DYNAMIC CONTENT GENERATION FOR THE EVALUATION OF NETWORK APPLICATIONS

By

VICTOR CRAIG VALGENTI

A dissertation submitted in partial fulfillment of
the requirements for the degree of

DOCTOR OF PHILOSOPHY

WASHINGTON STATE UNIVERSITY
School of Electrical Engineering and Computer Science

MAY 2012

To the Faculty of Washington State University:

The members of the Committeee appointed to examine the dissertation of VICTOR CRAIG VALGENTI find it satisfactory and recommend that it be accepted.

Min Sik Kim, Ph.D., Chair

Carl H. Hauser, Ph.D.

David E. Bakken, Ph.D.

ACKNOWLEDGEMENTS

First, I would like to acknowledge and thank my Advisor, Dr. Min Sik Kim, for his help and patience over the last five years. I greatly appreciate the opportunity afforded to me and the insights fostered in our discussions. I would also like to thank Dr. Carl Hauser and Dr. David Bakken for advice and help offered over the years. I further wish to acknowledge Aaron Crandal who served as an unofficial mentor to me over the years and a springboard for many of my ideas. Lastly, but not least, I wish to thank my wife who has had to indirectly deal with the volatility graduate study can create.

DYNAMIC CONTENT GENERATION FOR THE EVALUATION OF NETWORK APPLICATIONS

Abstract

by Victor Craig Valgenti, Ph.D.
Washington State University
May 2012

Chair: Min Sik Kim

Generating application-level content within network simulations and/or testbed environments tends toward an ad-hoc process reliant primarily on evaluator expertise. Such ad-hoc approaches are laborious and often fail to capture important aspects of how content is distributed within traffic. Further, while many tools allow for the generation of a wide-range of content types, there exists no coherent model for populating these tools with the necessary data. To address these issues we propose two models for dynamically generating content so as to provide a systematic means for populating a test with relevant data. First we create content targeting Network Intrusion Detection Systems (NIDS) that are severely impacted by the composition of the traffic combined with the set of known signatures. Most NIDS evaluation techniques employ on/off models where a packet is either malicious or not. Such evaluation ignores the case where the content of a benign packet partially intersects with one or many signatures, causing more processing for the NIDS. To address this hole in evaluation we propose a traffic model that uses the target NIDS signature set to create partially-matching traffic. This partially-matching traffic then allows the systematic examination of the NIDS across multiple scenarios. Such evaluation provides insight into the idiosyncrasies of a NIDS that would remain hidden if evaluated under current methodologies.

Next, we broaden our content generation model to account for all network applications.

We create a content generative model for identifying, harvesting, and assigning application-level content to simulated traffic. This model ties consumers of content to the producers of the content as well to a particular content category. This approach then allows for said content to be tied to a workload generator or simulator of choice to evaluate a given network application. Finally, we discuss the implementation of these models and potential optimizations for high-speed environments. Ultimately, the models provided here allow for the systematic generation of content for network applications and serves to bridge the gap in current evaluation methodologies between network traffic simulation and content.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER one

INTRODUCTION

1.1 Motivations Driving Research in Traffic Generation

The art of testing computer networks has perplexed researchers, engineers, and system administrators for years. Initial models and analysis stem all the way back to A.K. Erlang [1] and attempts to quantify Telephone switching networks. However, the ubiquity of the Internet and the ever increasing number of applications utilizing it as a medium for data transport has increased not only the need for successfully generating computer network test traffic, but for modeling that traffic as well. A significant symptom of this need is the number of traffic generation tools available freely, or for cost, to meet various traffic generation objectives. Yet, despite the best efforts of numerous researchers, engineers, and hackers, there does not yet exist a general purpose traffic generator that adequately and accurately captures all conceivable objectives for computer network evaluation. This stems from the heterogeneous nature of network applications and the desired test conditions and stimuli to evaluate them. In essence, the test objectives determine the level of abstraction and simplification allowed in any evaluation. Hence, many traffic generators are not the result of a determined effort to generate realistic traffic but just a need to evaluate a very specific aspect of a singular network application.

The task of generating network traffic representative of a particular network is quite daunting for any network larger than a few nodes. The creation of large test-bed networks is costly and time-consuming, while simulation alone is often inadequate. Evaluation on live networks carries numerous financial, privacy, and security concerns that might arise from a test. Thus, generated test traffic remains a necessary component in evaluating network applications. Since the quality of the traffic generated is a function of the application to test then it is necessary to adopt the proper tool for the task.

One of the more perplexing issues, however, is determining if the correct traffic generation tool is applied to the evaluation of a particular application. As demonstrated by Vishwanath et al. [2] the composition of network traffic will directly impact the performance of network applications. Burstiness in traffic, an every-day reality in most networks, can severely strain most network applications. Further, the depth of composition of generated traffic can have a unique impact on different applications. For example, replaying packet captures, such as done by tcpreplay [3], will replay traffic as experienced on a real network. Thus, the traffic generation represents a single time-slot which may, or may not, contain the necessary stimuli to test the desired application. While simplifying the generation of traffic to facilitate the testing of particular applications is admirable, it can insert bias into garnered results. Thus, it is necessary to ensure that generated traffic meet conditions sufficient and necessary to evaluate a particular application.

Due to ever increasing constraints in testing, as well as burgeoning privacy issues, simulation and traffic generation are slowly coming together. The optimal traffic generator is capable of simulating a variety of users across multiple applications and virtual networks such that the end result is a massive, simulated network that is responsive to user interaction. Of course, as more simulation is added to traffic generation the complexity of the generators increase. This added complexity can demonstrate itself in large resource requirements whether those resources be labor required in creation of sufficient test-bed environments or actual equipment purchased to house the projects. However, the added features of simulation can provide for more generalized traffic generators, even if the creation and use of such generators requires considerable effort.

A substantial motivating factor of this research stems from a personal encounter in trying to identify a traffic generator sufficient to test Network Intrusion Detection Systems (NIDS). While many traffic generators are available, few provide the necessary features for testing and none the exact combination of desired features. Further, little attempt has been made to quantify or catalog the vast array of traffic generators out there. It is left to the individual to dive into these many tools and try to divine the adequacy of a particular tool to a particular task. Thus, this research attempts to categorize and catalog many current traffic generators by their major functionality as well as list many of the features common to traffic generators. Next, we specifically identify the concerns for NIDS that employ deep packet inspection. Since these NIDS examine the payload of a packet traffic payload across all packets in an evaluation can prove a significant factor during evaluation. We then provide two distinct methods for generating packet payloads in order to evaluate NIDS. The first method is sufficient to produce load in a NIDS and the second is a more general approach to provide background traffic that can serve applications other than NIDS. Finally, we examine implementation concerns such as providing for high throughput and constrained resources. Our hope is that this research will provide a basis for improved evaluation of NIDS and Network Applications.

1.2 The Purpose of Network Traffic Generation

The purpose for network traffic generation falls into two broad categories. The first is to stimulate a particular network application. In this sense, traffic generated becomes a tool for determining an application's responsiveness to stimuli. This is the primary reason for the disparity between network traffic generators. Each tool is targeted at specific network applications. Thus, the traffic generated by such tools may not generalize well to other applications. For example, in Chapter 4, we examine traffic generation with respect to Network Intrusion Detection Systems (NIDS). Thus, the models and techniques provided in Chapter 4 create traffic well-suited to evaluating NIDS. However, such traffic would not necessarily serve as well to test other network applications like routing software. Regardless, traffic generation serves to demonstrate the functionality of a particular network application in isolation.

Secondly, network traffic generation is designed to produce background traffic. Quite often, the impact of background traffic is ignored. As such, any tool capable of spitting packets onto the wire is considered feasible for valid background traffic. However, background traffic can have a serious impact on the function of a network application as demonstrated by Vishwanath et al [2], and by ourselves [4] in this work as will be detailed in further chapters. The ultimate goal of background traffic generation is to convincingly create network traffic that will approximate desired conditions that might exhibit themselves upon a particular network.

In practice, it is common to utilize two separate tools to accomplish both purposes. One tool is used to generate background traffic and a second tool is used to create traffic specific to testing an application. While there is nothing wrong with this approach, it fails to recognize the fact that the background traffic can, in some instances, prove as meaningful to the application as the traffic specifically targeting the application. This is the case in NIDS and thus a primary factor in the research developed herein.

In order to provide useful traffic, any traffic generator must be capable of meeting a certain minimum of qualified traffic. In other words, the generated traffic must serve a purpose for an evaluation otherwise it provides no service. As such, generated traffic targets particular features found in real traffic. These features range from actual communications, like an FTP-client downloading a file, to statistical properties inherent in traffic like inter-arrival times between packets. Further, these features may also attempt to explore ranges that are possible given the definitions for protocols or applications, but might never occur in reality except in mischief. Figure 1.1 illustrates this dichotomy. Real traffic contains a set of features, represented by the circle on the left. Generated traffic contains its own set of features, represented by the circle on the right. The intersection

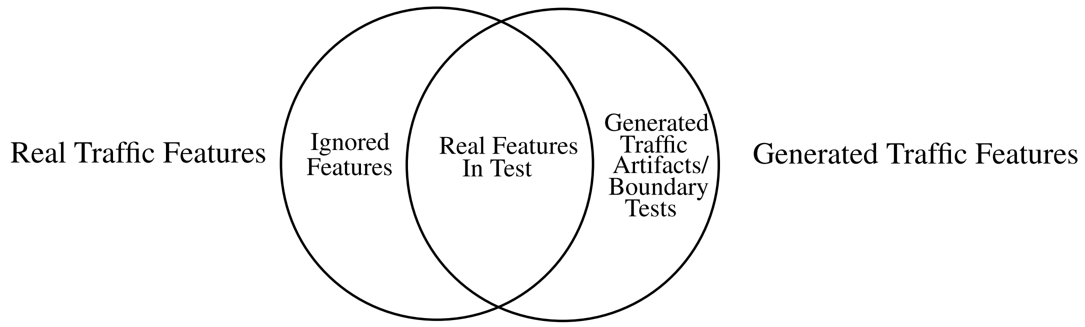

Real Traffic Features
Ignored Features
Real Features In Test
Generated Traffic Artifacts/ Boundary Tests
Generated Traffic Features

Figure 1.1: Generated traffic targets a subset of Real Traffic Features.

between these two sets represent the real traffic features that the generated traffic approximates. In other words, these are the features targeted by the traffic generator. The remainder of the real traffic features are ignored either because they are deemed irrelevant to a particular evaluation, because their relevance is unknown, or simply because they are too difficult to implement. The features in the generated traffic that are not part of real traffic are those features that either explore the boundaries of possible traffic (i.e. boundary testing) or are artifacts introduced into the traffic stream through the traffic generator. In fact, one of the complaints Jim McHugh [5] makes against the 1998 DARPA Intrusion Detection System Evaluation data sets [6, 7] is that due to a limited set of targets of attack within the generated traffic it is a simple matter to tune a system to watch only these targeted systems and ignore all others to greatly decrease the potential for false positives. This represents one of the many subtle ways in which the artifacts from generated traffic have the potential to bias an evaluation. It is also why generated traffic is more successful in very specific roles rather than broader, more general, roles.

1.2.1 Common Traffic Features Targeted by Network Traffic Generation

This section details the most common features targeted by traffic generators. Most of these features represent the more studied features of traffic which has indirectly resulted in the development of tools to test hypotheses. Other features are common to network administrators and evaluators to ascertain real-time statistics of live networks. As mentioned earlier, traffic generators tend to target features since simulating and generating completely realistic traffic is, to-date, infeasible. Thus, as illustrated in Figure 1.1 only the impact of a certain subset of features is examined with the assumption being that bias and/or ignored features do not impact the validity of a given test—an assumption that depends largely on the conclusions made from the results of the experiment. The list of features here is not conclusive, but captures the majority of the most popular traffic features targeted by traffic generators. Further, this list is primarily concerned with the traffic generation features that, while not necessarily distinct to NIDS evaluation, are at least commonly applicable to NIDS evaluation.

Inter-arrival Times

Inter-arrival times for network traffic has been examined as a key property of telecommunications since A. K. Erlang [1]. This stems, in part, from the fact that telecommunications translate well into queuing theory. As such, the tools for modern queuing theory can be used to study many of the properties of network infrastructure, in particular the relative reliability with which a network packet can expect to arrive at a destination. Of course, modern research has illustrated that this process is considerably more complex especially in the presence of self-similar network traffic [8–10]. In order to capture this feature, many traffic generators employ the ability to set packet inter-arrival times to either a Poisson process, or apply some mechanism for emulating bursty behaviors.

Packet Volumes

Much network research looks only at high-level traffic features. In particular, research concerned with on-line anomaly detection, like that forwarded by Lackhina et al [11] and Ahmed et al [12], use gross packet counts as a primary factor in detecting anomalies. Many traffic generation tools support methods for creating bursts of traffic as a result of inter-arrival times shortening due to self-similar inter-arrival times, or simple timed bursts of higher sending rates.

Packet-length Distributions

A large amount of recent research has demonstrated that packet-length distributions of flows can serve to distinguish particular types of communications. This particular feature appears as a strong indicator of Peer-to-Peer (P2P) traffic as illustrated by Erman et al in [13].

Port Distributions

Typically not an explicit feature in most traffic generators, the ability to generate traffic with a designated set of ports is an important part of many tests. The destination port is a very important feature in the performance of NIDS like Snort [14] as it can determine the number of rules applied to a given packet. Further, the port distributions (both source and destination) found in traffic can serve to help identify P2P networks as demonstrated by Erman et al. [13]. Most traffic generators allow ports to be designated at some level, but models for generating port distributions are ad-hoc in nature.

Bandwidth, Delay, and Latency

A large amount of research exists for Bandwidth testing. The typical goal is to provide tools that can measure the available bandwidth for a link. Early tools, such as iperf [15] simply fill the link with packets to derive the statistics. More refined approaches such as that exhibited in the Network Weather Service [16], coordinate small, periodic measurements in order to derive network statistics. As such, the generation is less about the traffic and more about how quickly that traffic is moved between two points. Evaluating these features is of less importance to NIDS, but there still exist many traffic generation tools dedicated to the derivation of these statistics. This is more a result of the need for system administrators to verify the performance of networks in real-life than the need for research tools. Despite the relative lack of realism in traffic generated from these tools, their simplicity mark them as one of the most common means of generating background traffic.

Flows

The number and distribution of network flows, as defined by the IP flow-level quintuple(Internet Protocol (IP), IP source address, IP destination address, source port, and destination port), demonstrates behaviors such as the fact that small flows account for about 80% or more of all flows, but large flows account for 80% or more of the total traffic [17]. Further, network traffic tends to heavily favor TCP traffic over all others. Several traffic generators allow for generating traffic to meet such statistics, albeit with considerable scripting required.

Load

Load generation is a common technique for evaluating network applications. The primary purpose for load generation is simply to create a huge number of packets or flows in order to overwhelm a target. The idea is that an application can be evaluated by simply examining its behavior as the volume of traffic increases. This type of evaluation is extremely common and is the motivation behind many traffic generators. Unfortunately, defining load for any network application is much more complex than simply increasing the arrival rate of packets as is illustrated later in this work as well as by Vishwanath et al in [2] and Sommers et al in [18].

Think Times

Think Times represent the pause between communications indicating the amount of time a user or client takes before sending or requesting more data. Think Times are slowly becoming a more popular feature in traffic generation, though only a few generators actually implement them.

Topology

All traffic generators allow the generation of traffic within specific topology constraints. However, few offer any tools for simulating, emulating, or creating a topology. Research like that from Sommers et al. [19] demonstrate methods for generating realistic IP Addresses, and research like the Rocketfuel project out of the University of Washington [20] demonstrate methods for mapping wide area topology. However, there exist few tools in traffic generation that actually employ or even allow for topology specific tools. This is one of the secondary motivations for the Generative Pyramid Model that will be illustrated in Chatper 5.

Application Content

Many traffic generators allow for the insertion of some kind of payload or content into packets. Some tools, like httperf [21] simply use standard application layer protocols to interact with a network application. The end result is generated traffic with specific content. Other tools like the Malicious trAffic Composition Environment (MACE) [22] use models to generate specific network behaviors; network attacks in this case. Overall, though, the generation of content is still very dependent on the application and an open issue. In fact, in the case of NIDS, most traffic is blindly considered either malicious (i.e. will cause a NIDS to alert) or benign (the NIDS will ignore that traffic). This point of view, however, is naïve as we will illustrate to more depth in the following Chapters.

1.3 The Role of Simulation in Network Traffic Generation

The line between traffic generation and simulation is quite subtle and blurring as more advanced traffic generators are created. The primary difference between the two is purpose. Traffic Generators create traffic on actual networks while simulators create traffic virtually. The blurring issue is that as traffic generators become more complex they slowly borrow more and more simulation techniques in the creation of traffic. Ultimately, traffic generators tend to target more specific features of traffic while simulators attempt to recreate all features of traffic.

1.3.1 Concerns with Live Traffic

A common tactic in traffic generation is to employ packet captures from a proprietary network, or from a publicly available repository [6, 23, 24], and replay these captures onto the testbed network with a tool such as tcpreplay [3]. While this tactic provides the simplest means of creating application-level content it fails in many respects. First, the traffic within the capture may contain sensitive information and require *scrubbing* in order to mitigate potential privacy issues. Unfortunately, *scrubbing* traffic traces is no simple feat and as privacy and security concerns continue to escalate the handling of the data requires great care. In essence, for privacy to be maintained, the traffic traces need to be anonymized. However, anonymizing traffic is a balance between maintaining the desired statistics of the traffic while removing or hiding potentially sensitive data. Policy-based anonymization schemes [25–28] have found popularity since they provide the ability to anonymize any feature of a traffic trace. These policy-based schemes allow users to balance the confidentiality required by the data owner with the needs of the application of the data. The user can filter-out (or filter-in [28]), z fields in each packet so that the data will be concealed or absent for these fields. In this manner, greater or lesser anonymization can be used to create greater or lesser confidentiality at the cost of the empirical value of the traffic trace. Unfortunately, the problem of identifying the sensitive fields is still an open issue. Thus, there is no guarantee that even if traffic traces are properly anonymized that they will be devoid of sensitive data. Further, it is difficult to quantify the exact impact anonymization will have on the ability of the traffic trace to adequately evaluate a given application.

Secondly, traffic regenerated by packet captures will exhibit behaviors and phenomenon existing when the traffic was captured but not necessarily realistic when regenerated in the testbed environment. While this does allow for the replay of peculiar or specific phenomena, it also severely limits the variability of the evaluation. In other words, every time the traffic is played, all packets will occur in the exact same sequence with the exact same headers and payloads with perhaps only the speed of regeneration changed. This can create inadvertent blind-spots as a particular network application might perform well against a commonly used traffic capture, but poorly against others. Traffic captures offer a good method for tuning an application to a specific environment [18], but are less well-suited to evaluating general cases.

Finally, live traffic or traffic captures are not always well suited to evaluating an application. For example, NIDS evaluation captures like the DARPA data sets [6], defcon capture the flag data sets [23] and the Army Information Technology and Operations Center (ITOC) data sets [24], as well as those from live networks, have an incredibly small amount of malicious traffic; less than 1% in this case. Further, the content of the traffic in these data sets does not intersect well with most rule sets for NIDS. Thus, nearly all packets are processed at optimal speeds and the only way to see how the IDS operates under pressure is to arbitrarily increase the transmission speed of the replay. Of course, increasing the transmission speed in this manner only serves to insert more bias into the evaluation. Thus, these captures offer no means of evaluating how the NIDS will act under strain from large numbers of alerts or if many packets nearly match rules.

1.3.2 The Balance Between Utility and Reality

Traffic generators need to mimic certain aspects of live network traffic depending on the ultimate purpose of the generator. As such, live traffic, or at least the statistical properties of the live traffic, typically serves as a base for most traffic generators. This is where simulation begins to take hold in traffic generation. Simulated traffic can overcome many of the privacy issues affecting live network captures. Further, since traffic generators typically target very specific needs, the simulation can target only the necessary features of evaluation. In this manner, a traffic generator can adopt, ad-hoc, simulation techniques to generate certain aspects of traffic and then simply employ either static, random, or live data to fill in other areas.

Complete simulation of networks often fails in a variety of ways, and has for some time as evidenced by the work of Paxson and Floyd [29] detailing the problems with simulating the Internet. However, the ad-hoc approach to simulation used in most traffic generators grants a large degree of freedom to blend real-traffic with simulation. The exact trade-off depends on the goal behind the specific traffic generator. For example, a traffic generator like the Security Assessment Simulation Toolkit [30] can generate traffic in a test-bed network such that the traffic will look as if it was generated by a large enterprise network. For such a lofty goal, it is necessary that many aspects of the traffic be simulated. Conversely, a traffic generator like Harpoon [31] attempts only to create statistically accurate flows between nodes and thus it need only simulate aspects such as the size of each flow, the packet length distributions, and inter-arrival times. In other words, depending on the goal of the traffic generator, it may be desirable to sacrifice certain aspects of reality in order to provide for more exhaustive evaluation.

1.4 Standard Methods for Traffic Generation

Before examining particular tools it is necessary to understand the three general methods used to generate traffic. The most popular method for generating traffic is to generate the traffic from the user-space using either sockets or a particular library that grants low-level access to network interfaces. This is easiest to program while serving most needs. When speed and timing become an issue, direct hardware implementations are common. Direct hardware implementations allow for maximum speed and precision timing, but are often difficult to use, lack adaptability, and can prove costly. Kernel-level packet generators offer a hybrid approach such that the traffic generator is closer to the hardware, and thus capable of higher output speeds, while incurring no additional cost over user-level tools.

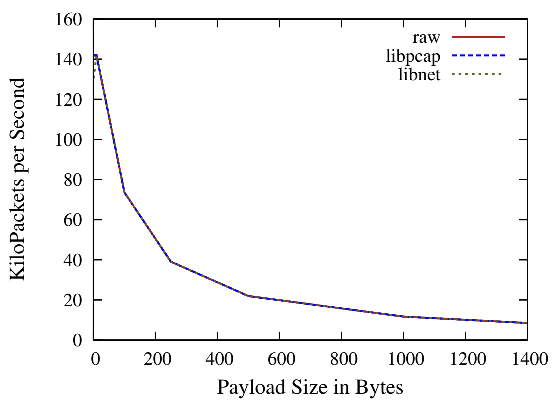
The primary motivating factor for using kernel or hardware techniques comes from the need for high performance in order to evaluate modern networks. A Gigabit Ethernet connection can, in a worst case, have nearly two and a half million minimum-sized packets (assuming zero length UDP packets with 8 byte Interframe Gaps) running across the wire. Any network application, a router for example, must be capable of handling that much traffic. This then motivates the creation of traffic generators that can create enough traffic to meet these needs. Ultimately, if user-space traffic generators were capable of meeting any link-speed and maintaining high precision timing then there would likely exist no hardware or kernel-level traffic generators.

1.4.1 User-space Traffic Generation

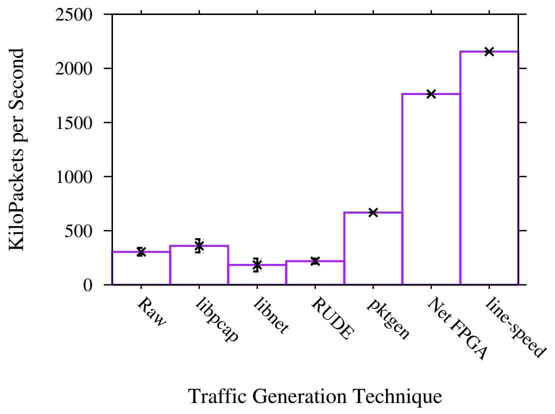
The easiest, and most common method for traffic generation is to build the tool in the user space and take advantage of the socket Application Programmer Interface (API) available for a given system. Using the socket API it is possible to simply open a TCP or UDP socket to a remote machine and expect valid protocol behavior as the kernel will take care of all the low-level management. This suffices for many traffic generation tools concerned primarily with evaluating a very specific application. For example, httperf [21] is a traffic generator that can create workload for a web server by automating multiple "GET" requests to that server. However, normal sockets are insufficient for creating aberrant or specialized packets. Further, normal sockets restrict the user to following only the predefined course of the protocols. A way around this is to employ raw sockets. Raw sockets, if supported by the operating system, allow the user to craft whatever traffic they wish and dump it to the NIC with limited processing of the packet by the IP stack in the kernel. This means that the user must handle all processing for the traffic, but also that the user can generate any kind of traffic.

Libpcap [32] (winpcap for windows [33]) is a popular wrapper to raw sockets that provides a user with relatively painless access to raw socket programming. While the API is somewhat cryptic, it is less confusing than working directly with raw sockets. As such, libpcap has become a de facto standard in network capture, analysis, and generation from the user space and is a primary component of popular network traffic applications like TCPDump [34], Wireshark [35], and Snort [14]. In fact, most scripting or programming languages have a wrapper class for libpcap like PacketX [36] which is an Active X wrapper for libpcap functionality, Net::Pcap [37] which provides access to libpcap for Perl, and jNetPcap [38] which provides a pcap wrapper for Java to mention just a few. Libnet [39] is a separate library that extends the functionality of libpcap to provide a host of tools for processing IP, TCP, and UDP traffic.

From a programming standpoint, user-space traffic generators are the most attractive as they are relatively easy to debug and there exist a wealth of information demonstrating how to program at this level. However, this ease-of-use comes at a cost in overhead from kernel-level stack processing and context shifts. The performance of the primary libraries (raw sockets, libpcap, and libnet) all perform identically as long as the network link is the bottleneck as is illustrated in



(a) Maximum packets per second by user-space technique and payload size (Fast Ethernet).



(b) Maximum packets per second by traffic generation technique (Gigabit Ethernet).

Figure 1.2: Maximum traffic output by technique.

Figure 1.2(a). These results depict the maximum output rate, in packets per second, by user-space technique given a stream of 1,000,000 udp packets, all of a set payload size. The results are the average of five runs, and even with this small number the difference in total packets per second between each API is less than a tenth of one packet per second. This implies that for fast ethernet any of these API serve equally well and are likely sufficient for most evaluation needs as the NIC is the bottleneck, not the CPU.

However, as the network link expands and reaches Gigabit or higher speeds, not only do the user-space libraries fail to meet link speeds, they also demonstrate the overhead present in each API. Figure 1.2(b) illustrates the maximum output capability of some user-space, kernel, and hardware methods for traffic generation. This data was gathered by generating 10,000,000 udp packets of 16 bytes in length under the particular tool or technique 10 times each and averaging the results. The transmission speeds exceeded the ability to capture the packets on the receiver (in most cases), so the timing and transmission counts of the sending machine were used as the primary factors for calculating the average packets per second for each test. These results are not meant to provide a definitive evaluation of these techniques as changes in hardware and/or operating system will directly impact the results. However, Firgure 1.2(b) does offer a relative comparison of the magnitude in maximum output capability of the various techniques. In Figure 1.2(b) libpcap is noticeable the most capable API for transmitting packets in the user-space. A primary factor distinguishing libpcap from raw sockets is the fact that raw sockets still calculate the Layer 2 and Layer 3 checksums for the packets while libpcap does not. This most likely accounts for most of the difference in performance between libpcap and raw sockets. However, the discrepancy between libnet and the other two user-space API is less clear. Libnet is a wrapper for libpcap, and it also implements checksum calculations in user-space. These two factors likely account for most of the disparity in maximum sending rate, though the overall coding of libnet may also be a culprit. Finally, Figure 1.2(b) clearly demonstrates that on a Gigabit link the API chosen will impact maximum performance. Further, the figure illustrates that user space implementations are a long ways from the roughly 2.15 million packets required to meet the link speed. At best, user-space traffic generators, in these tests, can fill only about $^1/_6$ of a Gigabit link.

1.4.2 Kernel-level Traffic Generation

Kernel-level traffic generators move the traffic generation directly to the kernel in order to improve maximum output performance. The lack of the context switches necessary to run a user-space application can improve the overall performance of the traffic generator in terms of maximum sending rates. However, if the operating system does not have a real-time patch and some method for maintaining accurate time (i.e. a GPS clock), the timing precision at the kernel is only slightly better than that from the user-space. Further, kernel programming is more complex than coding in user-space. As such, only a few traffic generators are written for the kernel. If performance is of sufficient concern most traffic generators will migrate to hardware where both timing and high throughput can be maintained. Kernel-level traffic generators are thus more of a "poor man's" attempt at a high-performance traffic generator.

