either there is no answer, in which case I would look to Machine Learning research for how to model such data sets, or there is such an answer, in which case if it can yield real-time labels of process state, then we could have data sets that don't violate the assumptions made by Machine Learning algorithms.

25