In Figure 1.2(b) RUDE [40] and pktgen [41] are two kernel-level generators used to illustrate the maximum performance of utilizing the kernel. The test is the same as described in the previous section. Uniquely, RUDE performed quite poorly in this test, not even as good as the user-space API. RUDE maintains sequence numbers and byte counts for each packet and adds this to the data of each packet so that the client CRUDE [40] can capture the stream and perform some calculations such as delay, jitter, and bandwidth. This extra overhead likely accounts for some of the performance issues, but does not answer them all. Conversely, pktgen, which is a linux kernel module, performed as expected exhibiting nearly twice the maximum sending rate of the best user-space traffic generation. Figure 1.2(b) demonstrates that kernel-level generation can greatly outperform user-space modules as well as the fact that simply migrating to the kernel does not guarantee improved performance. Still, even at these speeds, the kernel-level generators only filled about $^1/_3$ of the Gigabit link.

1.4.3 Hardware Implementation

Almost all high-end, commercial traffic generators come bundled with specialized hardware. There are many reasons for this. First, hardware implementation, when synced to a GPS clock, offers the best timing constraints for traffic where timing is an issue. This allows for distributed synchronization of traffic when multiple pieces of hardware are employed. Further, hardware offers the best platform for generating high workloads. However, the complexity and expense of creating a hardware-based traffic generator has resulted in very few researchers tackling this kind of project with the notable exception being the NetFPGA project traffic generator [42]. As such, it is often difficult to understand exactly what features these high-end traffic generators support aside from actually purchasing one of the devices and evaluating it.

The maximum line rate for a Gigabit Ethernet link under the test as evaluated in Figure 1.2(b) is roughly 2.15 million packets per second. Many commercial hardware traffic generators boast these speeds however there can be substantial variance in how a traffic generator's speed is calculated. For example, the NetFPGA traffic generator [42] is a hardware traffic generator based on the Virtex II NetFPGA board. The Virtex II board offers four 1 Gigabit Ethernet ports and a Field Programmable Gate Array (FPGA) so that it can be customized to perform as a user desires. The NetFPGA traffic generator simply takes a packet capture, reads as much of the capture that will fit into the buffers on the NetFPGA board and then will cycle through the uploaded packets sending and resending those packets until it has reached a designated number of packets sent. Their evaluations boasted complete filling of the Gigabit link. However, their evaluations used unequal packet sizes and reported Mbps rather than packets per second. We extrapolated based on the data provided in the paper and determined that given their test conditions they only managed about 1.76 million packets per second. This does not necessarily mean that the NetFPGA platform cannot meet full link capacity, only that truly saturating a link is difficult even for hardware platforms. Regardless, the comparison in Figure 1.2(b) is mostly meant to put into perspective the differences between the various traffic generation techniques.

1.4.4 Distributed Traffic Generation

Another solution to the problem of maximum output is to simply create a distributed architecture. If enough traffic generators are cobbled together then aggregate traffic can easily meet even very high link speeds. In essence, this is exactly what occurs in a Distributed Denial of Service attack. Several traffic generators adopt this very approach like the Distributed Internet Traffic Generator (D-ITG) [43] and the Security Assessment Simulation Toolkit (SAST) [30]. Of course, distributed traffic generators are not as attractive as single-box solutions as they require extensive setup, suffer from synchronization problems, and also require more resources (i.e. more machines). Still, they are often cheaper than full hardware solutions.

1.5 A Catalog of Current Network Traffic Generation Tools

Most traffic generators result as a byproduct of unrelated research or business need more so than as a the subject of research into traffic generation. As such, a large disparity exists amongst the various tools. Categorizing tools by features becomes somewhat impractical as many tools share a variety of features and almost all tools have one or two features very specific to a particular task. Instead, we build categories derived from the overall usage of the tool. Since the success or failure of most traffic generators depends so heavily on the targeted functions we feel that examining the level at which the tool functions as the primary factor in describing its usefulness. Thus, we have created five categories to broadly contain all traffic generators, each category defining the general level of network communication that the tool creates. As can be seen, each higher-level layer extends those below and adds simulation features to the mix.

- Packet-level Tools: These represent tools designed to craft very specific packets that may cause certain boundary conditions in a targeted application (i.e. buffer overflow). The key aspect of a Packet-level Tool is that it operates at the packet level creating single packets, even if the tool is capable of creating many such packets. Scripts are used to add further intelligence or functionality, but the tool alone will only spit out packets based on a set of parameters.

- Flow-level Tools: Flow-level tools create bursts of packets with a given set of characteristics from Poisson inter-arrival times to long-tailed packet-length distributions. These flows are unidirectional, are typically UDP, and are often used in creating large amounts of traffic (i.e.

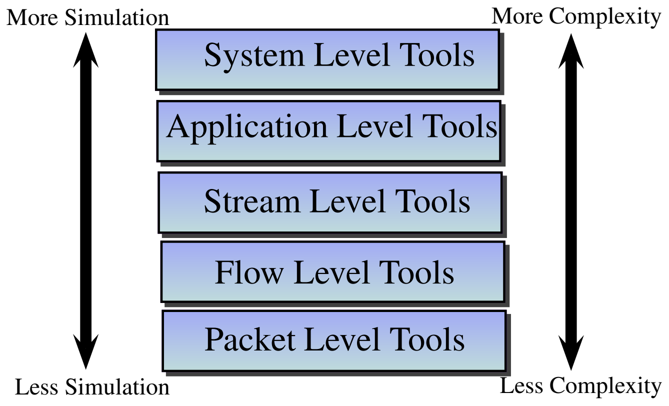


Figure 1.3: Simulation features and complexity.

load generators).

- Stream-level Tools: Stream-level tools are similar to Flow-level tools in most respects, except that an attempt is made to provide for bi-directional traffic. Further, Stream-level tools typically include basic user models such as the ubiquitous “think time” pause between bursts.
- Application-level Tools: Application-level Tools preserve all lower level semantics and attempt to add application specific behaviors. The complexity of Application-level Tools is much greater than lower-level tools as a much greater need of simulation is required.
- System-level Tools: System-level tools attempt to completely recreate networks. Such tools rely either upon actual infrastructure bound together through some sort of middleware, or upon heavy simulation to create a synthetic or virtual presence. Such traffic generators may even have enough intelligence that participants can interact with the system.

1.5.1 Packet Level Traffic Generators

Packet Level traffic generators typically grant users the ability to craft any kind of network packet they wish whether valid or invalid. These tools are sometimes referred to as packet crafting tools.

Even though some of these tools can create long bursts of crafted packets, the focus is at the packet level and any resulting streams are more of an added feature. A good example of this is ipgen [44] which is an open source tool that employs Raw Sockets to send TCP, UDP, or ICMP packets. It offers little functionality over designating the size, number, and protocol of packets. While ipgen does actually create flows, it makes no attempt to maintain any bookkeeping and is ultimately just sending a large number of single packets without regard for any larger context. Other similar packet crafting tools include: the Packet Generator [45], a libnet based tool for custom packets; packETH [46], a tool specializing in crafting Ethernet frames though it can also craft IP packets; packit [47] a tool that allows for real-time injection of packets; Packet Excalibur [48], a tool that can both receive and send custom network packets; GSpoof [49], another packet crafting tool capable of manipulating Ethernet, IP, and TCP headers; the Generator and Analyzer System for Protocols (GASP) [50], that allows for creating custom packets and scripting how these packets are utilized; and finally SendIP [51], which allows for the editing of a variety of protocol headers as well as payloads.

A second class of packet level traffic generators are the work of hackers who want to create packets that can cause network applications to fail. These tools have since fallen into the realm of security. Tools like nemesis [52], Scapy [53], hping [54], and IP Sorcery [55] all maintain not only the ability to craft individual packets, but to also script attacks, enact port scans, and other such nefarious activities. The IP Stack Integrity Checker (ISIC) [56] is another packet crafting tool that can be used to test a variety of network conditions. Somewhat differently, ISIC can create pseudo random packets boasting randomly generated values across various IP fields in order to test a broader range of circumstances.

1.5.2 Flow Level Traffic Generators

Flow level traffic generators produce a continuous flow of packets, typically UDP. Efforts are made to ensure that the flow is correct, but no effort is made to consider any potential response. In other words, efforts are made to ensure that each packet is well formed, that the general properties of the flow of packets are correct in timing and packet lengths but there is no attempt to account for interactions between nodes. The uni-directional nature of UDP makes it a natural choice for flow level generators. Further, for evaluations concerned primarily with number of packets, inter-arrival times, or packet length distributions, then flow level generators are the simplest and most effective tools available. Essentially, for such research it is possible to create two uni-directional flows, each matching the flow-level statistics for both directions of traffic as might be seen at some point in a live network.

The Multi-Generator (MGEN) [57] produced by the United States Naval Research Laboratory is one of the more robust flow generators. MGEN supports a large amount of scripting which enables this tool to serve a wide variety of roles. The Poisson Traffic Generator [58] allows for the creation of a flow of packets such that the inter-arrival times match a Poisson process for some average rate.

The other common use for Flow Level traffic generators is to fill a link with packets. This is done for purposes of evaluating bandwidth, delay, and jitter. As such, the timing of generated packets is crucial as is the ability to send high volumes of traffic. As such, several kernel traffic generators fill this role. The Real-time UDP Data Emitter (RUDE) and the Collector for the Real-time UDP Data Emitter (CRUDE) [40] allows for generation of flows to meet very tight timing constraints. With RUDE running on one node and CRUDE running on another (both in kernel) it is possible to create high volume streams with accurate timing and garner good measurements of bandwidth, delay, and jitter on a link. The Kernel-based Traffic Generator (KUTE) [59] is another generator like RUDE. It claims to be able to generate higher send rates than RUDE but it is not well maintained and we were unable to get it to work with modern Linux kernels. Finally, the Linux kernel module pktgen [41] allows for direct access to the NIC and sending of packets. It is capable of generating a high volume of traffic, but does not have a ready-made client for capturing the data.

In user-space, there are dozens of tools that meet this same purpose of bandwidth, delay, and jitter calculation. Tools like Mtools [60], iperf [15], packgen [61], Netperf [62], which is specifically designed for emulating bulk data transfers, Jugi's Traffic Generator [63], which boasts high precision timers, the Traffic Generator [64], which has a server for logging the results in a distributed manner, and TFgen [65], which is also capable of sending bursty traffic. Another unique flow-level tool is mxtraff [66]. In particular it can create TCP and UDP streams to emulate mice (small, intermittent TCP flows), elephants (large, continuous TCP flows) and dinosaurs (constant and continuous UDP flows). NetScan Pro [67] offers commercial tools for generating packets for bandwidth calculations, modifying header and payloads, as well as scripting specific interactions.

1.5.3 Stream Level Traffic Generators

Stream level traffic generators attempt to maintain the interactive behaviors between bi-directional flows of data. In other words, it attempts to better mimic the query and reply nature of actual communications. Properly generating stream level traffic requires some level of closed-loop network or distributed environment. In principle, network devices tend to communicate in conversations. Thus, traffic generation should also occur in terms of communications where both ends of the conversation can affect the nature of the communication. Further, enough book-keeping is maintained such that communications are valid. For example, TCP sequence numbers must be accurately maintained. In many respects, stream level traffic generators are simply two flow level generators that are synchronized. Harpoon [31,68] was one of the first generators to truly embrace the idea of stream generation. The Distributed Internet Traffic Generator [43] and Netspec [69] are distributed systems for generating traffic. They offer a synchronized set of flow-level generators that allows for the emulation of stream-level behaviors. Bookkeeping and management information is passed through separate data channels requiring two separated networks, one to house the generated traffic and the other to pass logging and test management information.

1.5.4 Application Level Traffic Generators

Application level traffic generators attempt to create traffic that will affect a particular application. In many instances, this is simply using given protocols as is exhibited by httpef [21] which simply connects to a web server and downloads a given web page. Geist [70] is a more advanced tool for stressing web and e-commerce sites. It offers the ability to finely tune the request process to meet a variety of traffic statistical properties, thus providing for much more extensive evaluations. Netcat [71], on the other hand, referred to as the "Swiss Army Knife" of network utilities, offers a wide variety of simpler network options from scanning systems, to acting as a proxy, to file transfer. As such, it is often commandeered as a general purpose tool for generating traffic with servers and can stand in to test not only web servers but nearly any network application. The Scalable URL Reference Generator (SURGE) [72] is a slightly different twist on generating http workloads where the size of downloads is considered in creating a distribution of web requests to match that seen in real life. Finally, Tsung [73] is an advanced tool for evaluating not only web servers but most major servers. Tsung operates by simulating users. Each user will act at a certain application layer (i.e. HTTP, MySQL, etc) and with specified behaviors. Traffic is then generated as multiple users interact with an in-place system.

Some other Application-level traffic generators will actually implement scripted scenarios, such as an attack on a system. MACE [22] is a tool to generate malicious traffic employing the meta-sploit framework. MACE is specifically targeted to NIDS evaluation. Essentially, attacks are described using the meta-sploit framework. Once an attack is specified, it can be started at any time. Once triggered the attack can advance through multiple actions and even applications to accomplish an attack. The end result is realistic attacks. A less sophisticated attack generator is the blackbox IDS stimulator [74], a tool that uses the rules of a NIDS to generate packets to specifically trigger alerts in a NIDS. While the blackbox IDS stimulator is less realistic than MACE, it actually offers a better method for benchmarking NIDS and formulates some of the core of the research in later sections of this work.

Most commonly though, Application-level traffic generators are simply replay tools that re-generate traffic from a capture of live traffic. Playback is a very cheap way to emulate application and even system level behaviors. However, playback has several limitations. First, the generated traffic is largely fixed to the content of the network capture. Thus the only control over replay traffic is the rate of speed with which packets are sent. Further, phenomena that exist in the capture will be replayed in the generated traffic even if there is no reason. This is, more often than not, a blessing rather than a burden as it allows evaluating a network application under very specific conditions. However, it also limits the usefulness of the traffic for exploring a range of circumstances. Unfortunately, the true weakness of playback is that it is not responsive. In other words, packets will always be sent in the recorded order even if the network upon which the traffic is generated is failing in some ways. Some playback generators attempt to limit this problem by providing better book-keeping and dividing the data between multiple processes in order to maintain better responsiveness. Ultimately, playback is a low-effort method for generating a fair approximation of the state of a network at one point in time. The most popular playback tools are: tcpreplay [3], bitwist [75], ostinato [76], TCPivo [77], which boasts higher precision timers for greater precision in sending, and netsniff-ng [78], which is a suite of tools for capturing and replaying network captures that is optimized to have zero copying in order to improve playback performance. Swing [79] is a closed loop replay with the ability to dynamically respond to introduced network behaviors. Swing also makes use of the emulab framework and thus utilizes strong simulation components.

Further, there are numerous commercial solutions to improve the performance of playback. NPulse Technologies offers the Hammerhead Packet Capture solution [80], a hardware solution for capturing and replaying packet captures capable of 20 Gbs. Fluke networks offers the ClearSight [81] software for traffic analysis, capture, and replay, including the ability to recreate and replay VOIP and MPEG. Absolute Analysis Investigator [82] is a commercial hardware-based traffic generator capable of injecting packets at up to 10 Gbs speeds. It is also capable of injecting errors, analyzing traffic, and other similar functions. Omnicor offers several tools [83–85] to create traffic for evaluating protocols, reaction to simulated impairments, and to test GPS synchronization and timing.

1.5.5 System Level Traffic Generators

System level traffic generators are a hybrid of simulator and traffic generator. They employ a large amount of simulation from creating virtual users and user patterns to virtual, sometimes evern real, servers. System level generators depend heavily on a hardwired infrastructure as well as the use of virtualization. The goal of these traffic generators is to recreate a network environment such that real users viewing this traffic would assume it to be real. To facilitate this, these systems combine real services with the simulated users. Thus, at some level, real users can actually interact with the system. One of the first such systems was the Lincoln Adaptive Real-time Information Assurance Testbed (LARIAT) [86] which was the testbed framework used to develop the 1998 and 1999 DARPA Intrusion Detection Evaluation data sets [6,7]. The Security Assessment Simulation

Toolkit [30] is a distributed simulation environment employing multiple nodes and network infrastructure to recreate entire working networks in a test-bed environment. The Real-time Immersive Network Simulation Environment for Network Security Exercises (RINSE) [87] attempts to provide a full simulation environment, similar to SAST, that can demonstrate attacks and which can respond to user actions. RINSE relies on more simulation than SAST. The Scalable and flexible WORkload generator for Distributed Data processing systems (SWORD) [88] employs decision trees to simulate a wide array of communications, including voice. Spirent Test Centers [89] offer a high-performance infrastructure for testing network scenarios. The packetstorm network emulator [90] and BreakingPoint Systems [91] are commercial, high-performance, hardware tools from Phoenix Datacom for emulating a variety of network conditions and phenomena. Similar commercial solutions are ByteBlower [92] and LANforge-Fire and LANforge-Ice [93].

CHAPTER two
DEFINING THE PURPOSE AND SCOPE OF NETWORK INTRUSION DETECTION SYSTEMS

Network Intrusion Detection Systems (NIDS) are devices used to detect the presence of anomalous or malicious traffic occurring on a particular network. NIDS employ two general techniques for segregating traffic: Anomaly Detection or Deep Packet Inspection (DPI). In fact, most modern NIDS employ both techniques at varying levels. For example, the open-source NIDS Snort [14] employs anomaly detectors that examine application layer protocol behaviors and alert when a protocol is abused. However, the primary function of Snort is its rich Deep Packet Inspection engine which examines the payload of each packet and can alert when specified content is identified. Bro [94] is another NIDS that is capable of a variety of separate analyzers that operate modularly and can alert on packet payload (i.e. DPI) or on behavioral aspects of traffic (i.e. anomaly detector).

Ultimately, the goal of any Network Intrusion Detection System is to identify network traffic that intends to abuse the system in some manner. Unfortunately, network traffic is anonymous, nearly any aspect of traffic can be forged, and is either too localized (i.e. a single packet) or too global (i.e. all traffic). This greatly complicates the process of determining the intent of any traffic. Since a network can have thousands of communications operating simultaneously, it is a difficult problem to definitively identify abuses of the system. As a result most NIDS, regardless the detection method used, merely serve to identify some of the anomalous or malicious traffic present in the network. In this vein, NIDS really only offer improved situational awareness of the network to network operators. Adding more detectors of varying levels will increase the view provided by the NIDS, but will increase the resources required to operate and interpret the NIDS. This increase in resources may out-pace the value of the information added. Thus, properly tuning a NIDS to an environment is a very important aspect of NIDS management and is one of the motivating factors of this work as well as the work of Dreger et al. [95], Sommers et al. [18,31], and Becchi et al. [96].

2.1 Anomaly Detectors

Anomaly detection employs some metric of traffic, like the total number of packets passing through a particular node, and performs some algorithm on this metric to arrive at a decision as to the intent of a particular subset of traffic. Qu et al. [97] described several layers of measurements with which to describe network traffic for anomaly detection. They start with measurements such as the number of incoming Protocol Data Units (PDU), number of outgoing PDUs, ARP request rate, and the invocation rate. They further propose applying similar ideas of incoming and outgoing PDU to applications like email services to detect a sudden increase in emails. Similarly, Lakhina et al. [11] employ Principle Components Analysis (PCA) on packet counts across distributed nodes in a wide area network. PCA provides the ability to drill down, iteratively, through the most common subspaces of traffic to arrive at the rarer subspaces. Their observations demonstrated that normal network traffic has relatively low dimensionality. In other words, common network traffic falls into the first four subspaces when applying PCA. Thus, any phenomena occurring below the first four subspaces might be considered anomalous. Ultimately, an anomaly detector employs one or many such metrics and then some sort of filtering or aggregating algorithm to exacerbate anomalous traffic in order to make it easier to spot. Finally, all anomaly detectors must employ some kind of threshold where the line is drawn between the normal and the abnormal.

There are a wide range of anomaly detectors from the detection of P2P networks [13] to defending against Distributed Denial of Service attacks [98]. The primary advantage of most Anomaly Detectors is that they can adapt to changing conditions. This adaptation occurs either as a bi-product of the algorithm used or as a specified learning process. Unfortunately, thresholds are never perfect and all anomaly detectors incur some level of error such that either non-anomalous events are classified as anomalous or vice versa. Further, Barreno et al. [99] have shown that anomaly detectors that employ machine learning can be trained in a manner opposite the goals of the detector. As such, there exists, as yet, no single anomaly detector capable of securely and accurately identifying all anomalies or abuses.

Generating traffic to evaluate anomaly detectors is something that can be handled with most current traffic generation tools. Since most Anomaly Detectors look only at packets as a whole, or just values within the headers of the packets, employing currently available traffic generation tools is sufficient to evaluating most Anomaly Detectors. Further, since Anomaly Detectors primarily rely on the aggregate statistics of traffic then it is possible to work with just the packet headers from live captures, or even just the statistics, to create traffic that will statistically approximate real-world conditions and provide a good idea as to the actual usefulness of a particular Anomaly Detector. Essentially, there exist many methods for generating traffic to effectively evaluate Anomaly Detectors. The same, however, cannot be said for Deep Packet Inspection engines.

2.2 Deep Packet Inspectors

Deep Packet Inspection (DPI) is composed of two primary elements: a matching engine and a rule set. The matching engine organizes the rules, such that only applicable rules are matched against any given packet. For NIDS, this organization, or categorization, is most often performed by grouping rules by the Internet Protocol (IP) flow-level quintuple (Protocol, Source IP address, Destination IP, Source Port, and Destination Port). Thus, each rule not only specifies certain bit strings that must match, but also specifies flow-level features such as destination port and IP Protocol. During operation, the matching engine scans the entire payload of each packet in some traffic stream and compares that content to all applicable rules. If no matches are found then the packet is considered benign. If one or many matches are found then one or more alerts are logged and potential additional actions taken.

The primary advantage of DPI is that it makes a binary decision without need of any threshold. A packet either matches one or many rules or it does not. In this manner it is possible to specify detailed parameters defining a particular signature in order to target very particular traffic. This strength is also a weakness of DPI in that poorly specified signatures will flag traffic that is not malicious or anomalous. Further, DPI is burdened by its signature set in two ways. First, if some malicious traffic has no rule within the signature set, then there is no means by which that traffic might be detected. Secondly, if a rule set is quite large then it can become a burden in resources and degrade the performance of the matching.

2.3 NIDS in the Real World

In perfect situations, the DPI engines and Anomaly Detectors will boast extremely high accuracy in identifying malicious and anomalous traffic. Unfortunately, in the "Real World", to gain a broad picture of the state of the network a NIDS must employ multiple engines. DPI engines are good at catching known exploits while Anomaly Detectors are good for identifying potentially new exploits. Thus, a holistic defense requires running several detectors including at least one DPI engine on network traffic. However, as illustrated by Dreger et al. [95] the load on a NIDS grows linearly with the number of detectors. Thus, it is necessary that a NIDS be properly evaluated in order to identify optimal operating parameters. As mentioned earlier, many of the currently available traffic generators are sufficient to evaluate Anomaly Detectors. However, for DPI there exists no specific tool that can adequately perform this task as will be further outlined in this chapter.

2.4 Common Metrics for Intrusion Detection Systems

There exist three primary metrics employed when evaluating NIDS. These metrics are the False Positive rate, the False Negative rate, and the throughput of the system. Some other commonly used metrics will employ the CPU usage of the NIDS hardware to demonstrate the amount of effort required to process traffic and even the time to update, train, or run a particular detector. However, these metrics derive more from Anomaly Detection than DPI. As such, the terms do not necessarily capture the full nuances of the different detection mechanisms. Thus, we detail the merits and demerits of these metrics.

2.4.1 False Positives

A false positive occurs when the NIDS alerts on traffic that is neither malicious nor anomalous. For Anomaly Detectors this occurs either through natural network phenomenon that closely mimics behaviors in anomalous traffic or due to overly stringent thresholds. As such, false positives are a significant factor for deciding on the suitability of a particular Anomaly Detector as large numbers of false positives can hide true positives in a sea of noise and greatly reduce the efficiency of the NIDS overall. However, for DPI technically there are no false positives. If a packet matches a rule in the signature database then the DPI engine must alert on that packet. Thus, the cause of false positives is the rule set of the DPI, not the DPI engine itself. This may seem a trivial distinction, but it has a couple significant implications. First, it illustrates that DPI engines are heavily influenced by the inputs (i.e. the rule set and the payload of the traffic). This implies that lack of false positives really offers very little evidence as to the quality of DPI. In other words, lack of false positives could be the result of a poor rule set, not due to any efficiency in matching. Second, a false positive is a similar burden to DPI as a full alert. Since DPI engines examine every packet, and since the overhead in processing a packet that causes an alert is vastly greater than packets that match no rules, false positives inhibit DPI greater than most Anomaly Detectors. Finally, most DPI engines produce many duplicate alerts since each individual packet is examined (i.e. an alert is produced for each packet). As such, a DPI engine has the potential to produce far more false positives than an Anomaly Detector. Various off-line alert processing tools and research exist to reduce the impact of such duplicate processing, but overall DPI engines are prone to generating large lists of alerts with many duplicate alerts.

Ultimately, this implies that treating false positives the same for DPI as Anomaly Detectors is not necessarily practical. First, tuning an Anomaly Detector simply requires adjusting a threshold or applying more filters. Tuning a DPI engine requires rewriting the rules in the rules database which is a difficult problem by itself. Certain automatic signature generation techniques like those employed in the automatic worm signature generators Earlybird [100], Hamsa [101], and Polygraph [102] offer techniques for automatically building signatures less prone to false positives. However, the evaluation of the rule set occurs separate from the evaluation of the DPI engine. After all, the DPI engine must be able to match any rule to a potential packet. Essentially, for DPI false positives are demerits on the rule set, while the ability to process packets is the more important feature of the DPI engine.

2.4.2 False Negatives

A false negative occurs when malicious traffic passes the system without generating an alert. For Anomaly Detectors this typically means that the anomalous traffic was too close to other traffic and not distinguishable beyond the threshold. For DPI, this can result in two ways. First, if no signature exists for a malicious packet, then DPI cannot detect that packet. Secondly, if the DPI engine does not have sufficient resources then as the volume of traffic increases the DPI engine will become overloaded. In this instance, some packets will be dropped from evaluation and by-pass the system which can result in malicious packets circumventing the DPI. Such behavior is common to network devices and represents queue exhaustion where no space exists in the queue to accommodate arriving packets (i.e. drop-tail queue). Certain attacks such as the algorithmic complexity attack by Smith et al [103] make use of this feature to harm DPI engines. Essentially, the algorithmic complexity attack creates packets that target specific rules of the target DPI engine. The goal is to make the processing of these crafted packets cause the DPI engine to exert exponentially more effort to arrive at a decision. Since NIDS are designed to passively monitor networks a discarded packet is not removed from the network but only from evaluation by the NIDS. Thus, an attacker can sandwich an actual attack between load causing packets. As the DPI engine becomes overburdened, it is likely that the actual attack packets will be dropped from evaluation and pass into the system unnoticed. Regardless, both cases represent examples of False Negatives. The former case depends entirely on the rule set and is only indirectly considered in this research as signature generation is its own complete avenue of research. The latter case, however, depends on the ability of the NIDS to process traffic, which is a primary quantity for evaluating NIDS.

Once again, False Negatives have a slightly different connotation for DPI engines over Anomaly Detectors. First, a false negative represents a failure of an Anomaly Detector similar to a false positive. While, once again, the false negative is typically a symptom of the rule set for a DPI engine. However, when a DPI engine becomes resource starved and begins dropping packets, then the idea of false negatives becomes a question of throughput.

2.4.3 Throughput

Throughput represents the maximum amount of traffic that the DPI engine or Anomaly Detector can handle and still function accurately. For Anomaly Detectors, this metric is often overlooked. The reason is that many Anomaly Detectors make use of simple counters and do not require a lot of per-packet processing. Further, these Anomaly Detectors typically employ aggregate statistics and thus are not overly concerned with single packets. DPI engines, however, must process each packet in relative isolation resulting not only in a large amount of redundant effort, but also a large amount of effort per packet. Thus, the primary question of throughput for most Anomaly Detectors is whether or not the anomaly detection algorithm can execute in sufficient time to react to the traffic. For DPI engines each packet must be matched, in order, as quickly as possible. Failure to match at the arrival rate of packets will constrain the proper function of the DPI engine. Finally, the term throughput is often under-defined for DPI engines. As has been mentioned, the inputs to a DPI engine are of paramount importance. Throughput measured under one set of inputs may have no correlation to throughput measured under another set of inputs. Thus, it is necessary to have a clear method for systematic evaluation of DPI.

2.4.4 Mean Decision Time

In order to properly account for the concerns noted earlier, it is necessary to apply a more holistic metric to DPI engine evaluation. For ease, from here forward all reference to NIDS imply DPI engines specifically unless otherwise stated. To achieve this, we propose the metric of Mean Decision Time. The efficiency of a NIDS follows closely to Little's Law as applied to queuing theory. Essentially, Little's Law is a basic accounting function such that $N_Q = \lambda W$ where N_Q indicates the number of packets waiting in a queue, λ the average arrival rate of new packets to the system, and W the wait time in the system. Ignoring the fact that Little's law assumes infinite queues, it is evident that as λ, W, or both increase, so too does N_Q. For NIDS, which are governed by finite queues, this implies that packets are dropped from the system if they arrive while N_Q has reached some threshold. Thus, the measurable performance of a NIDS is directly related to the amount of time it takes to examine a packet as well as the properties of the arrival rate of packets. The impact of the arrival rate on network devices has been well studied and will not be examined here. The time to process a packet, however, is independent of the arrival rate and depends largely on the content of a given packet.

If the content of the packet is closely related to one or many rules then it will take longer for the IDS to make a decision. On the other hand, if the content of the packet holds little or no intersection with the rule set then the NIDS can shallowly process each packet and quickly arrive at a decision. Essentially, as traffic content intersects to greater depths with the rule-set, then the time to process each packet increases. As the processing for each packet increases then so too does the wait time which causes the queue to grow along with the likelihood of packet drop and the chance for False Negatives. Since the arrival rate is largely outside the control of the NIDS, this processing time serves as the primary indicator of growing or falling wait times and ultimately the performance of the NIDS. We term this processing time the Mean Decision Time (MDT) and figure it as the average amount of time required for the NIDS to process a packet and arrive at a decision: malicious or benign.

The primary focus of Chapter 4 is creating content in traffic such that it becomes possible to witness trends in MDT even when alerts are not generated by the targeted NIDS. In fact, attacks on NIDS, like the algorithm complexity attack by Smith et al. [103], are a perfect example of increasing the MDT of NIDS beyond acceptable parameters such that the NIDS is rendered useless. As such, true evaluation of NIDS requires an examination of the MDT. Under such evaluation it becomes possible to specifically identify failures in the DPI, or potential rules that cause too much effort.

2.5 Effectively Evaluating Deep Packet Inspection Systems

Unfortunately, current evaluation methodologies for NIDS do not sufficiently account for the factors that impact the MDT of NIDS. Replaying traffic captures from a proprietary network, or from a publicly available repository [6, 23, 24], with a tool such as tcpreplay [3] or some other network playback tool is a common tactic. The problem with this approach is that it rarely offers a good view of the MDT for the NIDS. Most captures like [6, 23, 24] and those from live networks have an incredibly small amount of malicious traffic. To illustrate, we examined four distinct data sets to identify trends of malicious traffic. The data sets are spaced across a period of ten years and offer a view of attack trends both within each data set as well as global trends across data sets and time. The first data set we employed consisted of the first 7 weeks of the 1998 DARPA data set, as provided by Lincoln Laboratories [6]. Our second data set was collected locally at the Network Research Laboratory at Washington State University where it listened at the gateway between the lab and the University network. This data set comprises full-day captures across four weeks beginning in February 2008 and extending into March 2008. The third and fourth data sets were taken from the Defcon 11 and Defcon 17 Capture the Flag events respectively [23]. In all, the data comprised roughly 31 GB and is detailed in Table 2.1. The data was collected by running Snort [14] against the data sets in off-line mode and aggregating the results. The rule-set was the registered-users public rule-set as provided by the Sourcefire Vulnerability Research Team for November ninth 2009 [104]. As can be seen from the table, the number of alerts compared to the amount of total traffic is quite small, .0827% of the total, even amongst these relatively attack-rich captures.

More significant, the content of most live captures does not intersect well with most rule sets. Thus, nearly all packets are processed at optimal speeds and the only way to strain the NIDS is to arbitrarily increase the transmission speed of the replay. Of course, increasing the transmis-

Table 2.1: Breakdown of common NIDS evaluation data sets.

Data Set	DARPA	WSU—NRL	Defcon 17	Defcon 11	Totals
Total Packets	41,323,968	30,588,839	39,034,601	1,307,975	112,255,383
Total Flows	4,263,540	265,492	5,015,713	189,006	9,733,751
Total Alerts	86,183	307	89	6,345	92,924
Flows with Alerts	5,202	249	76	2,562	8,089

sion speed in this manner only serves to insert more bias into the evaluation and still results in overly optimistic views as the MDT remains unchanged, only the arrival rate is being manipulated. Finally, as discussed earlier, live captures require *scrubbing* in order to arrive at a *ground truth*, if such can be attained, and after such scrubbing may no longer contain any interesting phenomena.

Other techniques of evaluation have employed NIDS stimulators, like [74], which can arbitrarily inject malicious packets into a traffic stream. While such stimulators serve as necessary tools for creating quantifiable malicious traffic, they only provide evaluation of the extremes: benign or malicious. Using a set of tools such as MACE and Harpoon together [22, 31, 68] offers the ability to generate background traffic targeted to some environment and malicious traffic that extends beyond simple stimulus packets. However, this system still does not provide a framework for systematically evaluating the MDT of the NIDS. Further, traffic trends and patterns tend to evolve and thus a traffic capture at one point in time may prove utterly inadequate for evaluating NIDS. Finally, traffic may fit a NIDS rule set without ever triggering, or entering, entire sections of the NIDS code or rules in the rule set. Conversely, traffic may also have no intersection with traffic, thus allowing the NIDS to operate at maximum efficiency. Essentially, the composition of the traffic and the rule set will determine the performance of the NIDS and if either is not considered in the evaluation then there exists the potential for considerable bias in the results of the evaluation.

A final tactic is to employ completely synthesized traffic such as that generated by NESSI

[105], ns-2 [106], or ns-3 [107]. Such simulators are very capable at simulating network conditions and phenomena, but lack functionality for generating payloads in any systematic way. Thus the same problems still remain in that most packets will fall either into the benign category or malicious category with very little intersection, in general, between the content of the packets and the rule set. Further, generating content synthetically creates either very simple traffic (i.e. random or static characters), or pulls content from a sample set that can require extensive labor and is prone to misrepresentation.

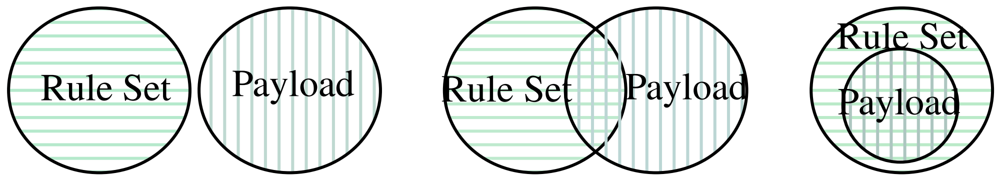
Some research has attempted to address this deficiency. Dreger et al. [95] demonstrated that NIDS resource usage climbs linearly with the number of detectors employed. Essentially, if the NIDS employs multiple, pipelined, detectors then the more detectors added, the greater the MDT and memory required to match each packet. A detector in this instance is a particular traffic analyzer that examines a specific set of conditions and alerts within tightly defined parameters as is demonstrated in the publicly available NIDS Bro [94]. In other work, Vishwanath et al. [2] illustrated how the generation of background traffic can have significant impact on network applications. Sommers et al. [18] also demonstrated how high mixes of malicious traffic can severely impact the function of a NIDS. However, these works have yet to arrive at a model that can fully explore the impact of traffic composition upon NIDS, a feat we attempt in the following chapters..

CHAPTER three
CONTENT GENERATION FOR DEEP PACKET INSPECTION

3.1 The Impact of Rule Sets, Packet Payloads and Background Traffic

As has been mentioned earlier, the inputs to a NIDS are comprised of a rule set on one hand and the traffic to be evaluated on the other. The exact composition of either of these can have significant impact on the efficiency of the NIDS. For example, a rule set filled with many overly broad rules will alert on any packet seen in traffic and thus rapidly become resource starved. Conversely, a NIDS with an overly restrictive rule-set may prove incapable of finding any malicious traffic but will at least be capable of doing so near line speed. Similarly, the traffic into the system can be manipulated to cause greater load on the NIDS such as demonstrated in the Algorithmic Complexity Attack [103] where time consuming regular expressions in a NIDS rule set are targeted to over-exert the system.

In order to properly evaluate modern NIDS, in particular, it is necessary to understand the relationship between the rule and the traffic. Figure 3.1 illustrates the three ways in which traffic and rule set will interact. In Figure 3.1(a) there is no intersection between traffic and the rule set. This implies perfectly neutral traffic (possibly random characters). As such, the expectation is that a NIDS will operate at maximum efficiency. In Figure 3.1(c) all of the traffic intersects with the rule set. This represents a worst case scenario where all traffic will cause an alert, defeating most or all pre-filters employed by the NIDS to enhance average case performance. In this case, the expectation is that the NIDS will operate at its lowest efficiency, though that may not necessarily be the case if the traffic is only hitting an easily identifiable pattern. Finally, Figure 3.1(b) illustrates the idea that some of the traffic will intersect to some degree with some of the rules. In other words,



(a) No intersection between rule set and payload.

(b) Partial intersection between rule set and payload.

(c) Full intersection between rule set and payload.

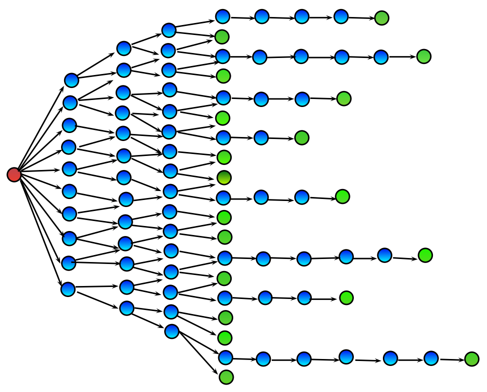
Figure 3.1: Illustration of intersection between payload and rule set.

there will be some alerts, and some searches deeper into the matching automata, but the magnitude depends on the amount of overlap. This represents the most likely operating condition for a NIDS but also the least understood of the three methods. Essentially, there are infinite number of ways that content in traffic can match content in the rules.

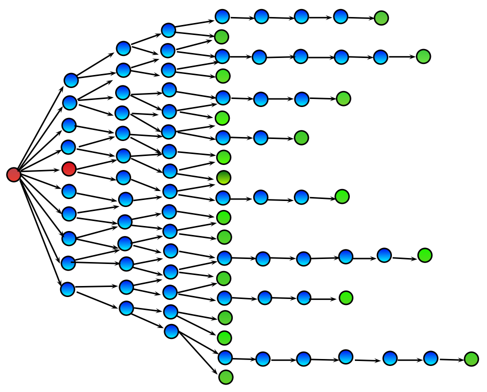
For example, most NIDS employ some form of automata for efficient pattern matching. The rule set is compiled into one of these large automata and the payload for each packet used as an input. Each character in the payload serves as an input symbol and can cause the current state to move within the automata. If the traffic is such that the current state is constantly moved back to the beginning of the automata, then it can be said that such traffic is neutral. This is because in operation the matching automata will likely be capable of caching all commonly reached states in cache and will operate without any cache misses (i.e. at maximum rate). This is essentially what is illustrated in Figure 3.1(a). However, as some packets cause the matching to move deeper into the automata, the previous sentiment does not hold. Thus, as the intersection between the rules and the traffic increases, so too does the effort required by the NIDS to reach a decision; even if that traffic does not cause alerts to the system. This is one of the primary principles motivating the Mean Decision Time metric. In other words, as the walks through the matching automata become more random and move deeper into the automata then more system resources are required to maintain the matching and ultimately the matching processes slows.

To better illustrate this matching process, we provide Figure 3.2. This Figure illustrates the basic structure of the typical matching finite automata. The automata is represented as a tree where each node deeper in the tree is further from the root, or initial state. First, it should be noted that the structure is quite uneven. This stems from the fact that some match strings are quite long. Second, there is a large density of states close to the root. This represents all the initial character combinations for the match strings. We should note that in an actual finite automata it may be possible that a transition will move to a state at the same level. Regardless, Figure 3.2(a) represents the finite automata at the start of a matching. In Figure 3.2(b) a character is matched and the state is moved deeper into the finite automata. In Figure 3.2(c) another state is matched, and we move yet deeper into the finite automata. However, the next input results in no match, and returns the matching state back to a shallower state as is illustrated in Figure 3.2(d). In fact normal traffic will typically exist in the first couple of tiers of the finite automata as most match strings are purposely designed not to be common patterns. Thus, as is illustrated in Figure 3.3, the matching engine typically only examines states in the very shallow tiers. This means that most states can be contained in cache memory and thus matching can occur at maximum efficiency. However, if it becomes possible to push deeper into the finite automata, as first demonstrated by Becchi et al. [96], then it is possible to push past this optimal processing and examine the behaviors of matching across a more clearly defined input.

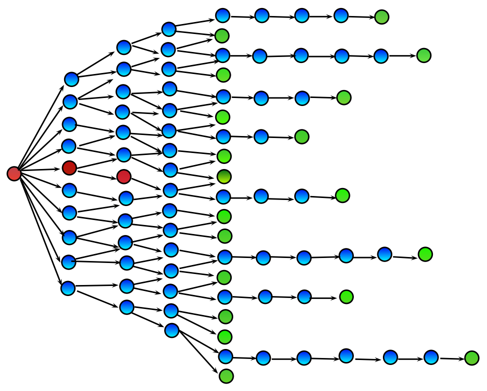
In order to effectively benchmark a NIDS, it is necessary to generate traffic that can systematically examine the behavior of the NIDS under varying levels of intersection between the traffic and the rule set. Current methodologies can account for the extremes of this type of behavior, but offer little aid in examining this middle-ground of increasing intersection. Essentially, current techniques use either neutral, or largely neutral background traffic into which is injected known malicious attacks. This type of testing methodology is much more relevant to anomaly detection



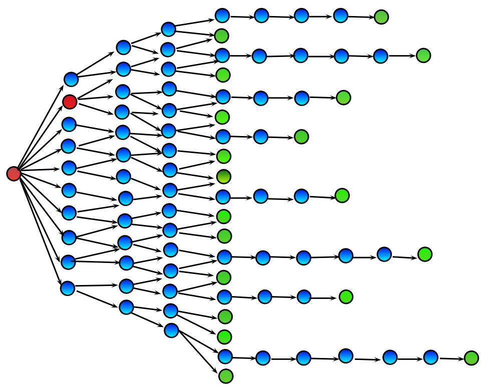
(a) Typical rule-set Finite Automata.



(b) Match to next deeper state.



(c) Match to next deeper state.



(d) Matching fails to go deeper, return to shallower tier.

Figure 3.2: Typical Finite Automata Matching.

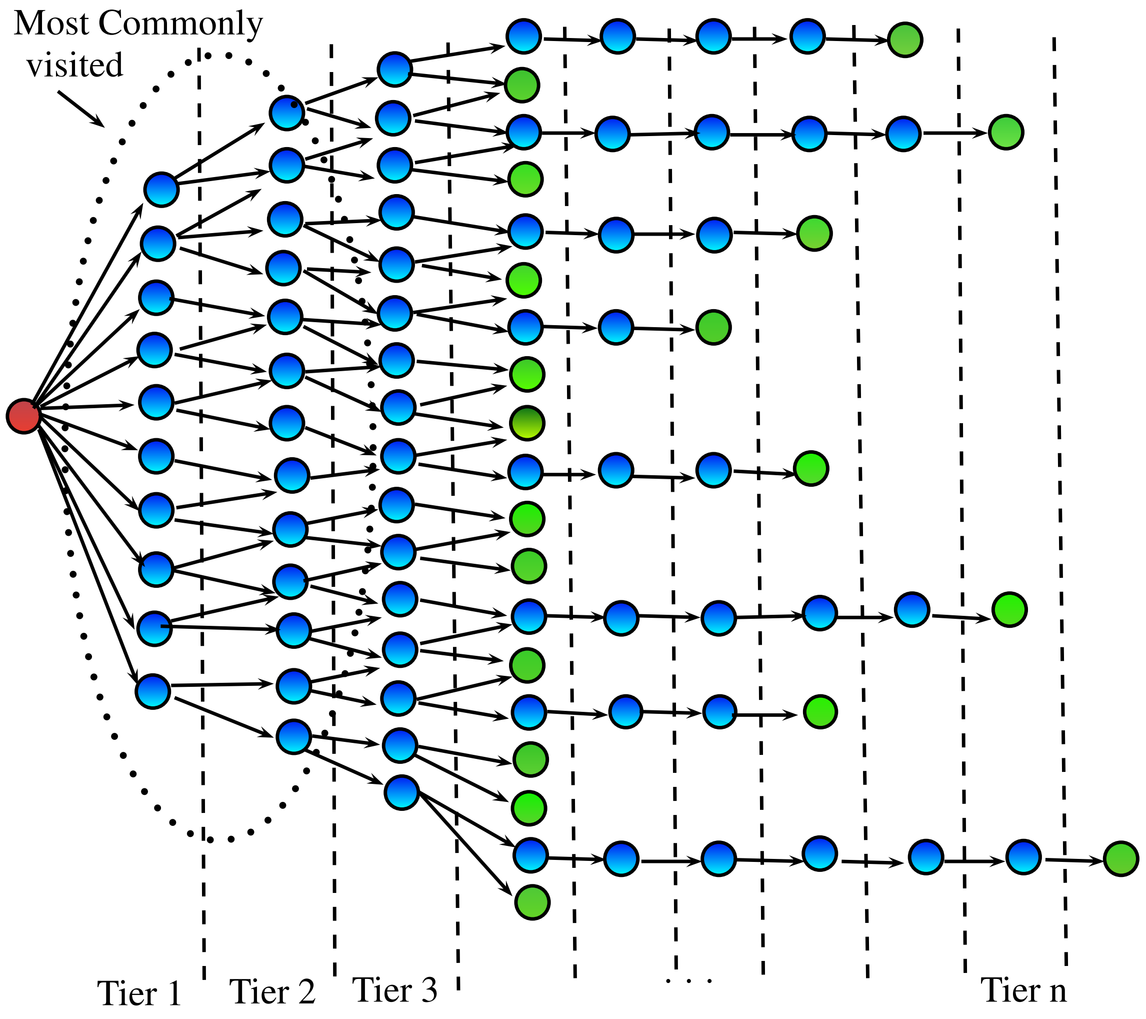


Figure 3.3: Typical matching only reaches a small subset of states.

systems and does not account for the need to create inputs that will more effectively explore the matching finite automata. Thus, in Chapter 4 we outline our method for addressing the effective benchmarking of NIDS.

However, benchmarking of NIDS is not sufficient for evaluation. NIDS must still be tuned to target environments. Common evaluation techniques use live traffic captures to accomplish this. However, there are a large number of issues with directly using live network packet captures from the lack of ability to vary parameters of traffic to the potential privacy and security issues involved. Thus not only must there exist a means to generate traffic with content that is capable of meeting the needs of NIDS as an application, but there must also exist a means for making traffic that can serve as realistic background noise. In order for content to adequately support a background simulation a content generative model must meet several constraints.

1. Appropriateness: Content should approximate that seen on a targeted network, at least in general distributions and context.

2. Variability: Any model must allow for perturbations to the content to explore potentials beyond the evidence provided from a targeted environment.

3. Privacy: Any model should protect user privacy and, where possible, not propagate user-specific sensitive data.

By addressing these points, it becomes possible to create traffic that will approach that seen on a particular network. Further, using a content-generative model over static replay can serve to both foster the ability to vary the traffic and maintain better privacy for users. We present ContextNet and a content-generative model in Chapter 5 to provide for this type of traffic. Finally, we explore implementation concerns for both approaches in Chapter 6.

CHAPTER four
CONTENT GENERATION FOR BENCHMARKING INTRUSION DETECTION SYSTEMS

Most traffic models consider raw packet counts and/or the inter-arrival times between packets with little thought to the payload of packets beyond whether or not the packet is designed to generate an alert in the system. Since NIDS are primarily tasked with identifying outliers, background traffic serves as the primary contributing factor in determining the overall performance of the NIDS. Unfortunately, aside from the presence or absence of alert-causing packets, current evaluation models offer no method for examining the potential intersection of traffic content to the rule set.

The primary contribution of this chapter is a content simulation model, loosely based on the work by Becchi et al. [96]. In this model, content is stochastically generated dependent on randomly selected rules within the rule set. The generated content will match a portion, or all, of a signature dependent on the selected parameters. The result is content that will intersect at varying levels with the rules of the NIDS. The Mean Decision Time (MDT) is utilized as the primary indicator of NIDS efficiency in matching. The MDT serves as a good indicator of performance as it provides a measure of how much processing the NIDS must exert to arrive at a decision. By monitoring the MDT, it becomes possible to gauge how efficiently the NIDS is processing packets and infer the overall performance of the NIDS. The end-result is an effective methodology for eliciting a NIDS ability to handle a spectrum of conditions from tranquil to pathological and everything in-between.

4.1 Stochastically Generating Content for Benchmarking NIDS

In order to explore the efficiency of a NIDS it is necessary to generate traffic that can impact the MDT. This requires an understanding of how each packet, as an input to the NIDS, can cause that NIDS to work harder. A packet is comprised of two pieces: a header and a payload. As mentioned in Chapter 2 the NIDS typically divides the rule set by header, using the header to reduce the number of possible rules to match against any packet. The more rules that must match against a packet, the greater the potential effort that the NIDS must exert as it navigates a larger data structure to identify the most applicable rules. The payload directly relates to the processing of the NIDS. The closer the content is to one or many rules in the rule set, the more effort for the NIDS to track multiple potential match paths through the matching automata or data structure as illustrated in Chapter 3.1. Finally, it is obvious that as the number of packets that cause more processing by the NIDS grows, so too will the MDT. These three factors represent the primary input into the NIDS and thus comprise the primary factors that can impact MDT. From this, we build our content simulation model for traffic using three parameters:

- *Frequency*: Relative ratio of packets containing content overlapping with rule set. Essentially, the overall percentage of packets that may intersect with a given rule set. Obviously, as this ratio grows so too should MDT.
- *Depth*: Depth of intersection for content with the rule set. The deeper content intersects with a rule the longer the NIDS must maintain state and the more potential effort required.
- *Density*: The number of rules against which the content must be matched. Navigating large data structures of rules can prove time consuming to identify those specific rules that apply to a packet.

This content simulation model works in the following manner. Given a series of packets derived from synthetic traffic, a traffic capture, or a live network, for every payload bearing packet, with probability *Frequency* that packet is chosen for the insertion of intersecting content and with probability $1 - Frequency$ that packet is left unchanged for some value of *Frequency* between 0 and 1. Packets that have their content changed select a rule from all possible rules categorized by the packet's header values. This may mean only a single rule in some instances. New content is then generated in the following manner. First, a new buffer is created of a size exactly equal to the content length of the current packet. The first character from the selected rule is added to the beginning of this new buffer. Next, a random walk through the rule automata is made similar to the work by Becchi et al. [96]. Essentially, with probability *Depth* the next character from the currently selected rule is concatenated to the buffer and with probability $1 - Depth$ a new rule is chosen and the first character from that rule is concatenated to the buffer. This process continues until the buffer is filled. Thus, the buffer will contain a series of partial, and potentially full, matches with rules in the signature database—the exact level of matching dependent on the value for *Depth*. The content of the packet is then overwritten with the contents of the buffer ensuring that the packet will match one or more rules to varying depths. Figure 4.1 illustrates the basic flow of this process.

4.2 Defining Base Content

One of the primary benefits of this approach is that an existing packet capture can be used as the basis for this traffic. This implies that any model capable of creating a valid series of packets can be used to create the basis on top of which this model is applied. This offers several advantages. First, it allows for side-by-side comparison. For example, a live capture of traffic can serve as a base, and then increasing levels of intersection can be inserted into the capture and the behavior of the NIDS monitored. Secondly, it means that multiple other simulations models concerning

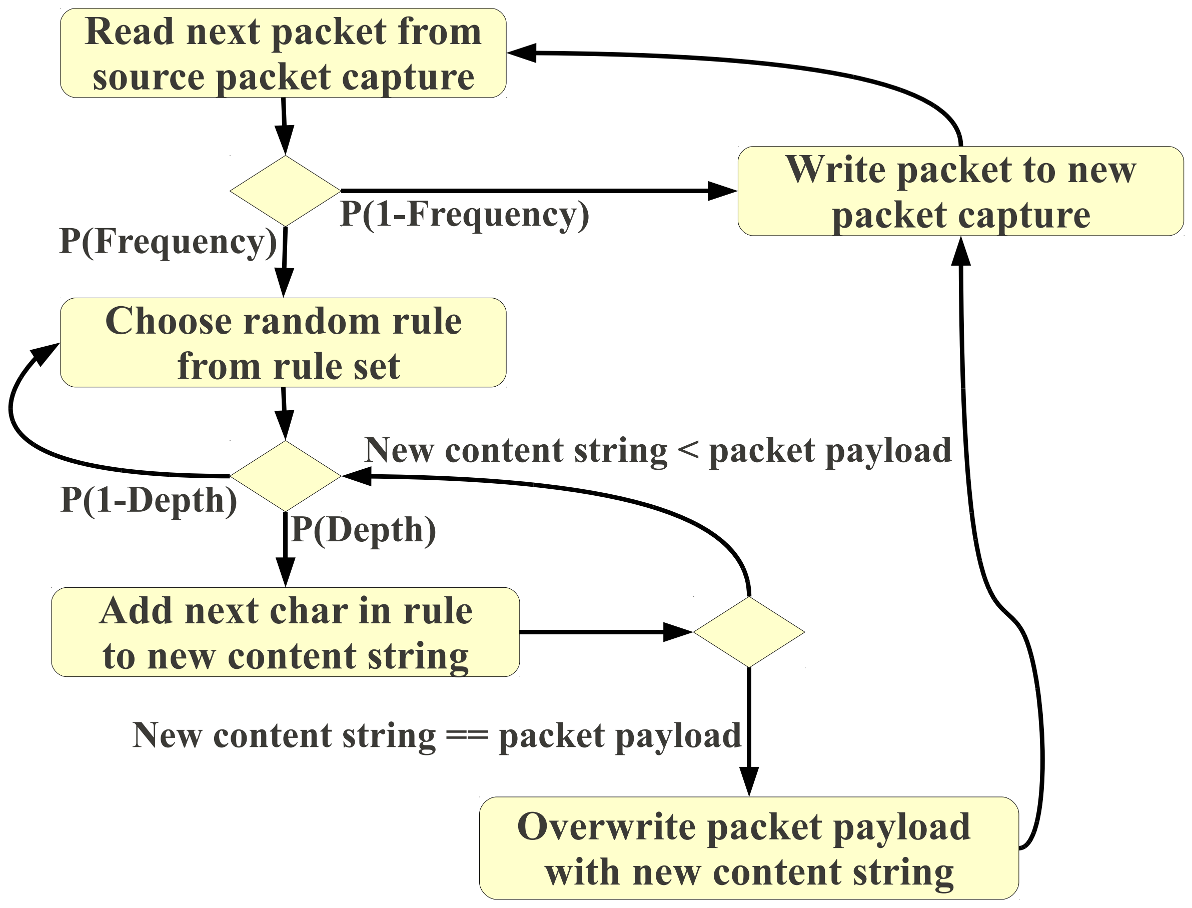
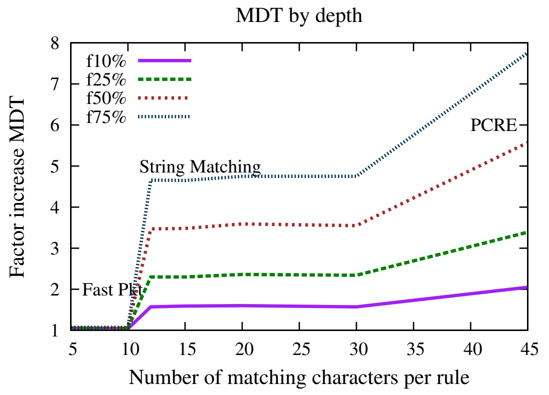

Read next packet from source packet capture
Write packet to new packet capture
P(1-Frequency)
P(Frequency)
Choose random rule from rule set
New content string < packet payload
P(1-Depth)
P(Depth)
Add next char in rule to new content string
New content string == packet payload
Overwrite packet payload with new content string

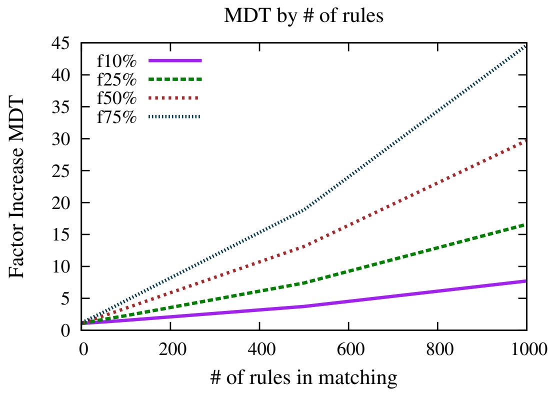
Figure 4.1: General process of generating content intersecting with a NIDS rule set.

packet counts or lengths may be employed in concert with this model. This is important as the flow-level quintuple typically determines the total number of rules applied to a packet. However, real life content is not well-suited to benchmarking as randomly overwriting packets can change the base behavior of how the series of packets impacts the NIDS. Further, real content can suffer from privacy issues. Finally, from an implementation view it is simply a good idea to have some "neutral" content that can be assured to have little impact on the NIDS. Randomized packet content offers the most attractive method for creating this kind of base packet payloads.

Simply overwriting traffic with random characters will eliminate any existing content and serves to remove circumstantial idiosyncrasies from the traffic as well as reduce the probability of intersecting content with the rule set. We posit that completely random content offers the most "neutral" content available. The reason for this is simple. Given a set of three characters, a, b, and c the probability that any one of these characters follows another character is simply $1/3$ assuming an independent equal distribution. However, since most rule sets target specific languages (human or machine) the distribution of characters in traffic is not equal, nor is it independent. Thus, after a the character b might be twice as likely to occur over c. As the string of characters grows longer, the chance that random content will match to actual language patterns drops significantly with each new character added. As a result, the probability of a randomly generated string of characters creating long strings that match to actual languages is quite low. In fact in order to successfully generate strings that emulate natural languages it becomes necessary to employ a Markov chain dependent on the distributions of character sequences not single characters. As such, random characters offer a simple, yet effective, means of creating anonymous content that will offer a NIDS near best-case processing. The near best-case processing is a result of the fact that the content strings are unlikely to match patterns in the rule set thus the NIDS will continually traverse only the shallow regions of the matching finite automata. Further, random characters are superior



(a) The three stages of Snort pattern matching starting with the fast-packet matcher followed by the Aho-Corasick string match and ending with Perl Compatible Regular Expression matching.



(b) As the number of rules applied to a packet payload increases the NIDS is forced to work harder.

Figure 4.2: Impact of payload intersection with Rule set on NIDS performance.

to using a single static character as they more closely model the unpredictability of characters within a language; at least more closely than a single repeated character. We have seen this in our investigations as traffic with randomly generated content cause a slightly higher MDT than traffic with strictly static characters (i.e. content replaced with a single, uniform character). Further, traffic with randomly generated content more closely aligned with the MDT behavior of actual traffic that contains no intersection with the rule set—at least in our empirical observations.

A final issue with the base file is the impact of packet size. Packet size can influence the MDT. This impact is negligible in “neutral” content but can prove significant in matching or partially matching content. Fixing the packet size can serve to make the impact of partially matching content more clear and offer a good base test. Employing realistic packet distributions offers evaluation which may demonstrate slightly more variation but which will illustrate more realistic conditions. It is necessary for the evaluator to decide which serves the test goals better.

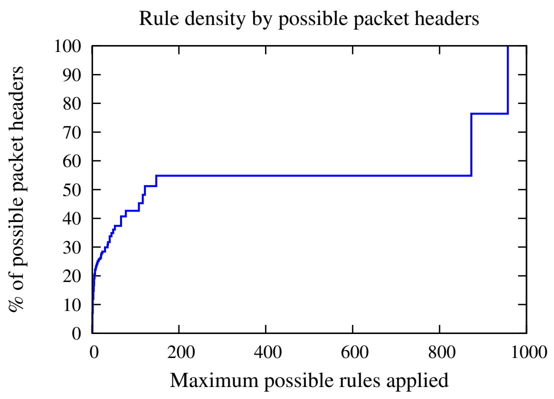


Figure 4.3: Maximum number of rules applied to a packet dependent on the packet headers.

4.3 Accounting for the Rule Set

The *Density* parameter determines the number of rules applied against any packet. As such, it can have a huge impact on the MDT as illustrated in Figure 4.2(b). In this figure, it is evident that as the number of rules applied to a packet increases so too does the average MDT for the traffic. In fact, even just a small percentage of packets hitting a large number of rules can increase the MDT six to seven times normal.

The *Density* parameter operates differently than *Frequency* or *Depth*, yet it is still quite simple. First, we note that for any rule set there is an inherent distribution of rules as is illustrated in Figure 4.3. This Figure displays the maximum number of rules, as categorized by IP flow-level quintuples for the Sourcefire Vulnerability Research Teams (VRT) Snort rule set for September 15, 2010 [104], that might apply to a packet. We posit that randomly selecting a rule from the rule set will follow the distribution imposed upon the rule set by the data structure used by the NIDS. Thus, in most cases the *Density* parameter is inherent in the rule set employed, at least if traffic is pulled from randomly selecting rules from the rule set. However, this can be modified to meet specific evaluation needs in two primary ways. First, a live capture can be used to derive the basic structure of the network traffic. In other words, the live capture will define all of the IP flows. In this instance, the choice of rules is no longer equal across all possible rule choices in the rule set, but is defined by the traffic exhibited in the traffic capture. This could potentially skew simulated content to originate from a reduced set of rules. However, since those rules more closely match specific traffic patterns, this allows the content simulation to demonstrate a good value of ranges that might be expected under normal operation for some target environment. A second method is to simply inject entire streams of packets into traffic dependent entirely on the rule set. This is similar to the approach employed by default with one exception in that emphasis can be placed on specific regions of the rule-set. This allows evaluation of the rule set overall and can help identify inefficient rules.

As a simple example, imagine a NIDS with a rule set containing two string signatures: “match one” and “match two”. A NIDS evaluator wishes to generate a test that will evaluate varying degrees of intersection with this rule set. The evaluator has a packet capture gathered from the targeted network which has 10 packets. The evaluator wishes to see first how the NIDS is impacted by increasing frequency of content intersecting packets. Thus, he creates a new packet capture employing our algorithm, with the original capture and rule set as inputs, and *Frequency* set at .1 and *Depth* fixed at .5. The algorithm reads each packet in the original packet capture and randomly choses a number between 0 and 1. On the fifth packet, the value randomly chosen comes in at .053 and this packet is set for modification. Once a packet is set for modification, a rule is randomly chosen from the rule set. Since there are only two rules in this rule set, and since they are equally likely (i.e. there are no constraints on what packets might compare against one over the other) then there is an equal chance that either is chosen. For this example, the second string is chosen first. Now the algorithm starts building a new content string for the packet. Given that rule two is chosen, the first character of this rule is added to the new content string. Following that *Depth* is .5, every time a random number is selected that is less than or equal to .5 then the next character in the match string is added to the new content string. In this example, the first four randomly chosen values are .3, .21, .08, .76, and thus the initial string becomes “match”. However, the payload of the chosen packet is twenty characters long, so the algorithm repeats until that length is met, ultimately resulting in a content string of: “matchmamatchmatch om”. This new content string then replaces the content for that packet in the newly generated packet capture. True to the .1 value for *Frequency* only one such packet is changed in the newly generated packet capture. However, the evaluator wishes to make a battery of tests and thus creates multiple files spanning *Frequency* values between .1 and .9. The evaluator could further change the values for *Depth* to see how increased intersection of content also impacts the NIDS. Essentially, under this Content Generation model, it becomes possible to incrementally manipulate the MDT of the NIDS and examine how it behaves at a variety of levels.

4.4 Evaluating the Benchmarking Model

We evaluate our traffic generation model using the NIDS Snort [14]. We begin the evaluation using 1661 synthetically generated rules representing flow-level quintuple groupings of: 1 rule, 10 rules, 50 rules, 100 rules, 500 rules, and 1000 rules respectively. Synthetic rules were chosen as a logical starting place in order to focus on the basic MDT behaviors for Snort. Each rule was built in the following manner. First, a unique IP flow-level quintuple was defined for all rules in a grouping. Next, an initial match string was chosen for all rules in a group. This is done purely for the Snort Fast Packet Processing which performs an initial match against the longest content match of all possible rules based on flow-level features in order to build the actual set of rules used in the ultimate matching of the packet. By synthetically fixing a match string we can examine how Snort handles larger sets of rules in the final matching. Next, we added a randomly chosen content match string for each rule, and then finally a randomly generated regular expression for each rule. Thus, each rule had two exact match strings, and one regular expression. Finally, we set all payload-bearing packets to a uniform 500 bytes of randomly generated content. Given this base, we created various captures using different parameters and noted the increases in the Mean Decision Time (MDT). The MDT was calculated by having Snort read the packet captures in off-line mode and dividing the process time by the total number of packets evaluated. The results of 5 tests were averaged for each data point.

For Figure 4.2(a) we employed only a single rule and for illustrative purposes we fixed the *Depth* to a set number of characters rather than a probability. The results in Figure 4.2(a) reaffirm the results found by Dreger et al. [95] demonstrating that the MDT rises very clearly in proportion to the number of detectors involved. In other words, while the initial match is less than 12 characters, which is the size of the first content string in each rule, then Snort is able to process each packet at near maximum speed due to the use of pre-filters. Essentially, Snort's Fast Packet processing engine looks for only the longest content pattern for each rule for a given set of rules and if a packet does not contain at least one of these patterns then it is ignored. However, once a pattern is matched, the entire packet must undergo the full fixed-string pattern matching. This is illustrated by the first jump in processing time. This also represents the fixed time matching of the Aho-Corasick string matching algorithm [108] in that the length of the partial match has minimal impact on MDT. However, as the matching pushes deeper into the rules it begins to hit the regular expressions which become quite burdensome as the character content extends close to a match. This finding is somewhat additional to that proposed in [95] in that it implies that some detection engines will increase in processing dependent on input, not just the number of detectors employed. Finally, Figure 4.2(a) also illustrates how the frequency of packets bearing intersecting content with the rule set can impact the MDT. If the *Frequency* of such packets is high enough, even if the *Depth* is quite low, the increase in the MDT is substantial.

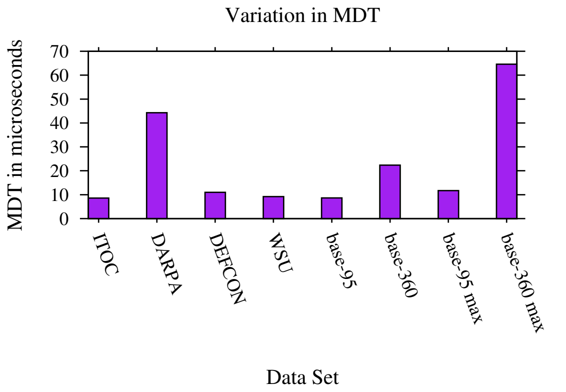
Likewise, Figure 4.2(b) illustrates how the number of rules applied in matching can adversely impact the MDT. For this test, traffic was generated for varying frequencies of content-bearing packets with a fixed content intersection of twelve characters, just enough to push Snort past its fast packet processing. Clearly, as the number of rules that the NIDS must match against increases, so too does the the MDT. This is directly related to the need to navigate a data structure, a tree in this case, to prune the results to the most likely candidates. In worst case scenarios, perhaps the result of a poor rule set, it is possible for the number of rules to dominate the processing time. This implies that in boundary cases, NIDS resource usage will not grow linearly to the number of detectors employed, as stated in [95], but will in fact increase dependent on input, or the rule set, as illustrated by Smith et al. [103].

These first tests illustrate how simulated content can significantly impact the MDT of a NIDS. However, synthetic rules are far from real rules and thus do not offer a particularly practical evaluation. Thus, for our second evaluation we employ a reduced subset of the rules from the Vulnerability Research Team (VRT) Snort rule set for September 15, 2010 [104]. We culled 3,348 rules from the 3,944 default rules. Rules removed from evaluation involved complex features of the Snort NIDS such as Byte jumps and relative distancing that we have yet to implement in our tools. The absence of these rules in this research has no significant impact on the overall results as more rules cannot make for less processing. As a general comparison we turned to several different packet captures that represent common methods for evaluating NIDS. The first is Wednesday from the first week of the 1998 DARPA data set [6] (Wednesday chosen randomly). The second data capture is the US Army side of the US Army Information Technology and Operations Center (ITOC) 2009 Cyber Defense Exercise data set [24], the third is from the Capture the Flag event for Defcon 11 [23], and the final is a traffic capture pulled locally from the Network Research

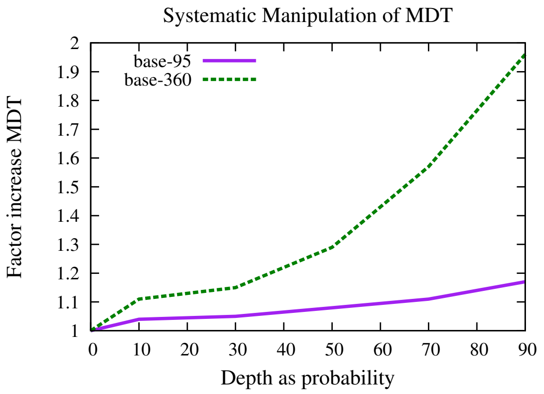
Laboratory at Washington State University where we listened at the gateway between our network and the university network. Finally, we generated two synthetic base files: one with an average packet size of 95 bytes and the other with an average packet size of 360 bytes. From these synthetic packet captures we also created two more packet captures, each representing a worst-case scenario where the content is simulated such that *Frequency* and *Depth* are both 95%.

Figure 4.4(a) illustrates how the MDT varies across these different packet captures. As is evident, the MDT can fluctuate greatly between different packet captures. This stems from differences in packet-size to total number of alerts in a file to, as we have posited throughout this paper, the intersection of content and rule set. Still, this Figure illustrates some significant phenomena. First, among the publicly available packet captures there is considerable fluctuation. Though the graph does not demonstrate it well due to the large outliers, the difference in MDT between both the ITOC and Defcon packet captures and the WSU and Defcon packet captures is nearly 50%. Such a discrepancy can cause substantial differences in evaluation if only one such file is used. More obviously, the DARPA packet capture demonstrated a very large MDT at nearly 4 times greater than any of the other publicly, or privately, available captures. This figure demonstrates that simply gathering a packet capture is not sufficient to evaluating the MDT of a NIDS. Multiple packet captures would be needed, and those would need to be categorized based on their impact to MDT.

Simulating content, however, offers the means to methodically evaluate the MDT of the IDS. To illustrate this we added the base-95 and base-360 to Figure 4.4(a) which represent packet captures where the number of bytes per packet were 95 and 360 bytes respectively. These packet captures contained entirely random content and represent lower bounds for an evaluation. We then created upper boundaries by adding two more captures with simulated content of *Frequency* and *Depth* of .95 each. These four simulated captures demonstrate a wide range of potential evaluation



(a) The variance in MDT across different traffic captures as well as the possible range available due to the proposed approach.



(b) A systematic evaluation of a NIDS employing the proposed model with monotonically increasing levels of content intersection.

Figure 4.4: Fluctuation in MDT by data set or Methodology.

encompassing and exceeding that offered by the other captures. By further refining the values for *Frequency* and *Depth* it becomes possible to more closely evaluate the MDT for the NIDS at varying compositions of traffic. As an example, we provide Figure 4.4(b) which illustrates how it is possible to explore a variety of traffic compositions against an IDS in a controlled and monotonic fashion. For this test, we based all simulated packet captures off the base-95 or base-360 captures also found in Figure 4.4(a). We arbitrarily set *Frequency* at .5 and then proceeded to increment the *Depth* from 0 (the initial file) to .9. In a similar manner it is possible to explore other compositions, though often the most desirable cases will be the boundary cases as illustrated in Figure 4.4(a).

Considering Appropriateness, Variability, and Privacy Simulating content in the manner outlined in this chapter easily meets privacy and variability considerations as outlined in Chapter 3.1. Essentially, under this model, content in traffic captures is removed and replaced with purely synthetic content eliminating any sensitive data. Further, if traffic is completely generated from the rule set without any adherence to the flow statistics of a given network then there exists no possible relation between the simulated traffic and users. Even if network flow statistics are used simple anonymization may be employed to remove the ability to relate communications back to actual users from a particular network. Likewise, variability is maintained in the three variables in this model which allow for the systematic generation of content. The *Depth* variable can allow for lesser or greater intersection between content and the rule set while the *Frequency* variable can regulate how many packets are thus affected. Lastly, *Density* can be utilized to target more specific areas of the rule set. Appropriateness, however, does not well apply to this benchmarking model. The goal of this model is to provide a workload to the NIDS. Figure 4.4 clearly shows that this model is appropriate to causing load in a NIDS. However, the content simulated under this model does not resemble real traffic. Thus, it is wholly inappropriate to generating traffic that might simulate a targeted network. This is one of the motivating factors behind the research we pursue in the following Chapter.

We end the evaluation with some final observations. First, we note that the impact on MDT by this content generation model is greatly reduced when tested against real world rules. This stems from the fact that those real world rules employ multiple techniques to limit the amount of time spent examining each packet. Even so, simulated content can still serve to increase the MDT significantly and potentially identify rules, or regions of rules, that need refinement. Second, the Mean Decision Time is not meant as a metric of absolute comparison. It is impacted by the platforms and systems used. However, this is not necessarily a bad thing. A NIDS running under improved hardware should demonstrate a smaller MDT than the same NIDS running on a less powerful platform. Thus, hardware does stand as a factor that can impact MDT, but it is a factor that is likely static for a given testing environment Further, this means that the MDT can serve to make distinctions between different hardware implementations. Thus, trends exhibited by the MDT extend across all NIDS—accounting for hardware differences when necessary. Finally, we have restricted our analysis to the impact on the MDT, but recognize that False Positives and False Negatives are still the primary metrics for NIDS. However, MDT is not completely independent from False Positives in particular. Essentially, False Positives rates mark packets that completely intersect with a rule set at maximum depth. As such, high False Positive rates can be expected to increase the MDT. Likewise, as the MDT increases, the number of False Negatives due to packet drop can be expected to increase. Regardless, MDT provides a solid core indicator of NIDS behavior that reflects the input into the system and foreshadows potential results (i.e. overburdened systems). Given this, the content generative model in this chapter can serve to systematically evaluate the performance of a NIDS and provide a better understanding of the efficiency of the NIDS under a particular rule-set. In the next Chapter, we will explore more generalized means of generating traffic.

CHAPTER five
DYNAMIC AND REALISTIC CONTENT FOR BACKGROUND TRAFFIC GENERATION

Creating content in network traffic to benchmark NIDS serves an important role in evaluating the performance of NIDS under varying levels of exertion. However, it does little to determine the fit of a NIDS to a particular environment. In fact, a primary difficulty in evaluation and educational testbed environments is the adequate generation of application-level content to support simulated network traffic. Such content is used to evaluate applications, such as a caching system or NIDS for example, or to create the atmosphere of a live network that can be used to train individuals in handling a variety of network scenarios. The quality of the content utilized in these evaluations can directly impact the value provided by the testbed environment. Unfortunately, the ubiquity of potential content as well as the large variance of such content encountered across different live networks makes populating testbeds with representative content problematic.

Common remedies to this problem range from ignoring it altogether by generating static content (i.e. random characters or static files) to laboriously recreating a network through the import of documents harvested from the Internet, or business intranet, to simple packet capture replay. While static and random files may prove adequate for benchmarking certain applications, as we demonstrated in Chapter 4, they offer little value to complex evaluations requiring traffic more closely aligned with a particular target network. Harvesting documents from other sources and recreating an external environment with a tool such as the Security Assessment Simulation Toolkit (SAST) [30] or the Scalable WORkloaD generator (SWORD) [88] can tune a testbed environment to closely match a target network. However, gathering enough content and configuring the environment can require extensive labor (as much as 160 to 240 hours of labor in the author's experience) for a single test. Even worse, content gathered in this way may fail to properly represent the targeted environment in many respects such as the relative diversity of content and distribution of content amongst users. Finally, packet capture replay can recreate the conditions of a target network and provide some meaningful evaluation but suffers from myopia, limited variance, embedded network phenomenon which may, or may not, be representative of the network, and potential privacy issues. Essentially, there exists no clear model for gathering and generating content in a testbed environment.

In order for content to adequately support a network simulation a content generative model must meet the constraints as outlined in Chapter 3.1: Appropriateness, Variability, and Privacy. We propose a multi-step approach for generating content that meets these constraints. Given a sample from a target network, we derive the major clusters of content dependent on the contextual distance between separate documents. This data is then abstracted to create a series of document content clusters that can be used to automatically harvest similar documents for generation using the b-model Surfer. Further, the behaviors of content consumers (users) and content producers (servers) are captured and clustered into like classes of consumers and producers. This information ensures that not only the contextual trends of the target network are captured, but that the user trends associated with those contexts are likewise retained. Armed with these data, it becomes possible to create the generative pyramid where any given content cluster can be tied to a particular consumer or producer. Thus, any workload generator can be used to create a workload of traffic where the generative pyramid can assign content, a producer, and a consumer to each flow created thus matching content to the workload. In this manner, content generation can be generalized to account for a much broader range of applications.

5.1 Clustering Traffic by Context

A large body of research has investigated the classification of network traffic. This research ranges from classifying the applications present in a particular network [109–112], to the binary distinction of traffic as either malicious or benign exhibited by most NIDS [14, 94]. However, the focus of such research is narrowly targeted to address traffic engineering, system operation awareness, and general security. As such, the content embedded within the traffic generally finds use as a lever for assigning a particular application or intent to a given communication and little else. Thus, the thematic and contextual trends within the traffic goes largely ignored despite the fact that such information is a valuable resource for business, system evaluation, and potentially for security.

We propose ContextNet, a technique for contextually describing network traffic built upon the WordNet project [113, 114]. The idea behind ContextNet is to exploit the Synonym and Hypernym sets present in WordNet to provide contextual clues as to the general subject of network traffic. ContextNet is able to categorize documents without prior knowledge or large rule sets. The desire with ContextNet is to provide the means by which a traffic set may be classified by the general context of the content witnessed within an application. Further, the information retained from ContextNet should remain such that traffic of similar content, but not identical, can be generated. This will then be used to facilitate the generation of realistic content for simulated network traffic.

5.1.1 Content Classification of Network Traffic

The context of application traffic, as derived from packet payload or reconstruction of documents, has consistently played a significant role in the classification of network traffic. Evidence of this is most explicitly demonstrated in the function of NIDS such as Snort [14] and Bro [94], which perform full payload matching against a set of known signatures in order to realize a binary classification of traffic: benign or malicious. The underlying premise of NIDS considers the presence of a particular signature within a packet (or packets) to indicate a contextual predisposition toward malicious behavior. Yet, in even more exotic classification techniques the content of the traffic still plays a significant role. BLINC [109], for example, employs a hierarchical classification system that utilizes social, functional, and application aspects of traffic, without content, to arrive at high accuracy traffic classification. However, the infrastructure to implement this classification depends on a low-level packet examination in order to derive the original behavioral and statistical mapping to enable accurate classification. Even though BLINC, in operation, foregos the use of content, content was fundamental in building the model by which BLINC can operate. Similarly, Trestian et al. [111] use Google to ascertain useful information about endpoints and leverage this information to classify traffic. Still, they employ keywords to properly tag URLs to help in identifying traffic. More recently, Lee et al. in [112] use keywords in Wireless Application Protocol (WAP) to classify traffic into predefined categories. Essentially, the content of an application serves as a strong corroborating factor for identifying a particular type of traffic.

In the realm of data mining, the content of a document typically reigns as the most significant factor in determining closely related documents. The key in most of this research is to identify a particular feature set that best describes each document and then employ various machine-learning or statistical techniques to identify closest matches. For example, similar in spirit to ContextNet, Wermter and Hung [115] use Self-Organizing Maps and WordNet [113] to classify news articles.

5.1.2 Building a Model for Contextual Classification

In order to adequately generate application-level content to approximate some target network it is necessary to quantify the traffic seen on that target network. The potential variance in content on a given network is huge. However, a concrete model that can classify general trends within the content of a target network can serve to at least delineate broad categories or trends within the traffic of a network. These trends can be reduced to clustering behaviors which can then be modeled in ways such that generated traffic will follow those same clustering behaviors. In this manner, it becomes possible to generate traffic that will follow the clustering behaviors of a target network and may also follow the same contextual trends, at least enough so that humans viewing the traffic would not immediately discount the traffic as not belonging as is often the case for randomly or statically generated content. To accomplish this classification stage we create the tool ContextNet.

ContextNet is formed on the assumption that text can be derived from an underlying traffic stream. The process of deriving said text may be direct, as in the immediate removal of text from a textual document, or secondary as in textual categories derived from sound [116, 117], video [118, 119], or other possible techniques. No limit is placed on the potential sources of textual data, only that such data exists and can be derived in a reasonable fashion. A secondary, and weaker assumption, is that there exist some discernible boundaries between documents, streams, etc. Absence of such boundaries would imply a single large document and belay the need for cross-document classification. Thus, in order for ContextNet to function it is assumed there exists a set of samples where each sample represents a bounded region of text. These samples may come from full documents, direct packet payloads sampled from network streams, or even derivative aggregations of text from a variety of sources. However, the results will follow the quality of the samples.

Once a set of samples has been taken each sample is reduced to the proverbial "bag of words" where each distinct word in the sample is mapped to the frequency of occurrence of the word in that sample. Several attempts are made to reduce possible confusion by removing overly common words or structures. The cleaning process normalizes whitespace and removes punctuation for ease of processing. All characters are converted to lower case for consistency. Further only verbs and nouns are retained as only these words pertain directly to the context of a sample. All other words are simply sentence or thought support (i.e. prepositions, conjunctions, etc). Specifically, the cleaning process follows these steps:

1. Remove extraneous whitespace, though retain spaces between distinct words.
2. Remove punctuation.
3. Reduce all characters to lower case equivalents.
4. Remove all words not nouns or verbs—WordNet is used to accomplish this task.
5. Remove all words found in an exclusion list (words like "more" or "has", and all words 2 characters or less in length.

The Bag of Words

The cleaning process reduces the set of words to those most relevant to the context of the sample. In essence, it removes prepositions and articles that offer little inclination to the subject of the sample and leaves behind a smaller set for classifying the sample. Thus, the "bag of words" now represents a feature vector for the given sample. These feature vectors demonstrate some interesting properties. Figure 5.1 illustrates the common distribution of the frequency of words within a document. This data was derived from two separate corpora. The first corpus was pulled from the Reuters-21578 News corpus [120] which contains a large body of news articles. The second second corpus represents 1000 randomly selected web pages harvested using the b-model surfer algorithm as illustrated in Section 5.1.4. The foremost feature of these distributions is the fact that words that occur only a single time (after the cleaning process) within a document account

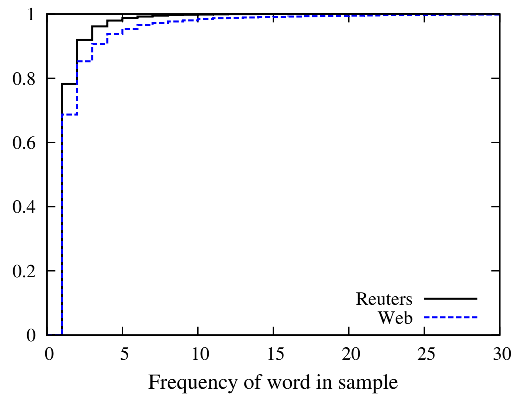


Figure 5.1: Cumulative distribution of frequency of words within a single sample.

for around 70 to 80 percent of all words in the document. Clear evidence of this phenomenon is illustrated in Figure 5.1. Not illustrated, however, is the fact that most words that occur with increased frequency within the document pertain directly to the subject of the given document.

A common technique in document processing is to create a working set of words that can best represent these feature vectors. This requires culling a set of most common words from all samples in a sample space and then deriving the individual frequency of each *feature* (word) within a given sample. This form of fingerprinting then relies on the presence of these words within samples to determine where the samples map to within the entire sample space. However, we posit that the most important words within a sample, at least for ascertaining general category, come from those that are more frequent—once sufficient cleaning of the sample has occurred. As a result, relying on the most common words across all documents may not be necessary. In fact, upon examining 1000 samples from the Reuters dataset of articles tightly grouped within a single category defined by the topic of "earnings" and place of "USA", 20% of the words encountered amongst the top 3% most frequent words (roughly 200 words) directly applied to the given topic and/or location (i.e. words like "dividends", "profits", "companies", and "Texas"). Conversely, within the bottom 50% of the most frequent words (roughly 1000 words) only about 4% of the words demonstrated any discernible relation to the given topic and location. Given this empirical evidence, it appears that the bulk of the words within a sample do not provide much information for categorization. Similar results have been demonstrated through different means by Wermter and Hung [115] illustrating that accurate categorization can occur simply by examining titles and by Bennett [121] using graph sampling.

These data imply that less is more. By utilizing only the most frequent words within a sample to serve as the significant features for that sample, it greatly increases the chance that those words directly relate to the actual categorical meaning of the sample. Further, by eliminating many other words of limited influence, there is less unintentional collision between samples of different categories but sharing many similar vocabularies. Finally, reducing the significant features to the most frequent words reduces the amount of data to be processed. As such, ContextNet sorts each vector of words by frequency and retains only the top 30%. The number 30% was derived from the division, as illustrated in Figure 5.1, between words with frequency of 1 and those with greater frequency for the "web" documents. We note that this threshold is only a rough approximation driven by the data we have examined. However, we further note that 30% is a very conservative estimate and an even smaller percentage could potentially serve as well.

Building the Hypernym Tree

While the above feature vector provides a good metric for comparing samples, it still does not necessarily capture categorical behaviors. For example, samples with several terms like "cash", "paper money", and "coinage" are all related to a broader category, "currency" in this case. This relationship is termed *hypernymy*. More specifically, a *hypernym* is a word or phrase with a meaning that encompasses the meaning of another word. WordNet [113] provides access to this hypernym relation. The hypernyms can provide an aggregate view of the sample with multiple words relating to

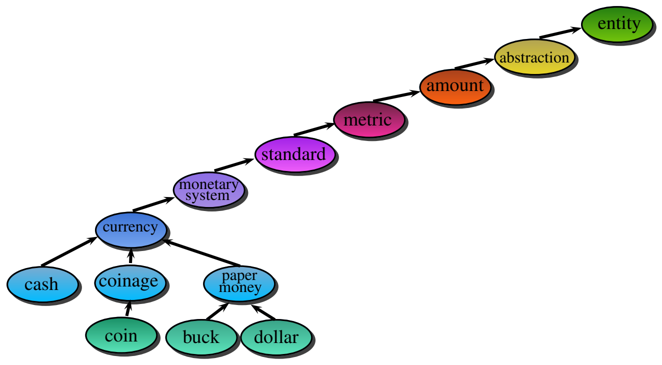


Figure 5.2: Example WordNet hypernym tree.

a single hypernym. As such, capturing this hypernymy can provide inter-word relationships within a single sample and reinforce the importance of those contextual clues. Thus, ContextNet attempts to capture these inter-word relationships in a second feature vector that employs the hypernyms of the words from the sample.

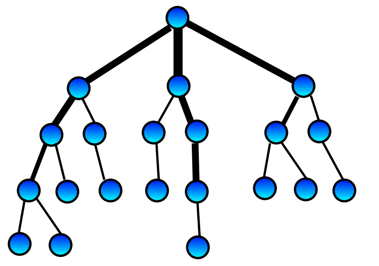
Within WordNet, the hypernyms are maintained as a tree structure with each node (word, or set of words, in this case) closer to the root a hypernym of all immediate children. As such, given a single word, it is possible to get the hypernym of that word, then the hypernym of the hypernym, and so on until the root of the tree has been reached. Figure 5.2 illustrates an abbreviated tree structure for the leaf terms “cash”, “coin”, “buck”, and “dollar”. The word “entity” serves as the root of the tree. Further, it should be noted that all siblings are synonyms. Also, nearer the root the terms become more and more abstract. It would be a simple task to create a classifier that could classify any sample into the category of “entity”, as “entity” is the root of the structure. However, the desired outcome is to generate a few useful categories that describe the sample. This can be done by having WordNet calculate the hypernym tree for each word from the “bag of words”. Thus, the entire set of words can be reduced to a set of hypernyms of equal or smaller size (assuming only usage of only one word sense per word).

However, determining a useful hypernym of a word for a particular context faces two major difficulties. First, it should be noted that many words have multiple senses. For example, the word "currency" has three senses. One sense is monetary as depicted in Figure 5.2. The second sense, however, follows a hypernym of "prevalence", and the third sense that of "currentness". As such, this can create ambiguities in determining which sense is most prevalent to a current context. It should be noted that WordNet orders the senses from most likely to least, with the first sense the most likely. The second difficulty stems from determining which internal node offers the best hypernym to describe the sample. We term this a category for the sample. In some instances a more intuitive category might be two or three levels higher than the leaf node. For example, in Figure 5.2, "dollar" is beneath the hypernym "paper money" while "currency", the hypernym of "paper money", might offer a better category for the context of a particular sample.

To thwart such ambiguity, it is necessary to pull more information from the entire document to ascertain the most probable hypernym to serve as a category for the sample. ContextNet attempts to solve the problem by building a tree from the hypernym branches, as provided by WordNet, for the most common sense of each word. The way this works is that WordNet is queried for a hypernym branch of a particular word resulting in a structure similar to Figure 5.2. This branch is added to a tree structure in memory; this tree structure initially empty prior to parsing a sample. The structure provided from WordNet is pushed onto this tree structure recursively. This is repeated for the most common sense of each word. The edges maintain a count incrementing every time a particular edge is traversed. Essentially, this count maintains the weight of traffic, or *flow*, across a particular path through the tree. Upon completion of parsing, the tree structure will have as many leaf nodes as distinct words within the sample, and each edge between nodes will be weighted with the frequency that particular edge was traversed. It should be noted that this stage occurs alongside the creation of the "bag of words" thus each word, even the low frequency words, impact this tree.

Now that the tree is built, it is time to prune the tree. As a general result of the frequency of synonyms and similar words in the sample, common paths begin to take shape through the tree. Further, all the frequency-one words ambiguous to a particular sample will create regions within the tree where the edges have very low weights. Even better, esoteric senses to words will also create these sparsely traveled regions. Thus, to prune the tree, we begin at the root and test each edge from that node to all children. If the weight of the edge is ever less than 1% of all branches that were added to the tree, then that edge is cut and all children beyond that point are lost. The algorithm then moves on to all connected children and continues recursively until there are no more children to reach. The threshold was chosen as a result of empirical evidence in that nearly all hypernyms directly relevant to a sample had entering edge weights greater than 1% of the total branches added to the tree. As such, 1% serves as a very simple threshold for focusing on the most favorable hypernyms to categorizing a sample. For ending nodes that are actually leaf nodes, then the hypernym is chosen and the leaf node pruned from the tree. This is a rare case though, as most often the edge weight between a final hypernym and its child is typically below the 1% threshold. At the close of the pruning, all the leaf nodes within the tree now represent the most relevant categories for the document. Figure 5.3 illustrates this pruning process. In Figure 5.3(a) the bolder edges represent more heavily traveled paths, while the light edges represent edges traveled less than the 1% threshold. In Figure 5.3(b) all the light edges have been pruned away leaving only 3 leaf nodes which are promoted to categories and added to the feature vector.

The remaining leaf nodes become a feature set with the weight of the connecting edge to that leaf node as the weight for that particular feature. These leaf nodes are sorted in descending order from heaviest weight to least and the top 10 categories are chosen as the category feature set to represent this sample. It should be noted that all categories could be used, and in fact ContextNet did use them all originally. However, in our experience, we noted a slight improvement from the further winnowing of categories. The primary culprit of this results from some hypernyms that


(a) Example hypernym tree with edge weights. Darker edges imply heavier weight.

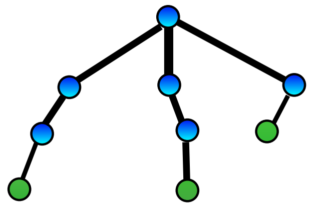

(b) Example hypernym tree after pruning—all light edges removed.

Figure 5.3: Pruning the hypernym tree.

occur across many common words. These categories are rarely in the top ten, but typically make it past the pruning. By limiting the number of categories to describe a sample then it reduces the number of inadvertent "hits" between dissimilar samples—much in the same way using only the most frequent words for the first vector. This then creates more distance between dissimilar samples.

Contextual Distance

Once both feature vectors have been created it is now possible to determine the distance between two samples. As stated earlier, standard techniques employ a sample space and used word frequencies to map a sample to the sample space. ContextNet, however, uses a graph structure to connect all documents. First, the Contextual Distance between each sample is calculated. Given a sample S_1 with two feature vectors F_1^{s1} and F_2^{s1}, created as described earlier, and a second sample S_2 with two feature vectors F_1^{s2} and F_2^{s2} and that the number of elements per feature vector are i_{s1} and j_{s1} for F_1 and F_2 respectively and i_{s2} and j_{s2} for S_2, then the distance between two samples is calculated as per Algorithm 1.

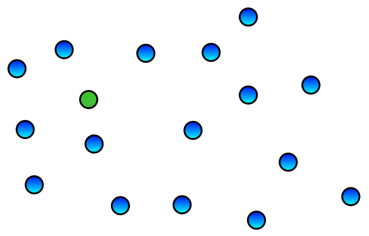
Essentially, Algorithm 1 calculates the intersection of elements between the two samples,

Algorithm 1 Calculate Contextual Distance

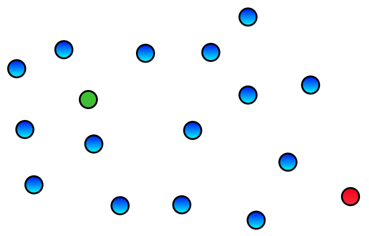
```
score = 0
for all $e$ in $F_1^{s1}$ do
    if $e \in F_1^{s2}$ then
        score++
    end if
end for
if $i_{s1} \geq i_{s2}$ then
    score = score / $i_{s1}$
else
    score = score / $i_{s2}$
end if
score2 = 0
for all $e$ in $F_2^{s1}$ do
    if $e \in F_2^{s2}$ then
        score2++
    end if
end for
if $j_{s1} \geq j_{s2}$ then
    score2 = score2 / $j_{s1}$
else
    score2 = score2 / $j_{s2}$
end if
distance = 2 - (score + score2)
if distance == 2 then
    distance = -1
end if
```

then normalizes the result by the larger of the two sets. This is done once for each feature vector. A result of 2 indicates that the two samples have nothing in common. In this instance, the value of 2 is replaced with a negative one to indicate that there is no intersection between the two samples. A result of 0 indicates that the two samples are identical, or that all the elements of the smaller sample are completed contained in the larger sample. A result between 0 and 2 indicates a varied level of intersection between these two samples. Each individual score is normalized to the larger set to ensure that calculations remain consistent.

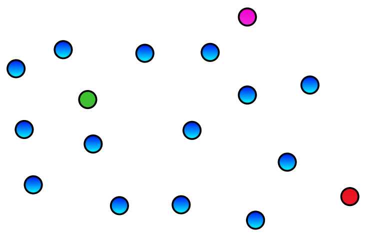
The distances between all samples is calculated and creates an $n \times n$ matrix where n represents the total number of samples. The value at any intersection within the matrix represents the distance between two samples. Essentially, we now have a graph of the samples. Since there exists the possibility that some samples have no distance to some samples (i.e. there was no intersection between feature sets) Dijkstra's algorithm is used to identify the shortest path between two disconnected samples. This is performed for all instances where there is no intersection between two samples. Further, the usage of Dijkstra's algorithm assumes the travel to disconnected nodes through intermediary nodes and thus preserves the triangle inequality. At completion, all negative values are removed from the matrix, except where the index i = i (i.e. the distance from a sample to itself). It should be noted that this approach could fail if a sample had no connection to any other sample in the sample set. In this instance, a disconnected graph would ensue and could skew results. In practice, we never noticed any disconnected graph as almost any sample will share at least one word or category with another sample. Thus, there is little probability that this will occur. However, current methods dictate removing any sample that fails to connect to the graph of all samples in order to preserve value of ContextNet. After all, the goal of ContextNet is to identify clusters of samples sharing contextual clues. If a particular sample shares nothing with other samples then it does not necessarily offer any new information. The only courses of



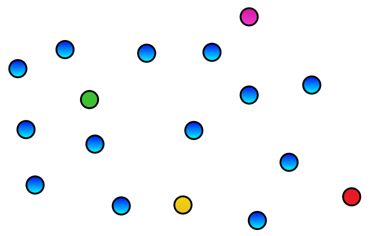
(a) K-center step one—choose random start.



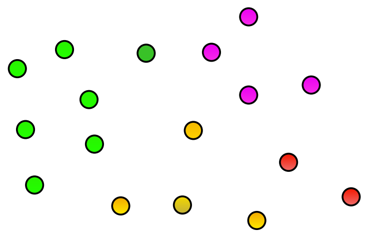
(b) k-center step two—Choose next center maximizing distance with current centers.



(c) k-center step three–choose next center maximizing distance with current centers.



(d) k-center step four–choose last center maximizing distance with current centers.



(e) k-center step 5—cluster to nearest neighbor.

Figure 5.4: k-center algorithm illustration with k=4.

actions are to ignore the sample (current method), place the sample in its own category, or attempt some alternate means to identify the sample. For now, we opt for the simplest approach and simply ignore the sample leaving the rest for future work.

The end result of the process is a fully connected graph where every sample is connected to every other sample, and each edge labeled with a distance determining the relative contextual proximity.

k-Center Clustering

To create clusters, ContextNet employs the k-Center algorithm. The k-center algorithm begins with the selection of a random sample as the starting point. This sample is added to the center set. At the next step, a new sample is selected such that the new sample maximizes the distance between the new sample, and all samples in the center set. The new sample is added to the sample set and

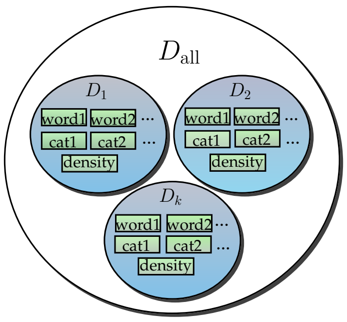


Figure 5.5: Illustration of the set of derived values from the content clusters.

this continues iteratively until the center set contains k values. Once the center set is complete, then all remaining samples cluster to the nearest center. Figure 5.4 illustrates this process. First, a random node is chosen as in Figure 5.4(a). Next, a new node is chosen such that it maximizes the distance with the first node as illustrated in Figure 5.4(b). Figure 5.4(c) and Figure 5.4(d) repeat the process. If we assume $k = 4$ then the center set is complete at Figure 5.4(d). Then, clustering occurs by having each unassigned node cluster to its nearest neighbor as illustrated in Figure 5.4(e).

It should be noted that the initial node choice can impact the results of the clusters. However, we evaluated the algorithm for use with ContextNet by simply iterating through all possible starting points to identify the best value to minimize the average contextual distance. Across all our experiments, including those detailed in Chapter 5.1.5, the ultimate difference between the minimized clustering and random clustering was minimal. As such, the random start selection for the k-center algorithm suffices for ContextNet's needs.

Clustered Samples

At the end of clustering, k clusters are derived and each cluster maintains a trio of attributes. First, each cluster maintains a list of the most frequent words and categories within the cluster. This is simply an aggregation of the feature vectors of the samples in the clusters. Secondly, each cluster maintains the ratio of samples from this cluster in comparison to the total number of samples. This may be used to determine the relative popularity of a cluster. Formally, each cluster D is a triple $D(\alpha, \beta, \delta)$, where α is a finite list of keywords aggregated from the cluster's keyword feature vectors, β is a finite list of categories aggregated from the cluster's categories feature vectors, and δ indicates the density of this cluster in comparison to other clusters. Further, we designate the set of these clusters as D_{all} with $D_{\text{all}} = \{D_1, D_2, \dots, D_k\}$ where k indicates the total number of clusters. Figure 5.5 illustrates this formalism.

It should be noted that it is not necessary to perform the classification process as outlined here. The content cluster set D_{all} can be generated synthetically as is described in Section 5.1.4. This allows significant freedom in building content clusters for a variety of possible uses. We address this more fully in Section 5.1.4.

5.1.3 The b-model

To more rigorously evaluate ContextNet we developed a model to simulate clustering behaviors. We adopt a modified version of the b-model originally demonstrated by Wang et al. [122]. In short, the b-model is a simple recursive algorithm for generating an unequal distribution—often used for generating self-similar workloads. The value of b in this model is the loose equivalent of the Hurst parameter with a value of .5 indicating relatively equal distribution and a value of 1 a single long burst. The algorithm works in a simple manner. Define a space within which the distribution will occur, typically a time period, and a set number of events, y, to occur. The object is then to determine where, within the entire space, those events will occur. This is done by cutting the space in half and sending $b \times y$ events into one half and the remainder of events into the other half; the choice of which half (right or left) chosen randomly with equal probability. This process continues

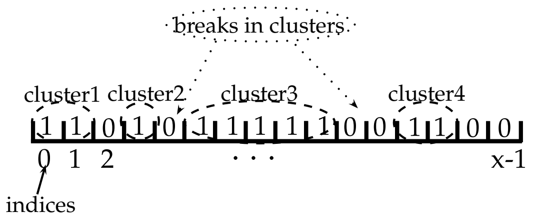


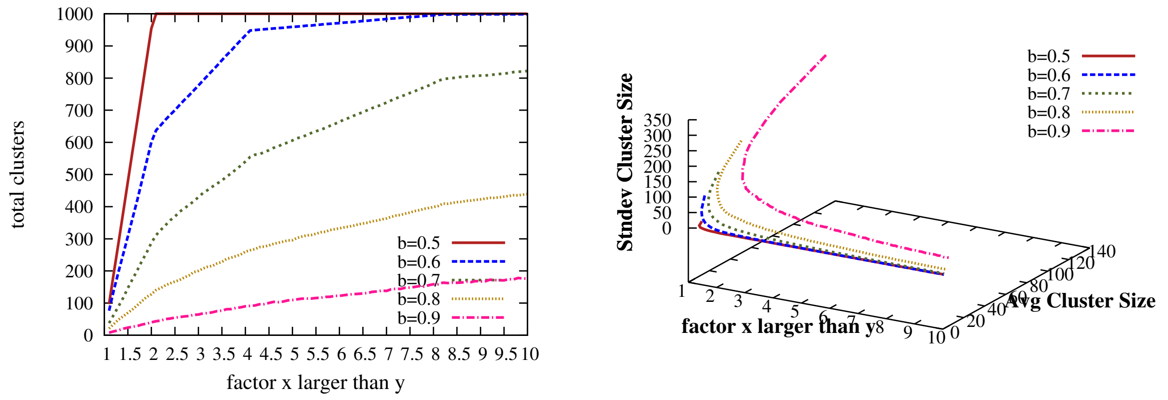
Figure 5.6: Illustration of the b-model filling a bit string x bits long to create clusters.

recursively until a minimum period is met (i.e. no more dividing the space). The events are then aligned within the space which may adjust the placement of other events within the space due to spillage off either side. When spillage occurs all events are simply shifted onto the space until all are accommodated which may result in events getting moved from their initial placement.

The key to employing the b-model for our purposes is to make the observation that a sample space might be considered a binary string x bits long. This binary string can be used to determine the number and size of clusters in a very simple way. If a bit is 1, then that indicates that a value exists at that index within the entire sample space. A consecutive string of ones within the bit string indicates a cluster while zeros within the bit string indicate a break in clusters. Thus, it is possible to count the total number of clusters simply by looking for all the distinct clusters of ones within the binary string. Further, this also allows the determination of the size of each cluster through merely counting the number of 1's within a particular cluster. Initially, the bit string is all zeros and is x bits long. The b-model is run on this space for y total values with $x > y$. As per the b-model above, when a value is added to the bit string at a particular index, that bit is simply flipped from 0 to 1. If a bit is already occupied with a 1, then a search for the next open spot ensues. The search iterates through the bit string first in one direction (either right or left from the initial spot) and continues until reaching either the beginning of the string or the end. If the search reaches an end without finding an open spot, it then returns and searches the other end of the string. By definition of $x > y$ a bit will be flipped eventually. Ultimately, this results in series of 1's and 0's in the bit string. A substring of 1's, one or more in length, indicates a cluster while each substring of one or more 0's indicates breaks between clusters. Depending on the value for b used, the result is something like Figure 5.6 which illustrates a bit string with $x = 15$ and y=10. Four clusters are evident in this figure though this need not be the case.

The b-model is impacted by the relative sizes of x and y in addition to the value for b. Obviously, the closer y is to x then the size of clusters will increase and the number of clusters will decrease. This extends from the fact that the values are crowded into a smaller space and are more likely to create contiguous blocks. Conversely, creating a very large space will allow the algorithm enough room to split the space more times and ultimately create more gaps between values which will result in smaller, but more numerous, clusters. Figure 5.7(a) illustrates the impact that the size of x and y have on the average number of clusters created by the b-model as used here. For this test 1,000 values ($y = 1000$) are inserted into a bit string that is 1,000 $\times$ the x-axis bits wide. The initial value for the x-axis is 1.1 extending out to 10 in increments of 0.1. The y-axis represents the average number of clusters resulting from applying the b-model given the above constraints and for the specified b value and aggregated across 100 runs of the algorithm. As is evident, the average number of clusters created increases as the size of the space increases. Further, it also increases dependent on the value for b. Figure 5.7(b) illustrates the average cluster sizes for this test. The size of the space has a significant impact on the average size of each cluster. However, as expected, it is the value of b that is the dominant factor in causing variance in cluster sizes. Essentially, the larger the b values, the much more significant the variance in cluster size.

The b-model can be used to model a variety of clustering behaviors and thus can serve to model the clustering of documents for ContextNet, or for users and servers in a system to be demonstrated later. The key to successful modeling is to determine the total number of samples



(a) Impact of overall space on number of clusters created for the b-model. These results represent 1,000 values placed into a bit string $x * 1000$ large where x is the x-axis.

(b) Average cluster size given variation in space size and b. Note: the z-axis represents the standard deviation.

Figure 5.7: Variability for creating clusters by the b-model.

and then the relative clustering behavior desired. Choosing the values for x and b will depend on the targeted clustering behavior.

5.1.4 b-Model Surfer

In order to evaluate ContextNet we devise a method for populating a set of clusters defined by D_{all}. However, automatically populating a sample space is not a simple process. The most often adopted approach is to simply use a corpus captured from a live network. If the tests are entirely in-house and properly secured then this may prove a viable approach. However, privacy concerns may abound. Also, there exists the potential that the corpus is not sufficiently robust for effective evaluation. Further, this approach is likely to require considerable investment in time to properly identify and cull the corpus. To address these issues we developed a content harvesting method we term the b-model surfer. The b-model surfer derives in part from the PageRank random surfer model [123] in part from the idea of the b-model [122], as already discussed, and in part from the ideas of Trestian et al. [111] which consider using other, publicly available, resources to gather information.

In essence, the b-model surfer mimics a web user making web searches. First, the b-model surfer creates a series of content clusters as per the b-model described earlier. For each cluster a term is pulled from a dictionary and WordNet is used to create a set of synonyms and hypernyms related to this word. These serve as the common context or subject of a particular cluster. Optionally, the feature vectors from a run of ContextNet may be used to generate these keyword lists. The b-model surfer utilizes a search engine to gather content using a randomly chosen key-word, or category, from a selected content cluster. The search engine used may be a public tool or one local to an organization. The results of the search are parsed, and the b-model surfer will randomly select one of the URLs and download the document. If the cluster needs more documents, then with a 60% chance another URL is downloaded from the same search results and with a 40% chance a new search is made from one of the terms for the cluster (the values for 60% and 40% are arbitrary here and could be changed to demonstrate slightly different browsing behavior). This continues until all the samples for a single category have been harvested. A number of documents are downloaded in this manner until a particular content cluster is filled. Then the b-model surfer will continue with the next cluster until the entire population of samples has been harvested. While our prototype only downloads HTML, the tool could be expanded to include other content types such as images, sound, and video without much difficulty.

The b-model surfer offers a simple method for populating the content of an evaluation with publicly available (or locally available) content. The b-model ensures that the content will follow a particular clustering policy and the fact that the content is publicly available mitigates privacy issues inasmuch as the content is not tied to users of any particular network. As such, it meets all three criteria for content (as stated in Chapter 3.1) in that the content will be largely appropriate to the test in the distribution of categories of content (if content clusters derived from actual content are used), variable (as the b-model can be changed), and maintain privacy (as all harvested documents come from a publicly available source). Of course, there are still some issues that need to be overcome with the b-model surfer. First, the search engine and search terms used can greatly impact the results. A poor search term or inadequate search engine may return either overly broad results, or no results at all. Secondly, it is possible that the search engine will skew the results of any search. This is partially mitigated by the fact that the b-model surfer is not tied to any specific search engine thus evaluators may choose the search engine that best suits their needs. Third, the b-model surfer does not mimic link following as per PageRank [123], though this is slated for future work. Finally, while the b-model surfer will protect individual user privacy in that no content is linked to real users of a system, it should be noted that ContextNet could well identify trends of a sensitive nature. In other words, sensitive keywords that imply misuse (i.e. pornography) will be retained in any clustering assuming enough such content is encountered. Thus, if there is substantive misuse in the sample network, then keywords representing this should propagate to clusters created by ContextNet and if those clusters are used by the b-model surfer then the sensitive content will likely find its way into the test. This is by design. User privacy is still maintained so long as safeguards exist to prevent evaluators from using the original data to identify the users responsible for the sensitive data. Further, content that ties directly to users on a system is unlikely to be promoted to a keyword by ContextNet. This stems from the fact that only the most common keywords for any cluster are preserved. Thus personal data is unlikely to make any keyword list unless this data appears in an anomalously large number of samples within a cluster. Even in the face of these deficiencies the b-model surfer offers a good tool for harvesting content for clusters.

Data Set	Random	Clustered
Reuters-21578	2.062086	1.885169
Web	1.663099	1.144325

Table 5.1: Average Contextual Distance.

5.1.5 Evaluation of ContextNet

To evaluate ContextNet we began with a small sample. Two data sets were created, one from the Reuters-21578 data set and one from random web pages. Both of the data sets were further subdivided into two further test-sets. In one test-set all the samples were from distinct categories and in the other all the samples but one were in the same category. The categories were determined by the Reuters-21578 data set or by manual verification. Each data set only contained 10 samples (i.e. a total of 40 samples across all 4 test-sets). For evaluation purposes, we define the metric average contextual distance as the averaged contextual distance between all nodes within the graph. Our insight is that the more random the samples, the larger the average contextual distance should be and the more clustered the smaller this value. Table 5.1 illustrates that this holds true for this simple test.

For further evaluation of ContextNET we employed the b-model surfer with a dictionary to download 25 web pages into a sample space 50 samples wide. ContextNet executed with k set to 5. The samples were run through ContextNet, then deleted, and then the process was repeated 15 times for each step in b-value (.5, .6, .7, .8, .9). The average contextual distance was computed at each run and further averaged across all fifteen runs with the results shown in Figure 5.8. Similar to expectations the average contextual distance demonstrates a downward trend as content clusters more tightly. However, there is a small spike at b = .9. This represents a failing in how the b-model surfer was implemented. Essentially, if a term for one of the clusters is a poor search term because it is either too broad or too restrictive, then poor searches result. A poor search may return no

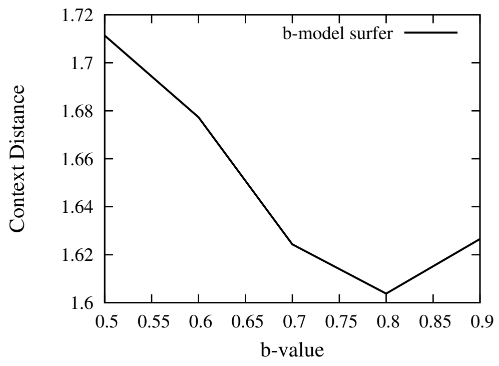


Figure 5.8: Downward trend in average contextual distance as clustering increases.

results or may return results completely unrelated to the category. Accounting for no search results is a simple matter of discarding a particular search term. However, accounting for overly broad search results is far more difficult. This was further compounded by the fact that for this evaluation we created generic clusters (i.e. the clusters were not based on a previous run of ContextNet) and the dictionary for search terms used was too large (more than 60,000 terms). Thus, for b-values of .9, only a few categories are used due to the density of one or two of the categories. Thus, if the search term for one of the dense clusters turns out to be a poor term (i.e. returns weird pages, too broad, too narrow) then the returned web pages may stray considerably from the intended category. When there are more categories as in the lower b-values, the impact of a bad search term is mitigated. Correcting this inconsistency is a matter of future work.

5.2 Producers and Consumers

The creation of content to mimic a network first requires the ability to contextually capture the essence of the network as was outlined in Section 5.1. However, this alone is insufficient. The content clusters must be brought together and combined with network Servers and Users to recreate a realistic atmosphere. Thus, the second step to successfully recreating traffic is to derive the behaviors of the Producers (servers) and the Consumers (users).

Deriving the producers and consumers for a particular network requires gathering statistics about communication trends. This aspect is also a popular mechanic of network traffic classification. First, we note that within a traffic capture it is possible to capture the set of hosts simply as the set of unique Internet Protocol (IP) addresses within the capture. However, distinctions need to be made between these IP addresses to determine if a particular IP address is a consumer, a producer, or both. Further, it is necessary to preserve clustering behaviors, in terms of popularity of a particular host, amongst consumers and producers. Finally, the consumers and producers need to link to the content groups. We will address each of these points in turn.

Distinguishing Consumers and Producers

First, it is important to note that a variety of means may be used to create consumer and producer clusters. One method is to use a list of clients and servers on the system as provided by network infrastructure documentation. Optionally, usage data could be derived from generalized user behaviors captured at a gateway. These data offer a ready-made means to cluster users into logical groupings. However, the actual behaviors of these users as demonstrated in network-level phenomena may exhibit high variance. As an alternative approach, we adopt a method to cluster consumers and producers by the relative "social" behavior of each identifiable host similar to that demonstrated in BLINC [109]. This ensures that clustered consumers and producers have demonstrated similar behaviors. Of course, it is possible to apply behavioral clustering after a logical clustering has already been applied to not only capture business-level (i.e. logical) behaviors but also usage trends (behavior). However, since logical clustering is purely ad-hoc in nature and dependent on an individual target network, we elaborate only on the behavioral clustering.

BLINC [109], a tool for classifying network traffic without payload inspection, examines

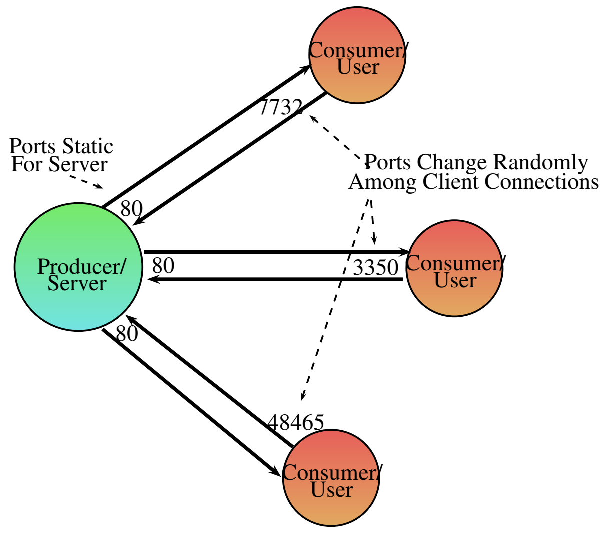


Figure 5.9: Distinguishing between Consumers and Producers.

traffic in a hierarchical fashion by utilizing the “social”, “functional”, and “application-level” behaviors of hosts within the network. In particular, BLINC makes the observation that most standard services (producers) will use a single source port when delivering content for multiple requests. Conversely, user machines (consumers) will use an operating system generated port for each distinct network flow. Figure 5.9 illustrates this idea. Given this, it becomes possible to identify consumers and producers by the ratio of of distinct source ports over the total number of flows for a particular host. The smaller this ratio, the greater the likelihood a particular host is a producer. While the larger the ratio then the greater the chance that the host is a consumer. In this manner it is possible to divide the set of hosts into two sets: consumers and producers.

However, there may also exists a set of hosts that do not neatly align to consumers or producers. This is often the case with hosts that are part of a P2P network [109] which act as both consumer and producer. While this may seem like a disadvantage, it is actually an added piece of information that can be well exploited. If a particular host rests in the middle, between the extremes, then it is possible to assume that the host is actually both producer and consumer. That host can then be added to both lists. During generation, some care must be made to prevent a consumer from consuming its own content but otherwise this scheme allows for a clean segregation of consumers and producers. For the purpose of clearly delineating consumers and producers, all hosts with a ratio of source ports to total flows above $1/3$ are added to the consumer lists and all hosts with a ratio less than $2/3$ are added to the producer list. Applying this methodology to one week of network statistics collected at the gateway to the Networking Research Laboratory (NRL) at Washington State University demonstrated only a very small amount of overlap between producers and consumers (about 2.6%). Otherwise, the method worked quite well in segregating the consumers from the producers.

Clustering Consumers and Producers

Once the consumers and producers have been identified, it is necessary to cluster each group based upon the general behaviors of those particular consumers or producers. BLINC [109] made use of the social aspect of hosts by clustering hosts according to popularity determined by the number of other hosts with which a particular host communicated. We expand on this idea by examining what we term the "super structure" of the network graph. Consider a network capture as a directional multi-graph where each vertex represents a host and each edge a unidirectional communication from one host to another, with potentially multiple edges between the same two hosts. Individual edges are defined by the Protocol and the TCP/UDP source port and destination port with the direction determined by the IP source and destination addresses. In this manner, it is possible to build a multi-graph of the network as illustrated in Figure 5.10.

Once the multi-graph is created, there are certain host behaviors that will exhibit themselves within the graph. First, more active hosts will maintain a much larger number of communications

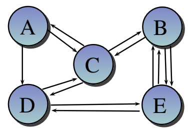
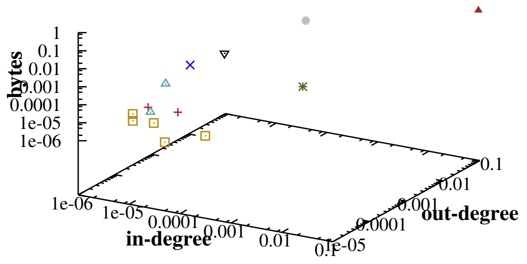


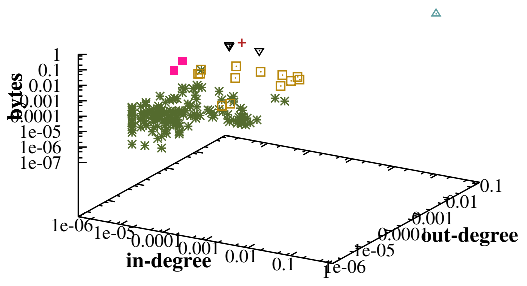
Figure 5.10: Example multi-graph of a network capture for a network with hosts A, B, C, D, and E.

that will manifest as increased in-degree and out-degree at each node. Thus, it becomes possible to cluster the set simply by the degree of activity attached to a node. This is further refined to include the amount of data into and out of the node to better differentiate heavier data flows. Thus, the in-degree, out-degree, and total bytes into/out of a host stand as three dimensions that can be applied to each consumer or producer. After calculating these dimensions and plotting them in a Euclidean space a simple k-means clustering can be applied to identify k clusters within each set of consumers and producers. An example of this is derived from one week of network statistics collected at the gateway to the NRL. The resulting data was segregated into consumer and producer groups using the techniques mentioned earlier, and then k-means clustering was applied to each set. The results are represented in the Figures 5.11(a) and 5.11(b). Note that the values for in-degree, out-degree, and bytes are normalized.

Clustering the data in this way preserves several phenomena embedded within the data. First, there is an unequal distribution of hosts to each cluster. In other words, most hosts are gathered into a few clusters, while most of the clusters have only a few values. Such unequal distributions are common to network behaviors and not necessarily captured when arbitrary grouping is employed. Second, choosing the correct value for k can often prove quite difficult for the most accurate classification. However, the goal in this clustering of data is simply to lump like with like. Thus, there is no need for a fine-grained classification of hosts. In our experiments with the NRL data we found little real difference in clusters produced with a value of k between 3 and 9—only



(a) Producers from NRL. Symbols mark clusters.



(b) Consumers from NRL. Symbols marks clusters.

Figure 5.11: Consumer and Producer Clustering.

more small clusters, no real change to the overall shape of the clusters. While the exact clustering will depend largely on the data set, the real objective of this clustering is to ensure that all hosts are part of a cluster more than that these clusters meet rigorous classification criteria. Ultimately, the evaluator must decide the level of precision necessary for an actual test. Finally, we note that this clustering method could serve to augment approaches that utilize available business data such as user groups. Thus, it becomes possible to cluster within a large logical category and retain both the logical underpinnings of a group as well as the network behaviors.

Assigning Content

Once the consumer and producer clusters are created, it is necessary to connect those clusters to individual content clusters as derived in Section 5.1. Arbitrarily assigning content clusters to consumer or producer clusters would not preserve the distribution of content amongst consumers and producers. However, those distributions may contain important trends or phenomenon. In order to preserve this data, it is necessary to quantify the sample-set such that each sample clustered as per Section 5.1 must also tie to at least one consumer and one producer. Such a requirement

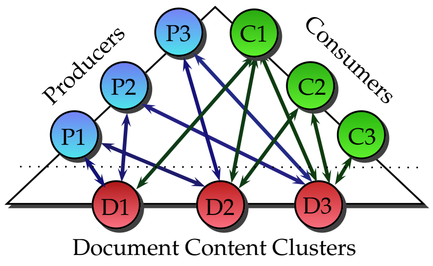


Figure 5.12: Illustration of the Generative Pyramid.

is viable when examining a sample directly from a target network, though may require secondary means of classification when dealing with a corpus of samples not derived directly from network traffic. Regardless, once the content, consumer, and producer clusters are known it is then a simple procedure to connect each consumer cluster to a content cluster and producer cluster to a content cluster. In fact, the relative frequency with which one cluster interacts with another can be used to simulate the likelihood of a particular content cluster applying to a given consumer or producer. This, in fact, is the basis of the generative pyramid discussed in Section 5.3.

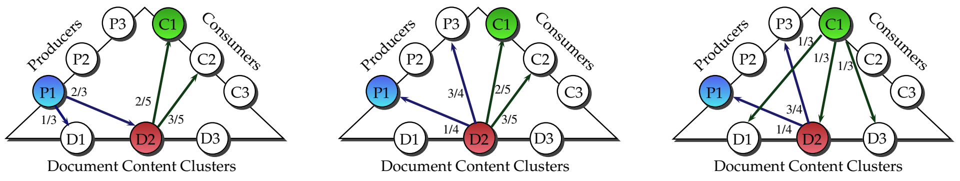
If it occurs that the sample corpus has no direct connection with any consumer or producer, then it becomes necessary to arbitrarily assign consumer and producer clusters to content clusters. A simple model for generating unequal distributions is the b-model. Using the b-model in this manner is elaborated in Section 5.3.1. The benefit of using the b-model in this instance is that it would allow an evaluator to systematically adjust the distribution trends between consumers and producers and the content clusters.

5.3 Generative Pyramid

The generative pyramid is the centerpiece of the content generative model. Essentially, given a set of content clusters as derived in Section 5.1, a set of consumer and producer clusters as derived in Section 5.2 and the ability to connect consumers and producers to content, then a pyramid can be formed. Figure 5.12 illustrates this pyramid with content clusters at the base, consumers on another angle, and producers the last. In fact, connecting the content between consumers and producers creates a bipartite graph. The content clusters serve as the middle ground for consumers and producers. The key to making this model work relies in the connections between consumers and content and producers and content.

The pyramid works in the following manner. In Figure 5.12 there exists a connection between $P1$ and $D2$. Notice that $P1$ is also connected to $D1$. From the assignment of content to producers as per Section 5.2 it is possible to derive the relative distribution of documents from that producer that fall into content clusters $D1$ or $D2$. Imagine that $1/3$ of all documents produced by $P1$ fall into the content cluster $D1$, and the remaining $2/3$ fall into the content cluster $D2$. As such the directional edge $P1D1$ could be weighted with the value $1/3$ and edge $P1D2$ with the value of $2/3$. Conversely, notice that $D2$ is connected to $P1$ and $P3$ as well as $C3$ and $C2$. Since the producer and consumer sets are known, it is possible to distinguish between a consumer and a producer. This allows $D2$ to consider its interaction with producers in isolation to consumers. As such, if $1/4$ of $D2$ content comes from $P1$ and $3/4$ comes from $P3$, the the directional edges $D2P1$ and $D2P3$ would be weighted $1/4$ and $3/4$ respectively. In addition, $D2$ might also demonstrate $2/5$ and $3/5$ of its content consumed by $C1$ and $C2$ respectively. Finally, the distribution of content consumed by $C1$ might be evenly distributed amongst all three content clusters $D1$, $D2$ and $D3$.

Once the pyramid is weighted in this manner, it becomes possible to probabilistically determine a combination of consumer, producer, and content by simply starting with one (consumer, producer, or content) and probabilistically choosing edges to other clusters. Figure 5.13 illustrates three different methods in which the combination of $P1$ $D2$ $C1$ could have been chosen given a different starting point with Figure 5.13(a) starting from $P1$, Figure 5.13(b) starting from $D2$ and



(a) Finding content and consumer from producer. (b) Finding producer and consumer from content. (c) Finding producer and content from consumer.

Figure 5.13: Example use of Generative Pyramid.

Figure 5.13(c) starting from $C1$. Even better, the initial cluster can be chosen based on distributions (i.e. more frequent consumers, more common content, etc). A further benefit of this approach is that a simulator or traffic generator need only determine that a communication is to take place based on low-level workload models. Once the presence of a communication is determined, the pyramid can be used to derive the consumers and producers and content. In the event that the low-level model has already determined a consumer or producer, or even both, it is still possible to use that information to determine the content, or vice versa.

More practically, the generative pyramid can be reduced to a directed incidence matrix with n rows (one row for each producer, consumer, and content cluster) and m columns (one for each edge in the pyramid). The intersection between any edge and one of the vertices of that edge is marked by the probability that given this vertex this edge will be chosen. In other words, it marks the outgoing edges. An example of this incidence matrix is illustrated by Table 5.2. Since the consumer and producer clusters are known, they can be listed within the columns of the matrix such that all edges for producers come before all edges with consumers. This is a simple method for organizing the matrix and maintaining what amounts to two separate matrices. Note that the probabilities for each consumer and producer cluster sum to 1 and that for each content cluster the sum is 2 when examined across the entire row, but 1, when examined only between consumers or producers.

Table 5.2: Example of incidence matrix of the Generative Pyramid.

Cluster	P1D1	P1D2	P2D1	P2D3	P3D2	P3D3	C1D1	C1D2	C1D3	C2D2	C2D3	C3D3
P1	$1/3$	$2/3$	0	0	0	0	0	0	0	0	0	0
P2	0	0	$2/5$	$3/5$	0	0	0	0	0	0	0	0
P3	0	0	0	0	0	0	0	0	0	0	0	0
D1	$1/4$	0	$3/4$	0	0	0	1	0	0	0	0	0
D2	0	$1/4$	0	0	$3/4$	0	0	$2/5$	0	$3/5$	0	0
D3	0	0	0	$2/3$	0	$1/3$	0	0	$1/6$	0	$1/2$	$1/3$
C1	0	0	0	0	0	0	$1/3$	$1/3$	$1/3$	0	0	0
C2	0	0	0	0	0	0	0	0	0	$3/5$	$2/5$	0
C3	0	0	0	0	0	0	0	0	0	0	0	1

The reduction of the relationships between consumers, producers, and content clusters to the directed incidence matrix is a powerful abstraction. The methods explored in Section 5.1 and Section 5.2 provide a meaningful method to produce the consumers, producers, and content clusters. Further, the connections between these clusters can be ferreted from network traces. Thus, the directed incidence matrix of the generative pyramid provides all the necessary information to tie content to any simulated flow. Further, this model preserves clustering trends inherent in the consumers, producers, and content of a targeted network. As such, the generative pyramid maintains by default, if the methods from Sections 5.1 and 5.2 are employed, appropriateness to a given target network. In the following sections we will explore how this model can be tuned to provide variability and to maintain privacy.

5.3.1 Employing the b-model for Clustering and Content Harvesting

The generative pyramid is a model that is ripe for variability. The simplest variance that may be exerted over the model is changing the probabilities within the directed incidence matrix. This will have an immediate effect of changing the overall content resulting from the tests and may be used to study extremes within content generation. Examples of such tests would be to adjust the probabilities such that all content originated from only one or a few producers and then perform consecutive tests when the distribution of producers moved closer to an equal distribution.

The b-model can be used to model the general clustering behavior of consumers, producers, or content. The key to successful modeling is to determine the total number of samples (consumers, producers, or content samples), and then the relative clustering desired. Choosing the values for x and b will depend on the targeted clustering behavior. The most straightforward approach is simply a brute force evaluation using least-squares to identify the best parameters given a desired clustering behavior. This is illustrated in Section 5.4. More importantly, the b-model allows for a range of clustering behaviors to be examined. Systematically manipulating the value for b can be used to investigate the impact of greater or lesser clustering within consumers, producers, or content. In fact, Figure 5.7(b) in Section 5.1.3 very nicely illustrates the range of clustering behaviors that might be explored with the b-model.

5.3.2 Allowing for Privacy

The b-model is useful for creating clusters, but those clusters still need to be populated. For consumers and producers this only requires that a set of IP addresses be designated and then these addresses be added to the clusters randomly, or under certain distribution guidelines. The set of IP addresses could be generated as illustrated by Sommers et al. [19], or could come from network captures as illustrated in Section 5.2. After which, we assume that there exists a method to map the addresses to clusters. This assumption is not particularly demeaning as the simplest approach is to match the list of IP addresses, in order, to the on-bits in the bit string produced by the b-model. In doing so, the IP addresses will naturally be added to clusters. However, we note that there may be subtleties in how the IP addresses need to be distributed and leave the investigation of such to future work.

5.4 Evaluating the Content Generative Model

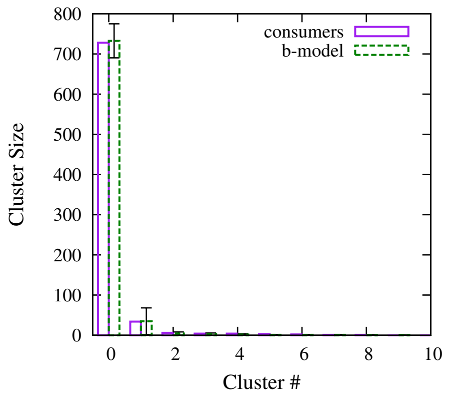
We evaluate the approach against the three primary constraints of Appropriateness, Variability, and Privacy for generating content as stated in Chapter 3.1 and restated again in this Chapter.

5.4.1 Appropriateness

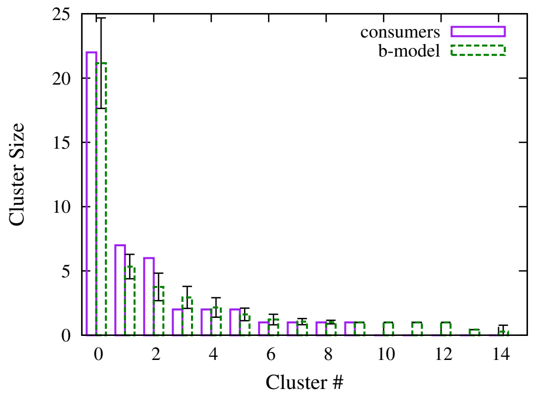
The Appropriateness of this approach follows that if the clusters are derived directly from network captures as illustrated in Sections 5.1 and 5.2 then they reflect, at least in the general clustering behaviors of content, consumers, and producers, the target environment. The b-model surfer allows for the harvesting of documents that can roughly meet the clustering as exposed by ContextNet. We illustrated the downward trend in the contextual distance between the documents harvested under higher values for the b-model in Section 5.1.4. What this evaluation demonstrated is that as documents became more clustered (i.e. fewer clusters and more documents per cluster), then the average distance between all documents shrinks. In other words, this means that there are more documents that are more closely related and is evidence that the b-model surfer can maintain the overall clustering behaviors of the target environment, though not necessarily with fine accuracy. In fact, we point out that the evaluation in Section 5.1.4 also demonstrated that poor search terms can have an adverse impact on the clustering. This implies that the b-model surfer will not perfectly fit content to the clustering trends.

However, we note that the primary goal of the generative pyramid is to provide content for a simulation that mimics that which might appear on a target network but does not duplicate it. Since the categories used for each cluster created by ContextNet are aggregates of the most common terms across the cluster it can still be expected that there is some degree of variance within the cluster. As such, variance by the b-model surfer due to search engine results or poor search terms is not out-of-line with what might occur in the target network. Thus, as long as such does not occur with too much frequency then the clusters created by the b-model surfer will still maintain a general theme similar to the clusters created by ContextNet. Further, the generative pyramid provides a coherent model by which content can be harvested that will still attempt to retain some of the statistical qualities of the content as seen in a target environment and, as such, is far superior to current models that rely completely on evaluator expertise. Ultimately, there is, as yet, no perfect fitness test for content produced or harvested under any method. At least ContextNet and the b-model surfer offer a means to approximate general themes but does not yet have the power to perfectly mimic content as might be seen on a target environment. We leave further pursuit of the fitness of b-model surfer results to future work.

Second, we examine the b-model as capable for fitting the clustering patterns of Consumers and Producers of a target network. That is, given a set of clusters (just number and size of each cluster) we examine how well the b-model can approximate the clustering behaviors. To do so, it is only necessary to derive the best parameters to fit the model to the desired target environment. Thus, we attempted to fit a b value to the clustering of NRL data as illustrated in Figures 5.11(a) and 5.11(b). To arrive at the best fit, we simply aggregated the results of running the b-model 50 times for a particular set of b, x, and y values and used least squares to determine the closest fit to the clustering exhibited in the NRL data. This fitting was accomplished by taking the clusters for the NRL data, sorting each cluster from greatest to least by size of cluster, doing the same with the clusters generated with the b-model and then calculating the square of the difference. We consider the NRL data as the vector $A = \{a_1, a_2, ..., a_n\}$ where each a_i represents the size of the i^{th} cluster for the n total clusters in the NRL data, and $B = \{b_1, b_2, ..., b_m\}$ where B is the set of clusters generated by the b-model and each b_j represents the size of the j^{th} cluster for m total clusters. If we designate l as the greater of m and n and set $a_i = 0$ for any $i > n$ and set $b_j = 0$ for any $j > m$ then the fitness is calculated as Equation 5.1.



(a) Best fit b-model to consumers from NRL with b=0.85, x = 1.1, and y = 783. Error bars mark 1 standard deviation.



(b) Best fit b-model to producers from NRL with b=0.6, x = 1.3, and y = 45. Error bars mark 1 standard deviation.

Figure 5.14: Fit of b-model to Producer and Consumer clustering behaviors.

$$\text{Fitness} = \sum_{i=1}^{l} (a_i - b_i)^2 \tag{5.1}$$

Figure 5.14(a) illustrates how closely the b-model is able to match to the target for consumers and Figure 5.14(b) the same for producers. As can be seen in the figures, the b-model can be tailored to approximate, with reasonable accuracy, the general cluster size and number of clusters for a particular data set. The calculation of the best parameters required roughly 15 minutes for the consumers and about 11 seconds for the producers. For very large data sets this process could require considerable time. However, this is a task that need only be performed once. Unfortunately, the randomness involved in the b-model means that there can be considerable fluctuation. First, the b-model has a tendency for creating more clusters than the actual set. The actual number is somewhat random due to the process of the algorithm, but will often exceed the desired number of clusters. Figure 5.14(b) illustrates this very clearly. This represents a small failing of the b-model for this purpose. Luckily, these extra clusters are always small in magnitude (size of 1). These extras can either be ignored, used as they are, or simply subsumed by the larger clusters.

Secondly, the b-model can fluctuate in the cluster sizes of the larger clusters. This is also evident in the figures as the error-bar marks the standard deviation. While this fluctuation will often cause the clusters to not match perfectly to the target, the order of magnitude difference is typically not beyond reason. Further, the general shape of the clusters in terms of largest to smallest clusters is typically preserved. On average, the b-model is capable of approximating a target clustering behavior. The implications of this is that the clustering behaviors of Consumers and Producers can then be reduced to the parameters for the b-model. This can allow for the easy simulation of Consumer and Producer clustering as illustrated in the next section. In fact, the b-model can also be used in matching the general clustering behavior of content given the necessary parameters. Then, content can be generated that might have absolutely no relation to the target network other than in general clustering behaviors. This is can be a valuable asset in preserving privacy as well as varying actual content in order to broaden the impact of evaluations

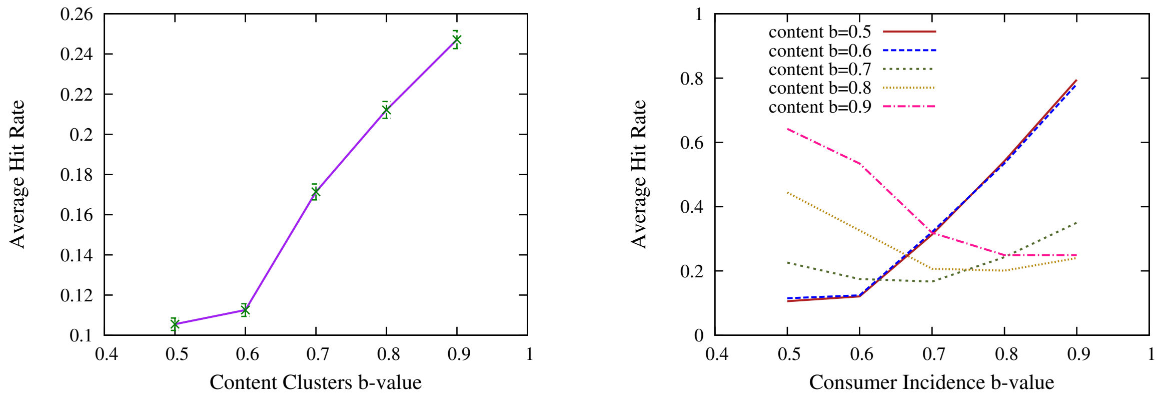
5.4.2 Variability

The content pyramid is a extremely powerful model where variability is concerned. To demonstrate this, we employ a simple example network caching algorith. The network caching algorithm was chosen for two primary reasons. First, it demonstrates that the techniques in this chapter can extend beyond NIDS to really any traffic evaluation scheme. Secondly, while the content generated under the generative pyramid can be used to create the feel of a target network without the need to scrub the data, it should be noted that it is difficult to say exactly how the harvested content might impact a NIDS. Thus, a caching algorithm presents a clearer example of how content can be varied to impact applications. However, this example represents only the potential for the generative pyramid and is by no means conclusive.

Regardless, we start by implementing a simple caching algorithm and then use the b-model surfer to harvest varying sets of documents with which to test this algorithm. By varying the clustering of the documents it is possible to increase the likelihood that the same page is harvested. The caching algorithm works like this. When a document is requested, the cache is checked to see if the document already exists in the cache. If it does, the caching algorithm counts this as a hit. If not, then the document is added to the end of the cache, potentially removing the first document in the cache if the cache size of 10 has already been reached. The average hit rate is then the number of cache hits divided by the total number of requests.

To evaluate this simple caching system, we utilize the b-model surfer at varying values for b to harvest 100 documents and create a space of documents that a user might request. There are no predefined clusters, only those that result due to the b-model. Further, the b-model surfer is restricted to a smaller set of search results encouraging the chance that, when a large cluster occurs, a single document may be selected multiple times. Such redundancy is retained within the content clusters—this embeds the likelihood of a particular document occurring more often than others directly into the cluster. We assume a single producer to which all requests are made. Further, we assume a single cluster of consumers making requests, each request utilizing the distribution of documents as the probability for choosing a particular document (i.e. a cluster with more documents will be selected more often). A series of 10,000 web requests are simulated in this manner. The intuition to this test is that as documents become more clustered, the hit rate should climb. In fact, this is the case of the 100 test runs aggregated and illustrated in Figure 5.15(a).

Figure 5.15(a) represents a good example of how the b-model can be used to systematically examine the impact of clustering against a network application. However, the potential for advancing the scenario is even greater. Adjusting the incidence matrix can further impact the results in non-intuitive ways. To illustrate this, we design a more advanced evaluation using the content clusters generated from Figure 5.15(a) and then adding multiple consumer clusters and adjusting the



(a) Average cache hit rate for documents gathered under the set value for b.

(b) Impact of adjusting consumer incidence probabilities.

Figure 5.15: Example usage of Content Generative Pyramid.

incidence matrices for each consumer cluster. To simulate the consumer clusters, we once again adopt the b-model to distribute 1,000 IP addresses amongst a varying number of clusters. While the number of consumer clusters will have limited impact on a caching algorithm in most circumstances, the probabilities linking consumer clusters with content clusters will have a significant impact on the results. To simulate this, we also apply the b-model to the incidence matrix between each consumer cluster and content cluster. To accomplish this, we use the b-model to create a clustering across 100 possible values. We use the size of each cluster generated in this manner to indicate a probability of this consumer choosing content from a particular content cluster. In order to adopt a little more order into the evaluation, we sort these clusters in descending order by size. The content clusters gathered under a particular b value are also sorted in descending order by size. These two sorted lists are aligned and the probability at a given index in the consumer incidence list becomes the probability of that consumer choosing that content cluster. The incidence matrix is then filled out with the values derived in this manner. Essentially, this weights the probabilities such that the largest probabilities align with the largest content clusters.

For the test we simulated 1,000 web page accesses by first randomly selecting a consumer from one of the consumer clusters. A random number between 1 and 100 is chosen and given this number, and the incidence matrix, it is possible to determine a target content cluster. A document is then randomly chosen from within the targeted content cluster and lastly the producer is selected (only one producer for this test). Figure 5.15(b) illustrates the aggregated results of 10 test runs. The adjustment of the incidence matrix has had a significant impact on hit ratios. First, notice that for incidence probabilities created under a b-model of 0.5 and 0.6, the hit rate for the documents gathered under b values of 0.8 and 0.9 are extremely high. The root cause of this phenomena stems from the fact that the clustering of documents in these instances consists of one or two large clusters and then several clusters of a single document. Since each cluster is chosen nearly equally when the incidence matrix probabilities are created using 0.5 or 0.6, then it becomes more likely that the clusters with only a single document are chosen. This directly results in the higher hit rates. However, for documents gathered under a b value of 0.8 and 0.9, and when the incidence matrix probabilities are created using a b value 0.8 or 0.9 the exact opposite happens. Essentially, the largest clusters become the target and the results then quickly approach the clustering inherent within the cluster as witnessed in Figure 5.15(a). Conversely, as the b values used in creating the incidence matrices grows, the documents clustered around lower b values (0.5 and 0.6) show increasing hit rates. This is a direct result of the fact that the largest content clusters in these instances contain 2 or 3 documents. As more and more traffic is directed to those clusters, the chance for a cache hit rises appreciably.

This example could be further expanded to include multiple producer clusters as well as different incidence matrices. Of course, the model extends well beyond the testing of caches as illustrated here. The harvesting of content, the clustering of consumers and producers, as well as the incidence matrix serve to allow for a huge amount of variety in potential evaluations beyond that explored here. This example merely illustrates how the model can be manipulated to create interesting effects on the application under evaluation. Ultimately, the model is only limited by the imagination of the evaluator.

5.4.3 Privacy

Finally, we remark on the privacy concerns mitigated by this approach. First, content harvested from public resources can be considered not only realistic (it is actual content), but public and thus not sensitive in nature. However, there are some potential privacy issues that may arise as discussed at the close of Section 5.3.1. In particular, there exists the potential skew from search engines or search terms and the chance for sensitive terms appearing as categories or search terms in results. The problem with search engines is partially mitigated as the b-model surfer is not tied to any particular search engine. The problem with search terms is more complex and the subject of future research. Finally, we note that the chance for sensitive terms appearing in searches is no more likely in the approach provide here than in current approaches. In fact, in most cases it will be less likely unless there is serious misuse in the system, or sensitive content is encountered repeatedly within the target network. As such, actual network captures and statistics must be maintained under strict procedures in order to reduce the potential for privacy infringement. However, the b-model has the potential to forgo all privacy concerns if the evaluator only wishes to use the general content clustering behaviors and not the actual content categories. In that instance, all that is needed are the necessary parameters for the b-model surfer to harvest content and a dictionary of search terms to use. Under this model of usage the b-model surfer offers completely anonymous content and cannot infringe on user privacy of the target system.

While the methods provided here mitigate the larger concerns of user privacy more advanced testbeds, training simulations for actual network security for example, require extremely realistic traffic. This will typically require email, voice over IP, and potentially other content types.

Harvesting such documents can still be handled under the framework provided here—though the documents must undergo an extra stage of cleaning. Essentially, a set of example documents must be harvested however possible. These documents must then be vetted dependent their potential for sensitivity. We posit that ContextNet (Section 5.1) working with a set of training data could serve to capture most of the sensitive content. The remainder would then need to be vetted by an evaluator before employing the framework provided here. This is not a perfect solution and we seek to further improve this process in future work. Regardless, the research here provides for a systematic framework by which content may be harvested, the relative clustering statistics maintained, and, in most cases, privacy concerns mitigated.

CHAPTER six

IMPLEMENTATION AND CONCERNS IN HIGH-SPEED ENVIRONMENTS

Generating traffic and content as illustrated thus far can prove a resource intensive process. Trivial amounts of traffic can be created with little trouble, but truly recreating a large network can require either powerful hardware or lots of hardware depending on the exact parameters of a test. We have attempted to alleviate some of these issues through the creation of PaCSim, the Packet Capture Simulator. The purpose behind PaCSim is that it allows the creation of captures that follow the benchmarking techniques illustrated in Chapter 4 as well as providing for the usage of content generated through Chapter 5. Thus, PaCSim allows the creation of simulated packet captures to evaluate NIDS in off-line, or even on-line mode using a traffic replay device. However, while PaCSim can greatly reduce the resources required to create large tests, we also note that generating a single test can create very large files which may not provide high-speed replay. Thus, it is necessary to consider methods that might be used to improve the regeneration characteristics of any tests. We start by explaining PaCSim, and then examine the hardware constraints typical in a high-speed regeneration environment. Finally, we illustrate our methodology for reducing the size of generated content in order to improve efficiency during regeneration.

6.1 The Packet Capture Simulator

The Packet Capture Simulator is a tool that can be used to simulate Network Traffic Captures. The tool is an agent-oriented simulator such that network actions are the results of particular agents attempting to complete a particular objective (i.e. download a web page). The underlying model follows the diagram in Figure 6.1. Essentially, when an agent becomes active, that agent will select a particular objective. Each objective then details the actual communications that need to take place

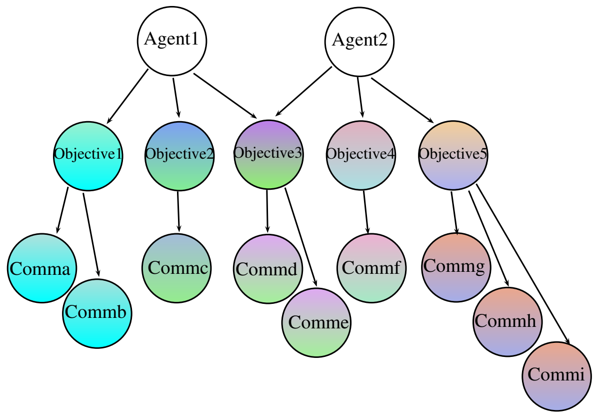


Figure 6.1: Illustration of PaCSim Agent Model.

in order to complete the given objective. The choice for each each objective is determined by a predefined distribution for each agent. Further, the actual communications attached to an objective can be designed to demonstrate a variety of behaviors matching most of the typically simulated network phenomena like inter-arrival times and packet length distributions.

Agents: The Agents of PaCSim designate the users of the system. Agents are directly tied to host IP addresses. In essence, the Agents perform the role of Consumers in the system. As such, the Agents can be set to follow various behavior by limiting the number of objectives which they pursue and modifying the think times between each Activity, with an Activity indicating an agent that has gone active. In this manner consumer distributions can be maintained, as well as think times (if desired). Further, multiple agents of a particular type can be created during a particular scenario and each will act independently.

Objectives: Each Objective defines a particular network task. This task might be to simulate a web-page download, or a Distributed Denial of Service Attack. The objective will define the potential targets, or Producers, that might be involved in that objective. Further, each objective will link to available communications. These communications may be set to occur in parallel or in order. Thus, the objective will largely define the actual behavior of the Agent as it interacts with a target. In essence, the objectives act like the Generative Pyramid Model in that they tie together the Consumers (i.e. the Agents) the Producers (i.e. the Targets) and the content (i.e. the Communications).

Communications: Communications are the low-level implementations, the network flows, that result from an Agent's interaction with a Target. Each Communication defines features such as the number of bytes, ports to use, and inter-arrival times. The definitions for these Communications are such that a single definition can provide for a wide degree of variety. Further, each Communication also defines a particular type of content. This may be a sample file as might be used in Chapter 5 or a finite automata representation of a rule-set as per Chapter 4. Regardless, the communication defines the payload for this communication.

The Simulator: The Simulator works in the following process. First, it randomly schedules the first Activity for each Agent. This scheduling is done through an iterative random selection. Essentially, each Agent has a probability to go active. For each Agent, the simulator starts at the first time-step and randomly selects a number. If the random number is equal or less than the "go active" probability for the Agent, then the Agent is scheduled to start at that time. If the random number is larger than the the go active probability then the time step is incremented and the process repeats at the next time step. This continues for each Agent until the Agent is scheduled or the time-step exceeds test parameters.

Once, all Agents are scheduled the simulator simply takes the first Agent in the schedule and chooses an Objective for that Agent. Choice of Objective will depend on the definition of the Users within the test and may be completely random or restricted to very specific objectives.

Regardless, once the Objective is chosen, the Simulator will identify all Communications necessary for that task. The Simulator will then proceed to simulate the Communications by building packets dependent on the definitions of the Communications. Each packet is pushed onto a heap, ordered by the time stamp for that packet. The timing data for that Agent is incremented dependent on the Communication features (i.e. inter-arrival times) and the test parameters (i.e. designated link speed). It should be noted that at this stage the packets are just the IP and layer 3 headers, payload and Ethernet headers are added later. Regardless, once all the Communications are completed that particular Objective is considered complete and the Agent's activity at an end. The Agent is then rescheduled for a new activity in an identical manner to the first activity, though the initial time-step will be after the end of the Agents last Objective. The next Agent is then pulled from the scheduled Activities and the process continues. Essentially, the Simulator executes every communication in isolation, and the ordering of the heap structure imposes relative order on the packets.

Once all Activities are converted into Communications and packets are pushed onto the heap, then the packets are written out into the a packet capture. As they are written, payloads and Ethernet headers are generated depending on the Communication definitions. Thus, one set of packets might possess random character payloads, while another set might demonstrate payloads as per the rules in Chapter 4. Further, the proper checksums are also calculated at this stage. Once all the packets have been written, then the process is complete.

Benefits of PaCSim: The primary benefit of PaCSim is a packet capture that meets the content generative models as outlined in Chapter 4 or Chapter 5. However, it should be noted that for PaCSim to meet the generative pyramid model as outlined in Chapter 5 would require building a test with this in mind which would require an additional stage of implementing the Consumer, Content, and Producer clusters. In other words, PaCSim, by itself, will not calculate those distributions. It will only preserve them in simulation. PaCSim also incorporates the ability to simulate many of the other features of traffic generation as discussed in Chapter 1.2. Further, PaCSim can run on commodity Linux hardware, though for very large tests processing can require considerable time. Finally, a packet capture can often serve as a first-run evaluation for a NIDS in order to determine the fitness of a particular technique or method. Even more, having an evaluation as a packet capture means that the test can be re-run again and again and should garner identical results. Thus, PaCSim offers some very attractive means for evaluating NIDS in particular, as well as most network applications.

PaCSim and High Speed Transmission: While packet captures have some strengths, in high speed transmission this can prove a weakness. First, the size of the packet capture can grow huge. This can make retransmission of such a file problematic as it may not be possible to put all the data in memory. Further, even if the entire capture is in memory, there exists a large chance that cache misses will occur regularly as the content and headers will change quite often since most network traffic is comprised of relatively short flows (10 packets or less) while most packets come from large flows (larger than 100 packets) [17]. This can create a large amount of churn in caching and result in less than optimal retransmission. We address these points in the following Sections.

6.2 Hardware Considerations

One of the primary constraints to high-speed regeneration of traffic is that the hardware platforms upon which these process can run have limited resources. First, we note, as discussed in Chapter 1.4, that there are three primary methods for generating traffic. As was also illustrated in that chapter only hardware implementations can meet Gigabit line-speeds (excepting distributed traffic generation). One of the problems behind this is the shear volume of data required to run a test. For example, 1 Gigabyte of data, ignoring any data structure overhead, can provide roughly 8 seconds of traffic at a one Gigabit line rate. Thus, a sixty second test would require 7.5 Gigabytes of data. While it is possible to simply take a few seconds of data and replay that data over and over, that type of evaluation is not really sufficient to evaluation as outlined in this work. Thus, even a small test is likely to require several Gigabytes of data and a large test may well require Tera-bytes of data.

Unfortunately, specialized hardware platforms with large stocks of fast RAM are extremely expensive, while other hardware platforms such as the NetFPGA Virtex II card are extremely limited in the amount of data that can be loaded onto the device. For example, the Virtex II NetFPGA card has 4.5 MB SRAM and 64 MB DDR2 DRAM. As such, it is impossible to load onto that board more than a small amount of data at any one time. Further, the Virtex II NetFPGA card has 4 Gigabit Ethernet ports. The bottleneck is clearly storage in this case. Thus, we examine methods for reducing the size of captures during retransmission in the following section.

6.3 Simplifications for High-Speed Environments

As we have attempted to justify, the size of a packet capture generated by PaCSim, or similar approach, for use within evaluation of a NIDS is simply too large for many platforms to efficiently handle. To address this we examine methods to reduce the size of the data. First, we note that in network traffic there is a large amount of redundancy in the form of packet headers. While the number of packets in a traffic capture may number in the tens of millions, the number of distinct flows (as defined by the IP-level quintuple) typically only numbers in the thousands or hundreds of thousands. We exploit this distribution to arrive at significant savings in the amount of data required for an evaluation.

First, let us consider a potential evaluation. This example evaluation seeks to generate

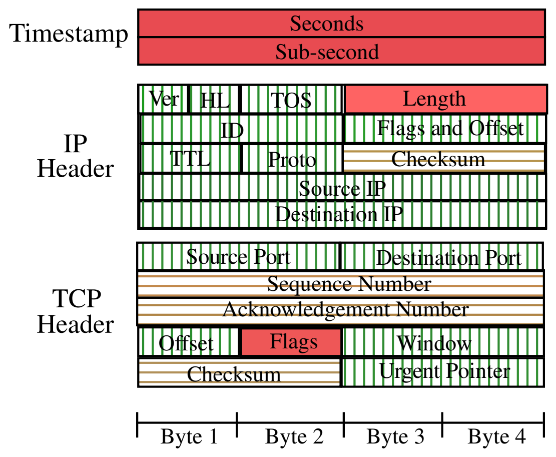
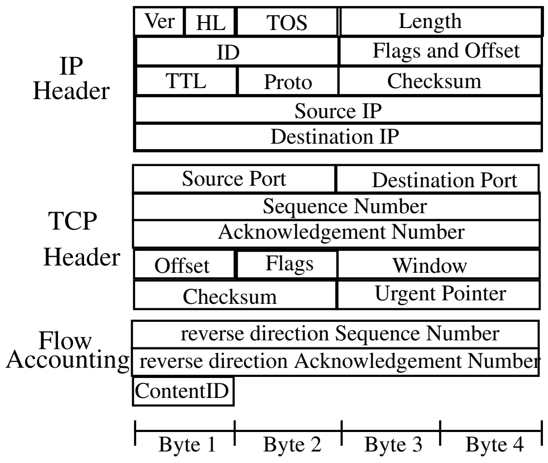


Figure 6.2: Typical minimum header requirements in bytes.

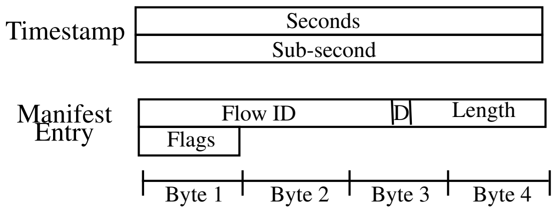
10,000,000 packets at high speeds. We choose the number 10,000,000 completely arbitrarily though any sufficiently large number would suffice. To begin with, we consider only the headers of the traffic. For simplicity we assume that all packets are TCP and that Ethernet headers are not necessary for this test. Further, we assume that at least 8 bytes are required as a time stamp for each packet in order to ensure proper timing for the given test. Figure 6.2 gives a representation of the information required for each packet. Under these assumptions, a minimum of 48 bytes are required to store each packet creating a total of 457 MB of space required to house just the headers for the 10,000,000 packets. Many platforms simply do not have sufficient memory to store that amount of data. This means that the data will need to be loaded onto the hardware from an exterior source more often. Further, this does not even consider the payload yet.

In order to alleviate this, we propose the idea of a flow template. Since most of the fields in the IP header and TCP header are static for a flow, these values can be stored once, and simply loaded when a particular packet is sent. Once again, Figure 6.2 illustrates all the fields that are largely static for any test with green vertical bars. Those fields represent most of the fields of the header and are values that will not change, or, as is the case with IP addresses and port numbers, only swap positions. This implies that the total amount of data required to handle the headers for a test can be greatly reduced. The fields marked in red represent the fields that must be represented for each packet. This includes the timing information, length of the packet, and the flags. Those fields marked in brown horizontal lines mark fields that must be accounted for, but do not necessarily require a value for every packet. For example, checksums can be calculated efficiently in hardware and that can be done as the packet is sent. Acknowledgement and Sequence numbers are a little more tricky, but can be maintained as counters as they are in actuality. Thus, creating a set of header templates based on each unique flow could greatly reduce the entire space required for the headers. In other words, where initially one header must be maintained for each packet when using templates only one header is needed for each distinct flow. Since most packets seen on the wire tend to come from large flows [17] then we can typically expect a far smaller number of flows than total packets. Thus, if there exist only 100,000 flows in our above example then all of the headers can be housed in roughly 4.7 MB of RAM.

However, naïvely applying the flow templates will cause a complete loss of packet scheduling information. While it would be possible to create a stochastic process to choose when to send packets, it is often desirable at regeneration time to have predictable packet events. Note, we remarked earlier that the static nature of packet captures is a weakness yet here we are touting it as an advantage. Let us clarify that all regeneration illustrated here assumes that one of the processes used in Chapter 4 or Chapter 5 was used to create a packet capture using a tool like PaCSim as described in Section 6.1. Thus, the packet captures used herein do not suffer from the static nature of normal packet captures as we can create any number of packet captures to meet a variety of scenarios. Regardless, in order to maintain timing information it is necessary to create a manifest of which packets are to be sent, how many bytes, and when. Further, it is necessary to have an adequate flow template model to allow for sending.



(a) Typical minimum header requirements in bytes.



(b) Manifest entry requirements in bytes.

Figure 6.3: Header and manifest structure requirements.

The flow template model is illustrated as Figure 6.3(a). As can be seen, the entire IP and TCP headers are maintained in the template. This is done for ease of reference and copying during a test. Thus, when a packet is up, the template need only be copied into memory and then the necessary changes applied. Further, we note the removal of timing information as that will be maintained with a manifest entry. Also, note the addition of 9 bytes. These extra bytes are to serve as counters for Sequence and Acknowledgement numbers in order to maintain the correct values. Essentially, the default Acknowledgement and Sequence numbers can be maintained as counters for the initial direction and the new values can be used in the reverse direction. Finally, a single byte field is maintained to designate content for packets and will be discussed later. In total, the template header is 49 bytes long.

Given the template, it is then necessary to create a manifest for each packet. Each manifest entry must maintain, at a minimum, a time stamp. This time stamp is comprised of a 32 bit integer designating the second resolution and another 32 bit integer designating the sub-second resolution, for a total of 8 bytes. Next, a field must exist that will designate the flow to be used, the length of the payload for the packet, and the direction of the packet (client to server or server to client). In order to make the most compact list possible, we make an assumption that might be relaxed at later times. That assumption is that for evaluation purposes, we wish to restrict our selves to no more than 1 million distinct flows. The reason for this assumption is so that we can squeeze all three fields (flow ID, Direction, and Length) into a single 4 byte integer. Thus, we can allocate 20 bits to the Flow ID, which will allow for 1,048,576 total unique flows. We designate 1 bit for direction, and the remaining 11 bits for length allowing for packets with length up to 2048 bytes which is well beyond the standard 1460 bytes of payload to a standard Maximum-sized Transmission Unit. Finally, we note that with only 1 million flows, it is likely that flows will be used multiple times. Thus, we must be able to embed the TCP flags in the manifest entry in order to distinguish between flow starts, flow ends, and normal data exchange. Thus, for each manifest entry 13 bytes are required rather than 48 bytes for each packet in a normal packet capture.

Figure 6.4 illustrates the progression of total savings given a certain number of total flows and a size of 13 bytes for each manifest entry and 49 bytes for each template and a total of 10,000,000 packets. The total reduction in data size ranges from $^3/_4$ reduction for just a couple thousand flows to $^2/_3$ reduction in total size for 1 million flows. Thus, the total space for the test can be reduced from 467 MB of RAM to roughly 170 MB RAM, assuming 1 million distinct flows. This reduction will also maintain all timing information. However, 170 MB RAM is still too large to fit all on some hardware. However, we note that with a maximum of 1 million flows that all the flow templates can reside in roughly 46 MB of RAM, a small enough space to fit fully on most hardware. Thus, loading data onto the device now becomes just an issue of loading manifest entries and potentially payload.

Figure 6.4 also illustrates that as the number of total flows increases, the effectiveness of this scheme falls. In fact, given a packet capture of 10,000,000 packets, only a 50% reduction in

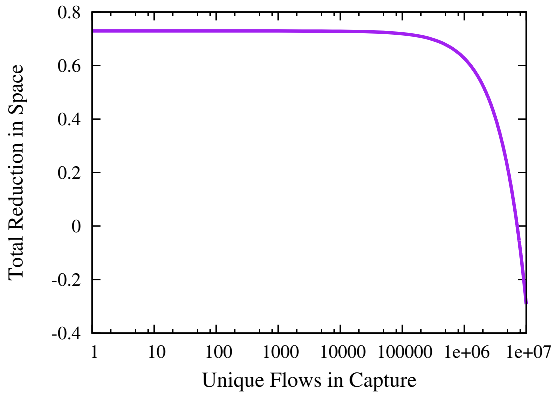
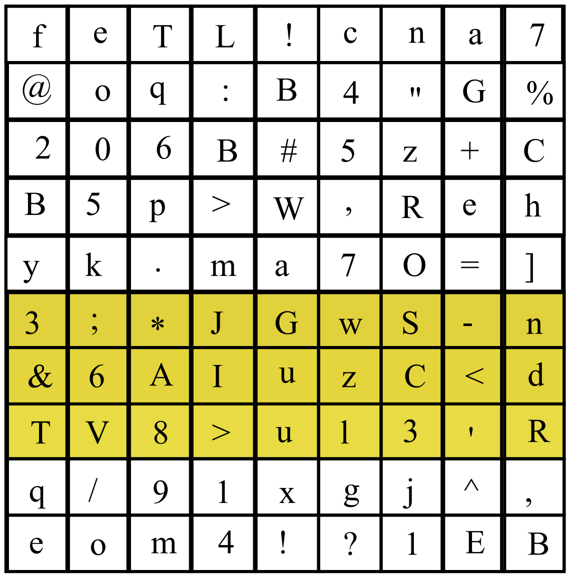


Figure 6.4: Reduction in storage requirements for headers.

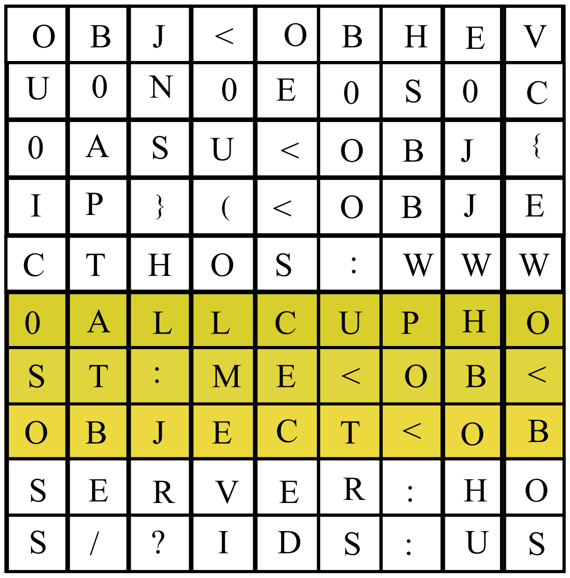
size is gained if there are 2 million flows and this method will actually require more space if more than 7 million flows are needed (hence the negative reduction rate on the chart). This is one motive for setting a maximum of 1 million flows. However, we note that the effectiveness of this method will depend largely on the ratio of flows to total packets. While that ratio is small, less than 10%, then maximum reduction can be maintained. However, when the ratio is roughly 70% (i.e. 7 flows for every 10 packets) then it is about break-even in terms of space. Thus, in order for this method to work, an upper limit must be placed on the total number of flows, and that upper limit must consider the total packets involved.

Thus far, this method can see a best case reduction of nearly 3/4 in the size of headers for a given packet capture. However, there is still the payload to consider. Given the example of 10,000,000 packets, the amount of data to house the payload for those packets could be substantial. Assuming that only half of these packets carry payload (with the remainder being zero byte ACK packets) and adopting an average packet size of 250 bytes, the total size required by the example packet capture is roughly 1.6 GB, with 1.2 GB of that in payload. For NIDS benchmarking, as illustrated in Chapter 4, there are really only two types of payload: base payloads or stimulating payloads. Base payloads consist of randomly generated characters. Stimulating payloads consist



f	e	T	L	!	c	n	a	7
@	o	q	:	B	4	"	G	%
2	0	6	B	#	5	z	+	C
B	5	p	>	W	,	R	e	h
y	k	.	m	a	7	O	=	]
3	;	*	J	G	w	S	-	n
&	6	A	I	u	z	C	<	d
T	V	8	>	u	1	3	'	R
q	/	9	1	x	g	j	^	,
e	o	m	4	!	?	1	E	B

(a) Pseudo Random Content Block for Random Characters.



O	B	J	<	O	B	H	E	V
U	0	N	0	E	0	S	0	C
0	A	S	U	<	O	B	J	{
I	P	}	(	<	O	B	J	E
C	T	H	O	S	:	W	W	W
0	A	L	L	C	U	P	H	O
S	T	:	M	E	<	O	B	<
O	B	J	E	C	T	<	O	B
S	E	R	V	E	R	:	H	O
S	/	?	I	D	S	:	U	S

(b) Pseudo Random Content Block for benchmarking content.

Figure 6.5: Pseudo Random Content Blocks.

of random walks through the NIDS matching engine finite automata. Both sets are only loosely tied to the actual flows. As such, we propose the idea of pseudo random content blocks.

A pseudo random content block is a contiguous block of characters such that all characters within the block was generated for either base payloads (i.e. random characters) or stimulating content as per Chapter 4. Figure 6.5 illustrates Pseudo Random Content Tables for random characters (Figure 6.5(a)) and for benchmarking data (Figure 6.5(b)). The blocks are generated prior to the test and stored in memory. The size of the block can be anything, but should be at least the size of one MTU and probably the size of several MTU. We adopt the size of 16 KB as offering good variety, up to 10 full MTU in this case, and yet greatly reducing space requirements. When content is needed, the pseudo random content blocks are employed in a manner similar to pseudo random number generators. Essentially, a number is chosen that falls within the total number of characters in the block. That number marks the offset to the payload for a packet. The length of the packet will mark the total number of characters used after that offset. Thus, grabbing the payload for a packet is a simple lookup to the pseudo random content block. The highlighted area in Figure 6.5 represent a potential payload block of 27 characters randomly chosen with an initial offset of 45. Of course, some care must be maintained to not run off the end of a block. Regardless, to accommodate the Pseudo Random Content Blocks 1 byte is designated in the flow headers to allow for up to 256 separate pseudo random content blocks. Note, that with each pseudo random content block at 16 KB, then 256 such blocks accounts for only 4 MB. Thus, payload that originally accounted for 1.2 GB of data now occupies 4 MB. In all, the example packet capture can be reduced in size from a grand total of 1.6 GB to roughly 174 MB, nearly a factor 10 reduction.

Unfortunately, as mentioned earlier, 174 MB is still too large to fit on some hardware. As such, there must exist an efficient means of moving the data to and from the larger storage. In the NetFPGA case this means that the data must be passed from the host machine to the NetFPGA board. First, we note that with a test containing 1 million flows that the total headers would require 46 MB of RAM. Further, the pseudo random content blocks would account for a maximum of 4 MB of RAM. If all of this was loaded onto the NetFPGA it would require roughly 50 MB of RAM, which the 64 MB of DDR2 DRAM can easily manage. However, the question then becomes how to manage the manifest. Every Megabyte of RAM can account for roughly 80,600 manifest entries. The total number of packets sent each second will depend greatly on the size of the packets. Buffers of 5 MB each could contain 403,000 Manifest entries and given the constraints for this example each buffer would require roughly half a second to be emptied by a Gigabit link. Two such buffers would only add another 10 MB to the total implementation and require a total of 60 MB, leaving 4 MB for system and the generator. Only the exact parameters of the test would tell, but in the case of the NetFPGA so long as the bus speed of the PCI board is sufficient to load 5 MB into a buffer in under half a second then this model can work. Since the bus speed is slightly faster than the

Gigabit transmission speed and since 5 MB is much smaller than the 62 MB of data pushed out the NIC, we feel that so long as the data is loaded into buffers in a parallel process then there should be no problem filling one buffer prior to the emptying of the previous. So long as the buffer load time remains less than the time required by the traffic generator to exhaust a buffer then the traffic generator should be able to operate at maximum efficiency and maintain line speeds.

There are several ways that this method could be further improved. However, these methods would not work in all cases. First, the manifest list could be compressed by adding a single byte to each entry indicating the number of packets immediately following that share the same Flow ID. This could, in some circumstances, greatly reduce the total number of entries required for the manifest. However, precision timing would be lost at each compression requiring some mechanism to account for timing. Also, if the manifest consists of heavily intermixed traffic then the benefit from this approach might prove minimal. Another method that might be used to reduce or reuse Flow IDs would be to allow for the randomization of IP Addresses. Thus, every time a particular flow is instantiated a set of IP Addresses is selected for the end points similar to methods employed in PaCSim. This would require saving state for each instantiation of a flow, and would further require additional information to properly handle each new flow. However, it would have the potential for allowing unlimited number of flows in the same reduced space.

There exist a few drawbacks to this approach. First, the pseudo random content blocks work well with random characters or content developed using the methods from Chapter 4. However, they do not work so well when attempting to generate realistic content as per Chapter 5. The problem is that realistic content would be cut and/or truncated by this approach and produce content that would look garbled. The only way to counter this would be to store complete files, or pieces of files in specifically sized content blocks and always set packet lengths to match the exact size of the block. In other words, it would employ static pages. However, the fact that the current method only accounts for up to 256 content blocks means that no evaluation using these means will ever provide truly realistic data. However, the goal of high-speed transmission is typically one of benchmarking. As such, this approach aligns well with such benchmarking and should prove adequate to that task.

A second drawback is the fact that it allows only 1 million flows. While that may sound like a large number, for a test requiring 200 to 500 thousand concurrent flows at all times it becomes a bit more challenging. Once again, realism begins to fade as the same flows must be used again and again throughout the evaluation, assuming the evaluation extends beyond a few minutes. This could be remedied as mentioned earlier. However, for benchmarking purposes, it is not necessarily a bad thing. After all, 1 million flows is more than sufficient to handle most benchmarking cases. The final drawback is that in the proposed methodology only 256 separate content types are supported. With at least one of these targeting random characters only 255 can target specific regions of rules within the finite automata of the target NIDS. Unfortunately, the current Snort Rule Set has roughly 400 distinct regions of rules dependent on the flow-level quintuple. Thus, it would not be possible to test every region in a single test. This is perhaps most easily solved by creating two tests. Optionally, more space could be added to house more content regions, after all, another 150 pseudo random content blocks would require roughly 2.3 MB extra space.

Finally, we consider the implications of these modifications towards the Appropriateness, Variability, and Privacy for tests. First, the simplifications provided here will likely reduce the Appropriateness of tests, even more so than the initial model as outlined in Chapter 4. The primary factor is the the reduction in the number of total possible flows which can reduce the potential realism of an evaluation. However, the simplifications here are primarily considered for a benchmark environment, and full realism can be reduced in order to better explore behavior of a NIDS across multiple workload signatures. As such, Appropriateness is limited to the testing of NIDS, and would not extend well to other applications. Also, the degree of Privacy maintained will depend on the actual packet captures used at the flow level, though all sensitive content will be removed eliminating any potential Privacy issues concerning content. Finally, this model does not allow for Variability alone, but is really a method for improving regeneration of packet captures. As such, it would require multiple packet captures generated under the models illustrated in this work in order to maintain variability. While these simplifications will weaken how well an evaluation stands up to Appropriateness, Variability, and Privacy, they do not invalidate the method and offer a small trade-off for a greatly reduced memory signature in implementation.

CHAPTER seven
CONCLUSION

7.1 Future Work

The primary focus of future work is to create a working, high-speed implementation as described in Chapter 6.3. This would entail not only building the generator, but also noting the actual performance and effects of the simplifications employed. Further, it would entail closer integration with PaCSim. Another step is to add more features to PaCSim to facilitate the building of tests and customizing traffic features so as to test not only DPI based features from the payloads, but to also address many Anomaly Detection features as well.

ContextNet and the b-model surfer have several areas that need improvement. First, we would like to explore the use of Deep Belief Networks for clustering of documents for ContextNet rather than the k-center algorithm currently employed. The primary benefit to that approach would be a dynamic number of potential clusters rather than the set number imposed by the k-center algorithm. Further, ContextNet's usage of the hypernym tree could be refined to try and find an optimal hypernym rather than simply settling for the last term on the most popular paths. The problem with the current approach is that it can result in overly broad categories while a more intelligent solution might serve to find a more specific term. However, such winnowing is likely to require an alternate method of judging the value of hypernyms. Finally, the b-model surfer needs to implement more user behaviors such as following links on pages and recognizing poor search terms.

The final avenue of future work is to develop a modular approach toward dynamically building content. The current methodology outlined in Chapter 5 builds traffic using available, real, content. This approach would attempt to reduce real content into components and derive a generative grammar from those objects. Upon regeneration rather than simply pulling down documents from some repositories (like the Internet), completely new documents would be generated using the generative grammar.

7.2 Discussion

The methods outlined here provide a means to generate content that can be used to evaluate the DPI engines of NIDS as well as provide background traffic for general network application evaluations. These methods address deficiencies in current traffic generation methodologies. However, there is still room for improvement. First, benchmarking content as illustrated in Chapter 4 can serve well to determine how well a NIDS can handle varying levels of content. However, there are a few issues with this approach in implementation. First, when implementing a random walk through the finite automata of a DPI engine it is necessary to maintain a large amount of state. First, it is necessary to know how deep within that automata a particular traversal has gone. Second, it is necessary to know if any given state is a final state or not. These two issues, if not handled correctly, will cause a large number of alerts to be raised in the NIDS. Optimally, no alerts should be raised if the *Depth* value is not set to 1. However, since the results of one walk can combine with the results of another walk, it is possible to inadvertently cause alerts where they were not intended. This will depend on the rule set, but represents one area that is a problem.

More generally, creating background traffic to match a target network is problematic. The approach here will retain some of the features of the target environment, but will certainly not retain all, as is discussed later. However, the larger problem is gathering the correct corpora for use. A large testbed evaluation requiring a host of different applications like email, video, audio, in addition to other types of documents, will still require a copora. The Internet is sufficient for many purposes, but not necessarily sufficient for mimicking a particular organizational Intranet. Thus, in order for truly accurate results an organization would need to build their own tools for mapping content on the Intranet as well as determining a means for culling a sample of “vetted” documents that could potentially contain too much personal information (i.e. emails, business memos, etc.). This motivates some of the need for future work concerning the generation of content from a generative grammar. While the techniques herein would provide a means to overcome these issues, albeit at a cost in evaluator labor, there is still a large body for potential future work to address this issue.

Appropriateness, Variability, and Privacy as outlined in this paper offer a means to quantify generated traffic. However, there still remain some issues. First, Appropriateness is still very test specific. For example, we claim that the benchmarking traffic generated in Chapter 4 is appropriate, at least to determining the limits of a NIDS. However, this traffic is not appropriate as far as matching a target environment. Further, we offer only one measure of appropriateness in this paper, that of clustering behaviors in content, consumers, and producers. There are likely many other areas that could be investigated. Despite these shortcomings, the idea of Appropriateness offers a valuable tool for capturing qualities in a target environment and retaining them in generated traffic. As such, this is a potential future avenue of research.

Variability is also an important factor of any approach. Static tests run with a static set of static files will more than likely generate traffic that misses many potential scenarios. This will lead directly to incorrect or misleading results and conclusions. The greatest difficulty with variability is maintaining the ability to repeat experiments. This requires a model that can perform roughly the same for each test run. The techniques within this work not only address this issue, but also utilize the idea of creating multiple packet captures. Thus, variability can be employed to meet a range of conditions but the outputs can be stored as static files that can be used again and again with expected outcomes. In a way, this ties together the best of both worlds. Ultimately, variability is the essence of the work provided here as it is the simulation model that provides for the variability. In the future, we hope to potentially expand the model to more variables to allow more precise control.

Privacy issues are largely ameliorated by the approaches outlined herein, with one exception. In the case where sensitive content is encountered regularly then that content, or similar content, will likely find its way into an evaluation as per Chapter 5. This could cause privacy issues for users within a system as investigations ensue. However, we note that most organizations that would use these approaches to test their systems have policies against taboo behaviors. Further, those that would employ these approaches to test their systems would likely be the same employees who would enforce network usage policies. In that instance, expectations of privacy by users are suspended and any reprimand warranted. However, in the case of research where the users of the data are not related to any means of authority and yet the discovery of these sensitive subjects leads to uncovering sensitive information concerning other employees, then discretion is warranted. The only real defense in this circumstance is to maintain tighter controls on the actual data such that only authorized personnel can view the original data.

Finally, we note that the b-model surfer can prove a great boon to automating content harvesting. However, it can be subtly influenced by the search engine employed. Further, search engines, like Google, do not necessarily appreciate the usage of their tools in this manner. In fact Google requires that if a user wishes to automatically grab more than 100 pages a day that they must subscribe to a particular service. Of course, there are ways around this but even so there are legal issues with utilizing a public tool such as Google.

7.3 Conclusion

In this work we have attempted to demonstrate the failure of current traffic generation techniques to account for application-level content. This failure is understandable and a direct result of the heterogeneity of network applications. To overcome this deficiency we created the benchmarking content generation approach to create content that specifically targets Network Intrusion Detection Systems (NIDS). Content generated under this model can evaluate myriad conditions that might impact a NIDS and demonstrates the impact both payload and the rule set can have on the NIDS. This model serves not only to evaluate the ability of a NIDS to process payloads during Deep Packet Inspection, but also identify potential weak areas in a rule set where rules are possibly ineffective or too costly in terms of processing.

Further, we broaden our content generation with the generative pyramid in order to provide for the population of content for any traffic generation scenario. This content can then be used for any scenario to match a variety of conditions. Content harvested under the b-model surfer follows the clustering of content whether that clustering is arbitrarily created or directly taken from some network. Consumer and Producer clustering can also be maintained in order to tie real content to any traffic generator. While there are still some issues involved with this model, it offers a clear methodology for the harvesting of content where no such methodology had existed previously.

Finally, we explored some of the implementation considerations concerning these models. The Packet Capture Simulator (PaCSim) is a direct product of this research and is slated to be offered to the public in May. As such, researchers will have another tool which may be used to evaluate NIDS. In fact, there has already been interest in this tool by other researchers. Further, the reduction methods examined here can serve to make hardware platforms like the NetFPGA viable platforms for high-speed evaluation. This can lower the cost factor involved in high-speed traffic generation and make it more palatable to smaller companies. Ultimately, we believe that the research here helps forward the ability to evaluate NIDS specifically, and network applications in general, and hope that it will serve research and industry other than our own.

BIBLIOGRAPHY

[1] E. Brockmeyer, H.L. Halstrom, and Arne Jensen. The Life and Works of A. K. Erlang. *Transactions of the Danish Academy of Technical Sciences*, 2:1–278, 1948. Available at: `http://oldwww.com.dtu.dk/teletraffic/Erlang.html`.

[2] Kashi Venkatesh Vishwanath and Amin Vahdat. Evaluating distributed systems: Does background traffic matter? In *Proceedings of USENIX Annual Technical Conference*, June 2008.

[3] Aaron Turner. Tcpreplay. Available at: `http://tcpreplay.synfin.net/`.

[4] Victor Valgenti and Min Sik Kim. Simulating content in traffic for benchmarking intrusion detection systems. In *Proceedings of the* 4^{th} *International ICST Conference on Simulation Tools and Techniques*, 2011.

[5] John 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*, 3(4):262–294, November 2000.

[6] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation. In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, volume 2, pages 12–26, January 2000.

[7] Massachusetts Institute of Technology Lincoln Laboratories. *DARPA Data Sets for Testing IDS*, 1998.

[8] Leland, Will E. and Taqqu, Murad S. and Willinger, Walter and Wilson, Daniel V. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Transactions on Networking*, 2:1–15, 1994.

[9] Vern Paxson and Sally Floyd. Wide area traffic: the failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3:226–244, 1995.

[10] M.E. Crovella and A. Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5:835–546, 1997.

[11] Anukool Lakhina, Mark Crovella, and C Diot. Diagnosing network-wide traffic anomalies. In *Proceedings of SIGCOMM 2004*, August 2004.

[12] Tarem Ahmed, Mark Coates, and Anukool Lakhina. Multivariate online anomaly detection using kernel recursive least squares. In *Proceedings of INFOCOM 2007*, May 2007.

[13] Jeffrey Erman, Anirban Mahanti, Martin Arlitt, and Carey Williamson. Identifying and discriminating between web and peer-to-peer traffic in the network core. In *Proceedings of the 16th international conference on World Wide Web*, pages 883–892, 2007.

[14] The Snort Project. *Snort User Manual 2.9.1*, September 2011. `http://www.snort.org/assets/166/snort_manual.pdf`.

[15] Jon Dugan and Mitch Kutzko. iperf. Source Forge, 2008.

[16] Benjamin Gaidioz, Rich Wolski, and Bernard Tourancheau. Synchronizing network probes to avoid measurement intrusiveness with the network weather service. In *Proceedings of the* 9^{th} *International Symposium on High-Performance Distributed Computing*, 2000.

[17] Liang Guo and Ibrahim Matta. The war between mice and elephants. In *Proceedings of the 9th International Conference on Network Protocols*, November 2001.

[18] Joel Sommers, Vinod Yegneswaran, and Paul Barford. Recent advances in network intrusion detection system tuning. In *Proceedings of the* 40^{th} *Annual Conference on Information Sciences and Systems*, March 2006.

[19] Joel Sommers and John Raffensperger. Efficient and realistic generation of IP addresses. In *Proceedings of the* 4^{th} *International ICST Conference on Simulation Tools and Techniques*, march 2011.

[20] Neil Spring, Ratul Mahajan, and David Wetherall. Measuring ISP topologies with rocketfuel. In *Proceedings of SIGCOMM*, 2002.

[21] David Mosberger and Tai Jin. httperf—a tool for measuring web server performance. Technical report, Hewlett Packard, 1998.

[22] Joel Sommers, Vinod Yegneswaran, and Paul Barford. A framework for malicious workload generation. In *Proceedings of ACM SIGCOMM Internet Measurement Conference*, 2004.

[23] Defcon. `http://www.defcon.org`. Capture the Flag Data Sets.

[24] Benjamin Sangster, T. J. O'Connor, Thomas Cook, Robert Fanelli, Erik Dean, William J. Adams, Chris Morrell, and Gregory Conti. Towards instrumenting network warfare competitions to generate labeled datasets. In *Proceedings of USENIX Security Workshop on Cyber Security Experimentation and Test*, 2009.

[25] Ruoming Pang, Mark Allman, Vern Paxson, and Jason Lee. The devil and packet trace anonymization. *SIGCOMM Computer Communication Review*, 36:29–38, 2006.

[26] Ruoming Pang and Vern Paxson. A high-level programming environment for packet trace anonymization and transformation. In *SIGCOMM*, 2003.

[27] D. Koukis, Spiros Antonatos, Demetres Antoniades, Evangelos P. Markatos, and P Trimintzios. A generic anonymization framework for network traffic. In *Proceedings of the 2006 International Conference on Communications*, June 2006.

[28] Vidar Evenrud Seeburg and Slobodan Petrovic. A new classification scheme for anonymization of real data used in IDS benchmarking. In *Proceedings of the 2nd International Conference on Availability, Reliability and Security*, April 2007.

[29] Vern Paxson and Sally Floyd. Why we don't know how to simulate the internet. In *Proceedings of the Winter Simulation Conference*, 1997.

[30] Wayne Meitzler, Steve Oudekirk, and Chad Hughes. Security Assessment Simulation Toolkit: SAST. Technical report, Pacific Northwest National Laboratory, 2009.

[31] Joel Sommers and Paul Barford. Self-configuring network traffic generation. In *Proceedings of 2004 Internet Measurement Conference*, October 2004.

[32] Van Jacobson, Craig Leres, and Steven McCanne. libpcap, 2012. Available at: `http://www.tcpdump.org/`.

[33] Riverbed Technology. Winpcap, 2012. Available at: `http://www.winpcap.org/`.

[34] Luis MartinGarcia. TCPDump, 2012. Available at: `http://www.tcpdump.org/`.

[35] Gerald Combs. *Wireshark*. Wireshark Foundation, 2012. Available at: `http://www.wireshark.org/`.

[36] BeeSync Technologies. PacketX, 2005. Available at: `http://www.beesync.com/packetx/index.html`.

[37] Marco Carnut, Tim Potter, Bo Adler, and Peter Lister. Net::pcap perl module. Comprehensive Perl Archive Network, 2012. Available at: `http://search.cpan.org/`.

[38] Sly Technologies. jNetPcap, 2012. Available at: `http://jnetpcap.com/`.

[39] George Foot and Chad Catlett. libnet. Source Forge, 2003. Available at: `http://libnet.sourceforge.net/`.

[40] Juha Laine, Sampo Saaristo, and Rui Prior. RUDE: Real-time UDP Data emiter and CRUDE Collector for the Real-time UDP Emitter. Source Forge, 2012. Available at: `http://rude.sourceforge.net/`.

[41] Robert Olsson. pktgen the linux packet generator. In *Proceedings of the Linux Symposium*, pages 11–24, jul 2005.

[42] Adam Covington, Glen Gibb, John Lockwood, and Nick McKeown. A packet generator on the NetFPGA platform. In *Proceedings of the* 17th *IEEE Symposium on Field Programmable Custom Computing Machines*, 2009.

[43] Donato Emma, Antonio Pescape', and Giorgio Ventre. Analysis and Experimentation of an Open Distributed Platform for Synthetic Traffic Generation. In *Proceedings of the* 10th *IEEE International Workshop on Future Trends of Distributed Computing Systems*, May 2004.

[44] Leo Liang. IPGen. Source Forge, 2001.

[45] Bo Cato. Packet generator. Source Forge, 2011. Available at: `http://sourceforge.net/projects/pacgen/`.

[46] Miha Jemec. packETH. SourceForge, 2003. Available at: `http://sourceforge.net/users/jemcek`.

[47] Darren Bounds. Packit: Packet analysis and injection tool, 2002. Available at: `http://packetfactory.openwall.net/projects/packit/`.

[48] Jitsu, Irib, Nono, Donnie Tognazzini, pkg forger, and Rafael. Packet excalibur, 2003. Available at: `http://freecode.com/projects/packetexcalibur`.

[49] Embyte. Gspoof. Source Forge, 2002. Available at: `http://gspoof.sourceforge.net/`.

[50] Laurent Riesterer. GASP: Generator and Analyzer System for Protocols, 2002. Available at `http://laurent.riesterer.free.fr/gasp/`.

[51] Mike Ricketts. SendIP, 2003. Available at: `http://www.earth.li/projectpurple/progs/sendip.html`.

[52] Jeff Nathan. Nemesis. Source Forge. Available at: `http://nemesis.sourceforge.net/`.

[53] Philippe Biondi. Scapy, 2012. Available at `http://www.secdev.org/projects/scapy/`.

[54] Salvatore Sanfilippo, Nicolas Jombart, Denis Ducamp, Yann Berthier, and Stephane Aubert. hping, 2005. Available at: `http://www.hping.org/`.

[55] Josiah Zayner. IP sorcery, 2004. Available at: `http://directory.fsf.org/wiki/IP_Sorcery`.

[56] Mike Frantzen and Shu Xiao. ISIC: IP Stack Integrity Checker. Source Forge, 2007. Available at: `http://isic.sourceforge.net/`.

[57] Naval Research Laboratory. *Multi-Generator (MGEN)*, 2009. Available at `http://cs.itd.nrl.navy.mil/work/mgen/`.

[58] Vinjay Ribeiro, Ryan King, and Niels Hoven. Poisson traffic generator. Rice University, 2003. Available at: `http://www.spin.rice.edu/Software/poisson_gen/`.

[59] Sebastian Zander, David Kennedy, and Grenville Armitage. KUTE—a high performance kernel-based UDP traffic engine. Technical report, Centre for Advanced Internet Architectures (CAIA), 2005.

[60] Stefano Avallone. Mtools: an udp traffic generator, 2002. Available at: `http://www.grid.unina.it/grid/mtools/`.

[61] Ghislain Mary. Packgen network packet generator, 2005. available at: `http://packgen.rubyforge.org/`.

[62] Rick Jones. NetPerf. Information Networks Division, Hewlett–Packard, 1995. Available at: `http://www.netperf.org/`.

[63] Jukka Manner. Jugi's traffic generator. University of Helsinki, 2005. Available at: `http://www.cs.helsinki.fi/u/jmanner/software/jtg/`.

[64] Robert Sandilands. Network traffic generator. Source Forge, 2011. Available at: `http://sourceforge.net/projects/traffic/`.

[65] Yumo. Tfgen, 2000. Available at: `http://www.st.rim.or.jp/~yumo/pub/tfgen.html`.

[66] Charles Krasic. mxtraf. Source Forge, 2002. Available at: `http://mxtraf.sourceforge.net/`.

[67] Incorporated NorthWest Performance Software. NetScanTools Pro, 2012. Information at: `http://www.netscantools.com/nstpro_packet_generator.html`.

[68] Joel Sommers, Vinod Yegneswaran, and Paul Barford. Toward comprehensive traffic generation for online IDS evaluation. Technical report, University of Wisconsin-Madison, 2005.

[69] Roel Jonkman, Joseph Evans, and Victor Frost. Netspec: A tool for network experimentation and measurement. University of Kansas, 1994.

[70] K. Kant, V. Tewari, and R. Iyer. Geist: A web traffic generation tool. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, 2001.

[71] Giovanni Giacobbi. netcat. Source Forge, 2012. Available at: `http://netcat.sourceforge.net/`.

[72] Paul Barford and Mark Crovella. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS Performance Evaluation Review*, 26:151–160, 1998.

[73] Tsung: A distributed load testing tool, 2012. Available at: `http://tsung.erlang-projects.org/`.

[74] Darren Mutz, Giovanni Vigna, and Richard Kemmerer. An experience developing an IDS stimulator for the black-box testing of network intrusion detection systems. In *Proceedings of the* 19^{th} *Annual Computer Security Applications Conference*, December 2003.

[75] Addy Yeow Chin Heng. Bit-twist. Source Forge, 2011. Available at: `http://bittwist.sourceforge.net/index.html`.

[76] P. Srivats. Ostinato. Google Code, 2011. Available at: `http://code.google.com/p/ostinato/`.

[77] Wu-chang Feng, Ashvin Goel, Abdelmajid Bezzaz, Wu-chi Feng, and Johnathan Walpole. TCPivo A High-Performance Packet Replay Engine. In *ACM SIGCOMM*, 2003.

[78] Daniel Borkmann and Emanuelle Roullit. netsniff-ng: the packet sniffing beast, 2010. Available at: `http://netsniff-ng.org/doc/netsniff-ng.html`.

[79] Kashi Venkatesh Vishwanath and Amin Vahdat. Swing: Realistic and responsive network traffic generation. *IEEE/ACM Transactions on Networking*, 17(3):712–725, June 2009.

[80] NPULSE Technologies. Hammerhead packet capture solutions, 2012. Available at: `http://npulsetech.com/`.

[81] Fluke Networks. ClearSight analyzer, 2012. Available at `http://www.flukenetworks.com/enterprise-network/network-monitoring/ClearSight-Analyzer`.

[82] Absolute Analysis. Absolute analysis investigator afdx traffic generator, 2012. Information at: `http://www.absoluteanalysis.com/products/traffic-generators/ethernet-traffic-generator.html`.

[83] Omnicor. IP Packet Generator, 2012. Information at: `http://www.omnicor.com/network_testing_tools.aspx`.

[84] Omnicor. IP Impairment Simulators, 2012. Information at: `http://www.omnicor.com/network_testing_tools.aspx`.

[85] Omnicor. GPS Synchronization/Timing, 2012. Information at: `http://www.omnicor.com/network_testing_tools.aspx`.

[86] Lee M. Rossey, Robert K. Cunningham, David J. Fried, Jesse C. Rabek, Richard P. Lippmann, Joshua W. Haines, and Marc A. Zissman. LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed. In *Proceedings of the 2002 IEEE Aerospace Conference*, March 2002.

[87] Michael Liljenstam, Jason Liu, David M. Nicol, Yougu Yuan, Guanhua Yan, and Chris Grier. RINSE: The Real-time Immersive Network Simulation Environment for Network Security Exercises (extended version). *Simulation*, 82(1):43–59, 2006.

[88] Kay Anderson, Joseph Bigus, Eric Bouillet, Parijat Dube, Nagui Halim, Zhen Liu, and Dimitrios Pendarakis. SWORD: Scalable and flexible WORkload generator for Distributed data processing systems. In *Proceedings of the 2006 Winter Simulation Conference*, 2006.

[89] Spirent. Spirent, 2012. Brochure available at: `http://www.spirent.com/`.

[90] Phoenix Datacom. Packetstorm Network Emulator–complete network simulation, 2012. Information at: `http://www.phoenixdatacom.com/packetstorm.html`.

[91] Phoenix Datacom. BreakingPoint Systems, 2012. Information at: `http://www.phoenixdatacom.com/breakingpoint.html`.

[92] Excentis. ByteBlower, 2012. Information at: `http://www.excentis.com/`.

[93] Candela Technologies. LANforge-Fire and LANforge-Ice, 2012. Information at: `http://www.candelatech.com/l`.

[94] Vern Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31(23–24):2435–2463, December 1999.

[95] Holger Dreger, Anja Feldman, Vern Paxson, and Robin Sommer. Predicting the resource consumption of network intrusion detection systems. In *Proceedings of the* 11^{th} *International Symposium on Recent Advances in Intrusion Detection*, September 2008.

[96] Michela Becchi, Mark Franklin, and Patric Crowley. A workload for evaluating deep packet inspection architectures. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization*, September 2008.

[97] Guangzhi Qu, S. Hariri, and M. Yousif. Multivariate statistical analysis for network attacks detection. In *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, page 9, 2005.

[98] Shuyuan Jin and D. S. Yeung. A covariance analysis model for ddos attack detection. In *Proceedings of IEEE International Conference on Communications*, 2004.

[99] Marco Barreno, Blaine Nelson, Russell Sears, Anthony D. Joseph, and J. D. Tygar. Can machine learning be secure? In *Proceedings of the 2006 ACM Symposium on InformAtion, Computer and Communications Security*, 2006.

[100] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated worm fingerprinting. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2004.

[101] Z. Li, M. Sanghi, Y. Chen, M. Kao, and B Chavez. Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2006.

[102] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of IEEE Symposium on Security and Privacy*, May 2005.

[103] Randy Smith, Cristian Estan, and Somesh Jha. Backtracking algorithmic complexity attacks against a NIDS. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, 2006.

[104] Sourcefire Vulnerability Research Team. *Sourcefire Vulnerability Research Team (VRT) Snort Rule-set*, 2.8.6 edition, September 2010. Available at `http://www.snort.org/vrt`.

[105] Ranier Bye, Stephan Schmidt, Katja Luther, and Sahin Albayrak. Application-level simulation for network security. In *Proceedings of the First International Conference on Simulation Tools and Techniques for Communications, Networks and Systems, SIMUTools 2008*, March 2008.

[106] ns 2 project. The ns-2 network simulator. `http://www.isi.edu/nsnam/ns/`.

[107] ns 3 project. The ns-3 network simulator. `http://www.nsnam.org/`.

[108] Alfred Aho and Margaret Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[109] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. BLINC: Multi-level traffic classification in the dark. In *Proceedings of ACM SIGCOMM*, 2005.

[110] Marco Canini, Wei Li, Andrew W. Moore, and Raffaele Bolla. GTVS: Boosting the Collection of Application Traffic Ground Truth. In *Proceedings of the Traffic Monitoring and Analysis Workshop*, pages 54–63, 2009.

[111] Ionut Trestian, Supranamaya Ranjan, Aleksandar Kuzmanovi, and Antonio Nucci. Unconstrained endpoint profiling (Googling the Internet). *SIGCOMM Computer Communication Review*, 38(4):279–290, 2008.

[112] Byungjoon Lee, Kisu Kim, Taeck-geun Kwon, and Youngseok Lee. Content classification of WAP traffic in Korean cellular networks. In *Proceedings of theIEEE/IFIP Network Operations and Management Symposium Workshop*, 2010.

[113] George Miller. Wordnet: A lexical database for english. *Communications of the ACM*, 38:39–41, 1995.

[114] Christiane FellBaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[115] Stefan Wermter and Chihli Hung. Selforganizing classification on the Reuters news corpus. In *Proceedings of the 19th international conference on Computational linguistics - Volume 1*, 2002.

[116] Vikramjit Mitra and Chia J. Wang. A neural network based audio content classification. In *Proceedings of the Interantional Joint Conference on Neural Networks*, 2007.

[117] Toni Makinen, Serkan Kiranyaz, and Moncef Gabbouj. Content-based audio classification using collective network of binary classifiers. In *Proceedings of the IEEE Workshop on Evolving and Adaptive Intelligent Systems*, 2011.

[118] Sutjipto Arifin and Peter Y. K. Cheung. A novel probabilistic approach to modeling the pleasure-arousal-dominance content of the video based on "working memory". In *Proceedings of the International Conference on Semantic Computing*, 2007.

[119] Toon De Pessemier, Tom Deryckere, and Luc Martens. Context aware recommendations for user-generated content on a social network site. In *Proceedings of the European Interactive Television Conference*, 2009.

[120] David Lewis. Reuters-21578 News Corpus, 1997.

[121] Chris Bennett. More efficient classification of web content using graph sampling. In *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Data Mining*, 2007.

[122] Mengzhi Wang, Tara Madhyastha, Ngai Hang Chan, Spiros Papadimitriou, and Christos Faloutsos. Data mining meets performance evaluation: Fast algorithms for modeling bursty traffic. In *Proceedings of the* 18^{th} *International Conference on Data Engineering*, 2002.

[123] Larry Page. PageRank: bringing order to the web. Technical report, Stanford Digital Library Project, 1998.